Um Algoritmo para Reconfiguração Dinâmica de Agentes Móveis

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

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

Resumo

Neste artigo, propõe-se um algoritmo para reconfiguração dinâmica de aplicações distribuídas implementadas de acordo com o modelo de agentes móveis. Mostra-se também que este algoritmo pode ser facilmente implementado na linguagem IPL, uma linguagem com diversas abstrações para desenvolvimento de aplicações móveis na Internet.

Palavras chave: reconfiguração dinâmica, agentes móveis, linguagens para programação na Internet.

Abstract

In this paper we show an algorithm for the dynamic reconfigutaion of distributed applications based in the model of mobile agents. We also show in the paper that the proposed algorithm can be easily implemented in the IPL language, a language with several abstractions for the construction of mobile applications in the Internet.

Key words: dynamic reconfiguration, mobile agents, Internet programming languages.

1 Introdução

Diversas aplicações distribuídas são projetadas para operar ininterruptamente. Em geral, estas aplicações não podem ser paralisadas nem quando necessita-se realizar uma manutenção corretiva ou evolutiva em algum de seus módulos. Como exemplos podem ser citadas aplicações para controle de processos industriais, para gerência de rede, para controle de centrais telefônicas etc. Portanto, é desejável que tais sistemas possuam a capacidade de serem modificados e estendidos enquanto encontram-se em execução. Esta capacidade é chamada de reconfiguração dinâmica [6].

Recentemente, o modelo de agentes móveis tem sido proposto como um modelo alternativo para desenvolvimento de aplicações distribuídas na Internet [15, 13]. Este modelo propicia a construção de aplicações robustas a flutuações na largura de banda da Internet e que, no limite, podem inclusive operar desconectadas desta rede. Dentre as aplicações que podem se beneficiar do conceito de agentes móveis encontram-se exatamente sistemas distribuídos para monitoramento e notificação [8], isto é, sistemas onde reconfiguração dinâmica é uma característica extremamente desejável.

No entanto, os trabalhos para reconfiguração dinâmica de sistemas distribuídos realizados até o momento sempre foram destinados a sistemas construídos de acordo com o tradicional modelo cliente/servidor. Neste artigo, apresenta-se então um algoritmo visando a reconfiguração dinâmica de aplicações distribuídas baseadas no modelo de agentes móveis. Mostra-se também que este algoritmo pode ser facilmente implementado na linguagem IPL [14, 12], uma linguagem com diversas abstrações para desenvolvimento de aplicações móveis na Internet.

O restante do trabalho está organizado como descrito a seguir. A Seção 2 apresenta de forma resumida a linguagem IPL, utilizada no restante do artigo. Em seguida, na Seção 3, são apresentados os três tipos de reconfiguração dinâmica possíveis de serem realizados em uma aplicação distribuída e mostrados os desafios que devem ser superados na implementação dos mesmos. A Seção 4 apresenta a solução proposta, isto é, o algoritmo de reconfiguração dinâmica proposto neste trabalho. A Seção 5 mostra então como este algoritmo pode ser implementado sem dificuldades em IPL. Na Seção 6, descrevem-se alguns trabalhos relacionados com a proposta deste artigo e, por último, a Seção 7 apresenta as conclusões.

2 A Linguagem IPL

IPL (*Internet Programming Language*) [14, 12] é uma linguagem especificamente projetada para o desenvolvimento de aplicações móveis na Internet, incluindo aplicações distribuídas baseadas no modelo de agentes móveis. As abstrações para mobilidade da linguagem são inspiradas no Cálculo de Ambientes de Cardelli e Gordon [4]. Trata-se de uma linguagem baseada em objetos com a mesma sintaxe de Obliq [2], porém sem o conceito de escopo léxico distribuído e sem uma manipulação transparente de referências de rede. Esta transparência, típica de linguagens distribuídas para redes locais, torna Obliq uma linguagem pré-Web.

Aplicações em IPL são estruturadas como um conjunto de objetos. Sendo uma linguagem baseada em objetos, não existem na linguagem classes, herança ou 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.

A principal abstração de IPL para suportar mobilidade é o conceito 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.

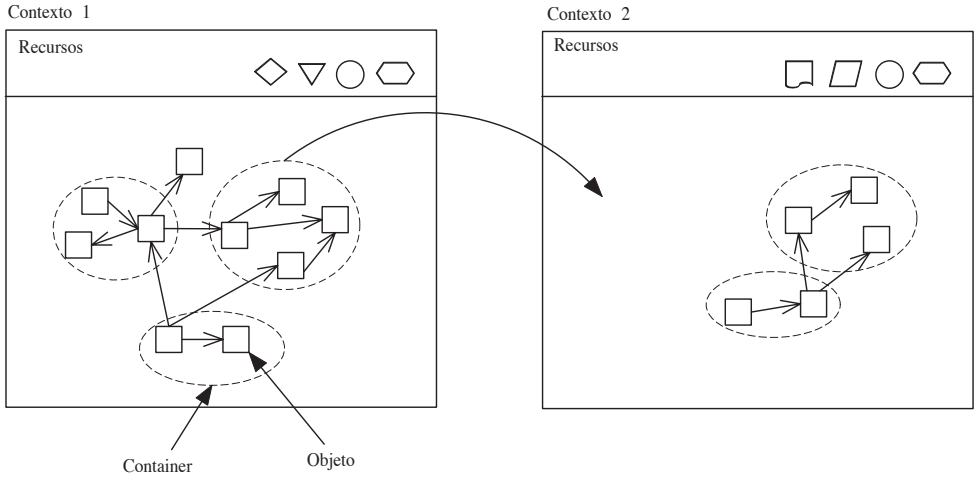

Figura 1: Principais Abstrações para Computação Móvel da Linguagem IPL

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*.

Mobilidade de containers é implementada pelas seguintes operações:

- `context_jump (d)`: move o *container* corrente para o contexto d. A execução prossegue na linha seguinte, porém no interior do novo contexto. Este tipo de mobilidade é chamado de Mobilidade Subjetiva [4].

- `move (c, d, p)`: move o *container* c para o contexto d. A execução no contexto de origem prossegue na linha seguinte. No contexto d, a execução inicia pelo método *start* do objeto p. Este tipo de mobilidade é chamado de Mobilidade Objetiva [4].

Objetos são manipulados em IPL 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. 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* [3], isto é, não permite a manipulação transparente de referências para objetos remotos, como ocorre de forma usual em linguagens para redes locais.

3 Reconfiguração Dinâmica

Reconfiguração dinâmica de sistemas distribuídos pode ser de três tipos [6]:

- Substituição de módulos: quando ocorre a substituição de um módulo da aplicação por outro. Também chamada de reconfiguração de implementação.
- Estrutural: quando ocorre o acréscimo ou remoção de um módulo na aplicação.
- Geométrica: quando a estrutura lógica da aplicação permanece a mesma, porém ocorre uma alteração no mapeamento desta estrutura para a arquitetura distribuída.

Suponha, por exemplo, uma versão distribuída do conhecido problema do Jantar dos Filósofos [7]. Neste caso, uma reconfiguração do tipo substituição de módulos ocorre quando decide-se trocar um dos filósofos por outro que tenha, por exemplo, um melhor apetite. Já uma reconfiguração estrutural ocorre quando adiciona-se ou remove-se um filósofo do sistema. Por último, uma reconfiguração geométrica ocorre quando decide-se mover um filósofo de um nodo da rede para outro. Esta última forma de reconfiguração pode ser necessária por diversos motivos, incluindo tolerância a falhas, balanceamento de carga ou para possibilitar uma melhor utilização dos recursos de comunicação disponíveis no sistema.

Os dois últimos tipos de reconfiguração dinâmica já estão disponíveis em IPL. Reconfiguração estrutural é suportada principalmente pelo recurso de *linkedição* dinâmica da linguagem, o qual permite que novos *containers* sejam dinamicamente incorporados ou removidos de uma aplicação. Já as construções para migração do contexto de execução oferecem uma solução natural para reconfiguração geométrica. Assim, neste trabalho, aborda-se apenas reconfiguração dinâmica do tipo substituição de módulos.

A principal dificuldade para substituição de módulos encontra-se em transferir o estado da execução do *container* antigo para o novo *container*. Esta transferência deve sempre iniciar o novo *container* em um estado consistente, isto é, em um estado a partir do qual o sistema pode continuar seu processamento normalmente [7].

Suponha por exemplo um *container* representando um agente consumidor de algum recurso, como mostrado a seguir:

```
1:  let p= { cont => 0,
2:           start => { resource buf;
```

```
3:                        var s;
4:                        while (true) do
5:                          s:= buf.get();
6:                          cont:= cont + 1;
7:                          "algoritmo para processar s"
8:                        end;
9:                      }
10:         }
11:
12:  let agent= new_container (p);
```

Uma reconfiguração deste agente pode, por exemplo, prever a substituição do objeto p por um novo objeto que implemente um algoritmo diferente para processamento do valor lido s.

Qualquer algoritmo de reconfiguração dinâmica deve então oferecer soluções para as seguintes questões:

- *Como caracterizar o estado corrente da execução de um* container *?* Este estado certamente deve incluir os atributos de todos os objetos do *container*. Mas como tratar atributos existentes no *container* antigo e removidos do novo *container* ? Ou então atributos que tiveram seu tipo alterado no novo *container* ? Em relação a esta pergunta, uma decisão consensual é que o estado não deve incluir o valor do contador de programa, pois o mesmo pode corresponder no código reconfigurado a uma instrução com objetivo totalmente distinto.

- *Quando efetuar a transferência de estado ?* A fim de evitar inconsistências, uma solução simples é aguardar que o *container* esteja inativo, isto é, não possua nenhum método em execução. Porém, como mostrado no exemplo do agente consumidor acima, é bastante comum a existência em sistemas para notificação e monitoramento de laços infinitos, os quais atrasariam a reconfiguração indefinidamente. Por outro lado, uma reconfiguração não pode acontecer a qualquer momento, pois isto certamente acarretaria inconsistências. Supondo que não haja transferência do contador de programa, uma inconsistência ocorre no exemplo acima caso a substituição do *container* aconteça após a leitura do *buffer*, mas antes do processamento do valor lido. Neste caso, haveria perda de um valor.

Já foi concluído em trabalhos anteriores visando a reconfiguração dinâmica de sistemas distribuídos que as questões levantadas acima somente podem ser adequadamente resolvidas caso os módulos envolvidos participem do processo de reconfiguração [6]. O módulo antigo deve, por exemplo, indicar o momento em que o estado corrente de sua execução permite uma reconfiguração. Chama-se tal estado de *estado reconfigurável*. Já o novo módulo deve participar da reconfiguração efetuando a transferência das "partes" do estado antigo que são importantes para prosseguimento da execução.

4 A Solução Proposta

Suponha que se deseja realizar a reconfiguração de um *container* de nome n, com objetos $p_1, p_2, \ldots, p_m$, o qual encontra-se em execução no contexto t. Suponha ainda que este *container* será reconfigurado de forma a conter os objetos $q_1, q_2, \ldots, q_m$. Para criar esta nova configuração do container n deve-se usar a seguinte operação:

```
let new_agent= new_container_config (n, q1, q2, ..., qm);
```

Em seguida, esta nova configuração deve ser enviada para o contexto t, por meio de um comando *move*. Observe então que para criar uma nova configuração de um *container* basta conhecer seu nome e sua localização. Portanto, por questões de segurança, o nome de um *container* não deve ser publicamente divulgado.

Conforme afirmado na Seção 3, cabe à configuração antiga do *container* n indicar quais pontos de seu código dão origem a um estado reconfigurável. Estes pontos são chamados de *pontos de reconfiguração* e devem possuir uma chamada à operação *reconfig_event()*. Esta operação gera um evento de reconfiguração que é capturado pelo contexto corrente. Este então verifica se existe localmente uma nova configuração para o *container* n. Em caso positivo, tem início o processo de reconfiguração. Inicialmente, bloqueia-se o envio de qualquer nova mensagem para um objeto da configuração antiga do *container*. Estas mensagens somente serão tratadas ao fim da reconfiguração, isto é, já pelos novos objetos do *container*.

Em seguida, executa-se o método *upgrade* de cada um dos objetos $q_1, q_2, \ldots, q_m$ da nova configuração do *container* n. Estes métodos são executados na ordem de inserção dos objetos no *container* isto é, iniciando por q_1 e finalizando por q_n. Um método *upgrade* pode acessar atributos do objeto que está substituindo por meio do identificador *old*. Assim, estes métodos devem ser usados para transferir o estado da configuração antiga para a nova configuração. Em um método *upgrade*, pode-se também acessar atributos de qualquer outro objeto do *container* antigo, bastando para isso indexar o identificador *old* da seguinte forma: *old* $[i]$, onde i identifica o i-ésimo objeto do *container* antigo, de acordo com a ordem de inserção. Esta forma alternativa de acesso a objetos do *container* antigo é útil, por exemplo, quando a nova configuração elimina um objeto da configuração antiga, mas requer que seu estado seja transferido para um dos novos objetos.

Terminada a execução dos métodos *upgrade*, retoma-se a execução pelo método *start* do novo *container*. Os objetos do *container* antigo passam a ser candidatos a coleta de lixo.

Mostra-se a seguir um exemplo de reconfiguração do agente consumidor da seção anterior:

```
1:  let q= { cont => 0,
2:           start => { resource buf= "buffer";
3:                      var s;
4:                      while (true) do
5:                        s:= buf.get();
6:                        cont:= cont + 1;
7:                        "novo algoritmo para processar s"
```

```
8:                          reconfig_event (this_container);
9:                        end;
10:                     }
11:
12:           upgrade => { cont:= old.cont; }
13:         }
14:
15: let new_agent= new_container_config (agent, q);
16: move (new_agent, t, p);
```

Pode-se ver, neste exemplo, que além do novo algoritmo para processamento do valor *s* (linha 7), o agente acima também é instrumentado com código específico de reconfiguração. Foi incluído um ponto de reconfiguração após o processamento do valor lido, o qual é indicado pela rotina *reconfig_event()* (linha 8). Além disso, foi acrescentado o método *upgrade* para transferir o contador de valores lidos da configuração antiga para a nova configuração (linha 12). Observe ainda que para geração da nova configuração do agente é necessário o nome da configuração corrente do mesmo (linha 15). Assim, este nome deve ser armazenado em algum meio que permita sua recuperação no momento em que for identificada a necessidade de uma reconfiguração.

5 Implementação do Algoritmo de Reconfiguração Dinâmica

O algoritmo de reconfiguração dinâmica proposto neste trabalho tira proveito da semântica de nomes usada em IPL para manipular objetos e também da semântica de bloqueio utilizada para tratar operações sobre objetos não disponíveis. Como todo objeto possui um identificador único na rede, pode-se bloquear o envio de mensagens simplesmente renomeando adequadamente este identificador.

Em IPL, nomes de *containers* e de objetos correspondem a identificadores com 144 *bits*, organizados da seguinte forma:

- *bits* 0 a 127: usados para identificar um *container* ou objeto em qualquer máquina da rede. Este campo é chamado de OID (*Object Identififier*). O seu valor pode ser gerado, por exemplo, usando o algoritmo proposto no padrão OSF DCE para criação de identificadores chamados neste padrão de GUIDs (*Globally Unique Identifiers*) [9].

- *bit 128*: *bit* de indisponibilidade, usado para indicar que mesmo local um objeto não deve processar mensagens. No caso de *containers*, é usado para indicar que o mesmo representa uma nova configuração que ainda não foi ativada.

- *bits* 129 a 143: identificador da configuração corrente de um *container*. Este campo é chamado de CID (*Configuration Identifier*) e não é usado em objetos.

A operação *new_container_config* gera um novo nome de *container* a partir do nome corrente. O novo nome possui o mesmo OID do nome antigo, porém seu CID é incrementado de um. Já o *bit* de indisponibilidade é igual a um, pois inicialmente o *container* encontra-se inativo.

O algoritmo de reconfiguração utiliza as seguintes funções:

- _new_config (n): dado o nome de *container* n, verifica se existe no contexto local uma nova configuração para este *container*. Isto é, se existe algum *container* local com mesmo OID, porém com um CID maior. Se existir, retorna o nome deste *container*; caso contrário, retorna zero.
- _unavailable (n): torna o *container* ou objeto n indisponível, ativando o seu *bit* 128.
- _rename (p, q): renomeia o OID do objeto denotado por p para o mesmo OID do objeto denotado por q.
- _start_object (n): retorna o nome do objeto que possui o método *start* do *container* n.

Dadas estas operações, o algoritmo final é bastante simples, como pode-se verificar a seguir:

```
1: proc OnReconfig (n) {
2:  n'= _new_config (n)
3:  if (n' > 0)
4:     _unavailable (n);
5:     para todo pi in n, _unavailable (pi);
6:     para todo qi in n',  qi.upgrade ();
7:     para todo qi in n',  _rename (qi, pi);
8:     _available (n');
9:     _start_object (n).start ();
10: }
```

O algoritmo inicia verificando se existe localmente uma nova configuração para o *container* de nome n (linha 2). Em caso positivo, seu identificador é maior que zero e, portanto, tem início o processo de reconfiguração (linhas 4-9). Inicialmente, o *container* corrente é tornado indisponível (linha 4), assim como todos os seus objetos (linha 5). Observe que a simples inversão do *bit* de indisponibilidade é suficiente para bloquear todas a mensagens enviadas a objetos do *container* antigo durante a reconfiguração. Em seguida, executam-se os métodos *upgrade* de todos os objetos da nova configuração do *container* (linha 6). Os objetos do novo *container* são então renomeados de forma a possuírem o mesmo OID dos objetos do *container* antigo (linha 7). Com isso, estes novos objetos passarão a receber todas as mensagens destinadas aos antigos objetos. Por último, ativa-se o novo *container* (linha 8) e inicia-se a sua execução pelo método *start* (linha 9).

6 Trabalhos Relacionados

O presente artigo é inspirado na proposta para introdução de primitivas de reconfiguração dinâmica no ambiente de programação distribuída Polylith [6, 10]. Neste ambiente, as aplicações são estruturadas como um conjunto de módulos interligados por meio de um barramento de mensagens. Cada módulo implementa um processo. A proposta apresentada suporta os três tipos possíveis de reconfiguração: de implementação, estrutural ou geométrica. Assim como pressuposto neste trabalho, a reconfiguração de uma aplicação somente pode ocorrer em pontos pré-definidos do código e, portanto, não é transparente ao programador. No entanto, diferentemente da abordagem adotada neste artigo, em Polylith a reconfiguração não ocorre automaticamente pela simples instalação de um novo módulo na mesma máquina onde encontra-se executando o módulo antigo. Além do código do novo módulo, é necessário o desenvolvimento de um *script* descrevendo tarefas inerentes ao processo de reconfiguração. Dentre estas tarefas, pode-se citar o bloqueio da comunicação entre módulos durante a reconfiguração; a cópia de mensagens destinadas ao módulo antigo e que, durante a reconfiguração, estavam em trânsito para o módulo novo; e a transferência de estado entre os módulos envolvidos.

Como a proposta deste artigo inclui a utilização de uma linguagem de programação especificamente projetada para suportar reconfiguração dinâmica, dispensa-se o desenvolvimento deste *script*. Tal fato ocorre principalmente devido à manipulação de objetos com uma semântica de bloqueio, a qual garante automaticamente a suspensão de qualquer requisição, inclusive as em trânsito, que tem como destino o novo *container*. Já a transferência de estado é especificada nos métodos *upgrade* dos objetos do novo *container*.

Argus, um sistema para construção de aplicações distribuídas na linguagem CLU, provê suporte para reconfiguração dinâmica do tipo substituição de módulos [1]. Permite-se neste sistema a substituição de objetos especiais chamados de *guardians*. A substituição somente ocorre quando estes objetos encontram-se em um estado consistente. No entanto, como o ambiente incorpora um sistema operacional com suporte a transações, pode-se a qualquer momento executar uma operação do tipo *rollback* para o último estado consistente de um *guardian*. Apesar de dispensar a inclusão de pontos de reconfiguração nos programas, esta abordagem tem a desvantagem de requerer um mecanismo de *rollback*, o qual impõe um *overhead* não desprezível no sistema.

Existe também uma proposta para incorporação de reconfiguração dinâmica no sistema Conic [7]. No entanto, a proposta apresentada considera essencialmente reconfiguração estrutural. Mais recentemente, em [5] mostra-se um algoritmo para reconfiguração dinâmica de sistemas distribuídos baseados em um modelo de agentes. No entanto, como os agentes não são móveis, o objetivo principal é permitir reconfiguração estrutural e geométrica. Reconfiguração dinâmica do tipo substituição de módulos, como a tratada no presente artigo, não é considerada.

Em [11] propõe-se a introdução de reconfiguração dinâmica em aplicações desenvolvidas em Eiffel. A proposta apresentada introduz no ambiente de execução desta linguagem um carregador e *linkeditor* dinâmicos, além de um gerenciador de configuração, onde o usuário de uma aplicação pode emitir comandos para substituição de objetos da mesma. Diferentemente da abordagem apresentada neste trabalho, requer-se então a interferência do usuário no momento da reconfiguração.

7 Conclusões

Reconfiguração dinâmica é uma propriedade essencial em diversos tipos de aplicações distribuídas. No entanto, os trabalhos já realizados nesta área foram todos destinados a aplicações construídas de acordo com o tradicional modelo cliente/servidor. Assim, mostrou-se neste artigo um algoritmo para reconfiguração dinâmica de aplicações distribuídas baseadas no modelo de agentes móveis. Além disso, mostrou-se que reconfiguração dinâmica pode ser facilmente incorporada a uma linguagem cujas abstrações para mobilidade são inspiradas no Cálculo de Ambientes, como é o caso de IPL. Particularmente, a semântica de bloqueio para envio de mensagens e o tratamento de objetos por meio de uma semântica de nomes são duas das características de IPL que tornam mais simples a implementação de um algoritmo de reconfiguração dinâmica como o proposto.

Além de ser destinado a aplicações baseadas em agentes móveis, o algoritmo proposto apresenta como vantagem em relação a outras propostas para reconfiguração dinâmica o fato de não requerer nenhuma intervenção do operador da aplicação durante o processo de reconfiguração, seja para implementar um *script* de reconfiguração, seja para iniciar manualmente este mesmo processo. No entanto, assim como nos algoritmos tradicionais, o algoritmo proposto requer que os estados de reconfiguração sejam explicitados no código da aplicação.

Pretende-se prosseguir o presente trabalho implementando uma versão da linguagem IPL que suporte o algoritmo proposto e que também possua as primitivas necessárias para instrumentar o código das aplicações com os pontos de reconfiguração. Com isso, espera-se obter um ambiente que possa ser efetivamente usado na implementação de agentes móveis que requeiram reconfiguração dinâmica.

Referências

[1] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *IEEE Software Engineering Journal*, 8(2):102–108, Mar. 1993.

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

[3] 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.

[4] 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.

[5] N. de Palma. Dynamic reconfiguration of agent-based applications. Technical Report Project SIRAC, INRIA, 1999.

[6] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, 1993.

[7] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

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

[9] Open Group. DCE 1.1: Remote Procedure Call. Technical Report C706, Open Group, Aug. 1997.

[10] J. Purtilo. The Polylith software bus. *ACM Transactions of Programming Languages and Systems*, 16(1):151–174, Jan. 1994.

[11] M. Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, Jan. 1991.

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

[13] M. T. Valente, R. Bigonha, A. A. Loureiro, and M. Bigonha. Linguagens para computação móvel na Internet (tutorial). In *IV Simpósio Brasileiro de Linguagens de Programação*. Sociedade Brasileira de Computação, May 2000.

[14] M. T. Valente, R. Bigonha, A. A. Loureiro, and M. Bigonha. Object oriented languages with abstractions for mobile computation. In *Eletronic Notes on Theoretical Computer Science*, volume 38. Elsevier Science, 2000.

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