2
Sistemas Multi-Agentes Móveis

Sistemas multi-agentes móveis (SMAs móveis) são constituídos por uma plataforma de mobilidade e pelos agentes móveis nela executados. A *plataforma* define todas as questões relacionadas aos mecanismos que dão suporte à mobilidade de agentes. Além disso, geralmente uma plataforma de mobilidade define um *framework* para programação ou instanciação de aplicações baseadas em agentes móveis. De fato, os *agentes móveis* são classes instanciadas a partir do *framework* de desenvolvimento disponibilizado por uma plataforma.

São apresentadas neste capítulo as definições gerais de SMAs, importantes no escopo deste trabalho. Posteriormente, são apresentadas definições específicas de SMAs móveis. A fim de se compreender as possíveis vantagens de utilização dos agentes móveis, é apresentada também uma comparação entre o paradigma de agentes móveis e as outras abordagens mais tradicionais. Finalmente, é apresentada uma análise sistemática de algumas das principais plataformas de agentes móveis da atualidade.

2.1.
Definições Básicas de SMAs

Nesta seção, é realizada a introdução de algumas definições importantes no domínio de SMAs, a saber: conhecimento, tipos de agente, e papéis de agente.

Conhecimento é o fato ou condição de estar consciente de algo (Bigus & Bigus, 2001). O conhecimento de um agente consiste em: (i) informações sobre o ambiente do agente e outras entidades que compõem o ambiente ou sobre o agente em si, e (ii) o conjunto de funcionalidades providas pelo agente. O conhecimento de um agente pode ser classificado em: conhecimento intrínseco e conhecimento extrínseco (Kendall, 1999).

O *conhecimento intrínseco* é o conhecimento relacionado às informações e funcionalidades básicas do agente (Kendall, 1999). O *conhecimento extrínseco* é o conhecimento relacionado aos diferentes papéis desempenhados pelo agente nos vários contextos de colaboração. Compreende a informação e os serviços necessários para que o agente possa desempenhar papéis em interações com outros agentes (Kendall, 1999).

O conhecimento de um agente pode ser estruturado de acordo com modelos específicos (Rao & Gerogeff, 1995). Existem várias propostas para modelagem do conhecimento, mas os *elementos do conhecimento* utilizados freqüentemente são designados por crenças, objetivos, ações e planos (Rao & Gerogeff, 1995). As *crenças* do agente são elementos do conhecimento que descrevem as informações sobre a agente em si, sobre o ambiente e outros agentes no ambiente.

Um *plano* descreve a estratégia usada para se alcançar um objetivo do agente. A seleção de planos é baseada nas crenças do agente. O comportamento de um agente é dirigido pela execução de seus planos, os quais selecionam ações específicas a fim de alcançar os *objetivos* definidos. As *ações* e os planos são usados para implementar os serviços do agente.

Tipo de agentes é outra definição fundamental em SMAs. Em geral, um SMA possui vários tipos de agentes de software (Nwana, 1996; Bradshaw, 1997). Diferentes tipos de agentes são necessários, porque a estrutura interna de um tipo define o mesmo conjunto de funcionalidades para agentes que pertencem ao mesmo tipo. Aos vários tipos também estão associadas diferentes propriedades de agentes como adaptação, colaboração, aprendizagem e mobilidade. As relações mantidas entre estas propriedades também são dependentes de cada tipo.

Há diferentes maneiras de classificar os tipos de agentes, segundo as várias características que os distinguem uns dos outros. O atributo de *mobilidade* constitui um primeiro critério para classificar tipos de agentes (Nwana, 1996). Um tipo de agente é dito *móvel* quando possui a habilidade de mover-se entre diferentes *hosts* de uma rede. Um agente sem esta habilidade é classificado como *estático* ou *estacionário*.

Os tipos de agentes também podem ser classificados de acordo com o conjunto de serviços que fornecem ao ambiente. Neste caso, os agentes predominantes em SMAs são agentes de informação, agentes de usuário e agentes de interface. *Agentes de informação* são tipos de agentes fortemente acoplados às

fontes de informação dos SMAs com o objetivo de encontrar informações em resposta a consultas solicitadas. Além disso, agentes de informação monitoram ativamente as fontes de informação para detecção de condições pré-especificadas (Nwana, 1996; Brugali & Sycara, 1999). Os agentes de informação ajudam a automatizar o processo de decidir se uma informação disponível é útil para um propósito determinado.

Agentes de usuário representam entidades físicas, em geral seres humanos, que agem como assistentes pessoais no cumprimento de tarefas do usuário (Nwana, 1996; Brugali & Sycara, 1999). Dentre outras responsabilidades, estes agentes armazenam informações e preferências dos usuários. *Agentes de interface* gerenciam interfaces gráficas e adaptam seu comportamento de acordo com as preferências dos usuários que usam tais interfaces (Lucena et al., 2004).

Papéis[8] *de agentes* é outra definição importante em SMAs, pois tem sido usada durante as fases de concepção (Silva et al., 2003), análise (Wooldridge et al., 2000; Zambonelli et al., 2001), projeto (Kendall, 1999; Zambonelli et al., 2001; Odell et al., 2003) e implementação (Fowler, 1997; Kendall, 1999) de SMAs. Diferentes papéis podem ser exercidos por um tipo de agente. Existem dois tipos de papéis de agentes: (i) *papéis intrínsecos*, que definem o conhecimento intrínseco e as funcionalidades básicas de um agente, e (ii) *papéis extrínsecos* ou *colaborativos*, que dizem respeito ao exercício de capacidades extrínsecas do agente em seus relacionamentos de colaboração.

A relação existente entre tipos e papéis de agentes é a seguinte: a definição de um tipo de agente está relacionada ao papel intrínseco exercido pelo agente nos sistemas, enquanto a definição de um papel captura o modo como um agente interage com outros agentes em seus relacionamentos de colaboração. Para efeito de simplificação, a palavra "papéis" e a expressão "papéis colaborativos" são usadas indistintamente neste texto.

Cada papel possui o conhecimento e o comportamento necessários para as colaborações com outros agentes. Assim como um agente, um papel possui crenças, objetivos, ações e planos. Um papel pode possuir comportamentos específicos para interação com outros agentes, algoritmos de decisão específicos, e estratégias de adaptação específicas. Um papel pode também possuir

8 Do inglês: *roles*.

comportamentos especializados para as propriedades adicionais de agentes. Conseqüentemente, a estrutura e o comportamento de um papel são similares à estrutura e comportamento de um tipo de agente.

2.2. Definições de SMAs Móveis

Nesta seção, são apresentadas definições específicas de SMAs móveis. *Agentes móveis* são agentes de software que possuem a habilidade de se mover de um *host* a outro em uma rede (Harrison et al., 1995; White, 1995). Em cenários comumente encontrados, os agentes móveis deslocam-se para o *host* onde os dados de interesse da aplicação estão armazenados, selecionam as informações de que o usuário necessita e retornam com os resultados para o *host* inicial.

Entretanto, em outros cenários, o agente é projetado não para selecionar as informações, mas apenas para transferir os dados encontrados, a fim de que outros agentes possam efetuar a tarefa de seleção das informações. É possível ainda que os agentes móveis sejam projetados para trocar mensagens com os agentes que efetivamente transferem, processam ou retornam as informações para os usuários. As funcionalidades básicas de agentes móveis estão relacionadas aos requisitos das aplicações onde serão utilizados.

Do ponto de vista de implementação, um *agente móvel* é constituído de código e estado, como qualquer outro agente de software. O *código* é o programa que define o agente; o *estado* consiste no conjunto de valores dos atributos internos de um agente durante sua execução. Um agente móvel se movimenta em um *ambiente distribuído*, entre *hosts* de uma rede aos quais possui acesso. O *acesso* é configurado através de um *servidor*, instalado nos *hosts*, que, dentre outras funções, possui a tarefa de fornecer aos agentes um *local* adequado à sua execução. Este local de execução é designado *contexto de execução* dos agentes.

As interações entre agentes móveis e o servidor de agentes de suas respectivas plataformas podem ser denominadas em conjunto como o *protocolo de mobilidade do agente*. Quando um agente é transferido na rede, sua execução é interrompida no *host* original, ele é transportado para outro contexto no ambiente distribuído e sua execução é reiniciada a partir do ponto de interrupção anterior. Quando migra de um *host* a outro, um agente leva consigo o código e os dados de

sua execução. Isto permite que o agente seja transportado na rede, sem que seus atributos internos devam ser reiniciados.

A Figura 2 ilustra uma estrutura geral para a execução de agentes móveis em uma rede: o servidor de agentes, o contexto de execução dos agentes fornecido pelo servidor e a infra-estrutura de rede necessária à mobilidade dos agentes.

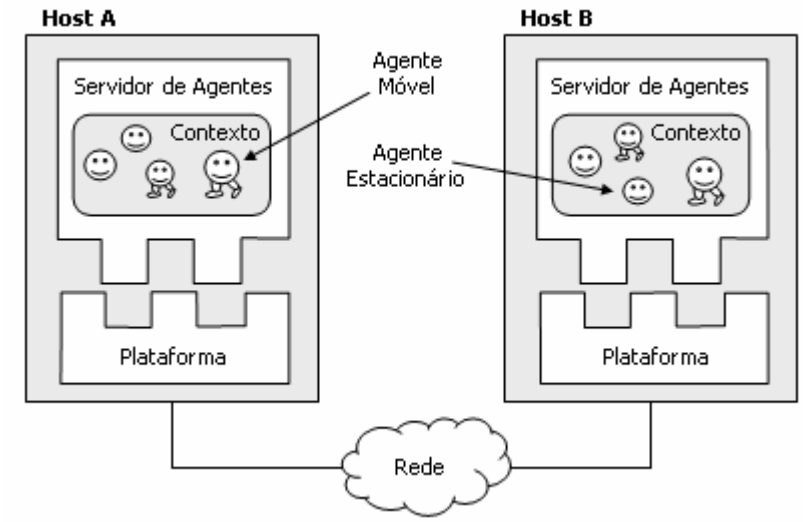

Figura 2. Estrutura Geral de SMAs Móveis

A existência de um agente móvel é representada segundo as etapas de um modelo denominado "ciclo de vida". Estas etapas correspondem às operações de instanciação, inicialização, movimentação e destruição de agentes. Os procedimentos invocados por estas operações são denominados em conjunto como *protocolos de instanciação, inicialização, movimentação* e *destruição* de agentes, respectivamente. A Figura 3 ilustra o ciclo de vida de agentes móveis.

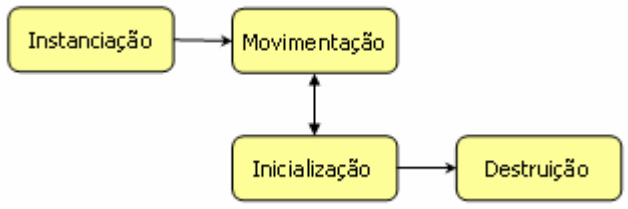

Figura 3. Ciclo de Vida de Agentes Móveis

A operação de *instanciação* é executada somente uma vez durante todo o ciclo de vida de um agente. Após sua instanciação, o agente recebe um identificador, tem seu estado interno iniciado e é preparado para instruções adicionais. A *inicialização* é a operação efetuada a cada vez que um agente chega

a um novo *host*. Esta operação só é possível porque um agente possui sua própria *thread* de execução. A operação de *destruição* implica na finalização das atividades de um agente, liberando os recursos sendo utilizados por ele. Após a destruição, o estado do agente é perdido definitivamente.

Durante o ciclo de vida, um agente pode se mover entre os *hosts* de uma rede. A *movimentação* ou *migração* permite ao agente transferir-se para o contexto onde se localiza um objeto com o qual deseja interagir, de modo a tirar proveito do fato de estar localizado no mesmo lugar que ele. As circunstâncias nas quais os agentes devem se movimentar, ou ainda, os instantes em que a ação de movimentação deve ser disparada, são especificados por desenvolvedores e são denominados *pontos de movimento* das aplicações. Nestes instantes, é possível haver a desistência quanto à necessidade de movimentação. Os *pontos de movimento* são então chamados *pontos de decisão do movimento*.

Os procedimentos executados por um agente imediatamente antes de sua transferência para um novo ambiente são denominados em conjunto como *procedimentos de partida*. O envio de mensagens de partida e a finalização de processos são exemplos de procedimentos de partida de agentes. Os procedimentos executados imediatamente após a transferência de um agente são denominados em conjunto como *procedimentos de chegada*. O envio de mensagens e a abertura de processos são exemplos de procedimentos de chegada.

Em geral, os procedimentos de partida e chegada estão associados à atualização de dados relacionados aos contextos de execução de agentes móveis. Estes dados são tratados neste trabalho sob o nome de *itinerário*. O itinerário inclui identificadores para *hosts* acessíveis denominados *hosts vizinhos*, identificadores para *hosts* já visitados denominados *hosts visitados*, e assim por diante. A configuração e manutenção do itinerário de agentes pode ser efetuada estática ou dinamicamente e segundo restrições de acesso impostas por servidores de plataformas. A manutenção de dados do itinerário é importante porque, quando um agente ingressa em um novo *host*, ele pode requerer o conhecimento específico de seu novo contexto de execução, o que inclui a lista de agentes presentes e/ou dos objetos que são acessíveis a ele, como referências a *agentes vizinhos*, *agentes mestres*, *localizações de hosts acessíveis*, etc.

A migração dos agentes pode ser classificada em fraca ou forte (Fuggetta et al., 1998). Em ambas, o agente tem sua execução reiniciada sem perda dos valores

de seus atributos internos. Porém, enquanto na *migração fraca* o agente reinicia a execução em um novo *host* a partir da primeira linha de seu código, na *migração forte* a execução reinicia exatamente do ponto onde foi interrompida, uma vez que a pilha de execução é integralmente recuperada no novo contexto do agente.

As plataformas de agentes móveis baseadas em Java permitem implementação apenas de migração fraca, pois devido à transparência destas linguagens em relação aos sistemas operacionais torna-se difícil reiniciar a execução do agente a partir do ponto de interrupção (Fuggetta et al., 1998). Existem implementações de plataformas Java que fornecem o mecanismo de migração forte; no entanto, tais ambientes utilizam versões modificadas da JVM. As plataformas de mobilidade utilizadas neste trabalho permitem implementação de agentes apenas com mobilidade fraca.

2.3. Agentes Móveis e Paradigmas da Computação em Rede

Para compreender as vantagens de utilização dos agentes móveis em aplicações distribuídas, é necessário comparar suas características às dos outros paradigmas comuns da computação em rede. São quatro os principais paradigmas da computação em rede (Fuggetta et al., 1998), a saber: (1) Cliente-Servidor, (2) Execução Remota, (3) Código sob Demanda e (4) Agentes Móveis. Para a descrição destes paradigmas, Fuggetta considera um cenário onde um componente computacional A, situado em um *host* Ha, solicita a execução de um determinado serviço na rede. É suposta a existência de um *host* Hb, que também estará envolvido na realização do serviço anterior.

O *paradigma cliente-servidor* é vastamente utilizado. Neste paradigma, um componente de software B, denominado servidor e localizado em um *host* Hb, oferece um conjunto de serviços na rede. O código e os recursos necessários à execução destes serviços estão disponíveis no *host* Hb. Um componente A, denominado cliente e situado em um *host* Ha, solicita ao servidor a execução de um determinado serviço. Como resposta, o componente B processa o código do serviço solicitado por A, utilizando, para isso, os recursos que estão disponíveis no *host* Hb. A execução do serviço produz um resultado que é enviado pelo servidor em reposta à solicitação do cliente. Exemplos de aplicação deste

paradigma: páginas dinâmicas com *script* interpretado no servidor, *Remote Method Invocation* (RMI), etc. A Figura 4 ilustra o paradigma cliente-servidor.

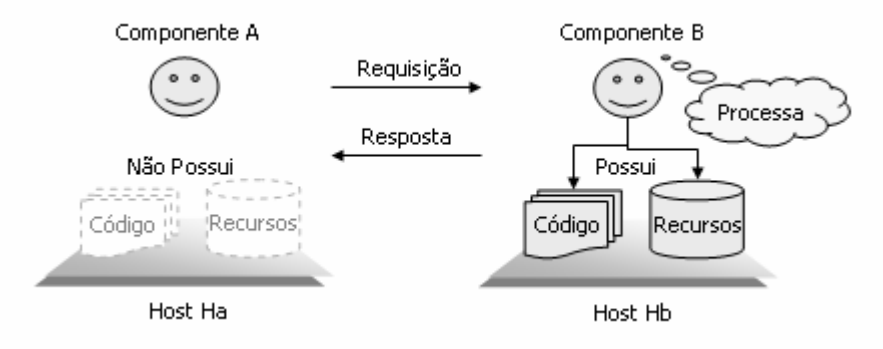

Figura 4. Paradigma Cliente-Servidor

No *paradigma de execução remota*, o componente A possui o código para executar os serviços que necessita, mas faltam os recursos necessários à execução das tarefas. O componente A então envia o código da tarefa para um componente B, localizado no *host* Hb, onde estão disponíveis os recursos necessários à execução do serviço. O componente B efetua o processamento e envia os resultados obtidos para o componente A. Exemplos deste paradigma geralmente estão relacionados a sistemas *Unix-like*: código gerado por um processador de textos *PostScript* e executado pelo interpretador de uma impressora; execução do comando *rsh*, através do qual um *script* de usuário pode ser executado remotamente; etc. A Figura 5 ilustra o paradigma de execução remota.

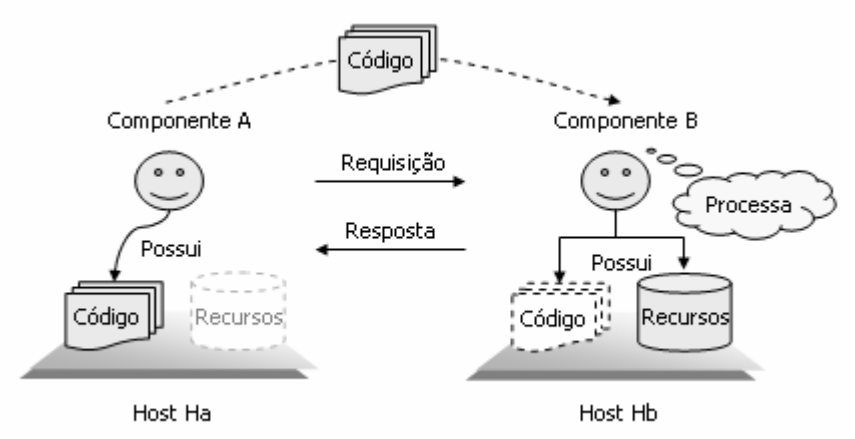

Figura 5. Paradigma de Execução Remota

No *paradigma de código sob demanda*, embora o componente A possua os recursos que necessita para processar uma tarefa, não dispõe do código que lhe permite manipulá-los. Desta forma, A solicita o código a um componente B,

localizado no *host* Hb da rede. B entrega o código ao componente A, que então executa o processamento. São exemplos do paradigma de código sob demanda: carga de arquivos *pdf* a partir do protocolo *http*; execução de *applets Java*; etc. A Figura 6 ilustra o paradigma de código sob demanda.

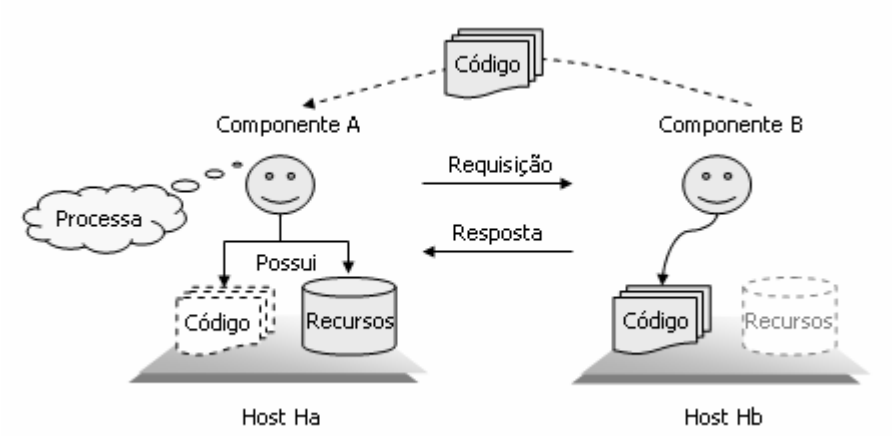

Figura 6. Paradigma de Código sob Demanda

No *paradigma de agentes móveis*, um componente de software A, hospedado inicialmente no *host* Ha, possui o código de um serviço que precisa ser processado; porém, alguns dos recursos necessários ao processamento estão disponíveis apenas em outros *hosts*, como no *host* Hb. A então migra para Hb, carregando o código e resultados intermediários de sua tarefa. O componente A termina o processamento utilizando os recursos disponíveis em Hb. Exemplos: agentes móveis em aplicações de busca de informações, comércio eletrônico, gerência de redes, computação móvel, etc. A Figura 7 ilustra o paradigma de agentes móveis.

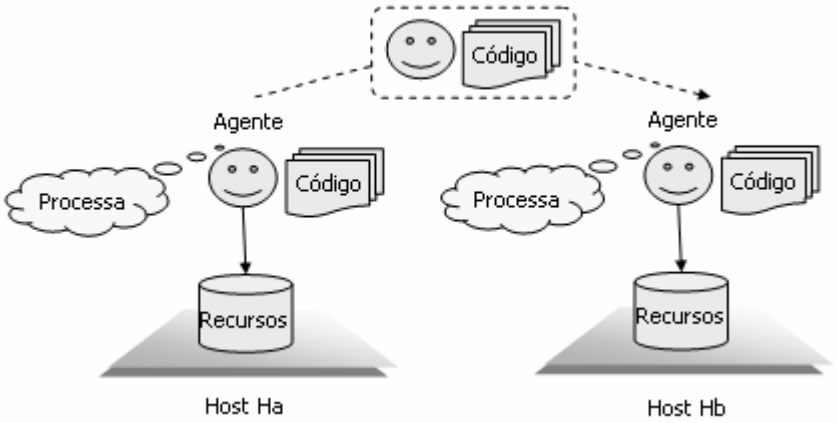

Figura 7. Paradigma de Agentes Móveis

De acordo com a descrição anterior, no paradigma cliente-servidor, os componentes de software e o código das aplicações não podem ser transferidos após sua criação. Os paradigmas de execução remota e de código sob demanda superam esta limitação fornecendo a característica de mobilidade ao código processado pelos componentes; isto é, estes paradigmas permitem a execução de código em uma localização remota. No extremo da facilidade de transferência, encontra-se o paradigma de agentes móveis que fornece mobilidade não somente ao código que deve ser processado, mas aos próprios componentes que efetuam a execução dos serviços.

A Tabela 1 esboça um paralelo das principais características dos paradigmas da computação em rede (Fuggetta et al., 1998). O paradigma de agentes móveis se diferencia dos demais paradigmas fundamentalmente porque envolve a mobilidade de um componente de software completo para o processamento de serviços. Em outras palavras, enquanto nos outros paradigmas, a facilidade geral diz respeito a uma possível transferência de código entre componentes, no paradigma de agentes móveis um componente de software completo é movido para um *host* remoto, juntamente com seu estado, código, e até mesmo alguns recursos necessários para a execução dos serviços.

Tabela 1. Comparação entre Paradigmas da Computação em Rede

Características x Paradigmas	Cliente-Servidor	Execução Remota	Código sob Demanda	Agentes Móveis
Transferência de Código	Nenhuma transferência de código.	Código é transferido do cliente para o servidor.	Código é transferido do servidor para o cliente.	Código está distribuído na rede.
Propriedade dos Recursos	Somente o servidor possui os recursos necessários ao processamento.	Somente o servidor possui os recursos necessários ao processamento.	Cliente possui os recursos necessários ao processamento.	Agentes migram para onde estão os recursos.
Controle do Processamento	Servidor controla o processamento.	Servidor controla o processamento.	Cliente controla o processamento.	Hospedeiro do agente controla processamento.
Exemplos	Exemplo: interpretação de páginas no servidor.	Exemplo: execução do comando *rsh*.	Exemplo: execução de *applets Java*.	Exemplo: aplicações de agentes móveis em geral.

2.4. Vantagens e Desvantagens de Agentes Móveis

A utilização do paradigma de agentes móveis pode conferir às aplicações de sistemas distribuídos pelo menos dois benefícios principais (Harrisson et al., 1995; White, 1995; Lange, 1998): (i) redução do tráfego de dados na rede, e (ii) execução assíncrona das aplicações.

É desejável que os dados necessários a uma aplicação sejam processados nos próprios computadores onde estão localizados. Em outras palavras, é interessante evitar que grandes volumes de dados necessários à execução de uma tarefa sejam transferidos pela rede. A solução mais adequada é transferir requisições de processamento à localização onde estão os dados remotos ao invés de transferir estes dados a um outro *host* onde possam ser processados. Os agentes móveis são, portanto, úteis *na redução do tráfego de dados em uma rede*, porque permitem que as solicitações de um aplicativo sejam transferidas junto com o agente para um *host* onde a interação entre dados e código possa ocorrer localmente (Harrisson et al., 1995; White, 1995; Lange, 1998).

Com a ascensão da Computação Móvel, o problema de desconexão dos dispositivos portáteis tornou-se um tópico de pesquisa importante, e junto com a portabilidade, a questão referente à *execução assíncrona de aplicações*. Os dispositivos portáteis freqüentemente utilizam conexões de rede frágeis e caras. Isto implica dizer que tarefas com requisição de conexão continuamente aberta entre um dispositivo móvel e uma rede fixa muito provavelmente não são econômica ou tecnicamente viáveis. Uma solução para este problema consiste em encapsular tarefas juntamente com agentes móveis e enviá-los para a rede fixa, reconectando algum tempo depois a fim de fazer retornar o agente já com os resultados encontrados. Não há necessidade de qualquer tipo de sincronização entre a conexão e o processamento das tarefas. Isto é possível porque, após a transferência, os agentes móveis tornam-se independentes da máquina original e operam assincronamente (Harrisson et al., 1995; White, 1995; Lange, 1998).

Apesar dos benefícios, a utilização do paradigma de agentes móveis também possui desvantagens potenciais. A primeira delas é a necessidade de sistemas adicionais para o suporte ao paradigma. Para o processamento de agentes móveis geralmente são utilizadas plataformas de mobilidade. O uso de uma plataforma

tem conseqüências tanto no desempenho individual de cada *host* como no aumento da complexidade das aplicações (Capítulo 3).

No entanto, o maior problema do uso de agentes móveis em sistemas diz respeito a questões de segurança. Não é razoável confiar irrestritamente em códigos provenientes de outras máquinas, sendo necessária alguma espécie de autenticação para garantir a confiabilidade do código ou restrições de acesso aos recursos das máquinas. Estes procedimentos podem impedir que um código malicioso ocasione danos aos *hosts* destinatários de agentes móveis.

Por outro lado, *hosts* destinatários também podem tratar indevidamente as informações recebidas. Por exemplo, alterando o resultado de uma computação, ou ainda, quebrando o sigilo de resultados da computação que representam informações confidenciais. Portanto, é desejável que código e dados enviados a uma máquina também sejam protegidos. Em geral, as plataformas de mobilidade não tratam devidamente o requisito de segurança (Karnik & Tripathi, 2001).

2.5. Áreas de Aplicação de SMAs Móveis

De acordo com Lange (1998), a tecnologia de agentes móveis pode ser empregada nas seguintes aplicações:

- *comércio eletrônico*: agentes móveis representam vendedores, gerentes e clientes em um mercado eletrônico. Os agentes negociam em nome de quem os enviou para a rede;
- *busca de informações*: um dos exemplos mais freqüentes, em que os agentes móveis são transferidos para fontes de informação onde podem efetuar buscas cujos resultados são retornados para o *host* original;
- *assistente pessoal*: a habilidade de migrar para *hosts* remotos torna um agente apropriado a atuar como um "assistente" que executa tarefas em nome de seu criador. Por exemplo, um usuário pode emitir seu agente pessoal a fim de interagir com agentes representativos de cada um das pessoas que devem ser convocadas a uma reunião. Os agentes podem então negociar e estabelecer uma hora apropriada para o encontro;

- *monitoramento e notificação*: uma das aplicações clássicas, porque destaca a natureza assíncrona da execução de agentes móveis. Os agentes podem ser transferidos para a rede a fim de notificar aos usuários que um determinado tipo de informação tornou-se disponível;
- *processamento paralelo*: utilização potencial da tecnologia de agentes móveis, em que tarefas podem ser executadas através da colaboração entre agentes localizados em múltiplos processadores;
- *gerenciamento de redes*: os agentes podem ser utilizados no gerenciamento de redes, uma vez que a mobilidade e a capacidade de colaboração dos agentes permitem a implementação de um gerenciamento distribuído, com tráfego de dados reduzido e capacidade de respostas rápidas a alterações ocorridas no domínio da rede.

É importante mencionar que ainda não existem aplicações em que as abordagens tradicionais da computação em rede não possam ser utilizadas (Kiniry & Zimmerman, 1997). Porém, a tecnologia de agentes pode ser empregada para aumentar a eficiência e facilitar a implementação das aplicações. O objetivo, portanto, não é substituir as abordagens tradicionais, e sim complementá-las.

2.6. Plataformas de Mobilidade

Para a abstração de SMAs móveis, é necessário identificar os conceitos e as funcionalidades recorrentes não só em aplicações baseadas em agentes, mas também em plataformas de mobilidade utilizadas por aplicações. Nas plataformas de mobilidade, são recorrentes os seguintes elementos: (i) abstrações fundamentais, (ii) ciclo de vida dos agentes e (iii) modelo de eventos. Nesta seção, as plataformas utilizadas neste trabalho são descritas a partir de seus elementos recorrentes, a fim de tornar claro o entendimento das funcionalidades tratadas pelo *framework* AspectM (Seção 6.1).

2.6.1. Aglets

Aglets Software Development Kit, ou simplesmente ASDK, é uma plataforma Java para desenvolvimento de agentes móveis, denominados *Aglets*. A plataforma *Aglets* implementa o padrão *Mobile Agent System Interoperability Facility* (MASIF) para interoperabilidade entre sistemas. Atualmente existem duas versões para a plataforma ASDK. A primeira, *Aglets Workbench*, foi desenvolvida na IBM Japão, no ano de 1996. Para o JDK 1.2 e superiores, uma versão de código-livre do *Aglets Software Development Kit*, o ASDK 2.0, ou *Aglets 2.0*, do *SourceForge*, deve ser utilizada. Durante o desenvolvimento dos estudos de caso deste trabalho foi utilizado a versão *SourceForge* do ASDK.

Aglets fornece um *framework* para implementação de agentes móveis com uma API própria. A API de *Aglets*, denominada AAPI (*Aglet Application Programming Interface*), é uma interface de programação que define os métodos necessários à execução dos *aglets*. "Agente", "Identificador", "Contexto" e "Proxy" são abstrações fundamentais da AAPI (Lange et al., 1997). A abstração "Agente" é implementada através da classe abstrata `Aglet`. De acordo com as necessidades dos usuários, um novo *aglet* pode ser criado fazendo-o herdar atributos e métodos finais desta classe ou sobrescrevendo-se outros métodos quando necessário. Trata-se, portanto, da classe base para a criação dos *aglets*.

Na AAPI, o identificador de um *aglet* é implementado através da classe `AgletID`. Este identificador é imutável e independe do contexto de execução. Em outras palavras, o identificador de um *aglet* não sofre mudanças durante todo o seu ciclo de vida e é válido para todos os *hosts* a que o *aglet* possui acesso. A abstração "Contexto" é implementada através da interface `AgletContext`.

Os agentes *Aglets*, ou simplesmente *aglets*, são objetos projetados para se mover através de *hosts*, que se caracterizam por apresentar a própria *thread* de controle e serem orientados a eventos (Lange, 1998). O ciclo de vida de um *aglet* inclui as seguintes etapas: (1) *criação* (método `createAglet()`, na interface `AgletContext`), o *aglet* recebe um identificador único e seu estado inicial é configurado, de modo que, a partir deste instante, ele está pronto para receber instruções; (2) *movimentação*, o *aglet* é enviado de um *host* local a outro *host* na rede (método `dispatch()`, na classe `Aglet`); (3) destruição (método `dispose`, na

classe `Aglet`), o *aglet* libera os recursos do ambiente que estão sendo utilizados por ele e finaliza todas as suas atividades, perdendo seu estado definitivamente.

Métodos como `dispose()` não são sobrescritos por subclasses de *Aglet*. Entretanto, existem alguns métodos que podem ser sobrescritos a fim de estabelecer comportamentos particulares para novos *aglets* sendo criados. Estes métodos são invocados automaticamente pelo sistema quando certos eventos ocorrem no ciclo de vida dos *aglets* (Lange, 1998). Isto é possível porque o modelo de programação dos *aglets* é baseado em eventos (Lange et al., 1997).

Um evento de mobilidade é gerado em função da transferência de um *aglet* para um outro *host* na rede. Este evento permite associar a *aglets* ações em instantes específicos através da sobrecarga de métodos como `onDispatching()` e `onArrival()`. Na API de *Aglets*, o responsável por tratar o evento de mobilidade é a interface `MobilityListener`. A subclasse correspondente ao *aglet* sendo instanciado deve declarar a implementação da interface `MobilityListener` a fim de que o sistema possa responder aos eventos de mobilidade.

Além de métodos referentes à mobilidade, outros exemplos importantes de métodos executados automaticamente pela plataforma são `onCreation()`e `onDisposing()`. Os métodos `onCreation()` e `onDisposing()` são executados apenas uma vez durante todo o ciclo de vida, enquanto outros métodos, como `onArrival()` e `onReverting()` podem ser executados várias vezes. A Tabela 2 apresenta eventos do ciclo de vida e métodos relacionados.

Tabela 2. Ciclo de Vida dos *Aglets* e Métodos Relacionados

Eventos	Métodos com modificador final	Métodos que podem ser sobrescritos nas subclasses	
		Invocados no momento em que ocorrem os eventos	Invocados depois que ocorrem os eventos
Criação	createAglet()		onCreation()
Envio	dispatch()	onDispatching()	onArrival()
Retorno	retractAglet()	onReverting()	onArrival()
Destruição	dispose()	onDisposing()	

De acordo com a definição da AAPI, a interface `AgletContext` oferece métodos através dos quais é possível manipular o contexto de execução dos *aglets*. Por exemplo, os métodos `createAglet()` e `retractAglet()`, que implementam, respectivamente, a criação de um agente e seu retorno a um contexto de onde foi anteriormente enviado. Além disso, um objeto `AgletContext` possibilita, através do método `getAgletProxies`, a obtenção de uma lista de todos os *proxies* de *aglets* disponíveis em um contexto. Em outras palavras, a interface `AgletContext` pode ser utilizada por um *aglet* para obter informações a respeito da localização de outros agentes.

Existe ainda um método relacionado a contextos de execução que não está disponível na interface `AgletContext` e sim na classe `Aglet`, definida no início desta seção. Trata-se do método `getAgletContext()`, que retorna o contexto do *aglet* corrente. Possuindo acesso ao contexto, um *aglet* pode, por exemplo, criar outros *aglets* ou fazer retornar *aglets* de outros *hosts* da rede para o contexto corrente. Estas operações são efetuadas pela expressão de métodos `getAgletContext().createAglet()`, para a criação de novos *aglets*, e `getAgletContext().retractAglet()`, para efetuar o retorno de *aglets*.

A interface particular de um *aglet*, que evita o acesso direto a seus métodos públicos, é denominada *proxy*. O *proxy* pode ser considerado então o representante de um *aglet*. Uma função importante do *proxy* é fornecer transparência de localização aos agentes, isto é, o *proxy* atua no sentido de ocultar a localização real de seu *aglet* correspondente. Numa situação de transferência de *aglet*, por exemplo, a comunicação com o agente continua sendo realizada da mesma maneira que antes do deslocamento. Em outras palavras, a comunicação com o *aglet* é efetuada como se estivesse ocorrendo localmente, embora nesta situação deva ser obtido um outro *proxy* para o *aglet*, correspondendo à sua interface de comunicação no *host* remoto. Utilizar um *proxy* como camada intermediária para a comunicação com *aglets*, além de fornecer transparência de localização e realizar proteção de referências, também possibilita resolver o problema de desalocação de um *aglet* pelo *garbage collector* quando outros agentes ainda fazem referências a ele (Lange, 1998).

2.6.2. JADE

Java Agent DEvelopment Framework, ou simplesmente JADE, é uma plataforma de agentes que disponibiliza um *framework* e a respectiva API orientada a objetos para desenvolvimento de aplicações em conformidade com especificações do padrão FIPA (FIPA) para interoperabilidade entre sistemas (Bellifemine et al., 1999). O objetivo de JADE é simplificar o desenvolvimento de aplicações de agentes ao mesmo tempo em que assegura um conjunto de serviços de sistema que obedecem ao padrão FIPA (Bellifemine et al., 1999). A plataforma foi desenvolvida na Universidade de Parma, na Itália. Durante o desenvolvimento dos estudos de caso deste trabalho, foi utilizada a versão 3.3 de JADE.

As classes e interfaces da API de JADE são projetadas para permitir o gerenciamento da plataforma de agentes e dos agentes individuais. Para a administração da plataforma, três elementos foram identificados como obrigatórios: o Agente de Administração do Sistema[9], que controla o acesso à plataforma sendo responsável pela autenticação e controle de inscrições de agentes; o Agente de Canal de Comunicação[10], responsável pela comunicação entre agentes internos ou externos a plataforma; o Diretório Facilitador[11], que provê um serviço de páginas amarelas para a plataforma.

Sob a perspectiva do tratamento de questões relacionadas aos interesses de mobilidade em SMAs, as abstrações mais importantes de JADE estão relacionadas aos agentes individuais. Assim como em *Aglets* (Seção 2.6.1), "Agente", "Identificador" e "Contexto" são abstrações fundamentais associadas com mobilidade. Evidentemente, existem diferenças importantes entre as plataformas de mobilidade nos detalhes de implementação referentes a estas abstrações. Estas diferenças são consideradas neste trabalho em função de suas implicações para o uso das funções de mobilidade de uma plataforma específica.

9 Do inglês: *Agent Management System*.
10 Do inglês: *Agent Communication Channel*.
11 Do inglês: *Directory Facilitator*.

Os *agentes* JADE são implementados como *threads* Java e inseridos dentro de repositórios de agentes chamados de *contêineres*, os quais fornecem um ambiente de execução para agentes executando concorrentemente. A noção de um contêiner JADE corresponde à noção de contexto de execução em sistemas multi-agentes móveis. Tal como em *Aglets*, o ciclo de vida de agentes JADE inclui as etapas de criação, movimentação e destruição de agentes.

Para criar um agente JADE, deve-se estender a classe `Agent`, implementar as tarefas da aplicação, instanciá-las e associá-las ao agente. Um contêiner JADE deve ser instanciado a partir da classe `AgentContainer`. Cada objeto `Agent Container` permite o gerenciamento local do ciclo de vida do conjunto de agentes associados a ele. Do ponto de vista do programador, um agente JADE é uma classe Java que estende a classe `Agent` da plataforma JADE. A extensão da classe `Agent` permite a utilização de um conjunto de métodos para manipulação do ciclo de vida de agentes e um conjunto de métodos abstratos para definição de comportamentos específicos das aplicações.

O identificador de um agente JADE é implementado através da classe `AID`. Este identificador possui um papel fundamental no mecanismo de comunicação, sendo usado para a especificação de emissores e receptores de mensagens. Por esta razão, uma diferença significativa em relação à plataforma *Aglets*, é o fato de não existir o conceito de "Proxy" entre as abstrações fundamentais de JADE. Alguns exemplos de métodos importantes da API de JADE são:

- `protected void setup()`: deve ser utilizado na inicialização de um agente e corresponde ao método `onCreation()` da plataforma *Aglets*;
- `public void doDelete()`: executa procedimentos referentes à destruição do agente, podendo ser chamado pela plataforma ou pelo próprio agente. Corresponde ao método `dispose()` de *Aglets*;
- `public void doMove(Location destino)`: é invocado pela plataforma ou pelo próprio agente para iniciar o processo de migração;
- `public void beforeMove()`: é invocado automaticamente pela plataforma imediatamente antes da partida do agente para outro *host*. Corresponde ao método `onDispatching()` da plataforma *Aglets*;

- `public void afterMove()`: é invocado automaticamente pela plataforma imediatamente após a chegada do agente em um novo *host*. Corresponde ao método `onArrival()` da plataforma *Aglets*.

2.6.3. Comparação entre Plataformas

Tanto em *Aglets* como em JADE, os agentes são implementados como *threads* Java e inseridos dentro de contextos de execução. Em JADE, a classe-base para a criação de agentes é a classe `Agent`; em *Aglets*, esta classe corresponde à classe `Aglet`. Em *Aglets*, o contexto de execução é implementado através da interface `AgletContext`; em JADE, um contexto de execução corresponde a um "contêiner" instanciado a partir da classe `AgentContainer`. Tanto um contexto de *Aglets* quanto um contêiner de JADE constituem um ambiente completo para a execução concorrente de agentes. Embora não haja uma equivalência entre os conceitos de "contexto" em *Aglets* e "contêiner" em JADE, para as questões de mobilidade tratadas neste trabalho, pode-se afirmar que existe uma correspondência entre estas definições.

Do ponto de vista do programador, tanto em JADE como em *Aglets*, um agente móvel é simplesmente uma instância de uma subclasse que representa o conceito de agente (`Agent` ou `Aglet`) à qual são adicionados comportamentos específicos segundo o objetivo das aplicações. Existe uma correspondência também entre os métodos disponibilizados pelas plataformas. Por exemplo, os métodos listados a seguir implementam a mesma função nas plataformas *Aglets* e JADE, respectivamente: (1) `onCreation()` e `setup()`, (2) `dispose()` e `doDelete()`, (3) `dispatch()` e `doMove()`.