Configuração de Aplicações Distribuídas no LuaSpace

Thaís Vasconcelos Batista *

Departamento de Infomática - DIMAp

Universidade Federal do Rio Grande do Norte – UFRN

Noemi Rodriguez

Departmento de Informática

Pontifícia Universidade Católica do Rio de Janeiro – PUC- Rio

thais,noemi@inf.puc-rio.br

Resumo

Os modelos de integração de objetos oferecem comunicação remota entre componentes heterogêneos de maneira transparente para a aplicação. Porém, estes modelos não oferecem facilidades para descrever a configuração de uma aplicação. Este artigo apresenta um ambiente para configuração de aplicações distribuídas baseadas em componentes chamado LuaSpace. LuaSpace é composto por um conjunto de ferramentas baseadas em uma linguagem de extensão interpretada que oferecem mecanismos dinâmicos para composição e evolução de aplicações.

Abstract

Object integration models provide support for remote communication between heterogeneous components in a transparent fashion. These models do not offer facilities to describe application configuration. This paper presents LuaSpace, an environment for configuring component-based applications. LuaSpace is composed by a set of tools based on an interpreted language that offer support for application composition and dynamic reconfiguration.

1 Introdução

Os modelos de integração de objetos, também chamados de infra-estruturas de componentes de software [OHE96] ou simplesmente modelos de componentes [Ber96, Sti94], apresentam abstrações importantes para facilitar o desenvolvimento de aplicações distribuídas. Suporte para comunicação remota entre componentes heterogêneos é o principal aspecto que explica a crescente disseminação desta tecnologia, cujo modelo de comunicação consiste de interações cliente-servidor. CORBA [Sie96], COM [Rog97] e JavaBeans [Jav96] são exemplos de modelos de componentes. CORBA tem se destacado em relação aos demais modelos por ser uma especificação aberta, independente de fabricante e de linguagem. Por este motivo, no contexto deste trabalho, consideraremos CORBA como representante desta categoria.

---

*Aluna de Doutorado da PUC-Rio

Apesar dos modelos de componentes oferecerem abstrações para diminuir as complexidades relativas à programação distribuída e utilização de componentes, eles não apresentam facilidades para a organização global de uma aplicação [BR96]. Para este propósito, são utilizadas linguagens de configuração [MDK93, BWD+93, End94]. O objetivo da linguagem de configuração é permitir a especificação da estrutura do sistema e a definição da interação entre componentes. Componentes são entidades de software compilados separadamente que interagem através de interfaces. Portanto, o programa de configuração usa apenas informações disponíveis na interface. Tipicamente, sistemas baseados em linguagem de configuração usam componentes cuja interface descrevem os serviços oferecidos e requisitados. No caso de CORBA, a interface do objeto não contém informações sobre os serviços requisitados, pois estes fazem parte da implementação do objeto. CORBA enfatiza a separacao entre especificação e implementação. A interface de um objeto CORBA, descrita em IDL, contém apenas informações sobre sua especificação: basicamente, já que se trata de uma arquitetura OO, as assinaturas dos métodos que podem ser chamados. Uma mesma interface pode ser implementada por programas diferentes. Dependendo da implementação utilizada, essa implementação poderá requisitar a presença de diferentes componentes, isto é, poderá conter chamadas a métodos de diferentes interfaces. Assim, no modelo de programação CORBA, a interligação entre objetos é implícita visto que a interação é definida dentro do componente.

Para configurar aplicações usando objetos CORBA, alguns sistemas estendem a interface IDL [BBB+98, IBS98] para possibilitar a definição de ligação explícita [FBC+98] entre componentes a nível de configuração. Apesar da ligação explícita oferecer a visão estrutural da aplicação, alguns trabalhos [DR97, MTK97, SN99] criticam o fato de se estender a interface do componente CORBA, alegando que compromete o desempenho e que utilizar o componente CORBA na sua forma original promove a reusabilidade.

Neste trabalho apresentamos LuaSpace, um ambiente que oferece uma outra abordagem para configuração de aplicações CORBA. LuaSpace é um ambiente para configuração de aplicações através de uma linguagem interpretada e de ferramentas desenvolvidas sobre esta linguagem. A linguagem interpretada utilizada é a linguagem Lua [IFC96]. Esta linguagem foi escolhida por possuir um sistema de tipos flexível que permite acesso dinâmico a objetos e por oferecer um conjunto de ferramentas que juntas oferecem um ambiente poderoso para reconfiguração dinâmica. Além disso, este trabalho está inserido em um projeto que investiga a flexibilidade que uma linguagem interpretada pode conferir a um modelo de componentes [CCI99, CRI99].

LuaSpace oferece suporte para definição da configuração de uma aplicação, além de permitir reconfiguração dinâmica em dois níveis: automático, onde a reconfiguração é transparente para o programador, e programado, onde a reconfiguração é estabelecida explicitamente pelo programador.

Este trabalho está estruturado da seguinte forma. A seção 2 discute o uso de uma linguagem interpretada no contexto de configuração de aplicações e apresenta a linguagem Lua. A seção 3 descreve brevemente as ferramentas que compõem o ambiente e apresenta a arquitetura de LuaSpace. A seção 4 ilustra o desenvolvimento de uma aplicação no ambiente LuaSpace, ressaltando os aspectos de configuração e reconfiguração dinâmica. A seção 5 discorre sobre os trabalhos correlatos que abordam ambientes para configuração de aplicações. Finalmente, a seção 6 apresenta as conclusões.

2 Uma Linguagem Interpretada para Configuração

Uma linguagem de programação é interpretada quando oferece mecanismos para execução de código criado dinamicamente [RI99]. Esta característica confere um estilo interativo à linguagem, facilitando a configuração e reconfiguração de aplicações. Através de um console interativo é possível determinar a execução imediata de comandos, inclusive adicionar e conectar componentes em uma aplicação, sem necessidade de recompilação. É possível também incluir nova funcionalidade na aplicação no decorrer da execução. Desta forma, interatividade pode ser explorada para prototipagem rápida, teste de configurações e para se estabelecer reconfiguração.

Muitas linguagens interpretadas são baseadas em um sistema de tipos dinâmico que elimina a necessidade de declarações de tipos. Este aspecto flexibiliza a ligação em tempo de execução entre componentes não previstos anteriormente, sendo bastante adequado para configuração de aplicações inerentemente dinâmicas. Assim, uma linguagem interpretada pode oferecer uma maneira de configurar aplicações com componentes CORBA sem necessidade de se modificar a interface do componente para incluir os serviços que serão usados.

Linguagens interpretadas e procedurais oferecem um modelo de programação onde não há necessidade de declaração de interconexão entre componentes. Interconexões são representadas por chamadas convencionais de métodos. O estilo procedural permite que se configure uma aplicação através de estruturas de decisão que na execução selecionam um componente ou outro dependendo de condições estabelecidas pelo programador. Em CORBA, as chamadas aos serviços utilizados fazem parte da implementação do objeto, portanto a interconexão com outro componente é definida internamente. Porém, para se estruturar uma aplicação, é necessário também se ter a flexibilidade de estabelecer interconexões entre componentes no programa de configuração da aplicação. Para este fim, pelos motivos apresentados, a linguagem interpretada pode oferecer um importante suporte.

A linguagem Lua [IFC96], utilizada nesse trabalho, possui o perfil discutido acima. Lua é dinamicamente tipada: variáveis não têm tipos, apenas valores é que estão associados a um tipo.

Como em linguagens procedimentais convencionais, Lua oferece estruturas de controle (`while`, `if`, etc.), definições de funções com parâmetros e variáveis locais. Além disso, inclui aspectos não usuais como:

- Funções são consideradas valores de *primeira classe*, o que significa que podem ser armazenadas em variáveis, passadas como argumentos para outras funções e retornadas como resultado. A definição de uma função cria um valor do tipo *function* e atribui este valor a uma variável global. Funções podem retornar vários valores, o que dispensa a passagem de parâmetros por referência quando se precisa obter mais de um retorno de uma função.

- O tipo *table* (tabela) implementa *arrays associativos*, e é o único mecanismo para estruturar dados em Lua. Tabelas são objetos criados dinamicamente e podem ser indexados por qualquer valor da linguagem (inteiros, strings, reais, tabelas e valores de funções), exceto `nil`. Esta flexibilidade é a base do mecanismo para descrição de objetos em Lua. Vários tipos de dados como arrays, conjuntos e registros, podem ser implementados através de tabelas. Tabelas são criadas com expressões especiais chamadas construtores. O construtor mais simples é { }, que cria uma nova tabela

vazia. Como tabelas são objetos, não são valores, variáveis nunca contêm tabelas, apenas referências a elas.
O suporte parcial à programação orientada a objetos em Lua (não permite herança) é oferecido através de funções e tabelas. Como funções são valores de primeira classe e campos de tabelas podem conter funções, a tabela representa o objeto e os métodos são funções armazenadas nos campos da tabela.

- As *facilidades reflexivas* oferecidas por Lua permitem, por exemplo, verificação de tipo através da função *type* que retorna um string descrevendo o tipo do seu argumento. Como registros e objetos são representados por tabelas, pode-se percorrer facilmente todos os campos do objeto ou verificar a presença de determinados campos. *Fallbacks* é o mecanismo mais genérico de reflexão que a linguagem apresenta, permitindo ao programador mudar o comportamento de Lua em condições especiais, como na presença de erros durante a execução ou no acesso a um campo inexistente em uma tabela. Por exemplo, se na expressão `receiver:foo(params)`, *receiver* não for um objeto Lua ou `foo` não for um método de *receiver*, um erro deve ser sinalizado. Com *fallbacks*, o programa pode tratar este erro de maneira significante para o contexto onde ele está inserido. Esta facilidade é a base da implementação de LuaOrb.

Uma outra consequência do uso de uma linguagem interpretada é permitir que trechos de código fonte sejam transmitidos entre processos executando em diferentes máquinas. ALua [UR99] é uma biblioteca associada à linguagem Lua que oferece um mecanismo para comunicação entre processos através de um modelo orientado a eventos. A função `send(processo_destino, mensagem)` é usada para se enviar uma mensagem com um código a ser executado no destino. O código é executado no ambiente global de Lua, portanto, é possível incluir neste código a modificação de variáveis e chamadas de funções, bem como outras ações de reconfiguração da aplicação.

Maiores informações sobre Lua podem ser encontradas em http://www.tecgraf.puc-rio.br/lua/.

3 LuaSpace

LuaSpace é um ambiente de desenvolvimento de aplicações distribuídas baseadas em componentes que segue o modelo de programação orientado à configuração caracterizando-se por separar os aspectos estruturais da aplicação da implementação dos componentes. O objetivo é oferecer suporte para configuração e reconfiguração dinâmica de aplicações.

O ambiente é composto pela linguagem de configuração Lua e por ferramentas baseadas na linguagem. Uma destas ferramentas é LuaOrb [CCI99], um *binding* entre Lua e CORBA, baseado na Interface de Invocação Dinâmica (DII), que disponibiliza o acesso dinâmico a objetos CORBA da mesma maneira que se faz acesso a objetos Lua. Adicionalmente, LuaOrb usa a Interface de Esqueleto Dinâmico (DSI) para permitir via Lua a instalação dinâmica de novos objetos em um servidor em execução. As demais ferramentas usam LuaOrb para acessar objetos CORBA.

3.1 LuaOrb

LuaOrb [CCI99] é um *binding* entre a linguagem Lua e CORBA que, da mesma maneira que todos os *bindings* entre linguagens de programação e CORBA, define mapeamentos entre tipos Lua e IDL. Diferentemente dos demais *bindings*, LuaOrb é baseado no mecanismo de tipagem dinâmica de Lua e na Interface de Invocação dinâmica de CORBA (DII) [Sie96]. Estes aspectos permitem o acesso a objetos CORBA em tempo de execução sem declaração prévia (sem stubs compilados).

Do lado do cliente, LuaOrb aproveita as características dinâmicas de Lua permitindo o acesso a objetos remotos CORBA, da mesma maneira que se faz acesso a objetos Lua. Além disso, o acesso é feito de forma dinâmica e transparente. Para usar um objeto CORBA, um *proxy* (representante) deve ser criado usando o construtor `createproxy`. Esta função retorna um objeto Lua que representa o objeto CORBA. Através do mecanismo de *fallbacks*, LuaOrb intercepta as operações aplicadas sobre o *proxy* e as transforma em operações remotas. DII permite diferentes modos de invocação e o LuaOrb permite que o programador os use. O modo de chamada síncrona adiada (com espera adiada) é disponível apenas via DII. Neste modo, o cliente pode chamar um método e continuar sua execução sem esperar pela finalização do método, solicitando o resultado mais tarde. Para usar este modo, o prefixo `deferred_` deve ser incluído antes do nome da operação. A chamada do método retorna um *handler* que pode ser usado para recuperar o resultado da execução do método. Como este tipo de chamada é bastante útil para programação orientada a eventos, LuaOrb oferece a função `completion_event` para observar a finalização da execução do método especificado na chamada adiada. Esta função tem dois parâmetros, o handler retornado pela chamada síncrona adiada e uma função que será executada depois da finalização do método, recebendo como parâmetro o resultado do método da chamada adiada. Por exemplo, na chamada `completion_event(paciente:deferred_monitora(), print)`, quando o método `monitora` é finalizado, a função `print` é invocada tendo como parâmetro o componente retornado por `monitora`.

Do lado do servidor, LuaOrb usa as facilidades oferecidas pela DSI (Dynamic Invocation Interface) de CORBA para permitir a extensão de objetos CORBA via Lua [CRI99]. Objetos Lua podem fazer o papel de servidores CORBA e clientes podem acessá-los via stubs ou DII. Analogamente ao que acontece do lado do cliente, é criado um *proxy* para o objeto Lua. Este objeto pode ser dinamicamente instalado no servidor.

3.2 Conector Genérico

O conector genérico [BCR00] é um mecanismo que dinamicamente seleciona componentes para executar serviços requisitados por uma aplicação. Usando este mecanismo, uma aplicação pode ser configurada declarando um conjunto de serviços sem estabelecer os componentes que oferecem tais serviços. Os componentes serão selecionados *on-the-fly* pelo conector genérico.

O conector genérico é um objeto Lua criado pela função `generic_createproxy()`. No programa de configuração, depois da criação do conector, o serviço pode ser chamado da seguinte forma: `nome_proxy:serviço(parâmetros)`. Na execução esta chamada é interceptada pelo conector genérico que, implicitamente, aciona a função `connect(servico, parâmetros, nome_solicitante)`, cuja finalidade é procurar em um repositório padrão (um serviço de Nomes ou um serviço de Trader) pelo componente que oferece `servico(pa-`

`râmetros)`. Uma vez encontrando vários componentes que oferecem `servico(parâmetros)`, um deles é selecionado (para isto pode ser solicitada a intervenção do usuário). Em seguida, o Conector Genérico constrói a solicitação, que consiste em "montar" uma sequência de comandos Lua para criação do *proxy* do componente e ativação do serviço. Finalmente o serviço é ativado usando LuaOrb.

A *tabela de configuração* mantida pelo Conector Genérico contém registros com os seguintes campos: `(Interface do Servico, Servidor que o oferece, Cliente que solicitou)`, onde `Interface do Servico` é o par `(nome, assinatura)`. Antes de procurar `servico(parâmetros)` no repositório padrão, o Conector Genérico consulta esta tabela para verificar se o mesmo serviço já foi encontrado anteriormente, evitando acesso ao repositório. Por outro lado, existe a flexibilidade de se estabelecer que toda busca deve ser feita sobre o repositório. Neste caso, a tabela de configuração não é consultada. Desta forma, como a seleção do componente é resolvida em tempo de execução, este mecanismo consiste em uma maneira de reconfigurar a aplicação visto que, a cada chamada de um mesmo serviço, um componente diferente pode ser selecionado para executá-lo. Como consequência, com o uso do conector genérico, configuração e reconfiguração estão misturadas no programa de configuração. O processo de busca do componente procede de maneira transparente. Desta forma, o conector genérico oferece suporte em tempo de execução para a inserção automática de componentes em uma aplicação.

O Conector Genérico é uma abstração sobre os serviços de busca de objetos como serviço de Nomes ou serviço de Trader. Sua função ultrapassa a destes serviços que são apenas repositório de informações sobre componentes e oferecem formas de consulta sobre este repositório. Ele confere à aplicação capacidade de reconfiguração e uso de serviços de componentes não determinados da mesma maneira que se usa serviços de componentes conhecidos. Além disso não apenas seleciona componentes que podem oferecer os serviços requisitados pela aplicação, mas também ativa o serviço e devolve o resultado.

O Conector Genérico apresenta semelhanças com o símbolo genérico descrito em [End94]. O símbolo genérico é uma variável de reconfiguração cujo valor é um objeto de configuração encontrado em tempo de execução seguindo condições estabelecidas no programa de configuração. Na execução, sempre que este símbolo é referenciado, o objeto ligado à ele é usado em seu lugar. A denominação "genérico" provém do fato de nenhum objeto concreto estar associado à este símbolo em tempo de design. O Conector Genérico funciona de maneira similar.

3.3 Meta-interface

A meta-interface [Cat98] é um mecanismo usado para introduzir dinamicamente objetos Lua e serviços sem necessidade de modificar o repositório de interfaces de CORBA. A meta-interface é uma interface IDL que define um servidor genérico ("servidor de servidores") para englobar objetos. O acesso a objetos em servidores é realizado através de um conjunto de operações para chamar funções, escrever e ler atributos. Apesar de poder ser usado por qualquer cliente CORBA, o acesso a meta-interface para um cliente Lua é transparente e implementado através de *fallbacks*. As principais facilidades oferecidas pela meta-interface são identificação dinâmica de interfaces de objetos e instalação dinâmica de novos objetos. A instalação dinâmica de novos objetos em servidores compreende a criação de um objeto remoto e a associação do objeto remoto a um servidor

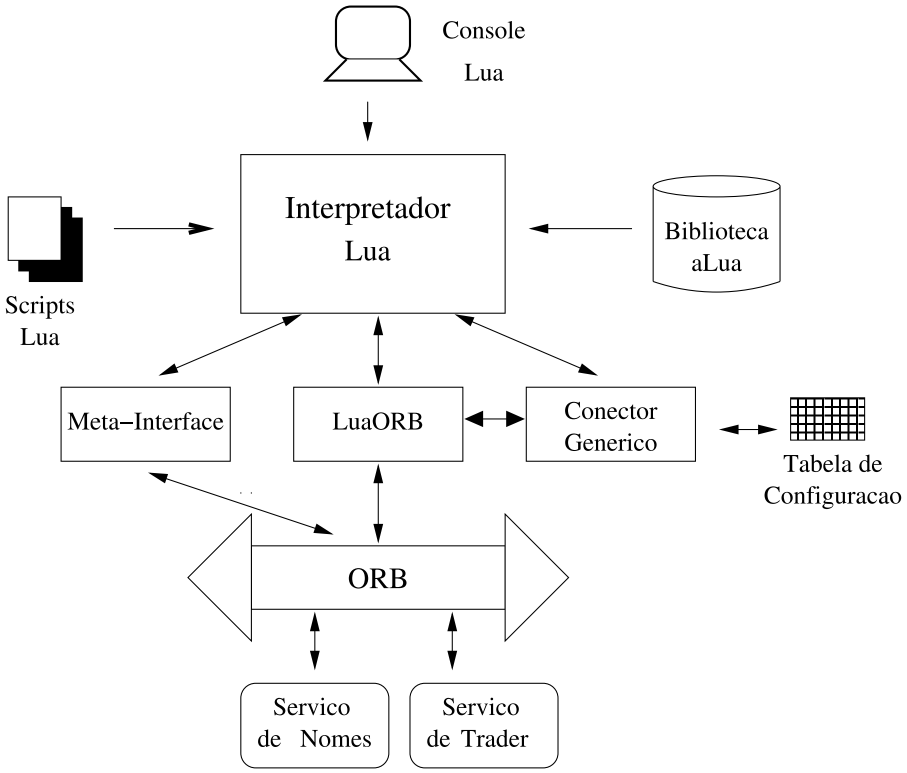

Figura 1: Arquitetura de LuaSpace

genérico. A meta-interface dispõe de métodos para identificação dinâmica de objetos existentes no servidor. O retorno destes métodos é uma estrutura contendo a descrição dos atributos e métodos do objeto.

3.4 Arquitetura

A Figura 1 mostra as ferramentas que compõem o ambiente LuaSpace. As ferramentas são desenvolvidas em Lua e usam LuaOrb para acessar objetos CORBA.

LuaSpace permite a construção interativa de aplicações através do Console Lua, uma interface orientada a comandos que funciona de forma similar a um shell Unix. O usuário utiliza o Console Lua para emitir os comandos a serem processados pelo interpretador Lua. O interpretador Lua analisa o comando recebido e, em alguns casos, a facilidade reflexiva oferecida pelo mecanismo de *fallbacks* é acionada para tratar o comando. Como cada objeto Lua tem uma *fallback* associada, a semântica de qualquer operação pode ser modificada e uma nova semântica ser associada à *fallback* da operação. Por exemplo, para chamada de métodos sobre *proxies* de objetos CORBA, a *fallback* define que LuaOrb deve tratar a chamada. O conector genérico também é acionado de maneira similar: em uma chamada de método sobre o *proxy* para o conector genérico (criado pela função `generic_createproxy`), a *fallback* repassa a chamada para o conector genérico. O conector genérico através do LuaOrb faz acesso a objetos e serviços CORBA. Na busca por componentes que oferecem determinados métodos, o conector genérico usa um serviço de catálogo de objetos (como o serviço de Nomes ou o serviço de Trader). LuaSpace permite que o programador estabeleça qual o serviço de catálogo deve ser usado pelo conector genérico. A tabela de configuração mantém o registro das conexões estabelecidas pelo conector genérico. O mecanismo de *fallbacks* também é responsável por direcionar as chamadas para a meta-interface, no caso da instalação dinâmica de objetos Lua em servidores CORBA.

O usuário pode também desenvolver o programa de configuração da aplicação em um script Lua e submeter este script ao interpretador Lua. O comportamento padrão do interpretador é executar o arquivo contendo o script de configuração. O script Lua pode ser chamado pelo usuário usando o console Lua, pode ser chamado pela aplicação em execução, ou ainda, ser enviado em uma mensagem, para o processo em execução, usando o mecanismo de envio de mensagens oferecido por ALua. A facilidade de execução de scripts e de envio de scripts em uma mensagem são aspectos importantes para a reconfiguração de uma aplicação.

4 Definindo uma Aplicação no LuaSpace

Para ilustrar a utilização das ferramentas disponíveis no ambiente LuaSpace, abordaremos a configuração e reconfiguração do Sistema de Monitoração de Pacientes (SMP), um exemplo clássico da literatura [KM85]. Originalmente este sistema foi configurado seguindo a estrutura estabelecida pelo paradigma da configuração, sendo a ligação entre componentes explicitamente declarada no programa de configuração. Nesta seção o SMP será modelado seguindo o modelo de programação procedural disponível no LuaSpace.

O SMP é composto por componentes *CentralEnfermagem* e *Paciente*. Cada *Paciente* monitora sensores ligados a um paciente e emite um alarme quando existem dados fora dos limites estabelecidos previamente. O módulo *CentralEnfermagem* recebe alarmes e, algumas vezes, solicita os dados vitais do paciente (representados na `struct status`). A Figura 2 mostra a interface IDL deste sistema.

```
struct status{
    string nome;
    short pressao_max;
    short pressao_min;
    short temperatura;
    short pulso;
}
interface Paciente{
   status statuscorrente();
   Paciente monitora();
}
interface CentralEnfermagem{
   void alarme(in Paciente);
}
```

Figura 2: Interfaces IDL do Sistema de Monitoração de Pacientes.

A figura 3 ilustra uma possível configuração do SMP. Neste exemplo, uma função Lua é passada como segundo argumento da função `completion_event`. Na finalização do método `monitora` a função `trataevento` é chamada. Esta função determina a execução do método `alarme` do componente *CentralEnfermagem* e incrementa o número de alarmes emitidos pelo *Paciente*. Caso o alarme seja repetido cinco vezes, supõe-se que é uma situação de extrema emergência e, para tratá-la, é invocado o método `urgência` de um componente a ser encontrado pelo Conector Genérico. Após tratar o evento recebido, a função `trataevento` faz a chamada ao método `monitora` novamente. Isto ilustra uma situação comum onde depois de tratar um evento, um programa redefine um *handler* para a próxima ocorrência do mesmo evento. A função `trataevento` atua como um mecanismo de interconexão de componentes, uma vez que, na finalização do método `monitora` do componente *Paciente*, esta função é chamada para determinar os componentes que executarão o serviço necessário para atender a solicitação do método `monitora`.

```
pac = 1
while pac<5 do
   p = createproxy(``Paciente'', pac)
   c = generic_createproxy()
   completion_event(p:deferred_monitora(), trataevento)
   pac = pac + 1
end

numalarme = 0

function trataevento(p)
   enf = createproxy(``CentralEnfermagem'')
   enf:alarme(p)
   numalarme = numalarme + 1
   if numalarme > 5 then
      c:urgencia(p)
   end
   completion_event(p:deferred_monitora(), trataevento)
end
```

Figura 3: Programa de Configuração do SMP

Neste exemplo, a capacidade de reconfiguração da aplicação é conferida pelo Conector Genérico e pelo comando condicional (if) que, tendo a condição satisfeita, determina a seleção de um componente que ofereça o método `urgência`. O uso destes recursos é bastante conveniente para uma aplicação como o SMP onde é importante que o paciente possa ser atendido o mais rápido possível por qualquer uma das entidades com competência para tal. Este exemplo ilustra o contraste entre a abordagem de configuração convencional, que interliga componentes diretamente sem conhecimento dos dados que fluem entre os componentes, e a flexibilidade do LuaSpace para analisar os valores gerados pelos componentes. Em geral, diferentes valores necessitam diferentes serviços. No programa de configuração Lua, os dados podem ser analisados para decidir qual serviço deve ser chamado. Como discutido em [PS93] isto adiciona flexibilidade à configuração. Além disso, em casos onde há incompatibilidade entre componentes, o programa Lua pode realizar a adaptação necessária. Por exemplo, suponha que o método `alarme` da interface *CentralEnfermagem* necessite de um segundo parâmetro, do tipo `status`, descrevendo o status do paciente. Como o método `monitora` não retorna este dado, não seria possível interconectar os dois componentes diretamente. A adaptação pode ser facilmente realizada pelo programa Lua, que pode chamar o método `statuscorrente` da interface *Paciente*. Para transformações mais complexas, um outro componente poderia ser chamado [SN99].

Este exemplo ilustra como os níveis de reconfiguração disponíveis no LuaSpace são usados em uma aplicação. A reconfiguração programada é estabelecida pelo comando condicional. A reconfiguração automática é realizada pelo Conector Genérico que, em diferentes execuções da aplicação, pode selecionar diferentes componentes para executar a chamada `c:urgencia(p)`.

```
interface Log{
   void getdados(in Paciente);
   seqdados historico(in Paciente);
```

Figura 4: Interface IDL do componente Log

Supondo a necessidade de se manter um histórico de todos os pacientes monitorados no SMP, pode-se incorporar um componente para registrar periodicamente os dados monitorados de cada paciente. O componente *Log*, cuja interface está descrita na figura 4, oferece o método `getdados` para registrar periodicamente os dados de status do paciente e oferece o método `historico` para consulta dos dados de status de determinado paciente.

```
l = createproxy(''Log'')
pac = 1
while pac<5 do
   l:deferred_getdados(p)
   pac = pac + 1
end
```

Figura 5: Script "reconf.lua"- Reconfiguração do SMP

Para reconfigurar a aplicação, inserindo o componente *Log*, o script descrito na figura 5 pode ser enviado em uma mensagem para o processo que corresponde ao SMP em execução (no exemplo, denominado processo A), conforme ilustra a figura 6. No script de reconfiguração a chamada adiada aplicada ao método `getdados` introduz um estilo de programação assíncrono, tornando possível a execução em paralelo do método `getdados` para todos os pacientes. Como o script de reconfiguração será executado no ambiente global de execução do SMP ilustrado na figura 3, a variável `p`, que faz parte deste ambiente, pode ser usada como parâmetro do método `getdados`.

```
msg = [[dofile(''reconf.lua'')]]
send('processo A', msg)
```

Figura 6: Mensagem de Reconfiguração

A figura 7 exemplifica a instalação dinâmica do componente *Log* e do método `historico` em um servidor usando a meta-interface. O exemplo ilustra a facilidade que a meta-interface introduz para instalar um novo objeto (*Log*), em um servidor remoto (no caso, *genserver*), realizando operações sobre objetos locais. Acesso à tabela local (*serverRoot*) implicará em uma operação equivalente no servidor.

```
serverRoot = OM_Bind(genserver) -- associa a tabela local ao servidor
Log = OM_CreateTable(genserver) -- cria um objeto Log no servidor
Log.historico = OM_CreateFunction(genserver, -- define um metodo historico
        [[function (number)
           data = tabpatient[number]
           return data
        end ]] );
serverRoot.Log = Log -- associa o objeto Log a tabela
```

Figura 7: Instalação Dinâmica do Componente Log usando a meta-interface

5 Trabalhos Correlatos

Outros trabalhos envolvendo ambientes de suporte a reconfiguração de aplicações usando linguagem de configuração, como os ambientes OLAN [BBB+98], Aster [IBS98] e Ábaco [SO98], são baseados no modelo de configuração declarativo estabelecido pelo paradigma da configuração. O ambiente apresentado neste trabalho oferece uma abordagem diferente, utilizando uma linguagem de configuração interpretada cujo modelo de programação procedural é usado para descrever uma aplicação. Seguindo este modelo, a ligação entre componentes é implícita, uma vez que não há comandos explícitos como *bind* ou *link*. Como consequência, não há uma forte distinção entre configuração e reconfiguração de uma aplicação. Comandos condicionais (if) ou de iteração (while) no programa de configuração podem dinamicamente determinar a seleção de novos componentes de acordo com condições satisfeitas em tempo de execução. Como a linguagem de configuração do LuaSpace é dinamicamente tipada, não é necessária a declaração prévia das instâncias dos componentes que serão usados no programa. Este aspecto permite a inserção dinâmica de novos componentes na configuração. LuaSpace oferece suporte as mesmas tarefas de reconfiguração oferecidas pelos ambientes de configuração tradicionais (inserção e remoção de componentes e interconexões entre eles) porém segue um modelo diferente, onde interconectar e desconectar componentes acontece de maneira implícita.

O OLAN define uma linguagem para descrição do componente, a linguagem OIL (Olan Interface Language), que é uma extensão da IDL, e uma linguagem para descrição da aplicação, a linguagem OCL (Olan Configuration Language). A interface do componente inclui as provisões e os requisitos do componente. O ambiente é baseado em stubs e wrappers gerados pelo compilador da linguagem.

O ambiente Aster caracteriza-se por usar um *runtime system* dedicado, adaptado para cada aplicação. Em contraste, LuaSpace usa LuaOrb, que é desenvolvido sobre a plataforma CORBA. Como o gerenciamento da interoperação entre componentes é realizado por um sistema *runtime*, o objetivo do Aster é oferecer um ambiente para facilitar a adaptação do sistema *runtime* de acordo com os requisitos da aplicação. O ambiente consiste da linguagem Aster, para descrição dos componentes da aplicação, e de um conjunto de ferramentas para adaptação do sistema *runtime*. Os requisitos da aplicação são especificados usando a linguagem Aster. O sistema *runtime*, adaptado para atender aos requisitos da aplicação, é construído sobre um ORB.

No Ábaco o objetivo é preservar as características do paradigma da configuração e da plataforma CORBA e uní-las. Para isto, modela-se componentes encapsulando a sua descrição CORBA em uma descrição de acordo com a estrutura estabelecida pelo paradigma da configuração. Desta forma, um componente no ABACO é descrito pela linguagem CDL (Component Description Language). Para estruturação da aplicação o ambiente oferece a linguagem de configuração CCL (Component Configuration Language).

Um problema comum dos três ambientes (Olan, Aster e Ábaco) é definir uma linguagem específica do ambiente para a descrição dos componentes. A linguagem proprietária é uma extensão da IDL cujo objetivo principal é incluir a declaração dos serviços requisitados pelo componente. Além disso, a maioria dos ambientes utiliza uma abordagem estática, uma vez que a construção de uma aplicação utiliza ferramentas que implementam verificação estática de tipos.

O estilo de configuração de LuaSpace é compatível com o modelo de programação CORBA, que não declara na interface do objeto os serviços requisitados. Em geral, os trabalhos em configuração de aplicações baseadas em componentes optam por estender o modelo CORBA para ser possível definir explicitamente a ligação entre objetos no programa de configuração. Em contraste, LuaSpace permite o uso do objeto CORBA descrito em IDL, sem extensões. Portanto, o ambiente segue a filosofia dos modelos de componentes de manter a independência entre a linguagem de descrição dos componentes (IDL) e a linguagem usada no desenvolvimento da aplicação. Além disso, promove a separação entre a implementação e a especificação do componente.

6 Conclusão

Neste trabalho apresentamos LuaSpace, um ambiente que dispõe de um conjunto de ferramentas para configuração e reconfiguração dinâmica de aplicações baseadas em componentes. Como nos demais ambientes de programação baseados em componentes, LuaSpace usa uma linguagem de configuração. Diferentemente da maioria dos sistemas, a linguagem é de propósito geral com as vantagens de uma linguagem de programação completa. Esta linguagem é baseada em um modelo de programação procedural, além de ser interpretada, aspectos que permitem a introdução dinâmica de componentes na aplicação sem necessidade de recompilação do programa de configuração. Este modelo é mais flexível que o modelo de configuração tradicional que impõe a declaração prévia de todos os componentes que compõem uma aplicação. Por isto, é bastante compatível com a filosofia de CORBA, cujos objetos são descritos por interfaces que contém assinaturas de métodos e não declaram os serviços requisitados. Os componentes que executam os serviços requisitados podem ser selecionados em tempo de execução.

Devido à integração da linguagem Lua com o modelo CORBA, LuaSpace utiliza o suporte à heterogeneidade e distribuição oferecido por CORBA, permitindo que os componentes sejam implementados em qualquer linguagem que tenha o *binding* para CORBA. Através do exemplo modelado na seção 4 pode-se comprovar o poder expressivo e a versatilidade do LuaSpace para modelagem de aplicações.

Uma vantagem deste ambiente é oferecer suporte à configuração de aplicações em dois níveis. Em um nível o programador determina os componentes que oferecem os serviços utilizados. No outro nível, mais automático, o programador especifica apenas os serviços que compõem uma aplicação e o Conector Genérico torna transparente as complexidades relacionadas à busca de componentes. Este aspecto é especialmente interessante para atender programadores de diferentes níveis técnicos. Programadores avançados podem usar o console Lua para configurar suas aplicações e adaptar a funcionalidade do Conector Genérico (programar o comportamento do Conector no caso de encontrar vários componentes que oferecem o serviço). Programadores menos técnicos podem utilizar o Conector Genérico com seu comportamento padrão.

Outra vantagem de LuaSpace é permitir, em tempo de execução, realizar mudanças tanto na estruturação da aplicação quanto na implementação dos componentes, além de permitir a instalação dinâmica de componentes em um servidor remoto.

As ferramentas que compõem o ambiente descrito neste trabalho encontram-se todas operacionais e integradas, com exceção de ALua, que, apesar de estar operacional e ter sido testada em separado, não está ainda integrada ao ambiente LuaSpace. Os exemplos ilustrados neste artigo foram testados em uma plataforma linux.

Referências Bibliográficas

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J. Vion-Dury. Architecturing and Configuring Distributed Application with Olan. In *Proceedings of IFIP Int. Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, 15-18 September 1998. available at http://sirac.imag.fr/PUB/98/98-middleware-olan-PUB.ps.gz.

[BCR00] T. Batista, C. Chavez, and N. Rodriguez. Conector Genérico: Um Mecanismo para Reconfiguração de Aplicações Baseadas em Componentes. In *Terceira Conferência Ibero-Americana de Engenharia de Requisitos e Ambientes de Software (IDEAS 2000)*, pages 253–264, Cancun, México, April 2000.

[Ber96] P. Bernstein. Middleware. *Communications of the ACM*, 39(2), February 1996.

[BR96] L. Bellissard and M. Riveill. Constructions des applications réparties. Ecole Placement Dynamique et Répartition de Charge, Juillet 1996.

[BWD+93] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra: a structure description language for developing distributed applications. *Software Engineering Journal*, pages 83–94, March 1993.

[Cat98] M. A. Catunda. Extensão Dinâmica de Agentes CORBA. Master's thesis, Departamento de Informática - PUC-Rio, Setembro 1998.

[CCI99] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*, pages 362–371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

[CRI99] M. Catunda, N. Rodriguez, and R. Ierusalimschy. Dynamic Extension of CORBA Servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, Aug 1999. Springer-Verlag. LNCS 1685.

[DR97] S. Ducasse and T. Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC/FSE'97, LNCS 1301*, pages 483–500, 1997.

[End94] M. Endler. A Language for Generic Dynamic Configurations of Distributed Programs. In *Anais do XII Simpósio Brasileiro em Redes de Computadores - SBRC*, pages 175–187, Curitiba - PR, 1994.

[FBC+98] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting Adaptative Multimedia Applications through Open Bindings. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 128–135, Annapolis, Maryland, May 4-6 1998.

[IBS98] 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*, pages 207–214, Annapolis, Maryland, May 4-6 1998.

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

[Jav96] JavaSoft. Javabeans, version 1.00-a, December 1996. http://java.sun.com/beans.

[KM85] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424–435, April 1985.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *IEE Software Engineering Journal*, 8(2):73–82, March 1993.

[MTK97] J. Magee, A. Tseng, and J. Kramer. Composing Distributed Objects in CORBA. In *Third International Symposium on Autonomous Decentralized Systems - ISADS 97*, pages 9–11, Berlin, Germany, April 9-11 1997.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons., 1996.

[PS93] V. Paxson and C. Saltmarsh. Glish: a user-level software bus for loosely-coupled distributed systems. In *1993 Winter USENIX Technical Conference*, 1993.

[RI99] N. Rodriguez and R. Ierusalimschy. Dynamic reconfiguration of CORBA-based applications. In Jan Pavelka, Gerard Tel, and Miroslav Bartošek, editors, *SOFSEM'99: 26th Conference on Current Trends in Theory and Practice of Informatics*, pages 95–111, Milovy, Czech Republic, 1999. Spinger-Verlag. (LNCS 1725).

[Rog97] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[Sie96] J. Siegel. *CORBA: Fundamentals and Programming*. John Wiley & Sons, 1996.

[SN99] J. Schneider and O. Nierstrasz. Components, scripts and glue. In John Hall Leonor Barroca and Patrick Hall, editors, *Software Architectures - Advances and Applications*. Springer 1999. available at http://www.iam.unibe.ch/cgi-bin/.

[SO98] C. Souza and M. Oliveira. ABACO: Um Ambiente de Desenvolvimento baseado em Objetos Distribuídos Configuráveis. In *Anais do XII Simpósio Brasileiro de Engenharia de Software*, pages 205–220, Maringá - PR, 1998.

[Sti94] J. Stikeleather. Why Distributed Computing is inevitable. *Object Magazine*, pages 35–39, March 1994.

[UR99] C. Ururahy and N. Rodriguez. ALua: An event-driven communication mechanism for parallel and distributed programming. In *Proceedings of PDCS'99*, Fort Lauderdale - Florida, 1999.