FACULDADE DE INFORMÁTICA
PUCRS – Brazil
http://www.inf.pucrs.br

Estudo de Linguagens de Programação com Suporte à Mobilidade de Código

Lucio Mauro Duarte e Fernando Luís Dotti

TECHNICAL REPORT SERIES

Number 015
October, 2001

Contact:

lduarte@inf.pucrs.br

http://www.inf.pucrs.br/~lduarte

fldotti@inf.pucrs.br

http://www.inf.pucrs.br/~fldotti

Lucio Mauro Duarte is a graduate student at PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul – Brazil. He is member of the Parallel and Distributed Processing research group since 2000. He receives a federal graduate research grant from CAPES (Brazil) to support his research.

Fernando Luís Dotti works at PUCRS/Brazil since 1998. He is an associate professor and develops research in Computer Networks, Distributed Systems and Code Mobility. He got his Ph.D. in 1997 at Technical University of Berlin (Germany).

Published by the Campus Global – FACIN – PUCRS
Av. Ipiranga, 6681
90619-900 Porto Alegre – RS – Brazil

Estudo de Linguagens de Programação com Suporte à Mobilidade de Código

Relatório Técnico 015/2001

Lucio Mauro Duarte (mestrando)
Fernando Luís Dotti (orientador)

1 Introdução

Nos últimos anos, a mobilidade de código surgiu como uma nova abordagem para a construção de aplicações distribuídas. Essa nova abordagem baseia-se na idéia de um processo poder, de forma dinâmica, suspender sua execução corrente em um determinado local, mover-se ou ser movido para outro local e lá continuar sua execução. Nessa abordagem, o programador possui controle sobre o momento em que se faz necessária a movimentação de seu processo e sobre o local para onde tal processo deve ir. Apesar disso, o processo de movimentação em si é totalmente transparente ao programador. Transparência essa fornecida por plataformas de suporte à mobilidade, tais como as plataformas Voyager da ObjectSpace (www.objectspace.com) e Aglets da IBM (http://www.trl.ibm.co.jp/aglets/index.html).

A partir da idéia de mobilidade de código, foram criados novos paradigmas de programação como execução remota, código por demanda e agentes móveis [1]. Tendo como base estes paradigmas, surgiram linguagens de programação que suportam a criação de aplicações de código móvel. Tais linguagens apresentam características distintas e, por isso, vantagens e desvantagens. As características dessas linguagens foram o alvo de estudo deste trabalho através da coleta de informações sobre as diversas linguagens existentes, identificação de linguagens que podiam ser agrupadas, devido a características comuns que possuíam, e implementação de aplicações seguindo as especificidades de cada um dos grupos encontrados.

Este trabalho apresenta, na seção 2, apresenta os grupos de linguagens identificados durante o estudo e as linguagens de programação com suporte à mobilidade estudadas de cada grupo, contendo uma breve descrição de suas características principais. Na seqüência, a seção 3 contém a descrição da aplicação utilizada como exemplo e as correspondentes implementações em uma linguagem representante de cada grupo. A seção 4 apresenta uma avaliação das implementações realizadas, a seção 5 traz as conclusões do estudo e a seção 6 apresenta as referências bibliográficas utilizadas.

2 Grupos de Linguagens

Após o estudo de algumas das linguagens de programação com suporte à mobilidade de código, propôs-se uma forma de classificação das mesmas de maneira a agrupá-las através de suas similaridades. Muitos tipos de classificações poderiam ser feitas mas, para efeito deste estudo, adotou-se como critério de classificação a estrutura

fundamental da linguagem. Ou seja, considera-se a estrutura mais representativa manipulada pela linguagem e na qual se baseia seu modo de programação. Tal critério, foi escolhido por parecer ser o mais importante do ponto de vista de quem pretende criar aplicações com código móvel, já que ele define o paradigma de programação e, é claro, a estrutura do agente móvel provido pela linguagem. Dessa forma, foram identificadas as linguagens *baseadas em objetos*, *baseadas em processos* e *baseadas em funções*.

A seguir são apresentados os grupos criados segundo o critério estabelecido e as principais características das linguagens que os compõem. Para cada grupo serão apresentadas as linguagens estudadas daquele grupo.

2.1 Linguagens Baseadas em Objetos

Este grupo reúne o maior número de linguagens dentre as estudadas. Isto é facilmente explicado pelo fato de apresentar uma idéia de programação muito difundida: a orientação a objetos.

Sendo linguagens baseadas em objetos, estes formam a estrutura fundamental de programação. São objetos as entidades movidas entre diferentes locais e a criação de agentes móveis neste tipo de linguagem é alicerçada sobre eles. Ou seja, um agente móvel, em uma linguagem baseada em objetos, é um objeto criado com um comportamento diferenciado que lhe permite mover-se entre hosts de uma rede. Este comportamento diferenciado é fornecido de maneira específica em cada linguagem deste grupo.

Compõem este grupo, dentre as linguagens estudadas, Java, Limbo, M0 Messengers, Obliq, Phantom, Python, TACOMA e Telescript. Estas linguagens são apresentadas a seguir.

2.1.1 Java

Desenvolvida pela Sun Microsystems, Java [1], [5], [15], [21], [22], [26], [27] é uma linguagem altamente portável que possui uma sintaxe parecida com C e C++ e contém bibliotecas que dão suporte ao desenvolvimento de aplicações para o ambiente da Internet. A sua capacidade de executar em ambientes heterogêneos (portabilidade) se deve principalmente à forma como seu código executável é gerado: o compilador Java transforma os código fonte em um código intermediário, independente de plataforma, chamado de *bytecodes*; ao executar-se o programa Java, a *Máquina Virtual Java (MVJ)* local traduz (interpreta) o código intermediário para a linguagem da máquina na qual será executado o programa. Como os *bytecodes* gerados são entendidos por qualquer MVJ, independentemente do ambiente em que foram gerados, os programas Java podem executar em qualquer sistema que possua uma MVJ.

Java provê um mecanismo para busca dinâmica de classes - *class loader* – caso um nome de classe constante em um programa não possa ser resolvido localmente ou caso tal busca seja explicitamente requerida pela aplicação. Além disso, há suporte à transmissão de código via rede a partir de uma requisição explícita do programador. Java não suporta a migração de códigos que estejam em execução.

Java possui o conceito de *interfaces*, as quais são esqueletos de classes, mostrando os métodos que a classe possui. Ou seja, uma interface define os cabeçalhos dos métodos que devem ser implementados pela classe que queira utilizar essa interface. No caso das interfaces, Java permite que uma classe implemente múltiplas interfaces.

Além disso, existe também o mecanismo de coleta automática de lixo (*garbage collection*) que provê a “destruição” de instâncias de objetos que não sejam mais

referenciados, garantindo que não haverá espaços de memória alocados indefinidamente.

Java é uma das mais populares linguagens de programação com suporte à mobilidade de código devido, em especial, a sua integração com a tecnologia da World Wide Web. Isto se deve em muito à existência das *applets.* Uma *applet* é uma classe Java que contém um aplicação completa, que pode ser transmitida juntamente com uma página HTML para permitir a execução de algum tipo de apresentação ativa ou interativa ao usuário e/ou para possibilitar a interação com um servidor remoto [15].

Por sua portabilidade e heterogeneidade, Java tem sido utilizada na construção de sistemas distribuídos e tem servido de base para a criação de plataformas de suporte à mobilidade de código, podendo citar-se a Voyager da ObjectSpace (www.objectspace.com) e a Aglets da IBM (http://www.trl.ibm.co.jp/aglets/index.html). Essas plataformas fornecem bibliotecas Java que estendem a linguagem com operações para movimentação de código. A movimentação provida por tais bibliotecas é implementada de duas formas básicas: na primeira possibilidade, o agente que se move tem seu código transferido para o local de destino, juntamente com os valores atuais de seus atributos internos; na segunda idéia, além do código e dos valores dos atributos internos, são transferidas também outras informações de contexto do agente que possam permitir que ele retome sua execução a partir da instrução seguinte ao comando de movimentação. Deve-se citar que a implementação segundo a primeira idéia não permite a retomada de execução do agente movido diretamente do ponto do código seguinte ao comando de movimentação. A retomada de execução, nesse caso, é implementada com a possibilidade de o programador fornecer o nome de um método do agente que será executado assim que o mesmo chegar a seu destino. Ou seja, defini-se o que será executado pelo agente ao fim de sua movimentação. Cabe lembrar que, seguindo a característica de portabilidade de Java, o código é interpretado em cada local em que é recebido, provendo a possibilidade de movimentação entre máquinas heterogêneas.

Em geral, as bibliotecas que suportam migração construídas sobre Java utilizam o conceito de *proxy* para prover independência de localização. Uma *proxy* é uma estrutura usada para referenciar um objeto móvel e permitir a comunicação de outros objetos com ele, sendo que uma *proxy* de um objeto móvel é criada logo que ocorre a criação deste. Esta estrutura é necessária para que, componentes que estivessem comunicando-se com um objeto antes de sua movimentação, possam continuar a comunicarem-se com ele no novo local em que se encontra. Qualquer objeto que queira comunicar-se com um objeto móvel, precisa obter uma *proxy* deste e utilizá-la para receber e enviar mensagens ao objeto móvel. Ou seja, todas as comunicações do objeto móvel são intermediadas por *proxies*, as quais são locais aos objetos que comunicam-se com ele. Com isto, comunicações remotas só ocorrem entre objetos móveis e suas *proxies* e entre *proxies*, quando há comunicação entre dois objetos móveis. A *proxy* utiliza o conceito de interfaces (descrito anteriormente nesta seção) para prover as comunicações que intermedia. Uma *proxy*, na realidade, possui, em sua estrutura, uma interface do objeto móvel que ela referencia, permitindo que ela possa receber ativações de métodos do objeto móvel e, caso sejam válidas (sejam relativas a métodos constantes na interface pública do objeto), encaminhá-las ao objeto em questão.

Não há uma área bem definida para utilização de Java. Na verdade, a maioria das aplicações distribuídas podem ser construídas utilizando as vantagens de Java.

2.1.2 Limbo

É uma linguagem de programação imperativa que foi desenvolvida pela Lucent Technologies Inc. para a execução de aplicações distribuídas de pouca escalabilidade (aplicações que executam sobre um número pequeno de nodos). Suporta programação modular, coleta automática de lixo e comunicação entre processos. Limbo [26] é a linguagem utilizada para a criação de aplicações que rodam sobre a plataforma Inferno [41], a qual é acompanhada por uma máquina virtual que possui um interpretador para a linguagem.

Limbo possui três tipos de objetos: *module*, *data* e *function*. O objeto *module* é o tipo básico, sendo que uma aplicação em Limbo consiste de um ou mais *modules*. Um *module* possui uma coleção de constantes, tipos abstratos de dados, dados e funções pertencentes a ele. O objeto *data* representa um dado de um tipo qualquer armazenado em um *module*, o qual pode ser manipulado através de operações aritméticas, atribuições, etc. O objeto *function* caracteriza uma função contida em um *module*, identificada pelos tipos de seus argumentos e retornos.

O tipo contido em Limbo que pode ser destacado entre os existentes (*byte*, *string*, *int*, etc.) é o tipo *channel*, que provê um mecanismo de comunicação capaz de enviar e receber objetos de tipos específicos, possibilitando um mecanismo de mobilidade de código. É através de *channels* também que torna-se possível realizar a comunicação entre processos, tanto locais como remotos. O único senão desta propriedade da linguagem é que cada *channel* transmite apenas um tipo de objeto. Desse modo, dever-se-á ter um *channel* para cada tipo de objeto que se quiser transmitir.

Informações sobre a linguagem Limbo podem ser obtidas através da URL http://inferno.lucent.com/inferno/limbo.html

2.1.3 M0 Messengers

Desenvolvida na University of Geneva, M0 [1], [11], [20], [21], [22], [27] é uma linguagem estritamente interpretada, ou seja, não gera um código intermediário. M0 não foi desenvolvida para a criação de aplicações de código móvel, e sim, para prover uma camada intermediária de suporte à mobilidade de código para as camadas de mais alto nível. É uma linguagem que lembra muito PostScript (http://www.adobe.com/products/postscript/)e usa uma estrutura denominada *dictionary* para definir uma memória global. Trocas de dados ocorrem somente através de memória compartilhada acessada através dos *dictionaries*.

Unidades de execução são os *messengers*, os quais podem submeter código para a criação de outros *messengers* em um local remoto onde esteja uma *platform*, que é o ambiente de execução dos *messengers*. O código enviado é executado logo que recebido. *Messengers* são threads que não possuem qualquer tipo de identificação. Ou seja, não há como referenciar um *messenger*. Portanto, caso um *messenger* não tenha sido programado para cooperar com outros *messengers* (através dos *dictionaries*), o primeiro permanece invisível aos últimos.

Uma thread remota é criada enviando-se um *messenger* por um *channel*, que é um canal de comunicação entre dois hosts, o que implementa o mecanismo de movimentação de M0.

Mais informações sobre esta linguagem podem ser obtidas na URL http://cuiwww.unige.ch/tios/msgr.

2.1.4 Obliq

Desenvolvida pela DEC, Obliq [1], [4], [10], [21], [22], [27] é uma linguagem estritamente interpretada e não tipada que foi criada para suportar aplicações distribuídas orientadas a objetos. É uma linguagem baseada em protótipos, logo, não existem classes e os objetos são criados pela utilização de objetos já existentes, que são os protótipos, através da adaptação destes às necessidades do programador. Dessa forma, todos os métodos necessários para um objeto já estão contidos no próprio objeto, já que não há superclasses. Computação em Obliq é transparente via rede; isto é, ela não depende do local onde é executada nem do local onde se encontram as estruturas sobre as quais ela é realizada.

Qualquer valor pode ser transmitido entre locais diferentes, denominados *execution engines*. Novos objetos criados por uma unidade de execução, que é uma thread, são armazenados localmente. A movimentação de objetos é fornecida pela clonagem dos mesmos para o local remoto, levando cópias dos valores de atributos simples do objeto original. Valores de atributos estruturados (um vetor, por exemplo) são compartilhados pelo objeto original e pelo clone. Acessos ao objeto no local de origem são redirecionadas ao local atual do mesmo. Além disso, Obliq também suporta a serialização de objetos e seu modelo de distribuição de objetos é fortemente baseado no modelo de Modula-3 [14].

Por ser um trabalho acadêmico, Obliq possui poucas utilizações. Pode-se citar como exemplo de aplicação usando Obliq, um construtor de aplicações distribuídas chamado Visual Obliq.

Outras informações sobre Obliq podem ser obtidas na URL http://research.compaq.com/SRC/personal/luca/Obliq/Obliq.html.

2.1.5 Phantom

Desenvolvida no Trinity College em Dublin, Irlanda, Phantom [9], [13] é uma linguagem interpretada, orientada a objetos e fortemente tipada desenvolvida para a criação de aplicações distribuídas. Seu interpretador é todo construído em ANSI C e provê uma biblioteca de interface para Tk (kit de ferramentas gráficas para interfaces de usuário distribuído junto com o interpretador Tcl [7], [17]).

Phantom não apresenta mecanismo de herança múltipla. Sintaticamente baseia-se, em grande parte, na linguagem Modula-3 [14] e seu modelo de distribuição é basicamente o modelo utilizado pela Obliq (vide 2.1.4). Por ser baseada em Modula-3, permite facilidade de utilização por programadores que tenham trabalhado com sucessores de Pascal. Phantom inclui objetos, interfaces, threads, coleta automática de lixo e tratamento de exceções. Também há suporte a listas com dimensionamento dinâmico.

Em Phantom, objetos nunca são implicitamente movidos para locais remotos devido a uma atribuição, uma chamada de procedimento ou comando de retorno; em vez disso, é passada uma referência ao objeto ao local remoto, permanecendo o objeto no local em que se encontra. A movimentação de um objeto só ocorre quando for explicitamente definida pelo programador. Ou seja, Phantom deixa a cargo do programador decidir quando mover um objeto e para onde movê-lo.

Phantom também provê transparência quanto à distribuição, permitindo que valores de um programa possam ser acessados de qualquer ponto da rede. Phantom permite a transmissão de fragmentos de código (funções e procedimentos) através da rede, o que garante a idéia de mobilidade de código da linguagem.

Algumas das aplicações conhecidas criadas sobre Phantom são sistemas distribuídos para conferências, jogos multi-jogadores e ferramentas para trabalho colaborativo.

Informações sobre a linguagem Phantom podem ser obtidas na URL http://www.apocalypse.org/pub/u/antony/phantom/phantom.htm.

2.1.6 Python

Desenvolvida inicialmente pela CWI (Centrum voor Wiskunde en Informatica), sendo desenvolvida agora pela CNRI (Corporation for National Research Initiatives), Python [19], [27] é uma linguagem de programação interpretada e orientada a objetos. Um programa em Python é construído a partir de módulos que definem classes. Estes módulos podem ter sido escritos em Python ou podem fornecer uma extensão à linguagem, sendo módulos implementados em uma linguagem compilada como C ou C++, podendo, assim, definirem-se novas funções e variáveis, bem como novos tipos de objetos. Esta possibilidade preenche uma lacuna deixada em Python que é a inexistência de tipos abstratos de dados, que normalmente é fornecida pelas linguagens. Em Python, todos os elementos da linguagem são passíveis de serem passados como argumentos para métodos.

Similarmente a Java, Python possui uma linguagem intermediária chamada *Pycode* (*Python Byte Code*), a qual é depois interpretada pelo interpretador Python. Um *Pycode* de um local remoto pode ser requerido e dinamicamente executado.

Uma característica marcante de Python é a dinamicidade e flexibilidade fornecidas. Funções definidas em Python não são tipadas, isto é, ao serem declaradas, não são definidos tipos específicos para os parâmetros, podendo estes assumir qualquer tipo dentre os possíveis da linguagem. Além disso, segundo [13], novas funções e variáveis de classe podem ser adicionadas às classes a qualquer momento. Isto é, novos componentes compilados podem ser dinamicamente passados ao interpretador sob demanda. Com isso, instâncias já existentes são alteradas em tempo de execução e novas instâncias geradas a partir de então apresentam as alterações feitas. Há, também segundo [13], a possibilidade, suportada pela linguagem, de redefinir-se o que ocorre quando uma nova função ou variável é adicionada a uma classe. Python também suporta herança múltipla e oferece mecanismos de suporte à transmissão de código através da rede. Estes mecanismos combinados com a capacidade de adição dinâmica de novos componentes, atribui a Python a possibilidade prover mobilidade de código.

Informações adicionais podem ser encontradas em http://www.python.org.

2.1.7 TACOMA

Desenvolvida pela University of Tromsø e pela Cornell University, TACOMA (Tromsø And COrnell Mobile Agents) [1], [21], [22], [24], [27] é uma extensão da linguagem Tcl [7], [17] que inclui primitivas de suporte à mobilidade de código. TACOMA é uma linguagem puramente interpretada, ou seja, não realiza a geração de código intermediário como *bytecode* de Java (vide 2.1.1).

Em TACOMA, as unidades de execução são implementadas como processos Unix, chamados de *agents*. Um *agent* pode requerer a execução de um novo *agent* em um local remoto (as funcionalidades de um local que suporta *agents* são implementadas pelo próprio sistema operacional Unix). Para isso, o primeiro envia ao local remoto, juntamente com com o *agent* a ser executado, uma estrutura de dados chamada *briefcase*, a qual contém o código e dados de inicialização deste novo *agent*. Ao

chegarem os dados do local de origem, uma nova unidade de execução é criada no local remoto usando o código recebido e o novo *agent*, então, acessa os dados da *briefcase* para a sua inicialização. A fim de permitir ações de um *agent* baseadas nas ações anteriores, os *agents* de TACOMA carregam consigo seu estado em cada local onde realizaram alguma tarefa. É usado o Tcl-TCP, uma extensão de Tcl que suporta comunicação TCP, para mover os *agents*.

2.1.8 Telescript

Desenvolvida pela General Magic, é uma linguagem orientada a objetos concebida para o desenvolvimento de aplicações distribuídas de larga escala, especialmente comércio eletrônico, que é executada sobre uma máquina virtual. Possui uma linguagem intermediária chamada *Low Telescript*, a qual é interpretada. Telescript [1], [12], [13], [21], [22], [27] permite apenas herança simples. Classes também podem herdar de *mix-ins*, que são classes abstratas.

Telescript oferece dois tipos de unidades de execução (threads): agentes (*agents*), que são capazes de migrar sobre a *Telesphere* (conjunto de lugares na rede onde interpretadores Telescript estão executando); e *places*, que são estáticas e podem conter outras unidades de execução (agentes ou outros *places*). Os agentes se movem a partir da operação *go*, que causa a finalização do agente no local de origem e a reconstrução do mesmo no local de destino. A partir daí, a execução prossegue do ponto posterior à chamada da operação *go*. Existe ainda uma operação chamada *send* que produz a clonagem de um agente para um local remoto. Os agentes movem-se entre *places*, os quais executam sobre *engines*, que são as abstrações de locais de execução dos mesmos.

Um agente pode conter vários objetos. Quando o agente migra, os objetos pertencentes a ele são dinamicamente replicados para o local da migração, juntamente com o código do agente e o estado de execução. O agente obtém acesso a recursos de um local através de um *place*. Assim, quando um agente entra em um *place* ele pode utilizar os recursos deste local através de uma referência ao *place*.

2.2 Linguagens Baseadas em Processos

Este grupo apresenta como característica principal a programação baseada no Cálculo-π [38]. O Cálculo-π é um modelo de computação concorrente que possibilita uma forma de descrever e analisar sistemas constituídos de agentes (processos), que interagem entre si e cujas configurações ou vizinhanças estão em constante mudança. Este modelo baseia-se na noção de nomeação (*naming*). Nomes são as entidades mais primitivas do Cálculo-π e servem para a criação de processos, além de permitir sua localização e comunicação [38]. A interação entre processos ocorre de forma síncrona, seguindo o modelo de comunicação *rendezvous*, no qual tanto quem envia quanto quem recebe fica bloqueado até a comunicação ser finalizada. Outras entidades fundamentais são os processos e, devido a isso, estas linguagens fornecem mecanismos para criação, migração e comunicação entre processos. A idéia de agentes móveis, por conseqüência, baseia-se na estruturação de processos com capacidade de serem transmitidos de um local para outro. Esta transmissão se dá pela utilização da outra entidade advinda do Cálculo-π: o canal. Ele representa o meio de comunicação entre processos e a via de movimentação dos mesmos; isto é, pode-se transmitir processos através de canais.

Deve-se notar que linguagens que baseiam-se em Cálculo-π podem criar mecanismos diferenciados para implementar conceitos relacionados a essa base. Como exemplo, pode-se citar a linguagem KLAIM (vide 2.2.2), a qual utiliza a idéia de tuplas

e não de canais. As idéias são similares na medida em que as tuplas servem, tal qual os canais, para sincronização e comunicação entre processos.

As representantes deste grupo, dentre as relacionadas, são April, KLAIM, Nomadic Pict e Pict, apresentadas a seguir.

2.2.1 April

Desenvolvida no Imperial College of London, April [18] é uma linguagem simbólica orientada a processos fortemente tipada. Sendo assim, fornece facilidades para a definição de processos e para permitir a comunicação entre processos, de maneira uniforme, em um ambiente distribuído.

April oferece recursos para processamento de macros de usuário, tornando possível construírem-se novas camadas sobre a linguagem básica. Ou seja, pode-se incorporar novas características à linguagem original. Esta capacidade da linguagem foi utilizada, por exemplo, para criar-se uma extensão de April provendo orientação a objetos [33].

Todos os processos em April são identificados por um nome chamado *handle*. Este *handle* normalmente é gerado automaticamente, mas pode ser fornecido pelo programador. *Handles* fornecidos pelo programador são locais ao lugar de invocação do processo (ambiente Unix) que executa April. O *handle* pode ser tornado público para todo o sistema quando ele é registrado no *name server* da invocação local de April. *O name server* executa papel semelhante ao de um DNS (Domain Name Server) [42] da Internet. Ele também permite que sejam construídas aplicações distribuídas em April pela possibilidade de referenciar processos remotos, já que uma mensagem é enviada a um processo através de seu *handle*, o qual deve estar registrado em um *name server*.

As mensagens podem ser enviadas entre processos independentemente de suas localizações. Isto é possível devido ao *handle* ser único dentro do sistema (distribuído ou não) e, portanto, um *handle* é sempre mapeado para uma única localização. As mensagens recebidas por um processo são colocadas em seu buffer.

As primitivas de comunicação usam protocolo TCP/IP, o que objetiva que April seja usada em ambientes heterogêneos, uma vez que uma aplicação April pode se comunicar com outra aplicação, em April ou não, via protocolo TCP/IP. O uso deste protocolo também permite que funções e procedimentos externos possam ser utilizados por uma aplicação April.

April fornece um mecanismo chamado *procedure abstraction* que permite a construção de um procedimento em tempo de execução. Esta facilidade provê a capacidade de suportar mobilidade de código. Para migrar, o processo se auto-envia para outro processo através do nome público deste último (*handle*) como uma *procedure abstraction*. Isto é, o processo migra como se fosse um procedimento criado e ligado a outro processo em tempo de execução. Qualquer informação de contexto que seja necessária à execução do processo migrante é levada junto com ele como argumento ao novo procedimento criado, o qual define um agente móvel. O processo de destino recebe o agente móvel executando uma instrução de *receive*. Ao receber o agente, o processo destino pode usar a *procedure abstraction* através de uma ligação dinâmica ou criar uma nova thread para executar o código recebido.

April é utilizada, principalmente, na construção de aplicações de inteligência artificial distribuída.

Novas informações sobre April constam no endereço da Internet http://www-lp.doc.ic.ac.uk/~klc/april1.html.

2.2.2 KLAIM

Linguagem desenvolvida na Universidade de Florença na Itália, KLAIM (Kernel Language for Agents Interaction and Mobility) [6] suporta a movimentação de dados e processos entre ambientes computacionais baseando-se no cálculo de processos (Cálculo-π). A linguagem inspira-se em Linda [30], uma linguagem que suporta um mecanismo de comunicações assíncronas baseado em um ambiente de compartilhamento global chamado *tuple space*. Um *tuple space* pode ser definido como uma coleção de *tuplas*, onde cada tupla é um conjunto de variáveis. Tuplas podem ser adicionadas ou removidas do *tuple space*. O modelo de comunicação assíncrona permite que o programador possa controlar explicitamente as interações entre processos através de dados compartilhados. Tal modelo permite, também, que possa ser usado o mesmo conjunto de primitivas tanto para manipulação de dados como para sincronização de processos. Com base nestas características, KLAIM vem adicionar a Linda uma estrutura para viabilizar múltiplos *tuple spaces* e permitir manipulação explícita de localidades e nomes de localidades. Uma localidade pode ser vista como um nome simbólico para um lugar físico onde processos e *tuple spaces* estão executando. Dessa forma, a idéia de localidade provê um mecanismo de abstração de localização.

Uma das modificações mais significativas em relação a Linda, foi a alteração em primitivas desta para suportar a programação de agentes móveis. Um exemplo é a primitiva *eval* que permite a adição de um componente a um *tuple space* e invoca a criação de um novo processo a partir do componente recebido. Esta primitiva, em Linda, tem como argumentos tuplas, mas, em KLAIM, ela foi modificada a fim de permitir que processos fossem recebidos como argumentos, viabilizando os agentes móveis.

O conjunto de operadores utilizados em KLAIM foi trazido do *Calculus of Communicating Systems* (CCS) de Milner [31] e a implementação do protótipo da linguagem foi feita em Java (vide 2.1.1) para assegurar portabilidade. Esta implementação de KLAIM em Java é chamada KLAVA [32] e consiste da extensão de Java com dois pacotes novos: um que implementa as primitivas padrão de Linda e outro que suporta a implementação de KLAIM.

Mais informações sobre esta linguagem podem ser obtidas no endereço http://music.dsi.unifi.it/klaim.html.

2.2.3 Pict

Desenvolvida na Universidade de Indiana, EUA, Pict [3] baseia-se no Cálculo-π, no qual, como visto no início da seção 2.2, existem apenas duas entidades: processos e canais. Processos (também chamados agentes) são os componentes ativos do sistema que interagem sincronamente, seguindo o modelo chamado *rendezvous*. Esta comunicação entre processos é fornecida pelos canais. Quando dois processos sincronizam, eles trocam um canal onde consta o resultado da interação dos processos envolvidos.

Pict oferece tipos básicos, tais como *Int*, *Bool*, *Char* e *String*, bem como canais predefinidos para prover operações sobre estes tipos básicos. Dessa forma, ao realizar-se a soma de dois números inteiros, por exemplo, pode-se enviar dois valores do tipo *Int* pelo canal predefinido "+". Juntamente com os dois valores do exemplo, deve ser passado também um canal, através do qual poder-se-á obter o resultado da operação quando esta for concluída. Também são fornecidos canais de interação, tal como o canal *print*, usado para enviar strings a serem impressas na saída padrão. Cada canal sempre

transmite o mesmo tipo de valor durante toda a sua existência, logo, deve ser criado um canal para cada tipo que se queira transmitir.

Em princípio, Pict não apresenta um mecanismo para mobilidade de código, mas tal característica pode ser representada com os recursos existentes na linguagem: cada processo possui canais associados a ele, que representam os seus meios de comunicação com outros processos e também formam o seu ambiente de execução; quando o conjunto de canais de um processo se modifica, pode-se denotar isto como uma mudança de ambiente de execução, ou seja, uma mudança de localização. Dessa forma, uma movimentação de um processo ocorre a partir da transformação de seu conjunto de canais.

Mais informações sobre Pict na URL http://www.cs.indiana.edu/ftp/pierce/pict.

2.2.4 Nomadic Pict

Desenvolvida na Universidade de Cambridge, Nomadic Pict [2] é uma extensão de Pict (vide 2.2.3) com a adição de primitivas para a criação e migração de agentes e para a troca de mensagens assíncronas entre agentes, obtida através de comunicação independente de localização. Esta comunicação independente de localização permite que agentes interajam sem precisarem, necessariamente, conhecer a exata localização um do outro.

Nomadic Pict baseia-se no Cálculo-π. A linguagem possui uma arquitetura em dois níveis: o *low-level*, que consiste de primitivas dependentes de localização, tal como comunicação e migração de agentes, e o *high-level*, que estende o *low level* fornecendo uma primitiva para comunicação independente de localização.

As três principais entidades de Nomadic pict são os *sites*, os agentes e os canais. Um *site* é uma máquina física na qual está executando uma instância do *runtime system* do Nomadic Pict. O *runtime system* consiste de duas camadas: a Máquina Virtual, que mantém uma *agent store* com os estados dos agentes, e o Servidor de E/S, que é o responsável por receber agentes e colocá-los na *agent store*. As duas camadas atuam sobre TCP. Agentes são unidades de código executável que localizam-se, a cada momento, em um *site*. O corpo de um agente pode ser composto por vários processos paralelos (*threads*) que interagem via troca de mensagens. Canais suportam a comunicação dentro dos agentes e entre agentes. Também há suporte da linguagem a mensagens internas de *sites* e entre *sites*. Tais mensagens são, internamente, enviadas via conexões TCP criadas por demanda. A migração suportada pela linguagem envolve a movimentação de um agente completo (agente com todos os processos que o compõem).

Programas em Nomadic Pict são compilados através da tradução de um programa do *high-level* para a linguagem do *low-level* da arquitetura, formando um código intermediário. Este código intermediário é independente de arquitetura e é executado pelo *runtime system*.

Outras informações sobre Nomadic Pict podem ser encontrados na seguinte URL: http://www.cl.cam.ac.uk/users/ptw20/nomadicpict.html.

2.3 Linguagens Baseadas em Funções

As linguagens baseadas em funções,também chamadas de linguagens funcionais, como o próprio nome já diz, têm como estrutura essencial as funções. A programação funcional é um estilo de programação que enfatiza a avaliação de expressões em vez da

execução de comandos. As expressões são formadas pelo uso de funções que combinam tipos básicos da linguagem. Funções, nestas linguagens são tratadas como tipos *first class*, ou seja, funções podem ser recebidas como parâmetros ou retornadas por outras funções como qualquer outro tipo básico da linguagem. Funções que recebem funções como argumento e/ou retornam funções são denominadas funções de alta ordem (*high order functions*). Funções também podem ser recursivas e polimórficas.

O desenvolvimento de linguagens funcionais foi influenciado, principal e fundamentalmente, pelo Cálculo Lambda [40], o qual é tido como a primeira linguagem funcional. Dessa forma, pode-se ver as linguagens de programação funcionais como versões aprimoradas do Cálculo Lambda original. O Cálculo Lambda baseia-se na utilização dos aspectos computacionais das funções (entenda-se aqui o termo função no seu sentido matemático). Um desses aspectos mais importantes é que funções podem ser aplicadas a elas mesmas; ou seja, permite atingir-se o efeito de recursão sem, explicitamente, escrever uma definição recursiva. As funções transmitidas e executadas em locais remotos oferecem a noção de agentes móveis.

As linguagens que compõem este grupo são Facile, Objective Caml e Tycoon e são descritas, sucintamente, a seguir.

2.3.1 Facile

Desenvolvida pela ECRC (European Computer-Industry Research Centre) em Munique, Alemanha, Facile (Fast and Accurate Categorisation of Information by Language Engineering) [1], [4], [8], [27] é uma linguagem funcional fortemente tipada que estende a linguagem Standard ML [25] com primitivas para distribuição, concorrência e comunicação. Unidades de execução são implementadas como threads que rodam sobre ambientes computacionais chamados *nodos*. A abstração canal (*channel*) é usada para permitir a comunicação entre threads, possibilitando a transmissão de qualquer valor legal dentro da linguagem, inclusive funções. Esta comunicação via canal segue o modelo *rendezvous*, tal como descrito em 2.2. Ao finalizar a transmissão, o canal usado na comunicação é fechado e a thread receptora pode utilizar o valor recebido. Fica a cargo do programador especificar se uma função enviada a um nodo deve ser executada diretamente, ligada a outra thread já em execução (ligação dinâmica), ou deve ser criada uma nova thread para executá-la.

O programador cria interfaces para os recursos a serem acessados por uma função. Cada interface possui um conjunto de assinaturas de métodos que podem ser executados sobre o recurso ao qual ela pertence. Dessa forma, uma função tem definidos os recursos aos quais ela tem acesso e utiliza-os através de suas interfaces. Através desse mecanismo, a função pode acessar qualquer recurso para qual possua uma interface em qualquer local onde ela venha a executar, desde que as operações oferecidas pelo recurso sejam compatíveis com as que a função possui na interface para aquele recurso.

Na sua implementação original, Facile não provia suporte à mobilidade, o que ocorreu a partir do que foi proposto por [4]. O compilador de Facile foi modificado para gerar código em uma representação padrão de modo a poder executá-lo sobre arquiteturas heterogêneas. Esta modificação foi feita a partir das operações de *marshalling* e *unmarshalling*. Estas operações servem, respectivamente, para transformar um elemento na origem de uma transmissão em uma representação padrão e, ao chegar ao destino, converter o elemento para a representação local apropriada.

Outras modificações importantes propostas por [4] foram a introdução de um modo de referência a valores remotos, através de estruturas *proxy*, as quais são usadas

para especificar interfaces de recursos remotos para acesso a valores e tipos dos mesmos, e um mecanismo para compilação implícita de um elemento transmitido quando este chega ao seu destino, o que permite que ele seja transformado para uma forma executável para o ambiente local onde ele se encontra no momento, provendo portabilidade para a linguagem.

A representação de um agente em Facile é simplesmente a de uma função. Uma função é denominada de agente caso um comando de movimentação (*send*) seja aplicado sobre ela. Dessa forma, não há uma representação específica para funções que são agentes e funções que não o são; qualquer função pode ser um agente e, portanto, movida de um local para outro. A primitiva *send* recebe um argumento (função) e dispara o processo de *marshalling* do mesmo para transmissão ao destino da movimentação.

Um agente pode ser estruturado contendo várias outras funções, sendo que uma delas é definida como a função principal para execução do código definido para o agente em questão.

Para mais informações sobre a linguagem Facile e sua especificação, pode-se ir em http://www.ecrc.de/research/projects/facile/.

2.3.2 Objective Caml

É uma linguagem de programação funcional desenvolvida pela INRIA (French National Institute for Research in Computer Science and Control) que segue a tradição de ML [25] e que originou-se na linguagem Caml, também desenvolvida pela INRIA. Objective Caml [26], [28], ou simplesmente O'Caml, passou a ser usada como linguagem para código móvel a partir do desenvolvimento do *browser* MMM (também da INRIA), o qual abria a possibilidade de executar applets O'Caml via rede a partir de uma ligação dinâmica, tal como ocorre com as *applets* Java.

O'Caml é fortemente tipada e suporta concorrência (através de threads) e orientação a objetos baseada em classes. O'Caml permite vários tipos de paradigmas de programação, como funcional, imperativo e orientado a objetos. Os processos em O'Caml são implementados sobre funções.

O'Caml provê mecanismos para transmissão de dados entre plataformas heterogêneas, coleta automática de lixo e para acesso a recursos de baixo nível do sistema através da disponibilização de chamadas de sistema (Unix), tais como gerência de sockets.

Mais informações sobre Objective Caml em http://www.inria.org.

2.3.3 Tycoon

Desenvolvida na Universidade de Hamburgo, Tycoon (Typed Communicating Objects in Open Environments) [21], [22], [23], [27] é uma linguagem que trabalha com threads como unidades de execução. Uma thread pode mover-se ou ser movida entre máquinas virtuais da plataforma Tycoon (para a qual a linguagem foi desenvolvida) pela primitiva *migrate* e pode especificar os tipos de recursos remotos que irá utilizar. Assim, quando a thread chega ao seu destino, os recursos pretendidos são reservados para ela. Incompatibilidades entre os recursos requeridos pela thread e os recursos existentes no local remoto são identificados no instante de partida da mesma.

Tycoon suporta programação multi-paradigma, permitindo que o programador possa escolher programar sobre o paradigma imperativo ou funcional.

3 Exemplo de Implementação

Nesta seção serão apresentadas implementações, utilizando uma linguagem de cada grupo descrito na seção anterior, para uma aplicação. Esta aplicação foi escolhida como forma de exemplificar a utilização do conceito de mobilidade de código e como meio de identificar as diferenças de implementação ao criar-se a mesma aplicação baseando-se em objetos, processos ou funções. São apresentadas as três implementações (baseada em objetos, em processos e funcional) devidamente comentadas a respeito de algumas questões de implementação e observações relevantes.

3.1 Aplicação de Lojas Remotas

Esta aplicação apresenta um conjunto de lojas remotas contendo uma lista de produtos e seus respectivos preços. Um agente móvel de consulta é enviado a cada loja existente consultando o preço para um determinado produto e retorna com o menor preço encontrado e a identificação da loja que possui tal preço para o produto procurado. Após o retorno do primeiro agente, um segundo agente móvel é enviado à loja que contém o produto pelo menor preço, levando consigo a quantidade do produto que deve ser comprada. Chegando à loja, o agente requisita a compra do produto na quantidade especificada. Feita a compra, o agente retorna para o local de origem com a quantidade comprada do produto.

Aplicações similares a esta podem ser encontradas em muitos manuais e tutoriais de linguagens de programação que suportam código móvel. Esta é, portanto, uma aplicação clássica envolvendo mobilidade de código e serve como um bom exemplo para demonstrar uma aplicação móvel implementada segundo cada uma das abordagens dos grupos de linguagens.

3.1.1 Implementação baseada em objetos

A implementação da aplicação descrita em uma linguagem baseada em objetos foi realizada utilizando-se Java em combinação com a plataforma de suporte à mobilidade Voyager, da ObjectSpace. Esta plataforma provê bibliotecas Java para suporte à criação e execução de códigos móveis. Sua utilização se deve ao fato de que ela fornece mecanismos para criação de agentes móveis em Java com mais facilidade e sem a necessidade de lidar-se com questões de formatação de dados e de comunicação. Certamente seria possível criar-se a mesma implementação utilizando-se apenas Java, o que é comprovado pela existência de uma plataforma feita em Java que realiza as tarefas necessárias para a criação e transporte de agentes móveis. A escolha pela linguagem Java deve-se ao fato de ser a linguagem mais conhecida dentre as vistas e que melhor representa as características do seu grupo.

A implementação em Java, sendo ela baseada em objetos, foi feita subdivida em classes. A classe principal, chamada *Customer*, representa o objeto que inicia a aplicação e cria os agentes móveis, representados pelas classes *QueryAgent* e *BuyerAgent*. Estas classes são apresentadas na Figura 1.

```
1. public class Customer
2. {
3.   public static void main (String[] args)
4.   {
5.     // Lista de lojas
6.     Vector stores = new Vector ();
```

```
7.     try
8.     {
9.       // Inicializa plataforma Voyager
10.      Voyager.startup();

11.      int port = 8000;

12.      // Cria os produtos
13.      Vector products = new Vector ();
14.      products.add((String)"monitor");
15.      products.add((String)"teclado");
16.      products.add((String)"mouse");
17.      products.add((String)"gabinete");
18.      products.add((String)"CD-ROM");

19.      for (int i = 0; i < args.length; i+=2)
20.      {
21.        // Cria loja remota
22.        IRemoteStore store = (IRemoteStore) Factory.create
          ("apps.java.RemoteStore", "//"+args[i]+":"+args[i+1]);
23.        stores.add(store);
24.        store.setRemoteStore("store@"+args[i]+":"+args[i+1],
   products);
25.      }

26.      // Cria agente para realizar a pesquisa
27.      IQueryAgent queryAgent = (IQueryAgent) Factory.create
        ("apps.java.QueryAgent");
28.      Enumeration storeList = stores.elements();
29.      int pos = (new Double((Math.random() * 10) %
        products.size())).intValue();
30.      queryAgent.goQuery((String)products.get(pos), storeList);
31.      double price = queryAgent.getPrice();
32.      String st = queryAgent.getStore();
33.      System.out.println((String)products.get(pos)+" = $" + price+",
        na loja "+st);

34.      // Cria agente para realizar compra
35.      IBuyerAgent buyerAgent = (IBuyerAgent) Factory.create
        ("apps.java.BuyerAgent");
36.      int amount = (new Double(Math.random()*10+1)).intValue();
37.      buyerAgent.goBuy((String)products.get(pos),
        queryAgent.getInterface(), amount);
38.      amount = buyerAgent.getAmount();
39.      System.out.println(amount + " unidade(s) de
        "+(String)products.get(pos)+" comprada(s) na loja "+st);
40.    }
41.    catch(Exception exception)
42.    {
43.      System.err.println(exception);
44.    }

45.    // Finaliza plataforma
46.    Voyager.shutdown();
47.  }
48. }

49. public class QueryAgent implements IQueryAgent, Serializable
50. {
51.   String product;
```

```
52.   double price = 0;
53.   String storeName;
54.   IRemoteStore storeInt;
55.   IAgent agent;

56.   public QueryAgent ()
57.   {
58.     System.out.println("Criou agente de consulta");
59.   }

60.   public void goQuery (String product, Enumeration storeList)
61.   {
62.     // Move-se para todas as lojas da lista recebida
63.     System.out.println("Enviando agente de consulta");
64.     agent = Agent.of(this);
65.     while (storeList.hasMoreElements())
66.     {
67.       this.product = product;
68.       IRemoteStore store = (IRemoteStore) storeList.nextElement();
69.       try
70.       {
71.         agent.moveTo(store, "atStore");
72.       }
73.       catch(Exception exception)
74.       {
75.         System.err.println(exception);
76.       }
77.     }
78.     try
79.     {
80.       Agent.of(this).moveTo(Agent.of(this).getHome());
81.       System.out.println("Agente de consulta estah de volta!");
82.     }
83.     catch(Exception exception)
84.     {
85.       System.err.println(exception);
86.     }
87.   }

88.   public void atStore (IRemoteStore store)
89.   {
90.     System.out.println("Agente de consulta na loja");
91.     Enumeration products = store.queryProducts();
92.     while (products.hasMoreElements())
93.     {
94.       ItemList item = (ItemList) products.nextElement();
95.       if (product.compareTo(item.getName()) == 0)
96.       {
97.         if (price == 0 || price > item.getPrice())
98.         {
99.           price = item.getPrice();
100.          storeName = store.getStoreName();
101.          storeInt = store;
102.        }
103.        break;
104.      }
105.    }
106.    System.out.println("Menor preco para "+product+" = "+price);
107.  }

108.  public double getPrice ()
```

```
109.  {
110.    return price;
111.  }

112.  public String getStore ()
113.  {
114.    return storeName;
115.  }

116.  public IRemoteStore getInterface ()
117.  {
118.    return storeInt;
119.  }
120. }

121. public class BuyerAgent implements IBuyerAgent, Serializable
122. {
123.   String product;
124.   int amount;

125.   public BuyerAgent ()
126.   {
127.     System.out.println("Criou agente de compra");
128.   }

129.   public void goBuy (String product, IRemoteStore store, int amount)
130.   {
131.     // Move-se para todas as lojas da lista recebida
132.     this.product = product;
133.     this.amount = amount;
134.     try
135.     {
136.       Agent.of(this).moveTo(store, "atStore");
137.     }
138.     catch(Exception exception)
139.     {
140.       System.err.println(exception);
141.     }
142.     try
143.     {
144.       Agent.of(this).moveTo(Agent.of(this).getHome());
145.       System.out.println("Agente de compra estah de volta!");
146.     }
147.     catch(Exception exception)
148.     {
149.       System.err.println(exception);
150.     }
151.   }

152.   public void atStore (IRemoteStore store)
153.   {
154.     System.out.println("Agente de compra na loja");
155.     amount = store.sellProduct(product, amount);
156.   }

157.   public int getAmount ()
158.   {
159.     return amount;
160.   }
161.}
```

Figura 1. Código para a classe *Customer* e para as classes-base dos agentes móveis.

Na classe *Customer* é feita, primeiramente, a criação da lista de produtos (linhas 13 a 18). Nas linhas 19 a 25 é feita a criação das lojas remotas. As lojas são criadas nos locais recebidos como argumentos (nomes de hosts ou endereços IP). Como a plataforma Voyager precisa ser instanciada em cada local onde um agente móvel irá passar, é fornecida também uma porta de comunicação que será utilizada pela instância Voyager no host onde a loja for criada. A instrução da linha 22 cria a loja remota no local especificado e retorna uma referência a ela (*proxy*). Através dessa referência é possível ativar todos os métodos associados à classe que implementa uma loja remota (*RemoteStore*) que constem em sua interface pública, como acontece na linha 24, onde é passado o nome da loja remota e a lista de produtos através da chamada de um método via referência.

O trecho da linha 27 à linha 30 corresponde à criação e envio do agente de consulta. A variável *pos* é utilizada para receber um número aleatório que refira-se a um dos produtos da lista, a fim de que produtos diferentes sejam sorteados a cada execução (linha 29). O código de implementação para a classe em que se baseia o agente de consulta é apresentado nas linhas 49 a 120.

O método *goQuery*, invocado pela classe *Customer* (linha 30) realiza a ativação do agente de consulta. Este método recebe como argumentos o nome do produto a ser procurado e a lista de lojas remotas a percorrer. A linha 64 cria um agente contendo como métodos os métodos da classe *QueryAgent*. Ou seja, cria-se um agente baseado na classe *QueryAgent*. Sendo um agente, pode-se realizar o comando *moveTo* provido pela plataforma Voyager, o qual pode ser visto na linha 71. Este método causa a movimentação do agente criado para um local especificado, passado como argumento. O outro argumento deste comando representa uma indicação ao agente sobre qual método ele deve executar ao chegar ao seu destino. Dessa forma, quando o agente de consulta chega em uma loja, ele executa o método *atStore*, conforme informa o segundo argumento do comando *moveTo* da linha 71. No método *atStore*, como pode ser visto no trecho da linha 88 à linha 107, o agente de consulta compara o menor preço já encontrado para o produto que ele procura com o preço deste mesmo produto na loja em que está. Caso o preço da loja atual seja menor do que o menor preço já encontrado (linha 97), o agente de consulta armazena o preço encontrado (linha 99), guarda o nome da loja (linha 100) e guarda uma referência à loja (linha 101), a qual será usada posteriormente pelo agente de compra. Após ter percorrido todas as lojas da lista recebida como argumento e ter realizado as comparações de preços, o agente de consulta retorna ao local de origem com os resultados (linha 80). A recepção dos dados do agente de consulta em *Customer* é feita nas linhas 31 e 32.

O trecho entre as linhas 35, 36 e 37 da classe *Customer* refere-se à criação e envio do agente de compra. A classe base deste agente tem seu código apresentado nas linhas 121 a 161. O método da classe *BuyerAgent* invocado para ativar o agente é o método *goBuy* (linhas 129 a 151), o qual recebe como argumentos o nome do produto a comprar, uma referência à loja onde deve ser efetuada a compra e a quantidade a ser comprada. Na linha 136 é criado o agente contendo os métodos da classe *BuyerAgent* que deve realizar a compra. Como já visto, o comando *moveTo* faz com que o agente se mova para o local determinado e execute o método especificado ao chegar lá. O método a ser executado é o método homônimo ao executado pelo agente de consulta, ou seja, o método *atStore*. Neste método, o agente solicita a compra do produto na quantidade que lhe foi passada e recebe, como retorno, a quantidade comprada (linha 155). De posse da quantidade comprada, o agente retorna com os resultados ao local de origem (linha 144). Na linha 38 é recuperado o resultado da compra do agente de compra.

Os comandos das linhas 10 e 46 da classe *Customer* executam, respectivamente, a inicialização e a finalização da plataforma Voyager.

3.1.2 Implementação baseada em processos

A implementação da aplicação de lojas remotas baseada em processos foi desenvolvida utilizando KLAIM, pois esta linguagem apresenta as características principais do grupo que representa, além de ser uma linguagem de programação com uma sintaxe próxima das sintaxes de linguagens conhecidas (principalmente Pascal).

Programas em KLAIM são divididos nos chamados *nodes*, que nada mais são do que locais onde processos estão executando e que contêm um *tuple space* (vide 2.2.2). Os *nodes* registram-se em uma *Net*, que é um conjunto de *nodes* que podem se comunicar. Na verdade, *Net* é um processo especial de KLAIM que recebe registros de *nodes*. Dessa forma, todos os *nodes* registrados através desse processo podem trocar informações e/ou processos. É esse processo *Net* que fornece a transparência de localização em KLAIM, já que apenas ele possui o registro da localização física dos *nodes*. Também cabe informar que KLAIM implementa um conceito de localidade, o qual é usado pelos *nodes* e pelos processos executando neles para referenciar outros *nodes* ou processos de outros *nodes*. Existem dois tipos de localidades: a localidade física, que diz respeito ao identificador pelo qual um *node* é unicamente referenciado dentro da Net; e a localidade lógica, que é um nome simbólico dado a um *node*. Outros conceitos e detalhes da linguagem serão em seguida esclarecidos no decorrer da explicação da implementação desenvolvida. A Figura 2 apresenta o código para o principal *node* da aplicação e para os processos que realizam a função de agentes móveis.

```
1. rec QueryAgent [product : str, storeList : ts, retLoc : loc]
2. declare
3. var i, price, newPrice, amount : int;
4. var storeName, bestName : str;
5. var again : bool;
6. var nextLoc, screen, thisLoc, bestLoc : loc;
7. var productList : ts
8. begin
9. out ("-> Agente criado\n")@screen;
10.   price := 0;
11.   again := true;
12.   thisLoc := self;
13.   i := 1;
14.   out ("-> Percorrendo lojas\n")@screen;
15.   # Percorre lista de lojas remotas
16.   while (i < 4) do
17.   # Obtem proxima loja a visitar
18.   in (!storeName, !nextLoc)@storeList;
19.   out ("-> Migrando para ")@screen;
20.   out (storeName)@screen;
21.   out ("\n")@screen;
22.   # Migra para proxima loja remota
23.   thisLoc := nextLoc;
24.   go@nextLoc;
25.   out ("-> Agente de consulta na loja\n")@screen;

26.   # Obtem preco da loja para o produto procurado
27.   out ("prodlist")@self;
28.   in (!productList)@self;
29.   if read (product, !amount, !newPrice)@productList within 10000
```

```
30.   then
31.   # Compara preco atual com o preco da loja
32.   # Se preco atual for 0 (inicial) ou for maior do que
33.   # o preco encontrado, atualiza valor do preco atual
34.   out ("-> Preco encontrado para ")@screen;
35.   out (product)@screen;
36.   out (" = ")@screen;
37.   out (newPrice)@screen;
38.   out ("\n")@screen;
39.   if ((price = 0) or (price > newPrice)) then
40.   price := newPrice;
41.   bestLoc := thisLoc;
42.   bestName := storeName
43.     endif
44.   endif;
45.   i := i + 1
46.   enddo;
47.   out ("-> Retornando resultado\n")@screen;
48.   # Retorna ao local de origem a informacao do menor preco
49.   # encontrado e onde ele foi encontrado
50.   go@retLoc;
51.   out ("result", price, bestName, bestLoc)@self
52.   end;

53.   rec BuyerAgent [product : str, req_amount :int, storeLoc : loc,
   retLoc : loc]
54.   declare
55.   var amount : int;
56.   var screen : loc
57.   begin
58.   # Vai para a loja onde deve efetuar a compra
59.   go@storeLoc;
60.   out ("-> Agente de compra na loja\n")@screen;

61.   # Requisita a compra do produto na quantidade especificada
62.   out ("sale", product, req_amount)@self;
63.   in ("amount", !amount)@self;
64.   out ("-> Retornando resultado\n")@screen;
65.   go@retLoc;
66.   out ("amount", amount)@self
67.   end

68.   on
69.   nodes
70.   Customer :: {screen ~ CustomerScreen, Rem1 ~ RemoteStore1, Rem2 ~
   RemoteStore2, Rem3 ~RemoteStore3}
71.   class "Klava.NodeConsole" messages
72.   declare
73.   var stores : ts;
74.   var price, i, amount : int;
75.   var storeName : str;
76.   var again : bool;
77.   var storeLoc, thisLoc, store, bestLoc : loc
78.   begin
79.   # Cria lista de lojas remotas
80.   out (1, "Store1", Rem1)@self;
81.   out (2, "Store2", Rem2)@self;
82.   out (3, "Store3", Rem3)@self;
83.   i := 1;
84.   newloc(stores);
85.   again := true;
```

```
86.   while (again) do
87.   if read (i, !storeName, !storeLoc)@self within 1 then
88.   out (storeName, storeLoc)@stores;
89.   i := i + 1
90.   else
91.   again := false
92.   endif
93.   enddo;

94.   # Envia agente de consulta
95.   out (self)@self;
96.   in (!thisLoc)@self;
97.   eval (QueryAgent("monitor", stores, thisLoc))@self;

98.   # Recebe resultado do agente de consulta
99.   in ("result", !price, !storeName, !bestLoc)@self;
100.  out ("-> Melhor preco para monitor = ")@screen;
101.  out (price)@screen;
102.  out (" na loja\n")@screen;
103.  out (storeName)@screen;

104.  # Envia agente de compra
105.  eval (BuyerAgent("monitor", 5, bestLoc, thisLoc))@self;

106.  # Recebe resultado do agente de compra
107.  in ("amount", !amount)@self;
108.  out ("-> ")@screen;
109.  out (amount)@screen;
110.  out (" comprada(s) de monitor\n")@screen
111.  end
112.  endnodes
```

Figura 2. Código para o *node Customer* e para os processos móveis.

O código para o *node Customer* começa na linha 70, apresentando o nome do *node* e o seu *environment*. No *environment* são definidas as associações entre localidades lógicas e localidades físicas. Dessa forma, *RemoteStore1*, *RemoteStore2* e *RemoteStore3* são localidades físicas, ou *nodes* da *Net*, e *Rem1*, *Rem2* e *Rem3* são localidades lógicas associadas, respectivamente, a cada uma delas. Com isso, toda vez que referencia-se Rem2, por exemplo, está-se referenciando a localidade física RemoteStore2. A localidade lógica *screen* é uma localidade padrão da linguagem utilizada para se obter uma saída gráfica das mensagens geradas pela aplicação.

O trecho entre as linhas 79 e 93 cria a lista de lojas a serem visitadas. O comando *out* causa a adição de uma tupla (conjunto de valores) ao *tuple space* do *node*. Portanto, os comandos das linhas 80, 81 e 82 são usados para criar tuplas contendo um valor de índice, um nome de loja e uma referência à loja em questão (localidade lógica). Cabe salientar que todos os comandos básicos em KLAIM são sempre direcionados a alguma localidade lógica. Por isso, após cada comando, deve ser informado o local de execução do comando com um sinal @ seguido da localidade. A localidade lógica *self*, também uma localidade padrão da linguagem, referencia o local atual de execução. O comando da linha 84 cria uma nova localidade lógica chamada *stores* que é utilizada como uma coleção de tuplas, funcionando semelhantemente a uma lista de estruturas. Na verdade, o comando *newloc* cria uma nova localidade lógica, à qual é associado um novo *tuple space*. Cada comando *out* direcionado para esta localidade causa a adição da tupla passada como argumento ao *tuple space* deste local. Este recurso é utilizado porque não existe suporte à criação de tipos abstratos de dados em KLAIM; somente pode-se

simulá-los através desta utilização de localidades. Nas linhas 85 a 93 as tuplas criadas anteriormente são adicionadas ao *tuple space* da localidade recém criada. As linhas 95 e 96 são utilizadas para obter-se a identificação da localidade atual, que será passada como parâmetro ao novo processo a ser criado, a fim de permitir o seu retorno após cumprida sua tarefa.

A linha 97 apresenta o comando que causa a criação de um novo processo. O código que tal processo irá executar está descrito entre as linhas 1 e 52. Este processo corresponde ao agente móvel que, assim como o *QueryAgent* da implementação baseada em objetos, é o encarregado de percorrer as lojas constantes na lista e pesquisar qual delas possui o menor preço para um determinado produto. O processo *QueryAgent* recebe como parâmetros o nome do produto para qual ele deve pesquisar o menor preço, a lista de lojas a percorrer e a localidade de retorno dos resultados (que, no caso, é o local onde o processo *Customer* está executando).

Entre as linhas 16 e 46 estão os comandos para que o agente móvel obtenha cada loja da lista, vá para a loja remota, obtenha a lista de produtos, selecione o produto desejado, e compare o preço encontrado com o menor já obtido. O comando *in* da linha 18 realiza o contrário do comando *out*, anteriormente citado: se o comando *out* adiciona tuplas a um *tuple space*, o comando *in*, por sua vez, retira tuplas deste. Uma tupla só é retirada de um *tuple space* através de um comando *in* caso todos os componentes da tupla sejam compatíveis com os argumentos do comando. Dessa forma, o comando *in (!storeName, !nextLoc)@storeList* só retira uma tupla de *storeList* se no *tuple space* desta localidade houver uma tupla cujo o primeiro valor seja uma string e o segundo uma localidade. Se não houver uma tupla compatível o comando *in*, como já explanado, bloqueia até que uma tupla coerente com a tupla procurada passe a existir no *tuple space* consultado. O sinal *!* antes do argumento significa que, ao encontrar uma tupla compatível em *storeList*, *storeName* receberá o valor string da tupla encontrada e *nextLoc* receberá o valor localidade. Pode-se entender que o sinal *!* significa que está-se passando os argumentos por referência.

O comando *go* da linha 24 causa a movimentação do agente (processo) para a próxima loja. Os comandos das linhas 27 e 28 são usados para obter a lista de produtos da loja. Cabe salientar que as instruções de *out* e *in*, respectivamente das linhas 27 e 28, possuem instruções *in* e *out* correspondentes a serem executadas pela loja remota visitada, nas quais a loja remota obtém a solicitação da sua lista de produtos pelo agente (linha 27) e devolve o *tuple space* (*productList*) onde o agente pode obtê-la (linha 28). O trecho de 29 a 44 refere-se à obtenção das informações do produto procurado e a comparação do preço atual com o preço encontrado na loja. O comando *read* da linha 29 realiza o mesmo que o comando *in*, como uma diferença: o comando *read* não retira a tupla do tuple space, mas apenas obtém os seus valores e os compara com os valores passados como argumentos do comando. O comando *within*, usado dentro do comando de seleção, é utilizado para que a procura por uma tupla compatível com os argumentos dure somente o tempo especificado em milisegundos, já que o comando *in* é bloqueante e, caso não fosse encontrada uma tupla correspondente, a aplicação ficaria paralisada neste ponto. Com um *time-out* determinado, a aplicação permanece naquele ponto somente o tempo desejado pelo programador. Cabe citar que, assim como o comando *in*, o comando *read* também é bloqueante, enquanto que o comando *out* é não-bloqueante.

Terminada a tarefa de percorrer as lojas da lista, o agente retorna ao local de origem e cria uma tupla contendo os resultados (linhas 50 e 51). Neste momento, o processo *Customer*, o qual estava esperando pelos resultados do agente (linha 99), exibe os resultados obtidos e envia o agente *BuyerAgent* para efetuar a compra de um quantidade determinada do produto pesquisado na loja de menor preço. O código a ser

executado por esse processo está entre as linhas 53 e 67. O agente *BuyerAgent* recebe como argumentos o nome do produto a ser comprado, a quantidade a ser comprada, a loja na qual deve ser efetuada a compra e o local de retorno dos resultados. Ele move-se para o local da loja indicada, requisita a compra e obtém a quantidade comprada, informação que é retornada ao local de origem assim que ele ali chega.

Deve-se informar que as lojas remotas também são processos (*nodes*) executando. Seu código de implementação não é apresentado por parecer ser desnecessário à boa compreensão da aplicação descrita, da mesma forma como ocorreu com a implementação baseada em objetos, onde o código das lojas remotas também não foi apresentado pelo mesmo motivo. Também é importante citar que, como a referência a uma dada localidade lógica recebe um nome que é registrado no *node* especial *Net*, duas localidades registradas na mesma *Net* não podem ter o mesmo nome. Este nome é, na verdade, o nome do processo executando que representa uma localidade lógica. Devido a isso, diferentemente do que pode ser feito na implementação em Java, teve-se que implementar um processo diferente para cada loja remota, em vez de criar-se apenas um processo genérico que fosse instanciado para cada loja. Ou seja, teve-se que criar processos *RemoteStore1*, *RemoteStore2*, ..., *RemoteStoreN* e não apenas um processo *RemoteStore* que executasse em cada local onde havia uma loja. Como resultado disso, tem-se que não há possibilidade de instanciar processos, uma vez que cada processo registrado como localidade lógica em uma *Net* deve ter nome único dentro da mesma. Processos de mesmo nome podem ser registrados em *Nets* diferentes, mas processos de *Nets* diferentes não se comunicam.

3.1.3 Implementação baseada em funções

A representante esccolhida para as linguagens baseadas em funções foi Objective Caml, por apresentar facilidade na obtenção de seu compilador e fornecer boa e detalhada documentação, além de apresentar todas as características do grupo que representa.

Para a implementação baseada em funções com Objective Caml, convencionou-se utilizar, tal como na aplicação baseada em objetos (vide 3.1.1), uma plataforma que provê uma biblioteca, criada a partir da referida linguagem, que facilita o desenvolvimento de aplicações envolvendo mobilidade de código. A plataforma a ser utilizada neste caso é a Jocaml [36], a qual possui um pequeno conjunto de primitivas trazidas do Cálculo-Join [37], o qual pode ser visto como um extensão da programação funcional que suporta concorrência. Apesar de estender a idéia de programação baseada em funções, o Cálculo-Join traz também alguns conceitos relacionados ao Cálculo-π, tais como os canais. A grande diferença entre o Cálculo-Join e o Cálculo-π é a introdução da idéia de localidade. Jocaml, seguindo o modelo do Cálculo-Join, baseia-se em localidades (*locations*), as quais residem em um local físico e abrigam tanto agentes quanto outras localidades, e em canais (*channels*), que são vias de comunicação. As localidades podem ser estáticas (*sites*) ou móveis (*agents).* As localidades móveis é que dão a idéia de agentes móveis. Uma localidade pode conter outras localidades. As localidades são organizadas hierarquicamente como uma árvore: localidades estáticas são criadas na raiz da árvore, enquanto localidades móveis são criadas dentro das localidades estáticas e, portanto, são como folhas da árvore de hierarquia. Localidades móveis também podem conter outras localidades móveis. Quando uma localidade se movimenta, ela mantém seus canais e leva consigo todas as localidades contidas nela. Ou seja, uma movimentação de localidade pressupõe a movimentação de todos os integrantes da árvore de hierarquia presentes abaixo da referida localidade. Ao chegar

ao destino, a localidade pode conectar-se a uma localidade raiz (estática) ou a uma localidade de nível inferior. Alterações na árvore de hierarquia são transparentes ao programador.

Jocaml utiliza, assim como KLAIM possui a idéia do nodo *Net* (vide 2.2.2), um servidor de nomes centralizado em que identificações são registradas e obtidas. Este servidor de nomes provê a transparência de localidade. Dessa forma, o programador não precisa saber onde está, fisicamente, uma localidade para onde ele deseja mover uma localidade móvel ou onde está uma função que ele quer utilizar; basta a ele saber a identificação da localidade e/ou função e utilizá-la para pesquisar no servidor de nomes, que é o encarregado de saber a localização física dos componentes.

As implementações em Jocaml são compostas por uma seqüência de definições e chamadas de funções e/ou definições de localidades, onde podem ser incluídas definições de novas funções. Todas as definições, sejam de função ou de localidade, iniciam pela palavra reservada *let*. O que diferencia uma definição de função da de uma localidade é o uso da palavra reservada *loc*.

O código de implementação da localidade cliente da aplicação usada como exemplo é apresentado na Figura 3. Esta localidade possui várias funções definidas, as quais são apresentadas entre as linhas 1 a 21. O uso da palavra reservada *def* logo após a palavra *let* define uma função pertencente à localidade em que são definidas. Funções remotas são a forma utilizada para troca de informações entre localidades. Como já citado, em Jocaml há um servidor de nomes. Este servidor de nomes é que permite obter referências a funções e localidades remotas e é, portanto, responsável pela viabilização da comunicação entre localizações e da movimentação de agentes (localidades móveis). Para viabilizar isto tudo, são providas, pelo servidor de nomes, duas operações: uma operação de registro e uma de obtenção de função ou localidade. A operação *Ns.register* realiza o registro de uma função ou localidade através de um nome de referência, como o que é feito na linha 58 da Figura 3. A operação *Ns.lookup* possibilita obter-se uma função ou localidade registrada através de seu nome de referência, como na linha 1. Seguindo este modelo, obviamente impede-se que duas funções ou localidades possuam o mesmo nome de referência, apesar disso não gerar erro por parte da linguagem, já que o último registro de uma mesma função ou localidade é assumido. Cada localidade ou função, portanto, possui apenas uma instância registrada no *name server*.

A palavra reservada *vartype*, que aparece em algumas declarações e obtenções (*lookup*) de funções ou localidades, é usada para permitir o polimorfismo das funções. Esta palavra é substituída em tempo de compilação segundo inferência do compilador (compilador analisa o código e atribui um tipo ao retorno da operação de obtenção de registro de acordo com o resultado da análise feita). Quando uma função ou localidade é registrada, seu tipo é armazenado neste registro. Ao ser realizada a operação de recuperação deste registro, o tipo armazenado é comparado ao tipo inferido pelo compilador e, caso não sejam compatíveis, é gerada uma exceção.

```
(* Definicao de funcoes utilizadas *)
1. let rem1 = Ns.lookup "rem1" vartype
2. let rem2 = Ns.lookup "rem2" vartype
3. let rem3 = Ns.lookup "rem3" vartype
4. let q1 : int -> int = Ns.lookup "qtde1" vartype
5. let q2 : int -> int = Ns.lookup "qtde2" vartype
6. let q3 : int -> int = Ns.lookup "qtde3" vartype
7. let monitor1 : int -> int = Ns.lookup "monitor1" vartype
8. let monitor2 : int -> int = Ns.lookup "monitor2" vartype
9. let monitor3 : int -> int = Ns.lookup "monitor3" vartype
10.let def menor (p1, p2, s1, s2, n1, n2, num1, num2) = if p1 < p2 then
```

```
11.                                                    reply (p1, s1,
12.                                                    n1, num1) else
13.                                                    reply (p2, s2,
14.                                                    n2, num2)
(* Funcoes para bloqueio e desbloqueio *)
15.let def new_lock () =
16.  let def free! () | lock () = reply
17.  and unlock () = free () | reply in
18.  free () | reply lock, unlock
19.let lock, unlock = new_lock ()

20.let def ind x = reply x
21.let def sub (x, y) = reply (if x >= y then y else x)

(* Definicao da localizacao estatica do cliente *)
22.let loc home
23.  do {
24.      let def qs n = reply (if n = 1 then q1 0 else if n = 2 then q2 0
25.                      else q3 0) in

26.      Ns.register "home" home vartype;

(* Definicao da localizacao movel query_agent *)
(* Este agente procura, entre as lojas, o menor preco *)
27.       let loc query_agent
28.        do {
29.             lock ();
(* Move-se para loja remota 1 *)
30.             go rem1;
31.             let store = rem1 in
32.             let name = "Rem1" in
33.             let num = ind 1 in
34.             let price = monitor1 0 in
35.             print_string ("Preco na loja 1 = "^ string_of_int
36.                           price^"\n");
37.             flush stdout;

(* Move-se para loja remota 2 *)
38.             go rem2;
39.             let num2 = ind 2 in
40.             let price2 = monitor2 0 in
41.             print_string ("Preco na loja 2 = "^ string_of_int
42.                           price2^"\n");
43.             flush stdout;

43.             let (price, store, name, num) = menor (price, price2,
44.                  store, rem2,  name, "Rem2", num, num2) in

(* Move-se para loja remota 3 *)
45.             go rem3;
46.             let num3 = ind 3 in
47.             let price3 = monitor3 0 in
48.             print_string ("Preco na loja 3 = "^ string_of_int
49.                           price3^"\n");
50.             flush stdout;

50.             let (price, store, name, num) = menor (price, price3,
51.             store, rem3, name, "Rem3", num, num3) in

(* Retorna para localizacao inicial com resultado *)
52.             let back : Join.location = Ns.lookup "home" vartype in
```

```
53.             go back;
54.             print_string ("Menor preco = "^ string_of_int price);
55.             flush stdout;
56.             print_string (" na loja "^ name ^"\n");
57.             flush stdout;

(* Salva resultados para posterior uso *)
58.             Ns.register "store" store vartype;
59.             Ns.register "num" num vartype;
60.             unlock ();
61.          }

(* Definicao da localizacao movel buyer_agent *)
(* Este agente realiza a compra de uma quantidade de um produto *)
62.     and buyer_agent
63.         do {
64.              lock ();

(* Recupera resultados do query_agent *)
65.              let store : Join.location = Ns.lookup "store" vartype in
66.               let num = Ns.lookup "num" vartype in

(* Move-se para loja remota *)
67.               go store;

(* Realiza compra *)
68.               let qtde = qs num in
69.               let total = sub (qtde, 20) in

(*  Volta para localização original com o resultado
70.               let back : Join.location = Ns.lookup "home" vartype in
71.               go back;
72.               print_string ("Total comprado = "^string_of_int
73.               total^"\n");
74.               flush stdout;
75.             }
76. in
77.    }
78.;;

79.Join.server()
```

Figura 3. Código para localidade cliente da implementação.

A implementação da localidade cliente começa na linha 22. Nas linhas 24 e 25 existe a definição de um função interna que será utilizada posteriormente. Na linha 26 é feito o registro da localidade no servidor de nomes.

Entre as linhas 27 e 70 está o código da localidade móvel *query_agent*, a qual tem a responsabilidade de mover-se entre as localidades onde estão as lojas remotas, procurando o menor preço para um produto (nesta implementação o produto procurado é monitor). A primeira instrução constante nesta localidade é da função *lock()*. Esta função, definida na linha 19, chama a função *new_lock()* (linha 15) que serve como função de sincronização. Ela recebe um primeiro pedido de bloqueio e deixa em suspenso qualquer pedido seguinte até que o primeiro pedido seja terminado, o que ocorre através da função *unlock()* (linha 19). Este mecanismo de bloqueio é usado para sincronizar dois processos (duas localidades realizando suas operações) paralelos e funciona semelhantemente a um semáforo. Ele é necessário porque todas as localidades definidas no mesmo processo (caso das localidades *query_agent* e *buyer_agent* na

implementação apresentada) executam suas operações paralelamente e, para garantir-se que a execução de duas localidades dependentes, como é a situação presente nesta aplicação, em que *buyer_agent* deve executar somente após *query_agent* ter finalizado, deve-se prover um mecanismo de bloqueio. Este bloqueio leva em consideração que a ordem de início de execução é a ordem em que as localidades aparecem no código. Dessa forma, *query_agent* será inicializado sempre antes de *buyer_agent*.

A instrução seguinte à instrução de bloqueio, executa o comando *go* (linha 30) que realiza a movimentação de *query_agent* para a localidade indicada (*rem1*), onde está a localidade que corresponde à loja remota 1. Os comandos das linhas 31 a 33 são usados para armazenar informações sobre a loja remota atual, assumindo que ela tem o menor preço para o produto procurado até o momento. A linha 34 utiliza a função remota *monitor1* (obtida pelo comando da linha 7) para descobrir o valor do produto monitor na loja atual e armazená-lo em *price*. Nas linhas 38 a 40 é realizado o procedimento descrito de movimentação para a loja remota 2 e obtenção do preço para o produto monitor. A linha 43 apresenta a utilização da função *menor* para definir se o preço obtido na loja atual é menor do que o menor preço encontrado até então. A função referida retorna um conjunto de valores que representam, respectivamente, o menor entre os dois preços comparados, a identificação da loja remota em que o preço foi encontrado, o nome da loja remota e o número de identificação da loja. Da linha 45 a linha 51 é efetuado o mesmo procedimento para o loja remota 3. Na linha 52 é obtida a referência da localidade original de *query_agent* e na linha 53 é realizada a sua movimentação para a localidade em questão, retornando com os resultados. As linhas 58 e 59 servem para registrar a referência à loja remota que contém o produto pelo menor preço e o seu número de identificação para posterior uso de *buyer_agent*. A última instrução (linha 60) serve para desbloquear *buyer_agent*, já que os dados de que ele necessitava para executar já estão disponíveis.

O código para a localidade móvel *buyer_agent* é apresentado entre as linhas 62 e 77. A primeira instrução faz com que a execução fique bloqueada até *query_agent* termine (realize o *unlock*). Após a liberação, os dados de resultado de *query_agent* são recuperados (linhas 65 e 66). A localidade *buyer_agent* é então movida para a loja remota que possui o menor preço para o produto monitor (linha 67) e requisita a compra de 20 unidades (linhas 68 e 69). A quantidade comprada é armazenada em *total*. Nas linhas 70 a 71 *buyer_agent* obtém a sua localidade original e retorna com os resultados.

A última instrução da localidade estática *home* (linha 79) serve para que esta localidade execute indefinidamente, mantendo os registros feitos por ela e pelas localidades internas a ela ativos.

4 Avaliação das Implementações Desenvolvidas

Para avaliação das linguagens utilizadas nas implementações e suas respectivas abordagens, serão utilizados alguns critérios, a fim de realizar uma análise sobre aspectos considerados importantes quanto à programação voltada a aplicações com mobilidade código.

Esses critérios são apresentados na seção 4.1. Na seção 4.2 são apresentadas as avaliações das implementações realizadas segundo os critérios estabelecidos na seção anterior.

4.1 Critérios de Avaliação

Os critérios aqui expostos baseiam-se em propriedades tidas como essenciais quando se tratam de linguagens de programação que dão suporte à mobilidade. Dessa forma, o estabelecimento desses critérios visa não só prover um meio de avaliar as abordagens estudadas, mas também uma forma de verificar se todas as exigências básicas de uma linguagem que suporta mobilidade são atendidas e como tais requisitos são supridos do ponto de vista do programador pelas linguagens usadas na implementações.

Os critérios a serem usados na avaliação das implementações são explanados a seguir:

- **Identificação:** Deve ser provida uma forma de identificar um componente, permitindo uma forma de referência unívoca a ele no ambiente distribuído;

- **Expressão de mobilidade:** Deve ser provida, ao programador, uma forma clara de declarar que um dado componente é móvel;

- **Componente móvel:** Deve ser fornecida uma entidade que apresente ou possa apresentar as características de um componente móvel, de acordo com a abordagem adotada;

- **Heterogeneidade:** Para que componentes móveis possam atingir toda a sua utilidade e capacidade, a construção dos componentes deve levar em consideração a possibilidade de realização de computações em locais com arquiteturas heterogêneas;

- **Transparência de localização:** Deve ser transparente ao usuário se uma aplicação está executando local ou remotamente;

- **Abstração de dados:** Deve ser fornecido algum tipo de construção que permita a criação de tipos abstratos de dados;

- **Tratamento de exceções:** Deve ser apresentada uma forma de identificar e tratar exceções, o que é algo importante para a execução de aplicações em um ambiente distribuído;

- **Suporte ao desenvolvimento:** Deve ser provido suporte ao desenvolvimento de aplicações com código móvel com mecanismos para auxílio na depuração.

4.2 Avaliação segundo os critérios

A partir dos critérios estabelecidos na seção anterior, esta seção apresenta uma avaliação das implementações realizadas em relação às linguagens de programação e às abordagens utilizadas. Cada critério é avaliado separadamente e, ao final, é apresentada uma tabela comparativa como resultado da avaliação desenvolvida.

4.2.1 Identificação

Identificar um componente em um ambiente distribuído é uma questão importante já que podem haver diversas aplicações contendo componentes com o mesmo nome (talvez instâncias de uma mesma aplicação) e que executam paralelamente. Nesse caso, deve ser possível referenciar-se um componente sem correr-se o risco de estar-se referenciando um outro componente de mesmo nome. Sendo essa uma questão chave quando se tratam de sistemas distribuídos, também é um fator essencial em relação a aplicações com código móvel.

As três linguagens utilizadas na implementação da aplicação exemplo fornecem mecanismos para identificar unicamente um componente. Java/Voyager baseia-se na idéia de localização através do uso do endereço ou nome do host em que um componente está executando, para identificá-lo entre diferentes locais, e do número da porta de comunicação utilizada, para identificá-lo local e remotamente. A porta, no caso de utilização da plataforma Voyager e sua biblioteca de primitivas, também serve para designar uma instância de execução da plataforma onde um objeto móvel pode ser recebido.

KLAIM e O'Caml utilizam a idéia de um servidor de nomes centralizado que é o responsável por identificar cada componente. A diferença entre as duas implementações desse servidor é que o servidor de nomes de KLAIM identifica cada instância pelo nome que designa o processo (dado pelo programador dentro do código) e gera uma exceção caso outro processo de mesmo nome tente registrar-se nesse servidor de nomes, enquanto o servidor de nomes de O'Caml permite que duas funções sejam registradas com o mesmo nome, mas mantém apenas o último registro feito para aquela função; ou seja, se duas funções forem registradas com o mesmo nome, o primeiro registro será sobreposto pelo último, o qual será o único registro válido.

4.2.2 Expressão de mobilidade

A idéia de ter-se uma forma de expressão de mobilidade refere-se à clareza quanto à identificação de componentes móveis. Isto é, deve ser possível ao programador visualizar, já na declaração de um componente, que o mesmo possui característica de mobilidade. Propiciar esta identificação de componentes móveis na declaração ajudaria ao programador a entender melhor uma aplicação que use mobilidade, facilitando a compreensão e, possivelmente, a reutilização de código. Esta possibilidade é ainda mais interessante quando o programador que analisa o código da aplicação não está bem familiarizado com a idéia de mobilidade de código ou, mesmo que já tenha experiência no assunto, não conhece bem a linguagem e quer entender como se cria um componente móvel e como pode-se utilizá-lo. Este aspecto não depende da abordagem seguida, e sim da implementação da linguagem de programação.

Neste aspecto, as três linguagens não apresentam uma identificação clara de quais componentes são móveis e quais não o são. Nestas linguagens, o programador só identifica a característica de mobilidade quando um comando de *moveTo* (Java/Voyager) ou *go* (KLAIM e O'Caml/Jocaml) é executado em algum ponto do código, não sendo possível identificar um componente móvel na declaração do mesmo. Com isso, existe certa dificuldade em reconhecerem-se os componentes móveis e fica prejudicada a análise do código.

Vale salientar que a utilização da biblioteca Java provida pela plataforma Voyager fornece algumas características específicas para códigos móveis, tais como a necessidade de possuir uma interface e de implementar a interface de serialização

(*java.io.Serializable*). Apesar disso, não é possível dizer que as instâncias de uma classe que possui estas características são móveis, uma vez que somente o fato de possuir uma interface definida e ser serializável não determina que um objeto é móvel.

4.2.3 Componente Móvel

A questão do componente móvel está intimamente ligada à abordagem seguida, já que é ela que define como um componente móvel é representado. Segundo já visto anteriormente (vide seção 2), Java baseia-se em objetos, KLAIM em processos e O'Caml em funções. Dessa forma, um componente móvel em Java é um objeto e, portanto, para definir-se um componente móvel deve-se implementar uma classe cujas instâncias (objetos) podem ser movidas. Em KLAIM, um componente móvel é um processo que se movimenta. Em O'Caml, funções que executam como uma thread, definidas como localidades móveis, são os componentes móveis que movimentam-se entre localidades estáticas.

Quanto a questões de mobilidade, a diferença, em relação às abordagens, é a estrutura do componente que se move. A facilidade ou dificuldade que o programador pode ter utilizando qualquer uma das abordagens relaciona-se ao costume que o mesmo possui de programar utilizando um dos paradigmas. Na verdade, a programação baseada em objetos e em processos é bastante semelhante, tendo-se apenas a necessidade de adaptação à programação de processos, onde não existem métodos, e o código é executado sempre na mesma ordem. Ou seja, enquanto um objeto pode executar seus métodos em ordens diversas, reagindo a eventos de ativação, um processo executa o mesmo código na mesma ordem, como se fosse um método único. Dessa forma, caso queira-se que um processo reaja a eventos tal qual um objeto, é necessário explicitar em seu código comandos de testes de eventos (como um esquema de *polling*, por exemplo), dividindo-se o código em blocos de comandos que são executados ou não dependendo do resultado do teste de ocorrência de eventos.

A dificuldade maior é, provavelmente, na adaptação ao paradigma funcional, onde também não existem métodos, sendo o código executado como no caso dos processos, descrito há pouco, e não existe, como ocorre com objetos e processos, a manutenção de um estado interno. Não existem atributos, mas apenas variáveis e todo o trabalho com valores envolve o uso de funções. No caso de uma atribuição simples de um valor a uma variável, isto só se torna possível pela atribuição de um valor de retorno de uma função à variável em questão.

4.2.4 Heterogeneidade

A heterogeneidade, que supõe a capacidade de componentes movimentarem-se entre locais com arquiteturas e/ou plataformas diferentes, é uma característica que restringe-se às linguagens de programação. As abordagens determinam apenas como um componente é estruturado e como os componentes se comunicam. Como o componente é executado em cada local e se ele pode ser criado em um local e ser executado em outro local que possui características diferentes, é determinado pela implementação da linguagem.

A capacidade de executar em ambientes heterogêneos se faz importante tratando-se de mobilidade de código, quando tem-se a idéia de que é possível criar-se um componente móvel e fazê-lo executar em diversos locais na rede, não importando como eles sejam constituídos. Acerca desse quesito, Java, por sua característica de portabilidade, é, dentre as três linguagens vistas, a que melhor provê suporte à

heterogeneidade. O que se deve, em grande parte, à utilização de um código intermediário (os *bytecodes*) que permite a execução de um componente em qualquer local que possua um interpretador Java, o qual traduz os *bytecodes* para a linguagem da máquina local. KLAIM, a fim de prover heterogeneidade, utiliza-se de uma biblioteca que transforma o código KLAIM em código Java. Esta biblioteca, chamada de Klava, possui a implementação em Java das primitivas de KLAIM e permite a execução de um componente KLAIM em ambientes heterogêneos. Portanto, KLAIM não fornece sozinha a capacidade de heterogeneidade. O'Caml também não possui capacidade de prover componentes independentes de arquitetura mas, ao contrário de KLAIM, não utiliza qualquer mecanismo para obter esta característica. Componentes O'Caml só executam em ambiente Unix e, recentemente, foi anunciada uma versão que deverá executar sobre ambiente Windows NT.

4.2.5 Transparência de Localização

A transparência de localização diz respeito a deixar a cargo da linguagem utilizada questões quanto a uma comunicação ser local ou remota e onde um determinado componente está fisicamente executando. Com isto, o programador não precisa se preocupar com esses aspectos e trabalha como se todos os componentes fossem locais. Quando uma movimentação é feita, o programador não precisa saber onde o componente está fisicamente, precisando apenas possuir uma forma de referenciá-lo.

KLAIM e O'Caml possuem essa característica. Apesar de suas abordagens (Cálculo-π e Cálculo-λ, respectivamente) não apresentarem o conceito de localização, tal conceito foi adicionado pelas linguagens. Para tanto, elas fornecem um servidor de nomes central que, além de permitir a identificação dos componentes da forma descrita em 4.2.1, provê a transparência de localização desejada. Isto porque o local físico em que um componente está executando é uma informação controlada apenas pelo servidor e é obtida quando o componente se registra. Assim, quando um componente quer comunicar-se com outro componente ou mover-se para o local onde outro componente está executando, ele apenas utiliza uma referência ao componente (nome de registro no caso de O'Caml) ou ao local onde ele se encontra (localidade lógica de KLAIM).

Já em Java, a transparência de localização na comunicação é fornecida pela utilização de uma biblioteca de uma plataforma de suporte à mobilidade de código. A independência de localização é obtida pela utilização das estruturas *proxy*, discutidas em 2.1.1. Como toda a comunicação de um objeto móvel se dá através da *proxy*, que é local ao objeto que se comunica com ele, apenas a *proxy* mantém o registro do local físico em que o objeto móvel se encontra. Em relação à movimentação de componentes, ocorre o mesmo: quando um objeto quer movimentar-se para um determinado local ele utiliza uma *proxy* para um objeto que esteja executando naquele local para obter o endereço físico. Salientem-se duas coisas: a primeira é que, como dito, um objeto para comunicar-se ou mover-se possui *proxies* a outros objetos, através das quais ele envia mensagens e obtém os endereços físicos necessários a sua movimentação; segunda, existem casos de plataformas que permitem a utilização de movimentação com ou sem transparência de localização, como é o caso da Voyager, onde pode mover-se um componente para executar em um local específico, informando-se o endereço físico e a porta de comunicação da instância da plataforma que irá recebê-lo, ou apenas possuindo uma referência a um componente (*proxy*) e informando que quer mover-se o componente para o local onde este outro está executando.

4.2.6 Abstração de Dados

Poder-se criar tipos abstratos de dados é um recurso bastante útil e, por vezes, necessário. Tipos abstratos de dados supõem a possibilidade de terem-se *arrays*, estruturas, listas e outras formas de expandir os tipos básicos da linguagem. Um tipo abstrato de dados facilita o armazenamento de grande quantidade de informações e de dados de diferentes tipos, além de agrupar informações que individualmente podem não conter significado útil. Um exemplo é a possibilidade de criar-se, para a aplicação apresentada no capítulo 3, uma lista de lojas remotas a visitar, em que cada elemento da lista seria uma estrutura contendo o nome da loja, uma referência ao local onde a loja está e o preço obtido para o produto procurado nesta loja. Isto torna mais fácil relacionar os valores obtidos para uma mesma loja, já que eles estão agrupados como um elemento da lista, e também auxilia a utilização desses valores pela facilidade em percorrer e acessar dados da lista.

A questão da possibilidade de criarem-se tipos abstratos de dados depende muito da linguagem, mas também é influenciada pela abordagem seguida. A orientação a objetos já pressupõe tipos abstratos de dados, uma vez que um objeto pode agrupar muitos valores, que formam seus atributos. Por isso, Java é muito bem provida de mecanismos para permitir a criação de tipos abstratos e a utilização de alguns já presentes no pacote da linguagem.

O Cálculo-π não prevê a utilização de tipos abstratos de dados, já que possui apenas processos e canais. Os processos trocam dados através dos canais, que suportam apenas tipos básicos, como inteiros, caracteres e strings. Seguindo a sua abordagem, KLAIM não possui tipos abstratos de dados e utiliza apenas tipos básicos. A idéia de um tipo abstrato de dados, no entanto, pode ser simulada pela utilização de tuplas, onde cada tupla representaria uma estrutura que poderia conter valores de múltiplos tipos; um conjunto de tuplas de mesmo formato e contendo um valor que seria usado tal como um índice, poderia ser visto como um *array* ou um lista.

Na utilização de funções, os tipos básicos são os únicos permitidos como parâmetros e retornos de funções. Variáveis também só podem assumir valores dos tipos básicos. O'Caml possui a idéia de uma tupla (conjunto de valores de múltiplos tipos) ser retornada por uma função, mas os valores deixam de ser trabalhados como um conjunto de valores logo após o retorno da função, sendo tratados como valores independentemente a partir de então. Com isso, O'Caml não proporciona qualquer forma de criarem-se tipos abstratos de dados. É importante dizer que leva-se em consideração, nessa análise, a programação funcional sobre O'Caml, visto que, segundo relatado em 2.3.2, O'Caml suporta programação multiparadigma, permitindo criar um programa que utiliza funções e que trabalha também com objetos. No caso de trabalharem-se com objetos, obviamente existem tipos abstratos de dados, mas aqui está-se discutindo O'Caml segundo a programação funcional apenas.

4.2.7 Tratamento de Exceções

A possibilidade de identificação e tratamento de exceções em aplicações com mobilidade de código é um aspecto bastante desejável, dado que, caso ocorra uma falha com um componente móvel durante a execução de uma aplicação, é necessário que o programador tome conhecimento deste fato a fim de corrigir ou tentar corrigir o erro acontecido. Algumas vezes, a falha não é crucial na execução da aplicação e, sendo possível tratá-la, há uma forma de não interromper a execução.

Em Java, o tratamento de exceções pode ser feito pelo uso de classes de exceções que estendam a classe *java.lang.Exception*, em conjunto com o comando *try-catch*. Todos os comandos colocados dentro do bloco *try* são testados e, caso algum deles resulte em uma exceção, o bloco de comandos *catch* é executado. O'Caml provê a função *failwith*, a qual recebe como argumento uma string. Esta função é colocada no bloco de comandos de uma outra função e, caso ocorra alguma exceção nesta última, a string passada como argumento é exibida como uma mensagem de erro. Ou seja, em O'Caml só é possível identificar uma exceção, mas não tratá-la. Para KLAIM, a biblioteca Klava fornece, como em Java, o comando *try-catch* e uma classe denominada *KlavaException*. Ou seja, na verdade, KLAIM utiliza o próprio tratamento de exceções de Java, apenas tendo a sua própria classe de exceções. Logo, KLAIM não possui, por si só, mecanismos para tratamento de exceções.

4.2.8 Suporte ao Desenvolvimento

Por suporte ao desenvolvimento entende-se um conjunto de mecanismos que auxiliem ao programador que desenvolve aplicações com mobilidade de código. Oferecer tratamento de exceções já é um bom passo em direção ao suporte ao desenvolvimento, só que deve ser provido mais do que isso. Deseja-se poder ter um código compilado/interpretado que, caso contenha erros, tenha esses erros claramente descritos. Muitas vezes, a dificuldade na depuração de um código é compreender o que uma mensagem de erro gerada significa. Há ocasiões em que esta falta de compreensão por parte do programador vem da sua pouca familiaridade com o próprio paradigma de programação da linguagem, visto que cada abordagem pode gerar erros específicos do tipo de programação desenvolvida. Isto é, em uma linguagem funcional não ter-se-á erros relativos a acessos a métodos privados, por exemplo. Outra questão são os erros na execução, quando o programador precisa saber se seu componente realmente executou o que deveria e nos locais onde deveria. Permitir que se saiba se um dado componente assumiu o comportamento determinado para ele ou se, por algum motivo, ele parou de executar em um ponto do caminho é um aspecto bastante interessante e importante em relação a códigos móveis. Dado que um objeto móvel pode entrar em um local e utilizar os recursos desse local, deve ser possível controlar a sua execução de forma a impedir que um objeto utilize recursos indefinidamente.

Nenhuma das três linguagens oferece mecanismos que possam ser caracterizados como bons suportes ao desenvolvimento. Mensagens de erro de Java são, algumas vezes, incompreensíveis à primeira vista e, por isso, requerem um bom conhecimento da linguagem para um melhor entendimento. O'Caml gera mensagens de erro inteligíveis apenas quando já se tem alguma noção de seu modelo de programação. Em geral, as mensagens dizem respeito a erros quanto a tipos de retornos ou de parâmetros de funções do tipo "esperava-se *'a metatype -> 'b* mas encontrou-se *int -> int*", que, após algum estudo da linguagem, descobre-se significar que o uso de uma função remota, cujo o tipo de retorno não se conhece localmente, está incorreto ou que é necessário definirem-se, explicitamente, os tipos dos parâmetros e dos valores de retorno da função desejada. Mensagens de erro durante a execução também são, certas vezes, não muito compreensíveis. KLAIM procura fornecer informações completas sobre tudo o que acontece em sua *Net*, mas o excesso de mensagens, muitas das quais não trazem informações relevantes, acaba causando confusão quanto ao que é uma mensagem de erro e o que é uma mensagem ordinária.

Em relação a mecanismos de controle sobre a execução de componentes, não existe nenhum disponível nas linguagens vistas. Dessa forma, o consumo de recursos e

o comportamento dos componentes móveis não podem ser monitorados, tornando difícil o processo de depuração de aplicações que possuem componentes que se movimentam.

Como forma de resumir a análise feita sobre linguagens e respectivas abordagens, segue uma tabela (vide Tabela 1) contendo os critérios adotados na avaliação e o resultado para cada uma das linguagens.

	Identif.	**Expressão de Mob.**	**Comp. Móvel**	**Heterog.**	**Transp. de Localiz.**	**Abstração de Dados**	**Trat. de Exceções**	**Suporte ao Desenv.**
Java (Voyager)	Sim	Não	Objeto	Sim	Sim	Sim	Sim	Não
KLAIM	Sim	Não	Processo	Não[1]	Sim	Não[2]	Sim[3]	Não
O'Caml (Jocaml)	Sim	Não	Função	Não	Sim	Não	Não[4]	Não

Tabela 1. Tabela comparativa entre as linguagens de programação trabalhadas.

5 Conclusão

O trabalho desenvolvido visou o estudo de algumas das linguagens de programação com suporte à mobilidade código, com o intuito de identificarem-se os tipos de abordagens utilizadas e trabalhar-se com uma linguagem de cada tipo de abordagem encontrada. O objetivo geral foi cumprido com o estudo das linguagens descritas na seção 2, agrupando-as segundo as abordagens identificadas, apresentadas no mesmo capítulo, e realizando-se um exemplo de implementação com uma representante de cada grupo, como apresentado na seção 3.

Conforme a avaliação feita na seção 4 sobre as implementações realizadas, fica claro que, apesar da multiplicidade de linguagens existentes que suportam a construção de aplicações móveis e das possibilidades de abordagens a serem seguidas, carecem-se de linguagens que supram por completo o que é necessário para o bom desenvolvimento de aplicações contendo código móvel. Algumas linguagens enfocam mais certos aspectos do que outros, dificultando a escolha, pelo programador, da linguagem mais adequada. A escolha passa a não ser, talvez, pela melhor linguagem mas pela linguagem que forneça a maioria dos requisitos desejados para o tipo de aplicação que se pretenda desenvolver. Com isto, o programador pode ter que se deparar com diferentes abordagens a cada nova aplicação que queira desenvolver.

Um exemplo de que a escolha entre linguagens pode se tornar diretamente ligada ao tipo de aplicação a construir é o caso da aplicação descrita na seção 3, a qual foi utilizada para verificar as diferenças entre as abordagens identificadas durante o estudo. Nesta aplicação, a possibilidade de usarem-se tipos abstratos de dados, ter-se transparência de localização e poder-se manter um conjunto de valores de atributos

[1] Heterogeneidade é atingida através da biblioteca Klava, que traduz primitivas KLAIM em código Java, dando portabilidade ao componente KLAIM.

[2] Tipos abstratos de dados podem ser simulados.

[3] Através da biblioteca Klava.

[4] Apenas provê identificação de exceções.

facilita, não só a compreensão do código, como também o próprio desenvolvimento deste. Poderia ter-se uma outra aplicação em que esses aspectos não fossem tão enfatizados e fosse possível utilizar-se uma linguagem com características diferentes, que provisse os fatores mais essenciais.

Alguns do aspectos realçados ou desconsiderados em cada linguagem estão relacionados com a abordagem seguida, mas poder-se-iam criar formas de prover os quesitos desejáveis em nível de linguagem de programação sem, no entanto, perder-se a ligação com os conceitos essenciais do paradigma implementado. Isto pode ser verificado em KLAIM, onde os aspectos de programação seguem o paradigma adotado, baseado em processos, mas mesmo assim a linguagem fornece recursos a mais do que o que seria esperado segundo a sua abordagem, principalmente pelo fato de usar mecanismos de Java, através do pacote Klava, para preencher algumas lacunas de aspectos desfavorecidos segundo a programação baseada em processos. Outra forma de englobar todos os aspectos desejados de uma linguagem de programação com suporte à mobilidade seria, como ocorre em O'Caml, a possibilidade de programação multiparadigma, o que poderia trazer a soma dos benefícios apresentados por mais de um paradigma. Neste trabalho, a discussão acerca de O'Caml resumiu-se à programação funcional por questões de critério de classificação, mas a implementação realizada com essa linguagem poderia ter sido facilitada pela combinação da abordagem usada com o paradigma de orientação a objetos, também fornecido por O'Caml.

Conclui-se, portanto, que, em aspectos gerais, as dificuldades ou facilidades de trabalhar-se com uma abordagem dependem do nível de conhecimento sobre a linguagem adotada e sobre o próprio modelo de programação do paradigma. Dessa forma, ao desenvolver uma aplicação móvel, o programador deve levar em consideração os seus conhecimentos e aptidões e os aspectos mais desejados para a aplicação que ele deseja construir.

6 Bibliografia

[1] FUGGETA, A., PICCO, G., VIGNA, G.. Understanding Code Mobility, *Transactions On Software Engineering*, IEEE, vol. 24, 1998, pp. 342-361. Disponível em na Internet em http://swarm.cs.wustl.edu/~picco/listpub.html

[2] WOJCIECHOWSKI, P., SEWELL, P.. Nomadic Pict: Language and Infrastructure Design for Mobile Agents, In *Proceedings of the ASA/MA'99*, 1999.

[3] PIERCE, B., TURNER, D.. *Pict: A Programming Language Based on the Pi-Calculus*, Tech. Report 476, Indiana University, 1997. Disponível na Internet em http://www.cis.upenn.edu/~bcpierce/

[4] KNABE, F. C.. *Language Support for Mobile Agents*. Ph.D. thesis, Carnegie Mellon University, 1995. Disponível na Internet em http://agents.umbc.edu/papers/knabe.shtml

[5] GOSLING, J., McGILTON, H.. *The Java Language Environment - A White Paper*. SunMicrosystems, 1996. Disponível na Internet em http://java.sun.com/doc/language_environment/

[6] DE NICOLA, R., FERRARI, G., PUGLIESE, R.. Klaim: a Kernel Language for Agents Interaction and Mobility, *Transactions on Software Engineering*, vol. 24, nº 5, IEEE Computer Society, 1998, pp. 315-330.

[7] OUSTERHOUT, J. K.. *Tcl and The Tk Toolkit.* Addison-Wesley, Reading, MA, USA, 1994.

[8] GIACALONE, A., MISHRA, P., PRASAD, S.. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, vol. 18, nº 2, 1989, pp. 121-160.

[9] COURTNEY, A.. Phantom: An Interpreted Language for Distributed Programming. *In USENIX Conference on Object-Oriented Technologies (COOTS)*, Monterey, CA, 1995.

[10] CARDELLI, L.. *Obliq: A Language with Distributed Scope*. DEC Systems Research Center, 1994.

[11] DI MARZO, G., MUHUGUSA, M., TSCHUDIN, C., et al.. The Messenger Paradigm and Its Implication on Distributed Systems. In *Workshop on Intelligent Computer Communication (ICC)*. Disponível na Internet em http://cuiwww.unige.ch/tios/msgr/msgr.html

[12] General Magic. *Telescript Language Reference*. General Magic, 1995.

[13] NICOLAOU, A.. A Survey of Distributed Languages. 1996. Disponível na Internet em http://www.cgl.uwaterloo.ca/~ anicolao/termpaper.html

[14] WYANT, G.. Introducing Modula-3. *Linux Journal*, 1994.

[15] KNABE, F. C.. An Overview of Mobile Agent Programming. *Proc. of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, number 1192 in Lecture Notes in Computer Science, Stockholm, Sweden, 1996.

[16] THOMSEN, B., LETH, L., PRASAD, S., et al. *Facile Antigua Release Programming Guide*. Technical Report ECRC 93-20, European Computer-Industry Research Centre, 1993. Disponível na Internet em http://www.ecrc.de/research/projects/facile/report/report.html

[17] OUSTERHOUT, J. K.. Tcl: Na Embeddable Command Language. In *Proceedings of USENIX Conference*, 1990.

[18] McCABE, F. G., CLARK, K. L.. April – Agent PRocess Interaction Language. In: *Intelligent Agents*, ed. Jennings & Wooldridge, LNCS, vol. 890, Springer-Verlag, 1995.

[19] ROSSUM, G. V.. *Python Language Reference Manual*. CWI Report, 1992. Disponível na Internet em http://www.python.org/~guido/Publications.html

[20] TSCHUDIN, C. F.. *An Introduction to the M0 Messenger Language*. University of Geneva, Swiss, 1994.

[21] CUGOLA, G., GHEZZI, C., PICCO, G. P., et al. Analyzing Mobile Code Languages. In *Mobile Object Systems: Towards the Programmable Internet*, Lecture Notes on Computer Science, vol. 1222, 1997, pp. 93-111. Disponível na Internet em http://swarm.cs.wustl.edu/~picco/listpub.html

[22] CUGOLA, G., GHEZZI, C., PICCO, G. P., et al. A Characterization of Mobility and State Distribution in Mobile Code Languages. In *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, 1996, pp. 309-318. Disponível na Internet em http://swarm.cs.wustl.edu/~picco/listpub.html

[23] MATTHES, F., MÜSSIG, S., SCHMIDT, J. W.. *Persistent Polymorphic Programming in Tycoon: An Introduction*. Technical Report, Fachbereich Informatik Universität Hamburg, 1993.

[24] JOHANSEN, D., VAN RENESSE, R., SCHNEIDER, F. B.. *An Introduction to the TACOMA Distributed System – Version 1.0*. Technical Report 95-23, Dept. of Computer Science, University of Tromsø and Cornell University, Tromsø, Norway, 1995.

[25] APPEL, A. W., MacQUEEN, D. B.. Standard ML of New Jersey. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, number 528 in Lecture Notes in Computer Science, Passau, Germany, 1991, pp. 1-13.

[26] THORN, T.. Programming Languages for Mobile Code. *ACM Computing Surveys*, vol. 29, no. 3, 1997, pp. 213-239.

[27] MARTINEZ, A. L.. *An Investigation of Mobile Code Languages*. CS-63101 Course Project, Kent State University. Disponível na Internet em http://aegis.mcs.kent.edu/~amartine/mcl2.htm

[28] LEROY, X.. *Objective Caml*. 1997. Disponível na Internet em http://pauillac.inria.fr/ocaml/

[29] MILNER, R., PARROW, J., WALKER, D.. A Calculus of Mobile Processes (Parts I and II). *Information and Computation*, no. 100, 1992, pp. 1-77.

[30] GELERNTER, D.. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, 1985, pp. 80-112.

[31] MILNER, R.. *Communication and Concurrency*. Prentice Hall International, 1989.

[32] BETTINI, L.. *Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile*. Tesi di Laurea, Dipartimento di Sistemi e Informatica, Università di Firenze, Italia, 1998.

[33] CLARK, K. L., McCABE, F. G.. *Distributed and Object Oriented Symbolic Programming in April*. Technical Report, Dept. of Computing, Imperial College, London, 1994.

[34] DE NICOLA, R., FERRARI, G., PUGLIESE, R.. Programming Access Control: The KLAIM Experience. In *Proceedings of CONCUR'2000*, LNCS, 2000.

[35] FOURNET, C., GONTHIER, G., LÉVY, J., et al. *A Calculus of Mobile Agents*. In *Proceedings of CONCUR'96*, 1996.

[36] CONCHON, S., LE FESSANT, F.. JoCaml: Mobile Agents for Objective-Caml. In *Ptroceedings of ASA/MA'99*, 1999.

[37] FOURNET, C., GONTHIER, G.. The Reflexive Chemical Abstract Machine and The Join-Calculus. In *Proceedings of the 23rd ACM Symposium on Principals of Programming Languages*, pages 372-385, St. Petesburg Beach, Florida, 1996.

[38] MILNER, R.. The Polyadic π-Calculus: a Tutorial. In *Proceedings of the 1991 Marktoberndorf Summer School on Logic and Algebra of Specification*, 1991.

[39] HUDAK, P.. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys*, vol. 21, no. 3, 1989, pp. 359-411.

[40] CURCH, A.. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, N. J., 1941.

[41] DORWARD, S., PIKE, R., PRESOTTO, D. L., et al. The Inferno Operating System. *Bell Labs Technical Journal*, vol. 2, no. 1, 1997, pp. 5-18.

[42] MOCKAPETRIS, P.. *Domain Name – Concepts and Facilities*. RFC 1034. USC/Information Sciences Institute, 1987.