Adaptação Dinâmica de Aplicações Distribuídas

Ana Lúcia de Moura Noemi Rodriguez
Departamento de Informática – PUC-Rio
Rua M. S. Vicente 225, CEP 22453-900, Rio de Janeiro
ana,noemi@inf.puc-rio.br

Resumo

Este trabalho descreve um ambiente que suporta a adaptação dinâmica de aplicações distribuídas às características não funcionais de seus componentes. O ambiente, baseado na linguagem de programação Lua, permite que as aplicações selecionem, dinamicamente, os componentes adequados a seus requisitos, verifiquem o atendimento desses requisitos ao longo do tempo, e reajam, quando necessário, a alterações nas propriedades não funcionais dos serviços utilizados. A biblioteca *LuaTrading*, que facilita a interação com o serviço de *Trading* CORBA, suporta a seleção dinâmica de componentes. A facilidade de monitoração extensível permite que as aplicações realizem a verificação de seus requisitos. O mecanismo de *proxies inteligentes* é usado para implementar a adaptação a alterações de propriedades não funcionais. O trabalho descreve ainda um exemplo de uso desses mecanismos.

Abstract

This work presents an environment that allows distributed applications to dynamically adapt to the non-functional properties of their components. The environment, based on the programming language Lua, allows applications to dinamically select the components that best suit their requirements, verify if these requirements are being met, and react, when appropriate, to variations in the non-functional properties of the services in use. The *LuaTrading* library, which provides a simplified interface to CORBA's Trading Service, supports dynamic component selection. The extensible monitoring facility supports monitoring of dynamically defined requirements. Smart proxies are used to implement the adaptation strategies. The paper also describes an example application based on the proposed environment.

palavras chave: middleware, adaptação dinâmica, CORBA, componentes, requisitos não funcionais

1 Introdução

A programação baseada em componentes constitui hoje uma importante tendência no desenvolvimento de aplicações distribuídas. Uma das principais características de sistemas de componentes é a rígida separação entre interface e implementação. Os componentes de uma aplicação distribuída implementam serviços específicos, oferecidos através de *interfaces funcionais*. Sistemas de componentes (ou plataformas de *middleware*), como CORBA, DCOM e JavaBeans, oferecem as abstrações e facilidades necessárias para a definição dessas interfaces e para a invocação de suas operações. Tradicionalmente, questões relacionadas à heterogeneidade e, de forma geral, a detalhes específicos do ambiente de

execução, são tratados internamente às plataformas de *middleware* e escondidos das aplicações. Contudo, com a crescente demanda por aplicações multimídia, sistemas colaborativos e aplicações de tempo real, passou a ser importante que as próprias aplicações levem também em consideração requisitos *não funcionais* como desempenho, disponibilidade, e segurança. Como as condições relacionadas a esses requisitos tendem a apresentar variações ao longo do tempo, as aplicações devem poder se adaptar dinamicamente a alterações em seu ambiente de execução.

Nesse contexto, vários trabalhos vêm propondo extensões às plataformas de *middleware*, mais especificamente à plataforma CORBA, que suportam a adaptação dinâmica de aplicações distribuídas. O conceito de *Qualidade de Serviço (QoS)* vem sendo usado por esses trabalhos em um sentido mais amplo do que o usual, em referência a quaisquer características, ou propriedades, apresentadas pelo ambiente de execução de uma aplicação. As extensões propostas destinam-se então a permitir que aplicações distribuídas possam se adaptar a variações na *qualidade do serviço* oferecido por seus componentes [ZBS97, BG97, KK98].

A forma de especificar a reação de uma aplicação a alterações nas condições de seu ambiente é uma questão importante. Essa reação pode variar não apenas de aplicação para aplicação como também ao longo do tempo. Novos serviços (com novas interfaces, por exemplo) podem se tornar disponíveis para substituir um componente que não esteja satisfazendo os requisitos de uma aplicação. O mecanismo de adaptação deve ser flexível a ponto de incorporar tais variações [AW99].

Neste trabalho, abordamos o problema de evolução dinâmica de condições de execução e de aplicações no contexto de um projeto de pesquisa, em desenvolvimento há alguns anos na PUC-Rio, que investiga a flexibilidade que uma linguagem interpretada pode conferir à programação baseada em componentes. Um dos principais resultados desse projeto é o sistema LuaOrb [Cer00], uma implementação de um mecanismo de composição dinâmica entre sistemas de componentes. A implementação de *bindings* dinâmicos entre a linguagem Lua [IFC96, FIC96] e os sistemas CORBA, COM e Java permite que componentes de quaisquer desses sistemas sejam incorporados a uma aplicação em tempo de execução.

Temos explorado a flexibilidade oferecida pela linguagem Lua e pelo sistema LuaOrb na implementação de diversos mecanismos de configuração dinâmica de aplicações distribuídas baseadas em componentes [RI99, BCR00]. Estendendo essa investigação, o objetivo do ambiente proposto neste trabalho é a provisão de um conjunto de mecanismos que suportem a adaptação dinâmica de aplicações distribuídas à qualidade do serviço oferecido por seus componentes. As facilidades implementadas por esses mecanismos permitem que as aplicações selecionem, dinamicamente, os componentes adequados ao atendimento de seus requisitos, verifiquem o atendimento desses requisitos ao longo do tempo, e reajam, quando necessário, a alterações na qualidade do serviço oferecido pelos componentes utilizados.

O restante desse trabalho é organizado da seguinte forma. A próxima seção oferece uma visão geral do *binding* LuaCORBA. A seção 3 descreve a biblioteca *LuaTrading*, que oferece uma interface simplificada para o serviço de *Trading* CORBA, um elemento de grande importância no contexto de adaptação dinâmica de aplicações. Em seguida, a seção 4 apresenta a arquitetura do ambiente proposto neste trabalho, e descreve os mecanismos por ele oferecidos. A seção 5 descreve um exemplo de uso do ambiente proposto. Finalmente, a seção 6 apresenta algumas considerações sobre o trabalho.

2 LuaCORBA

LuaCORBA é a parte do sistema LuaOrb responsável pelo *binding* entre CORBA e a linguagem Lua, uma linguagem interpretada desenvolvida na PUC-Rio. Lua é uma linguagem de *extensão*, implementada como uma biblioteca. Através de sua API, é possível chamar funções Lua a partir de código C, e também registrar funções C para serem chamadas por código Lua.

O padrão CORBA [Gro99] define um *Object Request Broker* (ORB) que é responsável pela comunicação entre objetos clientes e servidores distribuídos. A interface de cada objeto servidor é descrita em IDL, uma sintaxe independente de tecnologia. Tipicamente, essas descrições são compiladas para gerar *stubs* para clientes e *esqueletos* para servidores. O programa cliente pode ser simplesmente compilado e ligado com o *stub*. O programa servidor deve implementar os métodos declarados no esqueleto.

Essa abordagem, usada na maior parte dos *bindings* de CORBA (em especial, nos mais utilizados, para Java e C++), requer que o cliente seja recompilado a cada alteração na interface do servidor ou a cada vez que um novo tipo de servidor deve ser acessado. A arquitetura CORBA oferece dois mecanismos para evitar essa necessidade de recompilação: a *Interface de Invocação Dinâmica* (DII) para invocação de qualquer operação com uma lista de argumentos definida em tempo de execução, e a *Interface de Esqueleto Dinâmico* (DSI), para escrever implementações de objetos que não conhecem, em tempo de compilação, as interfaces que irão implementar.

LuaCORBA se baseia nas interfaces DII e DSI e nas características reflexivas de CORBA, e nas características dinâmicas e reflexivas de Lua, para criar um *binding* dinâmico, tanto para clientes como para servidores [ICR98, CCI99, Cer00].

Lua é uma linguagem tipada dinamicamente, com verificação de tipos ocorrendo em tempo de execução. Não existem declarações de tipos de variáveis e funções. Objetos em Lua (implementados por *tabelas*) não estão associados a classes; cada objeto pode ter quaisquer métodos e variáveis de instância.

Um cliente CORBA escrito em Lua utiliza um serviço CORBA da mesma forma que utiliza qualquer objeto Lua: sem declarações e com verificação dinâmica. Para isso, o *binding* cliente utiliza a DII, e representa objetos CORBA em programas Lua através de *proxies*.

Um proxy é um objeto comum de Lua; LuaCORBA utiliza *tag methods*, um mecanismo reflexivo de Lua, para alterar o comportamento default desse objeto. Quando o programa Lua faz uma chamada a um método do *proxy*, o *tag method* intercepta a chamada e a redireciona para LuaCORBA, que dinamicamente mapeia tipos de parâmetros de Lua para IDL, invoca o método remoto e mapeia eventuais resultados de volta para Lua.

Ilustraremos a seguir o uso de proxies LuaCORBA com base na interface IDL definida na figura 1

```
struct book {
  string author;
  string title; };
interface foo {
  boolean add_book(book abook); ...  };
```

Figura 1: Exemplo de interface IDL

Para criar um proxy de um objeto remoto que implementa a interface `foo`, usamos

o construtor `createproxy`, fornecendo o nome da interface do objeto remoto e uma referência (IOR) para ele. Depois que o proxy foi criado, os serviços do objeto correspondente podem ser requisitados conforme mostrado na figura 2.

```
a_foo = createproxy{interface="foo",ior=an_ior_string}
a_book = {author="Mr. X", title="New Book"}
a_foo:add_book(a_book)
```

Figura 2: Acesso a um objeto CORBA através de um proxy LuaCORBA

Do lado do servidor, o *binding* LuaCORBA faz uso da interface DSI para permitir que objetos Lua sejam usados como servidores CORBA. Resumidamente, a idéia da DSI é implementar todas as chamadas a um determinado objeto (chamado aqui de um *servidor dinâmico*) através de chamadas a uma única rotina, a *rotina de implementação dinâmica* (DIR). Essa rotina é responsável por desempacotar os argumentos recebidos e por ativar o código apropriado.

Cada objeto Lua que age como servidor CORBA é representado por um *adaptador* LuaCORBA, que é um objeto C++. Esse objeto trata todas as requisições destinadas ao objeto Lua, implementando a rotina DIR. O código ativado pela DIR é o método Lua correspondente à operação invocada.

```
interface listener {
  oneway void put(long i); };
interface generator {
  void set_listener(listener r); };
```

Figura 3: Interfaces IDL `generator` e `listener`

A interface IDL apresentada na figura 3 é usada a seguir em um exemplo muito simples de um servidor Lua. Nesse exemplo, o objeto `generator` gera números (aleatórios, por exemplo) e para cada número gerado chama o método `put` de seu objeto `listener`.

Podemos escrever um *listener* em LuaCORBA como mostrado na figura 4. Na primeira atribuição, criamos um objeto Lua com um único método, chamado `put`. Em seguida, criamos um proxy para o gerador, e passamos o objeto Lua criado anteriormente como argumento para `set_listener`. Quando LuaCORBA detecta que o parâmetro formal é um objeto CORBA `listener`, e o argumento real um objeto Lua, o sistema automaticamente cria um adaptador para esse objeto, permitindo que funcione como um servidor CORBA.

```
l = { put = function (self, i) print(i) end }

gen = createproxy{interface="generator", ior=gen_ior}
gen:set_listener(l)
```

Figura 4: Implementação de um objeto CORBA em Lua

A forma de criação de servidor CORBA ilustrada acima é denominada *implícita*, uma vez que o simples uso do objeto Lua como argumento determina a criação do servidor.

LuaCORBA também dá suporte à criação explícita de servidores CORBA através da função `CreateDSIServer`.

3 LuaTrading

O serviço de *Trading* definido pela arquitetura CORBA [Gro98, HV99] é um elemento importante em ambientes de suporte à adaptação dinâmica de aplicações distribuídas. Interagindo com um *trader*, clientes de uma interface funcional obtêm a localização das implementações dessa interface mais adequadas a seus requisitos, selecionando-as com base nas características, ou propriedades *não funcionais*, oferecidas por essas implementações.

A biblioteca LuaTrading, descrita nesta seção, foi desenvolvida com o objetivo de simplificar a programação das interações básicas com o serviço de *Trading*, facilitando a implementação de mecanismos de suporte à seleção e localização dinâmica de componentes servidores.

3.1 O Serviço de *Trading* CORBA

Um objeto *trader* CORBA pode ser visto, basicamente, como um repositório de *ofertas de serviço*. Uma oferta de serviço, além de conter uma referência para um provedor de uma determinada interface IDL, descreve características, ou atributos, não funcionais desse provedor através de um conjunto de pares "nome-valor" denominados *propriedades*. A interface IDL e o conjunto de propriedades que caracterizam o serviço oferecido são determinados pelo *tipo de serviço* ao qual a oferta é associada.

Descrições de tipos de serviço são armazenadas em um *Repositório de Tipos de Serviço* mantido pelo *trader*, que disponibiliza operações que permitem criar, examinar e remover descrições de tipos de serviço. Essas descrições consistem de um tipo de interface IDL, uma lista de definições de propriedades e uma lista de *super tipos* (cujas definições são herdadas pelo tipo de serviço descrito). Por sua vez, a definição de uma propriedade especifica seu nome, modo, e o tipo de seu valor.

Objetos servidores divulgam suas ofertas de serviço interagindo com um *trader* através de operações de *exportação*. Ao exportar uma oferta de serviço, um objeto servidor fornece ao *trader* o tipo de serviço associado, a referência para a implementação da interface IDL correspondente, e os valores das propriedades oferecidas por essa implementação. Além da operação de exportação, o objeto *trader* provê operações que permitem a modificação e a remoção de ofertas de serviço.

A *importação* de ofertas de serviço, isto é, a seleção de provedores de um tipo de serviço com base em um conjunto de requisitos, é disponibilizada pelo *trader* através da operação `query`. Nessa operação, a especificação de requisitos é realizada através de uma linguagem (*Trader Constraint Language*) utilizada na construção de uma expressão booleana que define restrições aplicadas sobre os valores das propriedades das ofertas disponíveis. O serviço de *Trading* permite também a um importador determinar o critério de ordenação das ofertas selecionadas, o subconjunto de propriedades retornado nessas ofertas, e as políticas adotadas na execução da operação de importação (como, por exemplo, o número máximo de ofertas pesquisadas, ordenadas e retornadas).

A programação das interações com o serviço de *Trading* é uma tarefa razoavelmente complexa. A biblioteca LuaTrading, descrita a seguir, tem como objetivo facilitar essa tarefa, oferecendo um conjunto de objetos e funções que permitem acessar esse serviço de forma mais simples e natural. Por restrições de espaço, apresentamos as facilidades

oferecidas por LuaTrading de forma bastante resumida. Uma descrição mais detalhada é encontrada em [Mou00].

3.2 Acesso ao Repositório de Tipos de Serviço

LuaTrading oferece acesso ao Repositório de Tipos de Serviço através de um objeto Lua que representa, localmente, esse repositório remoto. As operações aplicadas sobre o objeto Lua são mapeadas, automaticamente, em requisições ao serviço de *Trading*, para que operações equivalentes sejam aplicadas sobre o repositório por ele mantido.

Na figura 5 mostramos como a biblioteca LuaTrading pode ser utilizada para definir um novo tipo de serviço. Nesse exemplo, o tipo de serviço `servicoX` será associado à interface funcional identificada por `IDL:ServicoX:1.0`. As ofertas de serviço associadas a esse tipo serão caracterizadas pelas propriedades `disp`, do tipo *string*, e `tresp`, do tipo *float*, que representam, respectivamente, a disponibilidade do objeto provedor (`alta` ou `baixa`) e o tempo médio de resposta às solicitações de serviço. A propriedade `disp` deverá estar obrigatoriamente presente nas ofertas de serviço exportadas (modo `Mandatory`).

```
-- cria um objeto Lua para representar o repositorio remoto
typesRep = serviceTypesRep()

-- define um novo tipo de servico no repositorio
typesRep.servicetypes.servicoX = {
          if_name="IDL:ServicoX:1.0"      ,
          props={{name="disp", type=CORBA_string, mode="PROP_MANDATORY"},
                 {name="tresp", type=CORBA_float, mode="PROP_NORMAL"}},
          super_types={} }
```

Figura 5: Exemplo de definição de um tipo de serviço

Além da criação de novos tipos de serviço, LuaTrading oferece acesso às descrições de tipos de serviço armazenadas no repositório remoto, e permite eliminar descrições desse repositório.

3.3 Gerência de Ofertas de Serviços

LuaTrading provê facilidades para a divulgação, modificação e remoção de ofertas de serviços através de um objeto Lua que representa, localmente, o repósitorio de ofertas mantido por um *trader*.

Na figura 6 mostramos como uma oferta de serviço do tipo `servicoX` (criado no exemplo anterior) pode ser exportada por um *script* Lua que utiliza a biblioteca LuaTrading. A oferta exportada indica que a implementação do serviço apresenta alta disponibilidade e tempo de resposta médio de 1 segundo.

Um conceito importante definido pelo serviço de *Trading* é o de *propriedade dinâmica*. Ao invés de armazenar um valor constante, uma propriedade dinâmica armazena uma referência para um objeto responsável por fornecer ao *trader*, quando necessário, o valor corrente da propriedade. Propriedades dinâmicas permitem refletir condições de execução que evoluem dinamicamente e têm, por isso, um papel importante em nosso trabalho.

Para facilitar a exportação de ofertas de serviço que contêm propriedades dinâmicas, a biblioteca LuaTrading disponibiliza um construtor para essas propriedades: o método `createDynProp`. Esse método recebe como parâmetros o tipo de serviço que define a

```
-- cria o objeto Lua que representa o repositorio de ofertas
offersRep = serviceOffersRep()

-- exporta a oferta de servico do servidor srvobj
offer_id = offersRep:export{ server = srvobj, type = "servicoX",
                             properties = { disp = "alta",tresp = 1.0 } }
```

Figura 6: Exemplo de exportação de uma oferta de serviço

propriedade, o nome da propriedade, e a referência para o objeto a ser consultado para a obtenção de seu valor.

Além da *exportação* de ofertas de serviço, o gerenciamento de um conjunto de servidores em um ambiente dinâmico exige a manutenção das ofertas de serviço a eles associadas. Operações comuns nesse tipo de ambiente são, por exemplo, a remoção das ofertas de serviço de um determinado tipo, ou de um determinado servidor, ou ainda a modificação de uma oferta de serviço quando o valor de uma ou mais propriedades for alterado. A interface do serviço de *Trading* CORBA oferece operações que permitem a remoção e a modificação de uma oferta de serviço individual, especificada por seu identificador (retornado pela operação de exportação). Contudo, essa interface não provê suporte direto à remoção ou modificação de ofertas de serviço com base no tipo de serviço associado, e/ou no componente prestador desse serviço.

Para simplificar a gerência de ofertas de serviço, LuaTrading mantém um repositório local de ofertas e oferece os métodos `removeServer`, `removeService` e `modify`. Esses métodos suportam a remoção e a modificação de ofertas de serviço associadas a um determinado servidor, tipo de serviço, ou ambos, conforme é mostrado na figura 7.

É importante observar que as operações realizadas sobre o repositório local são automaticamente mapeadas em requisições ao *trader* para que operações equivalentes sejam aplicadas sobre o repositório remoto.

```
-- remocao das ofertas de servico de um servidor
offersRep:removeServer(srvobj)

-- remocao das ofertas do tipo servicoX de um servidor
offersRep:removeService{ type="servicoX", server=srvobj }

-- modificacao de uma oferta de servico
offersRep:modify{ type="servicoX", server=srvobj,
                  properties = { deleted = {"tresp"},
                                 updated = {disp = "baixa} }
```

Figura 7: Exemplo de remoção e modificação de ofertas de serviço

3.4 Importação de Ofertas de Serviços

A complexidade na programação de importações de ofertas de serviço baseada na interface oferecida pelo serviço de *Trading* é devida a dois fatores principais. O primeiro deles é o grande número de parâmetros definidos nessa interface, alguns deles irrelevantes para a realização de consultas simples. O segundo é a necessidade de acesso a uma interface

adicional: a operação básica de importação, `query`, retorna apenas algumas das ofertas selecionadas, e a referência para um objeto iterador ao qual deverão ser requisitadas as demais ofertas.

Para simplificar a programação de importações de ofertas de serviço, a biblioteca LuaTrading disponibiliza a função `importServiceOffers`. Essa função utiliza valores `default` para a maioria dos parâmetros definidos pela operação `query`, evitando a necessidade de sua especificação na realização de consultas simples. Além disso, a função `importServiceOffers` encapsula as interações com a interface `OfferIterator` e oferece como seu retorno a lista completa de ofertas de serviço solicitadas ao *trader*.

A função `importServiceOffers` recebe como parâmetro uma tabela que contém as informações necessárias para a realização de uma operação de importação de ofertas de serviço. Apenas o campo `type`, que especifica o tipo de serviço procurado, tem presença obrigatória nessa tabela; todos os demais campos possuem valores `default` e podem ser omitidos na chamada da função.

O campo `constraint` contém a expressão que determina os critérios para seleção de ofertas de serviço, isto é, as restrições aplicadas sobre os valores das propriedades. O campo `pref`, se presente, determina a ordenação das ofertas retornadas. O campo `maxret` indica o número máximo de ofertas de serviço desejado pelo importador, e corresponde a uma das políticas de importação aplicáveis sobre uma consulta ao *trader* (`return_card`). A seleção das propriedades cujos valores deverão ser retornados nas ofertas de serviço é realizada através do campo `properties`, cujo valor `default` é a string `none`, indicando que apenas as referências para os objetos servidores deverão ser retornadas.

No exemplo mostrado na figura 8 realizamos uma consulta ao *trader* para obter referências para servidores que provêem o tipo de servico `servicoX`, definido anteriormente, e oferecem alta disponibilidade e tempo de resposta não superior a 2 segundos. As ofertas de serviço selecionadas devem ser ordenadas crescentemente, de acordo com o tempo de resposta desses servidores (propriedade `tresp`). No máximo três ofertas deverão ser retornadas.

```
offers = importServiceOffers{ type = "servicoX",
                              constraint = "disp == 'alta' and tresp <= 2",
                              pref = "min tresp", maxret = 3 }
```

Figura 8: Exemplo de importação de ofertas de serviço

4 Ambiente de Suporte à Adaptação Dinâmica de Aplicações Distribuídas

O objetivo do ambiente proposto neste trabalho é suportar a adaptação dinâmica de aplicações distribuídas à qualidade do serviço oferecido por seus componentes. Para isso, o ambiente oferece facilidades e mecanismos que permitem que as aplicações:

- selecionem, dinamicamente, os componentes adequados ao atendimento de seus requisitos, expressos como um conjunto de *propriedades não funcionais* que caracterizam a qualidade do serviço oferecido por esses componentes.
- verifiquem o atendimento de seus requisitos ao longo do tempo através da monitoração das propriedades associadas.

- reajam a alterações relevantes nessas propriedades através do acionamento de estratégias de adaptação apropriadas.

Para introduzir essas facilidades de forma transparente ao comportamento funcional das aplicações, nosso ambiente emprega o conceito de *proxy inteligente*. Esse mecanismo oferece uma abstração através da qual as aplicações tem acesso não a componentes específicos, mas a representações de *tipos de serviço* que incorporam o tratamento transparente dos requisitos *não funcionais* demandados aos serviços representados. O uso dessa abstração permite que os componentes adequados aos requisitos de uma aplicação sejam selecionados dinamicamente, e possibilita a implementação de diferentes estratégias de adaptação.

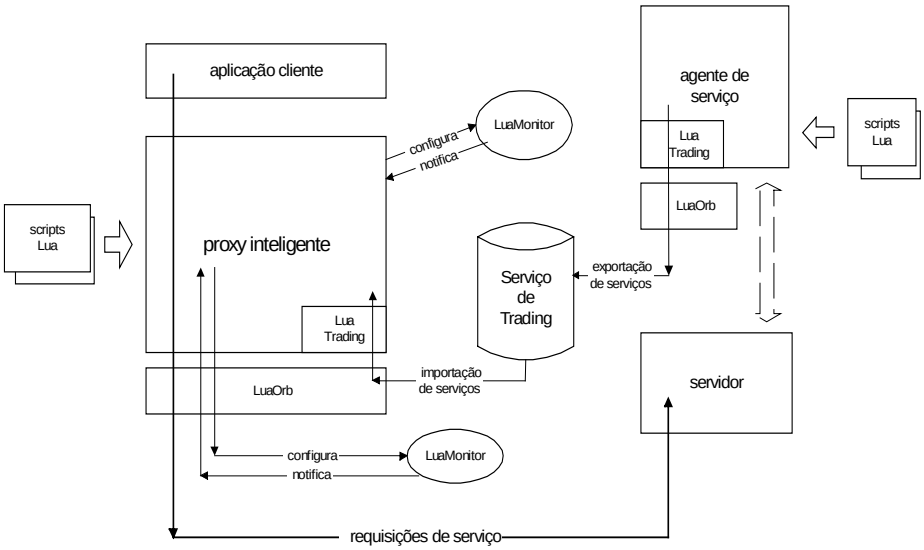

Figura 9: Ambiente de suporte à adaptação dinâmica de aplicações distribuídas

A seleção dinâmica dos componentes adequados aos requisitos de uma aplicação é realizada por um proxy inteligente através de consultas ao serviço de *Trading* CORBA, usando a biblioteca *LuaTrading* descrita na seção anterior.

Além de selecionar dinamicamente os componentes adequados aos requisitos de uma aplicação, um proxy inteligente deve ser capaz de verificar, e garantir, o atendimento a esses requisitos ao longo do tempo. Para isso, é essencial a provisão de um mecanismo que permita a monitoração das propriedades que caracterizam a qualidade do serviço representado. Dado que estamos interessados em dar suporte às necessidades específicas de qualquer aplicação, é importante que esse mecanismo seja configurável e extensível. O mecanismo de monitoração provido por nosso ambiente, denominado LuaMonitor, é descrito na seção 4.1.

Para permitir o uso do serviço de *Trading* na localização dos objetos servidores adequados aos requisitos de uma aplicação, introduzimos também em nosso ambiente a figura de elementos responsáveis pela divulgação de ofertas de serviços a um *trader*. Denominamos esses elementos *agentes de serviços*. Além de divulgar e gerenciar as ofertas de serviço de um ou mais componentes servidores, esses agentes de serviço, tipicamente implemen-

tados como *scripts* Lua, suportam, quando necessário, a configuração de um ambiente de monitoração remoto.

A arquitetura do ambiente proposto é ilustrada na figura 9. Nas próximas seções descrevemos rapidamente as facilidades e mecanismos disponibilizados nesse ambiente. Uma descrição mais completa desses mecanismos é disponível em [Mou00].

4.1 Mecanismo de Monitoração Extensível

O uso de um serviço de *Trading* permite localizar dinamicamente os servidores adequados aos requisitos de uma aplicação com base nos valores das *propriedades não funcionais* que caracterizam o tipo de serviço oferecido. Entretanto, alguns dos requisitos demandados por aplicações distribuídas podem estar relacionados a propriedades que evoluem ao longo do tempo, alterando, dessa forma, as características do serviço prestado por um determinado provedor. Estratégias de adaptação realmente efetivas exigem, portanto, que as aplicações sejam capazes de não apenas selecionar servidores adequados, mas também detetar e reagir a alterações no atendimento a seus requisitos. Para isso, é essencial a provisão de mecanismos que permitam monitorar e, quando possível, controlar, propriedades que evoluem dinamicamente.

Para permitir o desenvolvimento de monitores flexíves, capazes de suportar diferentes características de propriedades e estratégias de monitoração, propomos nesse trabalho um mecanismo de monitoração configurável e extensível, denominado LuaMonitor.

De forma similar a [ZBS97] e [AW99], o mecanismo LuaMonitor define um tipo especial de objeto (um *monitor*), responsável por representar uma determinada propriedade observada, como, por exemplo, o tempo de resposta associado à invocação de uma operação sobre um servidor, ou uma política adotada pelo prestador de um serviço.

O objeto monitor provê uma interface genérica para mecanismos específicos de observação e controle, permitindo a especificação dos requisitos das aplicações através de um conjunto de propriedades condizentes com seu nível de abstração. O mapeamento dessas propriedades para os atributos ou interfaces oferecidos pelos mecanismos utilizados é realizado pelos monitores, que podem, inclusive, agregar na propriedade representada informações obtidas a partir de diferentes mecanismos disponíveis no ambiente.

A funcionalidade básica de um monitor é implementada pelo objeto Lua `BasicMonitor`, que disponibiliza dois métodos (`getValue` e `setValue`), através dos quais o valor corrente da propriedade representada pode ser obtido e alterado.

Em muitos casos uma aplicação pode estar interessada não apenas no valor pontual de uma propriedade, mas em estatísticas ou caracterizações de evolução, como média ou variância. Para atender a essas situações, introduzimos no mecanismo LuaMonitor facilidades para a definição e obtenção de *aspectos* ou *qualificadores* de uma propriedade [FK98, GCS99]. Essas facilidades são oferecidas através da interface `AspectsManager`, definida na figura 10, implementada por todos os objetos monitores. Como os monitores são implementados por objetos Lua, a definição da função que realiza o cálculo de um aspecto (fornecida através do parâmetro `updatef`) pode ser realizada em tempo de execução, permitindo a extensão dinâmica de monitores para atendimento aos requisitos de seus usuários.

É importante observar que a facilidade oferecida pelo *binding* LuaCorba para a implementação de servidores CORBA a partir de objetos Lua permite que objetos monitores sejam acessados e manipulados por usuários remotos, tanto para a obtenção dos valores de uma propriedade e seus aspectos como para a definição de novos aspectos.

```
inteface AspectsManager {
   PropertyValue getAspectValue( in AspectName name );
   AspectList definedAspects();
   void define Aspect( in AspectName name, in string updatef ); };
```

Figura 10: Interface LuaMonitor::AspectsManager

Para facilitar a implementação de estratégias de monitoração orientadas por eventos definimos uma extensão do objeto monitor baseada no *pattern* `Observer` [GHJV95]. Esse padrão define um relacionamento entre um objeto responsável por um determinado estado (no nosso caso, um monitor) e um conjunto de observadores associados a *eventos de interesse* com respeito a alterações nesse estado. Quando um dos eventos é detetado, notificações são enviadas a todos os observadores que registraram interesse nesse evento.

```
interface EventObserver {
   oneway void notifyEvent(in EventID evID); };

interface EventMonitor: BasicMonitor {
  EventObserverID attachEventObserver( in EventObserver obj,
                                       in EventID evID,
                                       in string  notifyf);
  void detachEventObserver( in EventObserverID id ); };
```

Figura 11: Interface LuaMonitor::EventMonitor

Um monitor de eventos implementa a interface `EventMonitor` apresentada na figura 11, acrescentando à funcionalidade oferecida pelo objeto `BasicMonitor` facilidades para o registro de observadores de eventos e para a geração de notificações a esses observadores. Essa geração é suportada por um mecanismo interno de temporização que comanda a atualização do valor da propriedade e aciona, quando necessário, o mecanismo de deteção de eventos. Deve-se notar que a transferência de responsabilidade de deteção de eventos aos objetos monitores permite uma redução no número de interações com esses objetos, o que é particularmente interessante em ambientes que utilizam monitores remotos.

Novamente, o fato de Lua ser uma linguagem interpretada permite a definição de eventos através de `strings` de código Lua, evitando a necessidade de definirmos previamente os tipos de eventos passíveis de observação. A operação que registra um observador de eventos (`attachEventObserver`) recebe, portanto, como parâmetros uma referência para o objeto observador, uma identificação para o evento observado, e uma `string` contendo a definição da função Lua que determina se o evento observado ocorreu.

Os parâmetros passados pelo monitor de eventos a essa função incluem o valor corrente da propriedade monitorada e uma referência para a implementação da interface `AspectManager`, através da qual poderão ser obtidos os valores de aspectos da propriedade observada. A função retornará `true` caso o evento observado tenha sido detetado. Nesse caso, o monitor de eventos enviará uma notificação ao observador correspondente.

Na figura 12, mostramos a definição de um observador da propriedade `CPUstat`, que representa o percentual de tempo do sistema em modo `idle`. Esse observador, implemen-

tado pelo objeto Lua `eventObserver`, é registrado no monitor da propriedade (`mon`), e deverá ser notificado se o valor corrente e a média da propriedade ultrapassarem os limites mínimos definidos.

```
eventObserver = { notifyEvent = function(self,event) ...  end }

mon:attachEventObserver(eventObserver, "CPUstat",
                        [[function(self,currval,aspectmgr)
                             return currval < 90 and
                                aspectmgr:getAspectValue("Mean") < 80
                        end]])
```

Figura 12: Definição de um Observador de Eventos

Em algumas situações, a propriedade representada por um objeto monitor, ou alguns de seus aspectos, poderão corresponder a uma ou mais propriedades dinâmicas associadas a ofertas de serviço divulgadas por um *trader*. Para permitir uma representação única desse tipo de propriedade, o mecanismo LuaMonitor oferece uma função que incorpora, dinamicamente, uma implementação da interface `DynamicPropEval` a qualquer objeto monitor. A implementação dessa interface, definida pelo serviço de *Trading*, permite que, quando necessário, o valor corrente da propriedade representada, e o de seus aspectos, sejam obtidos pelo *trader*.

4.2 Proxies Inteligentes

Os proxies inteligentes de nosso ambiente são implementados como objetos Lua que encapsulam proxies LuaCORBA. Essa implementação confere ao mecanismo uma grande flexibilidade. O uso de proxies LuaCORBA para acesso aos componentes servidores permite a incorporação dinâmica de novos tipos de serviços às aplicações. A implementação de proxies inteligentes através de objetos Lua permite a definição de diferentes comportamentos para esses objetos, conforme os requisitos específicos a serem atendidos (como a substituição de um componente, a escolha do componente conforme a operação requisitada, o uso de métodos alternativos, ou a atuação sobre mecanismos de controle). Além disso, as facilidades oferecidas pela linguagem Lua permitem implementar, de forma trivial, a interceptação das invocações de serviço, e o acionamento do comportamento adaptativo adequado.

Conforme vimos anteriormente, nosso proxy inteligente utiliza o serviço de *Trading* CORBA para localizar componentes servidores que não apenas implementam a interface funcional requerida, mas que satisfazem, também, os requisitos *não funcionais* da aplicação. Utilizando a biblioteca LuaTrading para interagir com o *trader*, o proxy inteligente expressa esses requisitos como restrições sobre o conjunto de propriedades não funcionais associadas ao tipo de serviço representado, e obtém as ofertas de serviço que satisfazem as restrições especificadas.

Para verificar, e garantir, o atendimento aos requisitos da aplicação ao longo do tempo, o proxy inteligente observa e controla as propriedades associadas a esses requisitos. Para isso, utiliza os serviços de monitores, locais ou remotos, que implementam o mecanismo LuaMonitor, descrito na seção anterior. O uso desse mecanismo permite que os proxies inteligentes sejam informados de alterações relevantes nas propriedades observadas, e acionem, no momento adequado, as estratégias de adaptação apropriadas.

Alterações relevantes nas propriedades observadas por um proxy inteligente são definidas como *eventos de interesse* registrados nos monitores correspondentes. A deteção de um desses eventos provocará o envio de uma notificação ao proxy que, para recebê-la, deverá implementar a interface de *callback* `EventObserver`, definida na figura 11. Ao receber uma notificação, o proxy inteligente tanto poderá atuar, imediatamente, sobre mecanismos de controle, como registrar o evento notificado para tratamento posterior. Esse segundo procedimento permite que o proxy inteligente garanta que o acionamento de suas estratégias de adaptação seja realizado em momentos adequados (por exemplo, imediatamente antes e/ou após a invocação de um serviço).

As estratégias de adaptação implementadas por um proxy inteligente poderão também ser acionadas independentemente da notificação de eventos por monitores (através, por exemplo, da obtenção por demanda dos valores das propriedades observadas). Além disso, um único evento pode não determinar completamente o *estado* de qualidade de serviço corrente. Para complementar a identificação desse estado, o tratamento de um evento poderá utilizar valores de outras propriedades, acessando, para isso, os monitores correspondentes.

É importante observar que a implementação de um proxy inteligente é específica não apenas para o tipo de serviço representado como também para os requisitos e estratégias de adaptação adotadas para um determinado tipo de cliente. Contudo, para facilitar a construção de proxies inteligentes, o ambiente proposto oferece funções pré-definidas, que implementam alguns dos mecanismos descritos acima. Uma dessas funções implementa um *observador de eventos* que permite ao proxy inteligente receber notificações enviadas por objetos monitores, registrando os eventos notificados em uma fila de eventos para tratamento posterior. Uma outra função instala em um proxy inteligente um método que aciona as estratégias de adaptação definidas para cada possível evento detetado.

5 Uma Aplicação do Ambiente Proposto

Nessa seção descrevemos um exemplo simples desenvolvido para validar o ambiente proposto. O exemplo trata do compartilhamento de carga entre vários servidores que oferecem a mesma interface funcional (supõe-se que esses servidores não mantêm informações de estado dos clientes). O compartilhamento de carga é responsabilidade dos clientes, que localizam dinamicamente os servidores menos carregados, e dirigem a eles suas requisições.

[BKKD99] descreve um sistema onde os clientes têm o comportamento descrito acima. Esse sistema é baseado no uso do serviço de *Trading* CORBA, e em ofertas de serviço associadas a uma propriedade dinâmica que representa o tempo médio de resposta apresentado pelos servidores. O cliente utiliza esse suporte para determinar qual o servidor que irá utilizar, mas, uma vez feita essa seleção, o sistema não permite que o servidor seja substituído. Dessa forma, se as interações de um conjunto de clientes com os servidores forem longas, o sistema pode se tornar bastante desbalanceado.

Utilizando o ambiente proposto neste trabalho, desenvolvemos um sistema similar ao descrito em [BKKD99], que inclui a substituição dinâmica de servidores.

Para avaliar a carga em cada servidor, utilizamos duas medidas associadas a propriedades dinâmicas: o tempo médio de resposta do servidor (a propriedade `ResponseTime`) e a carga total em um sistema hospedeiro. (a propriedade `LoadAvg`). Desenvolvemos um monitor para cada uma dessas propriedades. O primeiro monitor registra o tempo médio de resposta de um servidor às requisições feitas por seus clientes e executa junto ao servidor. O segundo monitor executa na máquina servidora e computa, a cada minuto, o

número médio de processos na fila de prontos observados nos últimos 1, 5 e 15 minutos. Neste segundo monitor, definimos um *aspecto* que representa, através de um `string`, se foi observado um aumento na carga submetida ao sistema.

Os agentes de serviço de nossa aplicação foram implementados por um *script* Lua executado nos sistemas hospedeiros dos componentes servidores. Esse script é responsável pela criação dos monitores das propriedades `ResponseTime` e `LoadAvg`, pela criação do componente servidor (implementado por um objeto Lua) e pela exportação da oferta do serviço correspondente. Essa oferta de serviço conterá as duas propriedades dinâmicas descritas acima e duas propriedades do tipo `Object Reference`, através das quais são fornecidas aos proxies inteligentes as referências para os dois monitores.

O proxy inteligente utiliza essa infraestrutura para implementar o compartilhamento de carga. Inicialmente, ele seleciona o componente servidor que apresenta, no momento, melhor desempenho e disponibilidade. Essa seleção é realizada através de uma consulta ao *trader* que utiliza como critério de ordenação das ofertas de serviço o tempo médio de resposta dos componentes servidores, e elimina os componentes hospedados em sistemas que apresentam uma tendência de aumento de cargal. Caso nenhuma oferta atenda à restrição imposta, uma segunda consulta é realizada, onde apenas a ordenação das ofertas é especificada.

Através do registro de um *evento* junto ao monitor de tempo de resposta associado ao componente selecionado, o proxy inteligente é informado de uma queda de desempenho relevante nesse componente. Definimos esse evento como um incremento no tempo de resposta médio maior que a carga esperada para dois clientes. O evento notificado será tratado pelo proxy inteligente imediatamente antes da próxima invocação de serviço.

Para adaptar-se à queda de desempenho no serviço representado, o proxy inteligente tenta localizar um componente servidor mais adequado, realizando, para isso, uma nova consulta ao *trader*. Se obtiver uma oferta de serviço associada a um tempo médio de resposta que, somado à carga esperada para mais um cliente, seja pelo menos 20% inferior ao apresentado pelo componente atualmente selecionado, esse componente é substituído.

6 Considerações Finais

Descrevemos nesse trabalho um ambiente com suporte à adaptação dinâmica de aplicações distribuídas à qualidade do serviço oferecido por seus componentes. Estendendo trabalhos realizados anteriormente, exploramos na construção desse ambiente a flexibilidade e simplicidade de programação obtidas com o uso de uma linguagem interpretada na programação baseada em componentes.

Os conceitos e mecanismos empregados em nosso ambiente foram, em grande parte, baseados em um conjunto de trabalhos desenvolvidos nos últimos anos, e, em especial, em algumas propostas de extensões à arquitetura CORBA que suportam a adaptação dinâmica de aplicações distribuídas a variações em requisitos genéricos de qualidade de serviço [ZBS97, BG97, KK98]. Essas propostas, contudo, baseiam-se no uso de linguagens compiladas e em extensões de *stubs* de interfaces funcionais. Dessa forma, tanto a incorporação de novos serviços como de novos requisitos a uma aplicação somente pode ser realizada através da sua recompilação.

Nossa proposta, ao contrário das demais, é baseada no uso de uma linguagem interpretada, e em *bindings* dinâmicos dessa linguagem para sistemas de componentes. [PLS+00] aponta a semelhança de objetivos entre *linguagens de aspectos* [KIL+96] utilizadas para o tratamento de Qualidade de Serviço e linguagens de *scripting*, justificando, entretanto, a

implementação de linguagens específicas pela falta de suporte, em linguagens de *scripting* tradicionais, às caraterísticas adaptativas e de distribuição necessárias ao tratamento de QoS. Essas características, contudo, são adequadamente suportadas pela linguagem Lua e pelo sistema LuaOrb, evitando a necessidade de definição de uma linguagem de aspectos específica para nosso ambiente. Além disso, e principalmente, o dinamismo conferido pelo uso da linguagem Lua e do sistema LuaOrb permite que nosso ambiente não apenas suporte a evolução dinâmica das aplicações, através da incorporação de novos serviços, mas também seja capaz de acompanhar essa evolução através da configuração e extensão de seus mecanismos em tempo de execução.

É importante destacar que é necessária a verificação da aplicabilidade do ambiente proposto em ambientes mais realistas, envolvendo aplicações mais complexas, com diferentes requisitos e estratégias de adaptação. Essa verificação nos permitirá avaliar se os benefícios obtidos pela introdução da capacidade de adaptação dinâmica a aplicações distribuídas compensam a provável perda de desempenho que essas aplicações venham a sofrer.

A implementação de Lua é distribuída em código fonte, e pode ser obtida através de [Luaa]. A implementação do *binding* LuaCORBA, também distribuída em código fonte, pode ser obtida a partir de [Luab]. A implementação dos mecanismos que compõem o ambiente descrito neste trabalho pode ser obtida através do contato com suas autoras.

Referências

[AW99] N. Amano and T. Watanabe. An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++. In *OOPSLA'99 International Workshop on Object Oriented Reflection and Software Engineering*, November 1999.

[BCR00] T. Batista, C. Chavez, and N. Rodriguez. Dynamic Reconfiguration through a Generic Connector. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, Las Vegas, Nevada, USA, June 2000.

[BG97] C. Becker and K. Geihs. MAQS - Management for Adaptive QoS-enabled Services. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, September 1997.

[BKKD99] E. Badidi, R. Keller, P. Kropf, and V. Dongen. The Design of a Trader-based CORBA Load Sharing Service. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Computing Systems (PDCS'99)*, Fort Lauderdale, FL, August 1999.

[CCI99] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects (DOA'99)*, Edinburgh, Scotland, 1999. IEEE Press.

[Cer00] Renato Cerqueira. *Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software*. Tese de Doutorado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil, 2000.

[FIC96] L. Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, December 1996.

[FK98] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems Design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, April 1998.

[GCS99] A. Gomes, S. Colcher, and L. Soares. Um Framework para Provisão de QoS em Ambientes Genéricos de Processamento e Comunicação. In *Anais do XVII Simpósio Brasileiro de Redes de Computadores (SBRC'99)*, Salvador, Bahia, Maio 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Gro98] Object Management Group. *CORBAservices: Common Object Services Specification*. OMG Document formal/98-12-09, December 1998.

[Gro99] Object Management Group. *The Common Object Request Broker: Architecture and Specification v2.3.1*. OMG Document formal/99-10-07, October 1999.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*, chapter 19, pages 827–921. Addison Wesley, 1999.

[ICR98] R. Ierusalimschy, R. Cerqueira, and N. Rodriguez. Using reflexivity to interface with CORBA. In *International Conference on Computer Languages 1998*, pages 39–46, Chicago, IL, 1998. IEEE, IEEE.

[IFC96] R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[KIL+96] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

[KK98] T. Kramp and R. Koster. A service-centred approach to QoS-supporting middleware, September 1998. Work-in-Progress Paper at the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98).

[Luaa] The Programming Language Lua. `http://www.lua.org/`.

[Luab] LuaOrb Online. `http://www.tecgraf.puc-rio.br/luaorb/`.

[Mou00] Ana Lúcia de Moura. Um Ambiente de Suporte à Adaptação Dinâmica de Aplicações Distribuídas. Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil, 2000.

[PLS+00] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, Newport Beach, CA, March 2000.

[RI99] N. Rodriguez and R. Ierusalimschy. Dynamic Reconfiguration of CORBA-Based Applications. In *SOFSEM'99: 26th Conference on Current Trends in Theory and Practice of Informatics*, pages 95–111, Milovy, Czech Republic, 1999. Springer-Verlag. (LNCS 1725).

[ZBS97] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.