Linguagens para Computação Móvel na Internet

Marco Túlio de Oliveira Valente[1,2]
Roberto da Silva Bigonha[2]
Antônio Alfredo Ferreira Loureiro[2]
Mariza Andrade da Silva Bigonha[2]

Resumo

Na década de 90, o crescimento exponencial da Internet e, em especial, da *World Wide Web*, motivou o projeto de diversas linguagens de programação para desenvolvimento de aplicações distribuídas nesta rede. Em geral, estas linguagem têm em comum o fato de suportarem alguma forma de mobilidade, seja ela de código ou de agentes. Assim, o primeiro objetivo deste artigo é traçar um panorama das principais linguagens surgidas na última década com suporte a computação móvel na Internet. São discutidas as principais abstrações incorporadas a estas linguagens e mostrados exemplos de programas móveis desenvolvidos nas mesmas. O segundo objetivo do artigo é descrever os principais conceitos e princípios do Cálculo de Ambientes, um cálculo de processos proposto recentemente com o objetivo de servir de modelo teórico para uma nova geração de linguagens para desenvolvimento de aplicações na Internet. Tais linguagens têm sido chamadas de *Wide Area Languages* (WAL).

Keywords: Computação Móvel, Internet, Linguagens de Programação

[1]Departamento de Ciência da Computação, Pontifícia Universidade Católica de Minas Gerais
[2]Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
E-mail:{mtov,bigonha,loureiro,mariza}@dcc.ufmg.br

1 Introdução

Na década de 90, o crescimento exponencial da Internet e, em especial, da *World Wide Web*, motivou o projeto de diversas linguagens de programação para desenvolvimento de aplicações nesta rede [59]. Em geral, estas aplicações tem em comum o fato de serem dotadas de alguma forma de mobilidade, seja ela de código ou de agentes.
Neste artigo, mobilidade de código designa o tipo de mobilidade existente em aplicações cujo código pode trafegar por vários nodos de uma rede antes de ser executado em alguma estação de destino [24, 51]. Java [3] é o exemplo mais conhecido de linguagem com suporte a mobilidade de código. A principal vantagem desta forma de mobilidade é o fato de viabilizar a execução de uma mesma aplicação nas várias arquiteturas e sistemas operacionais que existem em uma rede como a Internet.
Já mobilidade de agentes designa o estilo de mobilidade presente em aplicações cuja execução não precisa se restringir a uma única estação, podendo migrar de uma máquina para outra da rede [66, 34, 20, 55]. Neste caso, migra-se para a estação de destino não apenas o código da aplicação, mas também o estado corrente de sua execução. Dentre as vantagens desta forma de mobilidade, destacam-se uma redução do tráfego gerado na rede e o fato de permitir a implementação de aplicações mais robustas a flutuações na largura de banda da Internet e até mesmo capazes de operarem desconectadas da rede [38].
Este trabalho objetiva então traçar um panorama das principais linguagens surgidas na década de 90 com suporte a mobilidade de código e de agentes. São discutidas as principais abstrações incorporadas a estas linguagens para suporte a mobilidade e mostrados exemplos de programas móveis desenvolvidos nas mesmas. O restante do trabalho está organizado da seguinte forma:

- A Seção 2 trata de mobilidade de código. Inicialmente, descrevem-se as motivações que levaram à proposição de aplicações dotadas deste tipo de mobilidade e os modelos propostos para garantir segurança na execução das mesmas. Em seguida, são descritas algumas das principais linguagens que suportam mobilidade de código, incluindo Java [3] e também outras linguagens que não alcançaram a mesma aceitação desta, como Juice [23], Limbo [41] e PLAN [32]. Como trata-se de uma forma de mobilidade mais simples e conhecida, estas linguagens são descritas de forma mais resumida.

- A Seção 3 trata de mobilidade de agentes. Inicialmente, relacionam-se alguns trabalhos da área de sistemas distribuídos que inspiraram a proposição deste estilo de mobilidade e as principais vantagens que o mesmo propicia. Em seguida, descrevem-se algumas linguagens que suportam mobilidade de agentes, iniciando por Obliq [10], considerada a precursora deste tipo de linguagem. Em seguida, descreve-se a linguagem Telescript [66], a qual foi pioneira na definição das construções existentes em tais linguagem. Por último, descreve-se a biblioteca

Aglets [37], desenvolvida com o objetivo de permitir a implementação de agentes móveis em Java.

- A Seção 4 inicia descrevendo o π-cálculo [44], um cálculo de processos que inspirou o desenvolvimento do Cálculo de Ambientes [16]. Este cálculo, descrito a seguir nesta seção, é um formalismo proposto recentemente com o objetivo de servir de base teórica para uma nova geração de linguagens para a Internet, chamadas de *Wide Area Languages* (WAL) [13].

- A Seção 5 descreve-se as principais características e construções que uma linguagem deve possuir para ser classificada como uma WAL.

- Por último, a Seção 6 apresenta algumas considerações finais e o Apêndice A lista os endereços *Web* das linguagens descritas neste artigo.

2 Mobilidade de Código

2.1 Introdução

O termo mobilidade de código é usado para caracterizar programas que trafegam por uma rede de estações heterogêneas, eventualmente cruzando diferentes domínios administrativos, e que são automaticamente executados ao atingirem uma determinada estação de destino [24, 59, 51]. Neste trabalho, considera-se que nesta forma de mobilidade o que trafega pela rede é apenas o código do programa, esteja ele no formato fonte ou em um formato intermediário. Este estilo de mobilidade é também chamado de mobilidade de código fraca [19]. Quando o que trafega pela rede não é somente o código da aplicação mas também o estado corrente de sua execução, usaremos neste trabalho o termo mobilidade de estado ou mobilidade de agentes.
O conceito de mobilidade de código, na verdade, não é novo em linguagens de programação. A linguagem Postscript [2], usada para descrição de serviços de impressão, é um bom exemplo. Postscript é uma linguagem de pilha cujos programas são automaticamente gerados por um *software* de editoração e então enviados para a impressora, onde ocorre a interpretação dos mesmos. Esta interpretação tem como efeito colateral a impressão do documento desejado.
No entanto, somente após o crescimento exponencial da Internet é que mobilidade de código ganhou maior destaque. O motivo é o fato de se tratar de uma solução natural para lidar com a heterogeneidade típica da rede e também, como pioneiramente proposto por Java, para permitir a construção de aplicações interativas na *Web*. Mobilidade de código contribui ainda para simplificar a tarefa de administração e manutenção de redes, pois viabiliza a instalação de aplicações apenas em servidores, enquanto permite o acesso dos clientes às mesmas por demanda. Com isso, simplifica-se bastante a instalação, configuração e atualização de aplicações. Permite-se também

uma maior segurança no uso das mesmas, impedindo a instalação de *software* não licenciado e a proliferação de vírus.
Linguagens e ambientes de programação com suporte a mobilidade de código devem satisfazer particularmente aos seguintes requisitos:

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

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

O restante desta seção encontra-se organizado da seguinte forma: na Subseção 2.2 relacionam-se as técnicas já propostas para segurança de código móvel e na Subseção 2.3 descrevem-se as principais linguagens que suportam a implementação de aplicações móveis.

2.2 Modelos de Segurança para Código Móvel

Existem basicamente quatro modelos para prover segurança na execução de código móvel: *sandbox*, assinatura de código (*code signing*), *firewall* e código portador de prova (*proof carrying code*) [56].
A idéia por trás do modelo *sandbox*, usado em Java, é confinar a execução do código móvel de tal forma que seja impossível a produção de qualquer dano à estação cliente. A implementação desta técnica em Java é descrita com maiores detalhes na Seção 2.3.1.
Já no modelo de assinatura de código, o cliente possui localmente uma lista de entidades certificadoras nas quais confia. Ao receber uma aplicação móvel para execução, ele então verifica se a mesma possui uma assinatura digital certificada por uma das entidades de sua lista. Normalmente, esta verificação utiliza um algoritmo de criptografia baseado em chaves públicas e privadas. Caso tenha sua assinatura verificada, a aplicação é executada com os mesmos privilégios do usuário que solicitou sua transferência. Caso contrário, a aplicação tem sua execução negada. Assinatura de código é empregada, por exemplo, no sistema Authenticode da Microsoft [42], o qual é usado para validar a execução de controles ActiveX. Mais recentemente, Java também passou a permitir a inclusão de assinaturas digitais em *applets*. Na versão 1.1 do JDK, se for carregada para execução uma *applet* que contenha uma assinatura certificada

por uma fonte confiável ao cliente, a mesma é executada como se fosse uma aplicação local, tendo acesso a todos os recursos do ambiente de execução, como sistema de arquivos e conexões de rede. Caso contrário, a execução da *applet* ocorre segundo o modelo *sandbox* tradicional.
Observe que o grau de liberdade oferecido a uma aplicação móvel nos métodos *sandbox* e de assinatura de código é exatamente o oposto um do outro. *Sandbox* pressume que todas as *applets* são potencialmente perigosas e, por isso mesmo, não permite quase nenhuma interação das mesmas com o ambiente local. Já no método de assinatura de código, uma vez reconhecida a assinatura digital de uma aplicação móvel, ela passa a ter os mesmos privilégios de uma aplicação normal.
Na terceira alternativa para segurança, chamada de *firewall*, a aplicação móvel tem sua execução liberada ou negada assim que entra na rede do cliente. Para isto, esta técnica incorpora aos sistemas tradicionais de *firewall* uma funcionalidade para examinar o código de toda *applet* buscada na rede e então decidir se a mesma será enviada ao cliente para execução ou não. O grande desafio deste modelo, portanto, é determinar com precisão se a execução de uma *applet* é segura. Apesar da dificuldade desta tarefa, já existem produtos comerciais que, a princípio, implementam soluções deste tipo [56].
Código Portador de Prova (PCC) [46] é uma técnica que permite verificar estaticamente se uma aplicação móvel atende a uma política de segurança previamente estabelecida pelos seus clientes, chamados no método de consumidores de código. Esta política de segurança é divulgada usualmente na forma de sentenças em lógica de primeira ordem. Cabe ao nodo de origem do código, chamado de produtor de código, criar uma prova formal atestando o fato de que seu código atende a política de segurança. O consumidor de código, ao receber a aplicação móvel, executa então uma validação desta prova, a qual é transmitida junto com a aplicação. Caso a prova seja válida, a aplicação tem sua execução liberada. Dentre as vantagens desta técnica inclui-se o fato de transferir grande parte do ônus da tarefa de garantir segurança para o produtor de uma aplicação móvel, melhorando assim o desempenho dos consumidores de código. PCC tem ainda as vantagens de realizar a verificação do código estaticamente, de não utilizar criptografia e de ser à prova de violações no código durante sua transmissão pela rede. No entanto, a fim de que a técnica seja usada na prática, ainda restam alguns problemas a serem resolvidos. Provavelmente, o maior deles é o desenvolvimento de um "compilador certificador", capaz de produzir provas automaticamente como parte do processo de compilação [39].

2.3 Linguagens com Suporte a Mobilidade de Código

2.3.1 Java

Dentre as linguagens que suportam mobilidade de código, certamente Java [28, 3], lançada oficialmente pela Sun em 1995, foi a que alcançou maior sucesso. Con-

tribuíram para este fato uma bem articulada estratégia de *marketing* e sua integração com a *Web*, através da possibilidade de se embutir aplicações Java, chamadas *applets*, em páginas HTML. Java é uma linguagem orientada por objetos baseada em C++, porém sem diversos dos excessos que caracterizam esta última. A linguagem é fortemente tipada e possui herança simples, semântica de referência, coleta automática de lixo, tratamento de exceções, herança de interface, recursos para construção de módulos, suporte nativo a programação concorrente e reflexividade.

Programas fonte em Java são compilados para uma linguagem intermediária de pilha chamada *bytecode*. A fim de garantir independência de plataforma, esta linguagem intermediária é então interpretada por uma Máquina Virtual Java (JVM) [40], a qual possui implementações para diversas arquiteturas. A JVM pode ser ainda integrada a um *browser*, viabilizando deste modo o desenvolvimento de aplicações *Web* interativas. Ressalte-se que a utilização de arquiteturas hipotéticas com o intuito de obter portabilidade não constitui novidade na implementação de linguagens de programação. No início da década de 70, esta mesma estratégia foi usada, por exemplo, na implementação do compilador Pascal-P. Este compilador gerava código para uma linguagem intermediária chamada P-Code, a qual era interpretada por uma máquina virtual denominada SC (*Stack Computer*) [47].

A fim de garantir a execução segura de *applets*, a infra-estrutura de execução de programas Java possui, além da JVM, três outros componentes: o Carregador de Classes (*Class Loader*), o Verificador de *Bytecode* (*Bytecode Verifier*) e o Gerenciador de Segurança (*Security Manager*) [29, 56]. O funcionamento destes componentes é ilustrado na Figura 1. O Carregador de Classes é chamado em tempo de execução pela JVM quando o código a ser interpretado referencia um nome de classe não resolvido. Cabe a ele disponibilizar o código desta classe, seja carregando-o do sistema de arquivos local, seja transferindo-o de uma outra estação da rede. Uma vez carregado, este código é então submetido ao Verificador de *Bytecode*, o qual testa se o mesmo encontra-se no formato correto e verifica a existência de erros mais complexos, como violação dos mecanismos de controle de visibilidade de membros de classes, *overflow* ou *underflow* na pilha de operandos da JVM, chamada de métodos com parâmetros incorretos e conversões ilegais de tipos, dentre outros [69]. Já o Gerenciador de Segurança controla, em tempo de execução, o acesso a recursos como o sistema de arquivos local ou a rede.

Para construção de aplicações cliente/servidor na Internet, Java oferece ainda a biblioteca RMI (*Remote Method Invocation*) [58], a qual permite que aplicações Java chamem métodos de objetos localizados em JVM remotas, isto é, introduz mobilidade de controle na linguagem. Aplicações baseadas em Java RMI são constituídas por dois tipos de programas: servidor e cliente. O servidor tipicamente cria alguns objetos e torna referências para os mesmos disponíveis na rede, geralmente através do registro destes objetos em um serviço de nomes. Feito isso, o servidor passa a aguardar requisições dos clientes. Estes obtém, através de consultas ao serviço de

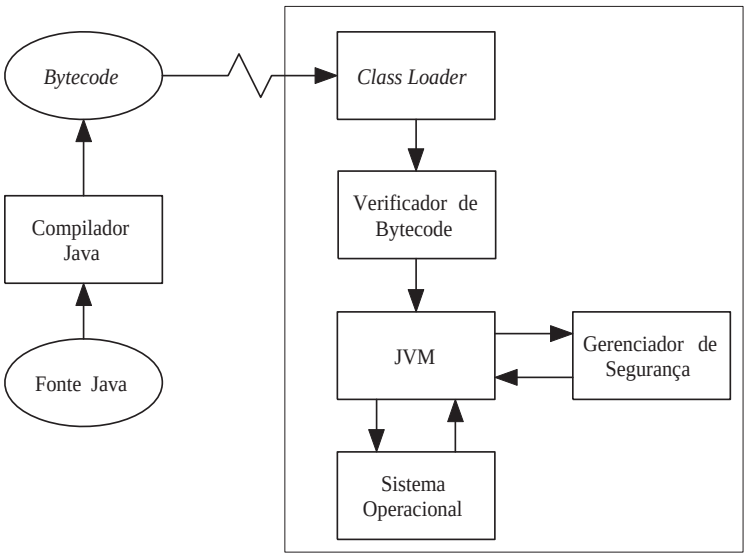

Figura 1: Infra-estrutura de Execução de *Applets* Java

nomes, referências para objetos remotos e então invocam seus métodos remotamente. RMI permite ainda que objetos sejam passados como parâmetros, característica esta que o torna mais flexível que os sistemas tradicionais de RPC, onde geralmente permite-se somente a passagem de valores de tipos básicos como parâmetros. Para tanto, Java RMI serializa o estado dos objetos de forma a transmiti-los pela rede. Além dos dados de um objeto, o código de seus métodos também é enviado dinamicamente para a JVM remota. Estes recursos de RMI para passagem de objetos como parâmetros e chamada remota de métodos são usados, por exemplo, na implementação de sistemas de agentes móveis em Java, como será mostrado na Seção 3. Por último, Java RMI implementa ainda coleta de lixo distribuída, envolvendo objetos remotos que não são mais referenciados por nenhum cliente.
Existe também disponível em Java uma série de outras bibliotecas para realização de tarefas como construção de interfaces gráficas, acesso a banco de dados, implementação de componentes de *software*, construção de aplicações multimídia, criptografia, comércio eletrônico etc.

2.3.2 Juice

O sistema Juice [23] foi proposto como uma alternativa a Java para geração de código móvel na Internet. Em Juice, utiliza-se Oberon como linguagem fonte. Além disso,

o sistema gera um formato de código portável chamado de *slim binary*, o qual é uma linguagem intermediária em forma de árvore que apresenta algumas vantagens em relação a uma representação linearizada como os *bytecodes*. A primeira delas é que trata-se de um formato mais adequado a realização de otimizações no código produzido. Uma segunda vantagem de *slim binary* é o fato de possuir uma taxa de compactação bem superior à conseguida com *bytecode*. Esta vantagem é especialmente importante em aplicações móveis, visto que as mesmas são transmitidas por uma rede antes de terem sua execução de fato iniciada. Em média, um código no formato de *bytecode*, mesmo quando compactado, é cerca de 50% maior que o mesmo código em *slim binary*. Dentre as desvantagens desta representação, temos o fato dela ser inadequada para interpretação e, portanto, não recomendada para uso em sistemas embutidos e em produtos eletrônicos.
Após a transferência da aplicação para a máquina onde será executada, ocorre um processo de compilação de *slim binary* para código nativo. A idéia é que o tempo economizado na transmissão pela rede devido a compactação seja utilizado para geração de código. Além disso, o processo de compilação é incremental, isto é, a primeira compilação é rápida, sem otimizações, de forma a obter rapidamente um código executável, evitando assim que o usuário aguarde um tempo maior antes que a aplicação tenha sua execução iniciada. Em seguida, tem início um processo de recompilação contínua em *background*, processo este que ocorre durante os ciclos ociosos da máquina. Como esta recompilação ocorre em *background*, torna-se viável o emprego de técnicas mais agressivas e demoradas de otimização de código. Além disso, as recompilações são subsidiadas por um *profile* da execução corrente, o qual também é útil em algumas formas de otimização.
Uma comparação entre Juice e Java é mostrada na Tabela 1:

	Juice	Java
Linguagem	Oberon	Java
Formato de distribuição	*Slim binary*	*Bytecode*
Execução	Geração de código nativo *on the fly*	Interpretação via JVM

Tabela 1: Comparação entre Juice e Java

2.3.3 Limbo

Limbo [41] é a linguagem preferencialmente utilizada para desenvolvimento de aplicações no sistema operacional Inferno*, desenvolvido pelos Laboratórios Bell e comercializado pela Lucent. Inferno é um sistema operacional de rede projetado para ser

*O nome é uma referência ao poema *A Divina Comédia*, do poeta italiano Dante Alighieri, o qual é dividido em três partes: *Inferno*, *Purgatório* e *Paraíso*. Limbo, na teologia católica, é o lugar onde

utilizado em produtos eletrônicos, como telefones celulares, assistentes pessoais, vídeo games, receptores de TV a cabo etc. Em seu desenvolvimento, visou-se obter um sistema operacional que fosse portável entre várias arquiteturas, versátil o suficiente para suportar o desenvolvimento de diversos tipos de aplicações e que demandasse pouco uso de recursos computacionais, como memória e *hardware*. Portabilidade no sistema é conseguida tanto entre diversos processadores, como Intel e Risc, como também pela possibilidade de se executar Inferno em outros sistemas operacionais, como Windows e Unix. Além disso, os aplicativos do sistema são preferencialmente desenvolvidos na linguagem Limbo, cujos programas possuem a mesma representação binária em todas as plataformas onde Inferno é executável.
Limbo é uma linguagem com sintaxe similar a C, no que tange a expressões e estruturas de controle, e a Pascal, no que tange a declarações. A linguagem é fortemente tipada, possuindo ponteiros, porém com uso mais restrito que em C, coleta automática de lixo, *threads* para programação concorrente e canais, inspirados em CSP, para comunicação entre processos. A linguagem não possui recursos de orientação por objetos, mas oferece o conceito de módulos para implementação de tipos abstratos de dados. Em Limbo, módulos são carregados dinamicamente.
Programas em Limbo são compilados para uma máquina virtual, chamada Dis. Diferentemente da JVM, que é uma máquina de pilha, Dis é uma arquitetura com registradores de propósito geral e instruções do tipo memória-memória. O fato de Dis possuir uma arquitetura mais próxima dos processadores reais facilita a compilação de seus programas para código nativo. Normalmente, esta tarefa é realizada pelos chamados compiladores *just in time*.

2.3.4 PLAN

A linguagem PLAN (*Packet Language for Active Networks*) [32], projetada na Universidade da Pensilvânia, tem como objetivo o desenvolvimento de programas para redes de computadores ativas, isto é, redes cujos roteadores são capazes de executar programas armazenados nos pacotes de dados que recebem da rede. Nas redes tradicionais, os roteadores são passivos, isto é, eles simplesmente analisam o cabeçalho de cada um dos pacotes que recebem e então decidem o destino que será dado aos mesmos. Portanto, os cabeçalhos destes pacotes podem ser vistos como sendo equivalentes a programas em uma linguagem bastante primitiva definida na especificação do protocolo de rede. Já em uma rede ativa, tais cabeçalhos são efetivamente constituídos por programas codificados em uma determinada linguagem de programação. Estes programas são interpretados em cada um dos roteadores por onde o pacote trafega. Redes ativas constituem, portanto, uma solução que torna simples a incorporação de novos serviços e funcionalidades a uma rede e que, deste modo, elimina

se encontram as almas das crianças que, embora não tivessem nenhuma culpa pessoal, morreram sem o batismo que as livraria do pecado original.

os transtornos que sempre ocorrem em uma rede tradicional quando se faz necessária uma atualização de protocolo.
PLAN é uma linguagem de programação desenvolvida especificamente com o objetivo de ser utilizada nos programas que serão executados em uma rede ativa. Trata-se de uma linguagem inspirada em ML, mas sem diversos dos conceitos de mais alto nível típicos de linguagens funcionais, como casamento de padrões, funções recursivas, funções de ordem superior e polimorfismo.
Mostra-se abaixo um programa PLAN que simula o comando *ping*, usado para diagnóstico de redes. Nas redes passivas, esta aplicação insere na rede um pacote de um protocolo específico, chamado ICMP [54]. Já em uma rede ativa, uma aplicação como *ping* pode ser codificada da seguinte maneira:

```
fun ping (src: host, dest: host): unit =
    if (thisHost <> dest) then
      onRemote (ping (src, dest), dest)
    else onRemote (ack(), src)

fun ack(): unit = print ("sucesso")
```

Em uma rede ativa, existe então uma aplicação que "injeta" um pacote com o programa acima em um nodo de origem da rede, onde é executada pela primeira vez a função *ping*. Supondo que o nodo de destino seja diferente deste nodo de origem, será então chamada a função *onRemote*, a qual possui a seguinte sintaxe: *OnRemote (I, evalDest)*[†]. O resultado desta função é a criação de um novo pacote que irá avaliar remotamente no nodo *evalDest* a função *I*. Veja, portanto, que o programa acima irá avaliar a função *ping* no nodo de destino. Neste nodo, será então novamente chamada a função *OnRemote*, desta vez para avaliar a função *ack* no nodo de origem, indicando assim o sucesso da operação.
Programas em PLAN podem ainda chamar *rotinas de serviço*, codificadas em linguagens mais poderosas e que são dinamicamente carregadas nos roteadores. Existem também recursos para limitar a quantidade de CPU, memória e banda de rede que um pacote pode consumir.
PLAN permite o desenvolvimento de programas com um grau de autonomia maior que Java, por exemplo. Como a linguagem possui primitivas para avaliação remota de funções, o programador em PLAN tem controle sobre a mobilidade de suas aplicações. Já em Java, o ambiente de execução de uma *applet* é determinado não pelo programador, mas sim pelo usuário da mesma. Neste sentido, PLAN é similar às linguagens com suporte a mobilidade de agentes, descritas na Seção 3.2. No entanto, como em PLAN dispõe-se apenas de mobilidade de código, optou-se por descrever a linguagem nesta seção.

[†]Na verdade, a função *OnRemote* possui dois outros parâmetros que especificam a rota a ser utilizada e a quantidade de recursos que será alocada ao novo pacote. No entanto, para simplificar o exemplo, estes dois parâmetros foram omitidos.

3 Mobilidade de Agentes

3.1 Introdução

Em um sentido bastante amplo, um agente é qualquer programa que realiza uma tarefa para a qual recebeu delegação de um usuário [8, 48]. É um termo normalmente empregado em diversas áreas da computação, como Inteligência Artificial, Robótica, Bancos de Dados e Recuperação de Informação, dentre outras [5]. Já agentes móveis são agentes cuja execução não necessariamente se restringe a uma única máquina, isto é, um agente móvel é um programa que autonomamente se desloca pelas diversas máquinas de uma rede a fim de melhor realizar a tarefa que lhe foi delegada [66, 34, 20, 55]. Quando se move de um nodo para outro da rede, o agente móvel carrega consigo não só o seu código mas também o estado corrente de sua execução. Esta forma de mobilidade é chamada de mobilidade de agentes, mobilidade de estado ou mobilidade de código forte [19]. Neste trabalho, adota-se a primeira denominação.
Algumas das características principais de agentes móveis são: habilidade para interagir e cooperar com outros agentes, autonomia no sentido de que sua execução procede com nenhuma ou pouca intervenção da entidade que o disparou, execução em diferentes plataformas de *hardware* e *software* (interoperabiliade), capacidade de responder a eventos externos (reatividade) e capacidade de se mover de uma estação para outra da rede (mobilidade) [55].
Inicialmente nesta introdução, descreve-se na Seção 3.1.1 os modelos de comunicação nos quais a idéia de agentes móveis se baseou. Na Seção 3.1.2 relacionam-se as principais vantagens e áreas de aplicação de agentes móveis. Em seguida, a Seção 3.2 apresenta as principais propostas de linguagens e sistemas para desenvolvimento de agentes móveis.

3.1.1 Trabalhos Relacionados

Para melhor entender as razões que motivaram a proposição do modelo de agentes móveis como uma alternativa para implementação de aplicações distribuídas na Internet é importante conhecer os principais modelos de comunicação propostos para o desenvolvimento de sistemas distribuídos.
Dentre esses modelos, o mais utilizado é o Modelo Cliente/Servidor. Nesse modelo, as aplicações são estruturadas como um conjunto de processos cliente e servidor que interagem através de troca de mensagens. Essas podem ser síncronas ou assíncronas. No caso de troca de mensagens síncronas, o processo cliente, após enviar uma requisição ao servidor, tem sua execução bloqueada até receber uma resposta. Já no caso de mensagens assíncronas, o cliente pode prosseguir sua execução logo após ter enviado uma mensagem. Mesmo sendo conceitualmente bastante simples, o Modelo Cliente/Servidor é adequado a uma grande variedade de sistemas. Prova disso é que as aplicações tradicionais da Internet, como WWW, *mail*, *ftp* e *telnet*, são todas

baseadas no mesmo.
As primeiras implementações de sistemas Cliente/Servidor tinham, no entanto, como desvantagem o fato de utilizarem primitivas de comunicação de muito baixo nível, como *send* e *receive*, as quais não fazem parte das construções que um programador tradicional de sistemas centralizados está habituado a utilizar. Para sanar este problema, foi proposto então o recurso de Chamada Remota de Procedimento (RPC) [7], com o qual permite-se a um processo cliente chamar um procedimento localizado em uma outra máquina da rede. Fica a cargo da implementação de RPC na máquina cliente capturar a chamada ao procedimento remoto, encapsular seus parâmetros em mensagens de rede (em um processo denominado *parameter marshalling*) e então enviar estas mensagens para a máquina remota. A implementação de RPC na máquina remota recebe estas mensagens, recupera os parâmetros (processo este chamado de *parameter unmarshalling*) e então chama localmente o procedimento. Uma vez terminada a execução do mesmo, acontece o processo inverso para enviar o resultado até a máquina cliente.
Posteriormente, CORBA (*Common Object Request Broker Architecture*) [64] foi proposto como uma padronização e adaptação do modelo RPC para linguagens orientadas por objeto. Sendo um padrão, CORBA permite interoperabilidade entre aplicações desenvolvidas em diferentes linguagens e sistemas operacionais. CORBA propõe ainda o conceito de *Object Request Broker* (ORB), o qual constitui uma espécie de "barramento lógico" que conecta os diversos programas de uma aplicação distribuída com o intuito de prover transparência na localização e manipulação de objetos.
Um outro modelo de comunicação proposto para sistemas distribuídos é o de Avaliação Remota (REV) [57], o qual pode ser visto como uma generalização do conceito de RPC que permite também a passagem de procedimentos como parâmetros. Estes procedimentos são então avaliados remotamente, vindo daí o nome do modelo. Uma primeira vantagem de REV é sua flexibilidade, pois não se define previamente um conjunto de serviços que podem ser invocados remotamente, como acontece em RPC. Uma outra vantagem é que, para algumas aplicações, REV pode reduzir o montante de comunicação necessário à realização de uma tarefa. Esta redução é obtida graças à possibilidade de se enviar o programa que manipula um conjunto de dados até a máquina onde os mesmos estão localizados, enquanto que em RPC, por exemplo, os dados é que devem obrigatoriamente serem movidos para a máquina onde encontra-se o programa.
O modelo de agentes móveis pode ser visto como uma evolução dos modelos RPC e REV, como mostrado na Figura 2. Neste modelo, um agente migra de uma máquina a outra da rede carregando consigo seus dados, como em RPC, e o seu código, como em REV. No entanto, ao contrário destes dois modelos, um agente é autônomo, não precisando retornar imediatamente ao nodo de origem após ter sua execução finalizada no primeiro nodo que visita. Ele pode, sempre com o objetivo de executar sua tarefa, migrar por outros nodos da rede antes de retornar ao nodo de origem.

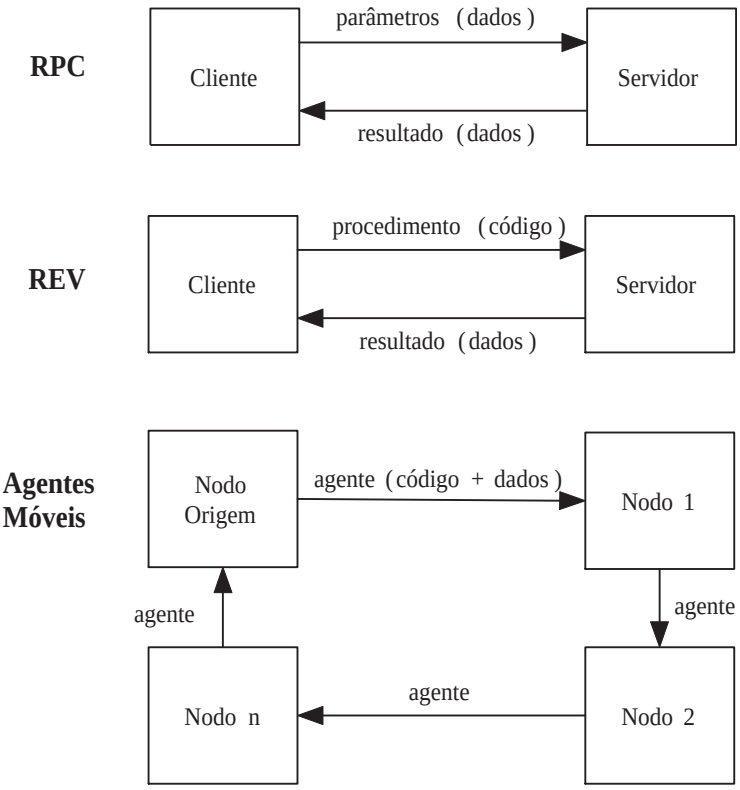

Figura 2: Comparação dos Modelos RPC, REV e Agentes Móveis

O modelo de agentes móveis, além de ser uma evolução de RPC e REV, é ainda inspirado no modelo de objetos móveis e no conceito de migração de processos existente em alguns sistemas operacionais distribuídos [27].
O modelo de objetos móveis surgiu a partir da disseminação das idéias de orientação por objetos na década de 80. Nesse modelo, permite-se o envio de objetos de uma máquina para outra da rede. Diversas linguagens foram então propostas para suportar o conceito de objetos móveis, como Emerald, Distributed Smalltalk e Cantor [4]. Dentre elas, Emerald [33] foi uma das que alcançou maior aceitação. No entanto, ao contrário do modelo de agentes móveis, o uso de Emerald é recomendado apenas em redes locais com um número modesto de máquinas, todas elas possuindo a mesma arquitetura.
Já o conceito de migração de processos prevê a possibilidade de transferência de pro-

cessos de uma máquina para outra da rede, tendo como principais objetivos balanceamento de carga e tolerância a falhas. Diferentemente de agentes móveis, no entanto, a decisão de quais processos, quando e para onde mover fica a cargo do sistema operacional. Na verdade, o processo em si não tem conhecimento de sua localização corrente, a qual também não influi na semântica de sua execução. Normalmente, requer-se ainda que todas as máquinas do sistema tenham a mesma arquitetura.

3.1.2 Vantagens e Aplicações do Modelo de Agentes Móveis

Argumenta-se que o modelo de agentes móveis é adequado ao desenvolvimento de aplicações distribuídas na Internet principalmente pelas seguintes razões [38]:

- Redução de tráfego na rede: agentes móveis permitem que toda comunicação entre dois nodos seja "empacotada" no nodo de origem e então enviada ao nodo de destino. Por exemplo, quando grandes volumes de dados são armazenados em nodos remotos, pode-se enviar um agente até o mesmo em vez de se transferir todos os dados até o nodo de origem, como ocorre no modelo cliente/servidor. Esta vantagem é ilustrada na Figura 3.

- Eliminação da latência da rede: agentes móveis permitem que uma computação seja deslocada para junto dos recursos que irá manipular ou controlar, evitando assim que sua execução seja impactada por eventuais atrasos causados por congestionamentos na rede. Como exemplo, agentes móveis podem ser usados em uma aplicação de controle de robôs em uma linha de montagem. Normalmente, esta aplicação é de tempo real e, portanto, não pode estar sujeita à latência da rede.

- Encapsulamento de protocolos: agentes móveis dispensam a implementação em todos os clientes de uma aplicação distribuída de protocolos previamente acordados para envio e recebimento de dados. Normalmente, a existência deste tipo de código nos clientes dificulta bastante a atualização destes protocolos de comunicação.

- Execução assíncrona e autônoma: uma vez disparados, agentes móveis tornam-se independentes do nodo de origem, passando a operar de forma assíncrona e autônoma. Esta vantagem é particularmente importante no caso de dispositivos computacionais móveis, como *notebooks* e assistentes pessoais, onde não é técnica e economicamente viável a suposição de conectividade contínua.

- Adaptação dinâmica: agentes móveis podem se valer de sua capacidade de mobilidade para se adaptarem a mudanças em seu ambiente de execução. Como exemplo, um agente móvel executando em um assistente pessoal pode migrar para uma estação da rede fixa ao detectar que a bateria do assistente não dispõe de energia suficiente para o término da execução de sua tarefa.

- Execução em diversas arquiteturas: como usualmente as redes de computadores são formadas por máquinas de diferentes arquiteturas e sistemas operacionais, agentes móveis constituem uma solução natural para interoperação e integração de aplicações neste tipo de ambiente.
- Robustez e tolerância a falhas: a capacidade de adaptação dinâmica de agentes móveis, contribui para simplificar a construção de sistemas distribuídos robustos e tolerantes a falhas. Por exemplo, ao se realizar um *shutdown* em uma máquina, os agentes em execução na mesma podem migrar para outra máquina, evitando assim que o processamento seja interrompido.

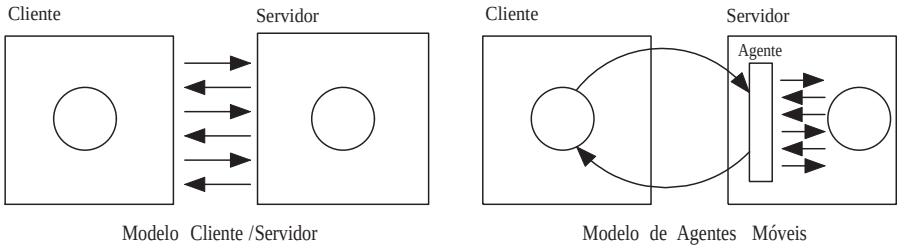

Figura 3: Tráfego Gerado na Rede pelos Modelos C/S e de Agentes Móveis

Atualmente, diversas aplicações vêm sendo relacionadas como beneficiárias diretas deste novo modelo de comunicação. Dentre elas, podemos citar Comércio Eletrônico, Recuperação de Informação, Gerência de Redes de Telecomunicações, Aplicações de *Workflow* e *Groupware* e Processamento Paralelo [38].

3.2 Linguagens com Suporte a Mobilidade de Agentes

3.2.1 Obliq

A linguagem Obliq [10, 11], desenvolvida nos laboratórios da DEC no início da década de 90, provê suporte para construção de aplicações distribuídas baseadas no paradigma de execução remota. Obliq é uma linguagem interpretada, com escopo léxico, não tipada e que possui apenas conceitos primitivos de orientação por objetos. O seu projeto teve como objetivo o desenvolvimento de uma linguagem onde recursos distribuídos pudessem ser acessados com um grau de transparência similar ao existente em linguagens convencionais para acesso a recursos locais.
Aplicações desenvolvidas em Obliq são estruturadas como um conjunto de objetos. Não há classes, herança nem chamada dinâmica de métodos. Um objeto para representar, por exemplo, um ponto no plano é definido da seguinte forma:

```
{ x => 10, y => 20,
  move => meth (self, x1, y1) ... end
}
```

Sobre objetos podem ser executadas quatro operações básicas: seleção/invocação, atualização/redefinição, cópia e redirecionamento (*aliasing*). A semântica das três primeiras é a usual de linguagens orientadas por objeto. Já a quarta operação possui a seguinte sintaxe: **redirect** a **to** b **end**. O seu resultado é o redirecionamento de todas operações futuras realizadas sobre atributos de *a* para atributos similares do objeto *b*.
Para tratar acesso concorrente a objetos, a linguagem oferece o conceito de *objeto serializado*. Um objeto é dito serializado quando, em cada momento da execução do programa, no máximo uma *thread* pode acessar seus atributos ou executar seus métodos. Este tipo de objeto é implementado associando-se implicitamente um *mutex* ao mesmo, o qual é adquirido ao se invocar um de seus métodos e liberado quando o método retorna. A fim de evitar *deadlocks*, invocações de um método a partir de um outro método do mesmo objeto não são sujeitas a travamentos.
Em Obliq, referências e procedimentos podem ser livremente transmitidos de um nodo para outro da rede. No caso específico de procedimentos, a principal novidade introduzida na linguagem em relação a outros mecanismos de execução remota é o fato de identificadores livres de procedimentos passados como parâmetros permaneceram associados a suas localizações de origem, conforme previsto nas regras de escopo léxico. Na definição da linguagem, argumenta-se que esta abordagem contribui para tornar a semântica de uma computação independente do nodo da rede onde ocorre sua execução.
Obliq oferece, portanto, mobilidade de *closures*. Na linguagem, um *closure* é definido como sendo um par que contém o texto fonte de uma computação e uma tabela de identificadores livres.
O exemplo abaixo mostra a exportação e registro de um objeto *obj* de um nodo *A* em um servidor de nomes chamado *Namer*. No nodo *B*, obtém-se uma referência de rede *r* para este objeto através de uma consulta ao servidor de nomes. Esta referência pode então ser transparentemente manipulada, podendo, por exemplo, ser usada para realizar uma invocação remota de um método *m1* de *obj*.

```
Site A:    net_export ("myObj", Namer, obj);

Site B:    let r= net_import ("myObj", Namer);
           r.m1 (b);
```

No próximo exemplo, o *site A* exporta um objeto com um método *rexec* e o *site B* importa este mesmo objeto. Em seguida, invoca-se remotamente em *B* o método *rexec*, passando como parâmetro um procedimento *proc*, como mostrado abaixo:

```
Site A:    var replay = proc() end;
           net_export ("ComputeServer", Namer,
           { rexec => meth (s,p) replay:= p; p(); end });

Site B:    let computeSever= net_import ("ComputeServer", Namer);
           var x = 0;
           computeServer.rexec (proc () x:= x+1 end );
```

A execução de *rexec* no *site A* armazena o procedimento *proc* recebido como parâmetro em uma variável global *replay* e então executa *proc*. Como x é um identificador livre em *proc*, pelas regras de escopo léxico e distribuído de Obliq, ele sempre permanecerá associado ao seu *site* de origem B, como mostrado na Figura 4. Assim, a primeira invocação de *rexec* incrementará o valor de x para 1. Veja ainda que uma chamada posterior a *replay* em A fará com que x seja novamente incrementado, sempre no seu *site* de origem B.

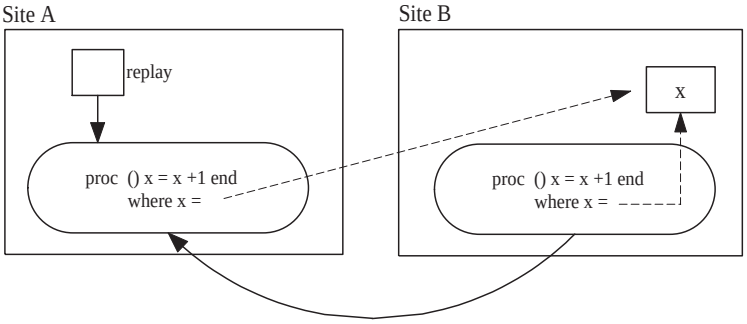

Figura 4: Avaliação Remota com Escopo Léxico em Obliq

Existe ainda a possibilidade de disponibilizar em um nodo um serviço para execução remota de procedimentos (e não de métodos, como nos dois exemplos anteriores). Este tipo de serviço é ativado pela seguinte rotina:

```
net_exportEngine (service_name, Namer, arg);
```

onde *service_name* é o nome dado ao serviço de execução remota, *Namer* é o servidor de nomes onde o serviço será registrado e *arg* é um argumento que será automaticamente fornecido a todos os procedimentos recebidos para execução. Este argumento permite, por exemplo, que estes procedimentos tenham acesso a recursos locais da máquina onde serão executados.
No exemplo abaixo, disponibiliza-se no *site A* um serviço de execução remota onde

os procedimentos recebidos para avaliação terão como argumento uma referência de nome *database*, a qual denota um banco de dados existente nesta máquina.

```
Site A:    net_exportEngine ("DBServer", Namer, database);
```

Abaixo mostra-se como uma máquina cliente B envia um procedimento para execução em A. No exemplo, o procedimento passado como parâmetro irá criar em A um objeto com métodos *start*, *report* e *stop*, os quais utilizarão o parâmetro formal *database* para acessar a base de dados. Deve ser observado que o cliente B fornece apenas o código do procedimento a ser executado em A. Já o argumento associado ao parâmetro formal *database* será automaticamente provido pelo serviço *DBServer* em execução na máquina A.

```
Site B:    let db= net_importEngine ("DBServer", Namer);
           db ( proc (database)
                { start => meth ... end,
                  report => meth ... end,
                  stop => meth ... end }
                end);
```

Migração de objetos não é um conceito primitivo em Obliq. No entanto, pode ser implementada enviando um procedimento para execução remota, o qual cria no *site* de destino uma cópia do objeto local e retorna uma referência para a mesma. Em seguida, no *site* de origem, deve-se então executar uma operação de redirecionamento dos atributos do objeto para sua cópia remota.
A implementação de Obliq baseou-se na biblioteca de Modula-3 chamada Network Objects[6], a qual suporta o desenvolvimento de aplicações distribuídas baseadas em invocação remota de métodos. Posteriormente, foi proposta também uma versão de Obliq para desenvolvimento de aplicações gráficas utilizando o modelo de agentes móveis, chamada Visual Obliq [5].

Comentários Finais: Obliq pode ser considerada a precursora das atuais linguagens para desenvolvimento de agentes móveis. A principal novidade introduzida pela linguagem é o conceito de escopo léxico distribuído, o qual dá origem a um modelo de programação bastante flexível e poderoso. No entanto, este mesmo conceito pode dificultar a depuração de aplicações ou até mesmo permitir a introdução de falhas de segurança nas mesmas, como mostrado no exemplo da variável global *replay*. Apesar de não analisado em sua definição, é razoável supor que aplicações Obliq geram um grande tráfego na rede, tanto para implementação de escopo léxico distribuído como para realização de coleta de lixo distribuída [59].

3.2.2 Telescript

O sistema Telescript [66, 25], desenvolvido pela General Magic em meados da década de 90, foi pioneiro na proposição do modelo de agentes móveis como uma alternativa para o desenvolvimento de aplicações distribuídas. O seu projeto teve como objetivo disponibilizar comercialmente uma infra-estrutura de *software* que induzisse ao desenvolvimento de aplicações para comércio eletrônico baseadas em agentes móveis. Telescript utiliza uma série de metáforas do mundo real para se referir a uma aplicação distribuída. A rede, em Telescript, é vista como sendo constituída por um conjunto de *places*, os quais constituem localizações virtuais capazes de receber e executar agentes móveis. Estes, por sua vez, são aplicações desenvolvidas na linguagem do sistema e com a capacidade de transferir a sua execução (código e estado) de um *place* para outro da rede. Um *engine* é o *software* responsável por disponibilizar *places* em máquinas físicas da rede e por executar os agentes que encontram-se localizados nestes *places*. A Figura 5 mostra a arquitetura de uma aplicação Telescript baseada em agentes móveis.

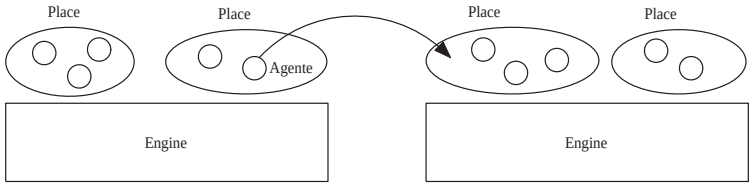

Figura 5: Arquitetura de uma Aplicação Telescript baseada em Agentes Móveis

A linguagem de programação do sistema Telescript é inspirada em C++ e Smalltalk, possuindo classes, herança múltipla e tratamento de exceções. Possui ainda uma biblioteca de classes destinadas a programação de agentes móveis. Já um *engine*, além de possuir um interpretador para esta linguagem, é responsável também por gerenciar e controlar o acesso a recursos da máquina hospedeira e pela interação com outros *engines* para enviar e receber agentes móveis.
Em Telescript, a autoridade (*authority*) responsável por um agente ou *place* é o usuário ou organização que o mesmo representa. Uma região (*region*) é uma coleção de *places* que operam sob a mesma autoridade. Uma permissão (*permit*) especifica as capacidades de um agente, como, por exemplo, seu tempo de vida, o máximo de memória que pode ocupar, o máximo de CPU que pode consumir etc. Um agente somente pode executar uma operação em um *place* se suas permissões, as permissões deste *place* e as permissões de sua região assim o permitirem.
Para viajar de um *place* a outro, um agente deve executar o comando *go*, o qual tem como parâmetro um *ticket* que especifica o destino da transferência. Dois agentes

podem ainda utilizar a instrução *meet* para programarem um encontro em um determinado *place*. Durante o encontro, um agente pode chamar procedimentos do outro e vice-versa.
Mostra-se a seguir um exemplo de aplicação em Telescript, a qual é descrita com detalhes em [66]. No exemplo, um agente "viaja" até um *place* que representa uma loja virtual e verifica o preço de um determinado produto nesta loja. Caso o mesmo seja inferior a um dado valor, o agente retorna a seu *place* de origem e comunica este fato ao usuário que o enviou. Caso contrário, o agente permanece na loja virtual aguardando que o preço do produto seja reduzido até o valor esperado ou até o fim de um intervalo de tempo especificado pelo usuário. A seguir, mostra-se o código das principais partes desta aplicação.
A classe *Warehouse* abaixo representa a loja virtual.

```
Warehouse:  class(Place,EventProcess)=
(
  public
    see initialize
    see live
    see getCatalog
  property
    catalog:  Dictionary[String, CatalogEntry];
);
```

Warehouse é uma subclasse das classes pré-definidas *Place* e *EventProcess* e possui três métodos públicos (*initialize*, *live* e *getCatalog*) e uma propriedade (*catalog*). Propriedade é o nome dado em Telescript a um atributo privado.
Dos três métodos da classe *Warehouse*, o mais importante é o método *live*. Métodos com este nome são automaticamente invocados pelo *engine* ao se criar um *place*, sendo executados como uma nova *thread*. Quando esta *thread* termina, o *engine* considera que o *place* não será mais utilizado e o remove do ambiente. No exemplo abaixo, o método *live* consiste em um laço infinito que sempre ao primeira dia de cada mês reduz o preço de todos os produtos da loja em 5%, notificando, em seguida, a ocorrência de tal evento aos agentes localizados no *place*.

```
live:  sponsored op (cause:  Exception|Nil) =
{
    loop {
        time:= Time();
        calendarTime:= time.asCalendarTime();
        calendarTime.month:= calendarTime.month+1;
        calendarTime.day:= 1;
        *.wait (calendarTime.asTime().interval(time));
            // espera primeiro dia do mês
        for product:  String in catalog
```

```
            // reduz todos os preços em 5%
        {
            try { catalog[product].adjustPrice(-5) }
            catch KeyInvalid
        };
        // notifica agentes da redução de preços
        *.signalEvent(PriceReduction(),'occupants)
    }
};
```

A classe *Shopper*, mostrada abaixo, implementa o agente que irá visitar a loja virtual.

```
Shopper:  class(Agent,EventProcess)=
{
  public
    see initialize
    see live
  private
    see goShopping
    see goHome
  property
    clientName:  Telename;
    desiredProduct:  String;
    desiredPrice, actualPrice:  Integer;
    exception:  Exception|Nil;
};
```

O método *live* é executado quando o *engine* cria um agente e, assim como ocorre na criação de *places*, constitui uma nova *thread*. Segue abaixo a implementação deste método.

```
live:  sponsored op (cause:  Exception|Nil) =
{
    homeName:= here.name;                       // Nome e endereço deste place
    homeAddress:= here.address ;
    permit:= Permit(                            // Regras da "viagem"
        (if *.permit.age == nil {nil}           // Duração:  90% tempo de vida
        else {(*.permit.age * 90).quotient(100)}),
        (if *.permit.charges == nil {nil}       // CPU: 90% do total disponível
        else {(*.permit.charges * 90).quotient(100)}
    );
    restrict permit {
        try {*.goShopping(Warehouse.name)}      // Visita loja virtual
        catch e:Exception { exception= e }
    }
```

```
    catch e:PermitViolated { exception= e } ;
    try {*.goHome(homeName, homeAddress)}      // Retorna ao place de origem
    catch Exception { }
};
```

Este método inicialmente salva o nome e endereço da localização corrente do agente, chamados em Telescript de *telename* e *teleaddress*. Em seguida, cria uma permissão que deverá ser obedecida durante toda a viagem. Esta permissão limita o tempo e consumo de CPU da viagem a 90% do total alocado ao agente, ou seja, reserva-se 10% do tempo e CPU disponíveis para que o agente possa retornar ao *place* de origem e notificar o resultado da viagem a seu cliente. A viagem e o retorno propriamente ditos são codificados, respectivamente, nos métodos *goShopping* e *goHome*. Mostra-se abaixo o código do método *goShopping*.

```
goShopping:  op (warehouse:  ClassName) throws ProductUnavailable =
{
   *.go (Ticket (nil, nil, warehouse));
   .........                  // habilita notificação do evento PriceReduction
   actualPrice:= desiredPrice + 1;
   while (actualPrice > desiredPrice) {
      *.getEvent(nil, PriceReduction());        // Aguarda redução de preço
      try { actualPrice:= here@Warehouse.getCatalog()[desiredProduct].price }
      catch KeyInvalid { throw ProductUnavailable() }
   }
};
```

Neste método, a instrução *go* transfere o agente para o *place* associado à loja virtual. Uma vez concluída esta transferência, a execução do agente prossegue na linha seguinte ao *go*. O agente então aguarda sucessivamente a ocorrência de eventos de redução de preço. Ocorrendo um evento deste tipo, chama-se o método *getCatalog* definido no *place* corrente do agente. Este método retorna o novo preço do produto que o agente está pesquisando. Quando o preço for inferior ao valor esperado, a execução de *goShopping* termina.

Comentários Finais: Telescript tem o mérito de ter sido o primeiro sistema proposto para desenvolvimento de agentes móveis, servindo inclusive de inspiração para diversos outros sistemas que surgiram em seguida. O sistema oferece recursos bastante completos para migração de agentes, para segurança e para controle de acesso a recursos. No entanto, Telescript não foi comercialmente bem sucedido, principalmente pelo fato de exigir o aprendizado de uma nova linguagem de programação em uma época em que as atenções do mercado estavam voltadas para a linguagem Java. Por este motivo, o seu projeto foi cancelado pela General Magic, que em seguida lançou o sistema Odyssey [26], baseado em Java.

3.2.3 Java Aglets

Aglets [37] é uma biblioteca de classes Java, desenvolvida pela IBM do Japão, para implementação de agentes móveis. O nome *aglet*, uma fusão das palavras *agent* e *applet*, designa nesta biblioteca um objeto Java capaz de se mover de um nodo para outro da Internet, carregando consigo seus atributos e métodos. A biblioteca é totalmente desenvolvida em Java, não exigindo nenhuma alteração na linguagem ou na JVM.
Além da popularidade alcançada pela linguagem, diversas características de Java recomendam sua utilização na implementação de uma biblioteca como Aglets. Dentre elas, podem ser citadas as seguintes: independência de plataforma, execução segura, carregamento dinâmico de classes, suporte nativo a programação concorrente, serialização de objetos e reflexividade. No entanto, o uso de Java também apresenta algumas desvantagens. A primeira delas é a ausência na linguagem de mecanismos que permitam controlar o acesso a recursos por parte de agentes. Assim, um agente desenvolvido em Java pode utilizar deliberadamente recursos como CPU ou memória. Além disso, a implementação atual da JVM, por motivos de segurança, permite apenas que os atributos de um objeto sejam salvos durante o processo de serialização. Informações como o valor corrente do contador de programa ou os dados armazenados na pilha não são salvos durante uma serialização. Assim, não se consegue restaurar completamente o estado de um agente quando este migra para um novo nodo da rede. Particularmente, nos sistemas para desenvolvimento de agentes móveis baseados em Java, todas as *threads* associadas a um agente são terminadas ao se efetuar uma migração.
Os principais conceitos implementados na biblioteca Aglets são os seguintes:

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

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

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

Aglets são criados através das operações *create* e *clone*. Já para encerrar a execução de um *aglet* existe a operação *dispose*. A migração de um *aglet* para um outro contexto pode ser ativa ou passiva. Na modalidade de migração ativa, suportada pela operação *dispatch*, o próprio *aglet* decide migrar o estado corrente de sua execução para um

novo contexto. Já na migração passiva, implementada através da operação *retract*, um contexto requisita o retorno de um *aglet* remoto. Além disso, a operação *deactivate* permite que a execução de um *aglet* seja temporariamente interrompida e seu estado salvo em disco. Posteriormente, a operação *activate* possibilita que esta execução seja retomada. A Figura 6 descreve o ciclo de vida de um *aglet*.

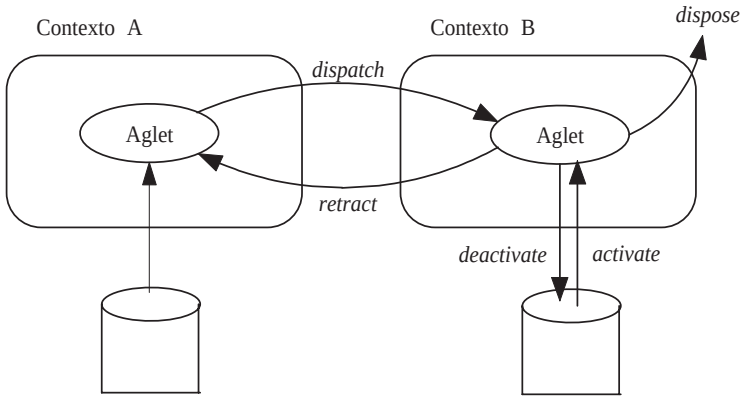

Figura 6: Ciclo de Vida de um *Aglet*

Antes de cada evento ocorrido sobre um *aglet*, é invocado um método do tipo *callback*, o qual permite ao programador especificar uma ação a ser executada neste momento. Por exemplo, antes que um *aglet* seja enviado para um novo contexto, é chamado o método *onDispatch()*. No novo contexto, a execução é retomada executando-se o método *onArrival()*. Assim, grande parte da programação em Aglets consiste na implementação de métodos *callback*.
Descreve-se a seguir um exemplo de *aglet*. Inicialmente, mostra-se a implementação da classe *MyFirstAglet*, a qual é uma subclasse da classe pré-definida Aglet.

```
import com.ibm.aglet.*;
public class MyFirstAglet extends Aglet {
    public void onCreation (Object init) {
      .........
      dispatch (new URL("atp://some.host.com/context"));
    }
    .........
}
```

O método *onCreation*, chamado logo após a criação de um *aglet*, é usado para inicializar seus atributos. No exemplo, este método executa a operação *dispatch* para

mover o *aglet* corrente para um novo contexto. A identificação de um contexto é definida como sendo uma URL contendo o nome do *host*, seguido então pelo próprio nome do contexto. O protocolo a ser utilizado na transferência é chamado de ATP (*Agent Transfer Protocol*), definido pela IBM para transporte de *aglets*.
A criação propriamente dita de um *aglet*, isto é, de um objeto da classe *MyFirstAglet*, é realizada da seguinte maneira:

```
AgletContext context= getAgletContext();
context.createAglet (null, "MyFirstAglet", null);
```

No código acima, obtém-se inicialmente uma referência para o contexto corrente, chamando para tanto a função *getAgletContext()*, que foi herdada da classe Aglet. De posse desta referência, chama-se então a função *createAglet*, definida em um contexto para criação de *aglets*. O segundo parâmetro desta função especifica o nome da classe do *aglet* que será instanciado.
Para interagir entre si, *aglets* podem trocar mensagens, as quais são definidas como sendo objetos da classe *Message*, como mostrado abaixo:

```
Message myName = new Message ("my name", "john");
Message yourName = new Message ("your name?");
```

O primeiro argumento do construtor da classe *Message* é uma *string* que define o tipo (ou nome) da mensagem. O segundo argumento é uma *string* opcional que permite associar um conteúdo à mensagem. Mensagens são tratadas em um *aglet* pelo método *handleMessage*, o qual retorna *true* se a mensagem foi tratada e falso, caso contrário. Mostra-se abaixo um exemplo de método *handleMessage*:

```
public boolean handleMessage (Message msg) {
    if (msg.somekind ("my name")) {         // Processa msg dotipo "my name"
      String name= (String) msg.getArg();  // Obtém conteúdo da msg
      return true ;
    } else if (msg.somekind("your name?") {  // Processa msg tipo "your name?"
      msg.sendReply("mary");                 // Envia resposta
      return true;
    } else return false;
}
```

Mensagens são enviadas não a um *aglet* diretamente, mas a seu *proxy*. Uma referência para o *proxy* de um *aglet* é retornada na criação do mesmo pela operação *createAglet*. Pode-se também obter o *proxy* de um *aglet* através da função *getAgletProxy*, fornecendo como argumento o identificador do *aglet*. Mostra-se abaixo um exemplo de envio de mensagens.

```
AgletContext context= getAgletContext();
```

```
AgletProxy proxy= context.getAgletProxy (id);
    // "id" é o identificador do aglet
proxy.sendMessage (myName);
    // Envia msg do tipo "my name"
String name= (String)proxy.sendMessage (yourName);
    // Envia msg do tipo "your name?" e espera resposta
```

A função *sendMessage*, usada no exemplo acima, permite o envio de mensagens síncronas. Para o envio de mensagens assíncronas, Aglets oferece a função *sendFutureMessage*. A biblioteca oferece ainda funções para alterar a ordem de tratamento das mensagens, permitindo, por exemplo, o estabelecimento de prioridades e o processamento paralelo de mensagens. É possível também a definição de mensagens de *multicast*.

Comentários Finais: Aglets foi um dos primeiros sistemas para desenvolvimento de agentes móveis a utilizar Java. Assim, beneficiou-se diretamente da popularidade de Java, mas também herdou alguns pontos negativos, como a ausência de mecanismos para controle de recursos e para migração completa de estado. As funções de segurança do sistema são também bastante primitivas. Não há como, por exemplo, evitar que qualquer contexto execute uma operação *retract* sobre um agente remoto. Atualmente, Aglets é utilizado pela IBM no *site* TabiCan[‡], uma loja virtual para venda de passagens aéreas e pacotes de turismo.

3.2.4 Outras Linguagens

Além de Obliq, Telescript e Aglets, já foram propostos diversos outros sistemas, linguagens e bibliotecas para desenvolvimento de agentes móveis. A exemplo de Aglets, a tendência atual é que estes ambientes sejam implementados como uma biblioteca de classes Java, aproveitando assim a aceitação desta linguagem [67]. Como regra geral, todos sistemas que utilizam Java são baseados nos recursos da linguagem para serialização de objetos e invocação remota de métodos. Assim, não há diferenças fundamentais de arquitetura entre eles. Como exemplo de sistemas que utilizam Java, temos Odyssey [26], Concordia [68], μ-Code [51], JavaSeal [9], Voyager [49] e Ajanta [60].
Existem também sistemas baseados em outras linguagens, como Facile [36] e Tcl [50]. Dentre os sistemas baseados em Tcl, um dos mais conhecidos é D'Agents [31], desenvolvido no Dartmouth College, e anteriormente chamado de Agent Tcl [30]. D'Agents utiliza um interpretador modificado de Tcl para executar *scripts* contendo o código de agentes. Ao contrário do que acontece em Java, este interpretador possibilita a captura completa do estado de execução de um agente, incluindo o estado de *threads* pertencentes ao mesmo. Além disso, migra-se no sistema o código fonte de um agente

[‡]A URL é: www.tabican.ne.jp. No entanto, grande parte do *site* encontra-se em japonês.

e não um código compilado, como em Java. D'Agents oferece ainda recursos para controlar e proteger o acesso de agentes a recursos, para comunicação entre agentes via troca de mensagens e para autenticação e criptografia de agentes, neste caso através de programas externos. Recentemente, o sistema passou a suportar também as linguagens Java e Scheme, além de Tcl.

4 Modelos Teóricos

4.1 π-Cálculo

O π-cálculo [45, 43, 44] é uma extensão do CCS proposta por Milner cuja principal novidade é a introdução da noção de mobilidade na especificação de sistemas concorrentes. Em π-cálculo, a exemplo do CCS, as duas únicas entidades disponíveis são processos e canais. Todas expressões denotam processos e computações são realizadas meramente através da troca de mensagens em canais[§]. Diferentemente do CCS, no entanto, em π-cálculo um processo pode criar novos canais e enviar canais através de canais, isto é, dispõe-se de mobilidade de canais. Apesar de bastante simples, estes conceitos possuem um poder de expressão suficiente para descrever uma grande variedade de sistemas concorrentes e, ao mesmo tempo, preservam a capacidade de realização de raciocínios e provas formais sobre os mesmos [52].
Processos em π-cálculo são construídos de acordo com a seguinte sintaxe:

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

A Tabela 2 descreve o significado das expressões usadas na gramática acima.
A semântica operacional do π-cálculo é definida através de reduções. Diz-se que P reduz em Q, isto é, $P \longrightarrow Q$, se P contém um conjunto de subprocessos que *podem* interagir e então se transformar nos subprocessos constituintes de Q. Veja que a semântica é não-determinística, pois especifica-se o que *pode* acontecer durante a evolução de um processo e não o que *deve* acontecer. As reduções possíveis são as seguintes:

$$\begin{array}{rcll} \overline{x}y.P \mid x(z).Q & \longrightarrow & P \mid [y/z]Q & \\ P \mid R & \longrightarrow & Q \mid R & \text{se } P \to Q \\ (\nu x)P & \longrightarrow & (\nu x)Q & \text{se } P \to Q \\ P & \longrightarrow & Q & \text{se } P \equiv P' \to Q' \equiv Q \end{array}$$

A última destas reduções requer a existência de uma congruência estrutural entre dois processos. Esta relação, denotada por $\equiv$, define um conjunto de regras para reescrita

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

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

Tabela 2: Expressões disponíveis em π-cálculo

e/ou simplificação de expressões com o propósito de viabilizar a aplicação de reduções sobre as mesmas. Como são regras que não alteram o comportamento dos processos, elas podem ser aplicadas infinitas vezes. No π-cálculo, definem-se as seguintes regras de congruência estrutural:

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

Existe desenvolvida para o π-cálculo toda uma teoria algébrica sobre equivalência de processos, cujo estudo ultrapassa os objetivos deste trabalho. Existe também a linguagem de programação Pict [53], cujo modelo computacional é totalmente baseado no π-cálculo.

Comentários Finais: O π-cálculo é um modelo pequeno, simples e elegante para especificação de diversos tipos de sistemas concorrentes. Recentemente, surgiram algumas propostas defendendo sua utilização na especificação de sistemas móveis. Tais propostas partem do pressuposto de que o conjunto de canais de um processo é um indicativo de sua localização e que, portanto, mobilidade pode ser descrita como uma mudança neste número de canais. No entanto, essa definição não reflete o conceito de mobilidade de estado tal como existente em linguagens como Obliq e Telescript. Nestas linguagens, mobilidade relaciona-se com uma mudança no ambiente de execução dos processos e não no número de canais de comunicação dos mesmos. Portanto,

o uso de π-cálculo é recomendado principalmente na especificação de sistemas com uma topologia de conexões dinâmica, como é o caso, por exemplo, de aplicações para dispositivos computacionais móveis.

4.2 Cálculo de Ambientes

O Cálculo de Ambientes [13, 16] é um cálculo de processos inspirado no π-cálculo que tem como objetivo a especificação de dois tipos distintos de mobilidade: *mobile computation* e *mobile computing*. *Mobile computation* designa o estilo de mobilidade descrito anteriormente neste trabalho, isto é, mobilidade de *software*, seja ela mobilidade de código ou de agentes. Já *mobile computing* refere-se ao tipo de mobilidade oferecido por dispositivos computacionais móveis, como *notebooks* e assistentes pessoais.
Um ambiente é um local delimitado em cujo interior acontecem computações. Todo ambiente possui um nome e uma coleção de processos e subambientes. Um ambiente pode ainda se mover para dentro ou para fora de outro ambiente. No Cálculo de Ambientes, portanto, mobilidade está relacionada com a noção de cruzar fronteiras, as quais delimitam ambientes, sendo que estes por sua vez estão organizados de forma hierárquica.
A sintaxe do cálculo, bastante similar à do π-cálculo, é a seguinte:

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

$$M \quad ::= \quad x \mid n \mid in\ M \mid out\ M \mid open\ M \mid M.M \mid \varepsilon$$

Nesta gramática, P e Q denotam processos e n, x e M referem-se, respectivamente, a nomes, variáveis e capacidades.
Um ambiente é denotado por $n[P]$, onde n é o nome do ambiente e P é o processo em execução no seu interior. O processo P, que pode ser resultado da composição paralela de diversos outros processos, permanece em execução mesmo durante a movimentação do ambiente a que pertence. O cálculo oferece ainda operações para alterar a estrutura hierárquica dos ambientes, as quais têm sua aplicação restringida por *capacidades*. A notação $M.P$ denota um processo que executa uma ação regulada pela capacidade M e então prossegue sua execução como se fosse o processo P. Existem três tipos de capacidade: para entrar (*in*), sair (*out*) e abrir (*open*) um ambiente.
A capacidade *in* m, usada em uma ação da forma *in* $m.P$, instrui o ambiente que a cerca a entrar em um ambiente vizinho de nome m, conforme ilustrado na Figura 7. Se não existir tal ambiente, a operação fica bloqueada até que o mesmo passe a existir. Se existir mais de um ambiente com este nome, um deles é escolhindo aleatoriamente. A redução associada a esta capacidade é definida pela seguinte regra:

$$n[in\ m.P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$$

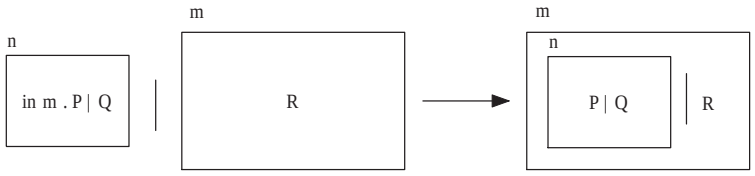

Figura 7: Redução *in*

A capacidade *out m*, usada em ações da forma *out m.P*, instrui o ambiente que a cerca a sair de seu ambiente pai, cujo nome deve ser m, conforme ilustrado na Figura 8 Se tal ambiente não se chama m, a operação fica bloqueada até que o nome do mesmo passe a ser este. A redução associada a esta capacidade é definida pela seguinte regra:

$$m[n[out\ m.P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$$

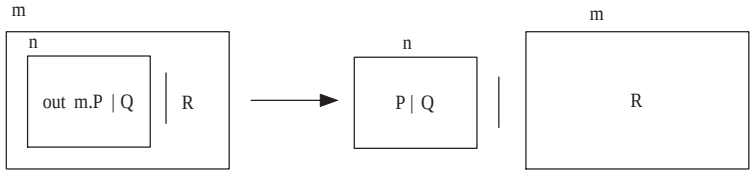

Figura 8: Redução *out*

Por fim, a capacidade *open m*, usada em uma ação da forma *open m.P*, abre as fronteiras do ambiente vizinho de nome m, conforme ilustrado na Figura 9. Como nas ações anteriores, a operação fica bloqueada até que exista um ambiente vizinho com este nome e, no caso de existir diversos ambientes com o mesmo nome, um deles é aleatoriamente escolhido. A redução associada a esta capacidade é definida pela seguinte regra:

$$open\ n.P \mid n[Q] \longrightarrow P \mid Q$$

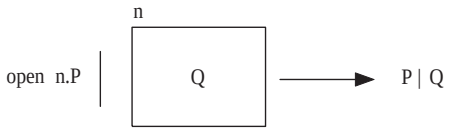

Figura 9: Redução *open*

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

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

Como exemplo de uso do Cálculo de Ambientes, mostra-se abaixo a especificação do acesso de um agente móvel a um *firewall*.

$$\begin{array}{lcl} \mathit{Agente} & \triangleq & k'[\mathit{open}\ k.k''[Q]] \\ \mathit{Firewall} & \triangleq & (\nu\ w)w[k[\mathit{out}\ w.\mathit{in}\ k'.\mathit{in}\ w] \mid \mathit{open}\ k'.\mathit{open}\ k''.P] \end{array}$$

Neste exemplo, supõe-se que o acesso ao *firewall* é controlado por uma senha k', a qual é também o nome de um ambiente que envolve o agente móvel. O *firewall*, por sua vez, é representado por um ambiente com um nome secreto w. Para guiar o acesso do agente móvel, o *firewall* cria o seguinte "ambiente guia": $k[\mathit{out}\ w.\mathit{in}\ k'.\mathit{in}\ w]$. Veja que este ambiente sai do *firewall* ($\mathit{out}\ w$) e então entra no agente k' ($\mathit{in}\ k'$). Se o agente não conhecesse a senha k', isto é, se não existisse um ambiente envolvendo o agente móvel com este nome, a ação $\mathit{in}\ k'$ ficaria bloqueada e, portanto, o acesso do agente ao *firewall* seria negado.
Uma vez "dentro" do agente móvel, o "ambiente guia" é aberto pela ação $\mathit{open}\ k$, permitindo assim que a ação $\mathit{in}\ w$ seja executada para transportar o agente para dentro do *firewall*. Uma vez no interior do *firewall*, as ações $\mathit{open}\ k'.\mathit{open}\ k''.P$ dissolvem os ambientes intermediários k e k'', possibilitando que os processos P (do *firewall*) e Q (do agente) ganhem execução. O ambiente intermediário k'' somente foi utilizado para evitar que o processo Q interferisse na execução deste protocolo de acesso ao *firewall*.
Atualmente, existe todo um esforço de pesquisa no sentido de desenvolver uma teoria de equivalência de processos para o Cálculo de Ambientes similar à existente para o π-cálculo. O objetivo é permitir a realização de raciocínios formais sobre ambientes. Em [17] propõe-se um sistema de tipos para controlar os valores que podem

ser trocados pelas primitivas de comunicação no interior de ambientes. O objetivo é garantir que estes valores são do tipo correto. Já em [15] propõe-se um sistema de tipos para controlar a migração e abertura de ambientes. A proposta apresentada permite classificar ambientes como móveis ou imóveis e como bloqueados (*locked*) ou não bloqueados (*unlocked*). Um ambiente bloqueado não pode ser aberto. Mais recentemente, em [18] propõe-se uma lógica modal que permite expressar propriedades espaciais e temporais de ambientes.
Já foi implementado também um interpretador, chamado Ambit, para uma linguagem contendo as primitivas básicas do Cálculo de Ambientes [12]. Este interpretador viabiliza a execução de pequenas especificações realizadas neste formalismo. Existe também uma "versão gráfica" do Cálculo de Ambientes, chamada Cálculo de Pastas (*Folder Calculus*) [13], desenvolvida com o objetivo de facilitar o entendimento do método. Para isso, o Cálculo de Pastas utiliza uma série de metáforas relacionadas com objetos normalmente encontrados em um escritório, como pastas, borrachas, copiadoras etc.

Comentários Finais: Tendo sido projetado após o crescimento da Internet, o Cálculo de Ambientes captura com precisão a noção de computação móvel nesta rede. O cálculo parte do pressuposto de que existem diversas localizações na *Web*, sendo algumas delas virtuais e outras móveis. Localizações virtuais são erguidas, por exemplo, para proteger domínios administrativos. Já localizações móveis correspondem a dispositivos computacionais conectados, por exemplo, a uma rede sem fio. Como estas localizações possuem propriedades e recursos diferentes, é de se esperar que as novas aplicações para a Internet sejam capazes de se mover entre elas. Todas estas noções são expressas com fidelidade no Cálculo de Ambientes. Finalizando esta seção, a Tabela 3 mostra uma comparação entre o π-cálculo e o Cálculo de Ambientes.

	π-cálculo	Cálculo de Ambientes
Entidades	processos e canais	processos e ambientes
Nomes	denotam canais	denotam ambientes
Unidade de Mobilidade	canais	ambientes
Modelo Computacional	comunicação em canais	mobilidade de ambientes
Comunicação	síncrona	assíncrona e local a ambientes
Localização	número de canais do processo	fronteiras de ambientes
Principais Aplicações	sistemas com uma topologia de conexões dinâmica	*mobile computing* e *mobile computation*

Tabela 3: Comparação das Principais Características do π-cálculo e do Cálculo de Ambientes

4.3 Outros Modelos Teóricos

Além do π-cálculo e do Cálculo de Ambientes, existem outras propostas de formalismos para expressar mobilidade na Internet. O *Join*-Cálculo [21] é uma extensão do π-cálculo que introduz uma noção explícita de localização. Já o *Join*-Cálculo Distribuído [22] permite a associação de nomes a localizações, além de introduzir a noção de falhas distribuídas. O π-cálculo também possui extensões com o objetivo de garantir segurança nas comunicações, como o spi-Cálculo [1] e o Cálculo *Seal* [65].
Existe ainda uma proposta para introdução de mobilidade de estado no método conhecido como Máquinas de Estado Abstratas (ASM) [63, 62]. Basicamente esta proposta procura tirar proveito do fato da noção de estado ser precisamente definida em ASM. Além disso, ela introduz no método uma operação *move*, similar às instruções para migração de agentes existentes nas linguagens descritas na Seção 3.2.
Embora não vise a especificação de nenhuma das formas de mobilidade apresentadas neste trabalho, um outro formalismo que merece citação são os Combinadores de Serviço [14]. Este formalismo oferece um conjunto de operadores para acesso e manipulação de páginas *Web*. O objetivo é a especificação de aplicações que simulem o comportamento de usuários diante de um *browser*, incluindo a especificação de estratégias normalmente usadas para tratar erros e para contornar a lentidão da rede. Combinadores de Serviço inspiraram o projeto da linguagem WebL [35], voltada ao desenvolvimento de aplicações que manipulam páginas *Web*, como meta-ferramentas de busca e agentes de pesquisa de preço.

5 *Wide Area Languages*

Chama-se de *Wide Area Language* (WAL) uma linguagem que possui construções semanticamente compatíveis com os princípios do Cálculo de Ambientes e, por conseqüência, de uma rede com as características da Internet [13]. Dentre estes princípios, os mais importantes são os seguintes:

- Completeza: uma WAL deve possuir poder de expressão suficiente para permitir a implementação de programas rotineiramente encontrados na Internet, como *applets*, *plug ins*, agentes móveis etc.
- Consistência: uma WAL *não* deve possuir primitivas que pressuponham *action-at-a-distance* ou conectividade contínua. *Action-at-a-distance*, uma característica comum em linguagens para redes locais, designa a capacidade de se acessar recursos remotos de forma transparente, independentemente da localização dos mesmos.

A fim de atender a estes dois princípios, uma WAL deve em resumo possuir as seguintes características:

- Uma WAL deve permitir a migração entre nodos da rede de estruturas hierarquicamente organizadas, similares aos ambientes do Cálculo de Ambientes e que contenham, portanto, tanto dados como computações. Esta característica diferencia uma WAL de linguagens que permitem, por exemplo, apenas a migração de objetos individuais, como as linguagens com suporte a mobilidade de agentes, descritas na Seção 3.

- Uma WAL deve permitir comunicação apenas entre entidades que compartilhem a mesma localização. Para não pressupor *action-at-a-distance*, comunicação entre entidades remotas deve ser implementada através de mobilidade. Devem existir também primitivas para sincronizar a migração de ambientes.

- Uma WAL deve acessar suas entidades através de nomes com significado único em toda a rede e não através de endereços físicos de memória. Com isso, a mobilidade destas entidades não fica restringida por "ligações estáticas" entre as mesmas e o contexto de execução. Veja ainda que, diferentemente do que ocorre com ponteiros em linguagens de programação tradicionais, em uma WAL pode-se possuir um nome sem que se tenha acesso imediato à entidade denotada pelo mesmo. Neste sentido, um nome é similar a um URL ou a um "ponteiro simbólico".

- Em uma WAL, o acesso a qualquer entidade deve ocorrer com uma semântica de bloqueio, isto é, caso uma entidade não esteja presente localmente no momento em que é referenciada, a execução permanece bloqueada até que a mesma se torne disponível. Este tipo de semântica torna mais simples a expressão de *linkedição* e reconfiguração dinâmicas, duas características desejáveis em aplicações voltadas para uma rede como a Internet, a qual não pode ser paralisada nem sequer por alguns instantes.

- Uma WAL deve implementar o acesso a recursos do contexto de execução através de *binding* dinâmico, permitindo assim que uma entidade ao migrar para um novo nodo da rede tenha automaticamente restabelecido o acesso aos recursos de que necessita.

Assim como foram projetadas linguagens baseadas em outros cálculos de processos, como CSP, CCS e π-Cálculo, espera-se que sejam também desenvolvidas linguagens que incorporem de maneira integrada as características relacionadas acima. Uma proposta visando a introdução destas características em linguagens orientadas por objeto é descrita em [61].

6 Conclusões

Este artigo procurou mostrar que mobilidade desempenha um papel essencial em linguagens voltadas para o desenvolvimento de aplicações Internet. Foram descritas algumas linguagens propostas recentemente com suporte a mobilidade de código ou mobilidade de agentes. Mostrou-se que mobilidade de código, presente não apenas em Java, mas também em linguagens como Juice, Limbo e PLAN, constitui uma solução natural para permitir a execução de uma aplicação nas várias arquiteturas e sistemas operacionais que existem em uma rede aberta como a Internet. Já mobilidade de agentes, disponível em linguagens como Obliq, Telescript e Aglets, permite o desenvolvimento de aplicações distribuídas com um maior grau de autonomia e que não dependem tanto da rede como no modelo cliente/servidor. Finalizando o artigo, foram apresentados dois modelos teóricos para computação móvel na Internet: o π-cálculo e o Cálculo de Ambientes. Este último foi proposto recentemente com o objetivo de servir de modelo teórico para uma nova geração de linguagens para computação móvel na Internet, chamadas de *Wide Area Languages* (WAL). A Tabela 4 mostra uma comparação entre linguagens com suporte a mobilidade de código, com suporte a mobilidade de agentes e *Wide Area Languages*.

	Mobilidade de Código	Mobilidade de Agentes	WAL
Linguagem Representativa	Java	Telescript	Ainda não existe
Estilo de Mobilidade	Código (fonte ou pré-compilado)	Estado (dados + código)	Computação (dados + código + *threads*)
Construção Móvel	Métodos de um objeto	Objeto	Estruturas hierarquicamente organizadas
Migração	Reativa	Pró-ativa	Pró-ativa
Identificação de Estruturas	Via endereços físicos de memória	Via endereços físicos de memória	Via nomes ("ponteiros simbólicos")
Comunicação	Apenas com estação de origem	Irrestrita (*action-at-a-distance*)	Apenas entre entidades localizadas no mesmo contexto de execução

Tabela 4: Comparação entre Linguagens com Suporte a Mobilidade de Código, com Suporte a Mobilidade de Agentes e *Wide Area Languages*

Conforme antecipado pelo Cálculo de Ambientes, pode-se prever para dentro em breve o surgimento de novos tipos de aplicações distribuídas na Internet. Estas aplicações devem utilizar modelos de comunicação mais poderosos que o modelo cliente/servidor e serem dotadas de formas de mobilidade mais agressivas que a oferecida atualmente

por Java. No entanto, para que elas se tornem de fato uma realidade, ainda restam questões a serem resolvidas nas seguintes áreas:

- Projeto e implementação de linguagens de programação: como mostrado na Seção 5, uma linguagem de programação voltada para implementação destas aplicações deve incorporar novos conceitos relacionados com comunicação e mobilidade, além de alterações em conceitos tradicionais, como modularização, tratamento de exceções e sistemas de tipo. Ainda não existe atualmente nenhuma linguagem ou biblioteca que resolva de forma integrada todas estas questões.

- Segurança: não existe atualmente uma proposta para segurança de código móvel que alie verificação estática, correção e flexibilidade. Uma proposta que parece promissora é a de código portador de prova (PCC), descrita na Seção 2.2. No entanto, como mencionado nesta seção, a efetiva implementação desta estratégia em compiladores comerciais ainda é um problema em aberto.

- Modelos teóricos: apesar de o Cálculo de Ambientes capturar com fidelidade a noção de computação móvel na Internet, ainda resta ser desenvolvida uma teoria de equivalência de processos para este formalismo similar à que foi elaborada ao longo dos anos para o π-cálculo.

- Metodologias de Projeto: certamente aplicações distribuídas em larga escala como as que são possíveis na Internet demandarão a proposta de novas ferramentas, técnicas e metodologias de projeto e testes.

A Apêndice: Relação de Sites *Web*

Aglets:	http://www.trl.ibm.co.jp/aglets/index.html
Ajanta:	http://www.cs.umn.edu/Ajanta
Ambit:	http://www.luca.demon.co.uk/Ambit/Ambit.html
Concordia:	http://www.meitca.com/HSL/Projects/Concordia/Welcome.html
D'Agents:	http://agent.cs.dartmouth.edu
Limbo:	http://inferno.bell-labs.com/inferno
Java:	http://www.javasoft.com
Juice:	http://caesar.ics.uci.edu/juice
Obliq:	http://www.luca.demon.co.uk/Obliq/Obliq.html
PLAN:	http://www.cis.upenn.edu/ switchware/PLAN
Voyager:	http://www.objectspace.com
WebL:	http://www.research.digital.com/SRC/WebL

Referências

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

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

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 2nd edition, 1997.

[4] H. E. Ball, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[5] K. A. Bharat and L. Cardelli. Migratory applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Nov. 1995.

[6] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, Dec. 1995.

[7] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

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

[9] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99) and Third International Symposium on Mobile Agents (MA'99)*, Oct. 1999.

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

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

[12] L. Cardelli. Mobile ambient synchronization. Technical Report 013, DEC Systems Research Center, July 1997.

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

[14] L. Cardelli and R. Davies. Service combinators for web computing. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 1–10. USENIX Association, Oct. 1997.

[15] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In J. Wiederman, P. van Emde Boas, and M. Nielsen, editors, *26th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer-Verlag, July 1999.

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

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

[18] L. Cardelli and A. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *27th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2000.

[19] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, May 1997.

[20] M. Endler. Novos paradigmas de interação usando agentes móveis. Transparências de mini-curso apresentado no XVI Simpósio Brasileiro de Redes de Computadores, 1998.

[21] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–385, Jan. 1996.

[22] C. Fournet, G. Gonthier, J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, Aug. 1996.

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

[24] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

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

[26] General Magic. Odyssey white paper, 1998.

[27] A. Goscinski. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, 1991.

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

[29] J. Gosling and H. McGilton. The Java language environment: A white paper, May 1996.

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

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

[32] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

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

[34] N. Karnik and A. Tripathi. Design issues in mobile agent programming systems. *IEEE Concurrency*, pages 52–61, July-Sept. 1998.

[35] T. Kistler and H. Marais. WebL - a programming language for the Web. *Computer Networks and ISDN Systems*, 30:259–270, Apr. 1998.

[36] F. C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Dec. 1995.

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

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

[39] P. Lee and G. Necula. Research on proof-carrying code for mobile code security. *DARPA Workshop on Foundations for Secure Mobile Code*, Mar. 1997.

[40] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[41] Lucent Technologies. Inferno: la commedia interattiva, 1996.

[42] Microsoft Corporation. Proposal for authenticating code via the Internet, Apr. 1996.

[43] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991.

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

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

[46] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., Oct. 1996.

[47] K. V. Nori, U. Ammann, K. Jensen, H. Nageli, and C. Jacobi. Pascal-P implementation notes. In D. Barron, editor, *Pascal the Language ans its Implementation*, pages 437–472. John Wiley & Sons, 1981.

[48] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, Oct. 1996.

[49] Object Space. Voyager core package technical overview. Technical report, Object Space Inc., 1997.

[50] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[51] G. P. Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility.* PhD thesis, Politecnico di Torino, Italy, Feb. 1998.

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

[53] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[54] J. Postel. Internet Control Message Protocol. RFC 792, Sept. 1981.

[55] G. Robson and A. A. F. Loureiro. *Introdução à Computação Móvel.* Décima Primeira Escola de Computação, 1998.

[56] A. D. Rubin and D. E. Geer. Mobile code security. *IEEE Internet Computing*, 2(6), Nov. 1998.

[57] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, Oct. 1990.

[58] Sun Microsystems. Java Remote Method Invocation Specification, Oct. 1998.

[59] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sept. 1997.

[60] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. D. Singh. Ajanta - a mobile agent programming system. Technical Report TR98-016 (revised version), Department of Computer Science, University of Minnesota, 1999.

[61] M. T. O. Valente, R. S. Bigonha, A. A. F. Loureiro, and M. A. S. Bigonha. Linguagens para computação móvel na internet. Technical Report LLP 03/2000, Laboratório de Linguagens de Programação, DCC/UFMG, Jan. 2000.

[62] M. T. O. Valente, R. S. Bigonha, M. A. Maia, and A. A. F. Loureiro. Aplicação de ASM na especificação de sistemas móveis. In A. C. de Melo and A. M. Moreira, editors, *II Workshop on Formal Methods*, pages 60–69. Sociedade Brasileira de Computação, 1999.

[63] M. T. O. Valente, R. S. Bigonha, M. A. Maia, and A. A. F. Loureiro. Especificação de agentes móveis usando Máquinas de Estado Abstratas. In G. R. Mateus, editor, *I Workshop de Comunicação Sem Fio*, pages 123–131. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 1999.

[64] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), Feb. 1997.

[65] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In H. E. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

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

[67] D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–102, Mar. 1999.

[68] D. Wong, N. Paciorek, T. Walsh, and J. DiCelie. Concordia: An infrastructure for collaborating mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, 1997.

[69] F. Yellin. Low-level security in Java. In *Fourth International WWW Conference (WWW4)*, Boston, MA, Dec. 1995.