Introduzindo Abstrações para Computação Móvel em Linguagens Orientadas por Objeto

Marco Túlio de Oliveira Valente

Departamento de Ciência da Computação, Pontifícia Universidade Católica de Minas Gerais, E-mail: mtov@dcc.ufmg.br

Roberto da Silva Bigonha, Antônio Alfredo Ferreira Loureiro, Mariza Andrade da Silva Bigonha

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, E-mail: {bigonha,loureiro,mariza}@dcc.ufmg.br

Abstract

Recently, the notion of mobile computation has been proposed as an alternative to the construction of distributed applications in the Internet. Some processes calculi have already been designed having in mind this style of computation, being the ambient calculus of Cardelli and Gordon the most distinguished among them. However, to transform these kind of distributed applications in a reality in the Internet, it is necessary the design of programming languages containing abstractions for mobile computation. This article shows then a proposal for the introduction of these kind of abstractions in object oriented languages.

1 Introdução

Atualmente, uma das principais dificuldades para a disseminação de aplicações distribuídas que explorem todo poder computacional da Internet é a ausência de abstrações que apóiem a implementação das mesmas [5]. As abstrações tradicionalmente usadas no desenvolvimento de sistemas cliente/servidor, como RPC [3] e CORBA [12], foram propostas para redes sem problemas críticos de desempenho ou de disponibilidade, como redes locais ou corporativas. No entanto, em uma rede mundial, aberta e descentralizada como a Internet, estes dois problemas ganham tamanha importância que qualquer abstração para computação nesta rede deve ser capaz de tratar e, se possível, atenuar os efeitos dos mesmos sobre as aplicações. Assim, abstrações para programação

distribuída na Internet devem propiciar o desenvolvimento de sistemas que satisfaçam aos seguintes requisitos de operação:

- capacidade de suportar flutuações na largura de banda devido a congestionamentos na rede;
- capacidade de operação desconectada da rede, já que não é razoável supor conectividade ininterrupta, principalmente no caso de dispositivos computacionais móveis, como *notebooks* e assistentes pessoais;
- capacidade de execução nas diversas arquiteturas e sistemas operacionais que existem em uma rede mundial como a Internet;
- capacidade de instalação automática nas estações clientes, evitando assim que esta atividade tenha que ser manualmente realizada nos potenciais milhares de clientes de uma aplicação Internet.

Recentemente, o conceito de computação móvel foi proposto como uma solução para implementação deste tipo de sistema [5]. Segundo este conceito, a execução de uma aplicação não precisa mais se restringir a um único nodo da rede, podendo migrar por vários nodos de acordo com as necessidades da tarefa que se propõe a realizar [1] . O conceito implica, portanto, em mobilidade de estado, isto é, tanto do código da aplicação como de seus dados. Argumenta-se que este estilo agressivo de mobilidade contribui para diminuir a dependência das aplicações em relação a falhas na rede e para permitir operação em modo desconectado.

Atualmente, já existem alguns modelos teóricos que formalizam as noções básicas de computação móvel. Dentre eles, destaca-se o Cálculo de Ambientes [6], um cálculo de processos desenvolvido por Cardelli e Gordon especificamente para este fim. No entanto, sendo um modelo teórico, o Cálculo de Ambientes possui apenas as abstrações fundamentais para especificação de computação móvel, não sendo recomendado o seu uso em aplicações práticas. Portanto, para que sejam efetivamente construídas aplicações baseadas neste formalismo, torna-se necessário o projeto de linguagens de programação que incorporem as abstrações introduzidas pelo mesmo. Tais linguagens são chamadas de *Wide Area Languages* [5].

Neste trabalho, propõe-se a introdução destas abstrações em uma linguagem orientada por objetos. Ou seja, em vez de se projetar uma linguagem de programação que tenha o Cálculo de Ambientes como único modelo computacional, optou-se por uma solução híbrida, onde as aplicações continuam sendo constituídas por um conjunto de objetos e apenas as construções para comunicação e mobilidade são inspiradas no cálculo de Cardelli e Gordon. Como não exige

[1] Neste artigo, computação móvel é usado como tradução do termo inglês *mobile computation*. Ressalte-se que existe ainda a expressão *mobile computing*, usada para designar o caso em que os elementos da rede é que são móveis, como ocorre nas redes sem fio.

o domínio de um novo modelo computacional, acredita-se que esta abordagem é a mais adequada para induzir o surgimento de aplicações distribuídas na Internet baseadas no conceito de computação móvel.

Descreve-se a seguir a estruturação deste artigo. Na Seção 2, é feita uma descrição do conceito de *Wide Area Languages*, destacando-se os princípios e as características que uma linguagem deve possuir para merecer esta classificação. Na Seção 3, define-se uma linguagem baseada em objetos bastante simples, na qual são incorporadas em seguida diversas abstrações para computação móvel. Ainda nesta seção, à medida que são apresentadas as construções para computação móvel, mostram-se também alguns exemplos de uso das mesmos, incluindo aplicações como *plug ins*, agentes móveis e um sistema usado pelo comitê de programa de uma conferência. Na Seção 4, mencionam-se alguns trabalhos relacionados com o estilo de mobilidade proposto neste artigo. Por último, a Seção 5 apresenta as conclusões e propostas de trabalhos futuros.

2 *Wide Area Languages*

Chama-se de *Wide Area Language* (WAL) uma linguagem que possui construções semanticamente compatíveis com os princípios do Cálculo de Ambientes e, por conseqüência, de uma rede com as características da Internet [5]. Dentre estes princípios, os mais importantes são os seguintes:

- Completeza: uma WAL deve possuir poder de expressão suficiente para permitir a implementação de programas normalmente encontrados na Internet, como *applets*, *plug ins*, agentes móveis etc.
- Consistência: uma WAL *não* deve possuir construções que pressuponham *action-at-a-distance* ou conectividade contínua.

Action-at-a-distance, uma característica comum em linguagens para redes locais, designa a capacidade de se acessar recursos remotos de forma transparente, independentemente da localização dos mesmos.

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

- Uma WAL deve permitir a migração entre nodos da rede de estruturas hierarquicamente organizadas que contenham tanto dados como computações. Esta característica diferencia uma WAL de linguagens que permitem, por exemplo, apenas a migração de objetos individuais.
- Uma WAL deve permitir comunicação apenas entre entidades que compartilhem a mesma localização. Para não pressupor *action-at-a-distance*, comunicação entre entidades remotas deve ser implementada através de mobilidade.

- Uma WAL deve acessar suas entidades através de nomes com significado único em toda a rede e não através de endereços físicos de memória. Com isso, a mobilidade destas entidades não fica restringida por "ligações estáticas" entre as mesmas e seu contexto de execução.
- Em uma WAL, o acesso a qualquer entidade deve ocorrer com uma semântica de bloqueio, isto é, caso uma entidade não esteja presente localmente no momento em que é referenciada, a execução permanece bloqueada até que a mesma se torne disponível. Este tipo de semântica torna mais simples a expressão de *linkedição* e reconfiguração dinâmicas.
- Uma WAL deve implementar o acesso a recursos via *binding* dinâmico, permitindo assim que uma entidade ao migrar para um novo nodo da rede tenha automaticamente restabelecido o acesso aos recursos de que necessita.

3 As Abstrações Propostas

3.1 A Linguagem Utilizada

Conforme afirmado na introdução, as abstrações propostas neste trabalho serão definidas para uma linguagem orientada por objetos. No entanto, em vez de se utilizar uma linguagem tradicional, baseada em classes, como C++ ou Java, optou-se por introduzir as abstrações em uma linguagem baseada em objetos, devido principalmente à simplicidade e flexibilidade oferecidas por este tipo de linguagem [1]. A linguagem utilizada é baseada em Obliq [4], porém sem o conceito de escopo léxico distribuído e sem um tratamento transparente de referências de rede. Esta transparência, típica de linguagens para redes locais, torna Obliq inadequada a computação na Internet [5].

Aplicações em Obliq são estruturadas como um conjunto de objetos. Sendo uma linguagem baseada em objetos, não há classes, herança nem chamada dinâmica de métodos. Um objeto com campos $x_1, x_2, ..., x_n$ é criado da seguinte forma:

{ x_1 `=>` a_1`,` x_2 `=>` a_2`, ... ,` x_n `=>` a_n }

onde cada a_i pode ser um atributo ou um método. Um atributo é definido como no seguinte exemplo:

```
x => 3
```

Um método, por sua vez, é definido da seguinte forma:

x => meth $(y, y_1, y_2, ..., y_m)$ b **end**

onde y é o parâmetro *self*, $y_1, y_2, ..., y_m$ são os demais parâmetros do método e b é o seu corpo.

Em Obliq, variáveis não possuem tipo. No entanto, valores carregam seus tipos para tempo de execução, permitindo assim que o *run-time* da linguagem seja fortemente tipado.

3.2 Containers, Contexto e Mobilidade

Um dos objetivos do projeto de uma aplicação distribuída na Internet deve ser a sua divisão em partes móveis e autônomas. Assim, torna-se necessário a definição de uma construção para delimitar estas partes. A esta construção, denominaremos de *container*. Um *container* é um "envoltório" de objetos com o propósito de torná-los móveis, isto é, *containers* podem se mover para *contextos* localizados em outros nodos da rede. Contextos são serviços disponibilizados por estações da rede para execução remota de objetos contidos em *containers*, como mostrado na Figura 1. Um contexto oferece também recursos a esta execução, como, por exemplo, um sistema de arquivos, uma estrutura de dados ou um sistema de janelas. Todo contexto é identificado por uma URL da forma *host/name*, onde *host* é o nome da máquina onde o contexto executa e *name* é o nome do contexto.

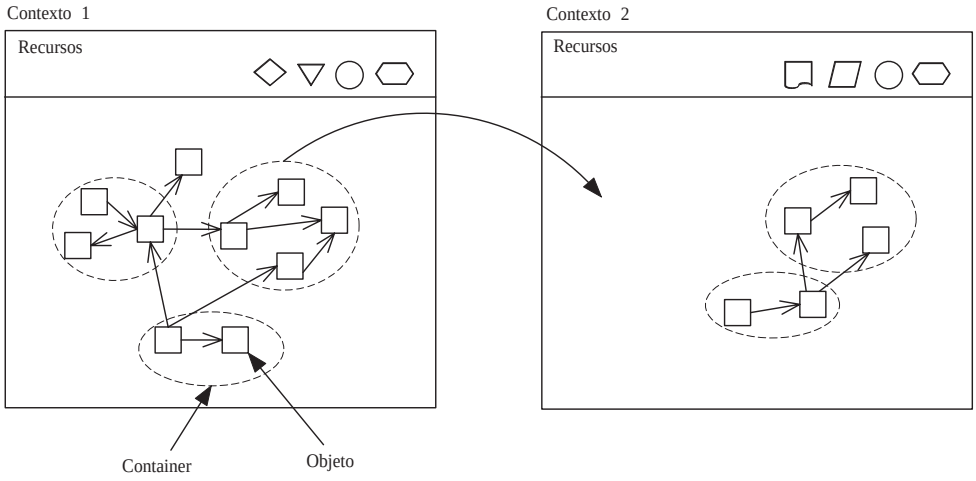

Figura 1. Arquitetura para Computação Móvel na Internet

Sintaticamente, um *container* com objetos $p_1, p_2, \ldots, p_n$ é criado da seguinte forma: *new_container* $(m, p_1, p_2, \ldots, p_n)$, onde m é um parâmetro opcional que designa o nome do *container*. Um objeto não pode pertencer simultaneamente a mais de um *container*.

Podem ser realizadas as seguintes operações envolvendo *containers*:

- insert_object (c, a): insere o objeto a no *container* de nome c

- context_jump (d): mobilidade subjetiva de *containers*
- move (c, d, p): mobilidade objetiva de *containers*
- this_container: retorna o nome do *container* corrente
- is_local (n): verifica se existe um *container* de nome n no contexto local
- new_name: retorna um nome de *container* único na rede
- here: retorna a URL do contexto corrente

A operação *context_jump* faz com que o *container* corrente migre para o contexto cujo nome é passado como parâmetro. Após esta operação, a execução prossegue na linha seguinte, porém já no interior do contexto de destino. No Cálculo de Ambientes, esta forma de mobilidade é denominada subjetiva, já que é o próprio objeto que decide mover o *container* em que se encontra [6]. Já a operação *move* é realizada sobre um *container* que não é o corrente, sendo por isso denominada de mobilidade objetiva. Após esta operação, a execução no contexto de origem prossegue de forma assíncrona na linha seguinte ao *move*. Já no contexto de destino, a execução inicia-se pelo método *start* do objeto p passado como parâmetro e termina quando este método retornar.

Em todo contexto, existe também um *container* pré-definido, de nome *Context*, ao qual pertencem por definição todos objetos que ainda não foram inseridos em nenhum outro *container*.

3.3 Manipulação de Objetos

Objetos são manipulados através de uma semântica de nomes. Todo objeto possui um nome implícito associado ao mesmo no momento de sua criação. Este nome identifica unicamente o objeto em qualquer contexto da rede [2]. Uma definição da forma **let** x = { ... } associa à variável x o nome do objeto criado. Quando executa-se uma operação do tipo $x.op$ sobre um objeto, verifica-se se existe no contexto local um objeto com o nome denotado por x. Se existir, executa-se a operação *op* deste objeto. Caso não exista, a operação fica bloqueada até que o mesmo esteja localmente disponível. Assim, objetos localizados no contexto corrente são ditos *disponíveis*, enquanto que objetos remotos são ditos *indisponíveis*.

A manipulação de objetos através de uma semântica de nomes mantém a linguagem fiel aos princípios do Cálculo de Ambientes, visto que não pressupõe *action-at-a-distance*, isto é, não permite a manipulação transparente de referências para objetos remotos, como ocorre de forma usual em linguagens para redes locais. Além disso, esta semântica de nomes facilita a livre migração

[2] Uma maneira de se implementar um identificador de objeto com significado global em uma rede é utilizar como chaves o endereço da placa de rede da máquina onde o objeto foi criado e a data e hora desta criação.

de objetos contidos em *containers*, pois não cria uma "ligação estática" entre estes objetos e seu contexto de execução.

Existe ainda a função *is_available* (r), a qual retorna *true* se o objeto denotado por r encontra-se disponível no contexto local e *false*, caso contrário.

Exemplo: Sistema para Avaliação de Artigos

Mostra-se a seguir, como exemplo de uma aplicação Internet, um sistema para avaliação de artigos, a ser utilizado pelo comitê de programa de uma determinada conferência. Este sistema é citado em [5] como exemplo de uma aplicação indicada para ser implementada em uma WAL. Uma facilidade desejada em um sistema como este é a possibilidade de operação desconectada da rede. Com isso, permite-se que um membro do comitê realize, por exemplo, avaliações de artigos em seu *notebook* ou em um microcomputador que não disponha de uma conexão dedicada à Internet.

Em um determinado estágio da execução deste sistema, pode-se, por exemplo, desejar enviar um artigo para avaliação por um dado revisor. Como pretende-se satisfazer ao requisito de operação desconectada, a solução é criar um *container* com os objetos necessários à avaliação e enviá-lo para o contexto do revisor, a fim de que a avaliação seja realizada então localmente. Uma possível implementação é mostrada abaixo:

```
let cont= new_container (artigo, aval, revisor);
move (cont, "diamante.dcc.ufmg.br/revisor", aval);
```

Neste código, *artigo*, *aval* e *revisor* são objetos já criados no programa e que denotam, respectivamente, o artigo a ser avaliado, o formulário de avaliação e o revisor. Após a criação e inicialização do *container* com seus objetos, ocorre então um envio assíncrono do mesmo para execução no contexto do revisor. Neste contexto, conforme definido na operação *move*, a execução terá início pelo método *start* de *aval*, o qual pode, por exemplo, exibir na estação o formulário de avaliação do artigo. Em seguida, o revisor pode decidir repassar o artigo para um outro avaliador. Para tanto, basta que seja executada internamente uma operação como a seguinte:

```
context_jump (n);
```

onde n é a URL associada ao contexto do novo revisor. Este novo revisor avalia então o artigo e, em seguida, executa um novo *context_jump*, fazendo com que o *container* retorne ao contexto do primeiro revisor. Este, após conferir a avaliação realizada, decide remetê-la de volta ao ambiente da conferência e notificar o coordenador do comitê de programa de que a mesma encontra-se concluída. Para isso, basta que seja executada a seguinte seqüência de código:

```
context_jump ("topazio.dcc.ufmg.br/sblp");
coord.notificarFimAvaliacao (artigo.num);
```

Observe que *coord* denota o objeto que representa o coordenador do comitê de programa, o qual permaneceu indisponível enquanto o *container* migrou pelos contextos dos revisores. No entanto, após a execução do código acima, o *container* de novo encontra-se em seu contexto de origem e, portanto, o objeto denotado por *coord* volta a estar disponível, tornando possível que uma mensagem como *notificarFimAvaliacao* seja enviada para o coordenador.

3.4 Sincronização de Containers

A fim de coordenar a migração de *containers*, oferece-se a seguinte operação de sincronização:

```
let i= wait (n);
```

A operação *wait* bloqueia a execução até que exista no contexto corrente um *container* de nome n. Quando este *container* existir localmente, i passa a denotar um objeto especial do mesmo, chamado de *objeto de interface*. O objeto de interface de um *container* é definido pela operação *set_interface_object* (c, p), a qual define p como sendo o objeto de interface do *container* c. Uma outra maneira é através do *flag* "i", passado como parâmetro em uma operação de inserção como a seguinte: *insert_object* (c, p, *"i"*).

Exemplo: Instalação de um Plug in

Suponha que no exemplo anterior do comitê de programa, seja necessário em algum ponto do sistema a visualização de um artigo no formato PDF. Caso não exista localmente um *container* para realização desta tarefa, deve-se buscá-lo em algum outro contexto. Ou seja, este *container* tem comportamento similar a de um *plug in* em um *browser Web*. Sua instalação pode ser implementada da seguinte forma:

```
if not is_local ("acrobat")
  then .... (* Transferência do plug in "acrobat" *)
let plugin= wait ("acrobat");
plugin.display (artigo.texto);
```

Neste exemplo, supõe-se que o nome do "*container plug in*" é *acrobat* e que o objeto de interface deste *container* possui um método *display*, que exibe um documento PDF. Este exemplo também demonstra que as construções descritas neste artigo possuem naturalmente a noção de *linkedição dinâmica*,

permitindo que novos subsistemas sejam adicionados em tempo de execução a um sistema maior.

3.5 Comunicação entre Objetos

Comunicação entre objetos de um mesmo *container* ocorre de forma tradicional em orientação por objetos, isto é, através do envio de mensagens síncronas. O mesmo ocorre entre objetos de *containers* distintos, porém localizados no mesmo contexto. Já a comunicação entre objetos de *containers* localizados em contextos diferentes não se dá com a mesma transparência, visto que mensagens enviadas a objetos indisponíveis são tratadas com uma semântica de bloqueio. Conforme proposto no Cálculo de Ambientes, este tipo de comunicação deve ser implementado por meio de "*containers* mensageiros" que migram para o contexto remoto, enviam localmente a mensagem ao objeto receptor e então retornam com o resultado. A fim de evitar que o programador tenha que codificar todo este procedimento, propõe-se então uma abstração para envio de mensagens através da migração de *containers*. Suponha que r denote um objeto indisponível, localizado no contexto t. Uma mensagem *msg* pode ser enviada a este objeto com a seguinte sintaxe:

```
let x= t::r.msg (y);
```

Mensagens como esta acima, qualificadas com o nome do contexto, são enviadas através de um "*container* mensageiro", gerado automaticamente da seguinte forma:

```
1:   let p = { home => here,
2:             local_r => r,
3:             start => meth (s)
4:                         let res = local_r.msg (y);
5:                         insert_object (this_container, res, "i");
6:                         context_jump (home);
7:                       end
8:           };
9:   let mensageiro = new_container (p, y);
10:  move (mensageiro, t, p);
11:  let x= wait (mensageiro);
```

No código acima, cria-se um "container mensageiro" contendo um objeto para enviar a mensagem e o parâmetro da mesma (linha 9). Este "container" é enviado para o contexto do objeto remoto (linha 10), onde então a mensagem *msg* é enviada localmente sem a ocorrência de bloqueio (linha 4). Enviada a mensagem, o seu resultado é então inserido no "container" (linha 5) e o mesmo retorna ao contexto de origem (linha 6), onde então a variável x passa

a denotar o objeto com o resultado (linha 11).

Observe ainda que no código acima y é um identificador livre no objeto denotado por p (linha 4). Em Obliq, devido ao conceito de escopo léxico distribuído, este identificador permaneceria associado a sua localização de origem mesmo quando o *container* do objeto migrasse. No entanto, na proposta deste trabalho tal fato não ocorre, pois objetos são manipulados através de um semântica de nomes e operações sobre objetos indisponíveis permanecem bloqueadas. Assim, para evitar um bloqueio em qualquer operação executada sobre o parâmetro y, o mesmo é inserido no "*container* mensageiro" (linha 9).

Exemplo: Sistema para Avaliação de Artigos (continuação)

No exemplo anterior do sistema para avaliação de artigos submetidos a uma conferência, o revisor responsável pelo julgamento de um determinado artigo pode necessitar de um prazo extra para emitir seu parecer. Para obter este prazo, ele deve solicitar um adiamento ao coordenador do comitê de programa. No entanto, como o objeto que representa este coordenador encontra-se indisponível no contexto do revisor, não é possível enviar uma mensagem imediatamente a este objeto. A solução então é enviar um mensageiro até o contexto da conferência com a mensagem *solicitaAdiamento (n)*, onde n é o número de dias, e aguardar seu retorno. Como descrito, este mensageiro é enviado da seguinte forma:

```
let r= topazio.dcc.ufmg.br/sblp::coord.solicitaAdiamento (3);
```

Veja que a mensagem acima é qualificada com o "endereço" a ser seguido pelo "*container* mensageiro".

3.6 Acesso a Recursos

Um *container* também pode se mover para um novo contexto para permitir que seus objetos acessem localmente os recursos de que necessitam para realização de sua computação, reduzindo assim o tráfego na rede. Primeiramente, propõe-se que a interface de acesso a um recurso seja definida e implementada por um objeto. Assim, para indicar que um objeto p implementa a interface de acesso a um recurso de nome n e exportar este recurso para o contexto corrente usa-se a seguinte operação:

```
export_resource (n, p);
```

Como os recursos de um contexto devem estar sempre disponíveis, *containers* que exportam algum recurso não são dotados de mobilidade.

Já em um *container* cliente, define-se uma referência r para um recurso de nome n da seguinte forma:

```
resource r= n;
```

Por último, propõe-se que recursos sejam acessados via *binding* dinâmico, isto é, um *container* ao se mover para um contexto tem suas definições de recurso automaticamente estabelecidas com os recursos de mesmo nome exportados por este novo contexto. Caso não exista no contexto de destino um recurso com o nome requerido, considera-se que todas referências para este recurso terão valor *nil*.

Exemplo: Agente Móvel para Pesquisa de Preços

Mostra-se abaixo a implementação de um *container* que representa um agente móvel que pesquisa o preço de um livro em um conjunto de livrarias da Internet, cada uma delas representada por um contexto. Em cada livraria, o agente (*container*) obtém o preço do livro desejado acessando um recurso que representa o catálogo de livros à venda. Após visitar a última livraria, o agente retorna a seu contexto de origem, onde então verifica qual livraria possui o menor preço.

Em uma aplicação como esta, cada livraria virtual deve definir um recurso chamado "catálogo", representando a lista de livros à venda.

```
let p = { ..., pesquisa => meth (s, nome) .... end, ... };
export_resource ("catalogo", p);
```

Já o *container* que realizará a pesquisa é implementado da seguinte maneira:

```
1:  let q = { home => here,
2:            num_livrarias => 3,
3:            livrarias => [ "bookpool", "amazon", "iBS" ],
4:            preco => [nil, nil, nil],
5:            pesquisa_preco => meth (s, nome)
6:                                resource lista = "catalogo";
7:                                for i= 1 to num_livrarias do
8:                                  context_jump (livraria [i]);
9:                                  preco [i]:= lista.pesquisa (nome);
10:                               end;
11:                               context_jump (home);
12:                             end,
13:
14:           menor_preco => meth (s)
15:                               (* retorna menor preço pesquisado *)
16:                         end
```

```
17:         };
18: let agente= new_container (q);
19: q.pesquisa_preco ("the art of computer programming");
20: let x= q.menor_preco ();
```

Neste exemplo, a variável *lista* é dinamicamente associada ao recurso "catalogo" de cada livraria visitada pelo agente móvel (linha 6). Com isso, garante-se que a operação *pesquisa* invocada sobre *lista* (linha 9) sempre realizará a procura do livro na livraria (contexto) corrente do agente.

4 Trabalhos Relacionados

Mobilidade sempre esteve presente em abstrações para desenvolvimento de aplicações distribuídas. Em RPC [3], por exemplo, dispõe-se de mobilidade de controle e de uma forma limitada de mobilidade de dados, geralmente apenas de tipos básicos de dados. Ou seja, trata-se de uma abstração adequada ao desenvolvimento de aplicações no modelo cliente/servidor tradicional, o qual não atende a nenhum dos requisitos de uma aplicação distribuída na Internet mencionados na Seção 1. CORBA [12], por sua vez, tem como objetivo a introdução de mobilidade de controle em linguagens orientadas por objeto, além de também permitir interoperabilidade entre diversos sistemas e prover transparência na localização de objetos. No entanto, sendo uma evolução do modelo RPC, CORBA também não atende aos requisitos de operação de uma aplicação distribuída na Internet.

Existem ainda linguagens que implementam mobilidade de objetos, dentre as quais uma das mais conhecidas é Emerald [7]. No entanto, a exemplo de Obliq [4], estas linguagens foram projetadas para redes locais, dando suporte ao desenvolvimento de aplicações distribuídas baseadas em *action-at-a-distance* e, por isso mesmo, inadequadas a uma rede com as características da Internet.

Sem dúvidas, atualmente Java [2] é a linguagem mais utilizada no desenvolvimento de aplicações Internet. No entanto, em Java, tem-se mobilidade de código, mas não de dados. Com mobilidade de código, garante-se execução na diversidade de arquiteturas que formam a rede e evita-se a instalação manual de aplicações nos clientes. No entanto, como a aplicação ainda depende da rede para obtenção dos dados necessários a sua execução, o modelo não é robusto a falhas de comunicação, nem capaz de operar em modo desconectado. Java também possui mobilidade de controle através da biblioteca Java RMI [10].

Recentemente, o modelo de agentes móveis foi proposto como uma alternativa ao desenvolvimento de sistemas distribuídos. Um agente móvel é um progra-

ma que têm a capacidade de migrar por entre as máquinas de uma rede, carregando consigo o estado de sua execução. A linguagem Telescript [13], em meados da década de 90, foi pioneira no oferecimento de recursos para desenvolvimento de agentes móveis. Posteriormente, com a popularização de Java, surgiram diversos outros sistemas baseados nesta linguagem, como Aglets [8] e Voyager [9]. Apesar de possuírem mobilidade de estado, isto é, a mesma forma de mobilidade proposta neste trabalho, agentes móveis são programas autônomos e monolíticos, projetados para execução de uma tarefa bastante específica. Em geral, estes sistemas dispõem apenas de um mecanismo primitivo para comunicação entre agentes, baseado em troca de mensagens e utilizando *action-at-a-distance*. Em nenhum deles existe uma abstração de mais alto nível para composição de diversos agentes em uma aplicação distribuída de maior porte.

5 Conclusões

Mostrou-se neste trabalho uma proposta visando a introdução de abstrações para computação móvel, inspiradas no Cálculo de Ambientes, em uma linguagem baseada em objetos. A linguagem resultante atende aos seguintes requisitos de uma *Wide Area Language* (WAL):

- Completeza: através dos exemplos mostrados, comprovou-se a usabilidade das construções propostas na implementação de aplicações típicas da Internet, como *plug ins* e agentes móveis, e também de aplicações que requerem operação em modo desconectado, como é o caso do sistema para avaliação de artigos.
- Consistência: o conceito de *containers* e a manipulação de objetos com uma semântica de nomes *não* permitem a construção de aplicações baseadas em *action-at-a-distance* ou que requerem conectividade contínua.

Em [11] mostra-se a semântica formal das construções propostas neste trabalho, usando como formalismo o próprio Cálculo de Ambientes. Antes de se realizar a implementação de uma linguagem que incorpore estas construções, pretende-se estudar ainda a introdução na mesma dos seguintes recursos:

- Um sistema de tipos que permita restringir a migração de *containers* e que possibilite, por exemplo, a definição de objetos imóveis (ou estáticos). Estes objetos não precisariam ser manipulados através de uma semântica de nomes, evitando assim o custo inerente a este tipo de semântica.
- Um sistema para tratamento de exceções relacionadas com comunicação e mobilidade. Este sistema deve, por exemplo, permitir uma certa flexibilização na semântica de bloqueio adotada quando da execução de operações sobre objetos indisponíveis.

- Construções para permitir a reconfiguração dinâmica de *containers*, possibilitando assim que um *container* remoto seja dinamicamente substituído por uma versão mais nova sem que seja necessário reiniciar seu contexto de execução. Esta característica é particularmente interessante em aplicações projetadas para funcionar ininterruptamente.

Referências

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

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

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

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

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

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

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

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

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

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

[11] M. T. O. Valente and R. S. Bigonha. Especificação formal das abstrações para computação móvel de IPL. Technical Report LLP 01/2000, Laboratório de Linguagens de Programação, DCC/UFMG, Jan. 2000.

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

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