ANDRÉ LUIZ GONÇALVES DOS SANTOS

UM SUPORTE PARA ADAPTAÇÃO DINÂMICA DE ARQUITETURAS

Dissertação submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do título de Mestre. Área de concentração: Processamento Distribuído e Paralelo.

Orientador: Prof. Dr. ORLANDO GOMES LOQUES FILHO

NITERÓI
2006

Dedico essa dissertação aos meus pais,

que não mediram esforços para que eu

pudesse alcançar os meus objetivos.

Agradecimentos

Agradeço principalmente a Deus, por me permitir mais uma vez aqui estar, vivenciando as diversas circunstâncias que nos permite evoluir nas vertentes de crescimento moral e intelectual. Ao amado Mestre Jesus, por me amparar ao longo da caminhada, revitalizando as minhas energias e mantendo o equilíbrio necessário para que os obstáculos pudessem ser superados.

De forma muito especial, agradeço aos meus pais, que sempre me apoiaram, incentivaram e, principalmente, acreditaram na minha capacidade. Serei eternamente grato pela educação que me deram e por não terem medido esforços para que eu pudesse alcançar os meus objetivos. Eu sei o quanto vocês trabalharam para que eu pudesse “vencer” na vida. Muito obrigado, papai e mamãe, por tudo que vocês fizeram por mim.

Agradeço com muito carinho aos meus queridos irmãos, que confiaram no meu trabalho e me apoiaram ao longo da minha caminhada. Agradeço também aos meus parentes mais próximos que me incentivaram e me apoiaram de alguma forma nessa investida.

Agradeço de forma muito carinhosa à minha querida esposa Beatriz, por todo apoio, carinho, amor e compreensão. Obrigado por ter me apoiado e me dado força para que eu pudesse alcançar mais essa conquista.

Ao Prof. Orlando Loques pela orientação, paciência, persistência e por acreditar que eu fosse capaz de realizar o trabalho necessário para defender o título de mestre. Obrigado por ter me acompanhado e me incentivado ao longo do mestrado. Sem isso, eu não teria conseguido alcançar o objetivo.

Aos professores do Instituto de Computação da Universidade Federal Fluminense que contribuíram para minha formação.

Aos eternos irmãos da Republica Mata Virgem por torcerem por mim durante esta caminhada.

Aos professores da Universidade Federal de Ouro Preto, que se empenharam em passar a base necessária para a continuação dos estudos e para o mercado de trabalho. Em particular, aos professores Marcone Jamilson Freitas Souza e Carlos Frederico Marcelo da Cunha Cavalcanti que me apoiaram para que eu desse mais esse passo em minha vida.

Aos companheiros de república Leonardo (Leopoldo), Vander (Vampeta), Marcel (Cabelo), Tiago (Jóvem) pelos momentos de alegria e aprendizado vividos. Ao grande irmão Shuranha (Alexsandro), pelo companheirismo, atenção, apoio e pelas ajudas prestadas.

Aos demais amigos do mestrado, Bruno, Rafael Guerra, Robson, Luciana, Glauco, Renatinha, Diego Leal, Idalmis, Paulo Motta, Cris, Vivi, Dani, André (Totó), Jacques, Bertini, Rodrigo Toso e todos aqueles que comigo conviveram e contribuíram para essa conquista. Em particular, aos amigos Jonivan, Alexandro (Shuranha), Leopoldo, Jóvem e Glauco pela paciência, atenção e cooperação nos trabalhos desenvolvidos.

Bem, eu não sei se consegui agradecer a todos que de alguma forma participaram e contribuíram ao longo do mestrado, mas tenho a certeza que sem o apoio e incentivo de todos, eu não teria concluído com vitória. A todos vocês, meu MUITO OBRIGADO.

“Quando me desespero,
lembro-me de que
em toda a história
a verdade e o amor
sempre venceram.

Houve tiranos e assassinos
e por algum tempo
pareciam invencíveis,
mas no final sempre caem.”

Mahatma Ghandi

Índice

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Siglas

ADL	*Architecture Description Language*
AOP	*Aspect-Oritented Programming*
Cbabel	*Building Applications by Evolution with Connectors*
CE	Aplicação de Comércio Eletrônico
CORBA	*Common Object Request Broker Architeture*
CR-RIO	*Contractual Reflective - Reconfigurable Interconnectable Objects*
CSR	Cliente/Servidor Replicado
EJB	*Enterprise JavaBeans*
IIOP	*Internet Inter-ORB Protocol*
J2EE	*Java 2 Enterprise Edition*
J2SE	*Java 2 Standard Edition*
JMX	*Java Management Extensions*
JSP	*Java Server Pages*
QoS	*Quality of Service*
RMI	*Remote Method Invocation*
SAR	*Service Archives*
SL&D	Serviço de Localização e Descoberta de Recursos
SMTP	*Simple Mail Transfer Protocol*
VM	*Virtual Machine*
VoD	Vídeo sob Demanda
XML	*eXtensible Markup Language*

Resumo

Este trabalho enfoca requisitos de adaptação de arquiteturas distribuídas, comuns em sistemas emergentes de computação ubíqua (ou pervasiva) e computação autônoma. O primeiro tipo atua em ambientes onde os dispositivos disponíveis e as necessidades dos usuários variam ao longo do tempo. O segundo tipo visa alcançar diversos objetivos automaticamente, tais como reparar falhas, otimizar a resposta temporal ou obter economia de energia, dentre outros. A disponibilidade de mecanismos que facilitem a adaptação em tempo de operação (dinâmica) dos componentes da arquitetura tem se demonstrado útil no atendimento dos requisitos dessas classes de sistema.

Nesse contexto, esse trabalho apresenta um suporte que viabiliza o processo de adaptação das arquiteturas distribuídas, denominado SDA-A (*Support for Dynamic Architecture-Adaptation*). Para isso, ele utiliza tecnologias e ferramentas de suporte à adaptação disponíveis, tais como o padrão *Java Management Extensions (JMX)* e o *toolkit* de manipulação de *Java bytecode Javassist.* Através do *SDA-A* é possível inserir, atualizar e/ou remover componentes de uma arquitetura, possibilitando ainda redefinir as ligações entre os novos componentes e/ou os já existentes. Experimentos realizados demonstraram que a sobrecarga gerada pelo *SDA-A* e os custos gerados pela inserção, remoção ou troca de componentes em uma arquitetura, são compatíveis com requisitos de flexibilidade e desempenho das aplicações consideradas.

Abstract

This work focus on distributed architectures adaptation requisites, common in emerging ubiquitous (or pervasive) computation systems and autonomous computation. The first kind acts in environments where available devices and user's needs vary in time. The second kind aims to achieve several objectives automatically, such as failure repairing, time response optimization or energy economy, among others. The availability of mechanisms that make easier the operation-time (dynamic) adaptation of components of the architecture has shown useful in attending the requisites of these system classes.

In this context, this work presents a support that makes viable the distributed architectures adaptation process, denominated SDA-A (*Support for Dynamic Architecture-Adaptation*). To do so, it utilizes available adaptation support tools and technologies, such as *Java Management Extensions (JMX)* standard and *Java bytecode Javassist* manipulation toolkit. Through *SDA-A* it is possible to insert, update and remove components of a syten architecture, making possible to redefine links between new or existing components. Experiments performed demonstrated that *the* overhead generated by *SDA-A* and the costs generated by a components insertion, deletion or exchange are compatible with the considered applications flexibility and performance requisites.

Capítulo 1

Introdução

Sistemas de computação distribuída vêm sendo utilizados por muitos anos em ambientes comerciais e industriais para atender às necessidades em termos tecnológicos e de desempenho exigidas por cada ambiente. Além disso, tais sistemas também são utilizados em sistemas autônomos, e.g., aplicações de missão crítica e de alta disponibilidade. No decorrer da existência dessas aplicações podem surgir novas funcionalidades, não previstas durante o projeto, que devem ser contempladas. Embora essas aplicações devam incorporar as novas funcionalidades, elas não podem permanecer indisponíveis por longos períodos de tempo devido a razões econômicas e/ou de segurança. Dessa forma, torna-se necessário que essas aplicações possam ser adaptadas dinamicamente para atender a novas demandas por elas requeridas. No contexto deste texto a adaptação dinâmica é considerada uma troca, inserção e/ou remoção de um componente da aplicação sem interromper totalmente sua execução. Um exemplo é a introdução de um componente que realize um filtro de produtos em promoção, mediante o perfil de usuários em um sistema de comércio eletrônico.

Além desses conjuntos de aplicações, de missão crítica e de alta disponibilidade, existe um outro conjunto de aplicações conhecidas como aplicações ubíquas (pervasivas), que vislumbram um mundo onde usuários, movendo-se em diferentes ambientes, interagem naturalmente com diferentes dispositivos computacionais para executar diversos tipos de tarefas. Essas aplicações podem utilizar recursos (dispositivos, serviços, aplicações, etc) cuja disponibilidade pode variar ao longo de sua execução. Como conseqüência da mobilidade, as aplicações em execução em um ambiente ubíquo apresentam um requisito inerente de adaptação dinâmica para atenderem às mudanças no ambiente onde estão imersas. Essas mudanças podem ser consideradas como a disponibilidade de novos dispositivos ou serviços,

o tipo de tarefa em execução, a indisponibilidade de alguns recursos, e até mesmo a presença ou ausência de pessoas no ambiente. Além disso, as aplicações podem migrar, juntamente com os usuários, por diferentes ambientes, reforçando a necessidade de adaptação para melhor uso dos recursos disponíveis.

Em algumas situações, como no caso das aplicações ubíquas, para atender aos requisitos da aplicação (a disponibilidade de novos recursos), é necessário realizar uma adaptação dinâmica da aplicação, de modo que ela seja capaz de interagir com novos recursos de hardware disponíveis, que poderão surgir ao longo de sua execução. Um exemplo é adicionar um trans-codificador de vídeo em uma aplicação de vídeo sob demanda, durante a transmissão de um fluxo de vídeo, de modo a adaptá-lo às características de um novo *display* em um determinado ambiente.

Na prática, podemos considerar as funcionalidades de uma aplicação, a presença de novos recursos ou serviços que as aplicações devem fazer uso, entre outros, como requisitos funcionais de uma aplicação, e os requisitos não-funcionais como os mecanismos de comunicação, qualidade de serviço, tolerância a falhas, entre outros aspectos que não são relacionados diretamente com a funcionalidade da aplicação em si. Em alguns trabalhos, como em [Sztajnberg 2002, Tarr et al. 1999], a separação dos requisitos funcionais e não-funcionais é considerada uma técnica que facilita a obtenção de *Separação de Interesses*. No contexto do desenvolvimento de software a separação de interesses contribui para uma maior facilidade na reutilização de componentes e organização do código. Pois, o código referente a um requisito funcional não se mistura ao código que implementa um requisito não funcional, tendo assim, dois códigos separados que podem ser utilizados em momentos distintos.

Uma outra abordagem que facilita a obtenção da adaptação dinâmica é o paradigma da Arquitetura de Software [Shaw et al. 1995], o qual considera que um sistema (aplicação) pode ser desenvolvido a partir de sua organização como uma composição de *componentes* e *conectores*. Os componentes de uma aplicação são chamados de *módulos* e carregam em si a funcionalidade da aplicação (contemplando os requisitos funcionais). Os *módulos* podem ser agrupados em dois grupos genéricos: os *servidores*, que são os módulos que executam alguma ação concreta no sistema (serviço), e os *clientes*, que são os *módulos* que requisitam os serviços oferecidos pelos servidores. Os *módulos* de uma aplicação podem ser interligados diretamente entre si, ou através de *conectores* que podem ser usados para encapsular os requisitos não-funcionais da aplicação. Na implementação de uma aplicação, tanto os *módulos*

como os *conectores* são representados por objetos ou classes de uma aplicação orientada a objetos.

As ligações entre *módulos* são determinadas em função de pontos específicos de interação, chamados de *portas*. As *portas* são responsáveis pela ligação entre *módulos* e *conectores*. Elas podem ser de dois tipos: as portas de saída, que são associadas a uma requisição de serviço, ou seja, a invocação de um método que presta um serviço, e as portas de entrada, que são associadas às assinaturas dos métodos disponíveis em um *módulo* servidor, e que podem ser encontradas na definição da interface de tal *módulo*.

Embora o uso da abordagem de *Separação de Interesses* em conjunto com o princípio da *Arquitetura de Software* facilite a obtenção da adaptação dinâmica das aplicações, ainda é necessário que sejam utilizadas técnicas e/ou mecanismos, tais como os descritos em [Carvalho et al. 2002, Loques et al. 2004, Nicoara e Alonso 2005, Almeida et al. 2001, Papadopoulos e Arbab 2001, Batista e Rodriguez 2000, Bidan et al. 1998], para que seja possível que as aplicações possam ser adaptadas dinamicamente sem causar impactos consideráveis nos serviços por elas oferecidas. Os impactos em questão são referentes ao atraso na execução de um serviço solicitado por clientes, a um determinado servidor, devido a uma adaptação da aplicação. É importante lembrar que as adaptações dinâmicas consideradas são trocas, inserções ou remoções de componentes ou até mesmo as modificações nas ligações entre os componentes de uma aplicação. Através dessas adaptações dinâmicas as aplicações podem ter seus comportamentos modificados de forma a atender as novas necessidades impostas por cada uma delas.

Nesse contexto, o objetivo desse trabalho é realizar um estudo das técnicas de adaptação dinâmica de arquiteturas visando utilizá-las em conjunto com tecnologias e ferramentas de suporte a adaptação disponíveis tais como o padrão *Java Management Extensions (JMX)* e o *toolkit* de manipulação de *Java bytecode Javassist*. Esse estudo propiciou o desenvolvimento de um suporte para adaptação dinâmica de arquiteturas distribuídas denominado SDA-A (*Support for Dynamic Architecture-Adaptation*). Através dele é possível inserir, atualizar e/ou remover componentes de uma arquitetura, possibilitando ainda redefinir as ligações entre os novos componentes e/ou os já existentes.

O *SDA-A* visa atender requisitos de flexibilidade e desempenho, típicos em aplicações de computação autônoma e ubíqua, permitindo que os componentes e interligações da arquitetura da aplicação sejam reconfiguradas, de modo a adaptá-la às suas necessidades ou do ambiente no qual estão imersas. No nível da implementação, a capacidade de adaptação é

obtida através do uso de técnicas de reflexão computacional para a alteração dos códigos das classes desenvolvidas pelo programador, das quais são derivados os componentes da aplicação. Essa alteração de código é realizada de maneira transparente e automatizada sem a necessidade da intervenção do programador. Ela visa incluir nas classes métodos auxiliares padronizados necessários para oferecer a capacidade de adaptação dinâmica às aplicações. Através desses métodos é possível realizar a interceptação das chamadas de serviços, solicitadas pelos componentes da aplicação, redirecionando-as para os *módulos* que representam os novos requisitos funcionais ou para os conectores que representam os requisitos não-funcionais. Essa solução utiliza tecnologias atuais amplamente disponíveis e adotadas em âmbito comercial, como Java e o padrão JMX [Sun 2005], não requerendo o uso de mecanismos especializados ou de uso restrito. Em particular, o uso dessas técnicas permite que aplicações usando o padrão Java *J2SE*, ou usando o padrão JMX, possam ser adaptadas dinamicamente conforme suas necessidades. Além disso, o *SDA-A* pode trabalhar em conjunto com servidores de aplicações comerciais, como o JBoss [Fleury e Reverbel 2003, STARK 2003], gerenciando as aplicações (ou *módulos*) carregadas neste servidor e permitindo configurá-las de forma customizada. O *SDA-A* também apresenta um desempenho, em relação ao custo para troca e/ou inserção de novos componentes, satisfatório comparado a outros trabalhos com objetivos semelhantes, e.g. [Almeida et al. 2001].

Através do suporte concebido às aplicações distribuídas, e.g., aplicações de computação autônoma e ubíquas, podem ser adaptadas dinamicamente permitindo assim, atender as novas necessidades impostas pelos ambientes nos quais elas são executadas.

Vale a pena ressaltar que os esforços realizados neste trabalho resultaram em uma publicação [Santos et al. 2006] na Conferência Latino Americana de Informática (CLEI, Santiago, Chile, 2006).

1.1 Organização do Texto

Os demais capítulos deste texto estão organizados da seguinte forma:

Capítulo 2, apresenta as ferramentas e tecnologias estudadas para o desenvolvimento do suporte a adaptação dinâmica de arquiteturas;

Capítulo 3, apresenta o *SDA-A* e os aspectos relevantes de sua implementação;

Capítulo 4, apresenta alguns exemplos utilizando o suporte desenvolvido e os resultados de medidas de desempenho obtidos através de experimentos realizados;

Capítulo 5, apresenta alguns trabalhos relacionados estudados, comparando-os com o *SDA-A*;

Capítulo 6, apresenta a conclusão deste trabalho e propostas para trabalhos futuros.

Capítulo 2

Ferramentas e Tecnologias Estudadas

Antes de abordar o suporte para adaptação dinâmica de arquiteturas (*SDA-A* o qual será apresentado no Capítulo 3), neste capítulo serão apresentadas as tecnologias e ferramentas que foram utilizadas em seu desenvolvimento. Através do uso dessas tecnologias e ferramentas, juntamente com as técnicas de adaptação dinâmicas citadas no Capítulo 1, o *SDA-A* possibilita realizar adaptações dinâmicas em aplicações sem que seja necessário interromper as execuções das mesmas.

Com o objetivo de identificar os recursos que contribuem para implementação do suporte de adaptação dinâmica, as tecnologias Javassist e *Java Management Extensions* (JMX) e o servidor de aplicações JBoss foram estudadas e posteriormente utilizadas.

O Javassist (Seção 2.1) foi utilizado para poder ajustar classes Java desenvolvidas pelos programadores, de forma que elas possam ser adaptadas dinamicamente e/ou manipular o fluxo da execução da aplicação através da inserção de códigos padronizados pelo suporte desenvolvido (os quais também serão apresentados no Capítulo 3).

O JMX (Seção 2.2) foi escolhido por fornecer facilidades para adicionar, remover e atualizar serviços dinamicamente, sem a necessidade de parar completamente uma aplicação em execução. Além disso, existem dois pontos importantes para a escolha do JMX que devem ser destacados: (i) ele encontra-se disponível a partir da versão 1.5.0 do Java, o que facilita a utilização do *SDA-A*, (ii) o padrão JMX é a tecnologia base do servidor de aplicações JBoss, o que possibilita que aplicações nele hospedadas sejam gerenciadas através do *SDA-A*.

Como atualmente existe um grande número de aplicações Java em âmbito comercial que utiliza o servidor de aplicações JBoss, foram analisadas então as possibilidades e

viabilidades de oferecer a capacidade de adaptação dinâmica para as aplicações nele hospedadas.

Ao longo desse capítulo serão apresentados os aspectos relevantes das tecnologias e ferramentas mencionadas anteriormente, visando demonstrar sua utilização no *SDA-A*.

2.1 Javassist

Javassist [Chiba e Nishizawa 2003] é um *toolkit* baseado em reflexão para instrumentação de Java *bytecode*. Ele inclui um conjunto de ferramentas que oferece suporte no desenvolvimento de aplicações Java, possibilitando criar novas classes, modificar o corpo de métodos, adicionar novos métodos a classes existentes, entre outras funcionalidades.

Projetistas utilizando o Javassist podem transformar arquivos *class* em vários meta-objetos, representando classes, métodos e atributos. Especificamente, o código dos meta-objetos podem ser acessados para realizar modificações, tais como: adicionar interfaces, métodos e atributos. As modificações realizadas nos meta-objetos são refletidas nos seus respectivos arquivos *class*.

O Javassist também disponibiliza mecanismos para realizar reflexão estrutural, como a API *reflection* do Java, e reflexão comportamental, também conhecida na literatura como reflexão computacional. A reflexão estrutural do Javassist é obtida através dos objetos `CtClass`, `CtMethod` e `CtField`, que também oferecem métodos que permitem realizar alterações em classes, métodos e atributos (Tabela 1). Nem todos os métodos apresentados na Tabela 1 são utilizados na implementação do *SDA-A*. Essa tabela apresenta uma visão dos principais métodos (funcionalidades) oferecidos pelo Javassist. Já no Código 1 e no Código 2 são apresentados dois exemplos que fazem uso de alguns métodos utilizados pelo *SDA-A*, como por exemplo, o `setBody` e o `insertBefore`. Esses métodos possibilitam redirecionar o fluxo de execução para os métodos padronizados do *SDA-A*. No próximo capítulo serão apresentados, em maiores detalhes, todos os métodos padronizados definidos pelo suporte.

Métodos de CtClass	Descrição
`void setName(String name)`	Muda o nome da classe.
`void setModifiers(int m)`	Muda as permissões da classe tal como public.
`void setSuperClass(CtClass c)`	Muda a super classe da classe.
`void setInterfaces(CtClass[] i)`	Muda a interface da classe.
`void addField(CtField f, String i)`	Adiciona um novo atributo.
`void addMethod(CtMethod m)`	Adiciona um novo método.

`Void addConstructor(CtConstructor[] c)`	Adiciona um novo construtor.
Métodos de CtField	**Descrição**
`void setName(String name)`	Muda o nome do atributo.
`void setModifiers(int m)`	Muda as permissões do atributo tal como public.
`void setType(CtClass c)`	Muda o tipo do atributo
Métodos de CtMethod	**Descrição**
`void setName(String name)`	Muda o nome do método.
`void setModifiers(int m)`	Muda as permissões do método tal como public.
`void setExceptionType(CtClass[] t)`	Define o tipo da exceção que o método pode lançar.
`void setBody(String b)`	Muda o corpo do método.
`void insertBefore(String b)`	Insere um trecho de código no inicio do corpo do método.

Tabela 1 - Alguns métodos do Javassist para modificar classes

Com a insersão de “ganchos” através do método `insertBefore` (Código 2, linha 8) ou através da modificação do corpo dos “métodos” com o método `setBody` (Código 1, linha 5) é possível realizar a reflexão comportamental oferecida pelo Javassist. O processo de adicionar “ganchos” no corpo dos métodos é utilizado pelo *SDA-A*, para garantir que os métodos padronizados, que possibilitam a capacidade de interceptação de chamadas, sejam executados quando uma chamada de serviço entre os componentes de uma aplicação é solicitada.

```
01 public void mudarConteurdoMetodo (String conteudo){
02   try{
03     CtClass cc = c ClassPool.getDefault().get(“Util”);
04     CtMethod m = cc.getDeclaredMethod(“aux”);
05     m.setBody(conteudo);
06     cc.writeFile(DIR);
07     cc.defrost();
08   } catch(NotFoundException nfe){
09       System.out.println(nfe.getMessage());
10   } catch(Exception e){
11       System.out.println(e.getMessage());
12   }
13 }
```

Código 1 - Método que modifica o conteúdo de um determinado método

O Código 1 apresenta um método de uma classe que utiliza o Javassist para modificar a implementação do método “`aux`” da classe “`Util`". Na linha 3 é criado um objeto do tipo `CtClass` que permite realizar a reflexão estrutural da classe “`Util`". Na linha 4 é criado um objeto do tipo `CtMethod` que recebe o método “`aux`” da classe “`Util`". Na linha 5 é feita a troca de todo o corpo do método “`aux`” pela string “`conteudo`”. Por fim, na linha 6 e 7 é feita a efetivação das alterações no arquivo *class* e liberado o objeto para que ele possa ser utilizado pela maquina virtual java (*Virtual Machine* - VM).

```
01 public void adicionarNovoMetodo(String metodo){
02   try{
```

```
03     ClassPool pool = ClassPool.getDefault();
04     CtClass cc = c pool.get(“Util”);
05     CtNewMethod novoMetodo = new CtNewMethod();
06     cc.addMethod(novoMetodo.make(metodo,cc));
07     CtMethod m = cc.getDeclaredMethod(“aux”);
08     m.insertBefore(“ novoMetodo(); ”);
09     cc.writeFile(DIR);
10     cc.defrost();
11   } catch(NotFoundException nfe){
12       System.out.println(nfe.getMessage());
13   } catch(Exception e){
14       System.out.println(e.getMessage());
15   }
16 }
```

Código 2 - Método que insere um gancho no corpo de um método específico

O Código 2 apresenta um método que insere um gancho no corpo de um método específico, neste caso o método “`aux`”, para desviar seu fluxo de execução para o método “`novoMetodo()`” (linha 8).

Com a utilização do Javassist no *SDA-A* é possível realizar a reflexão comportamental em uma aplicação. O Javassist apresenta várias funcionalidades que permitem realizar adaptações em uma aplicação sem que o projetista tenha o conhecimento do seu código fonte. Mas, há uma limitação, pois ele não oferece reflexão comportamental em tempo de execução, ou seja, as alterações realizadas em arquivos *class*, que já foram carregados pela VM, não terão efeito até que os mesmos sejam recarregados, sendo necessário, encerrar a aplicação e iniciá-la novamente. Isso ocorre devido a VM não oferecer recursos para realizar a atualização dos arquivos *class* dinamicamente. Para contornar este problema o *SDA-A* realiza as adaptações das classes antes da VM carregar as classes (antes de instanciar os objetos). Maiores detalhes serão abordados no Capítulo 3 na Seção 3.6. Detalhes sobre a utilização do Javassist no *SDA-A* podem ser obtidos no Apêndice B desta dissertação.

2.2 O Java Management Extensions (JMX)

O padrão *Java Management Extensions* (JMX) [Sun 2005] define uma arquitetura para gerenciamento dinâmico de recursos (aplicações, sistemas ou dispositivos de rede) distribuídos através da rede. No JMX os recursos devem ser organizados de forma a se tornarem gerenciáveis. A esses recursos, são associados um ou mais componentes de gerenciamento. O gerenciamento oferecido pelo JMX permite carregar, descarregar e mudar esses componentes dinamicamente, sem parar a execução das aplicações, sistemas ou dispositivos que estão sendo gerenciados. A arquitetura do JMX (Figura 1) compõe-se em três níveis, identificados na figura por linhas pontilhadas.

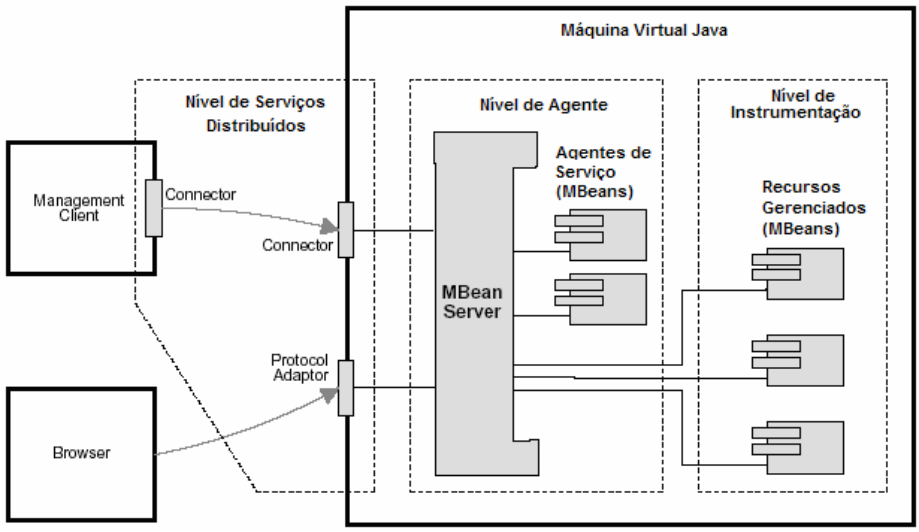

Figura 1 - Arquitetura do JMX

O nível de instrumentação define como organizar recursos de forma que eles possam ser monitorados e manipulados por aplicações de gerenciamento. Essa organização é provida por um ou mais *beans* de gerenciamento (*MBeans*), objetos Java que estão de acordo com certas convenções e expõem uma interface de gerenciamento para seus clientes. Nesse nível estão localizados, por exemplo, os *MBeans* que oferecem os serviços para as aplicações clientes. Os componentes das aplicações gerenciadas pelo *SDA-A* que são adaptados para o padrão JMX enquadram-se nesse nível.

O nível de agente define um agente que gerencia o conjunto de recursos que são organizados, de acordo com o nível de instrumentação, dentro de uma máquina virtual Java, em favor de aplicações de gerenciamento. Por exemplo, um agente pode ser considerado um *MBean* que desconhece as aplicações externas, mas que conhece e gerencia alguns *MBeans*, disponíveis no seu servidor de *MBeans*, podendo disponibilizar novos *MBeans* ou até mesmo remover *MBeans* existentes. Os Executores do *SDA-A* enquadram-se nesse nível.

O nível de serviços distribuídos especifica como aplicações de gerenciamento interagem com agentes remotos JMX e como a comunicação de agente-para-agente acontece. Isso consiste de conectores e adaptadores de protocolos, implementados como *MBeans*. Por exemplo, aplicações de gerência de recursos de rede podem utilizar agentes *MBeans* SMTP [Sun 2005] para obter informações sobre um determinado recurso.

O servidor *MBean* provê um registro para componentes JMX (*MBeans*) e faz a intermediação de qualquer acesso (solicitações de clientes) às interfaces de gerenciamento desses componentes. Ao se registrar um *MBean*, é associado a ele um nome de objeto que deverá ser único no contexto do servidor *MBean*. Clientes usam esses nomes de objetos, em

vez de referências Java, para referenciar os *MBeans*. Para invocar uma operação de gerenciamento em um *MBean*, um cliente local, tipicamente outro *MBean*, usa o nome do objeto do *MBean* alvo. O servidor *MBean* procura esse nome em seu registro e repassa a invocação para o *MBean* que foi invocado. O *SDA-A* utiliza servidores *MBean* para gerenciar uma aplicação utilizando o padrão JMX. A Figura 2 ilustra esse processo.

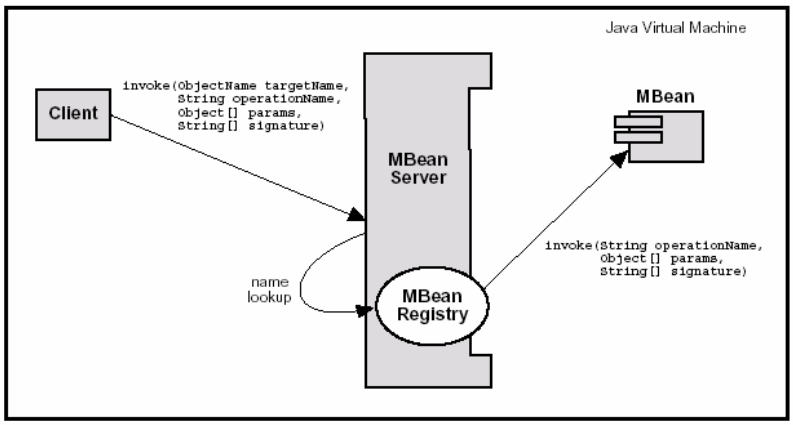

Figura 2 - Invocação de método em um MBean dinâmico

O servidor *MBean* introduz um nível de acesso indireto aos *MBeans*, eliminando assim o acoplamento dos mesmos com seus clientes. Dessa maneira, esses clientes não precisam referenciar o *MBean* diretamente. Além disso, os clientes não necessitam de nenhuma informação sobre as classes Java do *MBean*, nem das interfaces Java que ele implementa (nem mesmo em tempo de execução). Tudo que esses clientes necessitam saber é o nome do objeto do *MBean* e sua interface de gerenciamento, que pode ser obtida em tempo de execução (maiores detalhes podem ser encontrados em [Sun 2005]). Essa organização simples favorece a obtenção da capacidade de adaptação (explorada pelo *SDA-A*): a ausência de referências para um *MBean*, distribuídos através de seus clientes, facilita a sua substituição, ou seja, o fato de um cliente não precisar conhecer as classes e as interfaces Java de um *MBean* facilita as mudanças dinâmicas na implementação e na interface de gerenciamento do *MBean*.

2.2.1 *MBean* dinâmico e *MBean* padrão

O JMX suporta dois tipos de *MBeans*, o tipo dinâmico e o tipo padrão, que diferem na exposição de seus atributos e operações de gerenciamento para o servidor *MBean*. O tipo dinâmico implementa uma interface Java predefinida, sem levar em consideração sua interface de gerenciamento, e conta com um meta-dado para especificar essa interface. O tipo padrão implementa uma interface Java definida depois da interface de gerenciamento do

MBean. O tipo de um *MBean* é um detalhe de implementação oculto para os clientes, que acessam ambos os tipos da mesma forma.

MBeans do tipo padrão são fáceis de implementar, pois eles poupam os projetistas da tarefa de construir instâncias de meta-dado para descrever interfaces de gerenciamento. Por outro lado, *MBeans* dinâmicos são mais flexíveis, já que as definições de suas interfaces de gerenciamento podem ser postergadas até o tempo de execução. Ambos os tipos de *MBeans* suportam alguma forma de evolução de interface de gerenciamento. No entanto, no caso de *MBeans* do tipo padrão, a evolução requer substituição do objeto, pois é necessário redefinir a interface do *MBean* por completo. Já o tipo dinâmico, suporta a evolução sem requerer substituição do objeto, pois ele possui atributos, internos ao *MBean*, que armazenam as meta-informações que fornecem a lista de interfaces disponíveis para o *MBean* em questão.

O *SDA-A*, durante o processo de adaptação das classes do programador (Seção 3.6), adota o *MBean* tipo padrão criando as interfaces de serviços necessárias para gerenciar os componentes de uma aplicação, como será apresentado no Capítulo 3.

2.2.2 Reflexão no JMX

O JMX também pode ser visto como uma arquitetura reflexiva. Ele possui métodos que permitem a introspecção de *MBean*, ou seja, ele possui métodos que disponibilizam informações sobre as interfaces de serviços oferecidas por um determinado *MBean*. A substituição de objeto disponibiliza uma forma simples de adaptação ao nível de *MBean*: clientes referenciam *MBeans* pelo nome do objeto, assim, o comportamento de um *MBean* pode ser mudado apenas substituindo-o por outro objeto com um comportamento diferenciado. O nível de agente inclui um serviço de carregamento de classes dinâmico, que facilita a substituição de objetos. Esse serviço permite que *MBeans* sejam instanciados usando novas classes Java, que são carregadas de servidores remotos.

O *SDA-A* utiliza o JMX, explorando as facilidades de adaptação dinâmica oferecidas por ele, em conjunto com o Javassist, para transformar uma aplicação Java J2SE em uma aplicação que adota o padrão JMX. Dessa forma, essas aplicações transformadas terão a capacidade de adaptação dinâmica.

2.3 Servidor de Aplicações JBoss

Como foi mencionado no início desse capítulo, atualmente existe um grande número de aplicações, em âmbito comercial, que utilizam o servidor de aplicações JBoss. Com ele foi possível demonstrar a possibilidade de gerenciamento das aplicações nele hospedadas através de primitivas de alto nível de abstração. Este gerenciamento consiste em adaptar as aplicações de modo a atender as novas demandas por elas exigidas (como apresentado no Capítulo 1). Além disso, foi possível demonstrar a capacidade de adaptação dos módulos de serviço do JBoss. Maiores detalhes sobre esse gerenciamento serão apresentados no Capítulo 3, Seção 3.9.

O JBoss é um servidor de aplicação Java que inclui um conjunto de componentes que implementa a especificação *Java 2 Enterprise Edition* (J2EE), porém seu escopo vai além do J2EE. Ele é um *middleware* aberto, no sentido que os projetistas podem estender seus serviços implantando dinamicamente novos componentes no servidor [Fleury e Reverbel 2003, Stark 2003]. A base de seus componentes é provida pelo padrão JMX, sendo que a partir da versão 4, foi incorporado um mecanismo de programação orientada a aspectos (AOP, do inglês *Aspect-Oritented Programming*) através de um *framework* (JBoss-AOP) incluído ao seu núcleo [Burke e Bock 2004]. Isso habilitou desenvolvedores a adicionarem novos serviços não-funcionais (persistência de objeto, cache, replicação, transação, segurança, etc.) a uma aplicação, depois do seu ciclo de desenvolvimento, sem alterar uma linha de código.

Em servidores como o JBoss, extensíveis e configuráveis dinamicamente, dois tipos gerais de componentes podem ser implantados: componentes de *middleware* e componentes da aplicação. Nessa abordagem, a maior parte das funcionalidades do servidor de aplicação é realizada por um conjunto de componentes de *middleware* implantados em um servidor mínimo (micronúcleo), provavelmente, devido às diferenças de requisitos entre os componentes de *middleware* e os componentes da aplicação, múltiplos modelos de componentes existirão em um servidor de aplicação baseado em componentes. Uns atendendo às necessidades do *middleware* e outros às necessidades das aplicações.

2.3.1 Estrutura base do JBoss

O padrão JMX (Seção 2.2) provê a base para os componentes de *middleware* do JBoss. Em cima dele, o JBoss introduz o seu próprio modelo para componentes de *middleware* centralizado no conceito de componente de serviço. O modelo de componente de

serviço do JBoss estende e refina o modelo JMX, para lidar com alguns requisitos além do escopo do JMX: ciclo de vida do serviço, dependências entre serviços, implantação e re-implantação de serviços, configuração dinâmica e reconfiguração de serviços e empacotamento de componente. Quase todas as funcionalidades do servidor de aplicação do JBoss são providas, de forma modular, pelos componentes de serviços que são plugados ao micronúcleo baseado em JMX.

No contexto do *SDA-A,* o JBoss hospeda um componente de serviço do *SDA-A*, denominado *Executor* (o qual será abordado no Capítulo3, Seção 3.9). O *Executor* é o um dos elementos da arquitetura do *SDA-A* responsável por realizar as adaptações dinâmicas das aplicações ou de seus componentes, que estão hospedados no mesmo servidor onde o *Executor* se encontra. Maiores detalhes sobre os elementos da arquitetura do *SDA-A* e o funcionamento dos *Executores* serão apresentados no próximo capítulo.

2.4 Conclusão do Capítulo

O objetivo deste capítulo foi apresentar de forma breve as tecnologias e ferramentas que foram utilizadas no desenvolvimento do *SDA-A.* Foi mostrado que com a utilização do Javassist é possível adaptar as classes desenvolvidas pelos programadores, sem ter previamente o conhecimento das mesmas, adicionando a elas métodos padronizados pelo *SDA-A*, que oferecerão a capacidade de interceptação de chamadas de serviços (será apresentado no Capítulo 3). Adotando-se o JMX, o processo de adaptação dinâmica dos componentes de uma aplicação torna-se mais simples, pois seu modelo oferece um desacoplamento entre o componente provedor de serviço e o componente solicitante do serviço. Além disso, o JMX encontra-se disponível para os programadores na versão 1.5.0 do Java. Um outro ponto positivo é que o JMX é a tecnologia base do servidor de aplicações JBoss, possibilitando assim que aplicações hospedadas nele sejam gerenciadas através do *SDA-A.* Além do mais, o JBoss é um servidor de aplicações amplamente utilizado em âmbito comercial, incentivando assim, uma iniciativa de oferecer, através de um alto nível de abstração, a capacidade de adaptação dinâmica para as aplicações ou componentes de aplicações nele hospedadas.

Capítulo 3

Suporte a Adaptação Dinâmica de Arquiteturas

Mediante um estudo realizado sobre técnicas de adaptação dinâmica (apresentadas posteriormente no Capítulo 5), uma avaliação das tecnologias e ferramentas que oferecem recursos necessários para efetivar as adaptações (Capítulo 2), e com base nas necessidades de adaptação das classes de aplicações de interesse apresentadas no Capítulo 1, foi identificada a possibilidade do desenvolvimento de um suporte para adaptação dinâmica de arquiteturas. Além disso, também foi identificada a possibilidade deste suporte adaptar aplicações ou componentes de aplicações em execução no servidor de aplicação JBoss (apresentado na Seção 2.3).

Neste capítulo é apresentado o SDA-A (*Support for Dynamic Architecture-Adaptation*), um suporte para adaptação dinâmica de arquiteturas que se baseia no *design pattern Architecture Configurator* [Carvalho et al. 2002] para oferecer, através de um alto nível de abstração, a capacidade de adaptação dinâmica para aplicações. Através dele, os programadores de sistemas distribuídos podem utilizar recursos disponíveis sem se preocupar com a localização física (em qual *host* o recurso foi disponibilizado) dos mesmos (ver Seção 3.5). Nas próximas seções serão apresentados alguns conceitos preliminares e os aspectos de implementação que possibilitam oferecer a capacidade de adaptação dinâmica para aplicações.

3.1 Arquitetura de Software

Reutilização, modularidade, abstração e separação de interesses constituem-se em alguns conceitos há muito investigados. Tais conceitos são entrelaçados e têm como

fundamento a concepção de sistemas de software de qualidade, compostos de elementos reaproveitáveis em outros contextos e projetos.

Alguns desses conceitos, como reutilização e modularidade, remontam as linguagens de programação, em que o código, escrito através de módulos coesos e fracamente acoplados (*procedures*, *functions*, blocos, etc.), pode ser reutilizado em outros programas. Entretanto, um sistema de software não pode ser concebido partindo-se apenas da especificação de algoritmos ou estruturas de dados. É preciso que seja desenvolvido tendo-se sua organização como uma composição de componentes e interligações entre estes componentes [Shaw e Garlan 1996].

Um sistema de software também pode ser concebido a partir da descrição de uma arquitetura. Tal arquitetura especifica a estrutura do sistema e sua topologia, mostrando a correspondência entre os requisitos do sistema e os seus elementos. Os módulos funcionais do sistema (componentes) e os elementos responsáveis por suas interações, chamados conectores, são definidos através de tipos abstratos (classes, ver Seção 3.1.1), formando a estrutura da arquitetura. A topologia, por sua vez, é caracterizada pela configuração, através da qual instâncias de módulos e conectores são criadas e interligadas.

Além de *módulo* e *conector*, a *porta* é outro elemento fundamental que compõe a arquitetura de um sistema de software. *Portas* são objetos tipados que identificam pontos de interação entre um *módulo* e *conector* e o ambiente em que foi criado [Loques et al. 1999]. O conjunto de *portas* de um *módulo* ou *conector* representa sua interface. A definição de interface torna possível a interligação entre *módulos* e *conectores* para formar a topologia da arquitetura.

Módulos, *conectores* e *portas* podem ser definidos e especificados de forma independente uns dos outros e posteriormente configurados em uma arquitetura, ou ainda reutilizados em diferentes contextos. Por exemplo, em um sistema cliente-servidor de três camadas, o servidor, o banco de dados e o código que permite a comunicação entre eles (*conector*) podem ser construídos separadamente.

A pesquisa em arquitetura de software visa a redução de custos no desenvolvimento e manutenção de aplicações [Perry 1992, Shaw e Garlan 1996]. Tal pesquisa fornece um entendimento amplo do sistema, pois permite ao projetista concentrar-se nos elementos arquiteturais (*módulos* e *conectores*) e na sua estrutura de interconexão (configuração). Isso

proporciona ao projetista um alto nível de abstração, fazendo-o concentrar-se na estrutura global, e não em detalhes específicos do desenvolvimento do sistema.

Além de oferecer abstração no nível de sistema, o estudo e aplicação de arquiteturas incentivam a obtenção da propriedade de separação de interesses: uma arquitetura de software auxilia na separação entre o processamento feito por um sistema (realizado pelos *módulos*) e a interação entre suas unidades computacionais (realizada pelos conectores) [Medvidovic 1999]. Visando à separação de interesses, as funções básicas de uma aplicação (propriedades funcionais) são diferenciadas dos aspectos operacionais ou propriedades não-funcionais, ou de qualquer outro aspecto não coberto na descrição funcional. Dessa forma, Arquiteturas de Software permitem que projetistas concentrem-se separadamente nos aspectos funcionais da aplicação, podendo encapsulá-los nos *módulos*, e nos não-funcionais, podendo encapsulá-los nos *conectores* [Sztajnberg 2002].

A definição e especificação dos elementos de uma arquitetura (módulos, conectores e portas) são feitas através de uma linguagem de descrição arquitetural (ADL - *Architecture Description Language*). Através da linguagem, são especificados tipos abstratos para módulos e conectores, instanciações dos mesmos e definidas as ligações entre eles – o estabelecimento da configuração. Mas, nem todas as ADLs suportam a definição de instanciações dos elementos que compõem a arquitetura, tornando necessário a presença de um suporte ou mecanismo, como o *SDA-A*, que se encarregue de instanciar os elementos da arquitetura.

Serão tratados nas próximas seções os elementos fundamentais de uma arquitetura de software (*módulos*, *conectores* e *portas*).

3.1.1 Elementos de uma arquitetura

Módulos e portas

Um módulo refere-se a um processo, objeto, procedimento, ou qualquer pedaço de código ou dados identificável. Os *módulos* podem corresponder a unidades de compilação de linguagens de programação convencionais e objetos tais como arquivos [Shaw et al. 1996], ou ainda, no nível de linguagens de programação, correspondem a módulos de programação, classes, objetos ou um conjunto de funções [Buschmann et al. 1996].

Os *módulos* em geral possuem uma interface, através da qual podem fornecer ou requerer serviços. No nível de linguagem de programação, por exemplo, os métodos, funções

e procedimentos definidos podem representar serviços oferecidos, e invocações a métodos, funções e procedimentos podem representar requerimentos de serviços.

Na descrição arquitetural de um sistema, um *módulo* deve funcionar como um tipo ou uma classe, e possuir uma especificação que defina seus pontos lógicos de interação com outros componentes. O conjunto de tais pontos de interação (as *portas*), representa a interface do componente, podendo incluir assinatura, funcionalidade e propriedades da interação.

As *portas* podem ser de dois tipos: de saída ou de entrada. Portas de saída representam serviços que podem ser requisitados pelo módulo (invocações de métodos) e portas de entrada associam-se a serviços fornecidos pelo módulo (métodos definidos). A Figura 3 mostra a representação de dois componentes de um sistema cliente-servidor simples, onde Cliente possui uma porta de saída e Servidor uma porta de entrada.

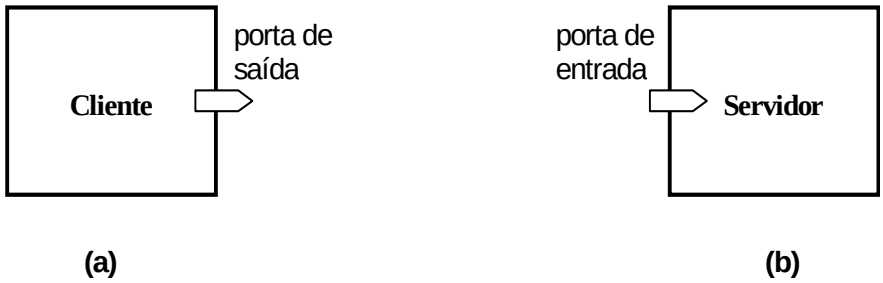

Figura 3 - Representação de componentes. (a) Cliente tem uma porta de saída. (b) Servidor tem uma porta de entrada.

Conectores

A interação entre os *módulos* em uma arquitetura de software é governada por um ou mais *conectores*. Em princípio, eles encapsulam a funcionalidade necessária à interação, sem que seja necessário modificar ou adaptar as interfaces dos *módulos* envolvidos. Eles também podem representar os requisitos não-funcionais de uma aplicação, como por exemplo, mecanismos de autenticação ou criptografia [Shaw et al. 1996].

Durante o processo de configuração, podem ser criadas várias instâncias de um mesmo tipo de *conector* para intermediar as interações entre instâncias de componentes. A configuração é realizada interligando-se as *portas* de um *conector* às *portas* dos componentes envolvidos.

Os *conectores* possuem características que os tornam especialmente atraentes para tratar os aspectos relacionados com a interação e a comunicação entre componentes [Bishop e Faria 1996]. Dentre estas características destacam-se:

- O conhecimento da interface e as referências dos componentes interligados por eles;
- A capacidade de examinar o conteúdo de requisições e respostas antes de encaminhá-las aos seus destinos finais;
- A possibilidade de controlar e manipular o repasse das requisições e respostas para os seus destinos;
- A sua similaridade com componentes quanto às propriedades de configuração, facilitando sua criação através da composição de outros *conectores*.

Como exemplo, considere a arquitetura da Figura 4 para um sistema cliente-servidor simples, no qual os *módulos* têm suas interações governadas por um *conector* (*C-S*). Cliente e Servidor têm suas *portas* interligadas às *portas* do *conector* C-S, tornando possível a interação entre os mesmos. Dessa maneira, este *conector* registra todas as solicitações provenientes do Cliente, ou seja, as requisições de Cliente, executadas através da porta de saída, são interceptadas e encaminhadas ao componente Servidor.

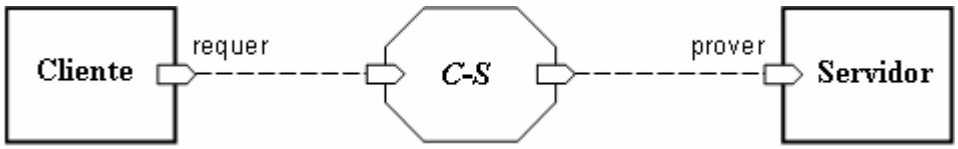

Figura 4 - Arquitetura de um sistema cliente-servidor simples. Os módulos Cliente e Servidor têm sua interação intermediada pelo conector C-S.

Na próxima seção será apresentada a linguagem de descrição arquitetural CBabel e um exemplo de como a arquitetura da Figura 4 é representada através da mesma.

3.2 A Linguagem de Descrição Arquitetural CBabel

CBabel (*Building Applications by Evolution with Connectors*) [Sztajnberg 2002] é a ADL utilizada pelo *SDA-A* para descrever a arquitetura de uma aplicação e a topologia de interconexão dos componentes que compõem essa arquitetura. Ela também permite descrever contratos de *Quality of Service* (QoS), que representam requisitos não-funcionais, como abordado em [Corradi 2005, Freitas 2005].

3.2.1 Exemplo de uma arquitetura descrita em CBabel

Para ilustrar uma utilização da ADL CBabel, considere a arquitetura da aplicação cliente-servidor apresentada na Seção 3.1.1, Figura 4. O Código 3 descreve através da ADL CBabel essa arquitetura. Na linha 02 são definidas as *portas* que representam os pontos de interação entre os *módulos* da aplicação. Nas linhas 03 a 08 são definidas as classes de *módulos* `Cliente` e `Servidor` e suas respectivas instâncias (`cliente` na linha 05 e `servidor` na linha 08). Uma instância da *porta* `requer` é declarada como porta de saída (`out`) em `Cliente` (linha 04), ou seja, representa a requisição de serviço ao servidor. Já a instância da *porta* `prover`, é definida como porta de entrada (`in`) em `Servidor` (linha 07), indicando o provimento de serviço. O *conector* da arquitetura (`C-S`) é declarado nas linhas 09 a 12. Nele, a *porta* `requer` é declarada como porta de entrada, pois ela oferece o ponto de conexão lógica com o `Cliente`, e a *porta* `prover` é definida do tipo saída, pois é através dela que o *conector* acessa o serviço oferecido pelo `Servidor`. Nas linhas 13 e 14 os *módulos* são instanciados (através do uso da primitiva arquitetural `instantiate`), sendo que as palavras `HostServidor` e `HostCliente` representam, respectivamente, os nomes remotos dos *hosts* onde serão instanciados o servidor e o cliente. Por fim, na linha 15 o *módulo* `Cliente` é ligado ao `Servidor` através de uma instância do conector definido (primitiva arquitetural `link`).

```
01  module ClienteServidor {
02    port requer, prover;
03    module Cliente {
04       out port requer;
05    } cliente;
06    module Servidor {
07       in port prover;
08    } servidor;
09    connector C-S {
10       in port requer;
11       out port prover;
12    } cs;
13    instantiate servidor at HostServidor;
14    instantiate cliente at HostCliente;
15    instantiate cs at HostCS;
16    link cliente to servidor by cs;
17 } cliente_servidor;
```

Código 3 – Descrição em CBabel da arquitetura de uma aplicação cliente-servidor.

A partir da descrição arquitetural, apresentada no Código 3, o *SDA-A* consegue implantar essa arquitetura e colocá-la em execução. Esse processo é realizado através da interpretação da descrição arquitetural, as instanciações, ligações e inicializações dos componentes da arquitetura (ver Seção 3.7). Além disso, novos *módulos* descritos em CBabel (que não foram previstos em tempo de *design*) podem ser carregados de forma independente, ou seja, não é necessário recarregar toda a ADL novamente.

3.3 *Design Pattern Architecture Configurator*

O *SDA-A* é fundamentado no *design pattern Architecture Configurator* [Carvalho et al. 2002], que fornece a base para a implementação de configurações arquiteturais, com base nos mecanismos de interceptação, encaminhamento e manipulação de requisições, permitindo a interligação dos *componentes* (*módulos* e *conectores*) de uma aplicação.

A Figura 5 apresenta um diagrama de classes de uma aplicação Cliente/Servidor que adota o *design pattern Architecture Configurator*. De acordo com essa aplicação, os mecanismos de interceptação, encaminhamento e manipulação de requisições são aplicados de forma transparente em relação aos elementos básicos da arquitetura, suportando propriedades como reutilização, abstração e separação de interesses. Em particular, esses mecanismos permitem que os *componentes* participantes da arquitetura possam ser implementados de forma autônoma e integrados de acordo com a configuração arquitetural desejada.

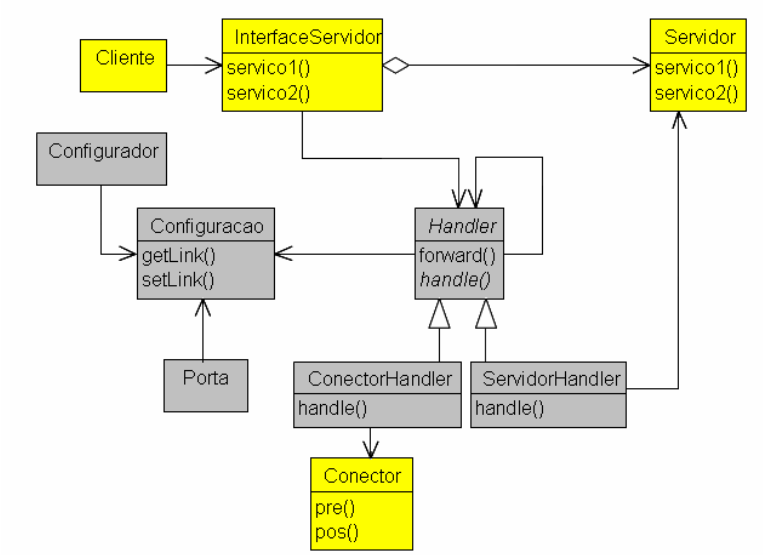

Figura 5 - Diagrama de classes de uma aplicação Cliente/Servidor utilizando o *design pattern Architecture Configurator*

O diagrama da Figura 5 é composto por classes da aplicação (Cliente, Conector, InterfaceServidor e Servidor) e classes definidas pelo pattern (Configurador, Configuracao, Porta, Handler, ConectorHandler e ServidorHandler).

As classes *Cliente* e *Servidor* representam os módulos funcionais e a classe *ConectorHandler* representa o conector da arquitetura. A programação da configuração está representada pela classe *Configuracao*.

Configurador é uma classe que define, de um modo geral, um mecanismo que interpreta as instruções de uma ADL, e a partir da descrição interpretada define os tipos para componentes e portas presentes na aplicação, e também procede com a instanciação e interligação dos mesmos para permitir a execução da aplicação. Para a realização de tais procedimentos são invocados os serviços da classe *Configuração*.

Configuração é uma classe que recebe as solicitações da classe *Configurador* para realizar a configuração de componentes e iniciá-los. Ela mantém duas categorias de informações: de descrição e de execução da arquitetura. Informações de descrição referem-se à arquitetura descrita através de uma ADL (por exemplo, o Código 3), enquanto as informações de execução relacionam-se às instâncias – referências – dos componentes e conectores durante o processo de execução da aplicação.

A classe *Handler* é uma classe abstrata que é responsável pelo encadeamento entre componentes conforme a descrição arquitetural, disponibilizada por *Configuração*. *Handler* utiliza-se também das informações de execução dessa mesma classe, uma vez que necessita

das referências aos objetos que representam componentes conectores no espaço de execução da aplicação. No modelo, a classe *Conector* é encadeada por *Handler*. *ServidorHandler* representa a classe *Servidor* no encadeamento e possui uma referência à mesma.

As requisições invocadas pelo *Cliente* são interceptadas pela classe *InterfaceServidor*, que possui a mesma interface de *Servidor*. *InterfaceServidor* busca em *Configuração* a referência ao *Conector* e invoca a operação *handle()* do mesmo. Conforme a porta de entrada configurada no *Conector*, *handle()* invoca a operação *servico1()* ou *servico2()* de *InterfaceServidor*. Ambas as operações invocam *forward(),* responsável por dar seguimento ao encadeamento controlado por *Handler*. Na seqüência, *ServidorHandler* tem sua operação *handle()* solicitada, a qual encaminha a requisição original para *Servidor*, finalizando o encadeamento.

De acordo com a Figura 5 é possível identificar algumas propriedades importantes:

As funcionalidades de módulos e conectores estão separadas dos processos de interceptação e encaminhamento (realizados pelas classes *InterfaceServidor* e *Handler*, respectivamente), possibilitando a reutilização de componentes.

A separação de requisitos é atendida, pois modificações funcionais ou de interface nos módulos não afetam os conectores, e vice-versa. Isso acontece porque são definidas classes distintas para módulos e conectores, nas quais as funcionalidades de cada *módulo* e *conector* são implementadas de forma bem definida e coesa – cabe ao *conector*, por exemplo, rotear as mensagens para os outros conectores e *módulos* interligados. Além disso, a classe Porta mantém o mapeamento entre os pontos de interação dos componentes – as portas de entrada e saída – e as operações definidas nas classes e invocações a operações, independentemente das interfaces.

Os processos de interceptação e encaminhamento das requisições ocorrem de forma transparente, sem que os *módulos* do sistema tomem conhecimento disso. *InterfaceServidor* possui a mesma interface de Servidor, o que permite que Cliente utilize seus serviços como se estivessem lidando diretamente com ele. A solução, portanto, é independente da arquitetura apresentada. Tal independência pode ser estendida a qualquer tipo de arquitetura, oferecendo ao desenvolvedor um nível de abstração bastante elevado.

A definição de novos conectores, e sua inclusão apropriada no encadeamento mantido por *Handler*, tornam possível a obtenção de novos requisitos não-funcionais, sem que seja

preciso alterar a interface ou o comportamento dos *módulos* funcionais. Tal fato contribui para a extensão do sistema de uma maneira independente de sua topologia corrente.

A classe *Configuracao* mantém informações de descrição e execução da arquitetura configurada. A descrição é composta basicamente pela definição das interfaces dos *módulos* e *conectores* e as ligações realizadas entre eles, enquanto que as informações de execução tratam de referências às instâncias dos objetos configurados. Nessa informação está contida a definição clara da interface dos *módulos*, e de como eles interagem entre si.

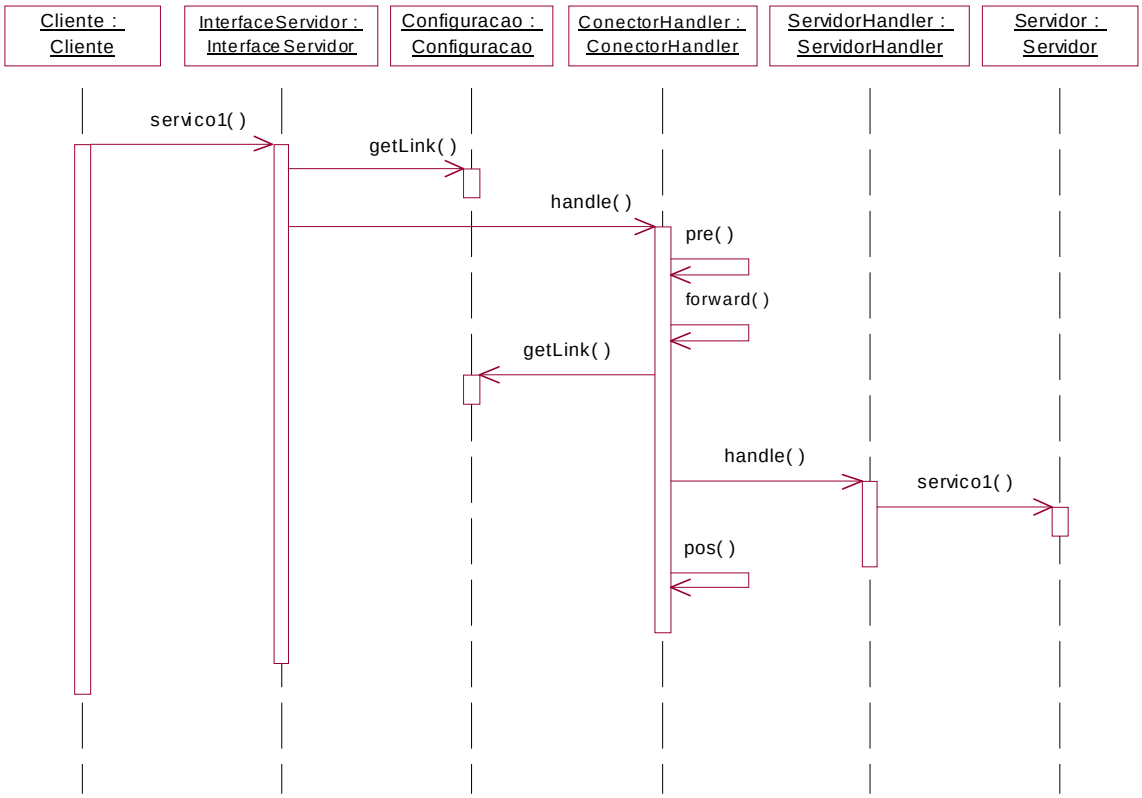

Figura 6 - Fluxo de execução da aplicação Cliente/Servidor

A Figura 6 apresenta o fluxo de execução da aplicação Cliente/Servidor ilustrada na Figura 5. O *Cliente* solicita um serviço (*servico1()*) oferecido por um *Servidor*. Todas as invocações a partir do *Cliente* são interceptadas por um objeto *InterfaceServidor*. Este encaminha a invocação através da cadeia de conectores, representados por instâncias de *ConectorHandler* e *ServidorHandler*. Cada uma das instâncias de *ConectorHandler* representa, preferencialmente, um aspecto não-funcional, implementado por um *conector*. Ou seja, a cada *conector* configurado, deve estar associada uma instância de *ConectorHandler*.

A invocação de um serviço corresponde a uma das portas de saída configuradas para o componente *Cliente*. A assinatura da porta é passada a *InterfaceServidor*, que, de posse disso, obtém em *Configuracao* a referência ao primeiro conector ligado a *Cliente*, e a porta de entrada ligada a sua porta de saída em questão. Feito isso, o método *handle()* de *conector* é invocado, com argumentos referentes à porta de entrada e à requisição realizada por Cliente.

O método *handle()* começa sua operação invocando o método *pre()* do respectivo *conector*, responsável pelos aspectos não-funcionais que devem ser executados antes do encaminhamento da requisição para o próximo *componente* configurado. Após a execução de *pre()*, o fluxo deve seguir por alguma porta de saída do *conector*. Definida a porta de saída, a operação *forward()* do conector é invocada. Tal operação utiliza a classe *Configuracao* para obter a referência ao próximo *componente* configurado, de modo análogo ao explicado anteriormente, e a requisição segue no encadeamento.

O final do encadeamento ocorre quando *ServidorHandler* é encontrado e tem sua operação *handle()* requisitada. Esta operação tem funcionamento análogo à de *ConectorHandler*, porém não invoca *forward()*, e sim concretiza a requisição junto a *Servidor*, invocando o método que implementa o serviço desejado.

No contexto das classes *InterfaceServidor* e *Handler*, portas de saída são representadas por invocações de operações, e portas de entrada são representadas por operações declaradas na interface de *componentes* (assinaturas).

É importante ressaltar que podem ser criadas várias classes que implementam aspectos não-funcionais, e também várias instâncias de cada uma delas. Em outras palavras, seria possível ter classes *ConectorHandler1*, *ConectorHandler2*, *ConectorHandler3*, etc., cada qual com uma ou mais instâncias. A mesma observação vale para as classes *Cliente*, *Servidor* e *ServidorHandler*, tornando fácil também a separação entre requisitos funcionais do sistema. Deve ser levado em conta o fato de que cada classe *Servidor* criada deve ter associada a si uma classe *InterfaceServidor* específica.

No *SDA-A* todas as classes presentes no *design pattern Architecture Configurator* são encapsuladas de forma automatizada junto às classes da aplicação desenvolvidas pelo usuário. Desta forma, a presença de cada uma delas se torna transparente do ponto de vista do usuário. Além disso, as informações sobre as configurações da aplicação ficam distribuídas em seus *componentes*, sendo assim, não é necessária a solicitação de informações para onde um fluxo de execução deve seguir. Com a utilização do SDA-A o usuário não necessita conhecer a

fundo os detalhes do *design pattern Architecture Configurator* durante o desenvolvimento das aplicações. Ele necessita apenas ter um conhecimento do comportamento do *pattern* (Figura 6), pois, o *SDA-A* se encarregará de gerar todas as classes e interfaces definidas no *pattern* como apresentado na Figura 5. Na Seção 3.7 será apresentado um fluxo de execução de uma aplicação, a qual utiliza o *SDA-A*, onde é possível identificar o mesmo comportamento do fluxo de execução entre os componentes da aplicação como apresentado na Figura 6.

3.4 Arquitetura do Suporte para Adaptação Dinâmica de Arquiteturas - *SDA-A*

A arquitetura do *SDA-A* é composta por dois componentes: *Coordenador* e *Executor*, através dos quais as aplicações podem ser gerenciadas dinamicamente. Cada um deles possui responsabilidades bem definidas, além de possuir métodos pelos quais eles podem interagir entre si; a Figura 7 ilustra um cenário onde o Coordenador interage com N executores.

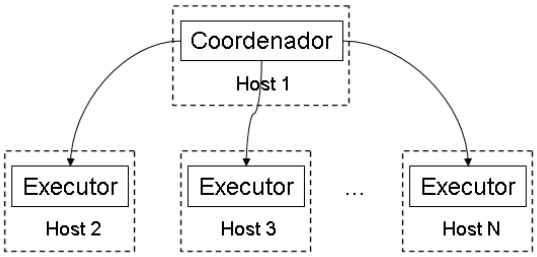

Figura 7 - Arquitetura do *SDA-A*.

3.4.1 Coordenador

O *Coordenador* tem a responsabilidade de interpretar a descrição da arquitetura de uma aplicação, elaborada em CBabel, e transformá-la em informações que possam ser armazenadas em uma estrutura de dados interna. O Coordenador também é responsável por implantar (através dos Executores) a configuração da arquitetura descrita através da ADL. Para esse fim, ele disponibiliza uma interface *apply(String command)*, para que as aplicações carregadas através do *SDA-A* possam ser gerenciadas externamente. Através do Coordenador, as aplicações podem obter informações sobre a configuração da arquitetura, como por exemplo, as interligações entre os elementos que compõem a aplicação. O *SDA-A* inclui apenas uma instância do Coordenador, que é responsável por delegar tarefas às instâncias remotas de *Executores*.

As interfaces oferecidas pelo Coordenador são:

Tabela 2 - Interfaces oferecidas pelo Coordenador

Application GetApplication ( )	Retorna uma estrutura de dados que contêm todas as informações da configuração arquitetural da aplicação. Aplicações externas podem utilizar essa interface para obter informações sobre a configuração da arquitetura de uma aplicação.
Void Apply (String command)	Interpreta os comandos passados como parâmetro e os executa sobre a configuração da arquitetura da aplicação gerenciada. Por exemplo, o comando *Apply ("instantiate Servidor at localhost;")* é interpretado e mapeado para a chamada de método interna *"Load("Servidor","localhost")"*, responsável por instanciar um objeto Servidor da arquitetura da aplicação no *host* local.

Na próxima seção serão apresentados os comandos interpretados pelo Coordenador.

Comandos interpretados pelo Coordenador

Com o intuito de facilitar a comunicação das aplicações externas com o *SDA-A*, foi criado um conjunto de comandos que possibilita a manipulação das aplicações por ele gerenciadas. Através desses comandos é possível carregar uma aplicação a partir de um arquivo que contenha a descrição de sua arquitetura, na linguagem CBabel. Além disso, também é possível adicionar, remover, atualizar, iniciar ou interromper a execução de uma instância de um componente (*módulo* ou *conector)*[1]. No contexto deste trabalho, uma instância de um componente é um objeto criado a partir de uma classe Java que representa um componente da arquitetura.

O conjunto de comandos predefinidos é:

Tabela 3 - Comandos interpretados pelo Coordenador

Define <Tipo da Tecnologia>	Define a tecnologia a ser utilizada no gerenciamento da aplicação. As tecnologias suportadas pelo *SDA-A* são: (a) "J2SE", que representa a especificação padrão do Java; (b) "JMX", que representa o padrão *Java Management*

[1] Durante o resto deste texto um componente representa um *módulo* ou *conector* de uma arquitetura.

	Extension do Java 1.5.0; (c) “JBoss”, que representa o gerenciamento de elementos do servidor de aplicação JBoss ou aplicações hospedadas no mesmo.
Load <nome do arquivo .adl>	Implanta a arquitetura de uma aplicação descrita em CBabel a partir de um arquivo *“.adl”*. Este arquivo contém todas as especificações da arquitetura, como visto na Seção 3.2.
New <nome do arquivo .adl>	Carrega novos *componentes* que não foram previstos em tempo de projeto. Os *componentes* carregados são adicionados à arquitetura em execução no *SDA-A.*
Start	Inicia a aplicação carregada pelo *SDA-A.* Através deste comando o Coordenador inicia todos os *componentes* da arquitetura de forma que a aplicação entre em execução.
Start <nome do componente>	Inicia um determinado *componente* separadamente. Este comando é comumente utilizado quando uma aplicação deseja substituir ou adicionar um *componente* na arquitetura. Através dele é possível que somente o *componente* adicionado ou atualizado seja iniciado, sem interferir na execução da aplicação.
Stop	Interrompe a execução da aplicação. Quando este comando é executado, a execução de toda a aplicação é interrompida.
Stop <nome do componente>	Interrompe apenas um determinado *componente*. Este comando é comumente utilizado quando se deseja remover ou atualizar um determinado *componente* da aplicação.
Unload	Descarrega a aplicação do *SDA-A.* Todos os *componentes* são descarregados da memória de todos os *hosts* envolvidos na aplicação.
Unload <nome componente>	Descarrega um *componente*. Este comando é utilizado quando se deseja remover ou atualizar um determinado

	componente da aplicação.
Block <nome componente>	Bloqueia a execução de um determinado *componente*. Este comando é utilizado durante todas as modificações arquiteturais com o intuito de garantir a integridade dos fluxos que trafegam na aplicação. Este comando será melhor compreendido nas próximas seções.
Resume <nome componente>	Libera a execução de um determinado *componente*. Este comando é utilizado para neutralizar a operação realizada pelo comando *"Block"*.
Instantiate <nome componente> **at** <nome host>	Instancia um *componente* em um determinado *host*, passado como parâmetro. Este comando é automaticamente executado quando se carrega uma aplicação a partir de um arquivo *".adl"* que contenha o comando "*instantiate"* em seu conteúdo.
Link <nome componente>.<nome porta saída> **to** <nome componente>.<nome porta entrada>	Efetiva a conexão de dois *componentes* em suas respectivas portas de saída e de entrada. Da mesma forma que o comando "i*nstantiate"*, este comando é automaticamente executado quando se carrega uma aplicação a partir de um determinado arquivo *".adl"* que contenha o comando "*link"* em seu conteúdo.
Link <nome do componente> **to** <nome do componente> **by** <lista de conectores>	Este comando é uma variação do comando anterior com apenas a modificação de especificar quais os conectores que intermediarão a comunicação entre os dois *componentes*. Neste caso, é passado como parâmetro uma lista de conectores, separados por vírgula, e sua respectiva ordem será obedecida quando a arquitetura for implantada. O processo de determinação de quais portas de entrada e saída serão utilizadas é determinado através do casamento de tipos das portas. Por exemplo, uma porta "*out port Prover"* irá casar com uma porta do tipo "*in port Prover"*.

Unlink <nome do componente>.<nome da porta>	Este comando desfaz a ligação entre os *componentes* da aplicação. Este comando é comumente utilizado quando se deseja adicionar, remover ou atualizar um determinado *componente* da aplicação.

Todos esses comandos são interpretados pelo Coordenador e mapeados para seus métodos internos. Desta forma, o Coordenador pode acionar os respectivos métodos nos Executores locais ou remotos. Este processo de mapeamento será detalhado nas próximas seções.

3.4.2 Executor

O *Executor* tem a responsabilidade de efetivar localmente as operações designadas pelo Coordenador, como:

- Carregar ou descarregar um *componente;*
- Ligar ou desligar uma porta de um determinado *componente* a outro;
- Interromper ou iniciar a execução de um *componente;*
- Bloquear ou liberar a execução de um determinado *componente.*

Em cada *host*, integrante do ambiente computacional no qual os *componentes* de uma aplicação são executados, deve existir uma instância do *Executor*. O *Executor* possui os seguintes métodos:

Tabela 4 - Métodos do Executor

Boolean Load (String className, String name)	Carrega (do disco local) uma determinada classe ("*className*") cujo nome é passado como parâmetro e instancia um objeto da mesma, registrando-o em uma lista de objetos com o nome que também é passado como parâmetro ("*name*"). Os objetos são armazenados em uma lista de objetos ativos para que eles possam ser manipulados no decorrer da execução da aplicação. Os objetos instanciados representam os *componentes* da aplicação.
Boolean Unload (String name)	Descarrega um determinado objeto cujo nome é passado

	como parâmetro (*componente* da aplicação), ou seja, remove o objeto da lista de objetos manipulada pelo suporte e o libera para que o *Garbage Collector*, da Máquina Virtual Java (JVM), possa coletá-lo.
Boolean Block (String name)	Bloqueia a execução de um determinado objeto com o intuito de garantir que nenhuma mensagem possa passar pelo objeto em questão. Através deste método (e do método *Resume*) é possível impedir que mensagens sejam perdidas durante adaptações em uma aplicação. Entretanto, o controle de estados e fluxo de mensagens em trânsito durante a adaptação de um componente são pontos relacionados aos trabalhos futuros.
Boolean Resume (String name)	Libera a execução de um determinado objeto.
Boolean Link (String nameLocal, String portLocal, String nameTarget, String portTarget, String ipTarget)	Liga a porta de um objeto local, existente na lista de objetos gerenciados pelo *Executor*, a um outro objeto que representa um *componente* local ou remoto. Este processo de ligação de objetos é, de fato, a obtenção de referências remotas ou locais para outros objetos (*componentes*). Para se obter uma referência de um objeto local ou remoto, o *Executor* verifica se o endereço IP do objeto destino (ipTarget) é igual ao seu endereço IP. Caso seja positivo, isto significa que o objeto destino está presente em sua lista de objetos gerenciáveis. Então, é realizada uma busca nessa lista para obter a referência do objeto destino. Caso o endereço IP destino seja diferente, o *Executor* obtém a referência do objeto remoto através dos métodos do *Remote Method Invocation* (RMI). Essa diferenciação na obtenção das referências dos objetos implica diretamente no desempenho da aplicação, pois, caso dois objetos não estejam localizados no mesmo *host*, todo o processo de comunicação entre eles passará pela pilha de protocolos RMI. No entanto, quando dois objetos estão localizados

	no mesmo *host* todo o processo ocorre diretamente através de endereçamentos de memória, ou seja, o processo não passa pela pilha de protocolos RMI, tornando o desempenho relativamente maior (ver Capítulo 4).
Boolean Unlink (String name, String port)	Desfaz a ligação de uma porta de um objeto (componente da aplicação) cujo nome é passado como parâmetro (nome do objeto e nome da porta). Este processo remove todas as informações referentes ao destino das mensagens que passam pelo objeto em questão, ou seja, remove todas as informações para onde as mensagens devem ser encaminhadas desligando assim a porta cujo nome é passada como parâmetro. Mensagens encaminhadas durante a execução da aplicação para portas desligadas são retornadas como exceções.

Todos esses métodos apresentados são utilizados pelo Coordenador para gerenciar um *componente* de uma aplicação remotamente.

Um ponto importante a ser considerado em relação ao Cooredenador e os Executores é a possibilidade de integrá-los em uma única entidade. Essa integração poderia ser realizada para facilitar a utilização do suporte proposto como um serviço distribuído onde uma única entidade exerceria as duas responsabilidades. Neste caso (no nível de implementação) os Executores ofereceriam as funcionalidades do Coordenador. Além disso, eles se comunicariam para oferecer a capacidade de adaptação dinâmica para as aplicações. Embora não tenha sido implementada a integração, em testes realizados durante o desenvolvimento do *SDA-A* foi possível identificar tal possibilidade.

3.5 Classes Padronizadas do Suporte

Para oferecer os recursos de interceptação e adaptação da arquitetura de uma aplicação (descritos na Seção 3.3), foi desenvolvido um conjunto de classes padronizadas que possuem alguns métodos que possibilitam tais recursos. Dentre esses métodos existem alguns que podem ser definidos pelo usuário para que sua aplicação possa ter um comportamento específico quando os mesmos são utilizados pelo suporte (ver Código 4).

Com o intuito de esclarecer em maiores detalhes o funcionamento do *SDA-A*, as principais classes que constituem a sua implementação são apresentadas. São elas:

PackageMessage → Esta classe foi desenvolvida para encapsular os parâmetros das mensagens que trafegam entre os componentes da aplicação. Ela contém o nome do método que será invocado no componente destino (porta de entrada) e algumas outras informações referentes aos parâmetros do método a ser invocado. Através da manipulação de um pacote de mensagem, é possível mudar o método a ser invocado no componente destino. Além disso, também é possível alterar os valores dos parâmetros.

ProxyConfigurator → Esta classe representa um *Proxy* dinâmico [Sun 2005] responsável por resolver a localização do método a ser invocado (seja ele local ou remoto) e obter todas as referências para que a invocação do mesmo possa ser realizada. Esta classe é utilizada quando um *módulo* da aplicação invoca um serviço em um outro *módulo* ou *conector* (local ou remoto). Na Figura 10 (Seção 3.7) é apresentado um diagrama de seqüência que representa o fluxo de execução de uma aplicação Cliente/Servidor que utiliza essa classe. Nesta aplicação, quando o Cliente solicita um serviço ao Servidor o *ProxyConfigurator* intercepta a chamada, redirecionando-a para os métodos que são responsáveis pelo encaminhamento das mensagens até o seu destino, como apresentado na Seção 3.3.

Base → Esta classe contém todos os métodos padronizados que o suporte utiliza para adaptar a arquitetura da aplicação. Ela também oferece os métodos que permitem a interceptação das invocações de serviços solicitados pelos *componentes* da configuração. Os principais métodos dessa classe são:

Boolean _defineLink_ASAC_ (String nameTarget, String port, String ipTarget) → Este método possui três parâmetros. O primeiro é o nome do *componente* para onde as mensagens devem ser encaminhadas (*componente* destino). O segundo é o nome da porta de saída a ser utilizada no *componente* corrente (*componente* origem) e o terceiro é o endereço IP do componente destino. Através destas informações, é possível obter as referências para o *componente* destino, seja ele local ou remoto. Todas as referências para o *componente* destino são armazenadas em uma estrutura de dados interna ao *componente* origem. Esta estrutura de dados é inserida pelo *Executor* do *SDA-A* quando ele realiza as adaptações das classes do usuário. Pode-se notar que não é informado o nome da porta de entrada a ser utilizada no *componente* destino. Esta informação não é necessária, pois, no nível de implementação a porta de entrada de um *componente* é mapeada para uma interface de serviço. Desta forma, o

suporte utiliza os mecanismos de reflexão estrutural da linguagem Java para executar o serviço desejado, o qual está especificado no pacote de mensagem.

Boolean undefineLink_ASAC_ (String port) → Este método desliga uma porta de um *componente* cujo nome é passado como parâmetro, ou seja, remove uma ligação entre dois componentes. Ele remove todas as referências para o *componente* destino e pode ser definido pelo programador com o intuito de possuir um comportamento específico quando for executado.

*Void _block_ASAC_* → Bloqueia o fluxo de execução do *componente*.

*Void _resume_ASAC_* → Libera o fluxo de execução do *componente*.

Void _init_ASAC_ (String args[])→ Inicializa o *componente.* Este método pode ser definido pelo programador caso ele deseje que o *componente* execute algum processamento antes de ser utilizado pelos demais *componentes* da aplicação. Esse método possui um parâmetro que poderá conter os parâmetros que forem definidos na descrição arquitetural da aplicação.

*Void _stop_ASAC_* → Interrompe a execução do *componente* corrente.

PackageMessage _handle_ASAC_ (PackageMessage message) → Este método pode ser definido pelo programador para determinar quais ações o *componente* deve tomar quando uma chamada de serviço passar por ele. De acordo com o *design pattern Architecture Configurator* (Seção 3.3), todas as chamadas que passam por um determinado *componente*, executam primeiramente este método antes de seguir o seu curso normal de execução. Ele invoca automaticamente o método *_foward_ASAC_ (PackageMessage message)*, caso ele não seja definido pelo usuário. Como todas as chamadas que passam por um *componente* executam este método, é através dele que se pode realizar as interceptações das chamadas de serviço.

PackageMessage _foward_ASAC_ (PackageMessage message) → Este método é responsável pelo encaminhamento das chamadas ao seu destino. Caso o destino da chamada seja o próprio *componente*, então, ele invoca o método determinado no pacote de mensagem que é passado como parâmetro. Caso esse método seja definido pelo programador, o Executor, durante o processo de adaptação das classes do programador, irá substituir o método definido pelo usuário pelo método original da classe padronizada. Essa substituição ocorre devido à lógica de encaminhamento de mensagens entre os componentes da aplicação contida nesse método. Tal lógica garante o comportamento de encaminhamento de mensagens

definido pelo *design pattern Architecture Configurator*. Além deste, os métodos *_block_ASAC_* e *_resume_ASAC_* também serão substituídos caso sejam definidos pelo programador.

```
01 import configurator.Comm.PackageMessage;
02 import java.util.StringTokenizer;
03 public class Filtro {
04  /** Redefine o método handle */
05  public Object _forward_ASAC(PackageMessage obj){return obj;}
06  public Object _handle_ASAC(PackageMessage obj){
07    String user = (String) obj.getArgs()[0];
08    StringTokenizer resultado;
09    PackageMessage rObj;
10    rObj = _forward_ASAC(obj);
11    //Manipula os dados do resultado
12    if (user.equalsIgnoreCase("usuario1")){
13      resultado = new StringTokenizer((String)rObj.getResult(),",");
14      String valor = (String)resultado.nextElement();
15      valor = valor + (String)resultado.nextElement();
16      rObj.setResult(valor);
17    }
18    return rObj;
19  }
20}
```

Código 4 - Definição dos métodos _handle_ASAC e _foward_ASAC

O Código 4 apresenta um exemplo de um *conector* desenvolvido que redefine o método *_handle_ASAC_* (linha 6 – 19) e o método *_foward_ASAC_* (linha 5). De acordo com o código, o programador definiu o método *_foward_ASAC_* com a intenção de evitar erros durante a compilação do código. Já a definição do método *_handle_ASAC_* serve para realizar uma interceptação e tratamento das mensagens que passam pelo *conector*. Com essa definição o usuário fica encarregado de fazer a chamada para o método *_foward_ASAC_* (linha 10), com o intuito de garantir o encaminhamento das mensagens até o seu destino. Na próxima seção será apresentado o fluxo de execução de uma aplicação Cliente/Servidor que também evidencia este processo.

3.6 Modificação do Código da Aplicação do Programador

Aplicações distribuídas que serão gerenciadas pelo *SDA-A* são modificadas para oferecerem a capacidade de adaptação dinâmica e atenderem o padrão JMX. O processo de modificação das aplicações consiste na junção dos métodos padronizados da classe *Base* com os métodos das classes desenvolvidas pelo programador. Além disso, também é verificado se as classes do programador seguem o padrão JMX. Caso elas não sigam o padrão, o suporte as

modifica criando as interfaces necessárias para atender o padrão JMX. A Figura 8 ilustra o processo de modificação das classes do programador.

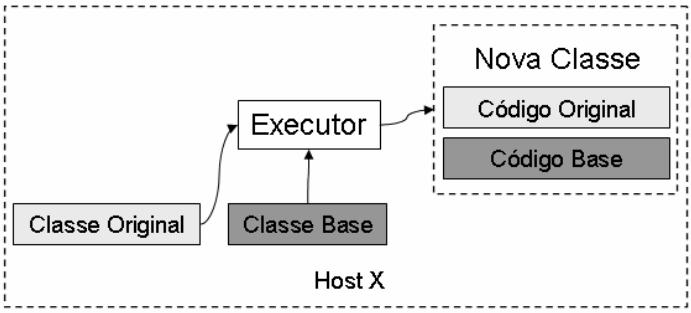

Figura 8 - Processo de modificação das classes do programador

Como se pode observar, o *Executor* localizado no *Host X* acessa as informações da classe do programador, denominada *Classe Original*, que representa um *componente* da aplicação. O mesmo *Executor* também acessa a classe padronizada do suporte, denominada *Classe Base*, que contém todos os métodos necessários para que um *componente* possa ser gerenciado pelo suporte. Obtidas todas as informações, o *Executor* cria uma nova classe, denominada *Nova Classe*, constituída pela junção da classe do programador com a classe padronizada do suporte (*Código Base*). Em seguida, ele instancia um objeto dessa nova classe e o registra no *SDA-A*. O Código 5 apresenta o conteúdo de uma classe desenvolvida pelo programador após a modificação realizada pelo *Executor*. Todo o código mostrado em negrito e itálico foi inserido automaticamente pelo *Executor*. É importante lembrar que o código abaixo é apenas ilustrativo. O mesmo é gerado em tempo de execução e as modificações são realizadas em nível de *bytecode*, ou seja, as modificações são realizadas nos arquivos de código intermediário Java.

```
01 public class Conector implements ConectorMBean{
02 import configurator.Comm.PackageMessage;
03  //Metodos do Conector definidos pelo usuário
04  public Conector(){...}
05  public String metodo1(){...}
06  public String metodo2(){...}
07  public String metodo3(){...}
08
09  //Metodo definido que faz o log!
10  public PackageMessage handle(PackageMessage obj){...}
11
12  //Métodos da Base adicionados pelo suporte em tempo de execução
13  public PackageMessage forward(PackageMessage obj){};
14  public void _init_ASAC(String[] args){...}
15  public void _stop_ASAC(){...}
16  public void _resume_ASAC(){...}
17  public void _block_ASAC(){...}
18  public boolean _defineLink_ASAC(
19          String nameTarget,
```

```
20          String portTarget,
21          String ipTarget){...}
22  public PackageMessage _forward_ASAC(PackageMessage obj){...}
23  public boolean _undefineLink_ASAC(String port){...}
24}
```

Código 5 - Código de uma classe adaptada pelo *SDA-A*

Um ponto importante a ser mencionado em relação à modificação das classes do programador é que o *Executor* possui um mecanismo interno que verifica se uma classe já foi modificada anteriormente. Caso ela tenha sido modificada, o *Executor* verifica se os métodos dessa classe foram alterados (reescritos pelo programador). Caso eles não tenham sofrido nenhuma alteração, o Executor não realiza o processo de modificação para a classe já modificada. Caso contrário, o *Executor* realiza o processo de modificação normalmente. Para cada classe modificadas pelo *Executor* é criada uma Nova-Classe (no disco local) em uma pasta do suporte cujo nome é "*New_File_ASAC*". É a partir das classes contidas nessa pasta que o *Executor* verifica se existe diferença entre as classes a serem modificadas e as classes já modificadas anteriormente.

Maiores detalhes de implementação de uma aplicação que utiliza o suporte de adaptação oferecido pelo *SDA-A* pode ser encontrado no Apêndice A desta dissertação. Maiores informações sobre a utilização do Javassist no processo de modificação das classes de aplicações consulte o Apêndice B desta dissertação.

3.7 Processo de Configuração de uma Arquitetura

Para implantar uma arquitetura de uma aplicação, é necessário instanciar em cada *host*, pertencente ao conjunto de *hosts* que executará a aplicação, uma instância do *Executor*. Após instanciado o conjunto de *Executores*, é necessário instanciar o *Coordenador* em um *host*, que não necessariamente precisa fazer parte do conjunto de *hosts* da aplicação. A Figura 9 apresenta o processo de implantação de uma arquitetura Cliente/Servidor.

No primeiro momento da implantação o Coordenador define qual tecnologia será utilizada no gerenciamento da aplicação (passo 1), no caso a JMX. No segundo momento ele obtém as informações da arquitetura da aplicação a partir do arquivo "<*nome*>.*adl*" descrito em CBabel (passo 2). A seguir, todas as informações contidas neste arquivo são convertidas para uma estrutura de dados interna de forma a facilitar o gerenciamento da configuração da aplicação. Essas informações podem ser repassadas para as aplicações que desejam obter as informações referentes à configuração da arquitetura da aplicação. Numa terceira etapa, o

Coordenador comunica-se com cada um dos Executores envolvidos no gerenciamento da aplicação e solicita a cada um deles que carregue a classe do programador, cujo nome é passado como parâmetro nas chamadas dos métodos *loader* (passos 3 a 8). Neste momento, cada *Executor* realiza a modificação das classes do programador, como descrito na Seção 3.6. Por fim, o Coordenador solicita aos Executores que definam as ligações entre cada um dos componentes da aplicação invocando a primitiva "*link*", passando como parâmetro o nome do componente alvo e o do componente destino (passos 9 e 10). Os passos 11, 12 e 13 representam o encerramento da comunicação do Coordenador com os Executores envolvidos na configuração da aplicação.

Após o processo de implantação, a aplicação está pronta para ser iniciada. Contudo, a seqüência de inicialização de seus componentes fica a cargo do programador. Ele poderá utilizar o comando "*start"*(passando-o como parâmetro para a interface *apply* do Coordenador) para que a aplicação seja colocada em execução. Caso seja necessário usar uma seqüência lógica para a inicialização dos componentes, o programador poderá utilizar o comando "*start <nome do componente>"* através da interface "*apply*" do Coordenador. Esse procedimento pode facilitar a configuração de aplicações que incluam componentes que devam ser inicializados individualmente, por exemplo, para evitar situações de bloqueio.

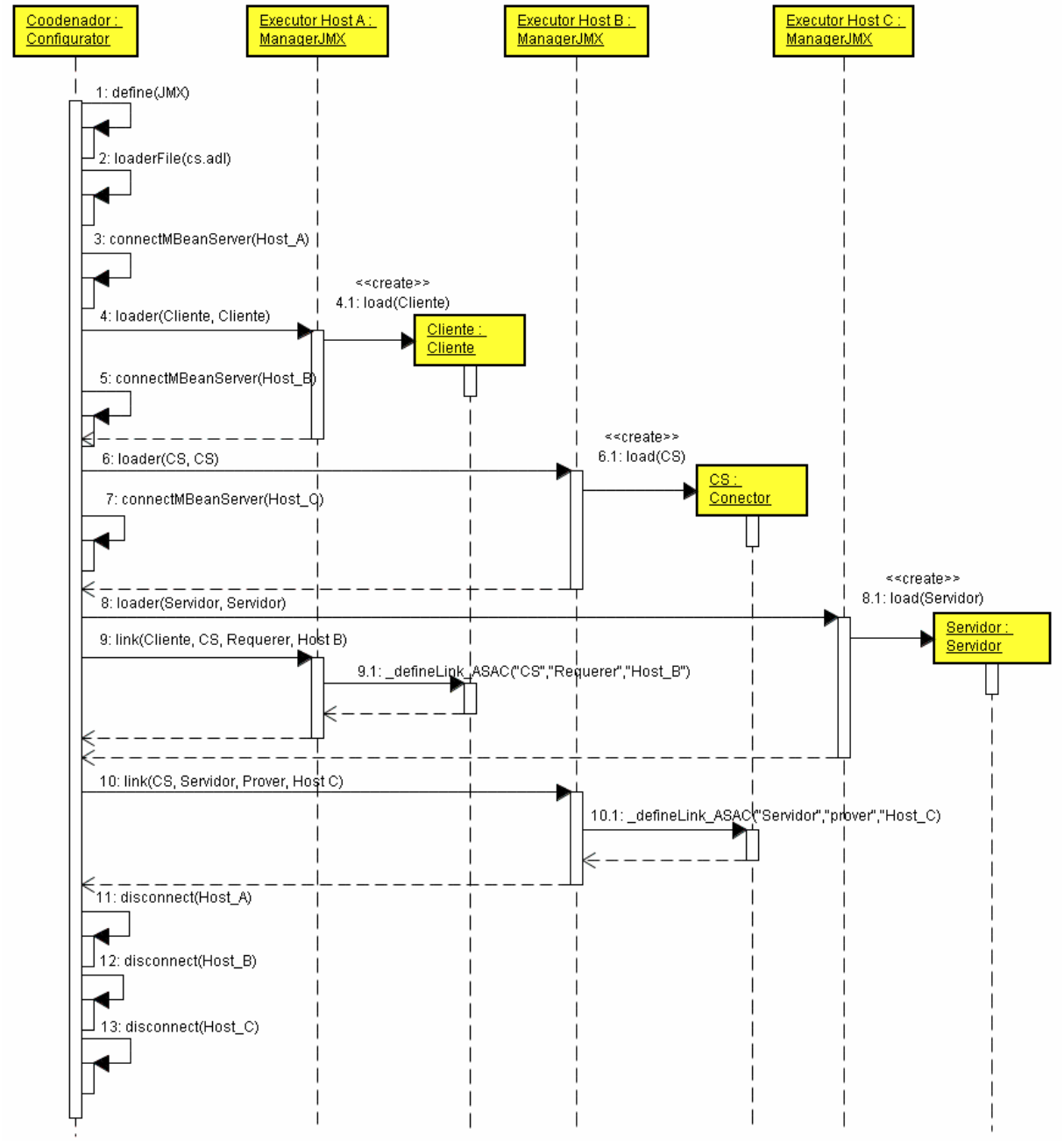

Figura 9 - Processo de implantação de uma arquitetura Cliente/Servidor.

A Figura 10 apresenta o fluxo de execução das chamadas de serviço entre um Cliente e Servidor (fluxo de execução típico). Inicialmente, quando um módulo Cliente realiza uma chamada de serviço para o módulo Servidor ele desconhece a existência do conector CS. Isso ocorre devido a transparência oferecida pelo *ProxyConfigurator* que resolve a localização do *conector*. Para o Cliente, a chamada é realizada diretamente para o Servidor, tornando possível assim adicionar ou remover conectores entre eles de forma transparente.

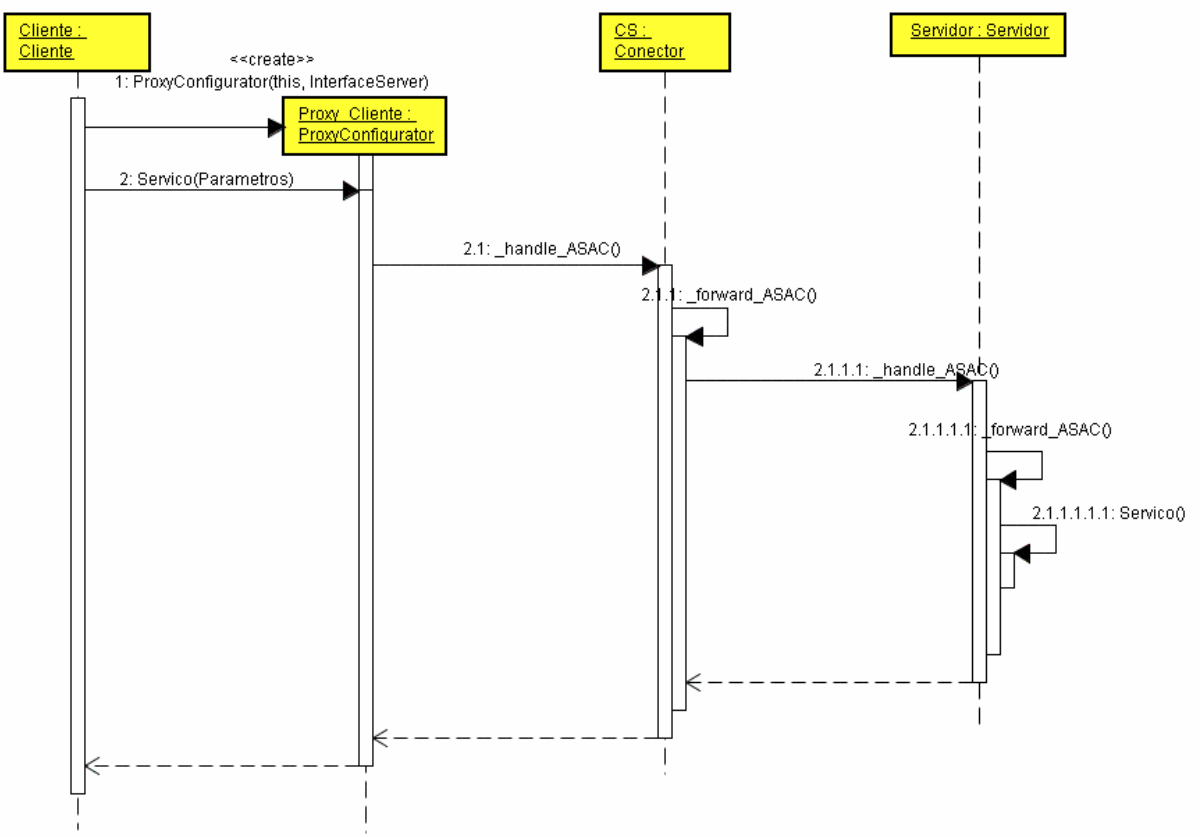

Figura 10 - Fluxo de execução da aplicação.

O primeiro passo do fluxo de execução é a criação do *proxyConfigurator* pelo cliente (passo 1). O processo de criação do *proxyConfigurator* é transparente para o cliente, pois o Executor, durante a adaptação das classes do usuário, identifica as chamadas de serviços externas (através da descrição arquitetural da aplicação) e insere um trecho de código que será executado quando uma chamada de serviço for realizada pela primeira vez, de modo a criar o *proxyConfigurator*. A partir de sua criação todas as chamadas realizadas pelo Cliente passarão pelo *proxyConfigurator* (passo 2), que por sua vez invoca o método *_handle_ASAC* do *conector* CS, que possui o código definido pelo desenvolvedor (passo 2.1). O método *_handle_ASAC* do *conector* CS invoca seu próprio método *_foward_ASAC,* que é responsável por encaminhar a chamada para o próximo *componente* da aplicação, neste caso o Servidor (passo 2.1.1). Então, o método *_foward_ASAC* do *conector* invoca o método *_handle_ASAC*

do *módulo* Servidor (passo 2.1.1.1). Como geralmente os desenvolvedores de *módulos* estão concentrados nos requisitos funcionais da aplicação, eles podem não definir o método *_handle_ASAC*. Desta forma, o método *_handle_ASAC* padrão invoca o método *_foward_ASAC* (passo 2.1.1.1.1). Quando o método *_foward_ASAC* do Servidor é invocado ele identifica que a chamada de serviço destina-se a si mesmo, e automaticamente executa o método da aplicação que é identificado por um parâmetro no pacote da mensagem. Logo após a execução do método, a mensagem de retorno contendo seus resultados é emitida (passo 2.1.1.1.1.1). Neste exemplo pode ser notado que o fluxo de execução é semelhante ao apresentado no *design pattern Architecture Configurator* (Seção 3.3) e que o *conector* CS pode interceptar e tratar todas as chamadas que passam por ele, apenas definindo o método *_handle_ASAC*.

3.8 Gerenciamento J2SE x Gerenciamento JMX

Como mencionado nas seções anteriores, o *SDA-A* oferece dois suportes de gerenciamento baseado nos padrões de desenvolvimento de aplicações Java J2SE e JMX [Sun 2005] respectivamente. Existe uma diferença entre esses dois suportes de gerenciamento; o primeiro não faz uso de nenhum mecanismo de gerenciamento dinâmico de objetos enquanto o segundo explora os recursos de gerenciamento dinâmico oferecido pelo padrão JMX.

O suporte de gerenciamento J2SE utiliza apenas os mecanismos disponibilizados pelo Java. Neste caso, ele restringe-se à utilização do RMI (*Remote Method Invocation*). Contudo, o RMI não oferece mecanismos de gerenciamento dinâmico de objetos. Desta forma, foram desenvolvidos mecanismos que possibilitam o gerenciamento dinâmico de objetos (remotos ou locais) utilizando apenas o RMI. No caso do suporte de gerenciamento JMX, ele explora os recursos de gerenciamento dinâmico disponibilizados pelo padrão JMX.

Através do suporte de gerenciamento JMX aplicações desenvolvidas utilizando o padrão J2SE podem ser gerenciadas através dos recursos de gerenciamento do JMX. O principal benefício do gerenciamento JMX, oferecido pelo *SDA-A*, é possibilitar que aplicações desenvolvidas usando o padrão J2SE ofereçam seus serviços a outras aplicações da mesma maneira que aplicações desenvolvidas utilizando o padrão JMX. Para tornar possível essa capacidade o *SDA-A* cria uma versão modificada da aplicação (ou componente da aplicação) seguindo o padrão de desenvolvimento JMX (ver seção 3.6 deste capítulo e o Capítulo 2, Seção 2.2).

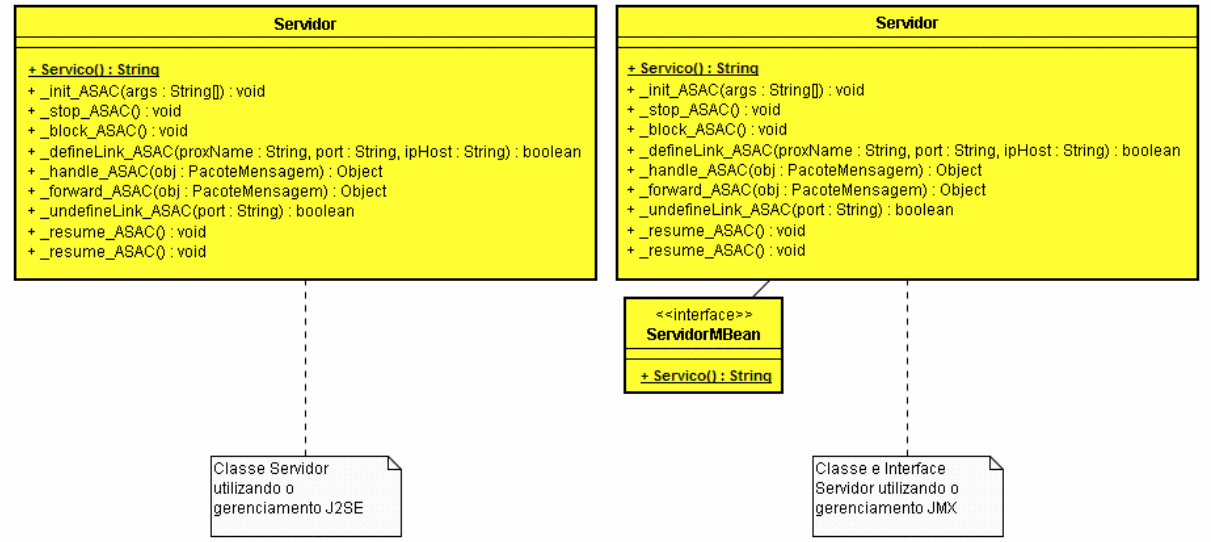

Figura 11 - Classes Modificadas: Gerenciamento J2SE x Gerenciamento JMX

A Figura 11 apresenta as classes modificadas pelos gerenciamentos J2SE e JMX oferecidos pelo *SDA-A*. De acordo com a Figura, a classe "`Servidor`" à esquerda representa a classe do programador após o processo de modificação utilizando o suporte de gerenciamento J2SE. Já para o processo de modificação utilizando o suporte de gerenciamento JMX é criada uma nova interface denominada "`ServidorMBean`" para atender o padrão JMX (ver Capítulo 2, Seção 2.2). Através desta interface aplicações externas podem utilizar o `Servico()` da classe "`Servidor`". Além do método `Servico()` criado pelo programador, são introduzidos os demais métodos padronizados pelo suporte utilizando o Javassist. Os métodos do Javassist utilizados pelo *SDA-A* para realizar a modificação das classes de aplicações desenvolvidas por programadores são:

- `makeInterface` utilizado para criar uma nova interface MBean (no caso do gerenciamento JMX);
- `addMethod` utilizado para adicionar os métodos padronizados do *SDA-A*;
- `addInterface` utilizado para adicionar uma interface a uma classe (no caso do gerenciamento JMX);
- `addField` utilizado para inserir os atributos da estrutura de dados interna definida pelo *SDA-A* a qual armazenará as informações sobre a localização dos elementos arquiteturais da aplicação;

- `removeMethod` utilizado para remover os métodos padronizados do suporte definidos por engano pelos programadores (utilizado no caso do método `_foward_ASAC()` ter sido definido pelo programador);
- `insertBefore` utilizado para inserir o código de criação dos `ProxyConfigurator` definido pelo suporte;

Maiores informações sobre a utilização do Javassist no processo de modificação das classes de aplicações consulte o Apêndice B deste documento.

Além dos suportes de gerenciamento J2SE e JMX o *SDA-A* também oferece um suporte de gerenciamento destinado ao servidor de aplicações JBoss que será abordado na próxima seção.

3.9 Gerenciamento no Servidor de Aplicação JBoss

Durante os estudos e o desenvolvimento do *SDA-A* foi observada a possibilidade de um gerenciamento voltado para servidor de aplicações JBoss, pois, a tecnologia base dos serviços e aplicações hospedadas nele é o JMX. Então, como o *SDA-A* já fornece um gerenciamento voltado para o JMX, foi disponibilizado um gerenciamento, através de primitivas definidas em um alto nível de abstração, para as aplicações hospedadas no JBoss e para os seus módulos de serviço (módulos do JBoss).

Como para as outras opções, ao iniciar um gerenciamento de uma aplicação, pode-se definir a tecnologia a ser utilizada sendo o servidor de aplicação JBoss (comando *define "JBoss"*). A partir deste momento, o *SDA-A* irá se comunicar com o Executor hospedado nas instalações do JBoss para realizar as adaptações das aplicações ou módulos do próprio JBoss. Neste ponto, temos dois tipos de gerenciamentos possíveis:

- O gerenciamento dos componentes de uma aplicação: este gerenciamento é idêntico aos demais gerenciamentos apresentados anteriormente. A diferença consiste na aplicação a ser gerenciada estar hospedada no JBoss e não ser necessário modificá-la para atender o padrão JMX, já que todas as aplicações hospedadas no JBoss seguem esse padrão. No entanto, ainda são realizadas as modificações referentes à adição dos métodos padronizados requeridos pelo *SDA-A*.

- O gerenciamento dos *módulos* do JBoss: este gerenciamento possibilita ao usuário carregar ou descarregar um determinado *módulo* do JBoss. Através deste gerenciamento uma aplicação externa poderá configurar o JBoss remotamente, através de um alto nível de abstração. A aplicação externa passará um comando para o Coordenador do SDA-A, cuja sintaxe é definida em CBabel, contendo as informações de qual serviço deverá ser inicializado ou descarregado. Mediante essas informações (recebidas através do Coordenador) o Executor se encarregará de realizar as operações de baixo nível, como por exemplo, instanciar o *módulo* de serviço desejado. Para isso, é necessário que o JBoss seja carregado com um núcleo mínimo de serviços e que juntamente com estes serviços o *Executor* esteja disponível.

Para tornar possível os dois tipos de gerenciamento, foi criado um serviço que é executado pelo JBoss em sua inicialização. Este serviço na realidade é um *Executor* do *SDA-A*, que fica hospedado nas instalações do JBoss como um componente de serviço (ver Capítulo 2, Seção 2.3.1) e recebe todas as operações a serem realizadas tanto nos componentes de uma aplicação quanto no servidor JBoss em si.

É importante ressaltar que durante o gerenciamento dos *módulos* do JBoss eles não sofrem nenhum tipo de modificação, já que são apenas carregados ou descarregados no servidor, não sendo necessário modificá-los. O processo de modificação ocorre somente com as aplicações que são hospedadas no JBoss. No próximo capítulo (Seção 4.1.3) será apresentado um exemplo de como é realizada uma configuração dos *módulos* do JBoss e um outro exemplo de como adaptar uma aplicação nele hospedada.

3.10 O SDA-A no Contexto do Projeto do CR-RIO

O *SDA-A* faz parte do projeto do CR-RIO (*Contractual Reflective - Reconfigurable Interconnectable Objects*) [Corradi 2005, Loques et al. 2004, Freitas 2005]. O CR-RIO é um *framework* que permite o desenvolvimento de sistemas de software a partir de uma visão arquitetural, como uma composição de componentes e interligações descritas explicitamente e separadamente, em oposição ao desenvolvimento em baixo nível, feito a partir da especificação de algoritmos e estruturas de dados dos componentes individuais. Ele inclui mecanismos para a especificação e suporte a contratos de Qualidade de Serviço (QoS) associados aos componentes das arquiteturas de software das aplicações.

A arquitetura do CR-RIO (Figura 12) é composta por um conjunto de elementos principais que são: o Gerenciador de Contratos (***Contract Manager***, CM), **Contractors** e **Agentes de QoS** (*QoSAgents).* O **CM** interpreta a descrição de um contrato e extrai desta a máquina de estados de negociação de serviços. Inicialmente, o **CM** identifica qual serviço será negociado e envia as descrições de configuração, e os perfis associados, aos ***Contractors*** residentes em cada *host* participante. Cada ***Contractor*** é responsável por interpretar a configuração local recebida e efetuar ações, tais como reserva de recursos e solicitações de monitoramento de recursos requeridos. O **Contractor** tem duas responsabilidades principais: (a) traduzir as propriedades dos perfis associados aos serviços do nível de arquitetura em serviços de suporte do nível de sistema, e requisitar estes serviços, com os parâmetros adequados, aos Agentes de QoS e (b) receber notificações dos Agentes de QoS. Um **Agente de QoS** (*QoSAgent*) encapsula o acesso aos mecanismos de nível de sistema, provendo interfaces para efetivar a alocação de recursos, iniciar serviços locais de sistema e monitorar os valores de propriedades requeridas. O **CM** também é responsável por solicitar ao configurador de arquiteturas (neste caso o *SDA-A*) que adapte as aplicações mediantes as configurações passadas como parâmetros. Maiores detalhes sobre o CR-RIO podem ser encontrados em [Corradi 2005].

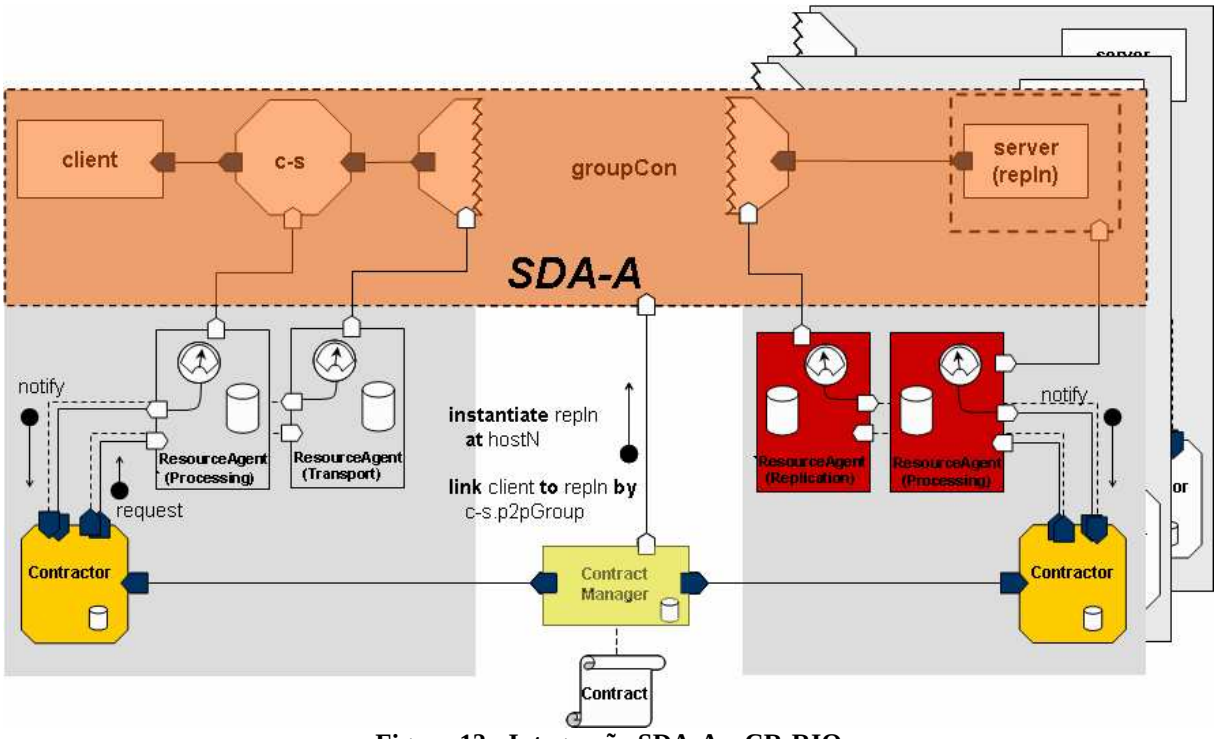

Figura 12 - Integração SDA-A e CR-RIO

No contexto do projeto do CR-RIO, o *SDA-A* desempenha o papel do configurador de arquiteturas, sendo assim, responsável por efetivar, em baixo nível, mudanças na arquitetura

de uma aplicação. A Figura 12 ilustra o *SDA-A* recebendo solicitações do **CM** para efetivar as adaptações da aplicação. Apesar do *SDA-A* fazer parte do projeto do CR-RIO, ele também pode ser utilizado de forma independente como será apresentado no Capítulo 4, Seção 4.1.3 e Seção 4.1.4.

3.11 Conclusão do Capítulo

Neste capítulo foi apresentado o *SDA-A*, um suporte à adaptação dinâmica, baseado em tecnologias atuais amplamente disseminadas em âmbito comercial. Ele atende as necessidades de adaptação dinâmica das aplicações, provendo primitivas de configuração que podem ser usadas independentemente para colocar em execução e adaptar configurações de suas arquiteturas. Como opções para os desenvolvedores, o *SDA-A* oferece três tipos de gerenciamento: um gerenciamento dirigido para aplicações usando o padrão Java *J2SE*, um gerenciamento dirigido para aplicações baseadas no padrão JMX e um gerenciamento voltado para o servidor de aplicação JBoss. Esse último gerenciamento oferece duas opções, uma voltada diretamente para as aplicações hospedadas no JBoss e outra voltada para o gerenciamento dos seus próprios módulos. Com a implementação atual do suporte foi possível demonstrar a possibilidade de gerenciar aplicações no ambiente do JBoss e/ou seus módulos de serviços dele através de primitivas definidas em um alto nível de abstração. No próximo capítulo apresentaremos um exemplo para cada tipo de gerenciamento oferecido pelo *SDA-A*. Além disso, apresentaremos também os resultados de medidas de desempenho do suporte proposto.

Capítulo 4

Exemplos e Avaliação Experimental

Este capítulo apresenta alguns exemplos de aplicações cujos requisitos de adaptação dinâmica são de crucial importância em seu funcionamento. Esses exemplos permitem demonstrar através de casos de uso a utilidade do *SDA-A* no gerenciamento de aplicações. São três os exemplos apresentados: uma aplicação de vídeo sob demanda em ambientes ubíquos (pervasivos), uma aplicação cliente-servidor com servidores replicados e uma aplicação de comércio eletrônico utilizando o servidor de aplicação JBoss.

São apresentados ainda alguns resultados de medidas de desempenho do *SDA-A*, demonstrando que ele atende os requisitos de flexibilidade e desempenho exigidos pelas classes de aplicações apresentadas no Capítulo 1, e.g., aplicações de computação autônoma e aplicações ubíquas.

4.1 Exemplos

Para os dois primeiros exemplos que serão apresentados adotamos o *framework* CR-RIO, que facilita a implementação de arquiteturas de software adaptáveis. Utilizando o *CR-RIO*, uma aplicação pode ter sua funcionalidade básica descrita no nível arquitetural, sendo as preocupações não-funcionais, descritas em tempo de projeto através de contratos de alto nível [Corradi 2005, Freitas et al. 2005, Loques et al. 2004]. Em tempo de execução, os contratos são utilizados para automatizar o gerenciamento das adaptações necessárias para prover os recursos ou os serviços utilizados pela aplicação. Complementando o *framework*, de modo a prover os mecanismos de baixo nível necessários à efetivação das adaptações de aplicações, utilizamos o *SDA-A* para permitir que os componentes e interligações da arquitetura da aplicação sejam reconfigurados de modo a adaptá-la dinamicamente. O CR-RIO utiliza a

interface *apply* oferecida pelo Coordenador para informar ao *SDA-A* quais operações devem ser realizadas. A Figura 13 (Seção 4.1.1) ilustra como o CR-RIO, da mesma forma que uma aplicação externa, é integrado ao *SDA-A*.

Para o terceiro exemplo, que será apresentado utilizamos apenas o *SDA-A* e o servidor de aplicações JBoss.

4.1.1 Vídeo sob demanda (VoD) em ambientes ubíquos (pervasivos)

As aplicações ubíquas possuem vários requisitos peculiares cujas formas de atendimento ainda se encontram sob pesquisa por diversos grupos. O exemplo que será apresentado contempla de maneira simplificada as principais características dessas aplicações.

A aplicação de Vídeo sob Demanda (VoD) ubíqua vislumbra um cenário onde o usuário transita por vários ambientes, sendo que cada ambiente possui características próprias em termos de recursos disponíveis; além disso, os usuários carregam sensores que informam em que ambientes eles estão localizados. Para fins deste exemplo, assumimos que as informações sobre as localizações dos usuários e sobre os recursos existentes nos ambientes são obtidas através de um serviço de localização e descoberta de recursos (SL&D) não enfocados nesta dissertação. Mais detalhes sobre *SL&Ds* relacionados a ambientes ubíquos podem ser obtidos em [Capra et al. 2005, Ranganathan et al. 2005, Román et al. 2002, Cardoso 2006].

A aplicação VoD (Figura 13) conta com uma fonte que disponibiliza vídeos que devem ser reproduzidos em um dos dispositivos de visualização disponíveis na sala em que o usuário se encontra. O usuário pode transitar por três salas nas quais ele pode permanecer para executar tarefas cotidianas, incluindo assistir a um vídeo que é transmitido exclusivamente para ele. Cada sala possui dispositivos de visualização que podem ser usados para reproduzir o vídeo para o usuário, ou podem ser usados por outros usuários para outras finalidades. Para este exemplo, foram considerados quatro tipos de dispositivos de visualização (Projetores, Monitores LCD, TV de Plasma e a tela de um dispositivo portátil tipo Pocket PC). Todos os dispositivos de visualização utilizados possuem um módulo Executor associado, de modo a facilitar as adaptações da aplicação, pois, como apresentado no Capítulo 3, os Executores são os responsáveis pelas adaptações da arquitetura da aplicação.

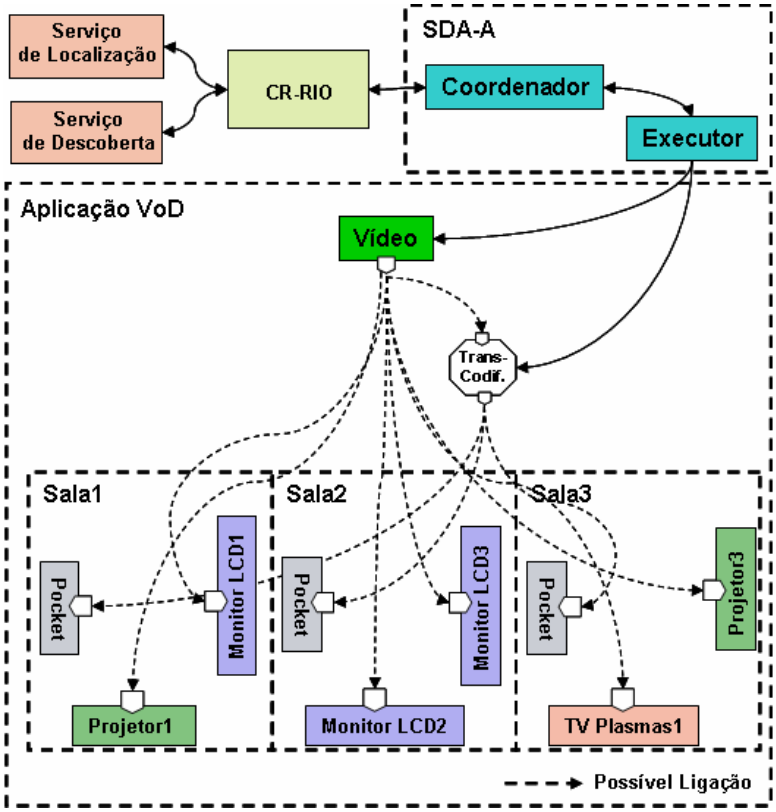

Figura 13 - Aplicação VoD no ambiente de computação ubíqua

Com relação à reprodução do vídeo para o usuário, foi considerada a possibilidade de presença de outros usuários no ambiente, que compartilham esse ambiente para executar suas próprias atividades. Desta forma, por questões de privacidade, conforme opção do usuário e caso existam outros usuários no mesmo ambiente, o vídeo deverá ser reproduzido em um dispositivo pessoal do usuário (tipo *POCKET-PC*). Para que esse tipo de dispositivo possa reproduzir o vídeo transmitido, será utilizado um *conector* (*transcodificador*) responsável por transcodificar o fluxo de vídeo para um formato compatível com o *Pocket-PC* (de MPEG para H261). Neste cenário, o *framework* CR-RIO utiliza o *SL&D* para obter as informações referentes ao ambiente e à localização do usuário. O *SL&D* também fornece as informações de quais dispositivos estão aptos a reproduzir o vídeo do usuário em um determinado ambiente.

Descrição do contrato

Utilizando o suporte a contratos do CR-RIO, uma categoria (Código 6) e dois serviços foram definidos (Código 7) de modo a expressar as necessidades do usuário. O Código 6 apresenta a categoria *Context*, que será utilizada pelos perfis associados aos serviços definidos no contrato. Essa categoria possui quatro propriedades, sendo três de verificação (*pessoas*,

salaAtual, *numeroDisplaysDisponiveis*) e uma de orientação (*displays*). A propriedade *pessoas* indica quantas pessoas há na sala onde o usuário se encontra. Através desta propriedade é possível identificar se o usuário está só ou acompanhado. A propriedade *salaAtual* informa o número da sala em que o usuário se encontra. Mediante essa informação o CR-RIO pode solicitar ao *SL&D* a lista de dispositivos disponíveis na sala atual do usuário. A propriedade *numeroDisplaysDisponiveis* informa a quantidade de dispositivos, independente do tipo, que estão à disposição do usuário na sala onde ele se encontra. Caso o número de dispositivos seja igual a zero o serviço não poderá ser estabelecido. E por fim, a propriedade *displays* orienta no processo de adaptação informando a preferência do usuário pelos dispositivos de reprodução do vídeo.

O Código 7 apresenta os serviços oferecidos ao usuário. O primeiro serviço (*sSalaAcompanhado* linhas 05-10) destina-se a uma situação onde o usuário está acompanhado por outras pessoas na sala. Este serviço é considerado secundário devido à preferência do usuário pela reprodução do vídeo em dispositivos de maior qualidade, como a TV de Plasma, o Projetor e o Monitor. Este serviço visa atender a um requisito de privacidade do usuário, reproduzindo o vídeo na tela de seu *POCKET-PC* pessoal. No segundo serviço (*sSalaSozinho* linhas 02-04), que é considerado preferencial, o usuário está só na sala e então o vídeo poderá ser reproduzido no melhor dispositivo de visualização disponível, segundo uma ordem de preferência previamente especificada; e.g., em primeiro lugar a TV de Plasma, seguido pelo Projetor e por último o Monitor LCD.

```
01 QosCategory Context {
02 pessoas: numeric increasing in;
03 salaAtual: numeric in;
04 numeroDisplaysDisponiveis: numeric increasing in;
05 displays: enum(TVPlasma, Projetor, MonitorLCD, Pocket) out;}
```

Código 6 - Categoria context

```
01 contract {
02 service {
03   link video to display with ExisteUsuario, Sozinho, ExisteDispositivo;
04 } sSalaSozinho;
05 service {
06   instantiate TransCoder with ExisteUsuario,
07      Acompanhado, ExisteDispositivo;
08   link video to display by TransCoder with ExisteUsuario,
09      Acompanhado, ExisteDispositivo;
10 } sSalaAcompanhado;
11 negotiation {
12   sSalaSozinho -> sSalaAcompanhado, out of service;
13   sSalaAcompanhado -> sSalaSozinho, out of service;
14 }
15 profile {
16   Context.pessoas >=1;
17 } ExisteUsuario;
```

```
18 profile {
19   Context.numeroDisplaysDisponiveis > 0;
20 } ExisteDispositivos;
21 profile {
22   Context.pessoas = 1;
23   Context.displays = select(TVPlasma, Projetor, Monitor);
24 } Sozinho;
25 profile {
26   Context.pessoas > 1;
27   Context.displays = select(Pocket);
28 } Acompanhado;
```

Código 7 - Contrato VoD

Ainda no Código 7 (linha 11 – 14) é definida uma cláusula de negociação, que é utilizada pelo CR-RIO para selecionar o serviço adequado para as condições reinantes na sala onde o usuário se encontra. De acordo com ela, o primeiro serviço que poderá ser estabelecido é o *sSalaSozinho.* Caso não seja possível estabelecer o primeiro serviço, o sSalaAcompanhado será selecionado. Essa aplicação poderá transicionar do serviço *sSalaSozinho* para o *sSalaAcompanhado* e vice-versa (linhas 12 e 13).

A aplicação pode ser ativada de diversas maneiras: por exemplo, pela ativação do *Pocket-PC* do usuário no ambiente ubíquo (caso 1), uma aplicação administrativa externa através do *SDA-A* (caso 2) ou manualmente pelo usuário (caso 3). No caso 1, quando o usuário ativar seu *Pocket-PC* no ambiente ubíquo e solicitar uma autenticação, a aplicação VoD é automaticamente inicializada. No caso 2, o usuário poderá solicitar a inicialização da aplicação VoD através de uma aplicação externa que conterá uma lista de aplicações do usuário. Esta aplicação por sua vez irá comunicar-se com o *SDA-A,* através da interface *apply*, solicitando a inicialização da aplicação através dos dois comandos respectivos “load vod.adl;” e “start;”, ambos passados como parâmetro na interface *apply*. Por fim, no caso 3 o usuário poderá interagir com o *SDA-A* através de um terminal de linha de comando, disponível no próprio *SDA-A*, solicitando a inicialização da aplicação VoD através da execução dos mesmos comandos apresentados anteriormente (*load* e *start*).

Após a aplicação VoD ser inicializada o CR-RIO tenta estabelecer o serviço s*SalaSozinho*. Para esta tarefa, ele verifica se os perfis *ExisteUsuario, Sozinho* e *ExisteDispositivo* são satisfeitos mediante suas respectivas propriedades. Para tal verificação, o CR-RIO solicita aos agentes de recursos, existentes no ambiente, as informações das propriedades de cada perfil. Os agentes de recursos, por sua vez, utilizam os serviços de monitoração oferecidos pelo *SL&D* para obter todas essas informações, retornando-as para os respectivos agentes. Desta forma, o CR-RIO, através dos agentes de recursos, obtém as informações relevantes para determinar o atendimento aos requisitos mínimos exigidos pelos

serviços, como a quantidade de usuários em cada ambiente, os dispositivos de visualização disponíveis, entre outras informações do ambiente ubíquo. Esse processo possibilita determinar qual serviço poderá ser estabelecido. Após a verificação dos perfis envolvidos no serviço a ser estabelecido (todos eles sendo satisfeitos), o serviço é implantado utilizando o *SDA-A,* que fica responsável por definir as ligações entre a fonte de vídeo e o dispositivo disponível no ambiente (obedecendo a preferência do usuário) onde o usuário se encontra. Caso o serviço não possa ser estabelecido (por algum perfil não ter sido satisfeito), o CR-RIO procura estabelecer o serviço *sSalaAcompanhado,* repetindo todos os procedimentos realizados para o serviço s*SalaSozinho*. É importante ressaltar que o processo de escolha do dispositivo de visualização a ser utilizado é baseado na preferência do usuário (Código 7, linha 23 e 27 para o serviços *sSalaSozinho* e *sSalaAcompanhado*, respectivamente). As diferenças entre o serviço *sSalaSozinho* e o serviço *sSalaAcompanhado* são definidas pelos perfis envolvidos (*ExisteUsuario, Acompanhado e ExisteDispositivo*), os dispositivos de visualização a serem utilizados e a criação do transcodificador de vídeo que irá converter o formato do vídeo. O processo de implantação do serviço *sSalaSozinho* será apresentado na próxima seção.

Aspectos de utilização do SDA-A

Para que o *SDA-A* possa reconfigurar a aplicação VoD dinamicamente ele precisa obter todas as suas informações arquiteturais antes de iniciar o processo de gerenciamento. O Código 8 apresenta a descrição arquitetural em CBabel da aplicação VoD. Para essa aplicação, inicialmente foram definidos 4 *módulos*[2] (linhas 3-7) sendo eles o *Video, TVPlasma, Projetor, MonitorLCD e Pocket.*

Esses módulos servem apenas para auxiliar o Coordenador nas instanciações dos módulos remotos (nos dispositivos) durante as adaptações da aplicação. Os arquivos *.class* necessários para a instanciação desses módulos estão localizados em cada dispositivo remoto. As informações referentes aos novos tipos de módulos (módulos que representam os dispositivos) que forem configurados ao longo da execução da aplicação serão carregadas dinamicamente através da interface `apply (String command)` oferecida pelo suporte (comando `“new <arquivo_com_a_definicçao.adl>;”`), por exemplo, `apply (“new novo_dispositivo.adl”)`. Através deste comando a definição do novo dispositivo contido

[2] Os módulos definidos inicialmente representam os tipos de dispositivos que poderão ser utilizados pela aplicação.

no arquivo "novo_dispositivo.adl" será incorporada à aplicação VoD e poderá ser utilizado pelo SDA-A durante as adaptações da aplicação.

Também foi definido um *conector* (linha 8) *Trans-codificador* e uma *porta* (linha 2) *provide* que será utilizada para o envio do fluxo de vídeo. Na linha 10 é instanciado o *módulo Vídeo* no *host1*, o qual implementa o serviço VoD responsável por fornecer o vídeo ao usuário.

```
01 module VoD {
02  port provider;
03  module Servidor { out port provider; } Video;
04  module TVPlasma { in port provider; } TVPlasma;
05  module Projetor { in port provider; } Projetor;
06  module MonitorLCD { in port provider; } MonitorLCD;
07  module Pocket { in port provider; } Pocket;
08  connector TransCoder { in port provider;
09                         out port provider;} Trans-codificador;
10  instantiate video at host1;
11 } vod;
```

Código 8 - Descrição da arquitetura em CBabel da aplicação VoD

O processo de implantação do serviço *sSalaSozinho*, gerenciado pelo CR-RIO através do *SDA-A*, ocorre de acordo com a Figura 14.

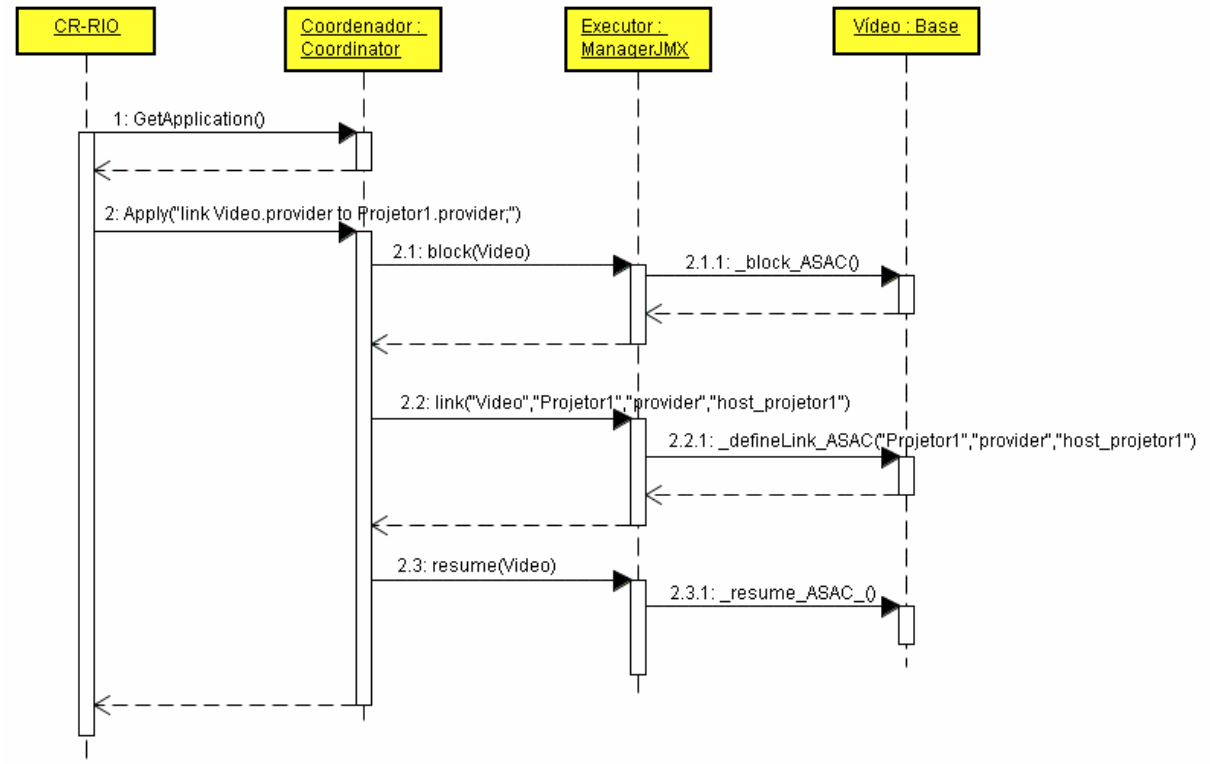

Figura 14 - Implantação de serviço *sSalaSozinho* do CR-RIO através do *SDA-A*

Inicialmente o CR-RIO verifica se o *módulo* que representa o dispositivo (no nível arquitetural da aplicação) a ser utilizado já foi instanciado anteriormente, neste caso o

Projetor1. Essa informação é obtida através da interface *GetApplication ( )* oferecida pelo Coordenador (*passo 1*). Caso o *módulo* já tenha sido instanciado, o CR-RIO obtém a sua localização. Caso negativo, o CR-RIO solicita ao *SDA-A* que instancie o *módulo* no dispositivo que deverá reproduzir o vídeo. A informação deste novo *módulo* instanciado será armazenada pelo Coordenador do *SDA-A.* Em seguida, o CR-RIO solicita ao *Coordenador* do *SDA-A,* através da interface *apply,* que realize a adaptação dos componentes envolvidos no serviço a ser estabelecido (*passo 2*). O Coordenador ao receber este comando, identifica qual a localização do *módulo Projetor1* (passado como parâmetro), através da estrutura de dados interna que contém a localização de todos os componentes de interesse da aplicação, e solicita ao módulo *Vídeo* que envie o fluxo de vídeo para o *Projetor1*. Para realizar esta adaptação o Coordenador solicita ao Executor responsável pelo *módulo Vídeo* que redefina o link do módulo Vídeo, fazendo que o mesmo envie o seu fluxo de vídeo para o *Projetor1 (passo 2.2).* O Executor por sua vez adapta o módulo Vídeo através da interface de serviço *_defineLink_ASAC* (*passo 2.2.1*). Antes de realizar as adaptações do módulo Vídeo, o Executor bloqueia o fluxo do vídeo através da interface de serviço *block* (*passo 2.1*) e, após a adaptação, ele libera o fluxo de vídeo através da interface *resume* (*passo 2.3*). Os mecanismos de bloqueio e liberação do fluxo de execução dos módulos são dois pontos de trabalhos futuros que podem ser melhor elaborados, pois, eles foram propostos para facilitar o controle da consistência durante as adaptações das aplicações.

A Figura 15 apresenta a implantação do serviço *sSalaAcompanhado,* processo semelhante ao do serviço *sSalaSozinho*. Inicialmente o CR-RIO solicita a criação de um trans-codificador (*passo 2*). Em seguida o Coordenador do *SDA-A* interpreta a requisição e solicita ao *Executor* do módulo Vídeo que instancie o trans-codificador (*passo 2.1*). Concluído a criação do trans-codificador, o CR-RIO solicita ao Coordenador do *SDA-A* que adapte o envio do vídeo para o *Pocket-PC* (*passo 3*). Mediante essa informação, o Coordenador interpreta a requisição e então solicita ao *Executor* que reconfigure o link do módulo vídeo, de modo que ele envie o seu fluxo de vídeo para o Pocket-PC através do trans-codificador (*passos 3.2 e 3.3*). Após concluir as adaptações das ligações entre o módulo Vídeo e *Pocket-PC,* o Coordenador libera o fluxo do vídeo através da interface de serviço *resume* (*passo 3.4*). É importante ressaltar que antes de qualquer instanciação de componente é realizada uma verificação da sua existência, como descrito na implantação do serviço *sSalaSozinho* (*passo 1*).

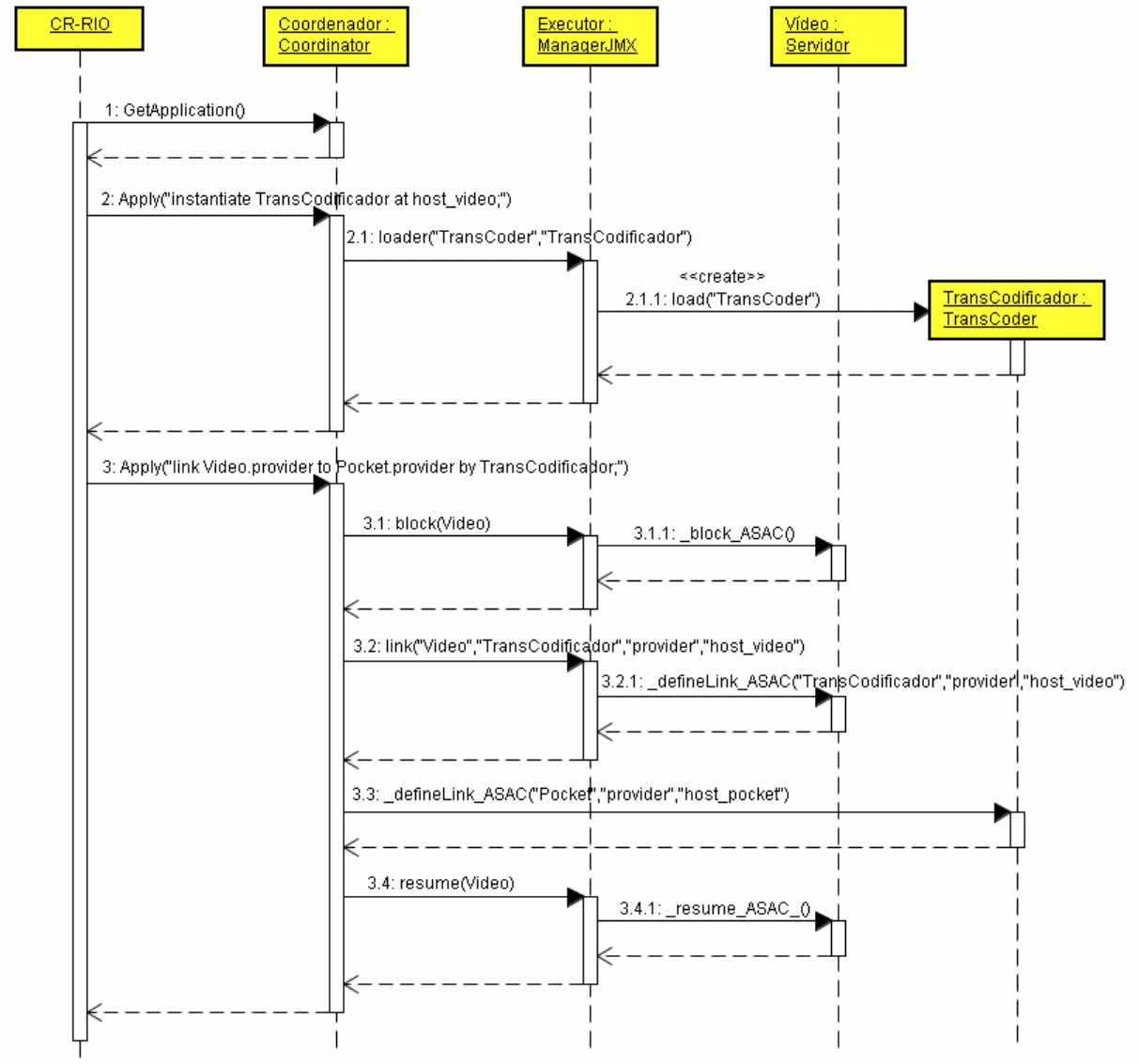

Figura 15 - Implantação de serviço *sSalaAcompanhado* do CR-RIO através do *SDA-A*

Quando o usuário muda de sala, o CR-RIO é notificado sobre a sua nova localização (através de seus agentes de recursos que atuam em conjunto com o *SL&D*), e reinicia todo o processo de estabelecimento de serviço baseado na nova sala onde o usuário se encontra.

Maiores detalhes de como implementar uma aplicação que utiliza o suporte de adaptação oferecido pelo *SDA-A* pode ser encontrado no Apêndice A desta dissertação.

4.1.2 Servidor replicado

Para ilustrar uma segunda utilização do *SDA-A* oferecendo suporte a capacidade de adaptação dinâmica, foi utilizada uma aplicação cliente-servidor com servidores replicados (CSR – aplicação Cliente/Servidor com servidores Replicados), que é melhor detalhada em [Freitas et al. 2005]. Nesta arquitetura, os clientes enviam requisições a grupos de servidores

replicados cujos componentes irão processar essas requisições em paralelo e enviar os resultados diretamente para os clientes. As requisições deverão ser atendidas dentro de um certo limite de tempo. Caso esse limite não seja cumprido, mais réplicas poderão ser alocadas ao grupo, com o objetivo de distribuir a carga e reduzir o tempo de resposta. Por outro lado, caso o tempo de resposta observado pelo cliente seja abaixo do esperado, ou seja, o processamento esteja ocorrendo mais rápido do que o requerido, as réplicas subutilizadas poderão ser desalocadas, visando obter uma melhor utilização dos recursos de processamento. A comunicação entre o cliente e o grupo de servidores é feita de forma transparente usando um *conector* especial que encapsula os mecanismos de comunicação de grupo (implementados através do suporte JavaGroups [Ban 1998]). A Figura 16 mostra a arquitetura do sistema com um cliente se comunicando com um grupo de servidores através do *conector*. Na verdade, o cliente estará ligado a um grupo de servidores através de um *conector* específico para comunicação de grupo, que também é responsável por monitorar o tempo de resposta das requisições observado pelo cliente.

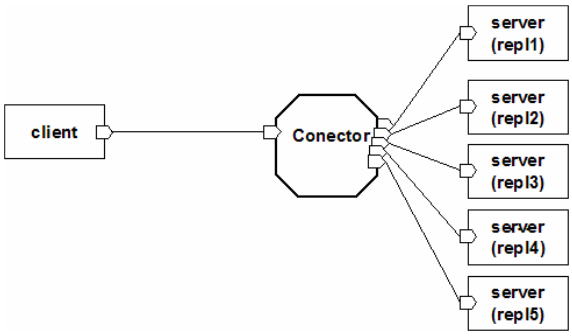

Figura 16 - Configuração com servidores replicados

Descrição do contrato

O contrato para esta aplicação deverá tratar um aspecto dinâmico na execução, que é o tempo de resposta percebido pelo cliente. O tempo de resposta do serviço varia dinamicamente de acordo com a carga dos servidores. Caso os servidores estejam sobrecarregados, o tempo de reposta do serviço para o cliente poderá aumentar, criando a necessidade de se adicionar réplicas de servidores para diminuir o tempo de resposta dos serviços solicitados. De forma análoga, caso o tempo de resposta se torne muito baixo isso indica que há recursos (réplicas) subutilizados no sistema. Isso possibilitará desalocar réplicas, disponibilizando seus recursos computacionais (processador) para outros serviços utilizarem.

O contrato do exemplo considera três estados em que o sistema pode se encontrar: (i) os tempos de resposta se mantêm dentro de uma faixa aceitável; (ii) Os tempos de resposta se encontram altos. Como foi assumido que isso ocorre devido a uma alta carga submetida aos servidores, ao entrar nesse estado o sistema irá acionar o perfil *addReplica* para adicionar servidores ao grupo, a fim de distribuir a carga e aliviar os servidores; (iii) Os tempos de resposta estão baixos. Isso provavelmente deve estar ocorrendo devido a existência de recursos (servidores) subutilizados no sistema. Ao identificar esse estado, o sistema irá acionar o serviço *sReplica* visando à liberação de recursos de processamento subutilizados e disponibilizá-los para outras aplicações.

```
01 QoScategory Processing {
02   utilization: decreasing numeric %;
03   clockFrequency: increasing numeric MHz;
04   priority: increasing numeric; }
05 QoScategory Client {
06   responseTime: numeric ms; }
07 QoScategory Replication {
08   numeberOfReplicas: increasing numeric;
09   maxReplicas: numeric;
10   replicaMaint: enum (add, remove, maintain) out;
11   groupComm: enum (p2p, multcast, broadcast) out;
12   distribPolicy: enum (bestCpu, roundRobin, random) out;}
```

Código 9 - Categorias de QoS Processing, Client e Replication

As categorias utilizadas no contrato são as apresentadas no Código 9, onde é considerado que o cliente deve obter um tempo de resposta para o serviço dentro de um intervalo especificado pelo projetista, ou seja, o tempo de resposta deve se manter entre 80 *ms* e 150 *ms*. Caso o requisito de tempo especificado pelo perfil *maintain* não seja satisfeito, o sistema deverá acionar outro serviço para tentar restabelecer o atendimento ao requisito. Em nosso exemplo, ele deve acionar o serviço *sAdd* caso o tempo de resposta se torne muito alto, e o serviço *sRemove* caso se torne muito baixo; a justificativa para essas ações foi explicada anteriormente. Usando essa metodologia teremos uma configuração dinâmica do número de servidores usados para compor o serviço, o que visa maximizar a eficiência no uso dos recursos, alocando mais réplicas quando o grupo se torna sobrecarregado e liberando-as quando o grupo se torna subutilizado.

A partir destes requisitos o contrato da aplicação CSR pode ser descrito pelo Código 10, onde são especificados três serviços. O serviço *sMain* (linhas 02-05) é o serviço preferencial, onde o tempo de resposta oferecido pelo serviço está de acordo com os requisitos de tempo requeridos pelo cliente, não sendo necessário adaptação na configuração da arquitetura (perfil *maintain*, linhas 20-24). Caso o tempo de resposta do serviço se torne maior do que o limite superior, o serviço *maintainReplica* é descontinuado e o sistema tentará iniciar

o serviço *sAdd* (linhas 06-09). Nesse caso, o perfil *addReplica* será imposto e uma ou mais réplicas serão criadas (linhas 25-28), sendo o número de réplicas limitado pelo perfil *pMax* (linhas 33-35). Caso esse limite seja alcançado, nenhuma outra réplica pode ser criada e o serviço não poderá ser provido. Por outro lado, caso o tempo de resposta observado se torne menor do que o limite, o serviço *sRemove* (linhas 10-12) é executado a fim de liberar recursos (réplicas) para outros serviços.

```
01 contract {
02   service {
03     instantiate server with maintain, cpu_01;
04     link client to server with pCom;
05   } sMain;
06   service {
07     instantiate server with cpu_01, addReplica, pMax;
08     link client to server with pCom;
09   } sAdd;
10   service{
11     remove server with cpu_02, removeReplica;
12   } sRemove;
13   negotiation {
14     sMain -> (sAdd || sRemove);
15     sAdd    -> sMain;
16     sRemove -> sMain;
17     sAdd -> out-of-service;
18   };
19} csr;
20 profile {
21   Client.responseTime <= 150;
22   Client.responseTime >= 80;
23   Replication.replicaMaint = maintain;
24 } maintain;
25 profile {
26   Client.responseTime > 150;
27   Replication.replicaMaint = add;
28 } addReplica;
29 profile {
30   Client.responseTime < 80;
31   Replication.replicaMaint = remove;
32 } removeReplica;
33 profile {
```

```
34   Replication.maxReplica = 8;
35 } pMax;
36 profile {
37   Replication.numberOfReplicas=3;
38   Replication.distribPolicy = random;
39 } pReplic;
40 profile {
41   Replication.groupComm = multicast;
42 } pCom;
43 profile {
44   Processing.utilization >= 90;
45 } cpu_01;
46 profile{
47   Processing.utilization <= 50;
48 } cpu_02;
```

Código 10 - Contrato para a aplicação CSR

Observando a cláusula de negociação, onde as mudanças entre os serviços são especificadas, pode-se ver que quando os serviços *sAdd* ou *sRemove* são efetivados eles são renegociados enquanto o tempo de resposta se mantiver fora do intervalo especificado (no exemplo, tempo de resposta < 80 *ms* ou tempo de resposta > 150 *ms*). Quando esses valores retornam para o intervalo desejado, o estabelecimento do serviço *sMain* é renegociado. Similarmente, caso uma das propriedades dos perfis envolvidos seja violada durante a operação, uma nova negociação pode ser iniciada. No pior caso, quando o serviço *sAdd* (ou *sMain*) é selecionado e nenhuma configuração de réplicas satisfaça o contrato, o estado *out-of-service* é alcançado e a aplicação é terminada.

Aspectos de utilização do SDA-A

Neste exemplo o *SDA-A* é utilizado para implantar a configuração inicial da aplicação e alocar ou desalocar réplicas de servidores remotos, mediante a necessidade observada pelo CR-RIO. A partir da descrição arquitetural da aplicação (Código 11) o *SDA-A* é capaz de realizar as alocações e desalocações que de fato resumem-se em criar e destruir instâncias de um servidor replicado em um *host* remoto.

```
01 module Cliente_Servidor {
02   port request;
03   module Client { out port request; } client;
04   module Server { out port request; } server;
05   connector ServerGroup {in port request; } serverGroup;
```

```
06   instantiate client at host1;
07   instantiate serverGroup at host2;
08   link client to serverGroup;
09 } csr;
```

Código 11 - Descrição arquitetural em CBabel da aplicação CSR

Observando o Código 11 pode-se notar que a arquitetura da aplicação é composta por:

- Um *módulo* cliente (client, linha 3), o qual representará os clientes;
- Um *módulo* servidor (*server*, linha 4), o qual representará os servidores replicados na configuração;
- Um *conector* de grupo (*serverGroup*, linha 5), o qual fará a ligação dos clientes ao grupo de servidores;
- Uma ligação do cliente com o grupo de servidores através do *conector serverGroup* (lin. 6-8);

Neste exemplo o CR-RIO solicita a adição ou remoção de réplicas, mediante o especificado no contrato, de acordo com o tempo de resposta observado pelos clientes. A Figura 17 apresenta o processo de adição de uma nova réplica no grupo de servidores.

Quando o CR-RIO solicita a criação de uma nova réplica (*passo 1*), o Coordenador do *SDA-A* interpreta o comando passado, no qual consta o *host* onde deverá ser criada a nova réplica.

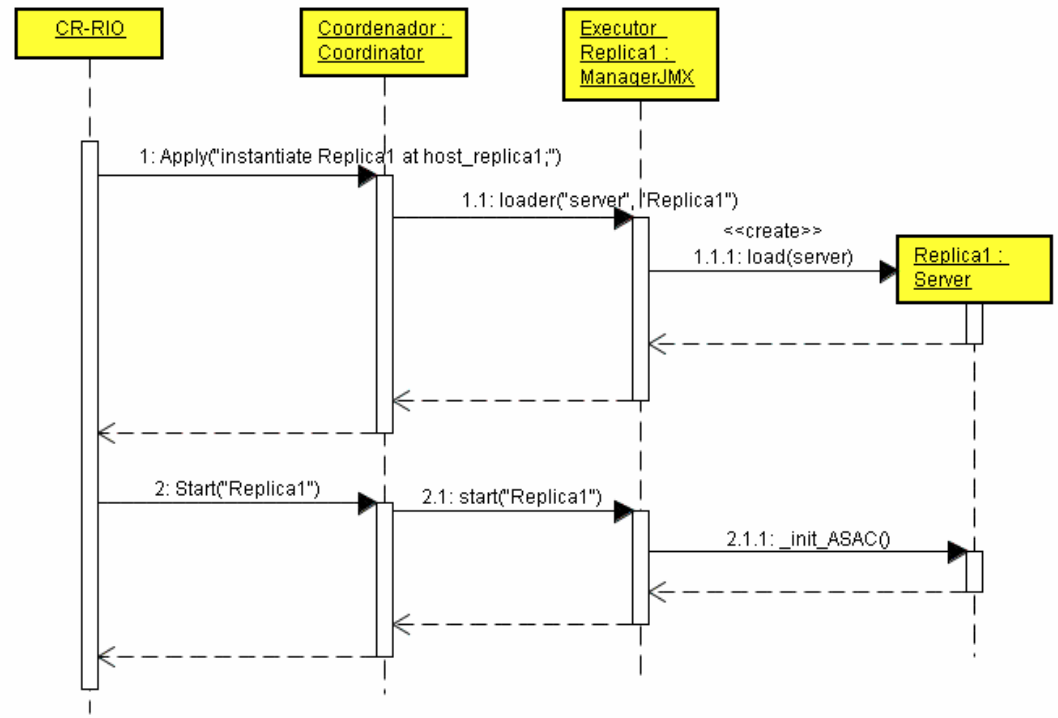

Figura 17 - Adição de uma nova réplica solicitada pelo CR-RIO via *SDA-A*

Uma vez interpretado o comando, o Coordenador comunica-se com o Executor remoto (*passo 1.1*), do *host* onde deverá ser criada a réplica, para que o mesmo possa instanciar localmente um objeto do tipo *Server* (*passo 1.1.1*) e inicializá-lo (*passo 2.1.1*) (através do método padronizado `_init_ASAC(String args[])` inserido pelo suporte durante o processo de adaptação das classes do usuário) com o intuito de que ele fique apto a receber requisições de serviços. Neste exemplo particular não é necessário realizar a ligação do conector de grupo com a nova instância do Servidor, pois os *módulos* Servidores (tipo *Server*) utilizam o mesmo mecanismo de comunicação de grupo utilizado pelo *conector* de grupo. Sendo necessário assim, apenas inicializar o *módulo* que representa a réplica no nível da configuração da aplicação, para que ele possa fazer parte do grupo. Já o processo de remover uma réplica (Figura 18), resume-se no CR-RIO a solicitar ao Coordenador a remoção da réplica desejada (*passo 1*). De posse da requisição o Coordenador solicita ao Executor, responsável pela réplica, a remoção da mesma da lista de objetos através da chamada do método `unload (String name)`, também oferecido pelo Executor (*passo 1.1*). A partir deste ponto a réplica não faz mais parte do grupo de servidores (*passo 1.1.1*).

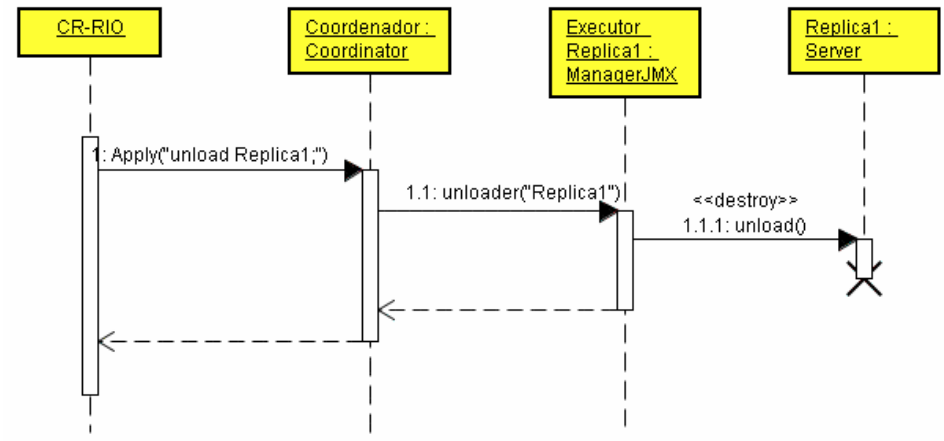

Figura 18 - Remoção de uma réplica solicitada pelo CR-RIO via *SDA-A*

4.1.3 Aplicação JBoss

Neste exemplo será ilustrado o gerenciamento dos módulos de uma aplicação de comércio eletrônico (CE), hospedada no servidor de aplicações JBoss, através do suporte oferecido pelo *SDA-A*. A aplicação CE consiste em um sistema que oferece para seus usuários serviços e produtos que podem ser adquiridos através da Internet. Inicialmente, o sistema foi projetado visando atender aos usuários de maneira uniforme, ou seja, todos os usuários recebem as mesmas informações referentes a promoções de produtos e serviços sem fazer

distinção entre os mesmos. Mas, com o passar do tempo viu-se a necessidade de uma personalização no atendimento de cada usuário do sistema. Então, foi estabelecido que cada usuário, ao autenticar-se no sistema, deverá receber as informações referentes às promoções de produtos e serviços oferecidos pelo sistema, de acordo com o perfil do usuário autenticado. O perfil do usuário é definido mediante as compras realizadas por ele no sistema. Para que o sistema possa atender esse novo requisito funcional, ele deverá ser capaz de identificar dinamicamente, para cada usuário autenticado, quais produtos e serviços são de seu interesse mediante as informações contidas em seu perfil.

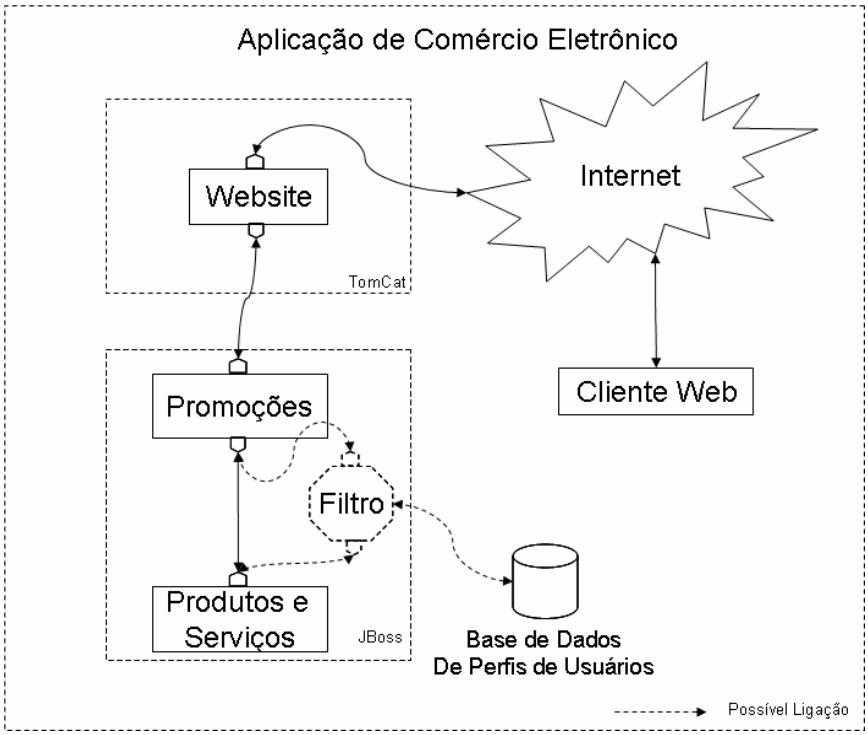

Figura 19 - Aplicação de comércio eletrônico

Como pode ser observado na Figura 19, a aplicação CE é utilizada pelos *Clientes Web*, os quais representam os usuários do sistema, possui um *Website* hospedado no servidor Web *TomCat* e dois módulos, *Promoções* e *Produtos e Serviços*, hospedados no servidor de aplicação JBoss. O módulo "*Produtos e Serviços*" fornece a lista de produtos e serviços disponíveis, e o módulo "*Promoções*" disponibiliza a lista de todos os produtos e serviços em promoção. Esses módulos representam uma parte dos módulos que compõem a aplicação CE.

Quando o usuário se autentica no sistema através do *Website*, são obtidas as informações sobre as promoções dos produtos e serviços disponíveis. Neste momento existem dois comportamentos possíveis: (i) o comportamento original no qual o sistema trata todos os usuários de forma uniforme e (ii) o comportamento personalizado onde cada usuário recebe informações sobre as promoções mediante o seu perfil. Este segundo comportamento é

representado na Figura 19 através da ligação do módulo *"Promoções"* com o módulo *"Produtos e Serviços"*, através do *conector "Filtro"*. O *Filtro* é responsável por determinar o perfil de cada usuário, estabelecido através de informações extraídas das compras realizadas por cada usuário do sistema, que são armazenadas na *"Base de Dados de Perfis de Usuários"*. Para o administrador, a tarefa de alterar o comportamento original do sistema é bem simples. Ele simplesmente pode utilizar um suporte, como o CR-RIO, ou desenvolver uma aplicação auxiliar que informa ao *SDA-A* quais as adaptações devem ser realizadas. Essa operação não é ilustrada na Figura 19, pois já foi ilustrada nos exemplos anteriores.

Aspectos de utilização do SDA-A

Em relação aos detalhes da utilização do *SDA-A* para a aplicação CE, foi elaborada uma descrição arquitetural (Código 12) que define a parte referente aos módulos hospedados no servidor de aplicação JBoss apresentados na Figura 19.

```
01 module CE {
02   port request;
03   module Promocoes {
04     in port request;
05     out port request; } Promocoes;
06   module ProdutoServicos {
07     in port request; } ProdutoServicos;
08   connector Filtro {
09     in port request;
10     out port request; } Filtro;
11   instantiate Promocoes at host1;
12   instantiate ProdutoServicos at host1;
13   link Promocoes to ProdutoServicos;
14 } ce;
```

Código 12 - Descrição arquitetural em CBabel da Aplicação CE

No Código 12 são definidos os módulos *Promocoes* (linhas 3-5), *ProdutosServicos* (linhas 6 e 7), o conector *Filtro* (linha 8-10) e a criação dos dois módulos e suas ligações (*Promocoes* e *ProdutoServicos*, linhas 11-13). A partir desta definição o *SDA-A* irá implantar a arquitetura padrão da aplicação CE. O processo de implantação resume-se em o Coordenador solicita ao Executor (hospedado no JBoss) que instancie os módulos `Promocoes` e `ProdutoServicos` (através dos comandos `"instantiate Promocoes at host1;"` e `"instantiate ProdutoServicos at host1;"`). Logo em seguida o Coordenador solicita ao Executor que realize as ligações entre os dois módulos (através do comando `"link Promocoes to ProdutoServicos;"`).

Para alterar o comportamento da aplicação para o atendimento personalizado, o administrador utilizará o *SDA-A* que criará o *conector Filtro* (através do comando

"instantiate Filtro at host1;") e reconfigurará as ligações entre os *módulos* "Promocoes" e "ProdutoServicos" fazendo com que o fluxo de execução passe pelo *conector* "Filtro" (através do comando "link Promocoes to ProdutoServicos by Filtro;").

Quando o administrador desejar remover o filtro implantado na aplicação, ele utilizará novamente o *SDA-A* para reverter o processo. O processo para remover o filtro resume-se em o administrador solicitar ao *SDA-A* que remova o conector filtro (através do comando "unload filtro;") e refaça as ligações entre os módulos Promocoes e ProdutoServicos (através do comando "link Promocoes to ProdutoServicos;").

4.1.4 Reconfiguração do JBoss

Nesta seção será apresentado um exemplo de uma reconfiguração do servidor de aplicações JBoss através do suporte oferecido pelo *SDA-A*. Como apresentado no Capítulo 3, Seção 3.9, o gerenciamento do servidor de aplicações é oferecido pelo Executor do suporte carregado nele como um componente de serviço. O exemplo apresentado descreve uma reconfiguração do servidor de aplicações para trabalhar em modo *Cluster*, ou seja, trabalhar em conjunto com outros servidores de aplicações JBoss. Desta forma, é aumentado o nível de tolerância a falhas e a capacidade de processamento do servido de aplicações. Maiores informações sobre o funcionamento do JBoss em modo *Cluster* pode ser encontrado em [Sun 2005].

A configuração em CBabel, apresentada no Código 13, foi extraída do arquivo XML (*cluster-service.xml*) pertencente ao servidor de aplicações JBoss. O conteúdo do XML foi adaptado para a sintaxe CBabel de modo que o *SDA-A* possa interpretá-lo e configurar o servidor de aplicações para trabalhar em modo *Cluster*. Maiores detalhes da configuração e funcionamento do JBoss em modo *Cluster* pode ser encontrado em [Fleury e Reverbel 2003].

```
01 module JBossCluster {
02  port default;
03  module org.jboss.ha.framework.server.ClusterPartition {
04    out port default;
05  } DefaultPatrition1;
06  module org.jboss.ha.hasessionstate.server.HASessionStateService {
07    out port default;
08  } HASessionState1;
09  module org.jboss.ha.jndi.HANamingService {
10    out port default;
11  } HAJNDI1;
12  module org.jboss.invocation.jrmp.server.JRMPInvokerHA {
13    out port default;
```

```
14  } invoker1;
15  module org.jboss.cache.invalidation.bridges.JGCacheInvalidationBridge {
16    out port default;
17  } InvalidationBridge1;
18  instantiate DefaultPatrition1 at localhost; ASAC_JBoss
19            jboss:service=DefaultPartition PartitionName=ATARI
20            NodeAddress=${jboss.bind.address} DeadlockDetection=False
21            StateTransferTimeout=60000
22  instantiate HASessionState1 at localhost; ASAC_JBoss
23            jboss:service=HASessionState
24  instantiate HAJNDI1 at localhost; ASAC_JBoss jboss:service=HAJNDI
25  instantiate invoker1 at localhost; ASAC_JBoss
26            jboss:service=invoker,type=jrmpha
27  instantiate InvalidationBridge1 at localhost; ASAC_JBoss
28            jboss.cache:service=InvalidationBridge,type=JavaGroups
29} jboss_cluster;
```

Código 13 - Configuração do JBoss através do *SDA-A*

De acordo com o Código 13, nas linhas 02-17 são definidos os módulos que serão inicializados no servidor e nas linhas 18-28 esses módulos são instanciados. O processo de instanciação dos módulos tratam na realidade da inicialização dos mesmos no contexto do servidor de aplicações JBoss. Pode-se observar também que alguns parâmetros necessários são passados nas declarações de inicialização dos módulos. O *SDA-A* utiliza a palavra chave *ASAC_JBoss* para identificar que este gerenciamento será efetivado no próprio servidor de aplicações e não sobre uma aplicação hospedada nele. Como este exemplo é de caráter ilustrativo, não foram apresentados todos os detalhes definidos no XML de configuração "*cluster-service.xml*".

4.2 Avaliação Experimental

Para avaliar o custo de uma adaptação e o *overhead* gerado pelo *SDA-A* na execução de uma aplicação, foi realizada uma bateria de testes. Além disso, foi calculada a média aritmética de 10.000 invocações de um serviço (de custo computacional zero[3]) com um intervalo de confiança de 95% para cada ponto encontrado nos gráficos (Figura 20 e Figura 21). Utilizou-se uma aplicação Cliente/Servidor como base para coletar as medidas de tempo de resposta do serviço solicitado pelo cliente e o tempo gasto para realizar uma adaptação. Para fins de comparação, a mesma aplicação Cliente/Servidor foi implementada em três versões: (i) uma versão usando apenas o padrão JMX (disponível a partir da versão 1.5.0 do Java), (ii) uma usando os recursos de comunicação remota RMI e (iii) uma usando o padrão Java *J2SE* sem fazer usos de mecanismos de comunicação remota (no caso o RMI ou JMX). Essa última versão implementada será gerenciada pelo *SDA-A* através dos suportes de

[3] O custo computacional zero apresentado foi definido como uma execução de um método que não realiza nenhum tipo de instrução. Ele apenas retorna um valor nulo "null".

gerenciamento JMX e Java *J2SE* ambos oferecidos por ele. Além disso, do ponto de vista do programador a implementação da comunicação remota entre os módulos da aplicação será resolvida pelo suporte de adaptação dinâmica.

O parque computacional utilizado para os testes foi composto por 10 *hosts*, sendo todos eles Pentium 4, com 3.5Ghz de freqüência de processador, 512MB de RAM e com sistema operacional Linux Fedora Core 3. Todos os módulos *(Cliente e Servidor)* e conectores da aplicação foram distribuídos em diferentes *hosts* da rede (no caso das execuções remotas). Durante os testes, os números de conectores interpostos entre o cliente e o servidor variaram de 0 a 8. No caso de 0 conector, o cliente faz a chamada direto para o Servidor. Foram realizados dois tipos de teste, local e remoto, pois o *SDA-A* possui um comportamento diferenciado quando todos componentes de uma aplicação são executados no mesmo *host* (ver Capítulo 3 seção 3.4.2). Nas próximas seções serão apresentados os resultados obtidos.

4.2.1 Testes de execução local

De acordo com a Figura 20, as versões da aplicação Cliente/Servidor desenvolvidas utilizando apenas o padrão JMX e o RMI, ou seja, projetadas para a execução distribuída, sendo executadas localmente obtêm um desempenho inferior em relação à aplicação Java *J2SE* gerenciada pelo *SDA-A*. Isso acontece porque o *SDA-A* possui um mecanismo que identifica quando um determinado *componente* realiza uma chamada de serviço para outro *componente* que se encontra no mesmo *host*. Desta forma, ele redireciona a chamada diretamente para o objeto (*componente*) na memória do *host*. Esse processo de identificação dos componentes locais contribui no desempenho da aplicação por não ser necessário a obtenção das referências a objetos remotos como é feito no caso das versões JMX e RMI. Desta forma, o *SDA-A*, em ambas as formas de suporte de gerenciamento, foi capaz de obter um melhor desempenho em relação ao JMX e ao RMI.

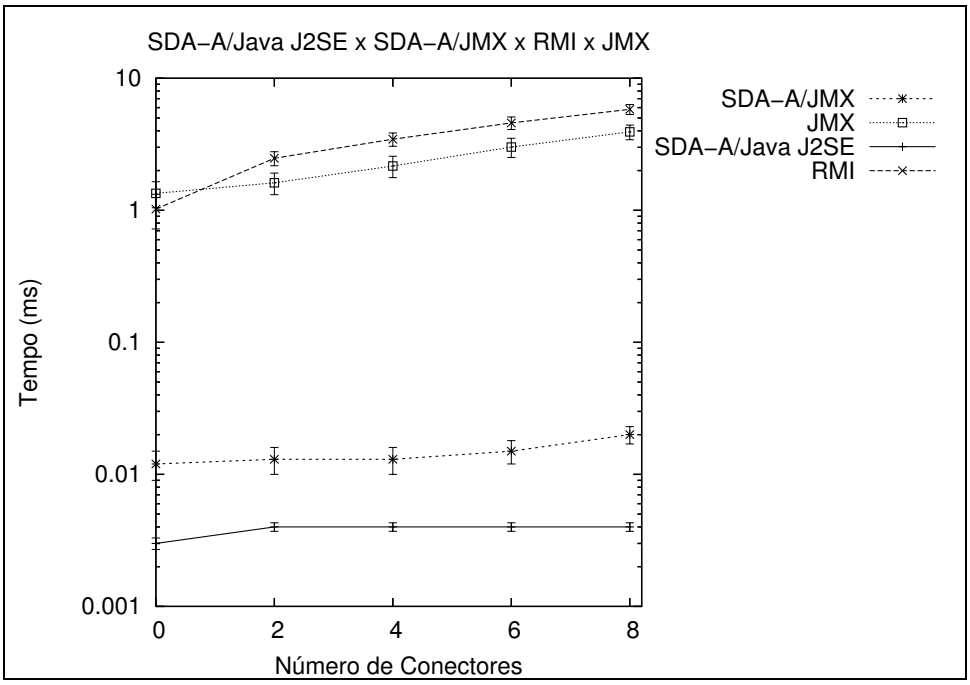

Figura 20 - Tempos de execução local

Existe ainda uma diferença de desempenho entre os suportes de gerenciamento oferecidos pelo *SDA-A*. O gerenciamento utilizando o *SDA-A/JMX* tem um desempenho inferior ao gerenciamento utilizando o *Java*/*J2SE*, devido ao *overhead* acrescentado pelo padrão JMX. O *overhead* considerado é referente ao redirecionamento de chamadas que passam pelo *MBean Server* [Sun 2005], o que torna possível o gerenciamento dos objetos tipo *MBean* de forma dinâmica [Sun 2005]. Finalmente, deve ser notado que para os dois suportes de gerenciamento disponíveis no *SDA-A*, o custo causado pela inserção de *conectores* é relativamente baixo, em relação às versões originais. Desta forma, torna atrativa a utilização do *SDA-A* para a implementação de adaptações locais em aplicações utilizando Java ou JMX.

4.2.2 Testes de execução remota

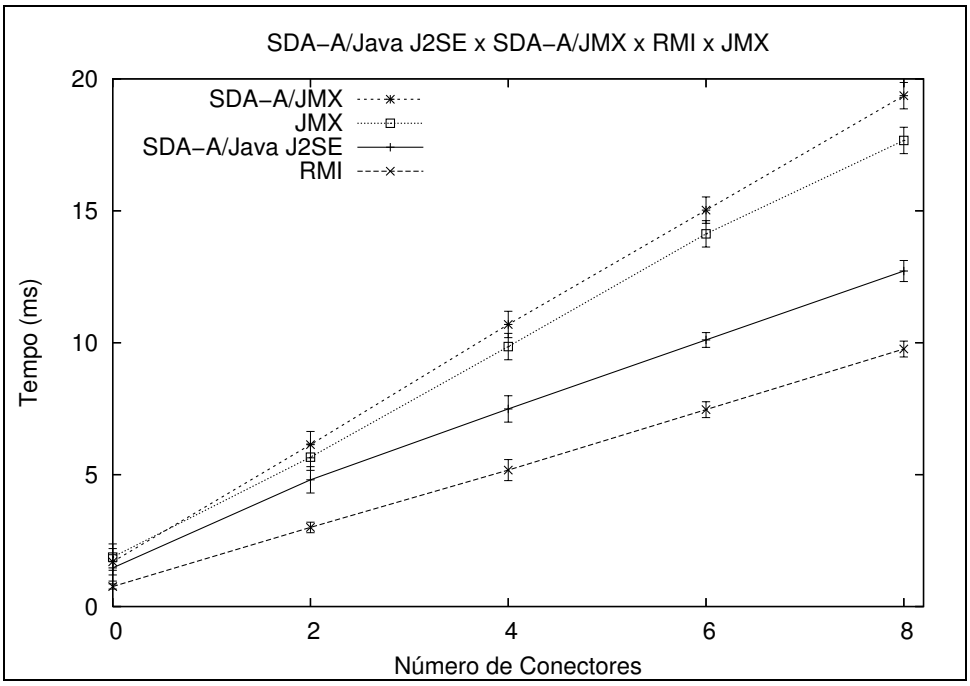

Figura 21 - Tempo de execução remoto

A Figura 21 apresenta os resultados dos testes remotos. Pode ser observado que o *SDA-A*, utilizando o suporte de gerenciamento *Java*/*J2SE* para gerenciar a aplicação Cliente/Servidor utilizando o padrão Java *J2SE*, obteve um desempenho satisfatório, ficando entre as versões das aplicações desenvolvidas utilizando o RMI e o JMX.

A Tabela 5 apresenta uma comparação de desempenho em termos percentuais das aplicações gerenciadas pelo *SDA-A* usando o suporte Java J2SE (*SDA-A/Java J2SE*) com relação às aplicações que utilizam o JMX e o RMI. A primeira linha mostra uma comparação de ganho na utilização do *SDA-A/Java J2SE* em relação à aplicação que utiliza o JMX (Aplicação JMX). A segunda linha apresenta o ganho da aplicação que utiliza o RMI (Aplicações RMI) em relação ao *SDA-A/Java J2SE*.

Tabela 5 - Comparação de desempenho

Nº de Conectores	**0**	**2**	**4**	**6**	**8**
SDA-A/Java J2SE melhor que aplicação utilizando o JMX	*21,97%*	*15,12%*	*23,97%*	*28,47%*	*28,01%*
Aplicação utilizando o RMI melhor que *SDA-A/Java J2SE*	*48,05%*	*37,61%*	*30,98%*	*26,15%*	*23,25%*

De acordo com a Tabela 5, o desempenho entre as Aplicações RMI e as gerenciada pelo *SDA-A* (*SDA-A/Java J2SE*), para o caso onde nenhum *conector* é interposto entre o cliente e o servidor, foi 48% melhor, já para 8 *conectores* foi apenas 23,25% superior. Em suma, é possível notar que na medida em que o número de *conectores* cresce, os benefícios de desempenho da utilização do RMI diminuem. É importante lembrar que o RMI não oferece

suporte para adaptação dinâmica. Desta forma, as aplicações projetadas que o utilizam e necessitam de adaptação dinâmica, precisam criar mecanismos extras os quais possam oferecer a adaptação dinâmica. A diferença de desempenho entre a Aplicação RMI e a aplicação gerenciada pelo *SDA-A/Java J2SE* é causada pelo *overhead* inserido pelo suporte para tornar possível o gerenciamento dinâmico dos *componentes*; mas, deve ser notado que o desempenho da aplicação gerenciada pelo *SDA-A/Java J2SE* é bem superior ao da aplicação que utiliza o JMX (Aplicação JMX). Já o desempenho das aplicações gerenciadas pelo SDA-A através do suporte JMX (*SDA-A/JMX*), ficou bem próximo da Aplicação JMX, tornando viável a sua utilização para adaptações em contextos JMX. A pequena diferença entre a aplicação gerenciada pelo *SDA-A/JMX* e a Aplicação JMX também é causada pelo *overhead* inserido pelo *SDA-A*.

4.2.3 Custo de adaptação

Além dos testes para avaliar o *overhead* inserido pelo *SDA-A,* foi mensurado o custo para adicionar ou trocar um *componente* dinamicamente (com a aplicação em execução) utilizando o suporte de gerenciamento *SDA-A/Java J2SE*. Através destas medidas podemos saber qual será o custo de cada adaptação realizada na aplicação. O tempo gasto na troca ou adição de um *conector* varia em função de ele já ter sido instanciado anteriormente ou não, ou seja, caso o componente já foi instanciado anteriormente, por uma utilização anterior na aplicação, o *SDA-A* não irá realizar todo o processo de instanciação e/ou modificação do *conector* como visto na Seção 3.6. A Tabela 6 apresenta as medidas coletadas referente ao tempo de troca e adição de um *conector* em uma arquitetura carregada em um único *host*. Já a Tabela 7 apresenta as medidas coletadas para uma arquitetura distribuída. Deve ser notado que os valores observados são similares aos obtidos em outros projetos com objetivos semelhantes, e.g. [Almeida et al. 2001]. Almeida utiliza uma aplicação Cliente/Servidor como base de teste para coleta de medidas do *overhead* gerado pelo suporte proposto, um *Suporte para Reconfiguração Dinâmica para Aplicações Baseadas em CORBA,* o qual será apresentado no Capítulo 5. Após a realização dos testes, o suporte proposto por Almeida introduz um *overhead* 12.4 % no desempenho da aplicação, enquanto o SDA-A introduz 21,97 % (Tabela 5). É importante ressaltar que a tecnologia utilizada por Almeida nos testes realizados foi CORBA, enquanto o *SDA-A* utiliza apenas Java.

Tabela 6 - Custos locais observados

Custos Locais	Substituir um Conector	Inserir um Conector

Criando uma Instância	23,309 (ms)	14,206 (ms)
Sem Criar uma Instância	12,275 (ms)	10,328 (ms)

Tabela 7 - Custos remotos observados

Custos Remotos	Substituir um Conector	Inserir um Conector
Criando uma Instância	28,665 (ms)	19,552 (ms)
Sem Criar uma Instância	21,455 (ms)	17,788 (ms)

4.3 Conclusão do Capítulo

Neste capítulo foram apresentados 4 exemplos de casos de uso onde o *SDA-A* foi utilizado para oferecer às aplicações a capacidade de adaptação dinâmica. Para o primeiro e o segundo exemplo foi utilizado o *framework* CR-RIO, que se encontra em desenvolvimento, para facilitar a concepção e o gerenciamento das tomadas de decisões de adaptações realizadas pelo *SDA-A*. No terceiro exemplo foi utilizado apenas o suporte desenvolvido, demonstrando assim que o mesmo pode ser utilizado independentemente do CR-RIO e de qualquer outro suporte extra. No último exemplo o suporte foi utilizado para configurar o servidor de aplicações JBoss para trabalhar em *Cluster*.

Foram apresentados alguns resultados de experimentos realizados para medir *overhead* e custos associados aos mecanismos utilizados na implementação do *SDA-A*. Além disso, o *overhead* gerado pelo *SDA-A* e os custos gerados pela inserção, remoção ou troca de componentes em uma aplicação demonstram que ele atende aos requisitos de flexibilidade e desempenho para as aplicações modernas. Foi realizada uma comparação entre as formas de suporte oferecidas pelo *SDA-A*, tendo como conclusão que o suporte de gerenciamento de aplicações *Java J2SE* é o mais indicado para aplicações que necessitem de uma menor interferência em sua execução. Com o suporte de gerenciamento voltado para o servidor de aplicações JBoss (oferecido pelo SDA-A), as aplicações hospedadas nele podem ser adaptadas dinamicamente através de primitivas de alto nível de abstração. Além das aplicações, os módulos de serviços do servidor de aplicações também podem ser configurados dinâmica e remotamente. No próximo capítulo serão apresentadas algumas propostas relacionadas a adaptação dinâmica de aplicações e um comparativo das mesmas com o *SDA-A*, em termos de como elas oferecem o suporte de adaptação dinâmica para aplicações.

Capítulo 5

Trabalhos Relacionados

Neste capítulo serão apresentados alguns trabalhos relacionados ao *SDA-A*. Além disso, para as comparações entre as propostas foram considerados os aspectos referentes à adaptação dinâmica das aplicações, ou seja, a forma pelas quais as propostas oferecem a capacidade de inserir, remover ou atualizar um componente em uma aplicação.

5.1 Uma Infra-estrutura para Adaptação Dinâmica de Aplicações Distribuídas

Em [Moura et al. 2002] é proposta uma infra-estrutura para adaptação dinâmica de aplicações distribuídas (assumido aqui como sendo *Smart Proxy Lua*) com o intuito de simplificar o desenvolvimento das mesmas. Através da infra-estrutura proposta, as aplicações podem adaptar automaticamente propriedades não-funcionais de seus componentes.

Para isso, a proposta possui um mecanismo de seleção de componente denominado *Smart Proxy*, que seleciona dinamicamente os componentes a serem utilizados pela aplicação. Essa seleção é realizada através do serviço de descoberta de componentes *Trading Service* [OMG 2000] definido pela arquitetura CORBA. O mecanismo *Smart Proxy* da infra-estrutura proposta explora os recursos da arquitetura CORBA para oferecer a capacidade de adaptação dinâmica para aplicações. Ou seja, enquanto o *SDA-A* oferece primitivas de alto nível de abstração para inserção, remoção e atualização de componentes em uma aplicação, o *Smart Proxy* faz uso de mecanismos, como o *Trading Service* e *LuaOrb*, para agregar novos componentes à aplicação. Embora o *Smart Proxy* explore os recursos da arquitetura CORBA, ele não oferece mecanismos para atualizar componentes de forma individual. Isto é, componentes que estão sendo utilizados não podem ser atualizados dinamicamente sem

causar impactos na execução da aplicação. Este problema é contornado no *SDA-A* através do uso das primitivas `block e resume`.

Já as ligações entre os componentes da aplicação na proposta *Smart Proxy Lua* são definidas através de chamadas de serviços interno ao *Smart Proxy*, tornando assim, necessária uma modificação de código para que uma nova ligação possa ser acrescentada na aplicação. Já no *SDA-A*, essas ligações são definidas através de primitivas de alto nível. Desta forma, as ligações podem ser adicionadas ou removidas a qualquer momento, sem a necessidade de alterar nenhuma linha de código.

5.2 LuaSpace

O *LuaSpace* [Batista e Rodriguez 2001, Batista e Rodriguez 2000] é um ambiente para reconfiguração dinâmica de aplicações distribuídas baseadas em componentes. Ele utiliza a linguagem interpretada Lua em conjunto com o *LuaOrb*, para oferecer os recursos de reconfiguração em tempo de execução. O *LuaOrb* é o elemento que faz a ligação entre a linguagem Lua e a plataforma CORBA. Ele define um mapeamento entre tipos de uma linguagem (Lua no caso) e IDL, disponibilizando o acesso dinâmico a objetos CORBA da mesma maneira que se faz acesso a objetos Lua.

O *LuaSpace* também segue o modelo de programação orientado à configuração, adotado pelo *SDA-A*, caracterizando-se por separar os aspectos estruturais de uma aplicação da implementação de seus componentes. Ele utiliza um conjunto de ferramentas baseadas na linguagem Lua (além do *LuaOrb*), como a *Biblioteca Alua*, um *Interpretador Lua* e um *Console Lua*. Além dessas ferramentas, o *LuaSpace* conta com um *Conector Genérico* [Batista et al. 2000] que oferece a capacidade de adaptar o comportamento de uma aplicação.

O Conector Genérico é um mecanismo que seleciona componentes dinamicamente para executar serviços requisitados por outros componentes da mesma aplicação. Essa seleção baseia-se em um repositório de componentes existente no *LuaSpace*. Através do *Conector Genérico* a aplicação pode ser adaptada dinamicamente. Contudo, a definição da adaptação é realizada no código do *Conector Genérico*, ou seja, comandos condicionais (if) ou iterações (while) determinam a seleção de novos componentes, de acordo com condições satisfeitas em tempo de execução. Essa característica torna baixo o nível de abstração do *LuaSpace*.

O *LuaSpace* permite a inserção e remoção de componentes e conexões em uma aplicação, como o *SDA-A*, através do *Console Lua*. Nele o programador pode definir as

reconfigurações em aplicações através de comandos para inserir ou remover componente e conexões. No *LuaSpace* o modelo de interconexões entre os componentes acontece de maneira implícita, ou seja, com simples chamadas de métodos remotos. Em contra partida, no *SDA-A* as interconexões entre os elementos da aplicação são feitas através das ligações entre portas de entradas e saídas. Neste caso, para desfazer uma conexão entre dois componentes de uma aplicação é necessário apenas remover a ligação (*link*) entre eles (maiores detalhes ver Capítulo 3, Seção 3.2), enquanto no *LuaSpace* esse processo se torna uma tarefa de baixo nível, sendo necessário uma possível modificação de código dos componentes envolvidos.

5.3 Suporte de Reconfiguração Dinâmica para Sistemas Baseados em CORBA

Em [Almeida et al. 2001] é proposto um suporte que permite a adaptação dinâmica de sistemas distribuídos baseados em *CORBA* (assumido aqui como sendo Suporte CORBA). Através deste suporte a adaptação pode ser realizada de forma transparente, tanto para o cliente, quanto para o servidor de uma aplicação. Essa proposta é baseada em CORBA e inclui os seguintes elementos: Reconfiguration Manager, Location Agent, Reconfigurable Object Factory, Reconfiguration Agents.

O *Reconfiguration Manager* é o elemento central do suporte (equivalente ao *Coordenador* do *SDA-A*). Ele interage com os demais elementos do suporte, possibilitando a adaptação das aplicações. É através do *Reconfiguration Manager* que os projetistas podem criar, remover ou atualizar os objetos da aplicação. O *Location Agent* oferece um registro para a localização de objetos reconfiguráveis, sendo encarregado de resolver a localização física desses objetos. O *Reconfigurable Object Factory* (equivalente ao *Executor* do *SDA-A*) implementa o *design pattern Factory*. Ele é responsável por criar os objetos das solicitações provenientes do *Reconfiguration Manager*. E, por fim, o *Reconfiguration Agents* é responsável por restringir o comportamento dos objetos que podem ser afetados durante uma reconfiguração.

Em linhas gerais, o suporte oferece a capacidade de adaptação dinâmica de objetos distribuídos através do *Reconfiguration Agent,* o qual informa ao *Reconfiguration Manager* quando os objetos estão prontos para serem manipulados. No entanto, o suporte não aborda os aspectos referentes às reconfigurações das interconexões entre os componentes de uma aplicação. As reconfigurações das interconexões são definidas pelos programadores no código

dos componentes da aplicação. Essa forma de reconfiguração torna o processo de redefinição de interconexões uma tarefa de baixo nível, dificultando assim, a adaptação das aplicações.

O Suporte CORBA proposto explora os recursos da arquitetura CORBA (os *interceptors*) para oferecer a capacidade de adaptação dinâmica para aplicações. Já o *SDA-A* utiliza o *design pattern Architecture Configurator* e um conjunto de classes padronizadas para oferecer tal capacidade. Além disso, as interconexões entre os componentes da aplicação podem ser redefinidas através de primitivas de alto nível de abstração. No *SDA-A* a determinação do momento seguro para realizar adaptação fica a cargo do desenvolvedor ou de um mecanismo externo, como visto no Capítulo 4, Seção 4.1, o qual determina quando é necessária uma adaptação.

5.4 Comparativo das Propostas

Após o estudo sobre as propostas apresentadas nas seções anteriores, foi possível montar um quadro comparativo que relaciona algumas características das propostas apresentadas com as do *SDA-A*, levando em consideração os seguintes requisitos:

Consistência Durante e Após Adaptação → Indica se a proposta apresenta algum mecanismo explícito que trata os aspectos referentes a consistência da aplicação (e.g, estados dos componentes), durante e após a sua adaptação.

Nível de Abstração → Indica qual o nível de abstração oferecido pela proposta para realizar uma adaptação dinâmica. O indicador **Baixo** representa que a adaptação é definida no nível de programação, utilizando cláusulas condicionais ("*if*") ou algo do gênero. O indicador **Médio** representa que a adaptação utiliza poucos recursos de baixo nível e alguns recursos de alto nível (interface, scripts de adaptação). E o indicador **Alto** representa que a proposta não faz uso de mecanismos de baixo nível para realizar a adaptação. Ela apenas utiliza recursos, como por exemplo, scripts de adaptação ou interfaces de interação para realizar a adaptação através de comandos pré-definidos.

Adaptação Dinâmica → Indica se a proposta oferece a capacidade de adaptação em tempo de execução.

Separação de Interesses → Indica se a proposta utiliza a técnica de separação de interesses. Este requisito foi considerado devido ao benefício que ele agrega durante o processo de adaptação dinâmica das aplicações [Sztajnberg 2002, Tarr et al. 1999].

Requer Mecanismos Especializados → Indica se a proposta faz uso de algum mecanismo específico para oferecer a capacidade de adaptação dinâmica. Ou seja, se a proposta utiliza mecanismos externos não conhecidos ou divulgados em âmbito comercial, como por exemplo, uma maquina virtual Java cujo núcleo foi modificado para oferecer mecanismos extras que facilite a adaptação de aplicações.

Tabela 8 - Comparativos das propostas

	SDA-A	**LuaSpace**	**Smart Proxy Lua**	**Suporte CORBA**
Consistência Durante e Após Adaptação	Não	Não	Não	Sim
Nível de Abstração	Alto	Baixo	Médio	Médio
Adaptação Dinâmica	Sim	Sim	Sim	Sim
Separação de Interesses	Sim	Sim	Não	Não
Requer Mecanismos Especializados	Não	Não	Não	Não

De acordo com a Tabela 8 todas as propostas avaliadas atendem o requisito de ***Adaptação Dinâmica*** e nenhuma delas faz uso de mecanismos de uso restrito ou especializado (**Requer Mecanismos Especializados**). Quanto ao requisito de ***Separação de Interesses,*** apenas o ***LuaSpace*** e o ***SDA-A*** atendem a esse requisito. Com relação ao requisito de ***Consistência Durante e Após Adaptação,*** a única proposta que atende explicitamente a este requisito é o ***Suporte CORBA.*** Porém, o ***SDA-A*** fornece mecanismos (comandos de alto nível de abstração, e.g., `resume` e `block`) pelos quais os desenvolvedores podem utilizá-los para garantir a consistência da aplicação durante e após sua adaptação. Referente ao requisito de ***Nível de Abstração,*** a proposta que apresentou o nível mais satisfatório foi o ***SDA-A***, já que utiliza uma linguagem de descrição arquitetural e primitivas de reconfiguração que elevam o nível de abstração em relação às demais propostas. Além disso, como no ***Suporte CORBA,*** o ***SDA-A*** também oferece uma interface para que as aplicações possam interagir diretamente com o suporte.

O ***Smart Proxy Lua*** possui um mecanismo de monitoração que, para este trabalho, não foi considerado durante as pesquisas. No entanto, existe um trabalho sendo desenvolvido em [Cardoso 2006] que pode ser utilizado em conjunto com o *SDA-A*, para expandir a sua capacidade de adaptação dinâmica, podendo assim, determinar o momento adequado para

realizar uma adaptação mediante o monitoramento observado, ou até mesmo tomar decisões sobre qual recurso a ser utilizado é mais conveniente em um determinado momento.

5.5 Conclusão do Capítulo

Neste capítulo foram apresentadas três propostas relacionadas à adaptação dinâmica de aplicações. Cada uma das propostas possui características diferentes, mas que em linhas gerais atendem às necessidades de adaptação dinâmica das aplicações distribuídas apresentadas no Capítulo 1, e.g., aplicações de computação autônoma e aplicações ubíquas. Foi elaborado ainda, um quadro comparativo onde é possível observar algumas características importantes para a utilização das propostas no contexto de adaptação dinâmica de aplicações. Além das propostas apresentadas, existem outras que não foram abordadas neste trabalho, tais como [de Lemos 2006, Dowling e Cahill 2001, Bidan et al. 1998, Keeney e Cahill 2003, Papadopoulos e Arbab 2001], mas que foram investigadas durante o desenvolvimento do *SDA-A*.

Capítulo 6

Conclusão e Trabalhos Futuros

O objetivo deste trabalho foi realizar um estudo das técnicas de adaptação dinâmica de arquiteturas, visando utilizá-las em conjunto com tecnologias e ferramentas de suporte a adaptação disponíveis, tais como o padrão *Java Management Extensions (JMX)* e o *toolkit* de manipulação de *Java bytecode Javassist*. Esse estudo propiciou o desenvolvimento de um suporte para adaptação dinâmica de arquiteturas distribuídas, denominado SDA-A (*Support for Dynamic Architecture-Adaptation*). Através de primitivas de alto nível de abstração (nível arquitetural) oferecidas por ele, é possível inserir, atualizar e/ou remover componentes de uma arquitetura. É possível ainda, redefinir as ligações entre os novos componentes e/ou os já existentes.

O *SDA-A* adota o *design pattern Architecture Configurator* para oferecer os mecanismos de interceptação, encaminhamento e manipulação de requisições para aplicações. Com a utilização deste *pattern* o processo de adaptação das aplicações é facilitado devido à estrutura que é definida por ele. O *SDA-A* atende às necessidades de adaptação dinâmica das aplicações apresentadas no Capítulo 1, e.g., aplicações de computação autônoma e aplicações ubíquas.

Como opções para os desenvolvedores, o *SDA-A* oferece três tipos de gerenciamento:

- Um gerenciamento dirigido para aplicações usando o Java *J2SE;*
- Um gerenciamento dirigido para aplicações baseadas no padrão JMX;
- Um gerenciamento voltado para o servidor de aplicação JBoss.

Esse último gerenciamento oferece duas opções, uma voltada diretamente para as aplicações hospedadas no JBoss, e outra voltada para o gerenciamento dos próprios módulos

do servidor de aplicação. Com a implementação atual do suporte foi possível demonstrar a possibilidade de gerenciar aplicações no ambiente do JBoss através de primitivas definidas em um alto nível de abstração.

Foram apresentados alguns resultados de experimentos realizados para medir *overhead* e custos associados aos mecanismos utilizados na implementação do *SDA-A*. Ademais, o *overhead* gerado pelo *SDA-A* e os custos gerados pela inserção, remoção ou troca de componentes em uma aplicação, demonstram que ele atende aos requisitos de flexibilidade e desempenho para as aplicações modernas. Foi realizada uma comparação entre as formas de suporte oferecidas pelo *SDA-A*, concluindo-se que o suporte de gerenciamento de aplicações *Java J2SE* é o mais indicado para aplicações que necessitem de uma menor interferência em sua execução, já que foi o que apresentou o melhor desempenho nos gráficos experimentos realizados apresentados no Capítulo 4.

Para finalizar, a seguir são apresentados alguns pontos específicos para trabalhos futuros:

- O tratamento de questões de coordenação de adaptação em cenários altamente dinâmicos. Em alguns testes realizados utilizando o suporte para gerenciar aplicações de vídeo sob demanda em ambientes pervasivos (exemplo apresentado no Capítulo 4), deparamo-nos com problemas de perda e sincronização de frames de vídeo durante as adaptações da aplicação. Esses entraves não chegaram a perturbar significativamente a recepção do vídeo, mas poderiam ser inaceitáveis em algumas aplicações (e.g., em um cenário de tele-medicina, a realização de uma operação à distância). O trabalho [Ensink e Adve 2004] descreve mecanismos de sincronização, que (se provados viáveis em aplicações reais) poderiam ser integrados no *SDA-A* para obter um comportamento mais gracioso durante as adaptações.

- Atomicidade associada às transações de adaptação. Este ponto é importante para garantir adaptação consistente de aplicações em presença de falhas, incluindo as de comunicação. Para este caso é necessário implementar um mecanismo que possibilite desfazer todas as adaptações realizadas, até um dado momento, caso ocorra uma falha durante o processo de adaptação.

- Transferência dos estados dos componentes durante a adaptação. O tratamento deste ponto é importante para garantir os estados dos componentes de uma

aplicação durante uma adaptação. Neste caso é necessário desenvolver um mecanismo que armazene as informações do estado atual de um componente e o repasse para o novo componente que irá substituí-lo.

- Restrição de reconfiguração de portas. Uma marcação para permitir que as portas dos módulos ou conectores de uma arquitetura sejam reconfiguradas aumentaria a segurança nas reconfigurações. Para isso seria necessário implementar um mecanismo que identifique (através de uma tag, por exemplo) quais as portas que podem ser reconfiguradas.
- Identificação de reconfigurações inviáveis. Um mecanismo que identifique se a reconfiguração sugerida não gera problemas de *deadlocks*. Neste caso seria necessário implementar um suporte interno ao *SDA-A* que a medida que uma nova reconfiguração for imposta, o mesmo avalie se a mesma não gera nenhum tipo de *deadlocks*.

Apêndice A

Detalhes de Implementação

Neste Apêndice serão ilustrados os detalhes de implementação de uma aplicação que utiliza o suporte de adaptação oferecido pelo *SDA-A*. Para isso será utilizado um exemplo de uma aplicação Cliente/Servidor com um mecanismo de Log. O Log irá registrar todas as chamadas de serviços solicitadas por cada cliente envolvido na aplicação.

A.1 Descrição Arquitetural

A descrição arquitetural da aplicação Cliente/Servidor é apresentada no Código 14. A arquitetura descrita é composta por dois módulos Clientes (`cliente1 e cliente2`) linhas 03-05, um Servidor (`servidor`) linhas 06-08 e um Conector de Log (`cs`) linhas 09-12. Para cada um dos componentes (módulos e conectores) é definida a porta de entrada e saída (`servico`). No caso dos Clientes essa porta de saída é mapeada, no nível de implementação, para uma invocação de um método remoto (definida em uma interface remota) para um outro componente. Já para o Servidor a porta de entrada é mapeada para a interface de serviço publicada pelo mesmo. O Conector possui uma porta de entrada e saída (`serviço`) que são mapeadas como o caso dos Clientes e o Servidor.

```
01  module ClienteServidor {
02    port servico;
03    module Cliente {
04       out port servico;
05    } cliente1, cliente2;
06    module Servidor {
```

```
07       in port servico;
08    } servidor;
09    connector CS {
10       in port servico;
11       out port servico;
12    } cs;
13    instantiate servidor at localhost;
14    instantiate cliente1 at localhost; Cliente1
15    instantiate cliente2 at localhost; Cliente2
16    link cliente1 to servidor by cs;
17    link cliente2 to servidor by cs;
18 } cliente_servidor;
```

Código 14 – Descrição em CBabel da arquitetura de uma aplicação cliente-servidor

A partir da descrição arquitetural o Coordenador do *SDA-A* irá identificar a classe de cada componente da arquitetura (`Cliente`, `Servidor` e `CS`) linhas 3, 6 e 9. Logo em seguida ele irá identificar quais componentes deverão ser instanciados e seus respectivos locais de instanciação (linhas 13 – 15). Neste momento o Coordenador também identifica quais são os parâmetros de inicialização[4] de cada componente a ser instanciado. Esses parâmetros são informados após o *ponto e vírgula* (*;*) de cada linha de instanciação (linha 14 `Cliente1` e linha 15 `Cliente2`). E por fim o Coordenador irá identificar quais são as ligações dos componentes da arquitetura (linhas 16 e 17). Todas essas informações são armazenadas em uma estrutura de dados interna ao Coordenador. Durante as adaptações da aplicação o Coordenador irá consultar e/ou atualizar essa estrutura de dados.

A.2 Implementação das Classes da Aplicação

Durante o desenvolvimento das classes da aplicação o programador dos módulos poderá se concentrar diretamente nas funcionalidades dos módulos. Ou seja, não precisará preocupar-se com a localização dos serviços remotos e as formas de adaptar o comportamento da aplicação quando for necessário. Para o exemplo da aplicação Cliente/Servidor o programador irá desenvolver os módulos da aplicação tradicionalmente (Código 15, Código 16 e Código 17). Já o desenvolvimento do conector da aplicação requer que o programador redefina alguns métodos que serão utilizados *SDA-A* para adaptar a aplicação (Código 18).

```
01 public class Servidor implements ServidorInterface {
```

[4] Essas informações são armazenadas pelo Coordenador e serão utilizadas quando os componentes forem inicializados.

```
02   public Servidor(){ }
03   public String servico(){
04     return "Serviço Executado.";
05   }
06 }
```

Código 15 - Classe Servidor

O Código 15 apresenta a definição da classe do módulo Servidor. Essa classe implementa a interface (Código 16) publicada pelo Servidor (linha 01) a qual contém o método `servico`. Esse método é definido nas linhas 03-05 e seu retorno é a mensagem "Serviço Executado.".

```
01 public interface ServidorInterface {
02   public String servico();
03 }
```

Código 16 - Interface Servidor

De acordo com o Código 16 o método `servico` é a representação das portas de entrada e saída dos componentes da arquitetura. Essa interface é utilizada pelo módulo Cliente (Código 17, linha 02) para fazer referência ao serviço (porta de entrada) do módulo Servidor. De acordo com a classe Cliente o programador não precisa obter as referências para o módulo Servidor. A responsabilidade de obtenção das referências remotas fica a cargo do Executor do *SDA-A* durante a adaptação das classes da aplicação.

```
01 public class Cliente {
02  private ServidorInterface servidor; //Interface remota do Servidor
03  private Thread t;
04  public Cliente() {
05    t = (new Thread() {
06      public void run() {
07        try {
08          System.out.println("Inicio da execucao aguarde...");
09          for(i=1; i<=1000;i++)
10            servidor.servico();
11          System.out.println("Fim da execucao!!!");
12        } catch (Exception e) {e.printStackTrace();}
13      }
14    });
15  }
16  public void _init_ASAC(String[] args){
17    try{
18      t.start();
19    } catch (Exception e) {e.printStackTrace();}
20  }
21 }
```

Código 17 - Classe Cliente

O processo de adaptação das classes da aplicação é realizado automaticamente quando o Executor instancia os módulos da arquitetura. Desta forma o Executor irá procurar na classe uma declaração de uma referência para a interface remota a qual contêm a porta de saída (`servico`) definida na descrição arquitetura. Neste caso o Executor irá identificar a referência uma declaração apenas na classe Cliente, linha 02.

Depois de identificar as declarações o executor irá adicionar, automaticamente, o código de obtenção das referências para os objetos remotos. Esse código é inserido no início do método `_init_ASAC(String[] args)` independentemente dele ter sido ou não definido pelo programador. É importante lembrar que o método `_init_ASAC(String[] args)` é inserido em todas as classes da aplicação as quais não o definiram e ele é invocado automaticamente quando um componente é inicializado.

```
01 public class CS {
02  public Object _forward_ASAC(Object obj) {
03    return obj;
04  }
05  public Object _handle_ASAC(Object obj) {
06    Object rObj;
07    System.out.println("Log Antes."); //Pré-condições
08    rObj = _forward_ASAC(obj);        //Encadeamento
09    System.out.println("Log Depois.");//Pós-condições
08    return rObj;
09  }
10 }
```

Código 18 - Classe CS (Conector de Log)

A definição da classe do Conector exige que o programador redefina alguns métodos, padronizados do *SDA-A*. Neste caso o conector de Log CS da aplicação Cliente/Servidor (Código 18) redefine os métodos **`_forward_ASAC e _handle_ASAC`**. Ambos os métodos são as implementações dos mecanismos de interceptação e encaminhamento do *Design Pattern Architecture Configurator* (Capítulo 3, Seção 3.3). De acordo com o *Pattern* o programador deverá definir o método **`_handle_ASAC`** (do *SDA-A*) de forma que ele registre todas as invocações de serviços remotos que passam por ele. Para isso, de acordo com o *Pattern*, ele deverá executar as Pré-condições (linha 07), encaminhar a invocação para o próximo elemento da arquitetura através da invocação do método **`_forward_ASAC`** (linha 08) e realizar as pós-condições (linha 09). Como mencionado no Capítulo 3, Seção 3.5, o programador poderá definir os outros métodos padronizados do *SDA-A* embora não tenha sido ilustrado neste exemplo.

A.3 Execução da Aplicação

A partir da definição de todas as classes e interfaces remotas da aplicação o programador poderá implantar a aplicação através do *SDA-A*. Neste caso o programador deverá instanciar um Coordenador e um Executor (ambos no mesmo *host*). Feita as instanciações o programador poderá utilizar o terminal de linha de comando do *SDA-A* para interagir com o Coordenador. Para implantar a aplicação o programador poderá utilizar o

comando “`load cs.adl;`” para carregar a arquitetura da aplicação. Em seguida para colocar a aplicação em execução o programador poderá usar o comando “`start;`” para que a aplicação seja inicializada.

Apêndice B

Utilização do Javassist no SDA-A

A Classe Util do *SDA-A* é utilizada pelos Executores para realizar as modificações das classes desenvolvidas pelos programadores. É nessa classe que é feito o uso do Javassist para manipular os Java bytecodes das classes dos programadores. A seguir será apresentado o código fonte da classe Util.

B.1 Código Fonte da Classe Util

```
package configurator.Classes;


import configurator.utilities.TokenList;
import java.net.URL;
import java.util.Hashtable;
import java.lang.reflect.*;
import java.util.StringTokenizer;
import javassist.CtNewMethod;
import javassist.CtClass;
import javassist.SerialVersionUID;
import javassist.NotFoundException;
import javassist.CtMethod;
import javassist.CtField;
import javassist.ClassPool;

public class Util{

private static Hashtable listaClasses = new Hashtable();
private static Hashtable listaInterfaces = new Hashtable();

public static synchronized Object updateClass(String className, String tipoApp,
String ip, String nome){
        Object obj = null;
        CtMethod metodo;
        CtField field;
        CtNewMethod novoMetodo;
        CtClass ccMBean = null;
        CtClass interfaces[];
        CtClass cAdptor;
        CtClass cInitialContext;
        CtField cfIc;
```

```
        CtField cfServerJBoss;
        Boolean existMBean =false;
        obj = listaClasses.get(className);
        if(obj==null){
            try{

                ClassPool pool = ClassPool.getDefault();
                //className = className.replace('.',' ');
                //TokenList tok = new TokenList(className);
                //className = tok.GetToken(tok.TokenCount()-1);
                java.net.URL path = pool.find(className+"MBean");
                CtClass cc = pool.get(className);
                CtClass cModel = pool.get("configurator.Classes.Base");
                CtClass cString = pool.get("java.lang.String");
                CtClass cInteger = pool.get("java.lang.Integer");
                CtClass cObject = pool.get("java.lang.Object");
                CtClass cMethod = pool.get("java.lang.reflect.Method");
                CtClass cClass = pool.get("java.lang.Class");
                CtClass cInterfaceManager =
pool.get("configurator.InterfaceManager");
                CtClass cProxyConfigurator =
pool.get("configurator.ProxyConfigurator");
                CtClass cMBeanServerConnection =
pool.get("javax.management.MBeanServerConnection");
                CtClass cBaseInterface =
pool.get("configurator.Classes.BaseInterface");
                //Caso seja pra carregar no jboss entao é necessario adicionar
esses campos.
                cAdptor = pool.get("org.jboss.jmx.adaptor.rmi.RMIAdaptor");
                cInitialContext = pool.get("javax.naming.InitialContext");
                CtClass cClient = pool.get("org.jboss.remoting.Client");
                // Criacao do arquivo de interfaces MBean
                interfaces = cc.getInterfaces();
                ccMBean = null;
                if(interfaces!=null){
                    for (int i = 0; i<interfaces.length;i++){
                        System.out.println(interfaces[i].getName());
                        if
(interfaces[i].getName().equalsIgnoreCase(className+"MBean"))
                            ccMBean = interfaces[i];
                    }
                }
                if (ccMBean == null){
                    try{
                        ccMBean = pool.makeInterface(className+"MBean");
                    }catch(Exception e){
                        ccMBean = pool.get(className+"MBean");
                    }
                }else{
                    existMBean = true;
                }
                try{
                    metodo = ccMBean.getDeclaredMethod("_defineLink_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cBaseInterface.getDeclaredMethod("_defineLink_ASAC");
                    novoMetodo = new CtNewMethod();

ccMBean.addMethod(novoMetodo.copy(metodo,"_defineLink_ASAC",ccMBean,null));
                }

                try{

                    metodo = ccMBean.getDeclaredMethod("_handle_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cBaseInterface.getDeclaredMethod("_handle_ASAC");
                    novoMetodo = new CtNewMethod();

ccMBean.addMethod(novoMetodo.copy(metodo,"_handle_ASAC",ccMBean,null));
```

```
                }

                try{
                    metodo = ccMBean.getDeclaredMethod("_undefineLink_ASAC");
                }catch(NotFoundException nfe){
                    metodo =
cBaseInterface.getDeclaredMethod("_undefineLink_ASAC");
                    novoMetodo = new CtNewMethod();

ccMBean.addMethod(novoMetodo.copy(metodo,"_undefineLink_ASAC",ccMBean,null));
                }

                try{
                    metodo = ccMBean.getDeclaredMethod("_startInit_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cBaseInterface.getDeclaredMethod("_startInit_ASAC");
                    novoMetodo = new CtNewMethod();

ccMBean.addMethod(novoMetodo.copy(metodo,"_startInit_ASAC",ccMBean,null));
                }

                try{
                    metodo = ccMBean.getDeclaredMethod("_stop_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cBaseInterface.getDeclaredMethod("_stop_ASAC");
                    novoMetodo = new CtNewMethod();

ccMBean.addMethod(novoMetodo.copy(metodo,"_stop_ASAC",ccMBean,null));
                }

                try{
                    metodo = ccMBean.getDeclaredMethod("_block_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cBaseInterface.getDeclaredMethod("_block_ASAC");
                    novoMetodo = new CtNewMethod();

ccMBean.addMethod(novoMetodo.copy(metodo,"_block_ASAC",ccMBean,null));
                }
                ccMBean.writeFile("./Meta_File_ASAC");
                ccMBean.defrost();

                ccMBean = pool.get(className+"MBean");

                CtField cfNext = new CtField(cObject,"_next_ASAC",cc);
                CtField cfNextHost = new CtField(cString,"_nextHost_ASAC",cc);
                CtField cfFlag = new CtField(cInteger,"_flag_ASAC",cc);
                CtField cfIp = new CtField(cString,"_ip_ASAC",cc);
                CtField cfName = new CtField(cString,"_name_ASAC",cc);
                CtField cfNameNext = new CtField(cString,"_nameNext_ASAC",cc);
                CtField cfObjectClass = new CtField(cClass,
"_objectClass_ASAC",cc);
                CtField cfMetodo = new CtField(cMethod, "_metodo_ASAC",cc);
                CtField cfTipo = new CtField(cString, "_tipo_ASAC",cc);
                CtField cfMBSC = new CtField(cMBeanServerConnection,
"_mbsc_ASAC",cc);
                cfIc = new CtField(cInitialContext,"_ic_ASAC",cc);
                cfServerJBoss = new CtField(cAdptor,"_serverJBoss_ASAC",cc);
                CtField cfClient = new CtField(cClient,"_remotingClient_ASAC",cc);

                try{
                    metodo = cc.getDeclaredMethod("_init_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_init_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_init_ASAC",cc,null));
                }
                //Cria os proxys dinamicamente caso o metodo _defineProxy_ASAC nao
tenha cido definido
```

```
                try{
                    metodo = cc.getDeclaredMethod("_defineProxy_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_defineProxy_ASAC");
                    novoMetodo = new CtNewMethod();

cc.addMethod(novoMetodo.copy(metodo,"_defineProxy_ASAC",cc,null));
                    metodo = cc.getDeclaredMethod("_defineProxy_ASAC");
                    CtField[] atributos = cc.getDeclaredFields();
                    CtClass tipo;
                    String nomeAtributo, nomeClasse;
                    for(int x=0; x<atributos.length;x++){
                        tipo = atributos[x].getType();
                        if(tipo.isInterface()){
                            nomeClasse = tipo.getName();
                            nomeAtributo = atributos[x].getName();

metodo.addLocalVariable("_proxy_ASAC"+x,cProxyConfigurator);
                            metodo.insertBefore(" _proxy_ASAC"+x+" = new
configurator.ProxyConfigurator(this, \""+tipoApp+"\"); "+
                                    nomeAtributo +" = ("+nomeClasse+")
_proxy_ASAC"+x+".getProxy("+
                                    nomeClasse+".class);" );
                        }
                    }
                }


                CtField cfManager = new CtField(cInterfaceManager,
"_manager_ASAC",cc);

                try{
                    field = cc.getDeclaredField("_manager_ASAC");
                    cc.removeField(field);
                    cc.addField(cfManager);
                }catch(NotFoundException nfe){
                    cc.addField(cfManager);
                }

                try{
                    field = cc.getDeclaredField("_remotingClient_ASAC");
                    cc.removeField(field);
                    cc.addField(cfClient);
                }catch(NotFoundException nfe){
                    cc.addField(cfClient);
                }

                try{
                    field = cc.getDeclaredField("_tipo_ASAC");
                    cc.removeField(field);
                    cc.addField(cfTipo, CtField.Initializer.constant(tipoApp));
                }catch(NotFoundException nfe){
                    cc.addField(cfTipo, CtField.Initializer.constant(tipoApp));
                }

                try{
                    field = cc.getDeclaredField("_mbsc_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfMBSC);
                }


                try{
                    field = cc.getDeclaredField("_objectClass_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfObjectClass);
                }
```

```
                try{
                    field = cc.getDeclaredField("_metodo_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfMetodo);
                }
                try{
                    field = cc.getDeclaredField("_name_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfName);
                }
                try{
                    field = cc.getDeclaredField("_nameNext_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfNameNext);
                }
                try{
                    field = cc.getDeclaredField("_ip_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfIp, CtField.Initializer.constant(ip));
                }
                try{
                    field = cc.getDeclaredField("_next_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfNext);
                }
                try{
                    field = cc.getDeclaredField("_nextHost_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfNextHost);
                }
                try{
                    field = cc.getDeclaredField("_block_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(new CtField(CtClass.booleanType, "_block_ASAC",
cc),"new Boolean(false).booleanValue()");
                }
                try{
                    field = cc.getDeclaredField("_flag_ASAC");
                }catch(NotFoundException nfe){
                    cc.addField(cfFlag,"new Integer(0)");
                }

                try{
                    metodo = cc.getDeclaredMethod("_forward_ASAC");
                    cc.removeMethod(metodo);
                    metodo = cModel.getDeclaredMethod("_forward_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_forward_ASAC",cc,null));
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_forward_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_forward_ASAC",cc,null));
                }

                try{
                    metodo = cc.getDeclaredMethod("_setTipo_ASAC");
                    cc.removeMethod(metodo);
                    metodo = cModel.getDeclaredMethod("_setTipo_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_setTipo_ASAC",cc,null));
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_setTipo_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_setTipo_ASAC",cc,null));
                }

                try{
                    metodo = cc.getDeclaredMethod("_defineLink_ASAC");
```

```
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_defineLink_ASAC");
                    novoMetodo = new CtNewMethod();

cc.addMethod(novoMetodo.copy(metodo,"_defineLink_ASAC",cc,null));
                }
                try{
                    metodo = cc.getDeclaredMethod("_handle_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_handle_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_handle_ASAC",cc,null));
                }
                try{
                    metodo = cc.getDeclaredMethod("_undefineLink_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_undefineLink_ASAC");
                    novoMetodo = new CtNewMethod();

cc.addMethod(novoMetodo.copy(metodo,"_undefineLink_ASAC",cc,null));
                }
                try{
                    metodo = cc.getDeclaredMethod("_block_ASAC");
                    cc.removeMethod(metodo);
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_block_ASAC",cc,null));
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_block_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_block_ASAC",cc,null));
                }
                try{
                    metodo = cc.getDeclaredMethod("_stop_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_stop_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_stop_ASAC",cc,null));
                }
                try{
                    metodo = cc.getDeclaredMethod("_setNext_ASAC");
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_setNext_ASAC");
                    novoMetodo = new CtNewMethod();
                    cc.addMethod(novoMetodo.copy(metodo,"_setNext_ASAC",cc,null));
                }
                try{
                    metodo = cc.getDeclaredMethod("_startInit_ASAC");
                    cc.removeMethod(metodo);
                    novoMetodo = new CtNewMethod();

cc.addMethod(novoMetodo.copy(metodo,"_startInit_ASAC",cc,null));
                }catch(NotFoundException nfe){
                    metodo = cModel.getDeclaredMethod("_startInit_ASAC");
                    novoMetodo = new CtNewMethod();

cc.addMethod(novoMetodo.copy(metodo,"_startInit_ASAC",cc,null));
                }
                System.out.println("Vai adicionar interface");
                System.out.println(tipoApp);
                System.out.println(path);
                if(!existMBean) cc.addInterface(ccMBean);
                System.out.println("adicionou");
                ccMBean.writeFile("./Meta_File_ASAC");

                cc.defrost();
                SerialVersionUID.setSerialVersionUID(cc);
                cc.defrost();
                obj = cc.toClass().newInstance();
```

```
                listaClasses.put(className,obj);

            } catch(NotFoundException nfe){
                System.out.println(nfe.getMessage());
            } catch(Exception e){
                e.printStackTrace();
            }

        }else{
            try{
                obj = obj.getClass().newInstance();
                Class c = obj.getClass();
                Method m = c.getMethod("_setTipo_ASAC", new Class[]{String.class});
                m.invoke(obj, tipoApp);
            }catch(InstantiationException i){
                i.printStackTrace();
            }catch(IllegalAccessException ill){
                ill.printStackTrace();
            }catch(NoSuchMethodException nsme){
                nsme.printStackTrace();
            }catch(InvocationTargetException ite){
                ite.printStackTrace();
            }
        }
        return obj;
    }
```

Bibliografia

[Almeida et al. 2001] Almeida, J. P. A.; Wegdam, M.; van Sinderen, M.; Nieuwenhuis, L.; **Transparent dynamic reconfiguration for CORBA**; IEEE Computer Society Press in Proceeding of the 3rd International Symposium on Distributed Objects & Application (DOA 2001), 17-20 setembro, 2001, Roma, Itália.

[Ban 1998] Ban, Bela; **JavaGroups – Group Communication Partterns in Java**; Dept. of Computer Science, Cornell University, 31 julho, 1998

[Batista e Rodriguez 2000] Batista, T.; Rodriguez, N.; **Dynamic reconfiguration of component-based applications**; Proceedings of PDSE-2000, 32-39, Limerick, IEEE Computer Society, Ireland 10-11 Jjnho 2000.

[Batista et al. 2000] Batista, T.; Chavez, C. V. F.; Rodriguez, N.; **Conector Genérico: Um Mecanismo para Reconfiguração de Aplicações Baseadas em Componentes**; In Third Ibero-American Workshop on Software Envoriments and Requeriment Engineering (IDEAS'00), Cancun – Mexico, abril 2000.

[Batista e Rodriguez 2001] Batista, T. V.; Rodriguez, N.; **LuaSpace: Um Ambiente para Reconfiguração Dinâminca de Aplicações Baseadas em Componentes**; In Workshop de Teses e Dissertações no Simpósio Brasileiro de Computação (CTD - SBC2001) , pp 1-8, Fortaleza, CE, agosto 2001.

[Bidan et al. 1998] Bidan, C.; Issarny, V.; Saridakis, A.; **A dynamic reconfiguration service for CORBA**; in Proc. IEEE International Conference on Configurable Distributed Systems, maio, 1998.

[Bishop e Faria 1996] Bishop, J.; Faria, R.; **Connectors in Configuration Programming Languages: Are They Necessary?**; IEEE Third International Conference on Configurable Distributed Systems. Annapolis (EUA), maio 1996.

[Buschmann et al. 1996] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.; **A System of Patterns – Pattern-Oriented Software Architecture**. Wiley & Sons, New York (EUA), 1996.

[Burke e Bock 2004] Burke, B.; Brock, A.; **Aspect Oriented Programming with JBoss 4**; Disponível na Internet. http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop em 22 outubro, 2004.

[Capra et al. 2005] Capra, L.; Zachariadis, S.; Mascolo, C.; **Q-CAD: QoS and Context Aware Discovery Framework for Mobile Systems**; In Proc. of International Conference on Pervasive Services (ICPS'05). Santorini, Greece. To appear. July 2005.

[Cardoso 2005] Cardoso, L. X. T. **Integração de Plataformas de Monitoração no Contexto de Arquiteturas Adaptáveis de Software**. Dissertação de mestrado em andamento, Instituto de Computação – Universidade Federal Fluminense (IC/UFF), 2005.

[Carvalho et al. 2002] Carvalho, S.; Lisboa, J.; Loques, O.; **Um Design Pattern para Configuração de Arquiteturas de Software**; The Second Latin American Conference on

Pattern Languages of Programming, SugarLoafPLoP 2002 Conference, Itaipava, agosto, 2002.

[Chiba e Nishizawa 2003] Chiba, S.; Nishizawa, M.; **An Easy-to-Use Toolkit for Efficient Java Bytecode Translators**; Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.364-376, Springer-Verlag, 2003.

[Corradi 2005] Corradi, A. M.; **Um Framework de Suporte a Requisitos Não-Funcionais para Serviços de Nível-Alto**; Dissertação de Mestrado, Instituto de Computação (IC), UFF, agosto, 2005.

[Dowling e Cahill 2001] Dowling, J.; Cahill, V.; **The K-Component architecture meta-model for self-adaptive software**; Proc. Reflection 2001, LNCS 2192.

[Ensink e Adve 2004] Ensink, B.; Adve, V.; **Coordinating Adaptations in Distributed Systems**; In Proceedings of the 24th International Conference on Distributed Computing Systems (Icdcs' 04); ICDCS; IEEE Computing Society, Washington, DC, 446-455; 24 – 26 março 2004.

[Fleury e Reverbel 2003] Fleury, M.; Reverbel, F.; **The JBoss Extensible Server**. In: International Middleware Conference, 2003, Rio de Janeiro. Resumos. Rio de Janeiro, 2003. p. 344-373.

[Freitas 2005] Freitas, G.; H.; **Um experimento de uso de um framework de suporte a requisitos não-funcionais de qualidade**; Dissertação de Mestrado, Instituto de Computação (IC), UFF, agosto, 2005.

[Freitas et al. 2005] Freitas, G.; Cardoso, L.; Santos, A. L.; Loques, O.; **Otimizando a Utilização de Servidores através de Contratos Arquiteturais**, VII Workshop de Tempo Real - XXIII Simpósio Brasileiro de Redes de Computadores, Fortaleza, maio 2005.

[Keeney e Cahill 2003] Keeney, J.; Cahill, V.; **Chisel: a policy-driven, context-aware, dynamic adaptation framework**; Dept. of Comput. Sci., Trinity Coll., Dublin, Ireland; Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop.

[de Lemos 2006] de Lemos, R.; **Architectural reconfiguration using coordinated atomic actions**; In Proceedings of the 2006 international Workshop on Self-Adaptation and Self-Managing Systems, SEAMS '06. ACM Press, New York, NY, 44-50, Shanghai, China, 21 – 22 maio, 2006.

[Loques et al. 1999] Loques, O.; Leite, J.; Lobosco, M.; Sztajnberg, A.; **Integrating Meta-Level Programming and Configuration Programming**; Workshop on Object Oriented Reflection and Software Engineering. OOPSLA'99, Denver (EUA), novembro 1999

[Loques et al. 2004] Loques, O.; Curty, R.; Ansaloni, S.; Sztajnberg, A.; **A Contract-Based Approach to Describe and Deploy Non-Functional Adaptations in Software Architectures**; Journal of the Brazilian Computer Society, V.10, N.1, julho 2004.

[Medvidovic 1999] Medvidovic, N.; **Architecture-Based Specification-Time Software Evolution**; Tese de Doutorado (PhD). University of California, Irvine (EUA), 1999.

[Moura et al. 2002] Moura, A. L.; Ururahy, C.; Cerqueira, R.; Rodriguez, N.; **Dynamic Suport for Distributed Auto-Adaptive Applications**; Proceedings of AOPDCS'02: Workshop on Aspect Oriented Programming for Distributed Computing Systems (held in conjunction with IEEE ICDCS 2002); pp. 451-456, Vienna, Austria, 2002.

[Nicoara e Alonso 2005] Nicoara, A.; Alonso, G.; **Dynamic AOP with PROSE**; International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) in conjunction with CAISE'05, Porto, 14 jnho 2005.

[OMG 2000] Object Management Group; **Trading Object Service Specification**; OMG Document formal/00-06-27, maio 2000.

[Papadopoulos e Arbab 2001] Papadopoulos, G. A.; Arbab, F.; **Configuration and dynamic reconfiguration of components using the coordination paradigm**; Future Generation Computer Systems, 17(8):1023-1038, junho 2001.

[Perry 1992] Perry, D.; Wolf, A.; **Foundations for the Study of Software Architecture**; ACM Software Engineering Notes, 17(4), pp. 40-52, outubro 1992.

[Ranganathan et al. 2005] Ranganathan, A.; Chetan, S.; Al-Muhtadi, J.; Campbell, R. H.; Mickunas, M. D.; **Olympus: A High-Level Programming Model for Pervasive Computing Environments**; In IEEE International Conference on Pervasive Computing and Communications (PerCom 2005), Kauai Island, Hawaii, 8-12 de março 2005.

[Román et al. 2002] Román, M.; Hess, C. K.; Cerqueira, R.; Ranganathan, A.; Campbell, R. H.; Nahrstedt, K.; **Gaia: A Middleware Infrastructure to Enable Active Spaces**; In IEEE Pervasive Computing, pp. 74-83, outubro-dezembro 2002.

[Santos et al. 2006] Santos, A. L. G.; Leal, D.; Loques, O.; **Um Suporte para Adaptação Dinâmica de Arquiteturas Ubíquas;** Conferência Latinoamericana de Informática (CLEI), Santiago, Chile, 21 – 25 agosto 2006.

[Shaw et al. 1995] Shaw, M.; DeLine, R.; Klein, D.; Ross, T.; Young, D.; Zelesnik G.; **Abstractions for Software Architecture and Tools to Support Then**; IEEE Transactions on Software Engineering, vol. 21, nr. 4, pp. 314-335, 1995.

[Shaw e Garlan 1996] Shaw, M.; Garlan, D.; **Software Architecture: Perspectives on an Emerging Discipline**; Prentice Hall, Upper Saddle River (EUA), 1996.

[Shaw et al. 1996] Shaw, M.; Deline, R.; Zelenisk, G.; **Abstractions and Implementations for Architectural Connections**; Third International Conference on Configurable Distributed Systems, maio 1996.

[Stark 2003] Stark, S.; The Jboss Group; **JBoss Administration and Development Third Edition (3.2.x Series)**; Atlanta, 2003.

[Sun 2005] **Sun MicroSystems**; **Java Management Extensions**; Disponível na Internet; Acessado em 02/06/2005 http://java.sun.com/products/JavaManagement

[Sztajnberg 2002] Sztajnberg, A.; **Flexibilidade e Separação de Interesses para a Concepção e Evolução de Aplicações Distribuídas**; Tese de Doutorado, COPPE/PEE/UFRJ, maio, 2002.

[Tarr et al. 1999] Tarr, P.; Ossher, H.; Harrison, W.; SM Sutton, Jr.; **N degrees of Separation: Multidimensional separation of concerns**; Proc. ICSE 99, IEEE, ACM press, pp. 107-119, Las Angeles, maio 1999.