Coordenação de Agentes Móveis através do Canal de Broadcast

Vera Nagamuta

DISSERTAÇÃO APRESENTADA AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA DA
UNIVERSIDADE DE SÃO PAULO PARA
OBTENÇÃO DO GRAU DE MESTRE EM
CIÊNCIA DA COMPUTAÇÃO

Orientador: Prof. Dr. Markus Endler

— São Paulo, novembro de 1999 —

Na elaboração deste trabalho, a autora obteve apoio financeiro da CAPES e CNPq.

Coordenação de Agentes Móveis através do Canal de Broadcast

Este exemplar corresponde à redação final da dissertação, devidamente corrigida, defendida por Vera Nagamuta e aprovada pela comissão julgadora.

São Paulo, 19 de novembro de 1999.

Banca examinadora:

- Prof. Dr. Markus Endler (orientador) — MAC-IME-USP
- Prof. Dr. Flávio Soares Correa da Silva — MAC-IME-USP
- Prof. Dr. Antônio Alfredo Ferreira Loureiro — DCC -UFMG

Aos meus pais,
à Sônia e Paulinho,

Agradecimentos

Aos meus pais, pelo apoio, incentivo e por tudo que proporcionaram em minha vida dando-me condições de batalhar para alcançar a realização deste Mestrado.

À minha irmã Sônia, pela força e incentivo, mesmo estando muito distante.

Ao meu irmão Paulo, pela força, apoio e momentos de descontração.

Ao meu orientador, Professor Markus Endler, pela sua ótima orientação, por toda sua dedicação e paciência durante a elaboração desta dissertação, pelos seus ensinamentos, pela sua cooperação e prontidão em me auxiliar em todos os momentos.

À Professora Nami Kobayashi, pela sua confiança, amizade e por sua orientação desde o início deste Mestrado.

Aos professores Takada Sensei, Yamada Sensei, Gotoh Sensei e Ishimitsu Sensei, da Universidade de Aizu, pela força e incentivo.

Ao Professor José Augusto, por sua atenção e gentileza em esclarecer dúvidas e enviar informações sobre o Mestrado, quando ainda estava no Japão.

Ao Professor Flávio, por estar presente na banca de qualificação e defesa, pelas suas sugestões e críticas.

Ao Professor Antônio Loureiro, pela sua presença em minha defesa, pela correção minunciosa da dissertação, pelas sugestões e críticas.

À Professora Dilma, pela sua presença na banca de qualificação.

Aos professores deste instituto, pelos preciosos ensinamentos que muito auxiliaram para o meu amadurecimento durante este Mestrado.

À amiga Maria do Carmo, pelo carinho e companheirismo nos bons momentos e nos momentos difíceis, pelos conselhos, pela força e pela grande pessoa que é.

À amiga Liliane e à sua irmã, Cristiane, pelos bons momentos que passamos.

Aos meus estimados amigos Uirá, Ricardo, Said, Sirley, Ana Paula, Marquinhos, Franklin, Lorena, Leandro, Emmanuel, Eduardo, Clara, Lucy Mari, Teo, Marcel, Luiz Carlos, Heraldo, Euler, Liliane, Nélson, Hernán, Antônio, Carlos, Lucy, pela ajuda que me deram, pelo companheirismo, pelas boas risadas e os bons momentos que fizeram com que este Mestrado se tornasse mais alegre e estimulante.

Ao Percy, pelo incentivo e apoio nos momentos difíceis, por sua amizade e compreensão.

Aos grandes amigos Juliano, Myrthes, Isabel e Alexsandro pelo companheirismo, cooperação e ajuda desde o Curso de Verão.

Ao Santos Alberto, por tudo que fez por mim neste final de Mestrado, pela força, amizade, pelo incentivo e acima de tudo, pelo seu humilde coração e pela grande pessoa que é.

Resumo

Em aplicações distribuídas baseadas em agentes móveis, a coordenação das ações dos agentes móveis é uma tarefa complexa. A maior dificuldade é devido ao fato que agentes móveis podem mudar de endereço dinamicamente.

Nesta dissertação, apresentamos o projeto e a implementação de um mecanismo de coordenação de agentes móveis que contorna este problema. Este mecanismo, que chamamos de Canal de Broadcast, está baseado na difusão de mensagens e possibilita que os membros de um grupo de agentes móveis interajam entre si, independentemente de suas localizações correntes.

Modelos de coordenação existentes oferecem formas de interação entre agentes móveis, mas todos eles impõem alguma exigência. Ou os agentes móveis devem conhecer a localização de outros agentes, ou devem estar localizados no mesmo lugar ou devem migrar para um lugar específico.

A principal vantagem deste mecanismo de coordenação está na total transparência de localização: as mensagens podem ser endereçadas a um conjunto de agentes móveis independentemente de sua localização corrente.

Este mecanismo foi implementado no ASDK (Aglets Software Development Kit) da IBM [2] e a sua utilidade foi demonstrada usando dois problemas típicos de coordenação em Sistemas Distribuídos: a Exclusão Mútua e o protocolo Manager-Workers. Testamos o desempenho do mecanismo e identificamos que o custo do Canal de Broadcast não é tão alto comparado aos benefícios que proporciona.

Através deste mecanismo, os agentes móveis poderão executar as suas tarefas e interagir entre sí com o propósito da coordenação sem as exigências impostas por outros modelos de coordenação.

Abstract

In distributed applications based on mobile agents, the coordination of mobile agents actions is a difficult task. The main difficulty comes from the fact that mobile agents can change their address dynamically.

In this dissertation, we present the project and implementation of a mechanism for coordinating mobile agents that overcomes this problem. This mechanism, which we called Broadcast Channel, is based on the difusion of messages and enables the members of a mobile agent group to interact with each other independently of their current locations.

Existing coodination models offer some means of interaction between mobile agents, but all of them impose some requirements. Either the mobile agents must know about each other's location, or must be located at the same place or must move to a specific place.

The main advantage of our coordination mechanism is the full location transparency: the messages can be sent to a set of mobile agents independently of their current locations.

This mechanism was implemented using IBM's ASDK (Aglets Software Development Kit) [2] and its use was demonstrated using two typical coordination problems in Distributed Systems: the Mutual Exclusion and the Manager-Workers protocol. We tested the performance of the mechanism and identified that the cost of the Broadcast Channel is not too high, compared to the benefit it provides.

Through this mechanism, the mobile agents can do their tasks and interact with each other for the purpose of coordination without the requirements imposed by other models.

Índice

Lista de Figuras **iv**

Lista de Tabelas **vi**

1 Introdução **1**

1.1 Objetivo 2

1.2 Agente e Agente Móvel 2

1.2.1 Vantagens de Agentes Móveis para aplicações distribuídas 3

1.2.2 Conceitos básicos 4

1.2.3 Aplicações baseadas em Agentes Móveis 5

1.3 Organização da dissertação 6

2 O Problema da Coordenação **7**

2.1 Modelos de Coordenação para Agentes Móveis 8

2.1.1 Coordenação Direta 8

2.1.2 Coordenação Orientada a Encontros 9

2.1.3 Coordenação baseada em Blackboard 10

2.1.4 Coordenação tipo Linda 10

3 Coordenação através do Canal de Broadcast **12**

3.1 Modelo do sistema e hipóteses 12

3.1.1 Conceitos básicos 13

3.1.2 Domínios e Lugares de Referência 14

3.1.3 Visão geral do mecanismo 15

3.2 Funcionamento do Canal de Broadcast 15

3.2.1 Instanciação do Canal de Broadcast e registro de agentes móveis 16

3.2.2 Envio e recebimento de mensagens 17
3.2.3 Migração de agentes móveis e re-envio de mensagens 17
3.2.4 Confirmação de mensagens 18
3.2.5 Ordenação de mensagens 19
3.3 Protocolo Hand-off 19
3.4 Término do Canal de Broadcast 21
3.5 Descrição informal do protocolo 21
3.5.1 Notação adotada no pseudo-código 21
3.5.2 Proxy 22
3.5.3 Reference Place 24
3.5.4 Place 26
3.5.5 Agent 27
3.6 Mensagens de controle do Canal de Broadcast 28
3.7 Uma seqüência para a instalação (configuração) do Canal de Broadcast 29
3.8 Comparação do Canal de Broadcast com outros modelos de coordenação 30

4 Exemplos de uso Canal de Broadcast 31
4.1 Exemplo 1: Exclusão Mútua usando relógios lógicos 31
4.1.1 Descrição do algoritmo 31
4.1.2 Implementação do Exemplo 1 33
4.2 Exemplo 2: Protocolo Manager-Workers 38
4.2.1 Descrição do protocolo 39
4.2.2 Implementação do Exemplo 2 40

5 Aglets Software Development Kit 44
5.1 A escolha da ferramenta 44
5.2 Conceitos básicos 45
5.3 Aglet API 45
5.3.1 Classe Aglet 46
5.3.2 Interface AgletProxy 46
5.3.3 Interface AgletContext 46
5.4 Criação e destruição de um aglet 47
5.5 Interação entre aglets 47
5.6 Mensagens em Aglets 47
5.7 Mobilidade 49
5.8 Modelo callback 49

6 Implementação 50

6.1 Principais classes 50
6.1.1 Descrição das classes 50
6.2 Criação da infra-estrutura do mecanismo 51
6.3 Criação de agentes móveis 53
6.4 Início do Protocolo 53
6.5 Classe InfrastructureLauncher 55
6.6 Classe AgentCreator 56
6.7 Classe Proxy 57
6.8 Classe Reference Place 59
6.9 Classe Place 60
6.10 Classe Agent 61

7 Testes **64**
7.1 Testes estáticos 65
7.1.1 Descrição 65
7.1.2 Objetivo 66
7.1.3 Parâmetros dos testes 66
7.1.4 Resultados 67
7.2 Testes dinâmicos 71
7.2.1 Descrição 71
7.2.2 Objetivo 71
7.2.3 Parâmetros dos testes 72
7.2.4 Resultados 72
7.3 Conclusão sobre os resultados dos testes 74

8 Conclusão **76**

A Resultados da Execução dos Exemplos **79**
A.1 Resultado da execução dos exemplos 79
A.1.1 Resultado do exemplo 1: Exclusão mútua usando relógios lógicos 79
A.1.2 Resultado do exemplo 2: Protocolo Manager-Workers 83

Lista de Figuras

1.1 (A): O modelo de programação distribuída tradicional (RPC) e (B): o paradigma de agentes móveis 4

2.1 Modelos de coordenação para Agentes Móveis 9

3.1 Arquitetura do modelo 15
3.2 Protocolo Hand-off 20

4.1 Liberação de recurso 32
4.2 A interface do agente recurso 39
4.3 O trabalhador T1 enviando mensagem de término da tarefa 40

6.1 Criação da infra-estrutura 51
6.2 Elementos instanciados 52
6.3 Criação do Proxy 52
6.4 Interface para criação de Agentes Móveis 53
6.5 Inicialização do Protocolo 54
6.6 Classe IntrastructureLauncher e LauncherInterface 56
6.7 Classe AgentCreator e CreatorInterface 57
6.8 Classe Proxy e ProxyInterface 58
6.9 Classe ReferencePlace 60
6.10 Classe Place 61
6.11 Classe Agent 62

7.1 Mensagens no *envio direto* (A) e no *Canal de Broadcast* (B) 66
7.2 Resultados dos testes para 2 máquinas e 6 lugares 67
7.3 Resultados dos testes para 4 máquinas e 6 lugares 68
7.4 Resultados dos testes para 4 máquinas e 9 lugares 68

7.5 Comaparação: *Envio Direto* 69
7.6 Comparação: Canal de Broadcast 69
7.7 Fator de sobrecarga Canal Broad./Envio direto 70
7.8 Resultado dos testes para 6 lugares 72
7.9 Resultado dos testes para 9 lugares 73
7.10 Comparação: Tempo de permanência baixa 73
7.11 Comparação: Taxa de permanência alta 74

Lista de Tabelas

3.1 Mensagens de Controle 28

7.1 Parâmetros considerados nos testes do *Canal de Broadcast* 66
7.2 Parâmetros considerados nos testes do *envio direto* 67
7.3 Parâmetros considerados nos testes dinâmicos 72

Capítulo 1

Introdução

Agentes móveis constituem um novo paradigma de programação distribuída que oferece uma maneira alternativa interessante e inovadora para o projeto e implementação de aplicações distribuídas. Este paradigma se fundamenta na idéia de que em vez de transmitir somente dados entre computadores de uma rede, como em sistemas distribuídos tradicionais, o código executável também pode ser transmitido.

Embora qualquer aplicação baseada em agentes móveis possa ser implementada também por métodos tradicionais, segundo Harrison, Chess e Kershenbaum [9] não existe nenhuma alternativa que englobe todas as funcionalidades suportadas por um sistema de agentes móveis. Estes autores ressaltam também que agentes móveis oferecem uma estrutura aberta e generalizada para o desenvolvimento e personalização de serviços de rede, enfatizando que, embora cada uma das caracteríticas de agentes móveis por si só não seja extremamente forte, a agregação de todas as características torna o paradigma extremamente poderoso, comparado aos métodos alternativos.

Um grande número de aplicações beneficiaria com a tecnologia de agentes móveis, dentre elas, podemos citar: comércio eletrônico, gerenciamento de redes e, principalmente, a busca de informações na rede (como a Internet). Neste último caso, agentes móveis oferecem uma forma efetiva para tratar este tipo de problema, devido às suas características de mobilidade e autonomia.

Apesar dessa diversidade de aplicações possíveis e vantagens oferecidas em relação a outros paradigmas, existem ainda várias questões em aberto para que esta tecnologia de agentes móveis possa ser largamente aceita. Os principais problemas estão relacionados com questões de coordenação, segurança e padronização.

A coordenação é um problema clássico em Sistemas Distribuídos e, em particular, também é relevante em aplicações multiagentes. Nestas aplicações, podemos ter um grupo de agentes executando uma determinada tarefa onde cada membro do grupo de agentes executa uma parte da tarefa global e, eventualmente, precisa interagir com os demais membros para garantir, por exemplo, a consistência do resultado global.

A segurança é um dos problemas mais sérios relacionados a agentes móveis onde uma das principais questões é como criar mecanismos que garanta a proteção dos agentes. Além disso, é necessário impedir que agentes maliciosos transitando pela rede acessem informações privadas

ou, até mesmo, alterem o seu conteúdo (código, estado).

Um outro problema relacionado é a padronização: a diferença entre arquiteturas e implementações dos sistemas de agentes móveis existentes impedem a interoperabilidade e a expansão da tecnologia de agentes móveis. Várias organizações estão envolvidas no desenvolvimento de padrões para agentes móveis, entre elas, a OMG [21].

1.1 Objetivo

Nesta dissertação, trataremos especificamente do problema da coordenação em grupos de agentes móveis. O conceito de coordenação, em sua definição clássica, envolve os aspectos de sincronização, comunicação e semântica da aplicação. No contexto de agentes móveis, no entanto, a complexidade do problema da coordenação é ainda maior, devido à característica de mobilidade, isto é, a inexistência de endereços fixos dos agentes móveis. Outro fator se refere à característica de autonomia de processamento que, dependendo da aplicação, pode provocar a mudança do itinerário do agente, dificultando a localização dos agentes para efeito da interação entre os mesmos.

No Capítulo 2, apresentamos a classificação dos modelos de coordenação existentes (segundo Cabri, Leonardi e Zambonelli [5]) e os sistemas que implementam estes modelos. Conforme será visto naquele capítulo, os modelos de coordenação citados oferecem alguma forma de interação de agentes móveis, mas nenhum dos modelos trata especificamente do problema da localização de agentes móveis.

Este problema pode se tornar crítico em aplicações em que é necessário que os agentes que participam de uma tarefa global recebam alguma informação espontaneamente, isto é, que não era esperada pelos mesmos. Neste caso, é preciso que a informação seja levada aos agentes de alguma forma, e para isso, é necessário localizá-los.

O objetivo desta dissertação é propor um novo mecanismo de coordenação de agentes móveis (que chamamos de *Canal de Broadcast*) que aborda o problema em questão. Este mecanismo está baseado na difusão de mensagens e garante que todos os membros de um grupo de agentes possam interagir através da troca de mensagens independentemente da localização dos outros membros. Através desse mecanismo, os agentes móveis são capazes de coordenar suas atividades a fim de alcançar uma solução global do problema.

No entanto, devido a algumas características do mecanismo proposto, o mesmo está mais voltado para o uso em redes locais.

1.2 Agente e Agente Móvel

Nesta seção definimos o conceito de agentes e agentes móveis, as principais características e vantagens, e apresentamos algumas aplicações baseadas em agentes móveis.

Agente é uma entidade de software (um programa) que executa um certo conjunto de tarefas em nome do usuário que o criou.

Agentes tipicamente possuem algumas (ou todas) as características citadas a seguir. Os agentes podem ser:

- autônomos: capazes de tomar decisões sem a interferência direta do usuário;
- adaptativos: capazes de se adaptar a diferentes situações/ambientes de execução;
- móveis: capazes de migrar de uma máquina para outra em uma rede;
- inteligentes: podem armazenar e manter dados que representam conhecimento em uma área;
- colaborativos: capazes de interagir com outros elementos.

Além disso, agentes tendem a ser programas pequenos. Assim, a maioria das aplicações baseada em agentes, geralmente, é composta por vários agentes cooperantes.

O conceito de agentes tem sido objeto de pesquisa em várias áreas da Ciência da Computação, sendo as duas principais:

- Sistemas Distribuídos: nesta área a pesquisa é feita em **agentes móveis**, que são agentes (entidades de software) que têm a capacidade de migrar de uma máquina para outra em uma rede.
- Inteligência Artificial: nesta área estuda-se o conceito de **agentes inteligentes**, que são agentes (entidades de software) que têm a capacidade de representar conhecimento e tomar decisões complexas de acordo com o perfil do usuário.

Esta dissertação adota o conceito de agente com o enfoque de Sistemas Distribuídos, uma vez que trata da criação de uma infra-estrutura para a coordenação de agentes móveis.

Definimos **agentes estáticos (estacionários)** como sendo agentes que executam somente em um *host*, no mesmo endereço. Por outro lado, **agentes móveis** não estão limitados à máquina onde iniciaram a execução, podendo migrar entre os *hosts* da rede. Após a sua criação em um ambiente de execução, um agente móvel pode transportar o seu código e estado para um outro ambiente de execução e retomar a sua execução do ponto imediatamente antes da migração. O termo **código** se refere ao código da classe (no contexto de orientação a objetos) necessário para que o agente possa ser executado. O termo **estado** corresponde aos valores dos atributos do objeto agente, que caracterizam o seu estado de execução.

1.2.1 Vantagens de Agentes Móveis para aplicações distribuídas

Em contraste com os modelos de programação distribuída existentes, nos quais apenas os dados são movidos de uma máquina para outra (como por exemplo, o modelo de chamada de procedimento remoto - RPC), o paradigma de agentes móveis também possibilita a mobilidade do código executável. Isto oferece diversas vantagens: em primeiro lugar, como os dados e o código são transferidos como um todo, não há a necessidade da transferência repetitiva de dados (por exemplo, no caso de uma aplicação que necessita da transferência de dados intermediários

ou no caso de uma chamada de várias funções remotas a partir de vários servidores - Figura 1.1), o que, em alguns casos, possibilita uma significativa economia da largura de banda (*bandwidth*) na comunicação.

Em segundo lugar, agentes móveis não requerem uma conexão estavél entre os nós da rede para garantir a execução da tarefa. De fato, após a migração para um servidor remoto, um agente poderá interagir com este sem que haja a necessidade que tal servidor esteja conectado com o nó original de forma estável. Ao terminar a sua tarefa, o agente aguarda que a conexão seja restabelecida, para retornar ao nó de origem com o resultado.

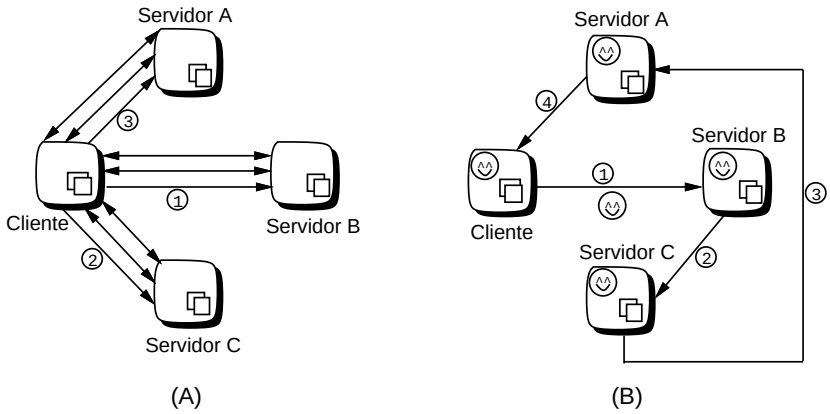

Figura 1.1: (A): O modelo de programação distribuída tradicional (RPC) e (B): o paradigma de agentes móveis

Além disso, agentes móveis executam assincronamente, uma vez que cada agente possui sua própria *thread* de execução. Também possuem a habilidade de se distribuírem entre os *hosts* da rede e, com isso manter uma configuração ótima para resolver um problema particular.

De um modo geral, agentes móveis oferecem um modelo de execução uniforme para a programação distribuída, incorporando igualmente, formas de troca de mensagens síncronas ou assíncronas, transferência de objetos e interação por intermédio de chamadas de objetos estacionários e móveis.

Apesar dessas vantagens que este paradigma oferece, existem ainda diversos problemas relacionados à coordenação, segurança e padronização, conforme mencionamos, que dificulta a aceitação e emprego deste paradigma em larga escala.

1.2.2 Conceitos básicos

Apresentamos os principais conceitos que constituem o modelo de agentes móveis comumente adotados em ambientes já existentes [12].

- *agente*: consiste essencialmente de código executável e estado de execução. Possui atributos como identificador, localização, autoridade, permissão. Pode ser estacionário ou móvel.
- *lugar* ou *contexto*: é o ambiente de execução ou localização de um agente. A cada momento, um agente pode estar em um único *lugar*. A localização física deste *lugar* (em que nó

da rede), na verdade, é irrelevante e transparente para o agente. Um nó na rede pode implementar vários *lugares*. Cada *lugar* possui um identificador único.

- *servidor* (de agentes): é um programa que deve estar executando em todos os nós da rede que devem abrigar um *lugar*. Suas principais funções são: ativar o agente que entra num *lugar*, implementar mecanismos de segurança e autenticação e disponibilizar recursos computacionais (CPU, memória, disco) para agentes executando em um (ou mais) *lugares* situados naquele nó.

- *autoridade*: é o usuário ou organização responsável pela execução de um ou mais agentes e ou *lugares*.

- *permissão*: é necessária para ter acesso a *lugares*, invocar procedimentos de outros agentes, criar novos agentes, etc. Permissões podem ser consultadas por qualquer elemento, mas podem ser alteradas apenas pela autoridade do *lugar* ou agente.

1.2.3 Aplicações baseadas em Agentes Móveis

Apresentamos algumas das aplicações possíveis de agentes móveis:

- **Coleta de dados de vários lugares de uma rede**

  Uma vez que agentes móveis possuem um itinerário de visita a vários lugares, uma aplicação seria a coleta de informações espalhadas em vários nós de uma rede. Por exemplo, um agente móvel poderia implementar uma ferramenta de *backup* que, periodicamente, deve passar por todos os discos dos computadores de uma rede, coletando dados. O agente móvel poderia migrar pelas máquinas, coletando informações sobre o estado de cada disco e, depois, voltaria ao lugar de origem para escrever esta informação em um determinado dispositivo de *backup*.

- **Busca e filtragem**

  Com o crescimento acelerado da Internet e de outras redes, a atividade de coletar dados relativos a determinado assunto ou interesse se torna cada vez mais difícil devido à grande quantidade de informação disponível. A interesse do usuário, um agente poderia, por exemplo, visitar muitos *sites* WWW, fazer a busca das informações disponíveis em cada *site* e construir um índice de *links*.

- **Negociação**

  Agentes podem obter informações também através da interação com outros agentes. Por exemplo, para marcar um encontro com várias outras pessoas, cada participante poderia enviar um agente móvel que o representasse e que possuísse informações sobre a agenda do participante. Os agentes móveis poderiam, então, negociar e estabelecer um dia e horário comum para o encontro.

- **Comércio eletrônico**

  Esta é outra aplicação que pode se beneficiar com a tecnologia de agentes móveis. Agentes móveis podem ser utilizados para tarefas de transações comerciais, pesquisa de mercado,

entre outras. A principal atividade de um agente móvel seria a de percorrer a rede de computadores, buscando fornecedores de produtos e serviços e comparando preços e funcionalidades. Antes de fazer uma viagem, por exemplo, poder-se-ia incumbir um agente de visitar bases de dados de agências de vôo, contendo preços de bilhetes de companhias aéreas para encontrar o melhor preço e data mais conveniente e proceder a reserva.

1.3 Organização da dissertação

Esta dissertação está organizada da seguinte forma: no Capítulo 2, discutimos o problema da coordenação e apresentamos a classificação de modelos de coordenação de agentes móveis (segundo Cabri, Leonardi e Zambonelli [5]) e a descrição sobre cada um desses modelos. No Capítulo 3, apresentamos o mecanismo de coordenação proposto, o *Canal de Broadcast*, descrevendo sobre o modelo do sistema, os conceitos básicos, o funcionamento básico e a especificação informal do protocolo usado para implementar o mecanismo. Também fazemos uma breve comparação com os modelos de coordenação apresentados no Capítulo 2. No Capítulo 4, apresentamos dois exemplos concretos de utilização do nosso mecanismo, descrevendo os passos seguidos para a implementação desses exemplos. No Capítulo 5, apresentamos o ambiente de programação de agentes móveis utilizado na implementação do *Canal de Broadcast*, o ASDK (Aglets Software Development Kit). No Capítulo 6, descrevemos a respeito da implementação deste mecanismo, apresentando os passos para a criação da infra-estrutura do *Canal de Broadcast* e uma descrição sobre as principais classes. Apresentamos, no Capítulo 7, os testes realizados com o *Canal de Broadcast* para dois casos: com e sem mobilidade dos agentes móveis e discutimos sobre os resultados desses testes. Finalmente, no Capítulo 8, apresentamos as conclusões a respeito deste trabalho.

Capítulo 2

O Problema da Coordenação

A **coordenação** é definida como o gerenciamento das dependências das atividades [19]. É uma teoria interdisciplinar que envolve pesquisas nas áreas de Ciência da Computação, Administração, Economia, Ciência Cognitiva, entre outras.

Em sistemas distribuídos e concorrentes, o problema da coordenação é um problema bastante estudado. Um exemplo comum é o compartilhamento de recursos (processadores, memória, dispositivos de entrada e saída) que precisam ser gerenciados através de mecanismos como semáforos, monitores, árbitros e protocolos de consenso ou exclusão mútua, etc. Em banco de dados foram desenvolvidos vários mecanismos como *locking* e transações atômicas para permitir que mútiplos processos possam acessar concorrentemente dados compartilhados sem que haja interferências indesejadas.

Agentes móveis são um caso particular de programação distribuída e, portanto, além de herdar todas as dificuldades encontradas em Sistemas Distribuídos assíncronos (tais como ausência de um estado global, ausência de relógios sincronizados, mensagens com tempos de transmissão desconhecidos, etc.), a sua característica adicional de mobilidade implica em novos problemas para a coordenação.

Segundo Lesser [17], em sistemas multiagentes com um número de agentes executando cooperativamente uma tarefa, a coordenação das atividades dos agentes se torna necessária quando existem interdependências entre essas atividades. Por exemplo, o compartilhamento de recursos para a execução das atividades dos agentes ou quando estas atividades são subtarefas que contribuem para a solução de um problema maior. Neste útimo caso, a coordenação envolve decisões como: se as subtarefas serão executadas seqüencialmente (e em que ordem) ou concorrentemente pelos agentes, se os resultados intermediários devem ou não ser transmitidos e, em caso afirmativo, quando e para quem deverão ser transmitidos, etc. Em muitos casos, a coordenação envolve o complexo problema de otimizar a ordenação de tarefas e a sua distribuição aos agentes.

Na área de Inteligência Artificial, foram desenvolvidos sistemas que se concentram em solucionar problemas de coordenação em sistemas multiagentes (não móveis) através de planejamentos e estratégias globais com relação às atividades dos agentes. Estas estratégias globais tratam de atribuir as ações aos agentes e garantir que outros agentes tomem decisões complexas tendo como base o resultado de ações de outros agentes do sistema.

Um exemplo, é o *JAFMAS* [8] que é um *framework* baseado em Java para a construção de sis-

temas multiagentes que oferece ferramentas para o desenvolvimento de estratégia de coordenação e a sua utilização em problemas a serem solucionados por agentes. Para cada agente responsável por executar uma tarefa são criados planos condicionais baseados nas possíveis ações e reações de outros agentes e que são extensíveis podendo ser modificados durante a execução. O *JAFMAS* adota os conceitos da linguagem de coordenação *COOL* [3] que se baseia, principalmente, em diferentes formas de interação entre agentes para solucionar o problema da coordenação.

Em [17] são discutidos diversos enfoques dados à coordenação em sistemas multiagente e é proposto uma arquitetura de agentes que permite a adaptação e a implementação de diferentes estratégias de coordenação de acordo com a situação.

Aplicações distribuídas compostas de agentes móveis também requerem formas de coordenação, pois tipicamente estas consistem de diversas tarefas que são delegadas a agentes e estes precisam trabalhar cooperativamente (trocar dados e resultados) no sentido de alcançar um objetivo global da aplicação. Agentes móveis não possuem um endereço fixo, o que dificulta a sua localização por outros agentes da mesma aplicação. Conforme veremos na próxima seção, existem diversos sistemas que implementam diferentes modelos de coordenação para agentes móveis. Apresentamos a seguir, os modelos de coordenação existentes, suas vantagens e desvantagens e sistemas que implementam estes modelos.

2.1 Modelos de Coordenação para Agentes Móveis

Nesta seção, apresentamos uma classificação dos modelos de coordenação de agentes móveis proposta por Cabri, Leonardi e Zambonelli [5] com base em **acoplamento espacial e temporal**.

Um modelo apresenta **acoplamento espacial** se as entidades envolvidas na coordenação devem compartilhar um espaço de nomes para se comunicar, ou seja, aquele que inicia a comunicação deve conhecer o nome (ou identidade) do parceiro.

Nos modelos de **acoplamento temporal** as comunicações são síncronas, isto é, o remetente e o receptor devem estar sincronizados no momento da comunicação.

Quatro categorias principais de modelos de coordenação podem ser derivadas: (i) *direta*, espacialmente e temporariamente acopladas; (ii) *meeting-oriented*, espacialmente não-acoplada e temporariamente acoplada; (iii) baseada em *blackboard*, espacialmente acoplada e temporariamente não-acoplada; (iv) *tipo Linda*, espacialmente e temporariamente não-acoplada.

2.1.1 Coordenação Direta

Em modelos de coordenação direta, os agentes enviam uma mensagem diretamente para outro agente. Este modelo implica em acoplamento espacial, pois o remetente deve conhecer o receptor e acoplamento temporal, pois o receptor deve existir no momento da comunicação (sincronização das entidades envolvidas). Na coordenação inter-agente, os agentes devem concordar em um protocolo de comunicação, tipicamente um envio de mensagens **ponto-a-ponto**. O acesso a recursos locais em *hosts* geralmente é da forma **cliente-servidor**: o *host* provê servidores locais para o gerenciamento de seus recursos, um agente requer um dado serviço fornecendo os parâmetros necessários.

Espacial \ Temporal	Acoplada	Nao-acoplada
Acoplada	Direta *ASDK, AgentTCL*	Baseada em Blackboard *Ambit, ffMain*
Nao-acoplada	Orientada a Encontros *Ara, Mole*	Tipo Linda *Jada, MARS*

Figura 2.1: Modelos de coordenação para Agentes Móveis

Este modelo é apropriado para as aplicações tradicionais baseadas no modelo cliente-servidor, mas não é apropriado para aplicações que envolvem agentes móveis. As desvantagens são:

- necessidade de protocolos de roteamento complexos e dinâmicos que possibilitem a localização de agentes móveis;
- necessidade de uma conexão estável da rede;
- agentes móveis podem ser criados dinamicamente o que dificulta a identificação de quais agentes existem em um dado momento.

Existem muitos exemplos de sistemas que adotam este modelo de coordenação direta, inclusive sistemas baseados em linguagens orientadas a objetos. Estes permitem que dois agentes se coordenem através da invocação de métodos em objetos/agentes remotos. *Sumatra* [1], *Odyssey* [13] e *ASDK* [2] são sistemas baseados em Java que adotam este estilo cliente-servidor. No *ASDK*, em particular, a comunicação entre agentes é muito restrita, uma vez que não é possível enviar uma mensagem a um agente (*aglet*) remoto a não ser que se tenha uma referência para este agente (que no entanto, só permanece válida enquanto o agente remoto não migrar para outro lugar) ou ter o conhecimento do identificador e o endereço corrente deste agente. *AgentTCL* [14] provê comunicação direta entre dois agentes através de trocas de mensagens assíncronas ou estabelecendo um encontro (síncrono) entre agentes móveis.

2.1.2 Coordenação Orientada a Encontros

Este modelo de coordenação *orientada a encontros* soluciona o problema da necessidade de localizar os agentes móveis (encontrado na coordenação direta) e um agente não precisa conhecer explicitamente o nome do parceiro da comunicação. Interações ocorrem na forma de encontros de agentes em um lugar comum. Uma entidade (o gerenciador do lugar) assume o papel de coordenador do encontro para abrir e definir um ponto de encontro (local onde será feito o encontro). Os agentes ingressam em pontos de encontro para se comunicar e sincronizar com outros agentes que estão participando em tais encontros. Encontros são restritos localmente: um encontro deve ser estabelecido em um determinado lugar e apenas agentes locais podem participar do mesmo. Encontros não evitam uma ausência completa de acoplamento espacial:

agentes devem compartilhar os nomes de lugares ou eventos que os forcem a se mover para os mesmos.

Este modelo oferece maior flexibilidade e dinamismo, resolvendo parcialmente o problema de identificar os parceiros envolvidos. A maior desvantagem é que força uma sincronização estrita entre agentes: agentes devem estar presentes no ponto de encontro para interagir. Além disso, se o encontro não for restrito localmente, ele deve ser implementado por troca de mensagens e, com isto, acabará herdando todas as desvantagens - em termos de eficiência e confiabilidade - do modelo de coordenação direta.

Este modelo de coordenação *orientada a encontros* está implementado em *ARA* [22], onde um agente faz o papel de servidor de encontros anunciando um ponto de encontro (uma localização) para cada *host*, nos quais agentes podem ingressar para se coordenarem. Outro exemplo é o *Sistema de Agentes Móveis MOLE* [4] que integra o conceito de comunicação baseada em eventos e sincronização definidos pela OMG.

2.1.3 Coordenação baseada em Blackboard

Na coordenação baseada em *blackboard*, os agentes interagem através de espaços de dados compartilhados locais em cada *host*, utilizados como repositórios comuns para armazenar e recuperar mensagens. De certa forma, o *blackboard* é similar ao modelo *orientado a encontros* sem a restrição à presença simultânea dos agentes envolvidos. O remetente simplesmente deixa a mensagem no *blackboard* e o receptor pode ler a mensagem recuperando-a do *blackboard*. Não há necessidade que o remetente e o receptor existam no mesmo instante.

A vantagem mais significativa de modelos de coordenação indireta deriva do não-acoplamento temporal: mensagens são deixadas no *blackboard* sem a preocupação de onde estão os receptores correspondentes e quando eles irão ler as mensagens. Além disso, uma vez que toda interação inter-agente deve ser executada através do *blackboard*, os *hosts* podem facilmente monitorar e controlar todas as interações e, portanto, este é um modelo de execução mais seguro se comparado ao de coordenação direta.

A desvantagem deste modelo vem do acoplamento espacial, que requer que as entidades interagentes concordem com os nomes das mensagens.

Recentemente vários sistemas implementam este tipo de coordenação para agentes móveis. *Ambit* [7] apresenta um modelo formal para computações móveis, introduz o conceito de coordenação indireta onde agentes podem fixar e ler mensagens em um *blackboard* local a cada *site*. O sistema *ffMAIN* [11] define agentes móveis que interagem entre si através de um *espaço de informações* que é acessado através do protocolo HTTP, onde dados podem ser armazenados, lidos e extraídos.

2.1.4 Coordenação tipo Linda

Nos modelos de coordenação do *tipo Linda*, os acessos ao *blackboard* local se baseiam em um mecanismo associativo: as informações são organizadas em tuplas e recuperadas de uma forma associativa através do mecanismo de reconhecimento de padrão. Este mecanismo permite

recuperar mensagens especificando apenas algumas partes da própria mensagem. *Blackboards* associativos (chamados de espaços de tuplas) garantem o desacoplamento completo entre os participantes de uma interação, não requerendo sincronização ou conhecimento mútuo dos agentes. O acesso ao espaço de tuplas é feito sempre com o conjunto básico de operações (tipo *Linda*).

Coordenação associativa é apropriada para aplicações que envolvem agentes móveis na Internet. Por ser um ambiente dinâmico e heterogêneo a Internet requer que os agentes possuam um mecanismo de reconhecimento de padrões para permitir uma interação anônima baseada na sintaxe da tupla. Assim, agentes conseguem interagir com agentes de outras aplicações e também com agentes previamente desconhecidos.

Estendendo o conceito de *blackboard* associativo, o *blackboard* reativo provê a capacidade de introduzir reações programáveis em resposta aos acessos feitos no *blackboard* pelos agentes. Este modelo é útil para implementar políticas locais específicas para a interação entre agentes a fim de proteger o ambiente ou implementar políticas de gerenciamento para alcançar melhor eficiência na execução de agentes.

Desvantagens dos modelos de coordenação indireta (baseados em *blackboard*) incluem: requerem uma grande quantidade de recursos para o armazenamento dos dados e causam uma dificuldade maior de programação, visto que os dados e controle estão mais dissociados.

O *Jada* [10] é uma implementação do *blackboard* associativo em Java que pode ser usada por agentes móveis para armazenar e recuperar referências de objetos de forma associativa. Outro sistema é o *MARS* [6], que implementa uma arquitetura de coordenação para agentes móveis baseados em Java. O *MARS* define um modelo de *espaço de tuplas* reativo no qual os efeitos de operações sobre o *espaço de tuplas* podem ser dinamicamente modificados. O acesso ao espaço de tuplas é feito sempre com o mesmo conjunto básico de operações (tipo *Linda*).

Conforme descrevemos acima, estes modelos de coordenação oferecem alguma forma de interação entre agentes móveis mas todos os modelos impõem alguma exigência para que isto seja alcançado. Além disso, nenhum destes modelos trata da localização dinâmica dos agentes móveis. A nossa proposta está enfocada no problema da coordenação de agentes móveis oferecendo uma forma alternativa para a coordenação. A idéia é que os agentes possam interagir de forma tranparente para sincronizar as suas atividades independentemente da localização e sem as exigências requeridas pelos outros modelos de coordenação.

Capítulo 3

Coordenação através do Canal de Broadcast

Conforme descrevemos no capítulo anterior, em aplicações distribuídas baseadas em agentes móveis uma das tarefas mais difíceis de ser alcançada é a coordenação de ações dos agentes móveis. Isto se deve, principalmente, à incerteza quanto à localização de agentes, dado que estes podem mudar de endereço dinamicamente.

Apresentamos um mecanismo para coordenação de agentes móveis que contorna este problema. Através do nosso mecanismo de coordenação, um agente não precisa conhecer a localização de outros agentes para que possa interagir com os mesmos: as mensagens são enviadas a um elemento cujo endereço é conhecido e este é encarregado de repassar as mensagens aos destinatários de forma que todos os agentes as recebam, independentemente de sua localização.

3.1 Modelo do sistema e hipóteses

Este mecanismo de coordenação de agentes móveis (chamado *Canal de Broadcast*) se baseia na difusão de mensagens. No entanto, para conseguir lidar com a mobilidade de agentes, o nosso mecanismo requer a presença de certas componentes de infra-estrutura. Uma delas é um elemento de endereço fixo e conhecido (*Broadcast Proxy* ou, simplesmente, *Proxy*) que é responsável por difundir e controlar as mensagens que devem ser enviadas a grupos de agentes móveis que cooperam na execução de uma determinada tarefa.

A estratégia para localizar os agentes móveis é a seguinte: cada agente móvel pertence a um domínio que é definido de acordo com a localização do agente (*lugar*). Cada *lugar* está associado a um único domínio e todos os *lugares* têm um domínio correspondente. Para cada um destes domínios, existe um representante (*lugar de referência*) que mantém a informação sobre a localização exata (*lugar*) dos agentes móveis executando dentro do respectivo domínio.

O *lugar de referência* e o *lugar* têm a função de intermediários na comunicação entre o *Proxy* e os agentes móveis. Todo envio de mensagem a partir do *Proxy* para os agentes móveis é feito passando-se a mensagem, inicialmente, para os *lugares de referência* correspondentes dos domínios. Em seguida, estes *lugares de referência*, passam a mensagem aos *lugares* pertencentes

aos seus respectivos domínios. Finalmente, os *lugares* passam a mensagem aos agentes que estão, no momento, executando localmente.

A escolha da difusão de mensagem como elemento fundamental para a implementação de nosso mecanismo foi motivada pela percepção de que muitos protocolos de coordenação são naturalmente expressos na forma de uma comunicação multiponto, conforme pode ser comprovado nos exemplos do Capítulo 4.

Como parte do modelo, assumiremos que:

- não há perda de mensagens;
- não há partição na rede;
- durante a configuração do *Canal de Broadcast* (descrita na Seção 3.7) não há migração de agentes;
- todos os elementos da infra-estrutura do *Canal de Broadcast* (descritos na Seção 3.1.1) estão sempre disponíveis;
- o grupo de agentes móveis que participam de uma determinada tarefa é estático;
- os itinerários dos agentes incluem somente lugares que foram previamente instanciados durante a etapa de configuração da infra-estrutura e cada um deles está associado a um domínio;
- os agentes podem migrar um número arbitrário de vezes, mas a média dos períodos de permanência de um agente em um *lugar* deve ser maior que a média das durações de migrações.

Na comunicação entre o *Proxy* e os representantes de domínio (*lugares de referência*) e entre *lugares de referência* e *lugares*, não fazemos nenhuma suposição sobre tempos de transmissão de mensagens (sistemas assíncronos). Apesar desta fraca suposição, este nosso mecanismo de coordenação garante a entrega de mensagens difundidas pelo *Canal de Broadcast* em ordem total, de forma que todos os agentes possam tratar as mensagens na mesma ordem em que foram enviadas pelo *Proxy*.

Ressaltamos que, por estar baseado em um elemento central e por requerer uma configuração prévia dos domínios, este mecanismo é mais apropriado para aplicações em redes locais.

3.1.1 Conceitos básicos

Nesta seção, apresentaremos os elementos que constituem este mecanismo de coordenação.

- **lugar** (Place): corresponde ao local de execução de agentes móveis. Representa uma localização para o agente e equivale ao conceito de *contexto* do ASDK. Cada *lugar* está associado a um *host* da rede, mas cada *host* pode "executar" vários *lugares*.

- **domínio**: é um agrupamento de *lugares*. Um destes *lugares* é escolhido para ser o representante do domínio. Um domínio pode ter qualquer número de *lugares*. É deixado a critério do programador a divisão dos *lugares* em domínios, assim como a escolha do representante do domínio.

- **lugar de referência** (Reference Place): é o representante de um domínio. É um caso particular (extensão) de *lugar* e possui as seguintes funções: administrar um conjunto de *lugares* e agentes móveis, remeter mensagens para os agentes localizados em seu domínio e alterar a referência de agentes quando estes mudam de domínio, notificando neste caso também o *Proxy*.

- **Proxy**: é um elemento fixo e conhecido pelos *lugares de referência* e pelo grupo de agentes móveis que participam na execução conjunta de uma determinada tarefa. O *Proxy* mantém informações sobre os *lugares de referência* existentes, os agentes móveis participantes do grupo e o domínio em que cada um deles está localizado. As principais funções do *Proxy* são: difundir toda mensagem recebida (de um agente membro do grupo) para os demais agentes do grupo (através dos *lugares de referência* e *lugares*) e atualizar a referência de um agente quando este migra para outro domínio. As mensagens são enviadas para todo o grupo de agentes de forma que todos os agentes tomem conhecimento das mesmas.

- **mensagem**: é o objeto de comunicação entre os elementos citados acima. Cada mensagem possui um identificador único (número de seqüência atribuído pelo *Proxy*). As mensagens podem ser de dois tipos: de controle (CONTROL) ou de aplicação (APPL). As mensagens de controle são mensagens pré-definidas que controlam o funcionamento do mecanismo *Canal de Broadcast*. As mensagens do tipo APPL são as mensagens específicas da aplicação, e que determinam as mensagens básicas para a coordenação específica na aplicação. Para maiores detalhes, na Seção 3.6 apresentamos uma tabela com os tipos de mensagens de controle e as suas respectivas funções.

3.1.2 Domínios e Lugares de Referência

Agentes móveis, por definição, possuem um itinerário indicando um roteiro de migração pelos *lugares* da rede nos quais acessam recursos, coletam informações, etc., com a finalidade de executar uma determinada tarefa. Os itinerários de todos os agentes que participam de uma tarefa formam um conjunto de localizações (endereços) onde cada uma das localizações poderá ser visitada por um agente em algum momento da execução. Este conjunto de localizações constitui o conjunto mínimo de *lugares* que farão parte da infra-estrutura do *Canal de Broadcast* e este conjunto de *lugares* é dividido em domínios.

Com relação à localização de um agente, o *Proxy* mantém apenas informações sobre o domínio no qual cada agente móvel está localizado, enquanto que o *lugar de referência* conhece a localização exata (endereço do *lugar*) de cada um dos agentes móveis atualmente em seu domínio.

Esta divisão em domínios facilita o trabalho do *Proxy* de localizar os agentes móveis: toda vez que um agente migra de um domínio para outro, os *lugares de referência* dos domínios envolvidos trocam informações e o *Proxy* só é notificado desta mudança mas não conhece a localização específica de cada agente. A única alteração que o *Proxy* precisa fazer é atualizar o

domínio para aquele agente móvel. Quando há uma migração intra-domínio, no entanto, somente o *lugar de referência* precisa alterar a sua informação sobre a localização do agente. Isto evita que o *Proxy* seja sobrecarregado com notificações de alterações de endereços de agentes.

Desta forma, o *Canal de Broadcast* implanta um gerenciamento de localização em dois níveis e de forma semi-descentralizada.

3.1.3 Visão geral do mecanismo

Na Figura 3.1, apresentamos uma visão geral da arquitetura deste mecanismo. Cada retângulo maior corresponde a um domínio (D_1, D_2,..., D_n) com seus respectivos *lugares de referência* (LR_1, LR_2,..., LR_n), representados por retângulos internos hachurados; um conjunto de *lugares* (l_1, l_2,..., l_j) representados por retângulos brancos, e um conjunto de agentes móveis representados por círculos. Neste exemplo, é suposto que todos os agentes móveis deste conjunto estão registrados no *Proxy*. Na parte superior, temos o *Proxy* instanciado para este grupo de agentes móveis. Em ①, temos a representação de AM_1 enviando uma mensagem para o *Proxy* e esta mensagem sendo enviada para os *lugares de referência* em ②. Em ③, temos cada *lugar de referência* enviando a mensagem para os *lugares* em seu domínio. Em ④, os *lugares* estão passando a mensagem aos agentes e em ⑤, os agentes estão confirmando o recebimento desta mensagem ao *Proxy*.

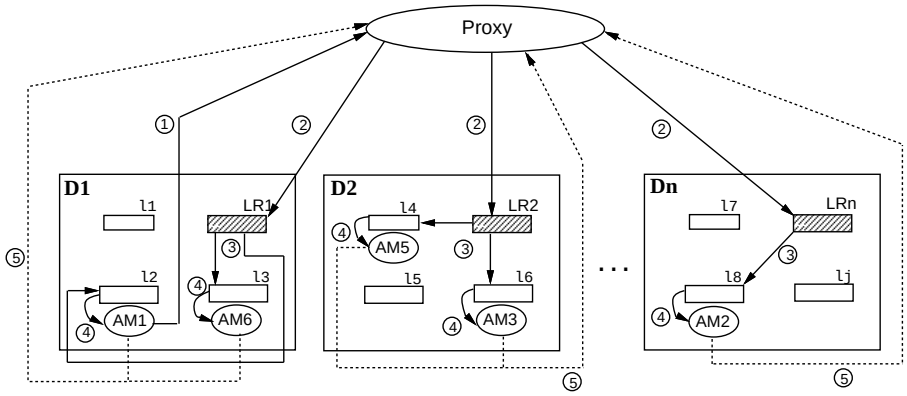

Figura 3.1: Arquitetura do modelo

3.2 Funcionamento do Canal de Broadcast

Nesta seção, apresentamos o funcionamento básico do *Canal de Broadcast* em relação à sua instanciação, o envio de mensagens, o tratamento das migrações dos agentes móveis e a ordenação de mensagens.

É importante ressaltar que o uso do *Canal de Broadcast* (conforme descrevemos a seguir) pressupõe uma configuração completa e correta dos elementos da infra-estrutura envolvida (ou seja, os *lugares*, *lugares de referência* e *Proxy* são válidos). Assumimos que a migração de todos os agentes móveis registrados junto ao *Proxy* tem início somente após esta configuração.

Apresentamos uma seqüência de passos para a configuração inicial do *Canal de Broadcast* na Seção 3.7 e, no Capítulo 6, apresentamos uma descrição detalhada sobre a forma de instanciação dos elementos da infra-estrutura.

3.2.1 Instanciação do Canal de Broadcast e registro de agentes móveis

Uma instância do *Canal de Broadcast* (*Proxy*) é criada para cada grupo de agentes móveis que participam de uma determinada tarefa. Cada uma destas instâncias possui um identificador único e imutável de modo que os agentes de um grupo possam interagir apenas com aquele *Proxy* que foi instanciado para o seu grupo. Desta forma, agentes que não pertencem a um determinado grupo não podem interferir na execução da tarefa relativa a este grupo (agentes de aplicações diferentes terão identificadores de *Proxy* diferentes).

Assim como o *Proxy*, os *lugares de referência* e *lugares* também possuem um identificador único e cada um desses elementos estão associados a um endereço. O *Proxy* pode ser instanciado passando-se como parâmetro o conjunto de identificadores e endereços dos *lugares de referência* dos domínios definidos. O *Proxy* não precisa ter informações sobre os *lugares* dentro de cada domínio, pois, não há comunicação entre *lugares* e *Proxy* definida para este mecanismo.

Após a instanciação do *Proxy*, são enviados o seu identificador e endereço para os *lugares de referência* que passarão para os *lugares* dentro dos respectivos domínios. Os *lugares de referência* necessitam destas informações a respeito do *Proxy* a fim de trocar mensagens com o mesmo. Os lugares, por sua vez, necessitam desta informação para identificar a que *Proxy* pertencem.

A seguir, os agentes móveis são instanciados seqüencialmente, passando como parâmetros o identificador e endereço do *Proxy* e o seu itinerário no caso de o agente ser móvel). Após a sua criação, o agente se registra no *Proxy* enviando uma mensagem de registro ao mesmo, passando como parâmetros o seu identificador e o endereço do *lugar de referência* do domínio em que está correntemente. Este registro possibilitará que o agente se comunique com outros agentes do grupo através do *Proxy*.

O *Proxy* ao receber esta mensagem, toma conhecimento da existência de um novo agente em determinado domínio e acrescenta os dados deste agente no conjunto de agentes participantes do grupo a ser coordenado. O *Proxy* possui uma tabela de referências (conjunto de agentes participantes) onde a associação (agente/endereco do *lugar de referência*) é mantida, possibilitando consultar, a qualquer momento, em qual domínio está localizado um agente móvel.

O *Proxy* notifica os *lugares de referência* a respeito do registro deste novo agente e esta mesma notificação (mensagem) é passada dos *lugares de referência* a *lugares*. De acordo com os argumentos desta mensagem, um *lugar de referência* determina se um agente está em seu domínio e em caso afirmativo, esta informação (agente/endereço do *lugar*) é armazenada em um conjunto local de agentes (uma tabela de referências relativa aos agentes do domínio). Da mesma forma, o *lugar* ao receber esta mensagem, verifica se o endereço em que o agente está localizado é o próprio endereço e, em caso afirmativo, as informações sobre este agente são inseridas no conjunto de agentes local (estrutura que contém os agentes que estão localizados atualmente no *lugar*). Após este procedimento, todos os agentes instanciados estarão registrados adequadamente no *Proxy*, *lugar de referência* e *lugar*, dependendo do local onde estará executando inicialmente. Quando há migração de agentes, estes registros são atualizados conforme descrevemos na subseção 3.2.3.

3.2.2 Envio e recebimento de mensagens

Os agentes móveis enviam mensagens para os outros agentes móveis do grupo através do *Proxy* instanciado para o grupo. O *Proxy* é responsável por difundir e manter a mensagem até receber a confirmação de todos os agentes do grupo. Isto é feito da seguinte forma: ao enviar uma mensagem, o *Proxy* cria para esta mensagem uma lista dos agentes móveis destinatários e armazena a mensagem. Um agente, ao receber esta mensagem, envia para o *Proxy* uma confirmação de recebimento referente àquela mensagem. Ao receber esta confirmação, o *Proxy* remove o agente da lista de destinatários daquela mensagem. Quando esta lista estiver vazia, todos os agentes móveis receberam a mensagem e portanto, a mensagem poderá ser removida do *Proxy*.

O *lugar de referência* tem informações sobre os agentes móveis em seu domínio e os seus respectivos endereços (*lugares*).

Conforme descrevemos anteriormente, uma mensagem enviada do *Proxy* para os agentes é sempre de forma indireta: o *Proxy* envia a mensagem para o *lugar de referência* responsável pelo domínio e os *lugares de referência* repassam a mensagem para os *lugares* em seu domínio e os *lugares*, por sua vez, passam a mensagem para os agentes que estão localizados em seu endereço. No entanto, somente o *Proxy* recebe a confirmação de recebimento de mensagens por parte dos agentes.

Na próxima seção, descreveremos como se comportam os *lugares*, *lugares de referência* e o *Proxy* para que um agente móvel receba uma mensagem que foi enviada no momento de sua migração.

3.2.3 Migração de agentes móveis e re-envio de mensagens

Um agente móvel está sujeito a não receber as mensagens endereçadas para ele enquanto está migrando de um *lugar* para outro. Isto porque o *lugar de referência* não é notificado antes da migração do agente e portanto, continua enviando as mensagens para o *lugar* em que o agente estava antes da migração. Este problema é solucionado conforme descrevemos a seguir.

Toda mensagem difundida pelo *Proxy* possui um identificador único que é atribuído pelo *Proxy* e trata-se de um valor inteiro (número de seqüência da mensagem). Este número de seqüência é importante para garantirmos a ordenação total das mensagens (conforme descreveremos na Seção 3.2.5) e para o controle de re-envio de mensagens aos agentes móveis.

Assim como o *Proxy*, os *lugares de referência* também armazenam cada uma das mensagens enviadas e, desta forma, poderão obter informações sobre quais mensagens foram enviadas para os agentes móveis dentro de seu domínio (e que ainda não foram confirmadas por todos agentes móveis do grupo no *Proxy*). Trata-se de uma *fila de mensagens* que cada *lugar de referência* possui. Toda vez que uma mensagem é enviada para os agentes no domínio, esta mensagem é inserida na *fila de mensagens*.

Um agente móvel tem informações sobre quais mensagens já recebeu em um dado momento e quais já foram tratadas pelo mesmo.

Estas informações sobre as mensagens enviadas e recebidas são importantes no momento

em que um agente perde mensagens durante a migração de um *lugar* para outro e necessite do re-envio destas mensagens. De acordo com este mecanismo, teremos dois tipos de migrações possíveis: intra- e inter-domínio. A seguir, analisaremos o re-envio de mensagens nestes dois casos de migração.

Na migração intra-domínio, o agente notifica a sua entrada ao *lugar* passando um conjunto de mensagens recebidas (identificadores ou número de seqüência das mensagens) e a última mensagem tratada pelo agente (a este conjunto de mensagens chamaremos de *repositório de mensagens*). Este novo *lugar* é encarregado de notificar o *lugar de referência* responsável pelo domínio a entrada do agente, passando o conjunto de identificadores de mensagens recebido. O *lugar de referência* através destes identificadores, compara com as mensagens da *fila de mensagens* e re-envia aquelas que não foram recebidas ao *lugar* onde se encontra o agente.

Na migração inter-domínio, o *lugar de referência* recebe uma notificação da entrada do agente no domínio. Neste momento, envia uma notificação ao *lugar de referência* do antigo domínio sobre a saída do agente e o antigo *lugar de referência* simplesmente remove o agente de seu conjunto local de agentes. E o novo *lugar de referência* envia uma mensagem de atualização de domínio para o *Proxy*. O *Proxy* verifica quais mensagens aquele agente confirmou até o momento e as re-envia para o novo *lugar de referência*. O novo *lugar de referência*, por sua vez, passa a mensagem para o *lugar* onde está localizado o agente e este, finalmente, repassa a mensagem para o agente.

Observamos que o re-envio de mensagens feito tanto pelo *lugar de referência* como pelo *Proxy* é direcionado somente para o agente específico que fez a migração e, portanto, a mensagem não é inserida novamente nas estruturas que armazenam mensagens (no *lugar de referência* e no *Proxy*).

Através deste procedimento, os agentes podem migrar constantemente para vários *lugares* sem o risco de não receber alguma mensagem enviada.

3.2.4 Confirmação de mensagens

Conforme descrevemos anteriormente, toda vez que um agente recebe uma mensagem, este deve confirmar o seu recebimento diretamente ao *Proxy*.

Nesta confirmação, o agente envia o seu próprio identificador e o identificador da mensagem recebida. Quando o *Proxy* recebe esta mensagem, além de remover o agente da lista de destinatários para aquela mensagem, verifica se ainda existem agentes que não confirmaram esta mensagem.

Se todos os agentes tiverem confirmado a mensagem, o *Proxy* remove a mensagem de seu conjunto de mensagens e envia uma notificação aos *lugares de referência* para que removam também esta mensagem de suas respectivas *filas de mensagens*.

Devido à suposição de que a média dos períodos de permanência em *lugares* deve ser maior que a média dos períodos de duração de migração, as mensagens serão confirmadas em algum momento e, assim, evitamos que as estruturas que armazenam as mensagens cresçam indefinidamente tanto no *Proxy* quanto nos *lugares de referência*.

3.2.5 Ordenação de mensagens

Conforme descrevemos anteriormente, toda mensagem recebe um identificador único (número de seqüência) no momento em que é difundida pelo *Proxy*.

O *Proxy* faz o controle e criação de identificadores de mensagem através de uma variável inteira (um contador) que inicialmente tem o valor zero e, para cada mensagem a ser difundida, incrementa o valor desta variável e o atribui à mensagem.

Os *lugares de referência* e *lugares* não se preocupam com este identificador, apenas passam as mensagens adiante.

A ordenação total é garantida pela forma pela qual os agentes tratam as mensagens que chegam. O agente possui uma variável que indica qual foi a última mensagem tratada e portanto, tem conhecimento de qual será a próxima mensagem que deverá ser tratada em seguida (valor desta variável incrementada de um). Enquanto receber mensagens que possuem identificadores maiores que o esperado, o agente insere estas mensagens em um vetor de mensagens não tratadas. Quando a mensagem esperada chega, o agente trata esta mensagem e verifica se a próxima esperada está no vetor de mensagens não tratadas (se já havia chegado anteriormente) e trata seqüencialmente as mensagens deste vetor buscando as que possuem o identificador esperado. Cada mensagem obtida do vetor, após ser tratada pelo agente, é removida do vetor de mensagens não tratadas.

Se o agente receber uma mesma mensagem mais de uma vez, o agente simplesmente ignora a mensagem (caso esta tenha o identificador menor que a última mensagem tratada) ou, esta mensagem poderá estar mais de uma vez no vetor de mensagens não tratadas. Porém, ao tratar a primeira destas, as próximas serão ignoradas pois a próxima mensagem a ser tratada deverá ter o identificador maior do que as que estão repetidas no vetor.

Através deste método, as mensagens são tratadas pelos agentes na mesma ordem em que saem do *Proxy*.

3.3 Protocolo Hand-off

A Figura 3.2 apresenta uma situação de envio (difusão) de uma mensagem pelo *Proxy* até a obtenção de uma confirmação do recebimento da mensagem (os detalhes das mensagens de controle estão apresentados na Seção 3.6). Neste exemplo, estamos supondo que temos dois domínios (D_1 e D_2), com *lugares de referência* LR_1 e LR_2, respectivamente. Em D_1, temos *lugares* L_1 e L_2, e em D_2, temos L_3 e L_4. Consideramos também um agente móvel k, registrado no *Proxy*, localizado inicialmente em L_1.

Inicialmente, o *Proxy* está difundindo uma mensagem $m1$. Porém, quando esta mensagem chegou em L_1, o agente k já havia migrado para L_2. Antes da migração, o agente notificou L_1 através de GOING_TO(ID), passando como parâmetro o seu próprio identificador. Ao chegar ao novo *lugar*, o agente enviou a mensagem ARRIVAL(ID,0), passando o *repositório de mensagens* (estamos supondo que o agente k não havia recebido nenhuma mensagem até o momento) e o identificador como parâmetros. O *lugar* L_2, ao receber esta mensagem, envia a mensagem REGISTRY($LR_1, L_2, 0$)) para LR_1, passando como parâmetros o *lugar de referência*, o próprio

identificador e o *repositório de mensagens* que o agente k havia passado como parâmetro. Quando LR_1 recebe esta mensagem e verifica que a migração é intra-domínio e que o agente k não recebeu a mensagem $m1$ que havia sido enviada, re-envia, então, a mensagem. Mas, suponhamos que neste interim, o agente k novamente migrou e desta vez para o *lugar* L_3, entrando no domínio D_2.

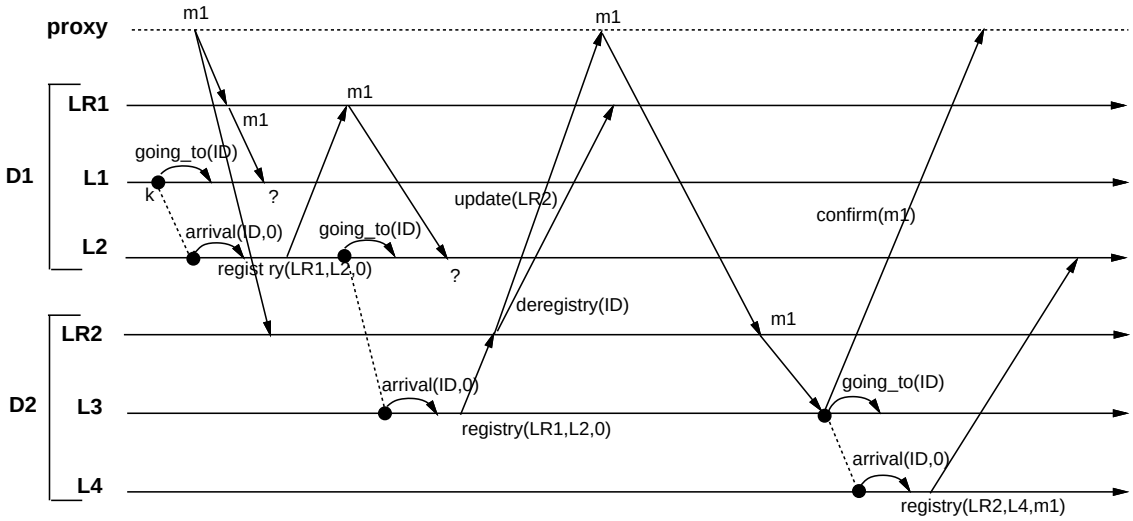

Figura 3.2: Protocolo Hand-off

O *lugar* L_3 notifica LR_2 sobre a entrada do agente e LR_2, ao receber esta notificação, verifica que a migração é inter-domínio e, envia uma mensagem de atualização UPDATE(LR_2) ao *Proxy* passando o seu identificador como parâmetro e uma mensagem para que o *lugar de referência* do domínio anterior (LR_1) de-registre o agente daquele domínio. Isto é feito através de DEREGISTRY(AgentID). O *Proxy* verifica que o agente k não confirmou a mensagem enviada anteriormente e, novamente a envia para o agente agora através de LR_2 (não faz a difusão). O agente finalmente recebe a mensagem e confirma o recebimento diretamente ao *Proxy* através de CONFIRM(msgID). Ao final, o agente migra novamente mas, neste caso, a mensagem não é re-enviada ao agente uma vez que o conjunto de mensagens recebidas é passado ao *lugar de referência.*

Devido ao fato de não fazermos nenhuma suposição sobre tempos de transmissão de mensagens, as mensagens de controle podem não chegar em um ordem esperada nos elementos intermediários. Por exemplo, em casos em que um agente tenha migrado duas vezes seguidas de um domínio para outro e a mensagem de atualização do domínio corrente chega ao *Proxy* antes da mensagem de atualização do domínio anterior, o *Proxy* atualiza o endereço do agente para o domínio incorreto e todas as mensagens são re-enviadas para este domínio (onde o agente não mais se encontra). Neste caso, o agente que fez a migração irá receber a mensagem em uma futura migração (além de atualizar corretamente seu endereço) ou, no momento de sua destruição.

3.4 Término do Canal de Broadcast

Quando os agentes móveis tiverem executado as suas tarefas e alcançado os seus objetivos, os elementos que compõem o mecanismo do *Canal de Broadcast* poderão ser removidos a pedido do usuário. Isto é feito em duas etapas, inicialmente, o *Proxy* envia a mensagem DISPOSE_ALL (vide Seção 3.6) aos agentes (através dos *lugares de referência* e *lugares*) que faz com que cada agente envie a mensagem DISPOSE_AGENT para o *Proxy* a fim de se de-registrar, e após o recebimento de uma confirmação do *Proxy* em que são recebidas todas as mensagens pendentes, se auto-destroem. Esta mensagem (DISPOSE_AGENT) é passada aos *lugares de referência* e *lugares* fazendo com que estes também removam o agente de suas estruturas locais. Em uma segunda etapa, quando todos os agentes já se de-registraram, o *Proxy* envia a mensagem END para que todos os demais elementos (*lugares de referência* e *lugares*) sejam removidos. Por fim, o próprio *Proxy* se auto-destrói.

3.5 Descrição informal do protocolo

Nesta seção descreveremos, de maneira informal, o protocolo utilizado para a implementação do nosso mecanismo de coordenação. Observamos que a fase de configuração/inicialização do *Canal de Broadcast* não faz parte do protocolo descrito a seguir, o termo protocolo se refere apenas à fase de utilização do mesmo. A configuração da infra-estrutura do *Canal de Broadcast* está descrita no Capítulo 6.

Este protocolo está baseado essencialmente em eventos de comunicação (envio e recebimento de mensagens) entre os elementos que compõem o mecanismo. Na Seção 3.6 apresentamos os tipos de mensagens definidas para o controle do *Canal de Broadcast*.

Os quatro elementos que fazem parte deste protocolo são: *Proxy*, *lugares de referência* (*Reference Place*), *lugares* (*Place*) e agentes móveis (*Agent*). O comportamento de cada um destes elementos foi especificado como um conjunto de pares `evento` → `ação`, onde `ação` é uma seqüência de operações executada atomicamente em reação à ocorrência de `evento`. Na implementação deste mecanismo, usamos esta especificação como base e utilizamos a facilidade de sincronização de métodos do Java para garantir a atomicidade da componente `ação`.

A seguir, apresentamos as notações adotadas no pseudo-código da especificação do protocolo, uma descrição geral de cada um dos elementos (*Proxy*, *Reference Place*, *Place* e *Agent*) com as respectivas seqüências de ações que devem ser executadas para tratar os eventos e o pseudo-código correspondente a cada um desses elementos.

3.5.1 Notação adotada no pseudo-código

Nesta seção, apresentamos a notação utilizada no pseudo-código para representar as operações básicas executadas pelos elementos que compõem este mecanismo. As principais operações são: criação e envio de mensagens, comparação de endereços e inserção, atualização e remoção de elementos em conjuntos de elementos.

Inicialmente, apresentamos o objeto de comunicação entre os elementos deste mecanismo: a

mensagem. A mensagem possui dois atributos principais: um, que corresponde ao tipo da mensagem que pode ser de controle ou de aplicação (CONTROL ou APPL) e, o outro, que existe somente se a mensagem é de controle, corresponde ao tipo da mensagem de controle. Este segundo atributo corresponde a um dos tipos de mensagens apresentados na Tabela 3.1 da Seção 3.6. Cada um dos elementos deste mecanismo pode criar, enviar, receber e tratar mensagens. Na criação de uma mensagem, pode-se incluir um ou mais argumentos que pode(m) estar associado(s) a um determinado valor.

As operações básicas utilizadas no pseudo-código são:

- `createMsg(controlType)`: cria uma nova mensagem de controle do tipo especificado como parâmetro.
- `send(Addr,msg)`: envia a mensagem passada como parâmetro ao endereço especificado.
- `add(set,element)`: adiciona o elemento ao conjunto especificado.
- `update(set,element)`: atualiza um dado no conjunto especificado.
- `rem(set,element)`: remove o elemento do conjunto especificado.
- `compareAddr(Addr)`: retorna verdadeiro se o endereço especificado é igual ao próprio endereço.
- `setArg(name,value)`: operação associada à mensagem, tem a função de acrescentar argumentos à mesma.
- `getArg(name)`: operação associada à mensagem, tem a função de obter o valor do argumento associado ao nome especificado.

3.5.2 Proxy

O *Proxy* é o responsável por manter atualizadas as informações sobre a localização (no nível de domínio) de cada elemento do grupo de agentes móveis, além de controlar as mensagens que os agentes devem receber. Para que isto seja alcançado, o *Proxy* deve ser notificado dos seguintes eventos: criação de agentes, migração inter-domínio de agentes e confirmação de recebimento de mensagens por agentes.

A criação de um agente é notificada ao *Proxy* pelo próprio agente no momento de sua criação, através do envio da mensagem REGISTRY_AGENT juntamente com o seu identificador, o domínio (endereço do *lugar de referência*) a que pertence e o *lugar* (endereço corrente) onde se encontra. Quando recebe esta mensagem, o *Proxy* cria uma nova mensagem NEW_AGENT para enviar aos *lugares de referência* de modo que o agente seja registrado corretamente no *lugar de referência* e *lugar* correspondente. Em seguida, o *Proxy* acrescenta o agente associado ao endereço do *lugar de referência* no conjunto de agentes móveis.

A informação sobre o recebimento de uma mensagem por um agente móvel é relevante para o *Proxy* uma vez que este precisa garantir o recebimento das mensagens por todos os agentes

do grupo. Conforme descrevemos anteriormente, para cada mensagem que o *Proxy* envia, é criada uma lista contendo agentes móveis que devem receber a mensagem. Quando um agente recebe uma mensagem, este envia uma mensagem CONFIRM para o *Proxy*, que ao receber esta mensagem, remove o agente da lista de destinatários referentes àquela mensagem. Se a lista estiver vazia (todos os agentes receberam a mensagem), o *Proxy* remove a mensagem e envia uma notificação (REMOVE_MESSAGE) aos *lugares de referência* para removerem também a mensagem.

Quando um agente faz uma migração inter-domínio, o *lugar de referência* do novo domínio deve notificar o *Proxy* desta mudança. Isto é feito através de uma mensagem UPDATE passando como argumentos o identificador do agente e o novo domínio (endereço do *lugar de referência*) para onde migrou. Quando o *Proxy* recebe esta mensagem, atualiza o registro daquele agente e envia as mensagens não confirmadas até o momento pelo agente para o *lugar de referência* responsável pelo novo domínio onde se encontra o agente móvel.

A seguir, apresentamos o pseudo-código correspondente à especificação do *Proxy*.

```
Proxy {
  // variaveis
  agents = set of (AgentID,refPlaceAddr);
  refPlaces = set of (refPlaceID,refPlaceAddr);
  messages = set of (Message,receivers);
  ( receivers = set of (AgentID) );

  // recebimento de mensagem
   m = receive(any) => {
    if ( m.type = 'CONTROL' ) {        // verifica se a mensagem e' de controle
      if ( m.controlType = 'REGISTRY_AGENT' ) { // msg de registro de um agente
        m1 = createMsg('NEW_AGENT');
        m1.setArg(args, m.args);          // acrescenta os mesmos argumentos a msg
        foreach refPlaceAddr in refPlaces  // envia para os lugares de referencia
          send(refPlaceAddr, m1);
        // adiciona o agente/endereco do LR em agents
        add(agents, (m.getArg(agent), m.getArg(refPlaceAddr)));
      }
      else if ( m.controlType = 'UPDATE' ) { // msg de atualizacao de dominio
        update(agents, m.newRefPlaceAddr); // atualiza o endereco do agente em agents
        foreach msg in messages
          if (notConfirmed(msg, m.getArg(agent))) {  // verifica se a msg foi confirmada pelo agente
            addr = get(agents, m.getArg(agent));
            send(addr, msg);  // re-envia as mensagens nao confirmadas
          }
      }
      else if ( m.controlType = 'CONFIRM' ) {  // confirmacao de mensagem
        receivers = getReceivers(messages, m.getArg(confMsg));  // obtem a lista de destinatarios
        rem(receivers, m.getArg(agent));  // remove o agente da lista de destinatarios
        if ( receivers = {} ) {  // verifica se a lista de destinatarios esta vazia
          rem(messages, m.getArg(confMsg));  // remove a msg
          m1 = createMsg ( 'REMOVE_MESSAGE' );  // cria msg de notificacao para remocao da msg
          m1.setArg(msg, confMsg);
          foreach refPlaceAddr in refPlaces  // envia a msg aos LRs
            send(refPlaceAddr, m1);
        }
      }
```

```
    }
    else if ( m.type = 'APPL') {  // mensagem da aplicacao
      foreach refPlacesAddr in refPlaces {  // envia para todos os LRs
        send(refPlaceAddr, m);
        receivers = createMsgReceivers(m);
        add(messages, (m, receivers));
      }
    }
  }
}
```

3.5.3 Reference Place

O *lugar de referência* (`Reference Place`) é responsável por manter atualizadas as informações sobre a localização (endereço do *lugar*) de cada agente móvel em seu domínio, enviar mensagens provenientes do *Proxy* para os agentes e re-enviar mensagens quando um agente faz uma migração intra-domínio. Para isto, os eventos relevantes a um *lugar de referência* são: a criação de agentes (para o registro no *lugar de referência*) e a migração intra- e inter-domínio dos agentes.

A criação de um agente é notificada pelo *Proxy* através da mensagem NEW_AGENT que possui como argumentos o identificador do agente, o *lugar de referência* e o *lugar* onde está localizado o agente. Ao receber esta mensagem, o *lugar de referência* verifica se o agente foi criado em seu próprio domínio (se o endereço do *lugar de referência* é igual ao próprio endereço) e, em caso afirmativo, o agente é registrado no *lugar de referência*, caso contrário, algum outro *lugar de referência* deve registrar o agente. Este registro corresponde simplesmente à inserção do agente no conjunto de agentes do domínio. A seguir, a mensagem é passada para todos os *lugares* do domínio.

Quando um agente migra para um outro *lugar*, o *lugar* que recebeu o agente deve notificar o *lugar de referência* do domínio com uma mensagem REGISTRY passando o endereço do *lugar de referência* em que o agente estava antes da migração e o conjunto de mensagens recebidas pelo agente até então (repositório de mensagens), entre outros argumentos (vide Seção 3.6).

Quando o *lugar de referência* recebe esta mensagem, primeiramente verifica se é uma migração intra- ou inter-domínio (verificando se o agente pertence ao conjunto local de agentes). No caso de uma migração intra-domínio, o *lugar de referência* deve atualizar o endereço (*lugar*) do agente e verificar quais mensagens o agente ainda não recebeu (comparando o conteúdo do repositório de mensagens do agente com a *fila de mensagens*) e re-enviá-las.

No caso de migração inter-domínio, o *lugar de referência* deve acrescentar o agente no conjunto de agentes do domínio, enviar uma mensagem de atualização para o *Proxy* (UPDATE) e uma mensagem DEREGISTRY com o identificador do agente para que o *lugar de referência* do domínio em que o agente estava antes da migração, de modo que este de-registre o agente. Ao receber esta mensagem, este *lugar de referência* antigo deve apenas remover o agente do conjunto de agentes do domínio.

A seguir, apresentamos o pseudo-código que corresponde à especificação do *lugar de referência*.

```
ReferencePlace {
  // variaveis
  domainAgents = set of (AgentID,placeAddr);
  places = set of (placeID,placeAddr);
  msgQueue = set of (Message);
  proxyAddr;  // endereco do Proxy
  Addr;       // endereco do LR

  // recebimento de mensagem
   m = receive(any) => {
     if ( m.type = 'CONTROL' ) {
       if ( m.controlType = 'NEW_AGENT' ) {  // msg de registro de um novo agente
         // verifica se o endereco do LR em que o agente foi criado e' o proprio endereco
         if ( compareAddr(m.getArg(refPlaceAddr)) )
           // acrescenta o agente em domainAgents
           add(domainAgents, (m.getArg(agent), m.getArg(placeAddr)));
         foreach placeAddr in places  // envia a mesma msg para os lugares
           send(placeAddr, m);
       }
       else if ( m.controlType = 'REGISTRY' ) {  // registro do agente
         // verifica se o agente ja estava no dominio
         if ( contains(domainAgents, m.agent) ) {
           // a migracao e' intra-dominio, atualiza o endereco do lugar do agente
           update(domainAgents, (m.getArg(agent), m.getArg(placeAddr)));
           // identifica as msg nao recebidas pelo agente
           notRecvMsg = compareMsg(m.mRep);
           foreach msg in notRecvMsg  // re-envia as msg nao recebidas pelo agente
             send(m.getArg(placeAddr), msg);
         }
         else {  // a migracao e' inter-dominio
           m1 = createMsg('UPDATE');
           m1.setArg(refPlaceAddr, Addr);
           send(proxyAddr, m1);  // envia a msg de atualizacao de dominio ao Proxy
           m2 = createMsg('DEREGISTRY');
           m2.setArg(agent, m.agent);
           send(m.oldRefPlace, m2);  // envia a msg de de-registro ao antigo LR
           // acrescenta o agente em domainAgents
           add(domainAgents, (m.getArg(agent), m.getArg(placeAddr)));
         }
       }
       else if ( m.controlType = 'DEREGISTRY' ) {  // msg de de-registro
         rem(domainAgents, (m.getArg(agent)));  // remove o agente de domainAgents
       }
     }
     else if ( m.type = 'APPL' ) {  // a msg e' da aplicacao, envia para os lugares
       foreach placeAddr in places
         send(placeAddr, m);
     }
   }
}
```

3.5.4 Place

Para o funcionamento deste mecanismo, basta que um *lugar* (`Place`) mantenha informações sobre o conjunto de agentes que estão localizados em seu próprio endereço, ou seja, estão correntemente executando em seu endereço. Os eventos que devem ser notificados para um *lugar* são: criação de agentes e entrada ou saída de um agente no *lugar*.

O *lugar* recebe a notificação sobre a criação de agentes a partir do *lugar de referência* responsável pelo domínio em que está associado através da mensagem NEW_AGENT. O *lugar* verifica o endereço onde foi criado o agente (que é um argumento desta mensagem) e se este endereço for igual ao próprio endereço, o agente é registrado no conjunto de agentes local.

Antes da migração, um agente móvel notifica o *lugar* onde está localizado com a mensagem GOING_TO, passando seu identificador. Ao receber esta mensagem, o *lugar* remove o registro do agente correspondente do conjunto local de agentes. Ao chegar em seu destino, o agente envia uma mensagem ARRIVAL ao novo *lugar*, passando seu identificador, o domínio em que estava antes da migração (pode ser o mesmo, caso a migração seja intra-domínio) e o conjunto de mensagens recebidas até o momento. O *lugar* (que pode ser um *lugar de referência*), quando recebe esta mensagem, envia a mensagem REGISTRY para o *lugar de referência* do domínio correspondente (no caso de ser o próprio *lugar de referência*, a mensagem é enviada para ele mesmo). São passados como parâmetros o identificador do agente, o *lugar de referência* do domínio que o agente estava antes da migração, o *lugar* atual e o repositório de mensagens recebido como argumento na mensagem ARRIVAL.

Apresentamos, a seguir, o pseudo-código correspondente à especificação do *lugar*.

```
Place {
  // variaveis
  placeAgents = list of (AgentID);
  refPlaceAddr;  // endereco do LR

  // recebimento de mensagem
   m = receive(any) => {
     if ( m.type = 'CONTROL' ) {
       if ( m.controlType = 'NEW_AGENT' ) {
         // verifica o endereco do lugar onde foi criado o agente e' o proprio endereco
         if ( compareAddr(m.getArg(placeAddr)) )
           add(placeAgents, (m.getArg(agent)));  // acrescenta o agente em placeAgents
       }
       else if ( m.type = 'GOING_TO' ) {  // notificacao de saida de um agente do lugar
         rem(placeAgents, (m.getArg(agent)));  // o agente e' removido de placeAgents
       }
       // notificacao da entrada de um agente ao lugar
       else if ( m.type = 'ARRIVAL' ) {
         m1 = createMsg('REGISTRY');
         m1.setArg(args, m.args);   // acrescenta os mesmos argumentos da msg ARRIVAL
         send(refPlaceAddr, m1);  // envia da msg de registro do agente ao LR
         add(placeAgents, (m.getArg(agent)));  // acrescenta o agente em placeAgents
       }
     }
     // a msg e' da aplicacao
     else if ( m.type = 'APPL' ) {
```

```
      foreach agent in placeAgents  // envia para os agentes
        send(agent, m);
    }
  }                                                                    30
}
```

3.5.5 Agent

Os eventos que ocorrem em um agente móvel podem ser causados por um elemento externo ou podem ser determinados pelo próprio código do agente. Dentre estes, os eventos relevantes para o mecanismo são: a criação do agente, o recebimento de uma mensagem e o deslocamento do agente.

No momento de sua criação, o agente é inicializado e preparado para executar suas tarefas. Além disso, para interagir com outros agentes do grupo através do *Canal de Broadcast*, o agente deve registrar-se no *Proxy* criado especificamente para o grupo ao qual deve pertencer. No momento de sua criação, o agente recebe as informações (identificador e endereço) do *Proxy*. Para isto, o agente envia para o *Proxy* uma mensagem REGISTRY_AGENT passando como argumentos o seu identificador, o endereço do *lugar de referência* e o endereço do *lugar* onde está localizado.

Antes de uma migração, o agente envia uma mensagem GOING_TO para o *lugar* onde se encontra, passando o seu próprio identificador como argumento. Quando o agente chega no destino, envia uma mensagem ARRIVAL para "avisar" o *lugar* sobre a sua chegada, passando, seu identificador, o *lugar de referência* do antigo domínio e o repositório de mensagens como parâmetros.

No momento em que um agente recebe uma mensagem, o agente deverá confirmar o recebimento da mesma ao *Proxy* através da mensagem CONFIRM que tem como argumentos o identificador da mensagem recebida e o identificador do próprio agente.

A seguir, apresentamos o pseudo-código correspondente à especificação do agente.

```
Agent {
  // variaveis
  proxyAddr;    // endereco do Proxy
  RefPlaceAddr; // endereco do LR
  PlaceAddr;    // endereco do lugar corrente
  AgentID;      // identificador do agente
  msgRep;       // repositorio de msg

  // na criacao do agente
  onCreation => {                                                      10
    m1 = createMsg('REGISTRY_AGENT');  // cria a msg de registro
    m1.setArg(agent, AgentID);
    m1.setArg(refPlaceAddr, RefPlaceAddr);
    m1.setArg(placeAddr, PlaceAddr);
    send(proxyAddr, m1);  // envia a msg de registro ao Proxy
  }
  // na saida de um lugar
  onDispatch => {
    m1 = createMsg('GOING_TO')  // cria a msg de notificacao de saida
```

```
    m1.setArg(agent, AgentID);                                                    20
    send(placeAddr, m1);  // envio da msg de saida ao lugar
  }
  // na chegada em um novo lugar
  onArrival => {
    m1 = createMsg('ARRIVAL');  // cria a msg de notificacao de entrada
    m1.setArg(agent, AgentID);
    m1.setArg(mRep, msgRep);
    send(placeAddr, m1);  // envia a msg ao lugar
  }
  // recebimento de uma mensagem                                                 30
  m = receive(any) => {
    m1 = createMsg('CONFIRM');  // cria a msg de confirmacao
    m1.setArg(confMsg, m);
    m1.setArg(agent, AgentID);
    send(proxyAddr, m1);  // envia a msg ao Proxy
    handleMessage(m);  // trata a msg
  }
}
```

3.6 Mensagens de controle do Canal de Broadcast

Tipo de msg	Argumentos	Origem - Destino	Significado
REGISTRY_AGENT	AgentID, RefPlaceAddr,PlaceAddr	AM → Proxy	Registra um novo AM no Proxy.
NEW_AGENT	AgentID, RefPlaceAddr,PlaceAddr	Proxy → LR LR → L, L → AM	Notifica aos outros elementos o registro de um novo AM no Proxy.
GOING_TO	AgentID	AM → LR, AM → L	Notifica L ou LR onde se encontra que está migrando para outro L ou LR.
ARRIVAL	AgentID oldRefPlaceAddr oldRefPlaceID msgRep	AM → L AM → LR	Avisa que o AM está entrando para este L ou LR.
REGISTRY	AgentID oldRefPlaceID oldRefPlaceAddr newPlaceID msgRep	L → LR LR → LR	Registra o AM no LR correspondente. Usado tanto na migração intra como inter-domínio.
DEREGISTRY	AgentID	LR → LR	De-registra o AM do antigo LR.
UPDATE	AgentID newRefPlaceAddr PlaceAddr	LR → Proxy	Atualiza o registro do AM no Proxy.
CONFIRM	msg,AgentID	AM → Proxy	Confirma o recebimento de msg.
DISPOSE_AGENT	AgentID	AM → Proxy Proxy → LR LR → L	Notifica que o AM foi destruído.
REMOVE_MESSAGE	msgID	Proxy → LR	Notifica o LR que a msg ja foi confirmada por todos AM.
DISPOSE_ALL	_	Proxy → LR L → AM,LR → L	Notifica os agentes sobre o término do protocolo.
END	_	Proxy → LR LR → L	Destrói o Proxy, LR e L.

Tabela 3.1: Mensagens de Controle

Legenda:
AM: Agente Móvel, LR: Lugar de Referência, L: Lugar
AgentID: identificador do agente móvel
msgID: identificador da mensagem
(old,new)RefPlaceAddr: endereço do (antigo,novo) *lugar de referência*
(old,new)PlaceAddr: endereço do (antigo,novo) *lugar*

A Tabela 3.1 apresenta o conjunto de mensagens de controle que são trocadas entre as componentes do *Canal de Broadcast*, indicando os argumentos que as mesmas transmitem e o significado de cada uma delas.

3.7 Uma seqüência para a instalação (configuração) do Canal de Broadcast

Nesta seção apresentamos, em linhas gerais, os passos para a configuração inicial na implementação deste mecanismo do *Canal de Broadcast*. Assumimos que durante esta configuração inicial não há migração dos agentes. Maiores detalhes sobre esta configuração inicial encontra-se no Capítulo 6 (Implementação).

1. Definição e criação do conjunto inicial de endereços onde estarão executando os *lugares*.
2. Divisão do conjunto de *lugares* em domínios.
3. Para cada domínio, escolha um *lugar* para fazer o papel do *lugar de referência*. Na instanciação do *lugar de referência*, é passado como parâmetro o conjunto dos *lugares* do seu domínio.
4. Criação do *Proxy*. É passado como parâmetro o conjunto de *lugares de referência* (identificadores e endereços).
5. Envio do endereço do *Proxy* para os *lugares de referência*.
6. Envio do endereço do *Proxy* dos *lugares de referência* para todos os *lugares* dentro dos respectivos domínios.
7. Criação do conjunto inicial de agentes móveis. Cada um deles é instanciado em algum *lugar* pertencente a algum domínio pré-estabelecido no item 2. Após a criação, o agente se registra no *Proxy* (são passados como parâmetros o endereço e identificador do *Proxy* na criação do agente).
8. O Proxy aguarda o registro de todos os agentes para iniciar a execução.

Após a configuração, é assumido que o conjunto de domínios permaneça fixo, mas que o conjunto de agentes móveis poderá sofrer alterações (inserção/remoção).

3.8 Comparação do Canal de Broadcast com outros modelos de coordenação

A coordenação através do *Canal de Broadcast* apresenta diversas vantagens em relação aos outros modelos de coordenação, apresentados no Capítulo 2.

Em relação ao modelo de coordenação direta, o *Canal de Broadcast* contorna um dos maiores problemas deste modelo: a necessidade de conhecer a identidade e a localização dos agentes para que estes possam se comunicar. Apesar de ser um caso especial de coordenação direta (por estar baseado na troca de mensagens), o *Canal de Broadcast* não requer que a aplicação tenha que se preocupar com a identificação e localização dos agentes para que estes possam interagir. Em vez disto, é o próprio mecanismo, através do *Proxy* e dos *lugares de referência* que tratam de gerenciar a informação sobre a localização e garantir a comunicação confiável para grupos de agentes móveis.

A maior desvantagem da coordenação *orientada a encontros* está na necessidade da sincronização dos elementos a serem coordenados, ou seja, os agentes móveis participantes precisam estar presentes presentes em um lugar comum em um período mínimo de tempo para que possam interagir. Ao contrário, o *Canal de Broadcast* não impõe nenhuma restrição desta natureza, dando a liberdade aos agentes de migrarem e executarem suas tarefas durante a comunicação com outros agentes móveis.

A coordenação indireta (baseada em *blackboard* ou tipo *Linda*) não requer a localização ou sincronização dos agentes para efetivar a comunicação. Entretanto, requer que os agentes móveis acessem um lugar específico, onde se encontra o repositório de mensagens, para inserir uma nova mensagem ou ler uma mensagem depositada por outro agente. Um problema deste modelo é determinar, portanto, quando uma informação esperada será depositada no repositório. Isto faz com que, eventualmente, vários acessos sejam necessários.

No *Canal de Broadcast*, por sua vez, os agentes de um grupo não precisam executar operações de consulta/leitura de dados, as mensagens são entregues aos agentes assincronamente durante as suas execuções. Ao contrário do conceito de tuplas em *Linda*, no entanto, o *Canal de Broadcast* não dá suporte diretamente para o endereçamento seletivo para um subgrupo de agentes. No entanto, isto pode ser facilmente implementado através de mensagens de aplicação que podem incluir uma indicação sobre quais agentes do grupo receptor devem efetivamente processar as mensagens.

Porém, o *Canal de Broadcast* requer que os *hosts* estejam preparados para executar agentes móveis e utilizar o *Canal de Broadcast*, isto é, requer que existam elementos *lugares* ou *lugares de referência* executando nos contextos que serão visitados por agentes e isto limita a sua utilização em uma infra-estrutura pré-definida.

Capítulo 4

Exemplos de uso Canal de Broadcast

Nesta e na próxima seção, mostraremos dois exemplos de uso do *Canal de Broadcast* para dois tipos de coordenação: a exclusão mútua no acesso a agente (que representa um recurso) e uma divisão de tarefa no padrão mestre-escravos. Para cada um destes exemplos, apresentamos uma descrição sobre o algoritmo, a implementação do mesmo com o uso do *Canal de Broadcast* e os resultados obtidos da execução destas implementações.

4.1 Exemplo 1: Exclusão Mútua usando relógios lógicos

4.1.1 Descrição do algoritmo

Este algoritmo é uma adaptação do algoritmo de exclusão mútua baseado em relógios lógicos (*timestamps*), conceito proposto por L. Lamport [18]. O objetivo deste algoritmo é controlar o acesso concorrente de processos a um dado recurso. Para isto, cada processo possui um relógio local (um contador) que é incrementado toda vez que o processo faz uma requisição ou liberação do recurso e no momento em que recebe mensagens de outros processos requisitando ou liberando o mesmo recurso. Neste último caso, o relógio local é atualizado com o máximo entre o valor corrente e o *timestamp* da mensagem recebida.

A idéia central deste algoritmo é a seguinte: cada processo possui um vetor onde cada elemento deste vetor contém informações sobre o estado de um processo. Estas informações correspondem às mensagens recebidas dos processos. Podemos ter três tipos de mensagens: REQ, REL e ACK que indicam se o processo está requisitando, liberando ou concordando com a requisição do recurso de outro processo. Além disso, cada mensagem carrega consigo um *timestamp* com o valor do relógio local do momento em que foi enviada pelo processo. Assim, todo processo tem informações sobre quais são os processos requisitantes, qual deles está com o recurso e qual será o próximo que terá o direito de acesso ao recurso, caso o mesmo seja liberado. O vetor é inicializado com REL, e *timestamp* é inicialmente zero, para todo processo.

Para que um processo possa acessar o recurso, este deve incrementar o valor do seu relógio e enviar uma mensagem REQ com o *timestamp* sendo igual a este valor para todos os outros processos. Ao receber esta mensagem de requisição, os outros processos inserem esta mensagem em seus vetores locais, incrementam o relógio local e enviam uma mensagem ACK com o *timestamp*

do valor corrente concordando com a requisição do mesmo. Esta mensagem, com certeza, terá um *timestamp* maior que a mensagem de requisição (para todo processo). No caso de haver mais de uma mensagem de requisição com o mesmo *timestamp*, a mensagem de requisição referente ao processo de menor `id` terá a prioridade, ou seja, este processo de menor `id` terá direito ao acesso.

Se o recurso estiver livre e a mensagem de requisição tiver o menor *timestamp* do vetor, o processo que fez esta requisição terá acesso ao recurso. Na liberação do recurso, o processo deve incrementar o relógio local e enviar uma mensagem REL (com o *timestamp* atual) para todos os outros processos.

Na adaptação deste algoritmo para a coordenação através do *canal de broadcast*, fazem-se necessárias algumas considerações especiais. No momento da instanciação do *Proxy* é preciso passar como parâmetro o número de agentes que efetivamente participarão deste protocolo. Quando um agente é criado, é enviada uma mensagem de registro para o *Proxy* e, quando este recebe esta mensagem de todos os agentes participantes, este envia um *broadcast* com a lista de agentes para todos os agentes participantes.

Cada agente possui um vetor de mensagens `q` que possui a função descrita acima e um relógio local `ts`. Quando um agente precisa acessar o recurso, este envia uma mensagem de requisição REQ com o valor de `ts` para o *Proxy* que se encarrega de enviá-la a todos os agentes do grupo. Cada vez que recebe uma mensagem (REQ, REL ou ACK), um agente atualiza o relógio local e armazena a mensagem em `q`. Em seguida, o agente verifica se ele próprio havia feito uma requisição anteriormente e se o *timestamp* desta mensagem é o menor de todos aqueles contidos no vetor. Se este for o caso e se o recurso estiver livre, este agente poderá acessá-lo.

No momento da devolução do recurso, o agente envia uma mensagem de liberação de recurso REL para o *Proxy*, que fará com que todos os outros agentes recebam esta mensagem e atualizem os seus vetores `q`. Portanto, o algoritmo garante um concenso entre todos os agentes sobre qual agente será o próximo a ter o direito de acesso ao recurso.

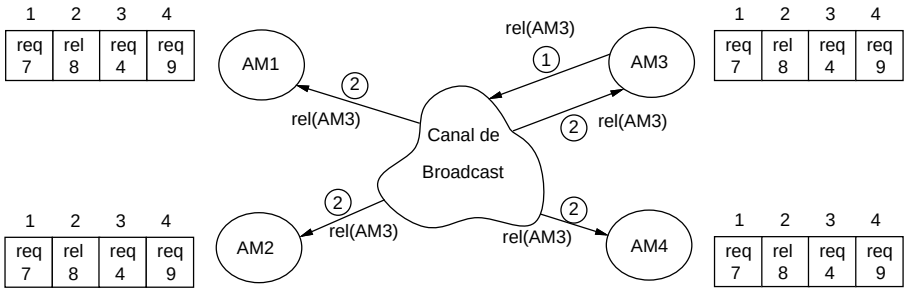

Figura 4.1: Liberação de recurso

Na Figura 4.1, temos um exemplo simplificado, ilustrando a descrição anterior. Nesta figura temos quatro agentes móveis AM_1, AM_2, AM_3, AM_4 usando o *Canal de Broadcast* com seus respectivos vetores de estados `q` em um dado momento da execução. Em ①, o agente AM_3 está liberando o recurso e enviando a mensagem REL ao *Proxy*. Em ②, esta mensagem está sendo difundida para os outros agentes, inclusive para AM_3. Ao receber esta mensagem, cada um deles atualizará o vetor inserindo a mensagem REL na posição de AM_3. Em seguida, AM_1 (que

já havia feito uma requisição) terá o direito de acessar o recurso.

Assume-se que o conjunto de agentes é fixo e que o protocolo tem início somente quando todos os agentes requisitantes estiverem registrados no *Proxy*. Assumimos ainda que os identificadores dos agentes são inteiros consecutivos e são inseridos em ordem crescente no vetor de estados `q` de cada agente.

Uma aplicação possível para a coordenação por exclusão mútua seria, por exemplo, a coleta de dados (palavras-chave, páginas WWW) em um conjunto de *hosts* (lugares) em uma rede através de um grupo de agentes móveis cooperantes. A função básica de cada membro deste grupo de agentes seria visitar um lugar especificado e coletar os dados indicados. Uma coordenação entre os agentes móveis poderia, eventualmente, ser requerida para evitar que um lugar fosse visitado por mais de um agente da mesma aplicação.

Para garantir isso, poderia ser usado um agente estacionário *recurso* (cujo endereço fosse conhecido por todos os agentes) com a função de gerenciar uma lista de endereços que contém o conjunto de lugares, cada um assinalado com um atributo "visitado" ou "não visitado". Os agentes participantes desta tarefa poderiam consultar esta lista para verificar se um dado lugar já foi visitado ou não. Caso um lugar ainda não tenha sido visitado, o agente poderia atualizar esta lista, modificando o atributo correspondente para "visitado" e então, migraria para o lugar, a fim de coletar os dados.

Deve-se notar que é necessário que o acesso à esta lista de endereços seja em regime de exclusão mútua para evitar que mais de um agente acesse a mesma ao mesmo tempo, fazendo com que mais de um agente visite o mesmo lugar. Através do algoritmo de exclusão mútua apresentado acima em conjunto com o nosso mecanismo de coordenação, esta exigência da aplicação poderia ser satisfeita.

Em particular, para acessar esta lista de endereços, um agente faria uma requisição, enviando, para isto, uma mensagem de pedido ao *Proxy*, conforme descrevemos na seção anterior. Desta forma, o acesso à lista seria exclusivo e os lugares não seriam visitados por mais de um agente móvel, uma vez que cada agente teria que aguardar o direito de acesso antes de migrar para o lugar especificado.

4.1.2 Implementação do Exemplo 1

No contexto do ASDK, cada processo corresponde a um *aglet* (estacionário ou móvel) que, em algum momento da execução, necessita acessar pelo menos uma vez o recurso. Por simplicidade, o agente implementado para este exemplo não faz nenhuma tarefa em especial, apenas faz a requisição do recurso após intervalos de tempo. No caso do recurso, trata-se simplesmente de um *aglet* estacionário cujo endereço é conhecido por todos os *aglets* requisitantes e que possui métodos que podem ser acessados por estes *aglets*. Convencionamos que para um agente acessar o recurso, este deverá migrar para o endereço onde está localizado o recurso. Para a implementação deste exemplo, seguimos os seguintes passos:

1. Definir os tipos de mensagens de aplicação.

2. Estender a classe `Agent`.

3. Estender a classe `Proxy`.
4. Definir e criar a infra-estrutura.
5. Instanciar o *Proxy* e agentes móveis a partir das classes estendidas em (2) e (3).

Passo 1: Definir os tipos de mensagens de aplicação

Inicialmente, definimos os tipos de mensagens de aplicação que precisam ser trocadas entre os agentes móveis participantes (requisitantes) para que a implementação da coordenação do tipo exclusão mútua seja alcançada. Estas mensagens são:

- REG: mensagem que é enviada pelos agentes ao *Proxy* e para registrar o agente como um eventual requisitante do recurso. Na verdade, além da mensagem de controle REGISTRY_AGENT que faz o registro do agente no *Proxy* (e, conseqüentemente, no *lugar de referência* e *lugar* - ou seja, registro no *Canal de Broadcast*), é necessário uma segunda mensagem de registro da aplicação que indica quais dos agentes serão os requisitantes e quais serão os agentes *recurso*.
- RESOURCE: mensagem enviada pelo agente que faz o papel do *recurso*. São passados como parâmetros o endereço e o identificador do agente. Esta mensagem é difundida para todos os agentes, de modo que estes consigam acessar o *recurso*.
- MEMBERS: mensagem enviada pelo *Proxy* depois que todos os agentes tenham se registrado com a mensagem REG. É passado como parâmetro um vetor com o conjunto de agentes registrados. O índice de cada elemento deste vetor corresponde ao identificador que cada agente receberá para efeito da coordenação de exclusão mútua.
- REQ: mensagem enviada por agentes para fazer a requisição ao recurso. São passados como parâmetros o valor do relógio local e o identificador do próprio agente.
- REL: mensagem enviada quando um agente está liberando o recurso. São passados como parâmetros neste caso também o valor do relógio local e o identificador do agente.
- ACK: mensagem de confirmação enviada por um agente quando este recebe a mensagem REQ de outro agente. São passados como parâmetros o valor do relógio local, o agente destinatário desta mensagem (o agente que enviou a mensagem REQ) e o identificador do agente.

As três primeiras mensagens não fazem parte do protocolo de exclusão mútua propriamente dito, mas são necessárias para preparar os agentes para executar o protocolo.

Passo 2: Estender a classe `Agent`

Para este exemplo assumimos que os agentes requisitantes são instâncias de uma mesma classe e esta classe deve ser subclasse da classe `Agent`. Os principais atributos e métodos que devem ser acrescentados para o controle deste algoritmo de exclusão mútua são:

- `id` (inteiro): identificador do agente com valores de 1 a `n`, onde `n` é o número de agentes móveis participantes. Observamos que cada um dos agentes terão dois tipos de identificadores: um, que é atribuído pelo ASDK no momento da instanciação do mesmo (`AgletID`) e outro, que é atribuído no momento de registro no *Proxy* como um elemento participante do protocolo de exclusão mútua;
- `q` (vetor): vetor de estados de tamanho `n`;
- `ts` (inteiro): valor do relógio local (*timestamp*)
- `granted` (booleano): indica se o agente está ou não com o recurso;
- `setupcomplete` (booleano): indica quando os agentes requisitantes poderão iniciar as requisições.

Os seguintes métodos foram sobrepostos:

- `onCreation()`: este método é executado no momento da instanciação do agente, e contém essencialmente: a inicialização das variáveis e a implementação do adaptador de mobilidade[1].

```
public void onCreation(Object init) {
  Vector data = (Vector)init;
  initialize(data);
  ts = 1;
  granted = false;
  setupcomplete = false;
  addMobilityListener(new MobilityAdapter() {
    public void onDispatching(MobilityEvent e) {
      // avisa o Place que está migrando para outro endereço
      sendGoingToMsg();
    }
    public void onArrival(MobilityEvent e) {
      // avisa o Place que esta entrando neste endereço
      sendArrivalMsg();
      // acessa o recurso se tiver ganhado a concessão de acesso
      if(granted)
        accessResource();
    }
  });
  if(isMobile)
    // inicializa o itinerário
    _itinerary.init(this);
}
```

- `handleApplMsg(Message)`: este método trata as mensagens específicas deste exemplo de exclusão mútua. A seguir, temos os tipos de mensagens que os agentes receberão e o tratamento específico dos agentes para cada uma delas no momento de seu recebimento:
  - RESOURCE: obtém o endereço e identificador do agente *recurso*.

[1] O adaptador de mobilidade trata-se de um objeto que tem a função de receber e tratar eventos de mobilidade de um *aglet*.

- MEMBERS: obtém o vetor com os elementos participantes, e o próprio id (índice do vetor do elemento que contém o identificador do agente (`AgletID`)).
- REQ, REL, ACK: atualiza o seu relógio local e armazena a mensagem no vetor `q` na posição correspondente ao identificador do agente remetente desta mensagem. No caso da mensagem REQ, é enviada uma mensagem ACK confirmando o recebimento desta mensagem. Em seguida, é feita a verificação se o próximo a ter o direito de acesso ao recurso é o próprio agente.

```
public void handleApplMessage(Message msg) {
  text.append("\n  ====> Tipo: "+msg.getArg("applMsgKind"));
  // a msg e' uma requisição de recurso
  if((msg.getArg("applMsgKind")).equals("REQ"))
    reqMessage(msg);
  // a msg e' uma liberação de recurso
  else if((msg.getArg("applMsgKind")).equals("REL"))
    relMessage(msg);
  // a msg e' uma confirmação a uma requisição feita pelo AM
  else if((msg.getArg("applMsgKind")).equals("ACK"))
    ackMessage(msg);
  // msg do ResourceAgent com seus dados
  else if((msg.getArg("applMsgKind")).equals("RESOURCE"))
    resourceMessage(msg);
  // msg do Proxy com o número de AM do grupo
  else if((msg.getArg("applMsgKind")).equals("MEMBERS"))
    membersMessage(msg);
  // verifica se o AM fez uma requisição e se pode acessar o recurso
  if(setupcomplete) {
    Message m = (Message)q.elementAt(id-1);
    if(((m.getArg("applMsgKind")).equals("REQ")) && min(id) && !granted)
      granted(m);
   }
   // verifica se o AM liberou o recurso para migrar para o próximo destino
   // somente migra após tratar uma msg pendente
   if(released)
     if(isMobile) {
       released = false;
       goNextDestination();
   }
 }
```

- `run()`: este método é chamado após a criação do agente e após uma migração (quando o agente chega em um novo *lugar*). Após a criação do agente, é preciso que este se registre no *Proxy* e aguarde a mensagem MEMBERS do *Proxy* com o conjunto de agentes participantes. Este procedimento, porém, é executado somente uma vez após a criação do agente e não toda vez que este método `run()` é chamado (controlado através da variável `firstTime`. Dentro deste método são feitas as chamadas para os métodos `request` e `release` a fim de pedir e devolver o recurso.

```
public void run() {
  // obtem o proxy para o Place corrente e para o Proxy
  place = getPlace();
  Proxy = getAgletContext().getAgletProxy(proxyAddr,proxyID);
```

```
    // só é executado na primeira vez apos a criação do AM
    if(firstTime) {
      waitMessage();      // espera pela msg START_AGENT do agente criador
      sendRegistryMsg();  // envia msg de registro REGISTRY_AGENT
      waitMessage();      // espera pela msg SETUPCOMPLETE do Proxy
      sendRegMsg();       // envia msg de registro da aplicação (REG)
      waitMessage();      // aguarda a msg MEMBERS
      firstTime = false;
    }
    if(!granted)  // faz o pedido do recurso
      request();
    sleep();
    if(granted)   // libera o recurso
      release();
  }
```

Além disso, os seguintes métodos foram acrescentados:

- `sendRegMsg`: envia a mensagem de registro `REG` ao *Proxy*.
- `initialize_q`: inicializa o vetor `q` de estados com mensagens `REL` e relógio local igual a zero, para todo elemento.
- `min`: devolve verdadeiro se o agente é o que possui o menor *timestamp* dentre as mensagens `REQ` em `q`.
- `request`: faz o pedido de requisição enviando a mensagem `REQ` com o *timestamp* corrente para o *Proxy*.
- `release`: libera o recurso e envia uma mensagem `REL` com o *timestamp* corrente para o *Proxy*.

Passo 3: Estender a classe `Proxy`

O único método que deve ser estendido na classe `Proxy` para esta aplicação é o método `handleApplMsg` para tratar especificamente das seguintes mensagens:

- `REG`: o *Proxy* insere a identificação do agente (`AgletID`) que enviou a mensagem na última posição de um vetor. Este vetor, posteriormente, é enviado a todos os agentes participantes.
- `RESOURCE`: esta mensagem é apenas difundida para os agentes participantes de modo que estes obtenham a referência ao agente recurso. O *Proxy* ao receber esta mensagem, verifica que um dos agentes não fará parte do grupo que compartilha o *Canal de Broadcast* (pois ao agente *recurso* não interessa as mensagens trocadas entre os agentes requisitantes) e então, decrementa o número de agentes participantes.

Passo 4: Definir e criar a infra-estrutura

São definidos os domínios, *lugares* e *lugares de referência* que deverão ser visitados pelos agentes durante a execução. Através do elemento `IfrastructureLauncher` são então instanciados cada um destes elementos conforme descrito no Capítulo 6, Seção 6.1.

Passo 5: Instanciar o *Proxy* e agentes móveis

Em seguida, o *Proxy* (classe estendida no passo 3)é instanciado em algum *lugar* (endereço). Após a definição do grupo de agentes móveis participantes e de seus itinerários, e o *sleepTime* (intervalo de tempo entre uma requisição e outra), os agentes são instanciados seqüencialmente (usando a classe estendida no passo 2) através do `AgentCreator` e enviados para o *lugar* onde deverão iniciar a execução. Neste exemplo, antes da instanciação dos agentes requisitantes, instanciamos o agente *recurso*, e isto é devido ao fato pelo qual o *Proxy* precisa saber que um dos agentes não enviará a mensagem de registro (e não fique eternamente esperando por este registro).

Resultado obtidos

A seguir, apresentamos os resultados da execução deste exemplo com os seguintes parâmetros:

- número de agentes: 4 (dois móveis e um estacionário) e um fazendo o papel do recurso;
- número de máquinas: 4;
- número de domínios: 2;
- número de *lugares*: 6;

O agente estacionário faz a requisição apenas uma vez enquanto que os outros dois fazem a requisição duas vezes com o mesmo intervalo de tempo e entre uma requisição e outra, migram para o próximo lugar previsto no itinerário.

No apêndice A, apresentamos o resultado e comentários relativos à execução deste exemplo. Com base neste exemplo, podemos verificar que através deste mecanismo, os agentes móveis foram coordenados de modo que a cada momento apenas um elemento acessou o recurso e de forma justa (o que requisitou primeiro teve acesso primeiro) e portanto, todas as mensagens foram difundidas e tratadas corretamente. A Figura 4.2 mostra o resultado obtido da interface do agente *recurso*.

4.2 Exemplo 2: Protocolo Manager-Workers

Nesta seção, apresentamos um segundo exemplo no qual usamos o *Canal de Broadcast* para uma forma diferente de coordenação: o Manager-Workers. Ao contrário do exemplo anterior, neste caso temos uma coordenação com um elemento ativador, em vez de uma coordenação orientada por demanda.

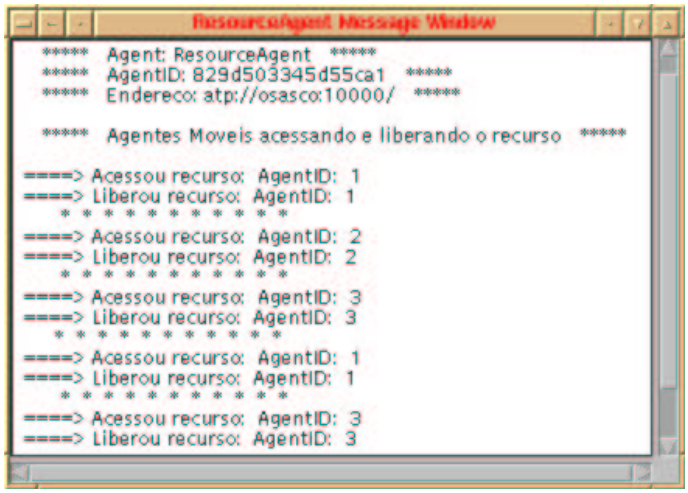

Figura 4.2: A interface do agente recurso

4.2.1 Descrição do protocolo

Dados um conjunto de tarefas e um conjunto de trabalhadores (workers), o objetivo deste protocolo é atribuir cada uma das tarefas a apenas um trabalhador. Cada trabalhador e cada tarefa possuem um identificador único e existe um coordenador (manager) que administra as tarefas. A regra é a seguinte: o trabalhador que não está executando nenhuma tarefa e possui o menor índice será o trabalhador escalado para executar a tarefa. Para isto, cada trabalhador possui informações sobre o estado de todos os outros trabalhadores do grupo. Quando uma nova tarefa é lançada pelo coordenador, estas informações são consultadas localmente e apenas um dos trabalhadores aceitará a tarefa, enviando uma mensagem para o coordenador e todos os outros trabalhadores. Ao terminar uma tarefa, um trabalhador anuncia este fato aos outros trabalhadores e ao coordenador. Para que o objetivo seja alcançado, é preciso que haja uma ordem total no recebimento das mensagens por todos os trabalhadores.

Vamos descrever este protocolo no paradigma de agentes móveis utilizando o mecanismo de coordenação proposto. O coordenador e trabalhadores são representados por agentes móveis, o coordenador possui um conjunto de tarefas e os trabalhadores possuem as funcionalidades para executá-las e enviar o resultado para o coordenador.

Cada trabalhador possui um vetor de estados q cujos elementos poderão ter valores ACC ou FREE indicando se cada um dos trabalhadores está ocupado com uma tarefa ou se está livre. Temos três tipos de mensagens: ACCEPT, FINISH e NEWTASK. No caso de ACCEPT e FINISH, o trabalhador que recebe estas mensagens deve apenas atualizar o estado do trabalhador que enviou a mensagem. No caso de NEWTASK, se o trabalhador possui o menor índice e estiver livre, então aceitará a tarefa enviando uma mensagem ACCEPT para o *Proxy* que se encarrega de enviar esta mensagem aos outros elementos, inclusive ao coordenador. No momento em que termina a tarefa, o trabalhador envia a mensagem FINISH para o *Proxy*.

Na Figura 4.3, temos uma ilustração simplificada deste protocolo. Temos o coordenador (Manager) e quatro trabalhadores (w_1, w_2, w_3, w_4), com os respectivos vetores de estados em um dado momento da execução. É suposto que todos os elementos estão registrados no *Proxy*. Os

trabalhadores w_1, w_2 e w_3 estão executando tarefas e w_4 está livre. Em ①, temos o trabalhador w_1 enviando a mensagem FINISH (notificando que terminou a sua tarefa) ao *Proxy*. Em ②, o *Proxy* está enviando esta mensagem para todos os outros trabalhadores. Ao receberem esta mensagem, os trabalhadores atualizam o vetor de estados local. A seguir, em ③, o coordenador envia uma nova tarefa ao *Proxy* fazendo com que todos tomem conhecimento da mesma. Por estar livre e ter o menor índice o trabalhador w_1 aceitará esta tarefa.

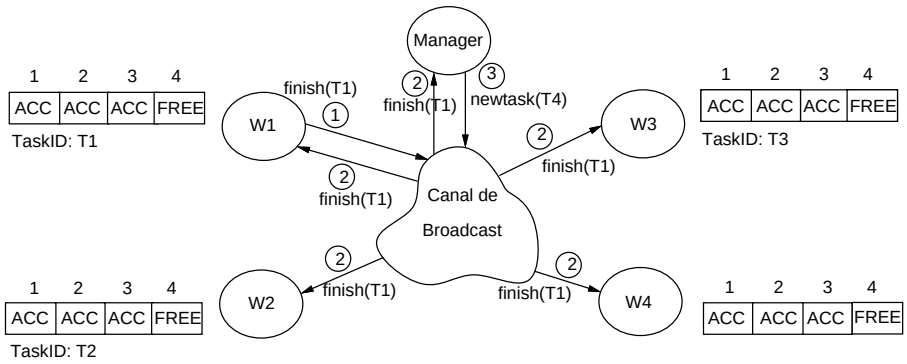

Figura 4.3: O trabalhador T1 enviando mensagem de término da tarefa

Como no exemplo anterior, é suposto que o conjunto de trabalhadores é fixo e que o protocolo tem início somente quando todos os trabalhadores estiverem registrados no *Proxy*. Assumimos que os identificadores dos trabalhadores são inteiros consecutivos e são inseridos em ordem crescente no vetor de estados de cada trabalhador.

4.2.2 Implementação do Exemplo 2

Neste segundo exemplo, cada trabalhador corresponde a um *aglet* (móvel ou estacionário) que possui as funcionalidades específicas para executar as tarefas. O coordenador corresponde a um *aglet* estacionário que possui um conjunto de tarefas para que serão enviadas arbitrariamente com um certo intervalo de tempo aos agentes trabalhadores. Por simplicidade, estas tarefas equivalem a uma mensagem (NEW_TASK) que possui um valor seqüencial inteiro que corresponde a seu identificador (`TASKID`). A seguir, apresentamos os passos para a implementação deste exemplo: (análogos ao exemplo 1)

1. Definir os tipos de mensagens.
2. Estender a classe `Agent`.
3. Estender a classe `Proxy`.
4. Definir e criar a infra-estrutura.
5. Instanciar o *Proxy* e agentes móveis a partir das classes estendidas em (2) e (3).

Passo 1: Definir os tipos de mensagens

Definimos os seguintes tipos de mensagens para este protocolo:

- REG: mensagem de registro do agente para o *Proxy* que possui a mesma finalidade da mensagem REG do exemplo 1.
- WORKERS: mensagem do *Proxy* com o conjunto de agentes trabalhadores. Possui como parâmetro um vetor que contém os agentes participantes.
- NEW_TASK: mensagem do coordenador lançando uma nova tarefa aos agentes trabalhadores. Possui como parâmetro o identificador da tarefa.
- ACCEPT: mensagem de um agente aceitando a tarefa lançada pelo coordenador. São passados como parâmetros o identificador da tarefa e o identificador do próprio agente.
- FINISHED: mensagem de um agente notificando o término de uma tarefa. São passados como parâmetros o identificador da tarefa e o identificador do agente.

Passo 2: Estender a classe `Agent`

Todos os agentes trabalhadores devem ser capazes de executar as mesmas tarefas e portanto, os agentes derivam de uma mesma classe (subclasse da classe `Agent`). O agente coordenador também deve ser derivado de uma classe que estende a classe `Agent`, uma vez que precisa interagir com os agentes trabalhadores para enviar tarefas através do *Canal de Broadcast* e receber mensagens que indicam qual trabalhador está executando ou terminou de executar a tarefa.

Os seguintes atributos e métodos foram definidos:

- `id` (inteiro): identificador do trabalhador com valores entre 1 e `n`, (onde `n` é o número de agentes participantes);
- `q` (vetor): contém o estado de cada trabalhador (`ACC` ou `FREE`);
- `newtask` (vetor): contém as novas tarefas que foram armazenadas enquanto todos os trabalhadores estavam ocupados;
- `setupcomplete` (booleano): indica quando a instanciação está terminada e o protocolo inicia efetivamente.

Os seguintes métodos foram sobrepostos:

- `onCreation()`: da mesma forma que no exemplo anterior, neste método as variáveis são inicializadas e o adaptador de mobilidade é implementado.
- `handleApplMsg(Message)`: trata as seguintes mensagens específicas desta aplicação que os trabalhadores receberão durante a execução:

- WORKERS: obtém o número de trabalhadores participantes e o próprio identificador a partir do vetor recebido como parâmetro. Inicializa o vetor `q` com `FREE` para todo elemento.
- NEW_TASK: verifica em `q` se o próprio agente é o trabalhador livre que possui o menor `id` e, neste caso, aceita a tarefa, enviando uma mensagem ACCEPT ao *Proxy*. Caso não haja nenhum trabalhador livre, esta mensagem (NEW_TASK) é inserida no vetor `newtask` para ser tratada posteriormente, quando algum agente declarar o término de uma tarefa.
- ACCEPT: modifica o vetor `q` na posição correspondente ao `id` do trabalhador que aceitou a tarefa com `ACC` indicando que este trabalhador está ocupado.
- FINISHED: modifica o vetor `q` na posição correpondente inserindo `FREE` indicando que o trabalhador está livre.
- `run()`: na primeira vez que este método é invocado, assim como no exemplo 1, o agente deve enviar a mensagem de registro ao *Proxy* e aguardar a mensagem WORKERS para iniciar a execução.

Acrescentamos os seguintes métodos:

- `sendRegMsg`: envia a mensagem de registro do agente (trabalhador) ao *Proxy*.
- `initialize_q`: inicializa o vetor `q` com `FREE` para todo elemento.
- `min`: devolve o trabalhador livre que possui o menor `id`.
- `doTask`: "executa" a tarefa e envia a mensagem FINISHED ao *Proxy*.

Passo 3: Estender a classe `Proxy`

O único método que deve ser estendido na classe `Proxy` para esta aplicação é o método `handleApplMsg` para tratar especificamente da seguinte mensagem:

- REG: o *Proxy* trata esta mensagem de forma análoga ao exemplo anterior: insere o identificador do agente (`AgletID`) que enviou a mensagem na última posição de um vetor, e este vetor posteriormente é enviado a todos os agentes participantes. O `id` de cada agente dependerá da ordem de registro no *Proxy*.

Passo 4: Definir e criar a infra-estrutura

Análogo ao passo 4 da implementação do exemplo 1.

Passo 5: Instanciar o *Proxy* e agentes móveis

Análogo ao passo 5 da implementação do exemplo 1.

Resultado obtidos

Neste exemplo 2, vamos utilizar os mesmos parâmetros do exemplo 1 em relação a número de agentes, *lugares* e domínios. Portanto, neste exemplo, teremos 3 agentes trabalhadores e um agente que faz o papel de coordenador. No apêndice A, apresentamos o resultado e comentários da execução deste exemplo. Verificamos que através deste mecanismo, os objetivos do protocolo descrito acima foram alcançados, devido às facilidades que este mecanismo de coordenação oferece.

Capítulo 5

Aglets Software Development Kit

O Aglets Software Development Kit (ASDK) [2] é um ambiente para programação de agentes móveis baseado na linguagem Java e desenvolvido pela IBM Tokyo Reasearch Laboratory.

O ASDK introduz o conceito de *aglet* (agent+applet) que é um programa em execução que pode mover-se de uma máquina para outra em uma rede. O *aglet* estende o modelo de código móvel de *applets* com a diferença que quando migra de uma máquina para outra, o *aglet* carrega consigo também o seu estado. Assim, um *aglet* executando em uma máquina pode suspender a sua execução em um dado momento, ser transferido para outra máquina e retomar a sua execução a partir do ponto em que estava antes da migração.

Um *aglet* é um agente móvel possuindo autonomia e possibilidade de definir dinamicamente o seu itinerário. Como um objeto tradicional, um *aglet* tem estado, comportamento e identidade. Diferente de um objeto, um *aglet* possui também *localização*.

5.1 A escolha da ferramenta

A escolha do ASDK para a implementação do *Canal de Broadcast* se deve aos seguintes fatos:

- está completamente implementado em Java, oferendo diversas vantagens para a programação de agentes móveis: programação *multithread* (permite implementar o comportamento autônomo do agente móvel), independência de plataforma (permite a criação de agentes móveis sem ter o conhecimento de onde serão executados), serializaçõ de objetos (permite a transferência dos agentes), a execução segura (protege o *host* contra agentes maliciosos, impedindo seu acesso).
- oferece formas de interação entre agentes móveis através da troca de mensagens síncronas e assíncronas. Em nosso mecanismo de coordenação, os agentes móveis interagem através da troca de mensagens assíncronas.
- possibilita o tratamento de eventos como criação, destruição e migração de agentes móveis, favorecendo a implementação de ações dos elementos deste mecanismo na ocorrência destes eventos.

- possibilita o tratamento de mensagens de acordo com o tipo definido pelo programador. Possibilita a distinção entre tipos de mensagens de controle e da aplicação e a implementação do tratamento específico a cada uma delas.
- permite a suspensão da *thread* de execução do agente por um certo intervalo de tempo e a retomada da execução. Permite a sincronização dos agentes móveis durante a criação da infra-estrutura do *Canal de Broadcast*.
- permite a invocação do método `handleMessage` que trata as mensagens do agente. Este método normalmente é invocado pelo gerenciador de mensagens (message manager). Esta invocação por parte do agente permite o tratamento posterior de uma mensagem, possibilitando a ordenação de mensagens.

5.2 Conceitos básicos

Os principais conceitos do ASDK são:

- *aglet*: é um objeto Java, autônomo pois possui a sua própria *thread* de execução e é capaz de migrar de uma máquina para outra na rede.
- *contexto*: é o lugar de execução de *aglet*. Trata-se de um objeto estacionário que gerencia os *aglets* em execução. Um nó na uma rede pode ter vários contextos.
- *proxy*: é o representante de um *aglet* e é utilizado para a interação com outros agentes. O *proxy* protege os métodos públicos de um *aglet* do acesso direto por outros *aglets* e provê transparência de localização (esconde a localização verdadeira do *aglet*).
- *messagem*: é um objeto de comunicação entre *aglets*. A troca de mensagens pode ser síncrona ou assíncrona.
- *itinerário*: é o plano de movimentação do *aglet*.
- *identificador*: é a identificação de um *aglet*. Este identicador é globalmente único e é imutável durante a vida do *aglet*.

O ciclo de vida de um *aglet* é definido por um conjunto de eventos que causam respectivas ações e modificam o seu estado [16]. Os possíveis eventos são: criação (`creation`), clonagem (`clone`), transferência para outro contexto (`dispatch`), volta para um contexto (`retract`), desativação (`desactivation`), ativação (`activation`) e destruição (`dispose`).

5.3 Aglet API

O Aglet API [15, 20] define as classes e interfaces que implementam as funcionalidades fundamentais para os agentes móveis. A seguir, apresentaremos uma breve descrição sobre essas classes.

5.3.1 Classe Aglet

Esta é a principal classe do Aglet API. É uma classe abstrata que deve ser usada como base para criar agentes customizados. Esta classe define métodos que são essenciais para controlar a vida de um *aglet* e também métodos que podem ser sobrepostos a fim de implementar o comportamento de um *aglet*.

Alguns métodos importantes desta classe são apresentados a sequir:

- `dispatch(URL)`: na invocação deste método, o *aglet* é enviado ao endereço especificado como parâmetro. Esta URL deve especificar os nomes do *host* e domínio no contexto destino e o protocolo usado na transferência do *aglet* através da rede. Por exemplo:

  ```
  dispatch(``atp://rebutosa:11000'');
  ```

- `dispose()`: termina a execução do *aglet*.
- `onCreation()`: este método pode ser visto como o construtor de um *aglet*. Pode ser sobreposto para implementar alguma inicialização específica do *aglet*. É invocado apenas uma vez na vida do *aglet*.
- `run()`: este método é invocado pelo sistema após a inicilaização do *aglet* e toda vez que o *aglet* chega em um novo lugar. Pode ser sobreposto a fim de implementar o comportamento do *aglet*.
- `onDisposing()`: este método é invocado logo após o método `dispose()` e é utilizado para implementar as tarefas antes de sua destruição (por exemplo, a liberação de recursos alocados).

5.3.2 Interface AgletProxy

Esta interface faz o papel do representante de um *aglet*, protegendo-o contra acessos indevidos em seus métodos públicos e garantindo a sua segurança contra *aglet* maliciosos.

Para que um *aglet* possa se comunicar com outro, é necessário, inicialmente, obter o *proxy* para o *aglet* e interagir através desta interface. Um *proxy* pode ser obtido através da invocação do método `getAgletProxy(URL,AgletID)` da classe *AgletContext*, passando como parâmetro o identificador e o endereço do *aglet*. Se o *aglet* estiver localmente no mesmo contexto, é necessário apenas o identificador do mesmo para a obtenção *proxy*.

5.3.3 Interface AgletContext

Interface que permite que o *aglet* interaja com o ambiente de execução. Uma referência para este objeto é obtida através de `getAgletContext()` que é um método da classe `Aglet`. Esta referência pode ser usada para se obter informações locais (endereço do contexto), para a criação de um novo *aglet*, para a obtenção do *proxy* para outros *aglets*.

5.4 Criação e destruição de um aglet

A criação de um *aglet* pode ser feita através da invocação do método `createAglet` da classe `AgletContext`. São passados como parâmetros um endereço (URL) onde se encontra o código, o nome da classe do agente e um objeto (opcional) que constitui os argumentos para a inicialização do *aglet*. Este método retorna um objeto *proxy* para o *aglet* recém-instanciado. Um exemplo da criação de um *aglet* da classe `ReferencePlace` com um vetor de elementos `placesList` como argumento:

```
AgletProxy refplace = getAgletContext().createAglet(getCodeBase(),"ReferencePlace",placesList);
```

Uma vez que um *aglet* possui sua própria *thread* de execução, o mesmo só é liberado quando o método `dispose()` é invocado explicitamente pelo usuário ou pelo próprio *aglet*.

5.5 Interação entre aglets

Aglets podem interagir através da invocação de métodos ou troca de mensagens. Em qualquer um dos casos, antes que um *aglet* possa se comunicar com outro, é preciso que o primeiro obtenha um *proxy* para o *aglet* com qual irá se comunicar. No primeiro caso, o *proxy*, quando invocado, consulta o gerenciador de segurança para que este verifique se o *aglet* requisitante tem o direito de executar algum de seus métodos públicos. Porém, este procedimento só é possível executar localmente: o Aglets não permite a invocação remota dos métodos de outro *aglet*.

Através do *proxy* para um *aglet* A, um *aglet* B pode invocar métodos deste *proxy* para criar um clone do *aglet* A, transferir o *aglet* A para um outro contexto, desativar e até mesmo destruir o *aglet* A. O *proxy* passa a requisição para o *aglet* A e este pode decidir se aceita ou não a requisição do *aglet* B.

A segunda forma de interação é através da troca de mensagens que além de possibilitar a troca de mensagens entre *aglet* localizados no mesmo contexto, possibilita também o envio de mensagens a *aglets* remotos desde que o *aglet* remetente da mensagem possua um *proxy* válido para o *aglet* com qual deseja comunicar-se. Um *proxy* perde a sua validade quando o *aglet* migra de um contexto para outro. Esta é uma outra restrição em relação à forma de comunicação do Aglets, impossibilitanto a comunicação entre *aglet* caso não se tenha sempre atualizado o endereço dos mesmos.

5.6 Mensagens em Aglets

Os *aglets* se comunicam através de objetos da classe `Message`. Cada um destes objetos possui um conjunto de carcteres que indicam o tipo da mensagem e através do método `setArg` é possível associar argumentos à mensagem.Abaixo, temos um exemplo da criação de uma mensagem:

```
Message msg = new Message(``REGISTRY'');
msg.setArg(``AgentID'',ID);
msg.setArg(``Addr'',Addr);
```

Uma mensagem pode ser enviada a um *aglet* através de seu *proxy*, seguindo-se os passos apresentados a seguir:

```
context = getAgletContext();    // obtenção da referência para o contexto
destAgent = context.getAgletProxy(URL,AgletID);  // obtenção do proxy para
                                  // aglet destinatário da mensagem
// formas de envio de mensagens
destAgent.sendMessage(Message);       // bloqueia a execução do aglet

destAgent.sendAssycMessage(Message);  // não bloqueia a execução do aglet

destAgent.sendOnewayMessage(Message); // não recebe nenhuma confirmação de
                                 // recebimento pelo destinatário
```

As mensagens que um *aglet* recebe são tratadas através do método `handleMessage(Message)` herdado da classe `Aglet` e que deve ser sobreposto a fim de tratar as mensagens de acordo com os tipos definidos pelo usuário. Este método é invocado pelo gerenciador de mensagens (message manager) que recebe mensagens, armazena em uma fila e passa seqüencialmente (uma a uma) de acordo com a ordem de chegada para ser tratada por `handleMessage`. O `handleMessage` retorna um valor booleano indicando se a mensagem foi tratada ou não pelo *aglet*. A seguir, temos um exemplo do método *handleMessage*:

```
public boolean handleMessage(Message msg) {
  if(msg.sameKind("CONTROL")) {
    // msg do AM para ser registrado no Proxy
    if((msg.getArg("controlMsgKind")).equals("REGISTRY_AGENT")) {
      registryAgent(msg);
    }
    // msg do AM confirmando o recebimento de uma mensagem
    else if((msg.getArg("controlMsgKind")).equals("CONFIRM")) {
      confirm(msg);
    }
    else if((msg.getArg("controlMsgKind")).equals("UPDATE")) {
      update(msg);
    }
    // msg do AM quando este e' destruido
    else if((msg.getArg("controlMsgKind")).equals("DISPOSE_AGENT")) {
      removeAgent(msg);
    }
    . . .
    return true;
  }
  // a msg e' da aplicacao
  else if(msg.sameKind("APPL")) {
    handleApplMessage(msg);
    return true;
  }
  else return false;
}
```

5.7 Mobilidade

Um *aglet* pode migrar para uma máquina remota por decisão própria (invocando o seu método `dispatch(URL)`, passando como parâmetro a URL da máquina destino) ou por meio da requisição feita por um outro *aglet*. Neste caso, o *aglet* que faz a requisição deverá primeiramente obter o objeto *proxy* do *aglet* a ser transferido e invocar o método `dispatch(URL)` neste *proxy*.

Para um *aglet* ser transferido de uma máquina para outra, é preciso transformá-lo em uma seqüência de *bytes* e, quando chegar ao destino, reconstruir o *aglet* a partir desta seqüência de *bytes*. Este processo é chamado de *serialização*.

Um fato importante é que nem tudo que faz parte do estado pode ser serializado. Por uma limitação da Java VM, não é possível serializar as pilhas de execução e os contadores de programas das *threads* gerenciadas pelo *aglet*. Assim, a serialização de objetos se limita apenas aos dados no *heap*. Portanto, quando um *aglet* é transferido, clonado ou desativado, todo estado definido pelas pilhas assim como o contador de programas serão perdidos. Este problema pode ser contornado através do modelo *callback* que apresentamos na próxima seção.

5.8 Modelo callback

Antes de da ocorrência efetiva de um evento na vida de um *aglet*, é chamado um método prévio que tem a função de fazer o *aglet* se preparar para o evento, ou em certos casos, de recusar o evento. Por exemplo, antes de um *aglet* ser transferido para um novo contexto, o método `onDispatching()` do *aglet* é invocado. O método `onDipatching()` contém o código para que o *aglet* possa completar seu trabalho e se preparar para a serialização, caso aceite ser transferido ou, caso contrário, gerar uma exceção. Quando retornar do método `onDispatching()`, se o *aglet* tiver aceito a tranferência, seu estado será serializado, todas as *threads* terminadas e será invocado o método `dispatch()` que faz o *aglet* ser transferido para outro contexto. Ao chegar ao destino, o método `onArrival()` é então invocado e o estado do *aglet* é reconstruído e inicializado, para prosseguir a sua execução.

A classe Aglet define cinco métodos *callback* que podem ser sobreescritos para customizar o comportamento de um *aglet*: `onCloning()`, `onDispatching()`, `onReverting()`, `onDeactivating()` e `onDisposing()`. Além destes, existem quatro métodos de inicialização que o servidor de *aglet* pode invocar cada vez que um *aglet* começa a sua execução em um *host*: `onCreation()`, `onClone()`, `onArrival()` e `onActivation()`.

Capítulo 6

Implementação

6.1 Principais classes

6.1.1 Descrição das classes

Definimos dois conjuntos de classes para a implementação do *Canal de Broadcast*: o primeiro, corresponde às classes que implementam os elementos responsáveis pela criação e configuração da infra-estrutura deste mecanismo (`InfrastructureLauncher`, `AgentCreator`) e às classes que correspondem às interfaces gráficas para a interação com o usuário (`LauncherInterface`, `AgentCreatorInterface`). O segundo conjunto de classes corresponde às classes que efetivamente implementam os elementos que compõem este mecanismo (`Proxy`, `Reference Place`, `Place` e `Agent`) e a interface usuário do *Proxy* (`ProxyInterface`).

A seguir, descrevemos de forma sucinta, cada uma dessas classes e suas principais funções (nas subseções seguintes temos uma explicação mais detalhada sobre cada uma delas):

- `InfrastructureLauncher`: estende a classe `Aglet` (Aglet API do ASDK). Uma instância desta classe representa um *aglet* estacionário que tem a função de criar os elementos (*lugares* e *lugares de referência*) em endereços específicos e assim compor a infra-estrutura deste mecanismo (definição dos domínios e a instanciação do *Canal de Broadcast* - elemento *Proxy*) a partir dos dados fornecidos pelo usuário.

- `LauncherInterface`: interface gráfica para `InfrastructureLauncher` através da qual o usuário fornece os dados para a criação de domínios, *lugares* e *lugares de referência*.

- `AgentCreator`: estende a classe `Aglet`, é usada para a criação de agentes móveis que participarão do grupo e compartilharão o *Proxy* instanciado.

- `AgentCreatorInterface`: interface gráfica para `AgentCreator` através da qual o usuário fornece os dados sobre cada agente móvel que deseja criar.

- `Proxy`: estende a classe `Aglet`, possui as funções do *Proxy* especificadas no mecanismo de coordenação (vide capítulo 3). Esta classe pode ser estendida, se necessário, para tratar mensagens específicas da aplicação.

- `ProxyInterface`: interface através da qual o usuário envia mensagens aos agentes móveis do grupo.
- `Place`: estende a classe `Aglet`, possui a funcionalidades de um *lugar* (local de execução de agentes móveis, conforme descrito no capítulo 3).
- `ReferencePlace`: estende a classe `Place` e além de ser um *lugar*, possui as funções do representante do domínio (*lugar de referência*).
- `Agent`: estende a classe `Aglet` e esta é a classe que deve ser estendida para que os agentes móveis tenham capacidade de interagir entre si através do *Canal de Broadcast* e acrescentam outras funcionalidades acrescentadas a critério do programador de acordo com as necessidades específicas da aplicação.

6.2 Criação da infra-estrutura do mecanismo

A infra-estrutura deste mecanismo é construída por um agente estacionário, instância da classe `InfrastructureLauncher` (vamos chamar este agente simplesmente de `launcher`). O `launcher` tem a finalidade de instanciar cada um dos elementos (*lugares*, *lugares de referência* e *Proxy*) em determinados endereços da rede (*host*:número_da_porta) onde estes irão executar. Daqui para a frente, chamaremos de endereço o par *nome_do_host*:*número_da_porta* que identifica unicamente o local na rede onde um *aglet* vai executar. No ASDK, este endereço é fornecido com a seguinte sintaxe: `atp://jaca:9000`. O `launcher` possui uma interface gráfica (`LauncherInterface`) que recebe do usuário os dados correspondentes a cada domínio (endereço do *lugar de referência*, nome do domínio e lista de endereços de *lugares* que pertencerão a este domínio). Na Figura 6.1 temos um exemplo de instanciação de um domínio através desta interface.

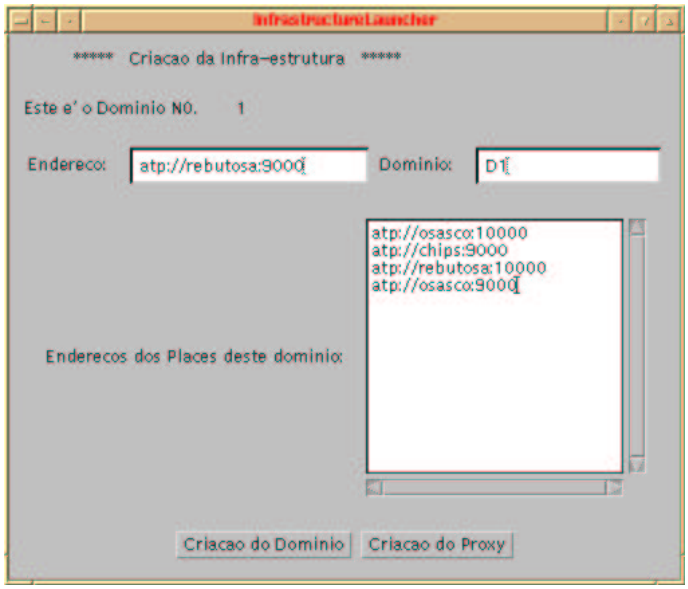

Figura 6.1: Criação da infra-estrutura

Quando o botão `Criação do Domínio` é acionado, inicialmente, o `launcher` obtém a lista de endereços (de *lugares*) e para cada elemento desta lista, instancia um *lugar* (objeto da classe `Place`) e envia o mesmo para o endereço correspondente. No término da instanciação dos *lugares*, o `launcher` cria o *lugar de referência* (classe `ReferencePlace`) e envia para o endereço especificado pelo usuário.

Cada um destes elementos notificam o `launcher` no momento em que chegam em seus respectivos endereços. Esta notificação é apresentada pelo `launcher` em uma janela de informações de maneira que o usuário possa se certificar que os elementos foram instanciados corretamente. Na Figura 6.2, temos um exemplo desta janela de informações para um domínio instanciado.

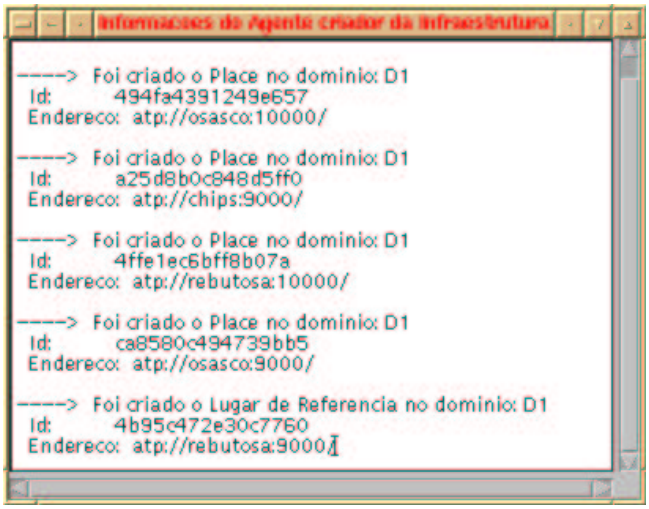

Figura 6.2: Elementos instanciados

Após este passo inicial, a infra-estrutura está criada e é fixa, não sendo mais permitida a inserção ou remoção de *lugares* ou *lugares de referência*.

O próximo passo é a criação do *Proxy* e para isto, o `launcher` requisita ao usuário o nome da classe do *Proxy* e o endereço onde o *Proxy* deverá estar executando. A Figura 6.3 mostra a interface para a criação do *Proxy*.

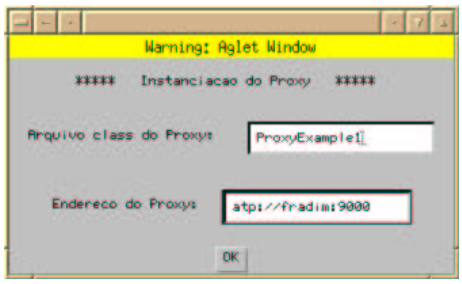

Figura 6.3: Criação do Proxy

6.3 Criação de agentes móveis

Após a criação da infra-estrutura do mecanismo, o `launcher` instancia um objeto da classe `AgentCreator` (vamos chamá-lo de `creator`) que é um agente estacionário que tem a finalidade de instanciar os agentes móveis que farão parte da aplicação distribuída. O `creator` possui uma interface gráfica (`AgentCreatorInterface`) através da qual o usuário insere os dados de cada agente móvel: a nome da classe do agente a ser usada, um endereço onde o agente deverá iniciar a sua execução (dentre a lista de endereços que correspondem aos endereços de *lugares* e *lugares de referência* instaciados), um conjunto de endereços que corresponde ao itinerário do agente (opcional) e um valor em milisegundos que corresponde ao tempo de permanência do agente em um determinado *lugar* (opcional). Dependendo da aplicação, este intervalo de tempo poderá ser ser utilizado ou não em função da tarefa específica do agente no *lugar*. Na Figura 6.4 apresentamos a interface do `creator`.

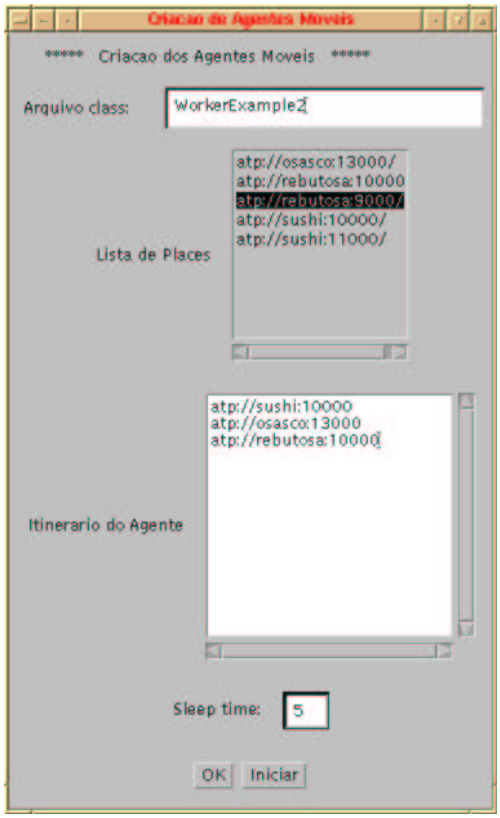

Figura 6.4: Interface para criação de Agentes Móveis

6.4 Início do Protocolo

O protocolo inicia-se efetivamente quando todos os elementos (inclusive os agentes móveis) já tiverem sido instanciados nos seus devidos endereços. O `creator` aguarda pela mensagem

AGENT_ARRIVAL de todos os agentes instanciados (certificando-se que a instanciação foi efetivada corretamente).

Como no ASDK, cada agente equivale a uma *thread* de execução, em princípio, após a criação, um agente iria começar a sua execução, podendo assim fazer com que antes mesmo da criação de todos os agentes necessários para a execução de uma tarefa, o primeiro agente poderá ter terminado sua execução.

Para evitar este problema, é necessário que o agente aguarde até o momento apropriado para iniciar a sua execução, o que é feito através do método `waitMessage()` da própria classe `Aglet`, que suspende a execução do agente até receber uma notificação.

Portanto, após a sua criação (invocação do método `onCreation`), o agente é enviado por `creator` para seu endereço inicial e ao alcançar o seu destino, o agente envia a mensagem AGENT_ARRIVAL e aguarda uma notificação até que todos os agentes sejam instanciados antes de se registrar no *Proxy*.

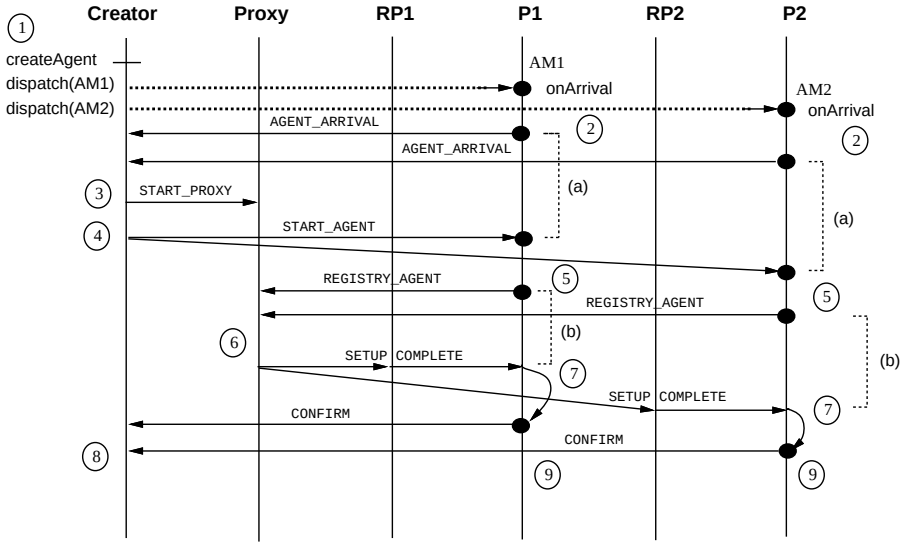

Figura 6.5: Inicialização do Protocolo

A mensagem START_PROXY é enviada pelo `creator` ao *Proxy* no momento em que todos os agentes móveis confirmaram a sua chegada em seus respectivos destinos e é passado como parâmetro o número de agentes instanciados. Em seguida, a mensagem START_AGENT é enviada diretamente para cada um dos agentes pelo `creator`, indicando que todos os agentes foram criados.

O *Proxy* precisa ter o controle do número de agentes que serão registrados e isto só é possível depois que o programador instanciar todos os agentes que desejar. O *Proxy* ao receber a mensagem START_PROXY toma conhecimento do número de agentes que deverão se registrar e aguarda pela mensagem de registro de cada um deles. Quando o registro de todos os agentes se completar, o *Proxy* envia a mensagem SETUPCOMPLETE à todos os agentes. Esta mensagem é importante para notificar os agentes o momento em que todos os outros já foram registrados e indica o início do protocolo.

A mensagem START_AGENT é a notificação que o agente aguardava para enviar a mensagem de registro REGISTRY_AGENT ao *Proxy*. Após o envio desta mensagem de registro, o agente deve aguardar novamente até que todos os agentes se registrem no *Proxy* (até o momento do recebimento da mensagem SETUPCOMPLETE).

O *Proxy* quando recebe a mensagem REGISTRY_AGENT, cria uma nova mensagem NEW_AGENT e envia aos *lugares de referência* notificando o registro deste novo agente. Em seguida, o *lugar de referência* repassa esta mensagem aos *lugares*. Os *lugares* e *lugares de referência* ao receberem esta mensagem, verificam se o agente em questão está localizado no domínio (no caso do *lugar de referência*) ou no próprio endereço (em ambos casos) para que o identificador do mesmo seja armazenado no conjunto local de agentes móveis. Esta verificação é possível pois um dos argumentos da mensagem é o endereço corrente do agente móvel.

Na Figura 6.5, apresentamos este procedimento descrito acima. Em ①, o agente `creator` está instanciando agentes móveis (AM_1 e AM_2) e os rementendo para os *lugares* correspondentes (`dispatch`). Em ②, há o envio de confirmação de chegada ao `creator` (AGENT_ARRIVAL). Quando o `creator` recebe esta mensagem dos agentes instanciados, envia a mensagem START_PROXY para o *Proxy* ③, e START_AGENT para os agentes em ④. Observamos que no intervalo (a) os agentes não executam nenhuma ação, aguardando a mensagem START_AGENT para se registrarem no *Proxy* ⑤. O *Proxy* também fica suspenso desde a sua criação atá receber a mensagem START_PROXY ③. Quando o *Proxy* receber as mensagens de registro de todos os agentes, envia a mensagem SETUPCOMPLETE ⑥ indicando o início do protocolo. Esta mensagem é passada para os *lugares de referência*, depois para *lugares* e finalmente aos agentes ⑦. Novamente, os agentes ficam suspensos (b) até receberem esta mensagem. Finalmente, os agentes confirmam o recebimento desta mensagem ao *Proxy* e iniciam a execução de suas tarefas ⑨.

Após este procedimento de instanciação da infra-estrutura, todos os elementos terão obtido as informações necessárias sobre a localização de cada agente móvel e o *Canal de Broadcast* poderá começar a ser utilizado.

6.5 Classe InfrastructureLauncher

Esta classe estende a classe `Aglet` (do ASDK) e tem o propósito de criar e configurar a infra-estrutura para o *Canal de Broadcast*. Esta configuração é feita por domínios, cada domínio é estruturado seqüencialmente. Na Figura 6.6, apresentamos os principais atributos e métodos das classes `InfrastructureLauncher` e `LauncherInterface`.

Os dados de cada domínio recebidos através da interface gráfica (`LauncherInterface`) são passados ao `launcher` (instância de `InfrastructureLauncher`) na invocação do método `createRefPlace` (passando-se como parâmetro um vetor com estes dados referentes ao domínio). No método `createRefPlace`, inicialmente, é obtido o endereço onde se ficará o *lugar de referência* e para cada endereço no restante do vetor, é criado um *lugar* (instância da classe `Place`). Logo após a criação, o *lugar* é enviado para o endereço correspondente onde deverá estar executando e o seu identificador (`AgletID`) e endereço são armazenados na tabela `places` (que será passada como parâmetro na criação do *lugar de referência*).

Após a instanciação de todos os *lugares*, o *lugar de referência* é instanciado (classe `ReferencePlace`),

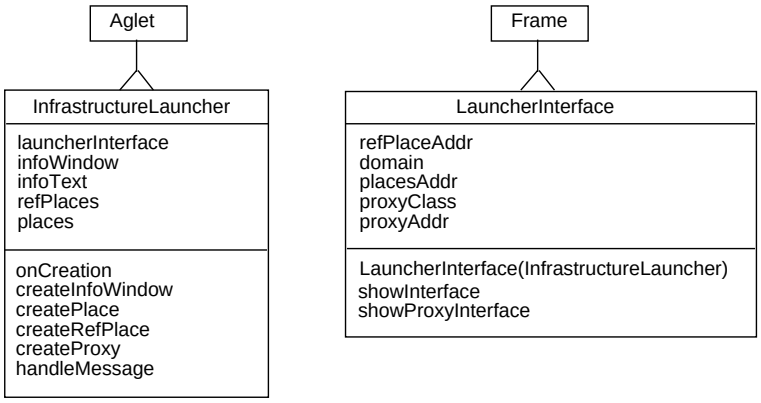

Figura 6.6: Classe IntrastructureLauncher e LauncherInterface

passando-se como parâmetro o conjunto de *lugares* instanciados (tabela `places`). Desta forma, cada *lugar de referência* responsável por um domínio terá informações dos *lugares* pertencentes a seu domínio. Da mesma maneira que *lugares*, após a sua criação, o *lugar de referência* é enviado para o endereço onde deverá estar executando e o seu identificador e endereço são inseridos na tabela `refPlaces` (que posteriormente será enviada como parâmetro na criação do *Proxy*).

Quando *lugares* e *lugares de referência* chegam em seu destino, enviam, respectivamente, mensagens PLACE_ARRIVAL e REFPLACE_ARRIVAL, respectivamente, ao `launcher` confirmando que chegaram em seus destinos corremente. Ao receber esta mensagem, o `launcher` notifica o usuário sobre a criação destes elementos em `infoWindow`.

A seguir, o *Proxy* é instanciado através do método `createProxy` passando-se como parâmetro a tabela `refPlaces` com os *lugares de referência* instanciados. O *Proxy* também é enviado para o endereço onde deverá estar executando e na chegada a este endereço, notifica o `launcher` através de PROXY_ARRIVAL. Em seguida, o `launcher` instancia o agente responsável pela criação de agentes da aplicação (`AgentCreator`).

6.6 Classe AgentCreator

Na instanciação do agente `creator` são passados como parâmetros o conjunto de endereços de *lugares* e *lugares de referência* que foram instanciados pelo `launcher`, além do identificador e endereço do *Proxy*. O conjunto de *lugares* e *lugares de referência* são apresentados em uma lista na interface gráfica a fim de que o usuário selecione o endereço onde o agente a ser instanciado deverá iniciar a sua execução. O identificador e endereço do *Proxy* são passados como parâmetros na criação de cada agente. Também são passados como parâmetros o itinerário do agente especificado pelo usuário e o intervalo de tempo (*sleepTime*) que deve permanecer em cada *lugar*. Quando não é fornecido o itinerário, o agente será instanciado como um agente estacionário do *lugar* destino. Na Figura 6.7 apresentamos os principais atributos e métodos definidos das classes `AgentCreator` e `AgentCreatorInterface`.

O `creator` instancia cada agente móvel através de `createAgent`, passando como parâmetros o identificador e o endereço do *Proxy*. Esta informação é importante para a diferenciação do

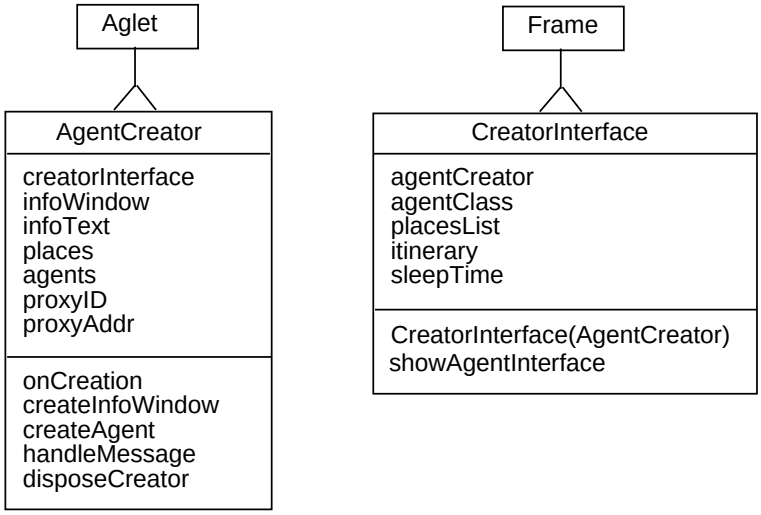

Figura 6.7: Classe AgentCreator e CreatorInterface

Proxy que coordena o grupo ao qual o agente pertence.

Cada agente deve ser uma instância da classe `Agent` ou de uma subclasse da mesma, uma vez que a classe `Agent` possui as funcionalidades necessárias para o funcionamento do protocolo.

De forma análoga aos outros elementos que compõem este mecanismo, após a sua criação o agente é transferido para o endereço especificado onde deverá iniciar a execução. Quando chega em seu destino, o agente envia uma notificação ao `creator` indicando que chegou no endereço corretamente (mensagem AGENT_ARRIVAL) e esta mensagem é tratada no método `handleMessage` do `creator`. Após terminar a instanciação de todos os agentes e receber a confirmação (AGENT_ARRIVAL) de todos eles, o `creator` notifica o *Proxy* indicando o número de agentes instanciados e os agentes indicando o início da execução (momento em que os agentes vão se registrar no *Proxy*). O usuário poderá verificar as informações dos agentes instanciados na janela de informações do agente `creator` (`infoWindow`). A Figura 6.2 mostra um exemplo desta janela de informações.

6.7 Classe Proxy

O *Proxy* possui uma interface (`ProxyInterface`) que é utilizada pelo usuário para o envio de mensagens aos agentes móveis membros de um grupo coordenado por um determinado *Proxy*. A Figura 6.8 apresenta os principais atributos e métodos da classe `Proxy` e `ProxyInterface`.

A classe `Proxy` possui os métodos e atributos para as funções de administração de agentes e domínios, controle de envio e re-envio de mensagens, assim como a atualização de endereços após a migração de agentes. Além disso, o *Proxy* possui também uma interface onde são mostradas todas as mensagens recebidas de agentes móveis (`msgWindow`).

Através da tabela `agents` é possível obter, a qualquer momento, o domínio no qual se encontra um agente (endereço do *lugar de referência*). Isto é importante, principalmente, para o re-envio

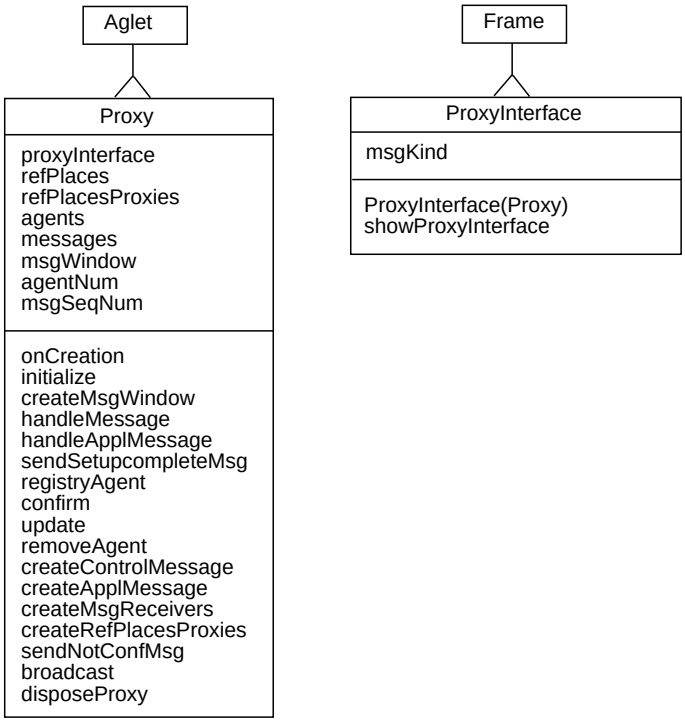

Figura 6.8: Classe Proxy e ProxyInterface

de mensagens, pois, através do endereço do *lugar de referência* é possível obter o identificador do mesmo consultando a tabela `refPlaces` e então obter o *proxy* para o *lugar de referência*.

Um *proxy* no ASDK equivale a um representante (referência) de um *aglet*, é um objeto através do qual os *aglets* poderão interagir entre si (trocar mensagens).

O vetor `refPlacesProxies` contém as referências de todos os *lugares de referência* definidos. Estas referências são obtidas no início da execução do *Proxy* (método `createRefPlacesProxies`) e devido ao fato em que *lugares de referência* são *aglets* estacionários, esta referência será sempre válida enquanto os *lugares de referência* existirem.

As mensagens que devem ser difundidas pelo *Proxy* são mensagens de controle (por exemplo, SETUPCOMPLETE, NEW_AGENT e DISPOSE_AGENT) e as mensagens da aplicação que podem ser requisitadas por agentes e pelo próprio usuário (neste último caso, através da interface `ProxyInterface`). As mensagens podem ser criadas através de um dos métodos `createControlMessage` ou `createApplMessage` dependendo se a mensagem é de controle ou da aplicação.

Através do método `sendBroadcastMsg(Message)` o *Proxy* incorpora à mensagem alguns argumentos necessários para o controle de difusão de mensagens. São estes: o identificador da mensagem (número de seqüência) e um argumento "Bcast" indicando aos *lugares* e *lugares de referência* que a mensagem é para ser difundida. Além disso, é criado um conjunto de agentes receptores da mensagem. Em seguida, a mensagem é difundida através do método `broadcast(Message)`. Este método simplesmente envia a mensagem para cada elemento em `refPlacesProxies`.

Toda mensagem difundida pelo *Proxy* recebe um identificador único (número de seqüência da mensagem) de acordo com a variável `msgSeqNum` que, na sua essência, é um contador. Além disso, as mensagens são armazenadas temporariamente em uma tabela `messages` para um eventual envio posterior. O acesso a esta tabela é feito através do número de seqüência da mensagem e cada um deles está associado a um vetor com o identificador dos agentes destinatários da mensagem (método `createMessageReceivers`). Na confirmação de uma mensagem por um agente, o identificador do agente é removido desse vetor de destinatários. No momento em que esse vetor estiver vazio, a mensagem é removida da tabela e é enviada uma mensagem REMOVE_MESSAGE para que os *lugares de referência* removam também esta mensagem de seus respectivos conjuntos locais de mensagens.

O re-envio de uma mensagem é necessário quando o *Proxy* recebe uma mensagem de controle UPDATE e existe pelo menos uma mensagem não confirmada pelo agente que fez a migração. Isto é feito através do método `sendNotConfMsg` que recebe como parâmetros o identificador do agente e o endereço do *lugar de referência* do novo domínio (recebidos como argumentos da mensagem UPDATE). Este método verifica todos os conjuntos de destinatários das mensagens na tabela `messages`. Ao encontrar o identificador do agente em um destes conjuntos, o *Proxy* obtém a mensagem e altera alguns argumentos: substitui "Bcast" por "Ucast" (indicando que a mensagem será direcionada especificamente a um agente) e acrescenta o identificador do agente destinatário. O número de seqüência da mensagem não é modificado.

O tratamento das mensagens recebidas é feito através do método `handleMessage` que distingue entre mensagens de controle e da aplicação. No caso das mensagens de controle, estas são tratadas de acordo com a especificação do protocolo (uma descrição informal deste protocolo pode ser vista no capítulo 3. As mensagens da aplicação são apenas difundidas, não há tratamento específico, mas o programador poderá estender esta classe e sobrepor o método `handleApplMsg`, acrescentando um tratamento específico para cada mensagem da aplicação.

6.8 Classe Reference Place

A classe `ReferencePlace` é uma subclasse de `Place` e portanto, além de se comportar como um *lugar*, um objeto desta classe possui funcionalidades para executar funções de administração do domínio. A Figura 6.9 mostra os atributos e métodos da classe `ReferencePlace`.

Da mesma forma que a classe `Proxy`, a classe `ReferencePlace` possui também uma tabela de agentes. Esta tabela representa o conjunto de agentes móveis que estão atualmente localizados no domínio (`domainAgents`). O acesso a esta tabela é feito através do identificador do agente, e que está associado ao endereço do lugar onde está localizado. Isto facilita o re-envio de mensagens, pois a localização corrente do agente pode ser identificada imediatamente e a mensagem ser direcionada para este endereço.

Novos agentes são inseridos no conjunto de agentes (`domainAgents`) quando a mensagem NEW_AGENT é recebida e o agente pertence ao domínio (ou seja, foi instanciado para, inicialmente, executar neste domínio) ou quando o *lugar de referência* recebe a mensagem REGISTRY. Quando um agente está saindo do domínio, o *lugar de referência* receberá a mensagem DEREGISTRY que será tratada pelo método `deregistry` (que remove o agente da tabela `domainAgents`). Um agente também é removido do domínio quando a mensagem DISPOSE_AGENT é recebida e é

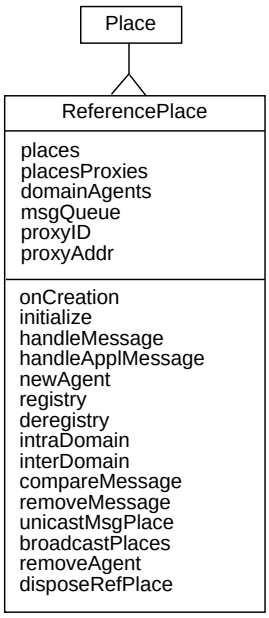

Figura 6.9: Classe ReferencePlace

tratada pelo método `removeAgent`. As mensagens enviadas para os agentes dentro do domínio são armazenadas em `msgQueue`. Uma mensagem é removida de `msgQueue` no momento em que o *lugar de referência* recebe a mensagem REMOVE_MESSAGE do *Proxy*.

Quando o método `registry` é invocado para tratar da mensagem REGISTRY, diferencia-se entre uma migração de agente é diferenciada entre intra e inter-domínio, quando ou o método `intraDomain` ou `interDomain` é invocado. No caso de migração intra-domínio, o método `compareMessage` é chamado para que as mensagens ainda não recebidas pelo agente sejam identificadas e re-enviadas ao mesmo (uma vez que o conjunto de mensagens recebidas pelo agente é passado como argumento da mensagem REGISTRY). Este re-envio é feito através de `unicastMsgPlace`, isto é, a mensagem é direcionada para o *lugar* específico onde está localizado o agente. Se o agente está localizado no próprio *lugar de referência*, o método `unicastMessage` (método da classe `Place`) é invocado. No caso de uma difusão de mensagem, além de `broadcastPlaces`, o método `broadcastAgents` (também da classe `Place`) é invocado para enviar aos agentes localizados no próprio *lugar de referência*.

6.9 Classe Place

A Figura 6.10 apresenta os principais métodos e atributos da classe `Place`. Esta classe mantém um vetor contendo identificadores dos agentes localizados correntemente no endereço (`placeAgents`). Para cada envio de mensagem aos agentes presentes no *lugar*, é obtida a referência a cada agente (em `placeAgents`) uma vez que o conjunto local de agentes pode estar modificando constantemente, dependendo da movimentação dos agentes. Ou seja, a saída/entrada de um agente pode ocorrer antes mesmo de o *lugar* receber uma notificação (mensagem ARRIVAL do próprio agente).

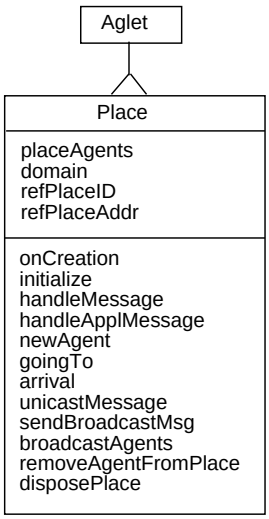

Figura 6.10: Classe Place

O envio de mensagens é feito através do método `broadcastAgents`. No caso de re-envio, as mensagens são direcionadas para um agente específico através do método `unicastMessage`.

A entrada de novos agentes no *lugar* pode ocorrer no momento da criação de um novo agente (método `newAgent`), quando o *lugar* recebe a mensagem NEW_AGENT ou quando o agente envia a mensagem ARRIVAL (método `arrival`). Após o teste de equivalência do endereço do *lugar* com aquele fornecido na mensagem, o identificador do agente é armazenado em `placeAgents`.

O *lugar* possui também informações sobre o *lugar de referência* (identificador e endereço) que são eventualmente consultados pelos agentes móveis antes de migrações.

A saída de um agente de um *lugar* para outro é notificada através da mensagem GOING_TO que é tratada pelo método `goingTo`. Neste momento, o agente é removido de `placeAgents`. O mesmo procedimento ocorre no caso de um agente ser destruído (através da notificação DISPOSE_AGENT).

6.10 Classe Agent

A classe `Agent` possui atributos e métodos que possibilita a um agente móvel instanciado a partir desta classe possa receber mensagens (com ordenação total) através do *Canal de Broadcast*. A Figura 6.11 mostra a classe `Agent` em maiores detalhes.

O agente possui um itinerário (`itinerary`) que é um vetor de endereços onde cada endereço representa um *lugar* que será visitado pelo agente durante a sua execução.

No entanto, um agente pode também ser definido como estacionário, e neste caso, seu itinerário é vazio. Um agente é caracterizado como móvel ou estacionário através da variável booleana `isMobile`.

O `sleepTime` é uma variável que representa o tempo que o agente permanece em um *lugar* ou o intervalo de tempo que o agente executa uma tarefa. O agente se comunica com o *Proxy*

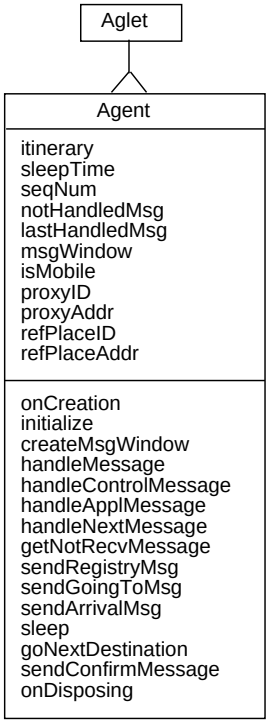

Figura 6.11: Classe Agent

obtendo-se a referência para o mesmo através dos atributos `proxyID` e `proxyAddr`. A obtenção da referência para o *Proxy* é feita toda vez que o agente chega em um novo *lugar*. Isto porque, no ASDK o objeto *proxy* (a referência) perde a sua validade na transferência do agente. As variáveis `refPlaceID` e `refPlaceAddr` são obtidas antes de cada migração (em `onDispatching`) e passadas como argumentos da mensagem `ARRIVAL` no momento em que um agente chega no novo lugar a fim de que o *lugar de referência* possa identificar se a migração é intra ou inter domínio.

O agente possui três atributos importantes para garantir a ordenação de mensagens. Estes atributos são: `seqNum`, `lastHandledMessage` e `notHandledMessage`. A variável `seqNum` é um contador através do qual o agente se baseia para tratar as mensagns. Cada mensagem possui um identificador (argumento `msgSeqNum` da mensagem) e quando a mensagem recebida não possui o próximo valor esperado, a mensagem é armazenada no vetor `notHandledMessage`, para ser tratada posteriormente.

A classe `Agent` possui quatro métodos para o tratamento de mensagens: `handleMessage` (que sobrepõe o método correspondente da classe `Aglet`); `handleApplMessage`, que trata mensagens do tipo `APPL`; `handleControlMessage` que trata as mensagens de controle (tipo `CONTROL`)e `handleNextMessage` para tratar as mensagens que foram armazenadas em `notHandledMessage`.

No momento da migração, o método `onDispatching` é invocado e é enviada uma mensagem para o *lugar* notificando esta migração através de `sendGointToMsg`. Na chegada ao destino, o método `arrival` é invocado e a mensagem de entrada no *lugar* (ARRIVAL) é criada e é passado como parâmetro a última mensagem tratada (`lastHandledMessage`) e o conjunto de mensagens

recebidas e não tratadas (conteúdo de `notHandledMessage`). O *lugar de referência* compara este conjunto de mensagens com o conteúdo de `msgQueue`, descarta aquelas que possuem o número de seqüência menor ou igual ao `lastHandledMessage` e re-envia toda mensagem que não estão no conjunto recebido, utilizando uma mensagem do tipo "Ucast".

Capítulo 7

Testes

Neste capítulo, apresentamos os testes realizados com o mecanismo de coordenação proposto. O objetivo destes testes foi, por um lado, submeter a implementação a diferentes padrões de mobilidade dos agentes e taxas de requisição de mensagens, para testar a robustez. Por outro lado, identificar a ordem de grandeza da sobrecarga de tempo causada pelo *Canal de Broadcast* em relação ao tempo de envio de mensagens a um grupo de agentes. Este "tempo de envio de mensagens" corresponde à diferença entre o instante de envio de uma mensagem aos agentes e o instante em que a última confirmação referente àquela mensagem é recebida pelo *Proxy*.

Ressaltamos que estes testes não tiveram como objetivo avaliar completa e exaustivamente o desempenho do *Canal de Broadcast* através de testes exaustivos.

O tempo (em milissegundos) obtidos na execução destes testes estavam sujeitos a diversos fatores como, por exemplo, às características particulares das máquinas usadas nos testes e à carga média das máquinas no momento da execução dos testes (devido a outros processos de usuários). No entanto, o intuito foi fazer os testes em condições de carga similares.

Os testes foram executados em quatro máquinas com as seguintes características: uma, com freqüência do *clock* de 167 MHz e 2304MB de memória; outra, com 60MHz de freqüencia do *clock* e 704MB de memória e duas máquinas com 110MHz de freqüencia e 65MB de memória.

Durante os testes, identificamos a existência de uma particularidade no ASDK, que cria uma sobrecarga no tempo médio do envio de mensagens conforme aumentamos o número de mensagens enviadas. Esta sobrecarga pode estar relacionada a uma coleta de lixo deficiente, isto é, como e quando o ASDK elimina mensagens que foram tratadas pelos agentes. Uma possibilidade é que o ASDK acumule mensagens durante o tempo de vida do agente, causando uma degradação no desempenho no tratamento de mensagens pelo agente e, conseqüentemente, no desempenho dos sistema como um todo. Este problema foi identificado quando implementamos uma difusão de mensagens utilizando somente o ASDK, onde observamos a variação de tempo no envio de 10 a 150 mensagens a um mesmo número de agentes.

Em consideração a este fato, limitamos o número de mensagens (20 mensagens) enviadas aos agentes em cada rodada de teste. Para cada teste, foram realizadas três rodadas (de 20 mensagens) em horários diferentes, a fim de compensar diferenças devido a uma eventual carga devido a outros processos no sistema.

Realizamos, essencialmente, duas classes de testes:

- testes estáticos: neste caso, são considerados grupos de agentes estacionários. O tempo de envio de mensagens a estes agentes através do *Canal de Broadcast* é comparado ao tempo gasto na difusão de mensagens (e recebimento de confirmações) ao mesmo número de agentes através de um programa usando somente o ambiente ASDK (isto é, um agente central que é o responsável por enviar mensagens diretamente aos agentes).
- testes dinâmicos: neste caso, são considerados grupos de agentes móveis que possuem um itinerário fixo e um tempo de permanência que é comum aos agentes. Esta tempo de permanência corresponde ao tempo em que o agente fica em um determinado lugar e, durante este intervalo de tempo, o agente está passivo e simplesmente responde às mensagens que recebe. O tempo médio de envio de mensagens aos agentes é analisado em relação à dois tempos de permanência (alto e baixo).

7.1 Testes estáticos

7.1.1 Descrição

Estes testes foram realizados com grupos de agentes estacionários, distribuídos de maneira uniforme em uma infra-estrutura pré-definida do *Canal de Broadcast*. Foi medido o tempo desde o momento em que a mensagem foi difundida pelo *Proxy* até o momento em que todos os agentes do grupo confirmaram o seu recebimento. Este tempo foi medido através de chamadas ao método `currentTimeMillis()` da classe `System` do Java no momento do envio da mensagem pelo *Proxy*.

Para a comparação do tempo de envio de mensagens com o *Canal de Broadcast*, foi implementado um programa simples no ASDK para envio de mensagens a `n` agentes (*aglets*). Este *envio direto* é constituído por um agente central (`CentralAgent`) estacionário que instancia um número de agentes (`AckAgent`) em diferentes endereços e mantém a referência (o *proxy*, na terminologia do ASDK) para cada um deles. Através desta referência, o agente central envia mensagens diretamente aos agentes. Cada um destes agentes, por sua vez, possui uma referência para o agente central (que é recebida no momento de sua criação) e sua única função é confirmar toda mensagem recebida ao agente central. Uma vez que não há intermediários, é esperado que o tempo médio deste *envio direto* de mensagens seja bem baixo do que o tempo do envio através do *Canal de Broadcast*.

A Figura 7.1 ilustra a diferença no número de mensagens usadas no *envio direto* e no *Canal de Broadcast* para 3 agentes. No diagrama (A) da Figura 7.1 temos a ilustração do *envio direto* de mensagens. Os retângulos representam um endereço (lugar) onde o agente central e os agentes receptores de mensagens estão executando. Neste diagrama, em ①, o agente central envia uma mensagem diretamente aos agentes através da referência que mantém para cada agente móvel. Em ②, os agentes confirmam o recebimento desta mensagem ao agente central. No diagrama (B), temos a ilustração do envio de mensagem através do *Canal de Broadcast*. Conforme descrevemos anteriormente, toda mensagem é enviada aos agentes de forma indireta, passando pelos *lugares de referência* e *lugares*. Em ①, temos o *Proxy* enviando a mensagem aos *lugares de referência*, em ②, esta mensagem é enviada do *lugar de referência* aos *lugares* e em ③, esta mensagem é

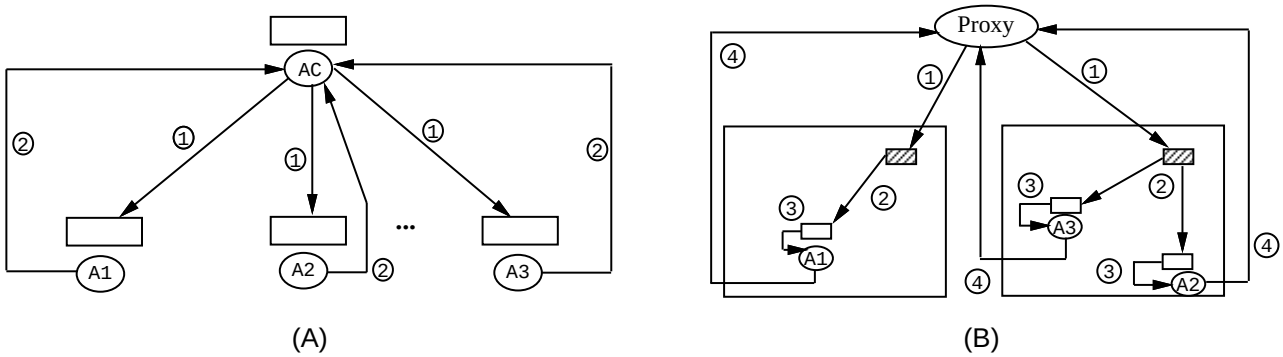

Figura 7.1: Mensagens no *envio direto* (A) e no *Canal de Broadcast* (B)

passada aos agentes. Em ④, temos a confirmação dos agentes ao *Proxy*.

Enquanto no *envio direto* são necessárias apenas duas mensagens por agente para completar o procedimento, no caso do *Canal de Broadcast* são necessárias o dobro de mensagens (quatro mensagens) para cada agente estacionário.

7.1.2 Objetivo

O objetivo destes testes foi comparar (para diversos parâmetros), qual é o fator de sobrecarga do *Canal de Broadcast* em relação ao *envio direto* de mensagens (usando somente o ASDK).

7.1.3 Parâmetros dos testes

As Tabelas 7.1 e 7.2 indicam os valores dos parâmetros considerados na execução dos testes do *Canal de Broadcast* vs *envio direto*, respectivamente. Para um dado número de máquinas, *lugares* (endereços) e domínios (no caso do *Canal de Broadcast*), os testes foram feitos variando-se o número de agentes. No caso do *Canal de Broadcast*, a distribuição dos *lugares* em domínios foi feita de maneira uniforme (número aproximadamente igual de *lugares* por domínio). Também a distribuição de agentes móveis em *lugares* foi feita de maneira uniforme (em ambos os casos). Consideramos também que o *Proxy* (no caso do *Canal de Broadcast*) e o agente central (no caso do *envio direto*), executaram em máquinas distintas dos *lugares*.

	$N^{\underline{o}}$ máquinas	$N^{\underline{o}}$ *lugares*	$N^{\underline{o}}$ domínios	$N^{\underline{o}}$ agentes
A	2	6	1	3,6,12,24,36,48
B	4	6	3	3,6,12,24,36,48
C	4	9	3	3,6,12,24,36,48

Tabela 7.1: Parâmetros considerados nos testes do *Canal de Broadcast*

Para cada linha das tabelas e cada valor do número de agentes, foram feitas três rodadas de teste. Uma rodada de teste consiste no envio de 20 mensagens a um grupo de agentes. Cada

	$N^{\underline{o}}$ máquinas	$N^{\underline{o}}$ lugares (endereços)	$N^{\underline{o}}$ agentes
D	2	6	3,6,12,24,36,48
E	4	6	3,6,12,24,36,48
F	4	9	3,6,12,24,36,48

Tabela 7.2: Parâmetros considerados nos testes do *envio direto*

mensagem requer um certo tempo desde o momento do envio até que seja confirmada por todos os agentes. Em cada rodada, é obtida a média aritmética dos tempos do envio das 20 mensagens. Cada ponto no gráfico corresponde à média dos tempos obtidos da execução de 3 rodadas de teste.

7.1.4 Resultados

A Figura 7.2 ilustra o resultado da execução dos testes considerando-se como parâmetros os valores contidos nas linhas A e D das Tabelas 7.1 e 7.2, respectivamente. Através do gráfico, observamos que o tempo médio do *envio direto* é melhor que o tempo médio do *Canal de Broadcast* (como era de se esperar) mas em ambos os casos, este desempenho degrada rapidamente com o aumento do número de agentes (devido ao problema do ASDK que mencionamos acima). Devido ao número maior de mensagens, o *Canal de Broadcast* apresentou um custo maior e isto é devido também, ao problema do ASDK mencionado acima. Este fato pode ser observado também nos demais gráficos deste capítulo.

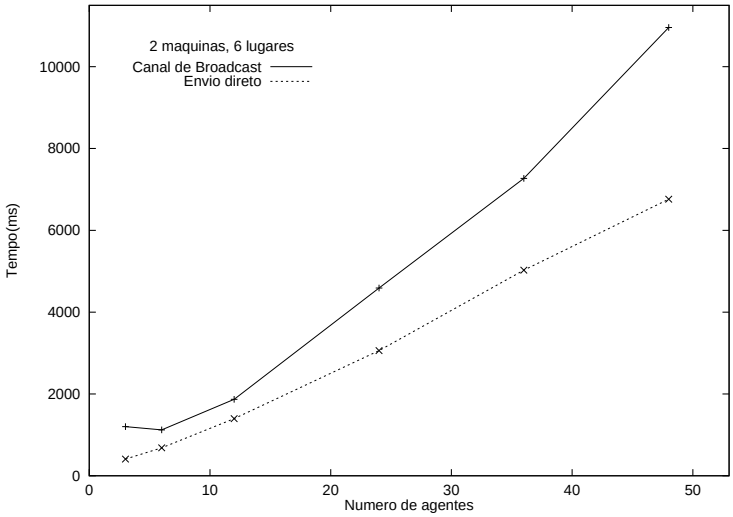

Figura 7.2: Resultados dos testes para 2 máquinas e 6 lugares

A Figura 7.3 representa o resultado da execução dos testes para os valores das linhas B e E das tabelas de parâmetros. Como podemos observar, a curva do *Canal de Broadcast* abaixou consideravelmente em relação ao gráfico anterior. A razão deste fato pode estar relacionada à

comunicação por rede (mais requisitada no teste das 4 máquinas) que afetou mais o desempenho no caso do *envio direto* do que no *Canal de Broadcast*.

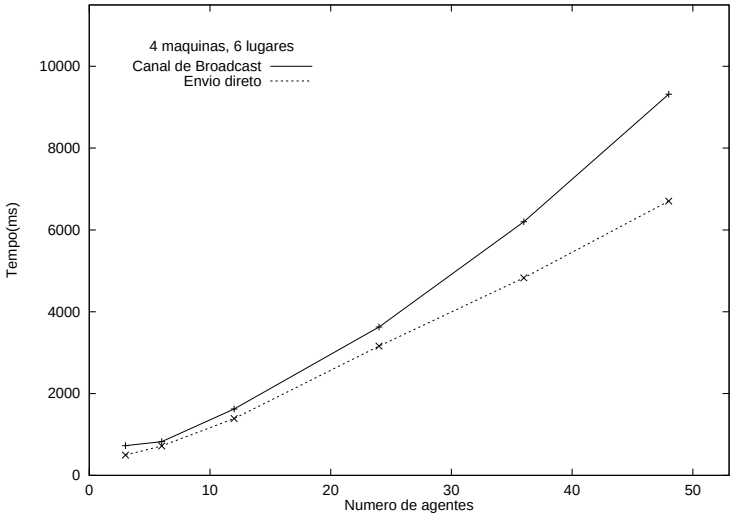

Figura 7.3: Resultados dos testes para 4 máquinas e 6 lugares

Este fato pode ser observado com mais clareza no gráfico da Figura 7.6 que mostra uma comparação das três curvas dos resultados das execuções dos testes no *Canal de Broadcast*.

No gráfico da Figura 7.4 podemos observar que com o aumento do número de *lugares* (de 6 para 9), a distância entre as curvas diminuiu ainda mais em relação aos gráficos anteriores.

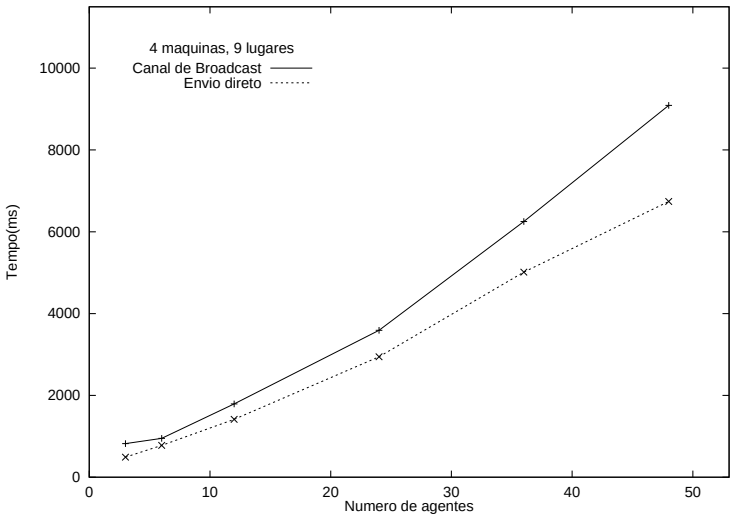

Figura 7.4: Resultados dos testes para 4 máquinas e 9 lugares

As curvas relativas ao *envio direto* praticamente não tiveram variação nestes três casos (conforme podemos observar no gráfico da Figura 7.5) mas, em relação ao *Canal de Broadcast*, há uma variação considerável em relação ao aumento do número de máquinas e *lugares* (Figura 7.6),

fazendo-nos supor que quanto maior o número de máquinas e *lugares*, menor é a sobrecarga causada pelo *Canal de Broadcast* devido à descentralização do processamento.

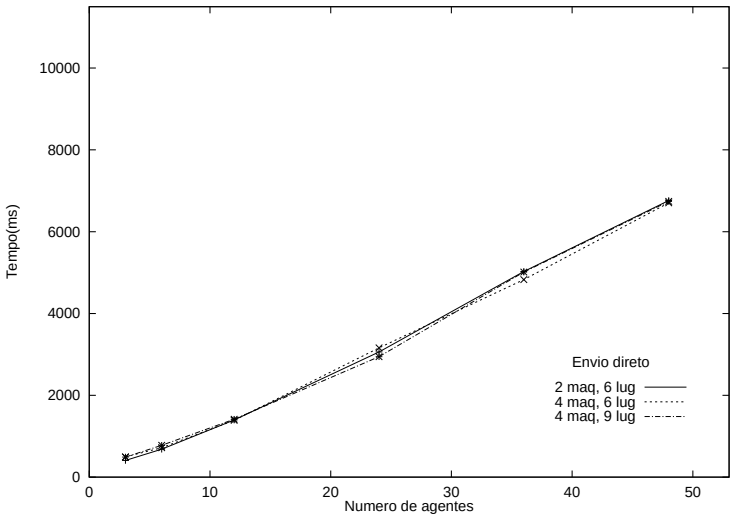

Figura 7.5: Comaparação: *Envio Direto*

Outro resultado interessante que podemos observar no gráfico da Figura 7.6, é a semelhança entre as curvas para os casos de 6 e 9 lugares. Isto indica que o segundo passo na difusão de mensagens através do *Canal de Broadcast* (*lugar de referência* $\rightarrow$ *lugares*) tende a ter uma influência menor na sobrecarga do *Canal de Broadcast* à medida que o processamento se torna mais distribuído (maior número de máquinas) e isto também é um ponto positivo em relação ao nosso mecanismo.

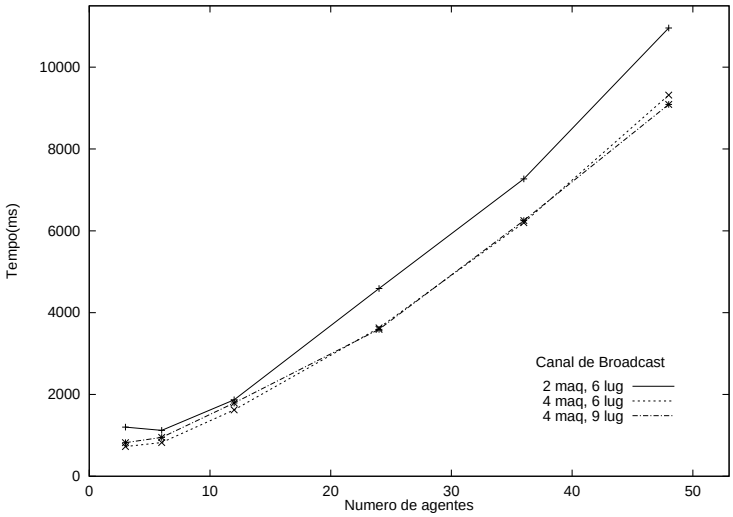

Figura 7.6: Comparação: Canal de Broadcast

Apresentamos no gráfico da Figura 7.7 as curvas referentes aos fatores de sobrecarga do *Canal de Broadcast* em relação ao *envio direto* (cada ponto no gráfico representa o valor da relação *Canal de Broadcast/envio direto*). Através do gráfico, verificamos que para pequenos grupos de agentes móveis, o custo do *Canal de Broadcast* chega a ser três vezes maior que o custo do *envio direto*. Mas, por outro lado, para 4 máquinas, no caso de 6 e 9 *lugares*, este valor não passa de dois, indicando que o custo do *Canal de Broadcast* não chega a ser o dobro em relação ao *envio direto* quando há uma distribuição maior dos agentes em *lugares*.

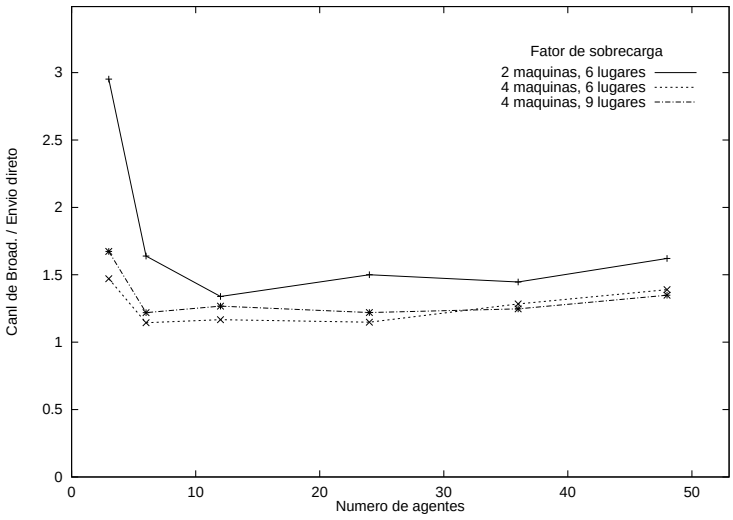

Figura 7.7: Fator de sobrecarga Canal Broad./Envio direto

7.2 Testes dinâmicos

7.2.1 Descrição

Ao contrário aos testes estáticos em que todos os agentes eram estacionários e portanto, permaneciam todo o tempo fixos em um mesmo *lugar*, no caso dos testes dinâmicos, consideramos apenas agentes que possuem um itinerário de *lugares* (visitados seqüencialmente), permanecem um certo intervalo de tempo em cada um destes *lugares* e que durante estes períodos confirmam todas mensagens recebidas.

Chamaremos este intervalo de tempo de *tempo de permanência* (TP) e utilizamos este valor como um dos parâmetros para estes testes. Consideramos, essencialmente, dois valores para o TP: um valor "alto" e um valor "baixo". Um valor alto para o TP tem como conseqüência um menor número de migrações do agentes móveis em um determinado intervalo de tempo (taxa de mobilidade baixa), enquanto que um valor baixo para o TP causa um maior número de migrações no mesmo intervalo de tempo (taxa de mobilidade alta). Como conseqüência imediata, para o nosso mecanismo do *Canal de Broadcast*, um maior número de migrações acarreta em um maior número de re-envios de mensagens e, com isto, um custo maior para o envio de uma mensagem. Portanto, o objetivo dos testes dinâmicos é avaliar o quão grande é este custo para duas taxas de migração.

Nestes testes não faz sentido uma comparação de tempo com um programa ASDK (como apresentado na seção anterior), uma vez que o ASDK sem o *Canal de Broadcast* não dá suporte para para o envio de mensagens de forma transparente à localização e migração dos agentes.

Nestes testes dinâmicos, consideraremos um agente requisitante estacionário que possui a tarefa de enviar mensagem ao *Proxy* requisitando a sua difusão aos agentes móveis de um grupo. A contagem de tempo inicia-se no momento em que a mensagem é enviada pelo *Proxy* aos *lugares de referência*.

Estipulamos para este teste também um número fixo de de mensagens (20 mensagens) que são enviadas em um intervalo de tempo (taxa de requisição) pelo agente requisitante.

Enquanto que nos testes estáticos o número de mensagens necessárias para completar o procedimento de um envio de mensagem até o recebimento das confirmações dos agentes pelo *Proxy* era fixo (4 mensagens), neste caso dinâmico, o número de mensagens é indefinido e depende da taxa de mobilidade dos agentes.

7.2.2 Objetivo

O objetivo destes testes foi comparar o tempo médio gasto pelo *Proxy* no envio de mensagens a um número de agentes móveis (até o recebimento da confirmação dos mesmos) em dois casos: com taxa de mobilidade alta (TP baixo) e com taxa de mobilidade baixa (TP alto).

7.2.3 Parâmetros dos testes

Para os testes dinâmicos usamos um número fixo de máquinas e domínios mas variamos o número de *lugares* e agentes móveis (Tabela 7.3). Consideramos os seguintes valores para os demais parâmetros destes testes:

- Tempo de Permanência em um lugar (TP): 8 segundos (baixo) e 25 segundos (alto);
- Taxa de requisição de difusão de mensagens pelo agente requisitante ao *Proxy*: 5 segundos.

	$N^{\underline{o}}$ máquinas	$N^{\underline{o}}$ *lugares*	$N^{\underline{o}}$ domínios	$N^{\underline{o}}$ agentes
A	4	6	3	4,7,13,25,31,37
B	4	9	3	4,7,13,25,31,37

Tabela 7.3: Parâmetros considerados nos testes dinâmicos

7.2.4 Resultados

Da mesma forma que no caso dos testes estáticos, foram feitas 3 rodadas de teste para cada número de agentes móveis indicado na Tabela 7.3, obtendo-se a média dos valores resultantes em cada uma das rodadas.

O gráfico da Figura 7.8 representa o resultado da execução dos testes para os valores da linha A da Tabela 7.3, com os valores dos TP's citados acima.

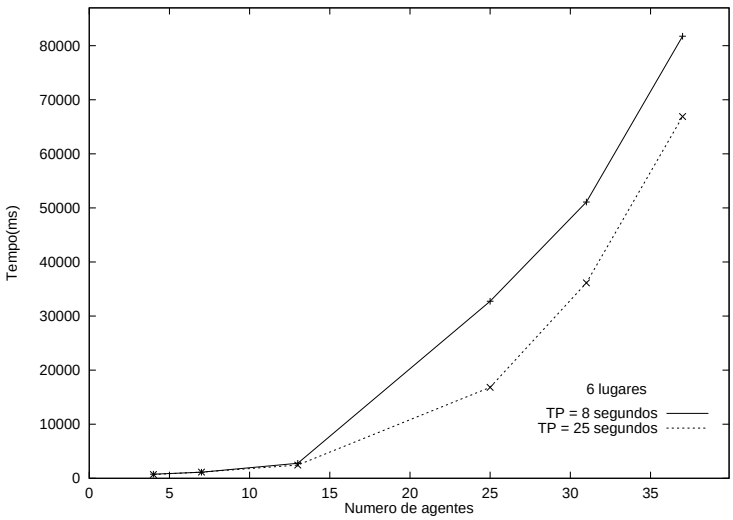

Figura 7.8: Resultado dos testes para 6 lugares

Conforme podemos observar, a curva relativa ao TP baixo cresce mais rapidamente que a curva de TP alto, como era de se esperar, uma vez que o número de migrações é maior e, conseqüentemente, o número de re-envios de mensagens também é maior. No entanto, nota-se

que com um número pequeno de agentes móveis (até 13 agentes) o tempo de permanência (ou taxa de movimentação) tem pouca influência sobre o tempo de processamento das mensagens.

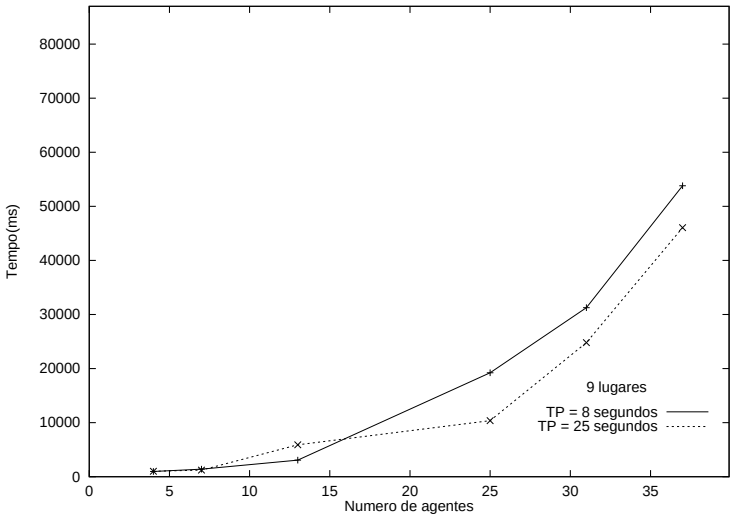

Figura 7.9: Resultado dos testes para 9 lugares

No gráfico das Figura 7.9, (correspondente aos valores da linha B da Tabela 7.3) os valores dos tempos médios em ambos os casos abaixaram consideravelmente, uma vez que o aumento do número de *lugares* aumenta também a possilibidade de haverem mais migrações intra-domínio, nos quais o re-envio de mensagens é descentralizado nos *lugares de referência*. Além disso, notamos que quando há uma incidência maior de migrações intra-domínio, também o aumento no número de agentes móveis tem uma influência menor no custo do envio, especialmente quando o tempo de permanência é pequeno (8 segundos).

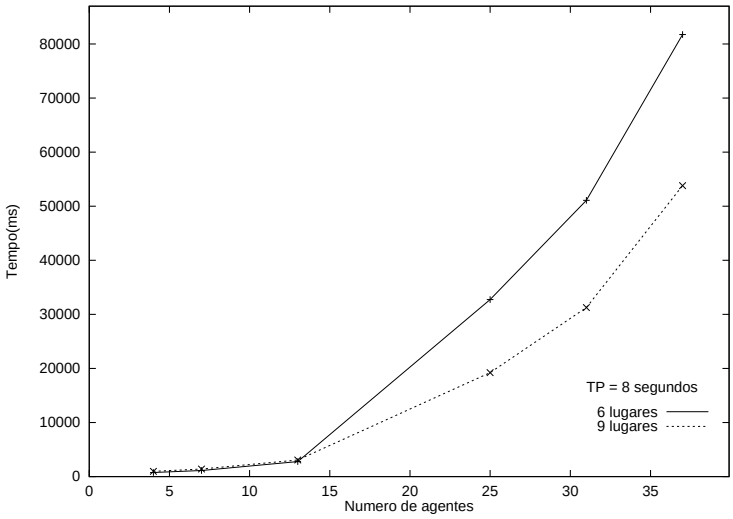

Figura 7.10: Comparação: Tempo de permanência baixa

Nos gráficos comparativos das Figuras 7.10 e 7.11, podemos observar que os testes feitos com 9 *lugares* apresentou um melhor desempenho tanto no caso do TP baixo como alto. Isto indica que quanto maior o número de *lugares*, menor é o custo requerido pelo *Canal de Broadcast*.

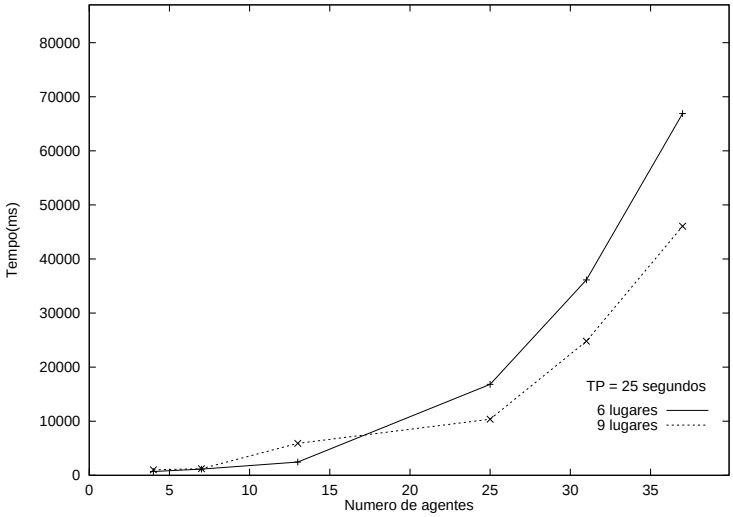

Figura 7.11: Comparação: Taxa de permanência alta

7.3 Conclusão sobre os resultados dos testes

Estes testes tiveram como principal objetivo identificar a sobrecarga (em termos de tempo de processamento) causada pelo *Canal de Broadcast* no envio de mensagens a grupos de agentes móveis, para dois casos: sem migração (testes estáticos) e com migração dos agentes (testes dinâmicos).

Nos testes estáticos, comparamos o desempenho do *Canal de Broadcast* com o envio direto utilizando somente o ASDK e observamos que *Canal de Broadcast* não chega a ser duas vezes mais custoso do que a difusão direta (usando o ASDK).

Nos testes dinâmicos, comparamos a sobrecarga causada pelo *Canal de Broadcast* em relação à taxa de mobilidade dos agentes.

Para ambas as classes de testes, concluímos que um melhor desempenho do *Canal de Broadcast* é alcançado quando há um número maior de máquinas, *lugares* e domínios, devido à descentralização do trabalho do *Proxy*.

Ressaltamos que estes testes não tiveram a finalidade de avaliar completa e exaustivamente o desempenho do *Canal de Broadcast* através de testes exaustivos. O conjunto de testes realizados não pode ser considerado suficiente para fornercer um quadro preciso sobre o custo de utilização do mecanismo. Para isto, seria necessário um número muito maior de testes com diversos outros parâmetros (como taxa de permanência bem menores ou bem maiores, um maior número de domínios e *lugares*, outros valores para a taxa de requisição de difusão de mensagens, etc.).

No entanto, acreditamos que os resultados obtidos sirvam para indicar a ordem de grandeza

da sobrecarga gerada do *Canal de Broadcast* e assim permitir um dimensionamento de aplicações distribuídas baseadas no nosso mecanismo. Por exemplo, verificamos que o *Canal de Broadcast* não é apropriado para aplicações que envolvam um grande número de agentes (30 ou mais), que tenham uma alta freqüência de migração e uma alta taxa de requisição de difusões. Ressaltamos também que a realização dos testes dinâmicos para um grande número de agentes e alta taxa de mobilidade foi problemática devido às próprias limitações do ambiente ASDK que utiliza conexões TCP/IP para a comunicação entre *aglets*.

Capítulo 8

Conclusão

Esta dissertação trata do projeto e implementação de um mecanismo de coordenação para agentes móveis baseado na difusão de mensagens.

A principal vantagem deste mecanismo de coordenação está na total transparência de localização: as mensagens podem ser endereçadas a um conjunto de agentes móveis independentemente da localização dos mesmos.

Em comparação aos outros modelos de coordenação, o *Canal de Broadcast* oferece as seguintes vantagens: não requer que os agentes se conheçam mutuamente ou que precisem conhecer a localização de outros agentes para que a comunicação seja possível, como é requirido no modelo de coordenação direta. No caso do *Canal de Broadcast*, os agentes podem interagir entre sí independentemente de sua localização.

Além disso, este mecanismo não requer que os agentes se movam para algum endereço específico na rede e também não há a necessidade da sincronização dos agentes móveis para que possam interagir (como nos modelos *blackboard* e *orientado a encontros*). Em vez disso, os agentes podem migrar continuamente para diferentes endereços e ao mesmo tempo, estar se comunicando com outros agentes através do *Canal de Broadcast*.

Foram apresentados dois exemplos concretos de coordenação que demostraram a utilidade deste *Canal de Broadcast*. Apresentamos também as etapas necessárias para o desenvolvimento de uma aplicação que se utiliza do mecanismo proposto e também os resultados da execução de uma implementação destes exemplos. Através destes resultados, pudemos comprovar que o *Canal de Broadcast* facilita muito a tarefa de coordenação de agentes móveis.

Realizamos, ainda, vários testes de desempenho do mecanismo. No caso dos testes estáticos, observamos que a razão de tempo gasto para uma difusão de mensagem entre o *Canal de Broadcast* e o *envio direto* de mensagens usando somente o ASDK é pequena. Já no caso dinâmico, o custo é maior, principalmente no caso de taxa de mobilidade alta. Porém, em ambos casos de testes, constatamos que uma infra-estrutura com maior número de *lugares* e domínios resulta em um custo menor e, conseqüentemente, um melhor desempenho.

Com relação ao uso do ASDK como ambiente de programação para a implementação do *Canal de Broadcast*, pode-se dizer que a experiência foi positiva. Isto se deve a vários fatores, entre eles, podemos citar:

- O ASDK está totalmente implementado em Java, fazendo com que os benefícios que esta linguagem oferece (independência de plataforma, carregamento dinâmico de classes, programação *multithread*, entre outros) possam ser aproveitados.
- Possibilita a comunicação entre agentes através da troca de mensagens de forma síncrona ou assíncrona: o *Canal de Broadcast* está baseado na troca de mensagens assíncronas.
- Possibilita o tratamento de eventos como a criação, migração e destruição de agentes móveis: estes eventos têm um papel fundamental no funcionamento do *Canal de Broadcast*.
- Permite a suspensão da execução do agente por certo intervalo de tempo e a retomada da execução, que foi usado no momento da criação da infra-estrutura do *Canal de Broadcast*.
- Permite a invocação flexível do método `handleMessage`[1], que possibilitou a ordenação de mensagens.

Além disso, o ASDK possui uma desvantagem que implica em uma motivação para este trabalho: não possibilita a comunicação com *aglets* remotos sem ter conhecimento do identificador e endereço de um *aglet*. No caso da interação local também, é preciso conhecer ao menos o identificador do *aglet* com quem deseja comunicar-se. Através do nosso mecanismo de coordenação, estes problemas são contornados.

O ASDK apresentou também outras deficiências que foram observadas durante a implementação:

- Um agente não aguarda o término do tratamento de uma mensagem no momento da migração. Ou seja, no momento em que o método `dispatch()` é invocado, o tratamento de mensagem que eventualmente esteja ocorrendo é interrompido e o agente acaba perdendo a mensagem.
- Se um agente possui referências (objeto *proxy* do ASDK) para outros agentes, estas referências perdem a sua validade durante a migração do agente, fazendo com que o agente tenha que obter essas referências toda vez que faz uma migração, antes de poder se comunicar com os outros *aglets* através dessas referências.
- O gerenciador de mensagens (`MessageManager`) do *aglet* não trata mensagens de uma subclasse de `Message`, impossibilitando ter uma extensão desta classe para incorporar os atributos e métodos específicos para este mecanismo de coordenação.
- o Aglets não permite a criação remota de um *aglet*. O *aglet* deve ser criado localmente e enviado para o lugar onde deve iniciar a execução.
- não possibilita a invocação remota de métodos.

Uma outra vantagem que o *Canal de Broadcast* oferece é a garantia da ordenação de mensagem, que é um requisito importante para alguns protocolos de coordenação, conforme citados no capítulo de exemplos de utilização deste mecanismo.

[1]Este método é invocado internamente pelo gerenciador de mensagens do *aglet* (que faz o tratamento de mensagens) em qualquer momento da execução.

Com relação à possibilidade de uso do *Canal de Broadcast* para aplicações na Internet vemos como principal empecilho a especialidade na configuração da infra-estrutura (*Proxy*, *lugares de referência* e *lugares*) necessária para o funcionamento do mecanismo. Ou seja, para que o mecanismo pudesse ser utilizado de uma forma mais ampla, o mesmo deveria estar incorporado a um ambiente de agentes móveis (como, por exemplo, o ASDK), de forma que os elementos da infra-estrutura possam ser instanciados a partir de qualquer *host* onde esteja executando o servidor de agentes específico do ambiente.

Como possíveis melhorias (ou extensões) para este mecanismo de coordenação, podemos citar:

- possibilitar o envio de mensagens a subgrupos de agentes;
- possibilitar criação dinâmica de grupos de agentes móveis cooperativos;
- possibilitar a expansão dinâmica de domínios (ou seja, a inclusão de novos lugares) e
- acrescentar mecanismos de tolerância à falhas.

Para finalizar, concluímos que o *Canal de Broadcast* é um mecanismo de coordenação que possui um custo razoavelmente pequeno em relação ao benefício que proporciona e, portanto, acreditamos que a realização deste trabalho foi satisfatória e rendeu resultados interessantes e que representou uma pequena contribuição para a área de coordenação de agentes móveis.

Apêndice A

Resultados da Execução dos Exemplos

A.1 Resultado da execução dos exemplos

A.1.1 Resultado do exemplo 1: Exclusão mútua usando relógios lógicos

Nesta seção, apresentaremos o resultado da execução do exemplo 1 com base nas informações apresentadas nas interfaces gráficas associadas a cada um dos agentes participantes. No resultado abaixo, as primeiras linhas identificam a classe do agente, o identificador recebido do Aglets no momento da criação e o endereço onde está localizado. As mensagens são identificadas pelo seu número de seqüência, por exemplo, as mensagens de 1 a 4 são as mensagens enviadas durante a criação da infra-estrutura e inicialização do protocolo.

Saída do agente 1:

```
  *****   Agente: AgentExample1  *****
  *****   AgentID: bd038621e678de73  *****
  *****   Endereco: atp://osasco:9000/  *****
====> Recebimento de msg: 1
====> Tratando a msg: 1
====> Tipo: START_AGENT
    - - - - - - - - - - - -
====> Recebimento de msg: 2
====> Enviou confirmacao para msg: 2
====> Tratando a msg: 2
====> Tipo: SETUPCOMPLETE
    - - - - - - - - - - - -
====> Recebimento de msg: 3
====> Enviou confirmacao para msg: 3
====> Tratando a msg: 3
====> Tipo: RESOURCE
    - - - - - - - - - - - -
====> Recebimento de msg: 4
====> Enviou confirmacao para msg: 4
====> Tratando a msg: 4
====> Tipo: MEMBERS
====> ID do Agente: 1
    - - - - - - - - - - - -
```

Na mensagem 4, o agente recebe o conjunto de agentes participantes e a partir deste parâmetro o agente obtém o seu próprio `id` (identificador da aplicação) que será utilizado no mecanismo de exclusão mútua.

Inicialmente, o agente 1 faz a requisição do recurso e em seguida, recebe as requisições dos

agentes 2 e 3 (mensagens 6 e 7). Observamos que o *timestamp* da mensagem de requisição do agente 1 é igual ao da mensagem do agente 2 e do agente 3, mas o recurso é concedido ao agente 1, uma vez que o seu `id` é o menor. O agente 1 migra, então, para o endereço onde está o recurso (`atp://osasco:10000`). A tabela abaixo indica a situação atual do vetor de estados `q` neste momento da execução para todos os agentes.

1	2	3
REQ 2	REQ 2	REQ 2

Saída do agente 1:

```
====> Fazendo o pedido do recurso: enviando a msg REQ, ts: 2
    - - - - - - - - - - -
====> Recebimento de msg: 5
====> Enviou confirmacao para msg: 5
====> Tratando a msg: 5
====> Tipo: REQ
====> REQ do Agente: 1, ts: 2
    - - - - - - - - - - -
====> Recebimento de msg: 6
====> Enviou confirmacao para msg: 6
====> Tratando a msg: 6
====> Tipo: REQ
====> REQ do Agente: 2, ts: 2
    - - - - - - - - - - -
====> Recebimento de msg: 7
====> Enviou confirmacao para msg: 7
====> Tratando a msg: 7
====> Tipo: REQ
====> REQ do Agente: 3, ts: 2
    - - - - - - - - - - -
**************************************************
 ====> Permissao concedida para Agente: 1, ts = 2
**************************************************
  *****   Agente: AgentExample1  *****
  *****   AgentID: bd038621e678de73  *****
  *****   Endereco: atp://osasco:10000/  *****
```

A seguir, os agentes recebem várias mensagens ACK confirmando mutuamente o recebimento de mensagens REQ dos outros agentes - linhas que foram omitidas nesta descrição. Ao liberar o recurso, o agente 1 envia a mensagem REL e em seguida, migra para o endereço `atp://rebutosa:10000` e novamente, faz uma requisição do recurso. O próximo agente a obter o acesso ao recurso é o agente 2, pois este possui o menor `id`. O agente 2 migra para o endereço onde está o recurso e em seguida, recebe a mensagem de requisição do agente 1. Podemos verificar isso no resultado abaixo.

Saída do agente 1:

```
====> Liberando o recurso: enviando a msg REL, ts: 15
    - - - - - - - - - - -
====> Recebimento de msg: 17
====> Enviou confirmacao para msg: 17
====> Tratando a msg: 17
====> Tipo: REL
====> REL do Agente: 1, ts: 15
    - - - - - - - - - - -
  *****   Agente: AgentExample1  *****
  *****   AgentID: bd038621e678de73  *****
  *****   Endereco: atp://rebutosa:10000/  *****
====> Fazendo o pedido do recurso: enviando a msg REQ, ts: 17
```

Saída do agente 2:

```
====> Recebimento de msg: 17
```

```
====> Enviou confirmacao para msg: 17
====> Tratando a msg: 17
====> Tipo: REL
====> REL do Agente: 1, ts: 15
    -  -  -  -  -  -  -  -  -  -  -  -
**************************************************
 ====> Permissao concedida para Agente: 2, ts = 2
**************************************************
  *****   Agente: AgentExample1  *****
  *****   AgentID: 52217d92646f75b2  *****
  *****   Endereco: atp://osasco:10000/  *****
====> Recebimento de msg: 18
====> Enviou confirmacao para msg: 18
====> Tratando a msg: 18
====> Tipo: REQ
====> REQ do Agente: 1, ts: 17
    -  -  -  -  -  -  -  -  -  -  -  -
```

Situação atual:

1	2	3
REQ 17	REQ 2	REQ 2

Em seguida, o agente 2 libera o recurso e o agente 3 ganha o direito de acesso. Assim como os outros agentes, o agente 3 migra para acessar o recurso e logo depois libera o recurso.

1	2	3
REQ 17	REL 22	REQ 2

Saída do agente 3:

```
====> Recebimento de msg: 22
====> Enviou confirmacao para msg: 22
====> Tratando a msg: 22
====> Tipo: REL
====> REL do Agente: 2, ts: 22
    -  -  -  -  -  -  -  -  -  -  -  -
**************************************************
 ====> Permissao concedida para Agente: 3, ts = 2
**************************************************
  *****   Agente: AgentExample1  *****
  *****   AgentID: b6c1374f18a51433  *****
  *****   Endereco: atp://osasco:10000/  *****
====> Liberando o recurso: enviando a msg REL, ts: 24
    -  -  -  -  -  -  -  -  -  -  -  -
====> Recebimento de msg: 23
====> Enviou confirmacao para msg: 23
====> Tratando a msg: 23
====> Tipo: REL
====> REL do Agente: 3, ts: 24
    -  -  -  -  -  -  -  -  -  -  -  -
```

O agente 1, novamente, terá o direito de acessar o recurso e recebe uma nova requisição do agente 3.

1	2	3
REQ 17	REL 22	REQ 26

Saída do agente 1:

```
====> Recebimento de msg: 23
====> Enviou confirmacao para msg: 23
====> Tratando a msg: 23
```

```
====> Tipo: REL
====> REL do Agente: 3, ts: 24
    - - - - - - - - - - - -
**************************************************
 ====> Permissao concedida para Agente: 1, ts = 17
**************************************************
  *****   Agente: AgentExample1  *****
  *****   AgentID: bd038621e678de73  *****
  *****   Endereco: atp://osasco:10000/  *****
====> Recebimento de msg: 24
====> Enviou confirmacao para msg: 24
====> Tratando a msg: 24
====> Tipo: REQ
====> REQ do Agente: 3, ts: 26
    - - - - - - - - - - - -
```

Finalmente, o agente 1 libera o recurso e o agente 3 acessa, e a execução termina.

No agente 3:

```
====> Recebimento de msg: 28
====> Enviou confirmacao para msg: 28
====> Tratando a msg: 28
====> Tipo: REL
====> REL do Agente: 1, ts: 31
    - - - - - - - - - - - -
**************************************************
 ====> Permissao concedida para Agente: 3, ts = 26
**************************************************
  *****   Agente: AgentExample1  *****
  *****   AgentID: b6c1374f18a51433  *****
  *****   Endereco: atp://osasco:10000/  *****
 ====> Liberando o recurso: enviando a msg REL, ts: 33
    - - - - - - - - - - - -
====> Recebimento de msg: 29
====> Enviou confirmacao para msg: 29
====> Tratando a msg: 29
====> Tipo: REL
====> REL do Agente: 3, ts: 33
    - - - - - - - - - - - -
```

1	2	3
REL	ACK	REL
31	27	33

Como podemos observar, os agentes conseguiram o acesso exclusivo ao recurso através do algoritmo de exclusão mútua adotado e este processo realmente foi possível devido ao emprego do nosso mecanismo de coordenação que possibilitou que os agentes recebessem todas as mensagens enviadas e que os mesmos pudessem tratá-las na mesma ordem, mesmo fazendo migrações durante a execução. Abaixo, apresentamos uma parte da saída do agente 1 em que este fato fica visível:

Saída do agente 1:

```
 ...
====> Recebimento de msg: 7
====> Enviou confirmacao para msg: 7
====> Tratando a msg: 7
====> Tipo: REQ
====> REQ do Agente: 3, ts: 2
    - - - - - - - - - - - -
**************************************************
 ====> Permissao concedida para Agente: 1, ts = 2
**************************************************
  *****   Agente: AgentExample1  *****
  *****   AgentID: bd038621e678de73  *****
  *****   Endereco: atp://osasco:10000/  *****
====> Recebimento de msg: 14
====> Enviou confirmacao para msg: 14
====> Armazenando a msg: 14
====> Tipo: ACK
    - - - - - - - - - - - -
====> Recebimento de msg: 11
```

```
====> Enviou confirmacao para msg: 11
====> Armazenando a msg: 11
====> Tipo: ACK
    - - - - - - - - - - -
====> Recebimento de msg: 8
====> Enviou confirmacao para msg: 8
====> Tratando a msg: 8
====> Tipo: ACK
====> ACK do Agente: 3 para REQ do Agente: 1, ts: 3
    - - - - - - - - - - -
====> Recebimento de msg: 13
====> Enviou confirmacao para msg: 13
====> Armazenando a msg: 13
====> Tipo: ACK
    - - - - - - - - - - -
====> Recebimento de msg: 10
====> Enviou confirmacao para msg: 10
====> Armazenando a msg: 10
====> Tipo: ACK
    - - - - - - - - - - -
 ...
====> Recebimento de msg: 9
====> Enviou confirmacao para msg: 9
====> Tratando a msg: 9
====> Tipo: ACK
====> ACK do Agente: 1 para REQ do Agente: 1, ts: 3
    - - - - - - - - - - -
====> Em handleNextMessage: Tratando a msg: 10
====> Recebimento de msg: 10
====> Tratando a msg: 10
====> Tipo: ACK
====> ACK do Agente: 2 para REQ do Agente: 1, ts: 3
    - - - - - - - - - - -
====> Em handleNextMessage: Tratando a msg: 11
====> Recebimento de msg: 11
====> Tratando a msg: 11
====> Tipo: ACK
====> ACK do Agente: 3 para REQ do Agente: 2, ts: 4
    - - - - - - - - - - -
====> Em handleNextMessage: Tratando a msg: 12
 ...
```

No resultado acima, verificamos que antes da migração o agente 1 havia recebido e tratado a mensagem 7, mas quando chegou no novo lugar, recebeu mensagens de número maior que o esperado (14 e 11) e estas foram armazenadas sem ser processadas. Quando a mensagem 8 é recebida, esta é tratada e em seguida, recebeu novamente várias mensagens com números maiores que foram armazenadas. Só quando a mensagem 9 chega, o agente trata esta e todas as mensagens com número de seqüência maior do que as que estavam armazenadas.

A.1.2 Resultado do exemplo 2: Protocolo Manager-Workers

Apresentamos o resultado da execução do exemplo Manager-Workers, de acordo com as informações obtidas das interfaces gráficas dos agentes. Inicialmente, o coordenador possui cinco tarefas que devem ser executadas pelos agentes de acordo com a descrição do protocolo apresentado acima. Essas tarefas são enviadas ao *Proxy* em intervalos de tempo regulares.

Abaixo temos a parte inicial deste resultado para um dos trabalhadores. Analogamente ao exemplo anterior, as primeiras linhas indicam o tipo do agente, o seu identificador e o endereço. Cada um dos trabalhadores recebe inicialmente duas mensagens para a configuração inicial do mecanismo de coordenação. A mensagem 3 é a mensagem do *Proxy* com o conjunto de agentes participantes. Da mesma forma que no exemplo 1, através desta mensagem o agente trabalhador obtém o seu próprio `id` (identificador para esta aplicação).

Saída do agente 1:

```
*****   Agente: WorkerExample2  *****
*****   AgentID: 74455fbccd5e3764  *****
```

```
  *****   Endereco: atp://osasco:9000/  *****
====> Recebimento de msg: 1
====> Tratando a msg: 1
====> Tipo: START_AGENT
    -  -  -  -  -  -  -  -  -  -  -
====> Recebimento de msg: 2
====> Enviou confirmacao para msg: 2
====> Tratando a msg: 2
====> Tipo: SETUPCOMPLETE
    -  -  -  -  -  -  -  -  -  -  -
====> Recebimento de msg: 3
====> Enviou confirmacao para msg: 3
====> Tratando a msg: 3
====> Tipo: WORKERS
====> ID do Agente: 1
    -  -  -  -  -  -  -  -  -  -  -
```

Inicialmente, uma mensagem NEW_TASK é enviada pelo coordenador e o agente 1 aceita a tarefa e envia a mensagem ACCEPT ao *Proxy*. A seguir, o coordenador envia uma nova tarefa e, neste caso, o trabalhador 2 aceita a tarefa e em seguida, o trabalhador 1 envia a mensagem de término da tarefa. A seguir, apresentamos a saída da interface do agente 1 com o procedimento descrito:

Saída do agente 1:

```
====> Recebimento de msg: 4
====> Enviou confirmacao para msg: 4
====> Tratando a msg: 4
====> Tipo: NEW_TASK
====> NEW_TASK, TaskID: 1
***************************************************************
 ====> Nova tarefa aceita pelo trabalhador: 1, TaskID: 1
***************************************************************
====> Recebimento de msg: 5
====> Enviou confirmacao para msg: 5
====> Tratando a msg: 5
====> Tipo: ACCEPT
====> ACCEPT do trabalhador: 1, taskID: 1
    -  -  -  -  -  -  -  -  -  -  -
====> Recebimento de msg: 6
====> Enviou confirmacao para msg: 6
====> Tratando a msg: 6
====> Tipo: NEW_TASK
====> NEW_TASK, TaskID: 2
    -  -  -  -  -  -  -  -  -  -  -
====> Recebimento de msg: 7
====> Enviou confirmacao para msg: 7
====> Tratando a msg: 7
====> Tipo: ACCEPT
====> ACCEPT do trabalhador: 2, taskID: 2
    -  -  -  -  -  -  -  -  -  -
***************************************************************
 ====> Terminou a tarefa, trabalhador: 1, taskID: 1
***************************************************************
```

Em seguida, uma mensagem com uma nova tarefa é recebida e é e aceita pelo trabalhador 1. O trabalhador 2 envia a mensagem de término da tarefa e em seguida, uma mensagem com uma nova tarefa é lançada e o trabalhador 2 aceita e executa a tarefa. O trabalhador 3 esteve todo o tempo disponível, não executando nenhuma tarefa.

Saída do agente 1:

```
====> Recebimento de msg: 9
====> Enviou confirmacao para msg: 9
====> Tratando a msg: 9
====> Tipo: NEW_TASK
====> NEW_TASK, TaskID: 3
    -  -  -  -  -  -  -  -  -  -  -
***************************************************************
 ====> Nova tarefa aceita pelo trabalhador: 1, TaskID: 3
***************************************************************
...
====> Recebimento de msg: 11
====> Enviou confirmacao para msg: 11
====> Tratando a msg: 11
```

```
====> Tipo: FINISHED
====> FINISHED do trabalhador: 2, TaskID: 2
    - - - - - - - - - - - -
====> Recebimento de msg: 12
====> Enviou confirmacao para msg: 12
====> Tratando a msg: 12
====> Tipo: NEW_TASK
====> NEW_TASK, TaskID: 4
    - - - - - - - - - - - -
====> Recebimento de msg: 13
====> Enviou confirmacao para msg: 13
====> Tratando a msg: 13
====> Tipo: ACCEPT
====> ACCEPT do trabalhador: 2, taskID: 4
    - - - - - - - - - - - -
*************************************************************
 ====> Terminou a tarefa, trabalhador: 1, taskID: 3
*************************************************************
====> Recebimento de msg: 15
====> Enviou confirmacao para msg: 15
====> Tratando a msg: 15
====> Tipo: NEW_TASK
====> NEW_TASK, TaskID: 5
    - - - - - - - - - - - -
...
*************************************************************
 ====> Terminou a tarefa, trabalhador: 1, taskID: 5
*************************************************************
====> Recebimento de msg: 18
====> Enviou confirmacao para msg: 18
====> Tratando a msg: 18
====> Tipo: FINISHED
====> FINISHED do trabalhador: 1, TaskID: 5
    - - - - - - - - - - - -
```

Portanto, através do nosso mecanismo, conseguiu-se implementar o protocolo proposto: as tarefas foram aceitas por apenas um trabalhador e em qualquer momento da execução, o estado de cada um destes esteve sempre consistente com o estado de todos os demais.

Referências Bibliográficas

[1] A. Acharya, M. Ranganathan, and J.Saltz. Sumatra: a Language for Resource Aware Mobile Programs. *Mobile Object Systems, Lecture Notes in Computer Science*, Spring Velarg(D)(1222):111–130, 1997.

[2] IBM's aglets Home Page. http://www.trl.ibm.co.jp/aglets.

[3] M. Barbaceanu and M. S. Fox. COOL - A Language for Describing Coordination in Multi-Agent Systems. In AAAi Press/The MIT Press, editor, *Lesser (ed) Proceedings of First International Conference on Multi-Agent Systems*, pages 17–24.

[4] J. Baumann, F. Hohl, and M. Straber K. Rothermel. Mole - Concepts of a Mobile Agent System. *WWW Journal, Special Issue on Applications and Techniques of Web Agents*, 1998.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. How to Coordinate Internet Applications based on Mobile Agents. *IEEE Seventh International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, June 1998.

[6] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agents Coordination. In *Proceedings of the 2. International Workshop on Mobile Agents, Lecture Notes on Computer Science*, number 1477 in Lecture Notes in Computer Science, pages 237–248. Stuttgart(D), September 1998.

[7] L. Cardelli and D. Gordon. Mobile Ambients. *LNCS*, Spring-Verlag(D)(1378):140–155, 1998.

[8] D. Chauhan. JAFMAS: A Java-based Agent Framework for Multi-Agent Systems. Technical report, Univesity of Cincinnati, 1997.

[9] D.M. Chess, C.G. Harrison, and A. Kerschenbaum. Mobile Agents: Are they a good idea? *Mobile Object Systems, Lecture Notes in Computer Science*, Spring Velarg(D)(1222):111–130, February 1997.

[10] P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. *LNCS*, Springer-Verlag(D)(1222):213–226, February 1997.

[11] P. Domel, A. Lingnau, and O. Drobnik. Mobile Agent Interaction in Heterogeneous Environment. *LNCS*, Springer-Verlag(D)(1219):136–148, April 1997.

[12] M. Endler. Agentes Móveis: um tutorial. Relatorio Técnico RT-MAC9805, Universidade de São Paulo, 1998.

[13] General Magic, Odissey Home Page. http://www.generalmagic.com/technology/odissey.html.

[14] R. Gray. Agent Tcl: A flexible and secure mobile-agent system. *Proceedings of the Forth Annual Tcl/Tk Workshop (TCL '96)*, July 1996. Monterey, California.

[15] D. Lange and M. Oshima. *Programming and Deploying Java[tm] Mobile Agents with Aglets[tm]*. Addison-Wesley, 1998.

[16] D. B. Lange. Java Aglet Application Programming Interface (J-AAPI). White paper, IBM Tokio Research Laboratory, February 1997. http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html.

[17] V. R. Lesser. Reflection on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. Technical Report 98-10, CMPSCI, 1998.

[18] L.Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] T. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(2):87–119, March 1994.

[20] M. Oshima, G. Karjoth, and K. Ono. Aglets Specification 1.1. Technical report, IBM, September 1998. http://www.trl.ibm.co.jp/aglets/spec11.html.

[21] OMG Home Page. http://www.omg.org/docs/orbos/98-04-05.pdf.

[22] H. Peine and T. Stolpmann. The Architecture of ARA Plataform for Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents*, number 1219 in Lecture Notes on Computer Science, pages 50–61, Berlin(D), April 1997.