Mobilidade de Grupos de Objetos em um Sistema para Programação Distribuída na Internet

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

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

Resumo

Com o crescimento exponencial da Internet, diversas formas de mobilidade têm ganhado importância no projeto de linguagens e bibliotecas para programação distribuída nesta rede. Seguindo esta tendência, este artigo apresenta um sistema para programação distribuída, chamado Jamp, que incorpora abstrações para permitir a migração de grupos de objetos e classes pelos nodos de uma rede como a Internet. Argumenta-se no artigo que o sistema proposto permite o desenvolvimento de aplicações que são menos sujeitas aos problemas de comunicação típicos da Internet e que são capazes de operar mesmo quando desconectadas da rede.

Abstract

Recently, several styles of mobility have started to gain importance in the design of languages and libraries for distributed programming in the Internet. Following this trend, this paper presents a system, called Jamp, with abstractions to support the migration of groups of objects and classes among the nodes of the Internet. It is argued that this system allows the construction of Internet applications more robust to communication failures and that can operate in disconnected mode.

1 Introdução

Na última década, a Internet transformou-se em uma infra-estrutura computacional que cobre todo o planeta. Assim, é natural que se pretenda atualmente construir aplicações distribuídas que explorem o poder computacional oferecido por esta rede. No entanto, a construção deste tipo de aplicação tem revelado-se mais difícil do que se poderia inicialmente prever. O principal motivo é a ausência de abstrações, linguagens e bibliotecas adequadas à implementação das mesmas [4, 15].

Devido aos problemas de comunicação e disponibilidade típicos desta rede, não é possível imaginar a Internet como uma "grande rede local", isto é, como um conjunto de máquinas que oferecem recursos que podem ser transparentemente acessados a qualquer momento. Em função desta constatação, as abstrações normalmente usadas em linguagens distribuídas para redes locais não atendem mais a todos os requisitos que podem ser exigidos de uma aplicação Internet. Especificamente, abstrações como referências de rede, que constituem a base de sistemas como CORBA [10] e Java RMI [11], não são mais adequadas para certos tipos de aplicações, especialmente aquelas que requerem operação em modo desconectado ou que são executadas nos diversos domínios administrativos nos quais a Internet é segmentada [4].

Atualmente, existe ainda uma tendência em permitir que dispositivos computacionais móveis, como assistentes pessoais digitais (PDA) e telefones celulares, disponham de acesso à Internet através de redes sem fio. No entanto, neste tipo de ambiente, os problemas de comunicação e disponibilidade mencionados anteriormente são ainda maiores que nas redes fixas [9].

Sendo assim, existem atualmente diversos trabalhos propondo novos modelos e linguagens para programação distribuída na Internet [12]. Dentre os modelos já propostos, destacam-se aqueles baseados na idéia de computação móvel (em inglês, *mobile computation*) [4, 15]. Basicamente, estes modelos propõem a construção de aplicações capazes de migrar autonomamente pelos nodos de uma rede, carregando consigo pelo menos parte do estado de sua execução. Neste artigo, denominam-se genericamente as aplicações construídas segundo tais modelos de *aplicações móveis*.

A motivação por trás da proposta de aplicações móveis deriva do fato de que este tipo de aplicação pode potencialmente lidar com alguns dos problemas típicos da Internet, conforme descrito a seguir [4]:

- Uma aplicação móvel pode se mover para junto dos recursos que a mesma utiliza, transformando assim interações remotas em interações locais. Com isso, elimina-se o problema de tratamento de desconexões, o qual ocorre quando chamadas remotas são utilizadas em uma rede com a imprevisibilidade da Internet. Além disso, a eliminação de interações remotas pode contribuir também para reduzir o tráfego na rede.

- Em uma rede com o alcance da Internet, é razoável supor que existirão nodos oferecendo recursos com as mais variadas qualidades de serviço. Uma aplicação móvel é capaz então de determinar de forma autônoma o nodo da rede onde cada etapa do seu processamento ocorrerá, dando origem assim a sistemas mais eficientes, capazes de se adaptar a mudanças no ambiente de execução e tolerantes a falhas.

Neste artigo, descreve-se o modelo de programação e a implementação de um sistema orientado por objetos para construção de aplicações móveis na Internet, chamado Jamp. Comparado com outros sistemas já propostos, Jamp apresenta contribuições em relação aos modelos de mobilidade e de comunicação adotados, conforme descrito a seguir:

- Modelo de Mobilidade: diferentemente de outros sistemas, existe em Jamp uma abstração própria para implementação de aplicações móveis, chamada *container*. Um *container* é um componente de *software* composto por um conjunto de classes e objetos e que é capaz de se mover por ambientes de execução disponibilizados ao longo da rede. Estes ambientes são organizados no modelo de programação de Jamp de forma hierárquica, já que em uma rede segmentada e com os problemas de conectividade da Internet não é razoável supor comunicação ponto-a-ponto entre quaisquer nodos da mesma.

- Modelo de Comunicação: ao contrário de outros sistemas, o modelo de comunicação proposto por Jamp não utiliza nenhuma construção que pressuponha conectividade contínua ou que crie "ligações estáticas" que dificultem a migração de *containers*.

O restante do artigo encontra-se organizado como descrito a seguir. A Seção 2 descreve o modelo de programação distribuída adotado em Jamp, enfatizando seus mecanismos de mobilidade e de comunicação. Em seguida, na Seção 3, são descritas as principais classes que compõem o sistema, bem como mostram-se pequenos exemplos de utilização das mesmas. Na Seção 4, descreve-se a implementação do sistema. Por último, a Seção 5 compara o sistema com outros sistemas semelhantes já propostos e a Seção 6 apresenta as conclusões do trabalho.

2 Modelo de Programação

Como Jamp destina-se a construção de aplicações distribuídas na Internet, o modelo de programação adotado no sistema deve ser capaz de lidar com os problemas de comunicação e disponibilidade que são típicos desta rede. Descrevem-se então nesta seção as principais abstrações para mobilidade e comunicação propostas pelo sistema para tratar tais problemas.

2.1 Modelo de Mobilidade

A principal abstração existente em Jamp para construção de aplicações móveis chama-se *container*. Em Jamp, um *container* é um grupo de objetos e classes capazes de se mover em conjunto pelos nodos da rede. O sistema oferece operações para criar *containers* e também para inserir e remover objetos e classes nos mesmos. Diferentemente de outros sistemas móveis, o programador em Jamp pode definir exatamente quais objetos devem pertencer a uma aplicação móvel, bem como se o código das classes destes objetos deve ou não se mover junto com os mesmos.

A opção de mover objetos e suas classes é útil para permitir operação em modo desconectado da rede, já que permite transferir junto com um *container* todo o código e os

dados necessários para sua execução. Já a opção de mover apenas objetos junto com um *container* deve ser utilizada para diminuir o tamanho do mesmo e, portanto, para reduzir o consumo de banda de rede, quando se sabe que a estação de destino possui localmente todo ou parte do código necessário para execução do *container*.

Para que um *container* possa se mover para um outro nodo da rede, este deve oferecer um *contexto* para sua execução, isto é, contextos são serviços disponibilizados por estações da rede para execução de *containers*. Um contexto pode oferecer também recursos a esta execução, como, por exemplo, uma estrutura de dados.

No entanto, em uma rede com as características da Internet, não é razoável supor que seja possível a um *container* migrar diretamente de qualquer contexto A da rede para um outro contexto B. Um dos motivos é que o contexto de destino B pode pertencer a um outro domínio administrativo, sendo o acesso ao mesmo protegido por um *firewall*. Além disso, pode ocorrer também de o contexto de destino estar sendo executado em uma estação que se encontra temporariamente desligada e/ou desconectada da rede, como, por exemplo, um computador móvel [4].

A fim de tratar tais problemas, o modelo de mobilidade adotado em Jamp pressupõe que existe uma hierarquia de dois níveis entre os diversos contextos utilizados por uma aplicação móvel. Assim, existem dois tipos de contexto no modelo proposto: *contexto de sistema* e *contexto de usuário*.

Um contexto de sistema é um programa capaz de receber *containers* provenientes de outros contextos da rede e então proceder de uma das seguintes formas: iniciar uma *thread* para execução do *container* recebido ou então armazenar o *container* em memória secundária. Neste segundo caso, o *container* deve ter sido enviado para um usuário previamente cadastrado no contexto.

Um contexto de sistema deve ser executado em uma estação servidora, isto é, em uma estação com alta disponibilidade e com uma conexão dedicada à Internet. Além disso, deve ser permitido a contextos localizados em outros domínios administrativos acessar especificamente esta estação para envio de *containers*.

Já um contexto de usuário encontra-se sempre associado a um contexto de sistema e a um usuário específico da aplicação. Periodicamente, um contexto de usuário conecta-se a seu contexto de sistema e transfere do mesmo todos os *containers* que estão ali armazenados em nome de seu usuário. A execução propriamente dita de um *container* em um contexto de usuário somente ocorre por solicitação do usuário do mesmo, isto é, *containers* não são automaticamente executados após serem transferidos para um contexto de usuário. Neste modelo, um contexto de usuário não necessariamente precisa estar em execução a todo momento, nem dispor de uma conexão ininterrupta à Internet. A Figura 1 ilustra uma possível interface de um contexto de usuário e a execução de um dos *containers* armazenados no mesmo.

O modelo de mobilidade de *containers* adotado em Jamp é similar ao usado em sistemas de correio eletrônico na Internet. Um *container*, por exemplo, pode ser comparado a uma mensagem eletrônica, com a vantagem de que o mesmo pode conter dados e código e não apenas um texto estático. Já um contexto de sistema é semelhante a um servidor de mensagens e um contexto de usuário, a um programa leitor destas mensagens. Uma segunda comparação pode ser realizada com o modelo de mobilidade de *applets* de Java [1]. Em certo sentido, um contexto de sistema desempenha papel semelhante ao de um servidor

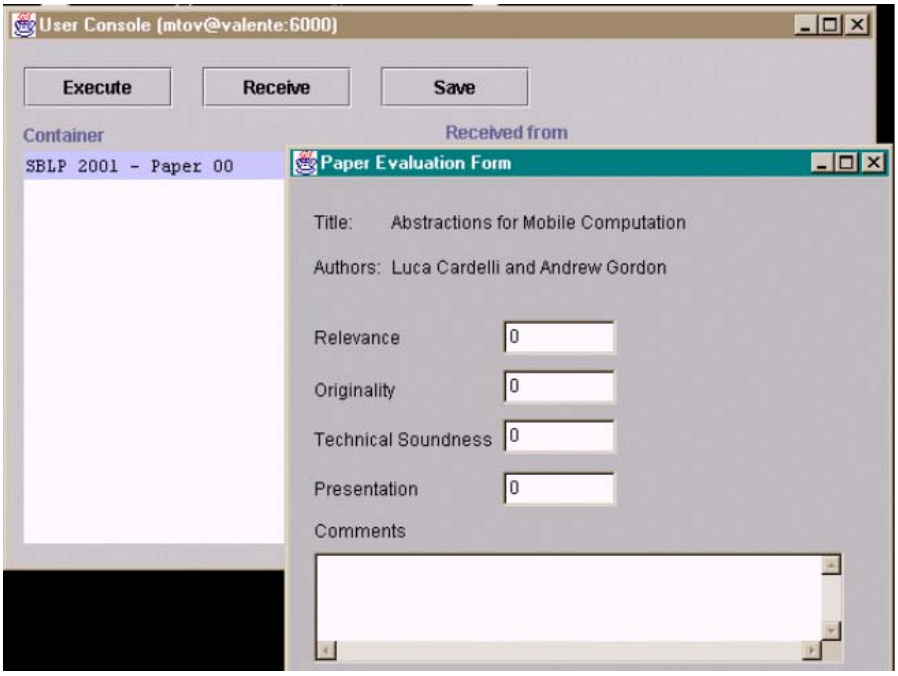

Figura 1: Interface de um contexto de usuário

Web de onde uma *applet* é transferida. Um contexto de usuário, nesta comparação, seria semelhante a um *browser Web*. Por fim, um *container* seria semelhante a uma *applet*. No entanto, *containers* são dotados de mobilidade não só de código, como uma *applet*, mas também de estado, além de serem capazes de migrar de forma autônoma de um contexto para outro.

Este modelo de mobilidade adapta-se também à construção de aplicações distribuídas no estilo *push*, onde o fluxo de informação ocorre do servidor para o cliente, mesmo que não tenha existido uma requisição explícita por parte deste último. Este tipo de aplicação é particularmente adequado para ambientes de redes sem fio, pois permite transferir para um computador móvel, quando este se conecta a uma nova rede, aplicações que podem ser úteis a seu usuário neste novo ambiente.

2.2 Modelo de Comunicação

Em Jamp, comunicação entre objetos ocorre por meio de chamadas de métodos, conforme usual em linguagens orientadas por objeto. No entanto, em um sistema de objetos móveis, pode ocorrer de se possuir uma referência para um objeto e o mesmo não se encontrar presente no ambiente de execução corrente. Por isso, no modelo de comunicação adotado em Jamp, supõe-se que uma referência para um objeto móvel pode se encontrar em dois estados: conectado ou desconectado. O significado de cada um destes estados é descrito a seguir:

- Uma referência é dita *conectada* quando referencia um objeto de um *container* presente no contexto local.

- Uma referência é dita *desconectada* quando referencia um objeto de um *container* que não se encontra presente no contexto local.

Uma referência conectada pode ser utilizada para se chamar um método do objeto referenciado. Já uma tentativa de se chamar um método utilizando-se uma referência desconectada produz uma exceção. Ressalte-se que uma migração pode alterar o estado de uma referência, mas nunca o objeto referenciado, isto é, a identificação do objeto referenciado é preservada mesmo quando ocorre uma migração do objeto de posse da referência.

Além de objetos, um contexto também provê recursos para execução de *containers*. Um recurso é um objeto estático, isto é, não móvel, e que, portanto, não pode ser inserido em nenhum *container*. Além disso, recursos possuem um nome, definido no momento da criação dos mesmos. De forma semelhante a referências para objetos móveis, uma referência para um recurso pode se encontrar conectada ou desconectada. Uma referência para um recurso de nome n encontra-se conectada, quando existe um recurso de mesmo nome no contexto de execução corrente; caso contrário, a referência é dita desconectada.

A Figura 2 ilustra o funcionamento de referências conectadas e desconectadas. Na situação 1 desta figura, mostra-se um *container* em execução no contexto P com três referências a, b e c, todas elas conectadas. Nesta figura, a e b são referências para objetos (representados como quadrados) e c é uma referência para um recurso (representado como um losango). Já na situação 2 da mesma figura, mostra-se a configuração do sistema após a migração do *container* para o contexto Q. Neste momento, as referências a e c encontram-se conectadas e a referência b encontra-se agora desconectada. Finalmente, caso o *container* retorne a seu contexto de origem, a situação 1 é restabelecida, estando novamente as três referências conectadas.

O modelo de comunicação adotado em Jamp não pressupõe conectividade contínua, pois referências para objetos remotos são mantidas desconectadas. Esta é uma diferença essencial entre o modelo de comunicação proposto e o tradicionalmente usado em sistemas de objetos distribuídos, onde dispõe-se de transparência de acesso a objetos remotos. Apesar de ser uma característica desejável em um sistema distribuído, a implementação deste tipo de transparência somente é viável em um ambiente estável e sem problemas de desconexões, como é o caso de redes locais [4].

Além de não pressupor conectividade contínua, o modelo de comunicação proposto também não cria restrições para a migração de *containers*. Um *container* pode possuir referências para objetos externos ao mesmo e também para recursos do seu contexto de execução. No entanto, em ambos os casos, quando um *container* migra pode ocorrer apenas uma alteração no estado destas referências. Ou seja, não existe no modelo proposto "ligações estáticas" entre um *container* e o seu contexto de execução, as quais poderiam vir a restringir a migração do mesmo.

3 Programando em Jamp

Jamp é um sistema que implementa em Java o modelo de computação distribuída descrito na Seção 2. O sistema encontra-se estruturado como um pacote com as seguintes classes:

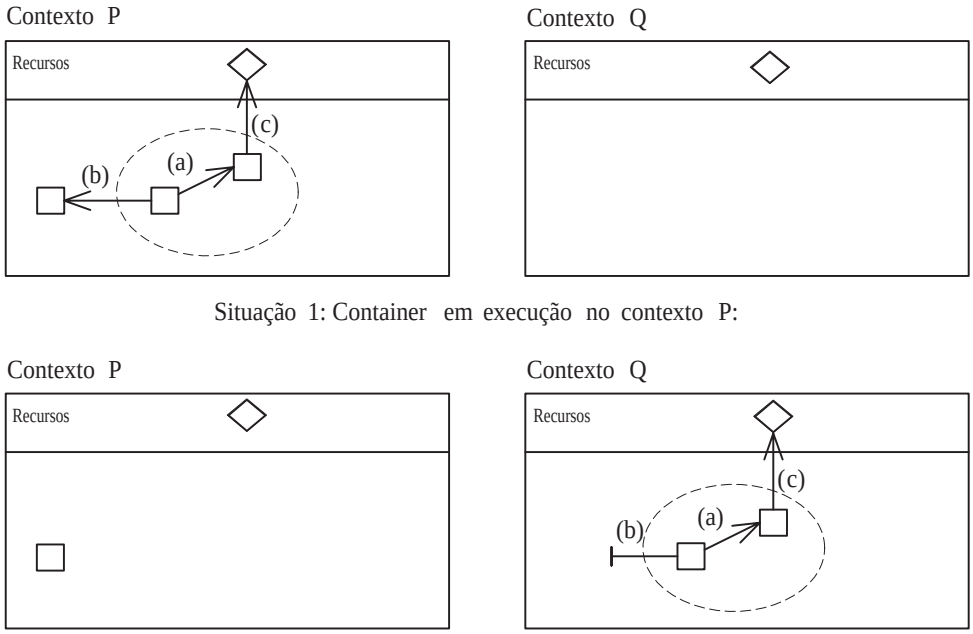

Figura 2: Exemplo de referências conectadas e desconectadas

`JContainer`, `JResource`, `JSystemContext` e `JUserContext`, as quais são descritas nas subseções seguintes.

3.1 Containers

Containers são objetos da classe `JContainer`, a qual possui os métodos públicos descritos na Tabela 1. Para simplificar a descrição das classes do sistema, as exceções ativadas pelos métodos das mesmas não são relacionadas nas tabelas apresentadas nesta seção.

Em `Jamp`, existem duas diferenças básicas entre um objeto móvel, isto é, um objeto que pode ser inserido em um *container*, e um objeto comum de Java:

- Objetos móveis são criados usando-se o método `newObject` da classe `JContainer` e não pelo operador `new` de Java.
- Toda classe de um objeto móvel deve implementar pelo menos uma interface. Uma destas interfaces, e não a classe, é que deve ser utilizada para declarar referências para este objeto. Conforme descrito na Seção 2, este tipo de referência pode se encontrar em dois estados: conectado ou desconectado.

O trecho de código a seguir cria inicialmente dois objetos móveis das classes `A_Impl` e `B_Impl` e um *container*. Em seguida, os dois objetos são inseridos neste *container*, juntamente com suas respectivas classes e com uma terceira classe `C_Impl`, a qual pode ser necessária, por exemplo, para criar remotamente um objeto. Por fim, envia-se o *container* para um outro contexto, usando para isso o método `move`:

Classe JContainer
`JContainer(String description)` Cria um *container*, com um texto explicativo sobre o mesmo
`void addObj(Object obj)` Insere o objeto especificado no *container*
`void addClass(String className)` Insere o código da classe especificada no *container*
`void removeObj(Object obj)` Remove o objeto especificado do *container*
`void removeClass(String className)` Remove o código da classe especificada do *container*
`void move(String contextName, JStartObject startObj)` Move o *container* para o contexto especificado e informa que sua execução deve se iniciar pelo método `run` do objeto especificado
`static Object newObject(String className)` Cria um objeto móvel da classe especificada

Tabela 1: Métodos públicos da classe `JContainer`

```
A a = (A) JContainer.newObject("A_Impl");  // A_Impl implementa interface A
B b = (B) JContainer.newObject("B_Impl");  // B_Impl implementa interface B
JContainer container = new JContainer ("example");
container.addObj(a);
container.addObj(b);
container.addClass("A_Impl");
container.addClass("B_Impl");
container.addClass("C_Impl");
container.move("john@server.foo.br:4000", a);
```

O método `move` espera que o contexto de sistema de destino seja especificado no seguinte formato: `user@host:port`, onde `user` é nome do usuário para o qual o *container* está sendo enviado, `host` é o nome da estação do contexto de destino e `port` é o número da porta TCP/IP associada a este contexto. Caso não se queira que o *container* seja armazenado no contexto de destino para posterior envio para um contexto de usuário, mas sim que seja executado automaticamente tão logo chegue ao mesmo, deve-se informar um nome de contexto no seguinte formato: `host:port`.

3.2 Recursos

Um recurso é um objeto estático, isto é, não móvel, que provê serviços para os *containers* em execução em um determinado contexto. Recursos são identificados por um nome, o qual deve ser de conhecimento prévio de seus clientes. Em Jamp, recursos são objetos de classes que implementam a interface `JResource`, descrita na Tabela 2.

Um objeto acessa recursos de seu contexto de execução por meio de atributos que são referências para estes recursos. Além disto, é responsabilidade deste objeto inicializar tais atributos com os recursos que eles denotam. Para isto, deve ser criado um

Interface JResource
`String getName()` Retorna o nome do recurso

Tabela 2: Métodos da interface `JResource`

objeto representando esta associação por meio do método estático `newBinding` da classe `JResourceBinding`, conforme descrito na Tabela 3.

Classe JResourceBindings
`static Object newBinding (String interfaceName, String resourceName)` Retorna um objeto que armazena uma associação para o recurso que possui a interface e o nome especificados

Tabela 3: Métodos públicos da classe `JResourceBindings`

No exemplo a seguir, mostra-se uma classe com dois atributos `buffer` e `calendar` que são referências para recursos:

```
class A_Impl implements A {
   BufferResource buffer;          // BufferResource implements JResource
   CalendarResource calendar;      // CalendarResource implements JResource

   void init() {
     buffer= JResourceBinding.newBinding("BufferResource", "buffer01");
     calendar= JResourceBinding.newBinding("CalendarResource", "cal2001");
     .....
   }
  .....
}
```

No método `init`, os atributos da classe `A_Impl` que representam recursos são inicializados com um objeto que armazena a associação entre tais atributos e o recurso denotado por eles.

3.3 Contextos

Em Jamp, contextos são criados usando-se as classes `JSystemContext` e `JUserContext`, conforme os mesmos sejam contextos de sistema ou de usuário. Os métodos públicos destas duas classes são descritos nas Tabelas 4 e 5, respectivamente.

Mostra-se a seguir um exemplo de um programa Java que cria um contexto de sistema com dois recursos e, em seguida, inicia a execução do mesmo.

Classe JSystemContext
`JSystemContext(int port)` Cria um contexto de sistema associado à porta TCP/IP especificada
`void addResource(JResource resource)` Adiciona o recurso especificado ao contexto
`addUser(String name, String password)` Adiciona o usuário com nome e senha especificados ao contexto
`void start()` Inicia a execução do contexto

Tabela 4: Métodos públicos da classe `JSystemContext`

Classe JUserContext
`JUserContext(int port,String user,String password,String context)` Cria um contexto de usuário associado à porta TCP/IP, ao usuário e ao contexto de sistema especificados
`void addResource (JResource resource)` Adiciona o recurso especificado ao contexto
`void start()` Inicia a execução do contexto

Tabela 5: Métodos públicos da classe `JUserContext`

```
public class ContextLauncher  {

  public static void main(String[] args) {
     JSystemContext context= new JSystemContext(Integer.parseInt(args[0]));
     BufferResource buffer= new BufferResourceImpl("buffer01", ....);
     CalendarResource calendar= new CalendarResourceImpl("cal2001", ....);
     context.addResource(buffer);
     context.addResource(calendar);
     context.addUser("john", "x2yz");
     context.start();
  }
}
```

No exemplo anterior, ao se chamar o método `start`, o programa entra em um *loop* infinito, onde permanece aguardando a migração de *containers*.

4 Implementação

Jamp foi implementado em Java, usando-se a versão 1.3 do JDK e possui aproximadamente 2500 linhas de código. A escolha de Java deve-se aos recursos oferecidos pela linguagem para geração de programas portáveis entre diversas arquiteturas, serialização de objetos,

carregamento dinâmico de código, reflexividade, criação de *threads* e programação distribuída usando-se *sockets*. Todos estes recursos são usados na implementação de Jamp.

O principal conceito utilizado na implementação de Jamp é o de *mediador*. Um mediador é um objeto interno ao sistema, usado para viabilizar a transferência de *containers* por meio de uma semântica de migração e para permitir a implementação dos conceitos de referências conectadas e desconectadas.

Todo objeto móvel em Jamp possui um objeto mediador. Este mediador é criado juntamente com o objeto móvel quando se chama o método `newObject`. Um mediador possui dois atributos: `guid`, o qual é um valor de 128 bits que identifica unicamente o objeto mediado em qualquer nodo da rede e `ref`, que é uma referência para o objeto mediado. O atributo `guid` é gerado concatenando-se o endereço IP da máquina onde o objeto foi criado com um inteiro cujo processo de geração garante que não é possível gerar nesta mesma máquina um outro valor idêntico ao longo do tempo. Já a referência `ref` é a única referência existente na aplicação para o objeto mediado, já que o método `newObject` não retorna uma referência para este objeto e sim para o seu mediador. A Figura 3, mostra um *container* na visão do usuário e o mesmo *container* na visão interna do sistema, com os mediadores de cada um de seus objetos representados como círculos.

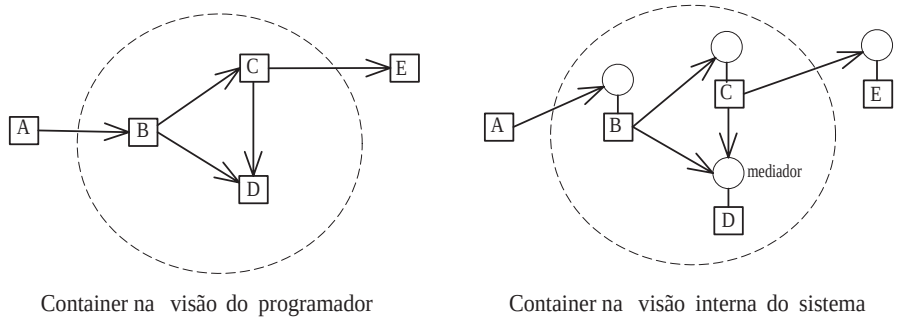

Figura 3: Container na visão do usuário e na visão interna do sistema

Suponha que se deseje enviar um *container* $c = \{(m_1, o_1), (m_2, o_2), \ldots, (m_n, o_n)\}$ para um determinado contexto, onde o_i representa cada um dos n objetos deste *container* e m_i seus respectivos mediadores. Inicialmente, gera-se uma representação serializada do *container* c, utilizando para isso uma versão modificada das rotinas de serialização de objetos de Java. A implementação padrão destas rotinas serializa um objeto e todos os objetos acessíveis a partir do mesmo. Já a rotina de serialização usada na implementação do sistema restringe a serialização a objetos pertencentes ao *container* que está sendo migrado.

Em seguida, a representação serializada do *container* é transmitida para o contexto de destino. Finalizada a transmissão, atribui-se `null` ao atributo `ref` de cada um dos mediadores $m_1, m_2, \ldots, m_n$ no contexto de origem. Como esta é a única referência mantida no sistema para os objetos $o_1, o_2, ..., o_n$, os mesmos se tornam inacessíveis após esta atribuição e, eventualmente, serão destruídos pelo coletor de lixo. Deste modo, consegue-se transferir um *container* utilizando uma semântica de migração de seus objetos e não de cópia, como ocorre, por exemplo, em sistemas baseados em Java RMI [11].

Como `newObject` retorna uma referência para o mediador de um objeto móvel, quando se chama um método deste objeto está sendo chamado, na verdade, um método do seu mediador. Por este motivo, um mediador deve implementar todos os métodos das interfaces de seu objeto mediado. A implementação destes métodos no mediador simplesmente verifica se o atributo `ref` é diferente de `null`, isto é, se objeto mediado encontra-se presente no contexto local. Caso esteja, a chamada é repassada para o objeto mediado. Caso contrário, ativa-se uma exceção do tipo *UnavailableObject*. Assim, nesta implementação do sistema, uma referência conectada é, na verdade, uma referência para um mediador com um objeto mediado diferente de *null*. Já uma referência desconectada é uma referência para um mediador com um objeto mediado igual a *null*.

Em Jamp, mediadores são criados usando-se o conceito de classes *proxy* dinâmicas (*dynamic proxy class*) do pacote de reflexividade da versão 1.3 de Java [1]. Uma classe *proxy* dinâmica implementa uma lista de interfaces especificadas em tempo de execução, tal como requerido pelo conceito de mediadores.

Suponha agora que um determinado contexto tenha recebido para execução o *container* $c = \{(m_1, o_1), (m_2, o_2), \ldots, (m_n, o_n)\}$ mencionado anteriormente. Inicialmente, este contexto "desserializa" a representação do *container* recebida do contexto de origem. Neste processo, é utilizado um carregador de classes diferente do normalmente usado em Java [7]. Em Jamp, o carregador de classes usado para obter as classes de um *container* é uma instância da classe `JampClassLoader`. A necessidade de um carregador de classes específico para o sistema deve-se ao fato de que o código das classes dos objetos de um *container* pode não se encontrar armazenado no disco da estação local, como ocorre normalmente em uma aplicação Java. Conforme afirmado na Seção 2, este código pode ter sido incorporado ao *container* e, portanto, se este for o caso, ele deve ser recuperado da representação serializada do mesmo. Daí a necessidade de um carregador de classes especial capaz de realizar tal tarefa.

5 Trabalhos Relacionados

Recentemente, com o crescimento exponencial da Internet, mobilidade tem ganhado importância no projeto de linguagens de programação. Java [1], por exemplo, disseminou a noção de mobilidade de código na Internet. Com mobilidade de código, garante-se execução na diversidade de arquiteturas existentes na rede. No entanto, como as aplicações ainda dependem da rede para obtenção dos dados necessários a sua execução, o modelo de mobilidade de código, quando utilizado isoladamente, não é suficientemente robusto a falhas de comunicação, nem capaz de operar em modo desconectado [4, 8].

Mais recentemente, o modelo de agentes móveis foi proposto como uma alternativa para programação distribuída na Internet. Um agente móvel é um programa que têm a capacidade de migrar por entre as máquinas de uma rede, carregando consigo o estado de sua execução [16, 12]. Dentre os diversos sistemas de agentes móveis já propostos, pode-se citar Telescript [16], Aglets [6] e JavaSeal [14]. No entanto, conforme descrito a seguir, em nenhum destes sistemas existem abstrações capazes de dar suporte ao modelo de programação proposto na Seção 2 deste artigo.

Em Telescript [16], um agente móvel é uma *thread*. Apesar de ser um conceito bastante poderoso, mobilidade de *threads* não é de fácil implementação nas atuais máquinas

virtuais. É impossível, por exemplo, compilar um programa em Telescript para *bytecode*, já que a implementação atual da JVM não permite a captura do estado completo de execução de uma *thread*. Por este motivo, optou-se em Jamp por migrar apenas o estado dos objetos de um *container* e, possivelmente, as classes dos mesmos.

Já em Aglets [6], um agente móvel é um objeto de uma subclasse da classe `Aglets`, isto é, não existe em Aglets nenhuma abstração específica para delimitar precisamente os objetos que serão transferidos em uma migração. O sistema utiliza a biblioteca de serialização padrão de Java, a qual transfere juntamente com um agente todos os objetos acessíveis a partir do mesmo. Além disso, no momento da migração, são transferidos apenas os atributos destes objetos. Seus métodos são transferidos posteriormente por demanda, à medida que se utiliza cada uma deles no nodo de destino. Esta última característica torna o sistema menos robusto a desconexões, pois requer comunicação com o nodo de origem mesmo após uma migração do agente.

Em JavaSeal [14], propõe-se uma abstração chamada *seal* para implementação de agentes móveis. De forma semelhante a um *container*, um *seal* possui um conjunto de objetos e classes. Além disso, *seals* são organizados hierarquicamente, isto é, um *seal* pode conter também *sub-seals*. Comunicação entre *seals* ocorre apenas por meio de troca de mensagens em canais. No entanto, troca de mensagens é um mecanismo de comunicação de mais baixo nível do que invocação de métodos, isto é, do que o mecanismo usado em Jamp para comunicação entre objetos de *containers* distintos.

Em nenhum dos sistemas de agentes móveis mencionados anteriormente existe uma hierarquia de contextos de execução, tal como proposto em Jamp. Em todos eles, assume-se que é sempre possível para um agente móvel migrar diretamente de um ambiente de execução A para outro ambiente B. Ou seja, estes sistemas não consideram a segmentação da Internet em diversos domínios administrativos, protegidos por *firewalls*, nem a ausência de conectividade que pode ocorrer no momento da migração de um agente.

Os conceitos de *containers*, contextos e recursos utilizados em Jamp foram inicialmente propostos em [13]. No entanto, nesta proposta, estas abstrações foram introduzidas em Obliq [3], uma linguagem baseada em objetos para construção de aplicações distribuídas em redes locais. Além disso, nesta proposta, mostra-se apenas a semântica destas abstrações, não tendo sido as mesmas incorporadas em um interpretador de Obliq. Neste artigo, complementa-se esta proposta inicial, mostrando como estas abstrações podem ser incorporadas e implementadas em uma linguagem orientada por objetos de uso geral, como Java. Além disso, este artigo introduz os conceitos de contexto de sistema e de usuário e também de referências conectadas e desconectadas, os quais não fazem parte da proposta mencionada anteriormente.

O estilo de mobilidade implementado em Jamp é inspirado em alguns cálculos de processos que surgiram recentemente para especificação da noção de computação móvel na Internet, destacando-se dentre eles o Cálculo de Ambientes [5] e o Cálculo *Seal* [15].

6 Conclusões

Apresentou-se neste artigo um modelo para programação distribuída na Internet, bem como descreveu-se a implementação de um sistema, chamado Jamp, para desenvolvimento de aplicações neste modelo. O modelo e o sistema descritos no artigo permitem a

construção de aplicações distribuídas cuja execução não é restrita a apenas um nodo da Internet.

Para construção destas aplicações foram propostas duas abstrações principais: *containers* e contextos. Um *container* é um grupo de objetos e classes que podem se mover autonomamente por entre contextos de execução disponibilizados nos diversos nodos da Internet. A idéia básica é permitir a migração do código e dos dados necessários para execução de uma tarefa para junto dos usuários da mesma. Com isso, estes usuários poderão utilizar tais aplicações mesmo quando se encontrarem desconectados da rede. Acredita-se que este requisito de operação em modo desconectado terá importância crescente em aplicações distribuídas para a Internet, especialmente em sistemas para dispositivos móveis, como assistentes pessoais e telefones celulares [8].

O modelo de mobilidade de Jamp propõe ainda uma hierarquização dos diversos contextos de execução da rede, de forma a lidar com os problemas de conectividade típicos da mesma e também para viabilizar a execução de uma aplicação móvel nos diversos domínios administrativos erguidos ao longo da rede. Diferentemente de sistemas tradicionais de objetos distribuídos, o modelo de comunicação adotado em Jamp não pressupõe transparência de acesso a objetos remotos, devido aos problemas de comunicação inerentes à rede. Além disso, os conceitos de referências conectadas e desconectadas são utilizados para evitar a criação de "ligações estáticas" entre um objeto e seu contexto de execução.

Jamp já foi utilizado para implementar parte de um sistema para revisão de artigos de uma conferência. Este sistema é citado em [4] como exemplo de uma aplicação distribuída que pode se beneficiar do tipo de abstração disponível em Jamp. Como trabalho futuro, pretende-se transformar um *container* em um *domínio de proteção* [2], isto é, fazer com que toda comunicação entre *containers* seja regulada por uma política de segurança previamente estabelecida em cada contexto da rede. O objetivo é atender aos requisitos de segurança que são colocados quando executa-se uma aplicação proveniente de uma rede aberta como a Internet.

Referências

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison Wesley, 3rd edition, 2000.

[2] C. Bryce and C. Razafimahefa. An approach to safe object sharing. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Oct. 2000.

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

[4] 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.

[5] 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.

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

[7] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. *ACM SIGPLAN Notices*, 33(10):36–44, Oct. 1998.

[8] C. Razafimahefa and C. Bryce. Towards the design of an internet operating system. Technical report, University of Geneva, 2000.

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

[10] J. Siegel. *Corba 3 Fundamentals and Programming*. John Wiley & Sons, 2nd edition, 2000.

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

[12] M. T. Valente, R. Bigonha, A. A. Loureiro, and M. Bigonha. Linguagens para computação móvel na Internet. *Revista de Informática Teórica e Aplicada*, 6(2):7–47, Dec. 1999.

[13] M. T. Valente, R. Bigonha, A. A. Loureiro, and M. Bigonha. Introduzindo abstrações para computação móvel em linguagens orientadas por objeto. In *IV Simpósio Brasileiro de Linguagens de Programação*, pages 15–28, May 2000.

[14] J. Vitek and C. Bryce. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

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

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