Seleção Dinâmica de Objetos Distribuídos no Ambiente LuaSpace

Thais Batista, José Neilton Morais, Milano G. Carvalho, Wander Teixeira
thais@ufrnet.br, neilton@dimap.ufrn.br, milano@lcc.ufrn.br, wander@natalnet.br

Departamento de Informática e Matemática Aplicada - DIMAP
Universidade Federal do Rio Grande do Norte - UFRN
Campus Universitário - Lagoa Nova - 59072-970 - Natal – RN

Resumo. Vários ambientes de programação de aplicações distribuídas têm sido desenvolvidos sobre a infraestrutura oferecida pelas plataformas de distribuição de objetos visando permitir o reuso de componentes. Para incentivar o reuso, é necessário que os ambientes ofereçam facilidades para seleção dinâmica de componentes. Este trabalho apresenta os mecanismos de seleção dinâmica oferecidos por LuaSpace – um ambiente de configuração dinâmica de aplicações baseadas em componentes CORBA. Os mecanismos permitem que o programador expresse facilmente vários critérios para a seleção dinâmica de objetos usando a mesma linguagem utilizada para configurar as aplicações. Além disso, para realizar a seleção dinâmica, os mecanismos exploram o suporte dos serviços de localização de objetos oferecidos pela plataforma CORBA.

Abstract. Many distributed applications development environments have been developed using the infrastructure provided by object distributed platforms in order to allow component reuse. To promote reusability, the environments should offer facilities for the dynamic selection of components. This paper presents the dynamic selection mechanisms provided by LuaSpace - an environment for dynamic configuration of CORBA-based applications. The mechanisms allow the programmer to easily express several selection criteria using the same language used to configure the applications. To realize the dynamic selection, the mechanisms explore the support of CORBA object location services.

1. Introdução

O desenvolvimento de aplicações distribuídas usando a estratégia de programação baseada em componentes [1] tem sido indicada como uma maneira promissora para diminuir o tempo e o custo do desenvolvimento de software. A programação baseada em componentes consiste em compor aplicações através da junção de pedaços de software existentes denominados componentes [1], promovendo o reuso de software [9]. Para dar suporte à programação baseada em componentes, as plataformas de distribuição de objetos, também chamadas de modelos de integração de objetos ou infra-estruturas de componentes de software [13], definem abstrações para comunicação remota e transparente entre componentes heterogêneos e distribuídos. O padrão CORBA [12] destaca-se entre as diversas plataformas de distribuição de objetos por ser uma especificação aberta e independente de fabricante. Vários ambientes para programação de aplicações baseadas em componentes [4,8,14] foram desenvolvidos usando como infraestrutura subjacente alguma plataforma de distribuição de objetos.

Em um ambiente distribuído, para o programador encontrar um componente que ofereça um serviço desejado e/ou que tenha determinadas propriedades é necessário um significante esforço de busca devido à grande quantidade de componentes e serviços disponíveis. Este esforço de realizar uma busca por componentes distribuídos em diversos locais do ambiente pode levar o programador a desistir de reusar componentes. Para solucionar este problema, as plataformas de distribuição de objetos oferecem serviços de localização de objetos por nome (serviço de Nomes) e por propriedades (serviço de *Trading*). Apesar destes serviços oferecerem opções interessantes para se encontrar objetos, a utilização deles envolve o conhecimento de detalhes do conjunto de comandos específicos de cada serviço além da definição dos parâmetros dos comandos. Estes aspectos adicionam um grau significativo de complexidade para o programador expressar os critérios de seleção de objetos. Esta complexidade aumenta bastante quando o programador deseja expressar uma combinação de critérios de seleção. Além disso, a codificação dos critérios usando estes serviços representa um esforço considerável de programação alheio ao domínio da aplicação.

Este trabalho apresenta os mecanismos de seleção dinâmica de um ambiente de configuração e reconfiguração dinâmica de aplicações baseadas em componentes CORBA – *LuaSpace* [4,5]. Um dos propósitos dos mecanismos é possibilitar ao programador expressar facilmente os requisitos funcionais e não funcionais dos componentes que deseja encontrar usando a mesma linguagem que ele utiliza para configurar aplicações. Outro propósito é promover a reconfiguração dinâmica uma vez que a seleção de componentes para compor uma aplicação é realizada em tempo de execução. Este trabalho descreve também como os mecanismos realizam a seleção dinâmica de objetos explorando o suporte dos serviços de Nomes e de *Trading*, bem como de repositórios de meta-informação sobre os objetos.

LuaSpace combina o suporte da plataforma CORBA com a potencialidade de uma linguagem de scripting interpretada e procedural, a linguagem Lua [6], que é utilizada para integrar componentes CORBA em uma aplicação. Em termos de facilidades para localização e seleção de objetos, na versão inicial LuaSpace propõe um mecanismo, denominado *conector genérico* [2], que realiza a seleção dinâmica de componentes usando como critério de busca a assinatura de um método que o componente deve oferecer. O conector genérico torna possível se configurar uma aplicação como um conjunto de serviços sem conhecimento prévio de quais componentes irão executar os serviços. Os componentes serão encontrados em tempo de execução pelo conector genérico que também é responsável por invocar o serviço sobre o componente encontrado. Este mecanismo facilita o reuso de componentes e permite que o programador abstraia-se da tarefa de selecionar componentes para executar um determinado serviço, no entanto o mecanismo é restrito à apenas um único critério de busca (a assinatura de um método) e também não retorna para o programador informações sobre o componente que está executando o serviço. Desta forma, o programador não tem como saber informações adicionais sobre o componente selecionado, tais como a interface funcional do componente e/ou as suas propriedades não funcionais.

A integração ao ambiente dos mecanismos apresentados neste trabalho significa oferecer ao programador flexibilidade em termos de critérios de busca, que pode incluir as características funcionais e/ou não funcionais dos objetos a serem procurados, e também o retorno da identificação do objeto que foi selecionado. Os critérios de busca podem ser descritos como uma expressão envolvendo operadores, de forma a compor requisitos muito abrangentes que os objetos procurados deverão satisfazer. As estratégias propostas aliadas ao conector genérico formam um importante suporte para seleção dinâmica de componentes usando diferentes critérios de busca. De forma a assegurar a interoperabilidade, LuaSpace evitou recorrer a extensões proprietárias de mecanismos de busca e optou por aproveitar o suporte dos repositórios, dos serviços e dos mecanismos da plataforma CORBA.

Este trabalho está estruturado da seguinte forma. A seção 2 discute os aspectos da plataforma CORBA e do ambiente LuaSpace relacionados com seleção dinâmica de objetos e também a biblioteca LuaTrading [11] que facilita o acesso ao serviço de *Trading* usando a linguagem Lua. A seção 3 descreve as estratégias para seleção dinâmica de objetos oferecidas por LuaSpace e comenta a sua implementação. A seção 4 apresenta os trabalhos correlatos. A seção 5 contém as conclusões.

2. Conceitos Básicos

2.1 CORBA

CORBA (Common Object Request Broker Architecture) [12] é um padrão proposto pela Object Management Group (OMG) cujo propósito é permitir interoperabilidade entre aplicações em ambientes distribuídos e heterogêneos. Este padrão estabelece a separação entre a interface de um objeto e sua implementação. Para descrição da interface do objeto, CORBA oferece a *linguagem para definição de interfaces (IDL).* Para implementação do objeto CORBA, pode ser utilizada qualquer linguagem de programação que tenha o *binding* para CORBA.

A arquitetura CORBA é composta por um conjunto de blocos funcionais que usam o suporte de comunicação do *ORB (Object Request Broker)* - o elemento responsável por coordenar as interações entre os objetos interceptando as chamadas dos clientes e direcionando-as para o servidor apropriado.

Todo objeto CORBA possui uma identificação, chamada *referência do objeto*, que é atribuída pelo ORB na criação do objeto. Para usar um objeto, o cliente deve obter a sua referência pois em uma invocação de um método sobre o objeto, o ORB o identifica através de sua referência.

O *Repositório de Interfaces* definido no padrão CORBA disponibiliza informações necessárias para a construção de chamadas dinâmicas. Este repositório armazena todas as definições IDL dos objetos CORBA disponíveis para uso.

CORBA oferece um conjunto de serviços para dar suporte ao desenvolvimento de aplicações distribuídas. Cada serviço é descrito por interfaces IDL e composto por objetos CORBA. Os serviços CORBA que dão suporte à localização de objetos são o serviço de Nomes e o serviço de *Trading*.

O *Serviço de Nomes* realiza um mapeamento entre um nome simbólico e uma referência de objeto, e define funções para inserção e remoção de associações de nomes, além de funções para consultas ao repositório de nomes que retornam a referência do objeto. Apesar de fornecer suporte para a localização de objetos, este serviço impõe que o cliente conheça o nome simbólico do objeto desejado.

Enquanto o Serviço de Nomes localiza objetos por *nomes*, o *Serviço de Trading* procura objetos por *tipos* e *propriedades*. Propriedades são pares "nome-valor". Um objeto *trader* oferece um *repositório de ofertas de serviço*. Uma *oferta de serviço* é uma instância de um *tipo de serviço*. Um tipo de serviço é descrito por um nome, uma interface IDL associada e uma lista de propriedades que definem os aspectos não funcionais associados ao tipo de serviço. Uma oferta de serviço consiste de um nome do tipo de serviço que associa a oferta a um determinado tipo, a referência do objeto provedor da oferta e de uma descrição das características não funcionais (propriedades) da oferta que deve ser compatível com os tipos de propriedades definidas pelo tipo de serviço.

Por exemplo, a Figura 1 ilustra um exemplo simples de criação do tipo de serviço Printer em C++ que é associado à interface funcional identificada por IDL:Printer:1.0. As propriedades especificadas para o tipo são: o nome da impressora (name), o tamanho do papel (paper), e a velocidade de impressão (speed).

```
using namespace CosTradingRepos;

CORBA::Object *obj = lookup->type_repos();
ServiceTypeRepository *repos = ServiceTypeRepository::_narrow(obj);

ServiceTypeRepository::PropStructSeq props;
ServiceTypeRepository::ServiceTypeNamesSeq super_types;

props.length(3);

props[0].name = CORBA::string_dup("name");
props[0].value_type = CORBA::TypeCode::_duplicate(CORBA::_tc_string);
props[0].mode = ServiceTypeRepository::PROP_NORMAL;
props[1].name = CORBA::string_dup("paper");
props[1].value_type = CORBA::TypeCode::_duplicate(CORBA::_tc_string);
props[1].mode = ServiceTypeRepository::PROP_MANDATORY;
props[2].name = CORBA::string_dup("speed");
props[2].value_type = CORBA::TypeCode::_duplicate(CORBA::_tc_string);
props[2].mode = ServiceTypeRepository::PROP_NORMAL;

repos->add_type("Printer", "IDL:Printer:1.0", props, super_types);
```

Figura 1 - Criação de um Tipo de Serviço

Para divulgar suas ofertas os objetos usam operações de *exportação*. Clientes interessados em procurar serviços invocam as operações de *importação*, buscando referências de objetos cujas propriedades satisfazem a um conjunto de requisitos aplicados sobre os valores dessas propriedades. Os critérios de busca para a operação de importação são estabelecidos na linguagem específica do serviço de *Trading*, denominada *Trader Constraint Language*.

```
using namespace CosTrading;
Register *regis = lookup->register_if();

PropertySeq props;
props.length(3);

props[0].name = CORBA::string_dup("name");
props[0].value <<= "InkJet";
props[1].name = CORBA::string_dup("paper");
props[1].value <<= "A4";
props[2].name = CORBA::string_dup("speed");
props[2].value <<= "4ppm";

Printer *print_serv = . . .;

OfferID_var offer_id = regis->export(print_serv, "Printer", props);
```

Figura 2 - Exportação de uma Oferta

A Figura 2 ilustra o código C++ da exportação de uma oferta de serviço do tipo `Printer`. Neste caso, o servidor responsável pela oferta de serviço está publicando a disponibilidade de uma impressora a jato de tinta (`"InkJet"`), configurada para impressão em papel A4 (`"A4"`), e cuja velocidade de impressão é de quatro páginas por minuto (`"4ppm"`).

A Figura 3 contém um exemplo de código C++ que implementa uma consulta simples ao Serviço de *Trading* para a localização de ofertas de serviço do tipo Printer. Neste caso, procura-se por impressoras cujo papel esteja configurado para A4 e cuja velocidade de impressão seja de pelo menos 4ppm.

```
PolicySeq policies;          //empty
Lookup:SpecifiedProps desired_props; //empty

desired_props._default();
desired_props._d(Lookup:none);

PolicyNameSeq_var policies_applied;
OfferSeq_var offers;
OfferIterator_var iterator;

lookup->query("Printer", "paper == 'A4' and speed >= "4ppm"), "",
      policies, desired_props, 0, offers, iterator, policies_applied);

if (!CORBA::is_nil (iterator)) {
   CORBA::Boolean more;
   do {
      more = iter ->next_n(5, offers);
      for (CORBA::Ulong i = 0; i < offers->length(); i++)
         /* … */
   } while more;
   iter->destroy();
}
```

Figura 3 - Importação de Ofertas

Uma restrição observada com relação à utilização do Serviço de *Trading* refere-se ao fato de que todo tipo de serviço criado deve estar associado a uma interface IDL e o *Trading* não fornece facilidades de localização de objetos que implementem tipos de serviços quaisquer.

O serviço de *Trading* oferece aos programadores um controle bastante detalhado sobre a forma de como as ofertas de serviço devem ser pesquisadas e selecionadas. Como ilustrado nos exemplos, este aspecto adiciona um grau significativo de complexidade para a realização de consultas. Além disso, para usar as operações fornecidas pelo serviço de *Trading* é necessário incluir, no código do programa, uma série de comandos CORBA para habilitar o uso do serviço e o acesso ao repositório.

2.2 LuaTrading

A biblioteca LuaTrading [11] foi implementada visando simplificar a interação do programador Lua com o serviço de *Trading*. A biblioteca explora o suporte a dinamismo oferecido pela linguagem Lua e a facilidade de acesso a objetos CORBA via Lua conferido por LuaOrb [7] – um *binding* entre CORBA e a linguagem Lua.

Lua é uma linguagem interpretada e dinamicamente tipada com sintaxe simples. Em Lua não é necessário declarar tipos de variáveis e funções. Lua inclui aspectos convencionais como estruturas de controle similares as de Pascal e também características não convencionais: funções são valores de primeira classe; *arrays* associativos (chamados de *tabelas* em Lua) consistem na única facilidade para estruturação de dados; *tag methods* é um mecanismo reflexivo de Lua. *Tag methods* podem ser especificados para serem chamados em situações que o interpretador Lua não sabe como proceder. Este mecanismo é a base para implementação de várias ferramentas baseadas em Lua pois quando o interpretador Lua não sabe executar um comando, ele invoca o *tag method* apropriado para tratar o comando.

LuaOrb é um *binding* entre Lua e CORBA baseado na interface de invocação dinâmica de CORBA (DII) que possibilita o acesso dinâmico a componentes CORBA via Lua da mesma forma que se utiliza qualquer objeto Lua. Além disso, LuaOrb usa a interface de esqueleto dinâmico de CORBA (DSI) para permitir a instalação dinâmica de objetos Lua como servidores CORBA. LuaOrb usa *tag methods* para estabelecer que o interpretador Lua deve direcionar para LuaOrb as chamadas sobre objetos CORBA.

Para usar o serviço de *Trading* via LuaTrading é necessário inicialmente criar, através da função serviceTypesRep(), um objeto Lua representante local do repositório de tipos de serviços. LuaTrading fornece operações para a criação de novos tipos de serviço, eliminação e consulta aos tipos de serviços além de divulgação, modificação e remoção de ofertas de serviços. As operações aplicadas sobre o objeto Lua são mapeadas automaticamente em operações correspondentes do serviço de *Trading*.

A Figura 4 ilustra a definição do tipo de serviço Printer usando LuaTrading. Na criação deste novo tipo de serviço junto ao Repositório de Tipos de Serviço (typesRep), são definidas algumas propriedades que caracterizam o serviço implementado pelo servidor. No caso do Servidor de Impressão, as principais propriedades especificadas foram: o nome da impressora (name), o tamanho do papel (paper), a velocidade de impressão (speed), a resolução especificada (resolution), a utilização de cores (color), e o local da impressora (hostname). Para simplificar esta especificação, todas estas propriedades são definidas como sendo do tipo string, porém o tipo de uma propriedade pode ser definido por qualquer tipo CORBA.

```
-- criação do objeto Lua para representar o repositório de tipos de serviço
typesRep = serviceTypesRep()

if not typesRep.servicetypes.Printer then
   props = { {name="name",type="string",mode="PROP_NORMAL"},
             {name="ppm",type="long",mode="PROP_NORMAL"},
             {name="color",type="boolean",mode="PROP_NORMAL"},
             {name="hostname",type="string",mode="PROP_NORMAL"}
           }
   typesRep.servicetypes.Printer = {interface="Printer",props=props}
   print("New Service Type: 'Printer'.")
else
   print("Service Type: 'Printer' already defined.")
end
```

Figura 4 - Criação de um Tipo de Serviço usando LuaTrading

O script Lua, associado a biblioteca LuaTrading, que codifica a exportação de uma oferta associada ao tipo de serviço Printer está ilustrado na Figura 5. Para isto, LuaTrading oferece a função export.

```
-- InkJet_Printer.lua
-- Criação do objeto representante do repositorio de tipos de servico
typesRep = serviceTypesRep()

-- Criação do objeto representante do repositorio de ofertas de serviço
offersRep = serviceOffersRep(typesRep)

-- Definicao de Propriedades
prop = { name="HP InkJet 640c",
         paper="Letter",
         speed="4ppm",
         resolution="300dpi",
         color="TRUE",
         hostname="pipa.dimap.ufrn.br"
       }

-- Exportacao de Oferta de Servico
offersRep:export{server=inkjet_servant,type="Printer",properties=prop}
```

Figura 5 - Exportação de uma oferta usando LuaTrading

Para importação de ofertas LuaTrading oferece a função importServiceOffers que utiliza valores *default* para grande parte dos parâmetros definidos pela operação de importação oferecida pelo serviço de *Trading* do CORBA (query). Esta estratégia facilita a expressão de consultas simples. A função importServiceOffers encapsula as interações com outras interfaces do serviço de *Trading* que dão suporte à importação de ofertas.

A função importServiceOffers recebe como parâmetro uma tabela Lua que especifica as informações para importação de ofertas. Esta tabela tem quatro campos: *type* (o tipo do serviço procurado), *constraint* (a expressão que determina os critérios para seleção de ofertas de serviço), *pref* (campo opcional para determinar a ordenação das ofertas retornadas) e *maxret* (o número máximo de ofertas de serviço desejado pelo importador).

A Figura 6 apresenta um exemplo de importação para obter servidores que oferecem o tipo de serviço Printer que imprima em papel A4 com velocidade igual ou superior a 4ppm. A consulta deve retornar no máximo 4 ofertas.

```
-- Importação de Ofertas

ofertas = importServiceOffers{type = "Printer",
                              constraint = "paper == 'A4' and speed >= '4ppm'",
                              maxret = 4}
```

Figura 6 - Importação de Ofertas usando LuaTrading

2.3 LuaSpace

O ambiente LuaSpace é composto pela linguagem Lua e por um conjunto de ferramentas baseadas nesta linguagem que juntos proporcionam um ambiente poderoso e flexível para fins de reconfiguração dinâmica [3]. Uma das ferramentas que compõem o ambiente é LuaOrb [7] que implementa o acesso dinâmico a objetos CORBA via Lua.

Usando LuaSpace é possível compor aplicações unindo componentes CORBA já existentes assim como é também possível desenvolver novos componentes CORBA. As aplicações são escritas em Lua e podem usar componentes desenvolvidos em qualquer linguagem que tenha o *binding* para CORBA. O acesso aos componentes CORBA é feito por LuaOrb de forma transparente para o programador e para o usuário da aplicação.

Em LuaSpace um programa de configuração é um script Lua que pode conter comandos Lua e comandos para invocação de componentes CORBA.

Em termos de suporte para seleção dinâmica de objetos, a versão inicial de LuaSpace oferece um mecanismo – o Conector Genérico [2] que permite que se escreva uma aplicação estabelecendo apenas os serviços que se deseja usar, sem determinar o(s) componente(s) que oferece(m) tais serviços. O conector genérico seleciona dinamicamente os componentes que oferecem os serviços estabelecidos e que irão compor a aplicação. No processo de seleção destes componentes, o conector genérico usa como critério de busca a assinatura dos métodos correspondentes aos serviços a serem localizados. O conector genérico é um objeto Lua criado através da função generic_createproxy() que retorna um *proxy* – o representante do conector genérico. No programa de configuração um serviço "órfão" é invocado da mesma forma que são invocados serviços de componentes conhecidos, mas com a variável que representa o conector genérico (o *proxy*) no lugar do nome do componente, dando a idéia de que o serviço é implementado pelo conector genérico.

Quando um método é invocado sobre o *proxy* do conector genérico o interpretador Lua intercepta a invocação e implicitamente invoca a implementação do conector genérico. O conector genérico invoca uma função de busca para procurar componentes em um repositório padrão ou sobre a *tabela de configuração* – uma tabela gerenciada pelo conector genérico que mantém a identificação dos componentes selecionados para executar serviços invocados pela aplicação via o conector genérico. O conector genérico pode ser instruído pelo programador da aplicação para realizar a busca no repositório ou para consultar primeiro a tabela de configuração. O comportamento padrão é primeiro consultar a tabela de configuração e o processo de busca deve prosseguir sobre o repositório apenas se nenhum componente é encontrado na tabela de configuração.

Considerando que um serviço de impressão é oferecido por servidores que implementam a interface Printer ilustrada na Figura 7. A Figura 8 exibe um programa de configuração que usa o serviço print via o conector genérico, isto é, sem especificar o componente que oferece o serviço. Na chamada c:print("arquivo.ps") o conector genérico intercepta a chamada e implicitamente invoca uma função interna que busca em um repositório padrão algum componente que ofereça o serviço print. Ao encontrar um componente que satisfaça tal critério de busca, o conector monta a solicitação CORBA e invoca o serviço sobre o componente selecionado. Por fim, após a execução do serviço o conector retorna os resultados para o programa de configuração.

```
interface Printer {
        short print(in file);
}
```

Figura 7 - Interface IDL Printer

```
c = generic_createproxy()
status = c:print("arquivo.ps")
```

Figura 8 - Programa de configuração com o conector genérico

Apesar do conector genérico oferecer uma facilidade importante para seleção dinâmica de objetos, a sua versão original possui a restrição de considerar como critério de busca apenas a assinatura de um método. Em algumas situações o programador pode desejar procurar um componente que satisfaça alguns critérios não funcionais ou que ofereça um determinado conjunto de métodos.

Um outro aspecto que em determinadas situações pode ser considerado uma restrição do conector genérico é o fato de não retornar para o programador a identificação do componente que irá executar o serviço. Este comportamento é útil nas situações em que o programador deseja que a operação seja realizada mas não se interessa em saber qual o componente que executa a operação nem informações adicionais sobre o mesmo. Em outras situações o programador pode precisar de tais informações, portanto o ambiente deve prover também um mecanismo adicional que ofereça outras facilidades para seleção dinâmica de objetos.

3. Mecanismos de Seleção Dinâmica em LuaSpace

O suporte para seleção dinâmica de objetos envolve a provisão de operações para a expressão dos critérios de seleção usando uma determinada linguagem e a implementação dos mecanismos subjacentes que irão efetivar a busca e retornar os resultados. Esta seção discute como estes aspectos são endereçados por LuaSpace. A subseção 3.1 descreve as opções que LuaSpace oferece para se estabelecer os critérios de seleção. A subseção 3.2 comenta sobre a implementação dos mecanismos de seleção.

3.1 Operações para Seleção Dinâmica

Para viabilizar o reuso de objetos LuaSpace fornece mecanismos de seleção dinâmica que disponibilizam operações para o programador expressar os critérios de seleção usando a mesma linguagem usada na configuração da aplicação – a linguagem Lua. Desta forma evita-se que o programador tenha que aprender linguagens específicas dos serviços de busca. LuaSpace disponibiliza objetos e funções Lua para seleção dinâmica e explora o suporte de LuaTrading e dos serviços de *Trading* e de Nomes de CORBA para efetivar a seleção dinâmica. As consultas escritas em Lua são mapeadas em operações correspondentes na linguagem disponibilizada pelo serviço usado por LuaSpace para realizar a seleção.

O ambiente também possibilita que o programador use diretamente as operações oferecidas pelo LuaTrading e pelos serviços CORBA. Para facilitar o uso destas opções, LuaSpace habilita automaticamente estes serviços e seus repositórios de forma que o programador pode invocar as operações destes serviços sem antes ter de emitir os comandos para torná-los operacionais. Com esta facilidade, além de dispor de um mecanismo flexível e simples de usar, que permite a expressão dos critérios de seleção usando Lua, o programador dispõe também dos serviços CORBA e da biblioteca LuaTrading e portanto pode escolher em que nível de abstração deseja expressar os critérios de seleção.

Os seguintes critérios de seleção e suas combinações abrangem diversas possibilidades de localização de componentes:

- Nome do Objeto
- Assinatura de um Método
- Propriedades (características não funcionais)
- Assinatura de um conjunto de Métodos

Para possibilitar a expressão de tais critérios de seleção, LuaSpace adotou duas estratégias: estender o conector genérico permitindo que o programador expresse também propriedades como critério de busca e oferecer uma função que recebe como parâmetro uma lista de critérios de seleção e retorna as referências dos objetos que satisfazem os critérios.

3.1.1 Conector Genérico

Para especificação de propriedades a sintaxe do conector genérico foi estendida para incluir, além da assinatura do método, a descrição opcional de propriedades que serão consideradas para a seleção de componentes.

```
c = generic_createproxy()
c:print("arquivo.ps") ("speed > '4ppm' ")
```

Figura 9 - Especificação de propriedades com o conector genérico

A sintaxe para expressar os dois critérios de busca é `c:método(parâmetros) (propriedades)`, onde `c` é o *proxy* do conector genérico. A Figura 9 ilustra um trecho de programa de configuração que utiliza o conector genérico para selecionar um componente que tenha o método `print` e a propriedade `speed > '4ppm'`.

A especificação de propriedades é opcional e quando não é usada, o conector genérico procede a busca considerando como critério de seleção apenas a assinatura do método. A chamada `c:print("arquivo.ps")` corresponde ao exemplo da invocação ilustrada na Figura 9 omitindo-se a descrição das propriedades.

Considerando também como critério de busca as propriedades que um objeto deve satisfazer, o conector genérico pode encontrar um resultado semanticamente mais correto que o encontrado quando o mecanismo realiza apenas um *matching* sintático da assinatura do serviço solicitado com os serviços cadastrados em um repositório de objetos. Como o processo sintático não considera qualquer característica semântica do serviço, o resultado pode não corresponder ao que se espera. Por exemplo, o programador pode invocar um método print sobre o conector genérico, passando como parâmetro um arquivo do tipo *postscript* e omitindo as propriedades. Sem especificar as propriedades, o conector genérico pode selecionar um objeto com um método print que não imprima arquivos do tipo *postscript*.

Com esta nova funcionalidade, o conector genérico permite que o programador realize a localização de objetos com base na assinatura de um método e também com base na combinação de propriedades com a assinatura de um método. Ao selecionar um objeto, o conector invoca o método e após a sua execução retorna os resultados.

O conector genérico inclui um mecanismo de tolerância a falhas cujo objetivo é garantir que o mecanismo tenta satisfazer todas as invocações sob sua responsabilidade resolvendo, sempre que possível, as falhas decorrentes da não disponibilidade de um componente selecionado. Neste caso, ao invés de reportar o problema para o usuário, o conector seleciona um outro componente que ofereça o serviço solicitado. A falha de uma invocação só é propagada para o usuário caso o conector genérico tenha esgotado as possibilidades de encontrar algum componente para executar o serviço. O mecanismo de tolerância a falhas é discutido detalhadamente em [15].

3.1.2 Função *search*

Para dar suporte à especificação de diversos critérios de seleção e inclusive a combinação de vários critérios, LuaSpace oferece a função search(lista_critérios) que recebe como parâmetro uma lista de critérios de busca e fornece como resultado uma lista contendo as referências de objetos que satisfazem os critérios. De posse desta lista, o programador pode escolher qual(is) objeto(s) utilizar. Esta função realiza os seguintes tipos de seleção: baseado em nomes do objeto, baseado na assinatura de um método, baseado nas propriedades não funcionais do objeto e baseado assinaturas de diversos métodos.

O parâmetro lista_critérios é uma tabela Lua que contém os tipos de critérios de seleção e suas restrições e, em caso de vários tipos de critérios de seleção, contém a identificação da operação lógica que indica o relacionamento entre os tipos de critérios. Um campo desta tabela contém o identificador do tipo de critério de seleção (NAME, OPERATION, ou PROPERTIES) e a expressão associada com o critério de busca que indica as restrições a serem consideradas na seleção. Diferentes tipos de critérios de busca podem ser estabelecidos na lista de critérios usando-se operadores lógicos (and, or, not) para expressar o relacionamento dos tipos de critérios. O campo do operador lógico, se presente, indica a operação lógica que deve ser aplicada sobre os resultados da seleção em dois tipos de critérios. Operadores lógicos também podem ser usados para relacionar diferentes opções em um mesmo critério de seleção.

---

a. Seleção de objeto por nome: buscar um objeto cujo nome seja “print”:
   search{ {NAME = “print”} }

b. Seleção de objeto por nome especificando uma combinação de alternativas:
   search{{NAME = “print or printer or LaserPrinter and not DeskJetPrinter”}}

c. Seleção de objeto usando uma combinação de critérios (nome do objeto e propriedades):
   search{{NAME = “print”}, ”and”,
   {PROPERTIES = {constraint = “speed >= ‘4ppm’ and resolution = ‘600dpi’ ”}}}

Figura 10 - Exemplo de especificações de critérios de seleção usando a função *search*

A Figura 10 ilustra o uso da função *search* com diferentes tipos de critérios.

O objetivo desta função é oferecer uma maneira uniforme de se expressar diversos critérios de busca evitando que o programador use diferentes serviços para cada tipo de critério. Por exemplo, usando-se os serviços de seleção oferecidos por CORBA, para especificar uma seleção por nome, o programador tem de usar o serviço de Nomes, especificamente a operação resolve. Para realizar seleção considerando propriedades, o programador tem de usar o serviço de *Trading*, especificamente a função de importação. Portanto, é necessário o conhecimento de cada um dos serviços e de suas operações. Em contraste, usando-se LuaSpace pode-se usar uma única função para se expressar os critérios de seleção sendo possível também determinar a combinação de diferentes tipos de critérios.

3.2 Implementação dos Mecanismos

Para que os mecanismos de seleção dinâmica de LuaSpace aproveitem o suporte dos serviços existentes - o serviços de Nomes, de *Trading* e da biblioteca LuaTrading – o passo inicial da implementação consistiu em integrar tais serviços ao ambiente.

Para viabilizar a seleção dinâmica em um alto nível de abstração, evitando que o programador tenha de conhecer os mecanismos subjacentes, o ambiente LuaSpace implementa uma função de habilitação automática dos serviços, que invoca as funções para colocar em operação os serviços e os seus respectivos repositórios.

As invocações aos componentes e serviços CORBA realizadas via LuaSpace são mapeadas dinamicamente para a plataforma CORBA. O Repositório de Interfaces (RI) definido no padrão CORBA armazena as descrições IDL dos objetos CORBA e disponibiliza informações necessárias para a construção das chamadas dinâmicas. No entanto, a utilização do RI no suporte à seleção dinâmica de objetos apresenta limitações uma vez que o RI não oferece suporte para se consultar se existe alguma IDL que apresenta determinado método. Os serviços de Nomes e de *Trading* também não dispõem de operações para consulta por métodos.

Para permitir seleção dinâmica por método ou conjunto de métodos, o ambiente LuaSpace implementa um procedimento de busca no RI, chamado search_RI, para descobrir todas as interfaces IDL que apresentam o método com a assinatura especificada. Este processo de busca verifica, para cada interface, se esta define um método com o nome especificado no critério de seleção. Em caso afirmativo, são comparados os argumentos do método presente no RI com o especificado na seleção. Se os argumentos são iguais, esta interface é incluída na tabela a ser retornada pela função search_RI. Para cada interface retornada pela função search_RI, é invocada uma consulta ao serviço de *Trading*, usando-se a função search_trading, para a localização das ofertas de serviços associadas a cada interface IDL selecionada. Esta função retorna o conjunto de ofertas de serviço que implementam o método solicitado.

Se a seleção por método for especificada usando o conector genérico, a implementação deste mecanismo após receber a lista retornada pela função search_trading, armazena esta lista em uma *tabela de grupos de componentes* que dar suporte ao mecanismo de tolerância à falhas. Em seguida, seleciona um dos componentes retornados (por *default*, o primeiro da lista). O próximo passo é a montagem da solicitação de execução do serviço. Esta solicitação consiste em uma série de comandos Lua incluindo comandos disponibilizados pela interface LuaTrading que irão criar automaticamente um *proxy* Lua para a ativação do componente que implementa o método especificado. A próxima ação consiste em registrar na tabela de configuração o método e o componente selecionado. Finalmente, o conector genérico invoca o método e depois da execução, retorna o resultado emitido pela execução do método.

Se a seleção por métodos ou conjunto de métodos for especificada usando a função *search*, a lista de componentes emitida pela função search_trading é retornada para o programador que pode escolher qualquer um dos componentes da lista e invocar a execução do método sobre o componente selecionado.

LuaSpace implementa uma função para registro automático de componentes no serviço de *Trading*. Quando o programador solicita, via LuaSpace, o registro de uma nova interface IDL no Repositório de Interfaces, o ambiente realiza também a definição automática de um novo tipo de serviço no *Trading* e pergunta ao programador se ele deseja especificar as propriedades do tipo de serviço. Para realizar a definição automática, a função register_interface de LuaOrb foi estendida para fazer chamada à função de LuaTrading responsável pela criação de um novo tipo de serviço. Além disso, quando um objeto Lua é definido como um objeto CORBA, isto representa uma nova oferta de serviço que deve também ser criada automaticamente. A função lo_createservant de LuaOrb foi também estendida para fazer a chamada automática da função de exportação de ofertas disponibilizada por LuaTrading. Se o programador optar inicialmente por não registrar as propriedades do componente, ele poderá fazer isto posteriormente chamando a função set_properties oferecida por LuaSpace. Mesmo que o programador não estabeleça as propriedades do componente, o registro no serviço de *Trading* é necessário pois a seleção baseada na assinatura de um método ou de um conjunto de métodos é realizada sobre o repositório mantido pelo *Trading*. O registro automático de tipos de serviços e de ofertas de serviços poupa o programador de fazer esta tarefa e garante que todos os componentes CORBA instalados via LuaSpace irão estar registrados no repositório do serviço de *Trading*, sendo possível localizá-los para fins de reuso.

Para realizar a seleção dinâmica por propriedades, LuaSpace explora o suporte de LuaTrading. Quando o conector genérico é usado para realizar a seleção por método e por propriedades, a implementação deste mecanismo além de realizar a consulta descrita anteriormente para seleção por método, também invoca a operação de importação de ofertas de LuaTrading passando as propriedades como argumento. Como resultado, o conector genérico obtém componentes que apresentam o método e que satisfazem as propriedades especificadas pelo programador. Seguindo o seu comportamento padrão, o conector genérico realiza os demais passos para invocar o método sobre um dos componentes selecionados e, após a execução, retorna o resultado.

Quando a função *search* é usada para expressar um conjunto de tipos de critérios, a implementação desta função realiza a seleção de cada critério em separado e em seguida, aplica a operação lógica sobre os resultados obtidos. Por exemplo, na função search{{NAME = "print"}, "and", {PROPERTIES = {constraint = "speed >= '4ppm' and resolution = '600dpi' "}}} são especificados dois critérios de seleção (por nome e por propriedades) e o operador and que determina a intersecção entre os dois critérios. A implementação da função *search* realiza a seleção por nome invocando a operação resolve do serviço de Nomes, em seguida executa a seleção por propriedades usando a função de importação de ofertas de LuaTrading e por fim realiza a intersecção dos resultados obtidos nas duas seleções e obtém a lista com o resultado final. Esta lista é retornada para o programador.

4. Trabalhos Correlatos

O *Visibroker Location Service* [10] é uma extensão da especificação CORBA que oferece facilidades de propósito gerais para localização de objetos. Este serviço mantém um catálogo que contém uma lista de instâncias com suas respectivas descrições IDL. O objetivo do serviço é dar suporte ao balanceamento de carga e monitoração. Neste serviço as solicitações são baseadas no identificador do RI ou na combinação do identificador do RI e o nome da instância. Os resultados são referências de objetos ou descrições mais completas sobre a instância (referência, nome da interface da instância, nome da instância, nome da máquina e da porta onde está a instância). Apesar deste serviço oferecer uma maneira de se selecionar objetos distribuídos, seu propósito difere do mecanismo de seleção dinâmica de LuaSpace. LuaSpace não mantém repositórios de objetos pois por questões de interoperabilidade optou por aproveitar o suporte dos serviços de seleção de CORBA (Nomes e *Trading*) bem como de seus respectivos repositórios. As tabelas mantidas pelo conector genérico armazenam apenas a identificação de um subconjunto de componentes que estão nos repositórios CORBA e são utilizadas pelo mecanismo de tolerância à falhas. Além disso, LuaSpace visa oferecer um nível superior de abstração em relação aos serviços CORBA provendo facilidades para que no programa de configuração o programador estabeleça seus critérios de seleção usando a mesma linguagem usada para configuração da aplicação.

O FTDA (*Fault Tolerant Dynamic Access*) [16] é um mecanismo cujo propósito é prover um serviço que englobe facilidades para localização dinâmica de objetos e tolerância a falhas para o ambiente Aster [19]. Aster utiliza o suporte da plataforma CORBA mas o FTDA optou por não usar os serviços CORBA para fins de seleção dinâmica e desenvolveu sua solução proprietária. Para permitir que aplicações acessem serviços oferecidos por objetos desconhecidos a priori, o FTDA implementa um serviço de seleção dinâmica semelhante ao serviço de *Trading* de CORBA possibilitando consulta por propriedades. O diferencial do FTDA em relação ao serviço de *Trading* de CORBA é unir um serviço de localização com um mecanismo de tolerância a falhas fazendo com que estes serviços cooperem entre si. Em LuaSpace o conector genérico também oferece um mecanismo de tolerância a falhas [15]. Diferentemente do Aster, LuaSpace não reimplementa facilidades oferecidas por serviços CORBA mas aproveita o suporte dos serviços existentes e cria um nível de abstração sobre estes serviços, com intuito de oferecer uma maneira uniforme para o programador estabelecer seus critérios de busca. A seleção de objetos no FTDA apresenta a restrição de considerar apenas um único critério, as propriedades do componente. LuaSpace permite que o programador estabeleça como critério de seleção expressões complexas que correspondem à combinação de vários critérios de seleção. Esta facilidade evita que o programador tenha de interagir com diferentes serviços de localização para realizar uma seleção considerando diferentes tipos de critérios.

MAGNET [17] é uma arquitetura que permite que aplicações solicitem serviços usando como critério de seleção o tipo de serviço ao invés de usar o nome do objeto. O objetivo do MAGNET é permitir que usuários conheçam os recursos do sistema através da seleção por tipo de serviço. Por exemplo, para conhecer qual a impressora que o sistema oferece, o usuário pode fazer uma consulta ao MAGNET especificando o tipo de serviço Printer. MAGNET implementa um objeto *Trader* que armazena informações sobre serviços e recebe solicitações de seleção de serviços. Este objeto mantém um repositório que armazena informações sobre objetos em um formato similar ao usado no espaço de tuplas da linguagem Linda [18]. As consultas ao objeto *Trader* são definidas em termos de operações sobre tuplas. Este mecanismo limita-se apenas à seleção por tipo de serviço e implementa uma solução proprietária para a seleção de componentes.

O enfoque do mecanismo de seleção dinâmica dos três trabalhos (Visibroker Location Service, FTDA e MAGNET) é dar suporte a outras funcionalidades: no Visibroker o objetivo é apoiar uma estratégia de balanceamento de carga, no FTDA a finalidade é permitir tolerância a falhas e no MAGNET o propósito é oferecer suporte a gerência de recursos do sistema permitindo que os usuários consultem quais os recursos disponíveis. Em contraste, os mecanismos de seleção dinâmica de LuaSpace têm como objetivo incentivar o programador a reusar componentes e para isto propõe formas simples do programador expressar os critérios de busca. Desta forma, o programa de configuração da aplicação é simples e compreensível pois não inclui uma série de comandos alheios ao domínio da aplicação para iniciar os serviços de seleção dinâmica e estabelecer os critérios de seleção. Além disso, os três trabalhos apresentam estratégias proprietárias para a seleção de componentes e mantém seus próprios repositórios. LuaSpace aproveita o suporte dos serviços CORBA e dos seus repositórios.

5. Conclusões

Este trabalho apresentou os mecanismos para seleção dinâmica de objetos distribuídos oferecidos por LuaSpace. Os mecanismos promovem o reuso de componentes uma vez que conferem flexibilidade para o programador expressar facilmente diferentes critérios de seleção e inclusive a combinação de vários critérios usando a mesma linguagem empregada para configurar a aplicação – a linguagem Lua. Desta forma, evita-se que o programador tenha de aprender linguagens específicas de serviços de seleção, de interagir diretamente com estes serviços e de realizar a seleção de componentes de maneira separada da configuração da aplicação.

Em LuaSpace a seleção pode ser realizada por nomes, por propriedades, por assinatura de um método ou de um conjunto de métodos e também considerando qualquer combinação de tais critérios.

Para possibilitar a expressão de diferentes critérios de seleção, LuaSpace fornece dois mecanismos: o conector genérico e a função *search* que recebe como parâmetro uma lista de critérios de seleção e retorna a lista de objetos que satisfazem os critérios. O conector genérico não apenas seleciona componentes mas também ativa a execução de um método sobre determinado componente e devolve o resultado. Portanto, o conector genérico oferece suporte em tempo de execução para inserção automática de componentes em uma aplicação.

Os mecanismos de seleção disponibilizados por LuaSpace estão em um nível de abstração superior ao dos serviços de Nomes e de *Trading* de CORBA. Apesar destes serviços fornecerem operações para busca de objetos baseada em nome e em propriedades, a utilização destas operações envolve um conjunto de comandos específicos de cada serviço, obrigando o programador a conhecer detalhes dos serviços.

Para efetivar a seleção dinâmica, os mecanismos de LuaSpace aproveitam o suporte dos serviços de seleção de objetos e os repositórios de meta-informação oferecidos pela plataforma CORBA. A interação dos mecanismos de LuaSpace com os serviços CORBA é transparente para o programador. Portanto, LuaSpace agrega a funcionalidade dos serviços de Nomes e de *Trading* de CORBA e oferece uma maneira uniforme de se explorar a potencialidade destes serviços.

Em LuaSpace o programador tem a flexibilidade de interagir diretamente com os serviços de seleção de objetos disponibilizados por CORBA. De forma a facilitar esta interação, LuaSpace habilita automaticamente o acesso à biblioteca LuaTrading e aos serviços de Nomes e de *Trading,* tornando-os operacionais para uso a qualquer momento. Assim o programador está liberado da tarefa de emitir os comandos iniciais para habilitação dos serviços e dos seus repositórios.

LuaSpace automatiza também o registro de novos tipos de serviços e a exportação de novas ofertas de serviço para serem armazenadas no repositório do serviço de *Trading*. Esta estratégia garante que todos os componentes CORBA instalados via LuaSpace irão estar publicados, possibilitando assim o reuso.

Outros trabalhos envolvendo suporte para seleção dinâmica de objetos [10,16,17] oferecem soluções proprietárias e limitam-se à realizar a busca considerando apenas um tipo de critério de seleção.

O ambiente LuaSpace com os mecanismos de seleção dinâmica descritos neste trabalho encontra-se operacional na plataforma Linux usando LuaOrb, LuaTrading e a implementação CORBA fornecida pela Iona Technologies - ORBacus [20].

Agradecimentos

Este trabalho foi parcialmente financiado pelo CNPq.

Referências

[1] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley. 1998.

[2] T. Batista, C. Chavez and N. Rodriguez. Dynamic Reconfiguration through a Generic Connector. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00*), vol. II, pages 1127-1132, Las Vegas – Nevada – USA, June 2000. CSREA Press.

[3] T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-based Applications. In *5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'2000),* pages 32-39, Limerick – Ireland, June 2000. IEEE.

[4] T. Batista and N. Rodriguez. Configuração de Aplicações no LuaSpace. *Anais do 18° Simpósio Brasileiro de Redes de Computadores (SBRC 2000),* Belo Horizonte, MG, Maio 2000, pp 169-182.

[5] T. Batista. LuaSpace: Um Ambiente para Reconfiguração Dinâmica de Aplicações baseadas em Componentes. Tese de Doutorado. Departamento de Informática da PUC-Rio. Outubro 2000.

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

[7] R. Cerqueira, C. Cassino and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. *In International Symposium on Distributed Objects and Applications (DOA´99),* 362-371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

[8] P. Merle. CorbaScript and CorbaWeb: Propositions pour l´acces à des objets et services distributes. PhD Thesis, Université des Sciences et Technologies de Lille, Janvier 1997.

[9] C. Krueger. Software Reuse. *ACM Computing Surveys*, Vol. 24, N° 2, June 1992.

[10] VisiBroker for Java 4.5: Programmer's Guide. Available at http://www.borland.com/techpubs/books/vbj/vbj45/programmers-guide/

[11] A. L. Moura, N. Rodriguez. Adaptação Dinâmica de Aplicações Distribuídas. *Anais do 19° Simpósio Brasileiro de Redes de Computadores (SBRC 2001),* Florianópolis, SC, Maio 2001, pp 338-353.

[12] Z. Tari and O. Bukhres. Fundamentals of Distributed Object Systems – The CORBA perspective. John Wiley & Sons. 2001.

[13] R. Orfali, D. Harkey and J. Edward. The Essential Distributed Objects Survival Guide. John Wiley & Sons. 1996.

[14] R. Balter, L. Bellissard, F. Boyer, M. Riveill and J. Vion-Dury. Architecturing and Configuring Distributed Application with Olan. *In Proceedings of IFIP Int. Conf. On Distributed Systems Platforms and Open Distributed Processing (Middleware´98),* The Lake District, Sptember 1998. Available at http://sirac.imag.fr/PUB /98/98-middleware-olan-PUB.ps.gz.

[15] T. Batista and M. Carvalho. Component-Based Applications: A Dynamic Reconfiguration Approach with Fault Tolerance Support. In *Software Composition Workshop (SC) - affiliated to European Joint Conferences on Theory and Practice of Software*

(ETAPS), Grenoble - FR, April 2002. Published in Electronic Notes in Theorical Computer Science, Vol. 65, Number 4, 2002. *http://www.elsevier.nl/locate/entcs/volume65.html*
[16] T. Saridakis, C. Kloukinas and V. Issarny. Fault Tolerant Access to Dynamically Located Services for CORBA Applications. *In Proceedings of Computer Applications in Industry and Engineering (CAINE-99),* 12th Int'l. Conference, pages 21-24 November 4-6, 1999, Atlanta, Georgia, USA
[17] P. Kostkov. MAGNET: A Dynamic Resource Management Architecture. PhD Thesis. Department of Computing. The City University London. July 1999.
[18] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), pages 80-112, January 1985.
[19] V. Issarny, C. Bidan and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype. *In Proceedings of the Fourth International Conference on Configurable Distributed Systems (ICCDS),* pages 207-214, Annapolis, Maryland, May 1998.
[20] ORBacus Home Page – http://www.ooc.com/