Configuração de Aplicações baseadas em CORBA

Thaís Vasconcelos Batista
Departamento de Informática e Matemática Aplicada - DIMAP
Universidade Federal do Rio Grande do Norte - UFRN
thais@dimap.ufrn.br

Noemi de La Rocque Rodriguez
noemi@inf.puc-rio.br

PUC-RioInf.MCC15/99
Julho, 1999

Abstract: Recently much effort has been devoted to realize the ideal of building applications using plug-and-play software components. The development of this kind of applications separates components programming from the description of the application structure (configuration programming). For interoperability, standardization infra-structures offer support for heterogeneous components to work together. In this class, the CORBA standard is playing an important role. Configuration of CORBA applications is not in line with proposals in the configuration paradigm because the component interface does not describe the required services. Also, there is a need for dynamic access to new components. This work describes how an interpreted language can deal with these issues, acting as a glue language between components and supporting dynamic reconfiguration of CORBA based applications.

Keywords: CORBA components, interpreted configuration languages, configuration specification, components interation mechanisms, dynamic reconfiguration.

Resumo: Ultimamente tem se investido grandes esforços em concretizar o ideal da construção de aplicações através de componentes de software *plug-and-play*. Uma maneira natural de se desenvolver estas aplicações consiste em separar a programação dos componentes do aspecto estrutural da aplicação - sua configuração. Para interoperabilidade entre componentes heterogêneos tem se desenvolvido infra-estruturas de padronização onde se destaca o padrão CORBA. A configuração de aplicações CORBA não segue a linha tradicional do paradigma da configuração uma vez que na interface os componentes não declaram os serviços requisitados. Há necessidade de mecanismos de configuração que permitam acesso dinâmico a componentes. Linguagens interpretadas atendem à este propósito e apresentam características que lhe conferem o potencial para integrar componentes. Neste trabalho pretendemos endereçar o suporte para reconfiguração dinâmica de aplicações CORBA através da linguagem interpretada Lua.

Palavras-chave: componentes CORBA, linguagens de configuração interpretadas, especificação da configuração, mecanismos de interação entre componentes, reconfiguração dinâmica.

1 Introdução

A necessidade de reuso de software de forma a reduzir o custo e o tempo de desenvolvimento fez emergir a idéia de usar os conceitos e mecanismos de composição de hardware na produção de software, influenciando o surgimento da Engenharia de Software baseada em Componentes, cujo ideal é a construção de aplicações através de componentes de software *plug-and-play*[35]. Não há um consenso geral sobre conceito de Componente na Engenharia de Software, porém vários trabalhos adotam conceitos correlatos com diferenças sutis. No contexto deste trabalho, consideraremos que *Componentes* são módulos ou estruturas de dados que são escritos e compilados separadamente e unidos para formar aplicações [10], interagindo entre si através de interfaces. A interface de um componente é um conjunto de pontos de interação entre ele e o mundo externo [22]. Componentes apenas conhecem os serviços oferecidos por outros componentes (suas interfaces), sem conhecer como eles oferecem tais serviços (sua implementação). O fato de expor apenas a interface garante a substitutibilidade da implementação atrás da interface, tornando possível o ambiente de software de componentes *plug-and-play*. Programar aplicações usando componentes elimina a necessidade de compilar as partes reusadas e torna mais fácil a reconfiguração da aplicação [29].

Para compor software da mesma maneira que hardware, um sistema para composição de software deve suportar a integração de componentes de diferentes origens e deve oferecer aos desenvolvedores das aplicações um ambiente interativo. Tal ambiente deve permitir programar componentes, integrá-los e usá-los para construir grandes aplicações. Além disso, um serviço de comunicação é necessário para que mensagens sejam enviadas de um componente para outro. Os componentes devem ser desenvolvidos seguindo um modelo de objetos padrão e deve haver uma forma padrão de se obter informações sobre os componentes. Estes aspectos são oferecidos por Infra-Estruturas de Componentes de Software, também chamadas de Modelos Integradores de Objetos e Modelos de Componentes.

Estes modelos definem um conjunto de abstrações a serem seguidas por todos os componentes para que eles possam ser desenvolvidos separadamente, em diferentes linguagens, executando em vários processadores, subordinados à diferentes Sistemas Operacionais, e mesmo assim, possam ser integrados. Ou seja, incluem suporte aos problemas inerentes à execução de um serviço distribuído, como: nomes, heterogeneidade e transparência de comunicação entre os objetos. Para este suporte, o modelo de componentes especifica interfaces em termos de um formato independente de linguagem, que é expressado via uma Linguagem para Definição de Interfaces (IDL). Definições de Interfaces são tipicamente registradas em um repositório dedicado. Aplicações clientes podem interrogar este repositório para descobrir as interfaces exportadas pelos objetos. O modelo também especifica como implementações são armazenadas independentemente das suas interfaces. Finalmente, ele especifica como solicitações de clientes são ligadas a implementações para realizar os serviços invocados. Ou seja, em termos gerais, dois aspectos oferecem o suporte à heterogeneidade e distribuição de componentes: o uso de interfaces de programação padrão e de protocolos padrão para comunicação entre plataformas diferentes. São exemplos de modelos de componentes de software CORBA (Common Object Request Broker Arquitecture) [34], COM (Component Object Model) e JavaBeans. O padrão CORBA tem se destacado em relação aos outros devido à sua independência de fabricante e linguagem. O JavaBeans trata apenas de componentes Java e o COM foi desenvolvido pela MicroSoft e é direcionado para ambientes deste fabricante. Portanto, consideraremos o CORBA para fins modelagem de componentes neste trabalho.

Seguindo a proposta da Engenharia de Software baseada em Componentes, as aplicações são estruturadas como um conjunto de componentes que se comunicam via suas interfaces. Desta forma, é natural separar a tarefa de desenvolvimento em dois níveis: o nível básico, chamado *nível do kernel*, contendo os componentes básicos do sistema, neste trabalho serão usados componentes CORBA, que em geral são programados em uma linguagem compilada; e o *nível de Configuração*, responsável pelo aspecto estrutural do sistema, conectando os componentes do kernel para dar a forma final ao sistema [18]. A complexidade do nível de kernel está nos algoritmos e estruturas de dados usados na programação dos componentes; no nível da configuração a complexidade está no gerenciamento das conexões entre os componentes. A configuração de um sistema pode ser estática ou dinâmica. A configuração estática é aquela não muda durante a execução. Neste caso, modificar alguma parte significa interromper a execução e reiniciá-la posteriormente. A configuração dinâmica permite a evolução do sistema durante a execução sem necessidade de recompilação.

No nível da configuração, é bastante explorado o paradigma da configuração, inicialmente proposto por [18] e amplamente usado em diversos trabalhos [3, 27], que oferece suporte para a interconexão entre os componentes do kernel. Diferentemente dos componentes considerados pelos modelos integradores, neste paradigma os componentes explicitamente declaram na sua interface *portas de saída*, através das quais mensagens podem ser enviadas, e *portas de entrada* nas quais mensagens podem ser recebidas. Desta forma, a interconexão entre os componentes é realizada por uma linguagem de configuração associando-se as portas de saída de uma instância do componente às portas de entradas de outras instâncias de componentes.

Em relação aos componentes, no modelo CORBA a sua interface e estrutura são especificados em IDL e as comunicações são mediadas pelo ORB, um middleware [5] que estabelece um modelo de interação cliente-servidor entre os componentes de diferentes origens. Na interface, os componentes CORBA apenas declaram os serviços oferecidos (equivalentes às *portas de entrada* do paradigma da configuração), ou seja, não se conhece os métodos que serão chamados pelos componentes [31]. O CORBA segue o modelo cliente-servidor onde os componentes são estruturados em termos dos serviços que oferece, e apenas na implementação são decididos quais os serviços utilizar. Como na filosofia do CORBA clientes veêm apenas a interface do objeto e os aspectos de implementação não são tratados na interface, informações sobre tais aspectos não estão disponíveis para os clientes. Assim, para controlar a configuração das aplicações com componentes CORBA não pode ser usado o mesmo modelo estabelecido pelo paradigma da configuração, pois não se sabe a priori os serviços requisitados, não se podendo fazer a interligação das interfaces antes da sua execução.

A plataforma CORBA oferece suporte suficiente para descrever interações elementares entre grupos isolados de componentes de uma aplicação, mas não dispõe de facilidades para descrever a organização global de uma aplicação em termos de componentes e esquemas de interação entre eles [4]. É necessário então algum mecanismo para dar apoio a estruturação da aplicação - sua configuração. Em determinadas classes de aplicações críticas como Controle de Tráfego Aéreo, Sistemas de Automação Bancária e Sistemas de Telefonia encerrá-las e reiniciá-las para modificações acarreta atrasos inaceitáveis, aumenta o custo e o risco, podendo ter sérias repercussões. O caráter dinâmico destas aplicações requer que os mecanismos de configuração ofereçam suporte à modificações no progresso da execução, como inclusão e remoção de componentes e modificação nas conexões entre eles. Como as interações entre os componentes CORBA não são previsíveis, é necessária uma política de gerenciamento de configuração direcionada para tratar este problema. Este é o aspecto da reconfiguração dinâmica que pretendemos endereçar neste trabalho. Diversas operações são categorizadas como operações de reconfiguração [25]: adicionar componentes, remover componentes, trocar de componentes, mudar a topologia da arquitetura adicionando ou removendo conexões entre componentes, alterar o mapeamento de componentes a elementos de processamento, mover o elemento de processamento associado ao componente, etc. Estamos interessados em endereçar especificamente o problema da interconexão entre componentes CORBA.

Várias técnicas para definição estática da estrutura de softwares estão disponíveis na literatura. Com a importância da evolução dos sistemas se tornando evidente, mecanismos dinâmicos para gerenciamento da configuração tem sido alvo de recentes pesquisas [11, 15, 27] porém, poucos trabalhos têm se dedicado ao tema de reconfiguração dinâmica de aplicações formadas por componentes CORBA [31]. Segundo Merle [13], os mecanismos dinâmicos são complexos para implementar em um ORB e devido a este problema, poucos trabalhos de pesquisa têm demonstrado interesse em explorar os benefícios dos mecanismos dinâmicos do CORBA.

Em [26] é sugerido o uso de linguagens de scripting para conexão dos componentes criados em uma linguagem de programação qualquer. Através de scripts é fácil adaptar e parametrizar aplicações, além disso, os scripts permitem descobrir componentes dinamicamente. Linguagens de scripting são também chamadas de *glue languages* ou *linguagens de integração de sistemas*. Com base nesta sugestão, a idéia neste trabalho é usar uma linguagem de scripting como ferramenta para reconfiguração dinâmica de componentes. Para isto, a linguagem deve ter um sistema de tipos flexível o suficiente para permitir o acesso dinâmico aos componentes, o poder de expressão de uma linguagem de alto nível e um mapeamento para o modelo de componentes subjacente,no caso o modelo CORBA, porém na sua essência deve ser o mais independente possível deste modelo. Além disso,deve ser interpretada pois evita a recompilação e linkagem do programa depois da modificação, facilitando a reconfiguração das aplicações e evitando que a execução seja interrompida. Este aspecto permite prototipação rápida e testes imediatos. A tipagem dinâmica e falta de declarações, presentes em algumas linguagens interpretadas, flexibiliza a ligação em tempo de execução entre componentes não previstos anteriormente. TCL/Tk, CorbaScript e Lua são linguagens que possuem este perfil. Java, por exemplo, não satisfaz tais requisitos uma vez que um programa Java é compilado e produz um executável portável - ByteCode Java - que é carregado dinamicamente e interpretado pela máquina virtual Java.

A linguagem Lua [17] possui as características citadas acima e será usada como ferramenta para reconfiguração dinâmica de aplicações CORBA. Uma das vantagens em usar Lua está na facilidade de adicionar novos objetos em tempo de execução devido ao sistema de tipos dinâmicos característico da linguagem aliado ao fato de ser uma linguagem interpretada. Outra vantagem é a disponibilidade do LuaOrb [9], um binding entre a linguagem Lua e o CORBA baseado na Interface de Invocação Dinâmica (DII) que permite o acesso dinâmico através de Lua aos objetos disponíveis no barramento do CORBA. Além disso, uma outra ferramenta, o LuaDSI [20] explora a potencialidade da Interface de Esqueleto Dinâmico (DSI) do CORBA em conjunto com a linguagem Lua para possibilitar a instalação dinâmica de novos objetos nos servidores sem a necessidade de interrupção do servidor. Este aspecto facilita a construção dinâmica de objetos CORBA o que introduz flexibilidade para a reconfiguração da aplicação. Em termos de flexibilidade deve-se também ressaltar que a maioria das linguagens de configuração são declarativas [18, 27] impondo a restrição de declarar todas as instâncias de componentes que serão usadas e suas interconexões. Como Lua é procedural, pode-se configurar a aplicação através de estruturas de decisão que na execução selecionam um componente ou outro dependendo de condições estabelecidas pelo programador. Um outro mecanismo, a meta-interface [20], define um servidor genérico que permite a incorporação de objetos Lua dinamicamente.

Em suma, nosso propósito oferecer suporte via Lua para que aplicações sejam configuradas interativamente e modificadas em tempo de execução, acessando dinamicamente objetos CORBA através do LuaOrb. Isto irá permitir a construção de aplicações baseadas em CORBA através da descrição da configuração na linguagem Lua.

Este artigo está estruturado da seguinte forma. A seção 2 apresenta brevemente os conceitos subjacentes ao tema, que inclui noções principais da linguagem Lua, do LuaOrb, do LuaDSI, da biblioteca aLua e do mecanismo de meta-interface. Na seção 3 são descritos os mecanismos de reconfiguração dinâmica usados para integração de componentes CORBA. Na seção 4 são apresentadas as tarefas de reconfiguração possíveis. A seguir, na seção 5, a arquitetura do sistema de reconfiguração é discutida. Na seção 6 são classificadas as atividades de reconfiguração. Finalizando, na seção 7, concluimos ressaltando a importância do tema que estamos abordando e comentando a continuidade deste trabalho.

2 Conceitos Subjacentes

2.1 A linguagem Lua

Lua é uma linguagem de configuração de propósito geral que surgiu da necessidade de se usar uma linguagem de extensão para customizar aplicações industriais [17]. Não há o conceito de programa principal em Lua, pois por ser uma linguagem embutida, funciona associada a um programa host.

Lua é uma linguagem com sistema de tipos dinâmicos. 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 referencia 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 inteiros, strings, reais, tabelas e valores de funções. Esta flexibilidade é a base do mecanismo para descrição de objetos em Lua. Vários tipos de dados como arrays, sets, records, podem ser implementados através de tabelas. Records são implementados por tabelas usando os nomes dos campos como índices. Em Lua a representação **a.nome** é açúcar sintático para **a["nome"]**.

Tabelas são criadas com expressões especiais chamados construtores. O construtor mais simples é que cria uma nova tabela vazia. Outra forma de se construir tabelas consiste em fazer inicialização com atribuição de campos, onde os campos identificados são inicializados com os valores estabelecidos:

t = {primeiro = "maria", último = "joão"}

Também pode-se fazer a inicialização por indexação, onde escreve-se uma lista de expressões entre chaves e as expressões na lista são atribuídas a índices numéricos consecutivos começando por 1. Por exemplo:

t = {"Segunda", "Quarta", "Sexta"}

tem efeito semelhante à: t = { }, t[1] = "Segunda", t[2] = "Quarta", t[3] = "Sexta"

Como tabelas são objetos, não são valores, variáveis nunca contém tabelas, apenas referncias 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. A diferença entre métodos e outras funções é que o próprio objeto é passado implicitamente como o primeiro parâmetro de seus métodos. No corpo da função o objeto é acessado usando o parâmetro self.

Definições de métodos:

```
function objeto:método(parâmetros)
...
end
```

Chamada do método:

```
objeto:método(parâmetros)
```

que é açúcar sintático para:

```
objeto.método(objeto, parâmetros)
```

- 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. Parte da natureza relexiva é oferecida por **fallbacks** que permitem 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 abaixo **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 do LuaOrb.

  ```
  receiver:foo(params)
  ```

- A linguagem possui uma extensão para tratamento de eventos através da primitiva **send** que permite o envio de uma mensagem assíncrona para um processo. A chegada da mensagem dispara um evento no processo receptor. O comportamento default deste evento é executar a mensagem, assumindo que é um pedaço de código Lua. Este comportamento confere flexibilidade ao mecanismo de comunicação permitindo que o emissor da mensagem modifique variáveis globais, faça chamadas à funções ou defina novas funções no ambiente do receptor [32, 36].

  Exemplo:

send(Bid, 'lua_dofile('Teste')')

Quando o processo B recebe esta mensagem ele executa o comando **lua_dofile('Teste')**. O comando **lua_dofile** determina a execução do arquivo **Teste** que deve conter código Lua.

2.2 A Biblioteca aLua

Agent Lua (aLua) [36] é uma biblioteca que implementa um mecanismo de comunicação entre processos para a linguagem Lua usando um modelo de orientação a eventos. O mecanismo permite que mensagens contendo trechos de código Lua sejam enviadas entre processos.

Para enviar uma mensagem (um evento de comunicação), um processo utiliza a função:

send(nome_processo_destino, mensagem).

Para tratar eventos de comunicação a aplicação usa a função callback alua_Receive_cb. Esta função é chamada pelo sistema quando um evento de comunicação é recebido. Dois parâmetros são recebidos: o processo que gerou o evento e a mensagem. A aplicação que recebe a mensagem pode estabelecer o tratamento conferido à esta função. O tratamento default é considerar a string da mensagem como sendo um trecho de código Lua que será executado pelo interpretador Lua. O código recebido é executado no ambiente global de Lua, podendo portanto alterar variáveis e chamar funções já existentes no ambiente do processo. Através deste mecanismo uma aplicação pode usar outros processos para auxiliá-la em alguma tarefa sem tais processos conhecerem previamente o código a ser executado.

Este protocolo apresenta duas importantes propriedades: é não bloqueante, pois utiliza orientação a eventos na recepção, e cada evento recebido é considerado um bloco atômico, não podendo ter sua execução interrompida.

2.3 O Binding LuaOrb

O LuaOrb [9] é um binding entre a linguagem Lua e o CORBA que, da mesma maneira que todos os bindings entre linguagens de programação e o CORBA, define mapeamentos entre tipos Lua e IDL. Este mapeamento é baseado no sistema de tipos dinâmicos da linguagem Lua e na Interface de Invocação Dinâmica (DII) do CORBA. Como a DII permite que o usuário selecione o tipo do objeto e a operação durante a execução, o LuaOrb optou por usar este mecanismo na sua interface com o CORBA ao invés de usar stubs IDL. Estes aspectos permitem que objetos CORBA sejam acessados via Lua como qualquer outro objeto Lua, sem declaração prévia.

Para usar objetos CORBA, inicialmente tem-se de criar um proxy para este objeto chamando a função **identificador = createproxy(nome_da_interface, nome_da_instância)**. O segundo parâmetro desta função é opcional e, quando é especificado, a operação tem sucesso apenas se existe um objeto servidor com o mesmo nome deste parâmetro. Caso este parâmetro não seja especificado o ORB tenta encontrar algum servidor da interface desejada que tenha sido previamente ativado. Se nenhum servidor estiver ativado o ORB tenta ativar um deles. Caso o ORB não encontre um objeto para a interface desejada, a função createproxy retorna **nil**. Encontrando o objeto, um proxy é criado e é retornado um identificador para este objeto.

Depois da criação do proxy, os métodos da sua interface podem ser chamados. A implementação da createproxy é basicamente um mapeamento direto para a equivalente no ORB API.

A seguir mostraremos um exemplo do uso de objetos CORBA via Lua.

A figura 2 ilustra o uso do objeto definido na interface da Figura 1.

Como mostrado no exemplo acima, a chamada a um método de um objeto CORBA via LuaOrb é feita usando a sintaxe de chamada de métodos em Lua e os seguintes passos são realizados pelo LuaOrb:

- Interceptação da chamada que contém o nome do método, a referência proxy e os parâmetros da chamada do método

- Obtenção da assinatura do método no Repositório de Interfaces

- Mapeamento dos parâmetros Lua para IDL e construção da chamada dinâmica

```
struct dados_paciente{
    string nome;
    short pressao_max;
    short pressao_min;
    short temperatura;
    short pulso;
}
interface paciente{
   long adiciona_paciente(in dados_paciente dados);
   dados status(in long id);
   string monitora(in long id);
}
```

Figure 1: Interface IDL do Paciente

```
paciente = createproxy(''paciente'')
dados_pac = {nome=''Joao da Silva''; pressao_max=12; pressao_min=8; temperatura=37; pulso=90}
ident = paciente:adiciona_paciente(dados_pac);
```

Figure 2: Uso de Objetos CORBA via Lua

- Chamada ao método de acordo com a semântica de invocação
- Mapeamento dos resultados para Lua
- Envio dos resultados para a aplicação cliente

O LuaOrb oferece suporte para o modo de invocação síncrona adiada (com espera adiada). Para fazer esta chamada deve-se incluir o prefixo **deferred_** antes do nome do método. A chamada adiada retorna um handler a ser usado para recuperar o resultado da execução do método.

paciente:**deferred_**monitora(ident)

Como este tipo de chamada é bastante útil para programação orientada a eventos, o LuaOrb oferece a função **completion_event** que observa a finalização da execução do método especificado na chamada adiada e estabelece a nova função que será executada após o término. Esta função é declarada da seguinte forma:

completion_event(objeto:**deferred_**método(parâmetros), função)

A função determinada no segundo parâmetro da completion_event é invocada com o resultado do método da chamada adiada como parâmetro quando esta chamada retorna. Por exemplo, na chamada **completion_event(paciente:deferred_monitora(ident), print)**, quando o método **monitora** é finalizado, a função **print** é invocada tendo como parâmetro o string retornado pelo **monitora**.

2.4 LuaDSI

O LuaDSI [20] permite a extensão de objetos CORBA usando Lua. Usando o mecanismo de esqueleto dinâmico do CORBA, um objeto Lua pode ser instalado dinamicamente em um servidor passando a atuar como um objeto CORBA comum. Um cliente pode acessar este objeto através de stubs ou da DII.

Para definir um objeto dinamicamente é preciso descrever sua interface e escrever o objeto Lua que implementa as operações desta interface.

Dois aspectos foram introduzidos para tornar possível a instalação de novos objetos/serviços no servidor sem recompilá-lo ou interrompê-lo:

- Uma interface IDL responsável pela transmissão do código Lua para o servidor

- Do lado do servidor, um objeto DSI genérico que faz a conversão de cada operação proveniente do ORB para uma operação sobre um objeto Lua. Para isto o método invoke foi reimplementado para fazer esta conversão.

A instalação dinâmica do objeto Lua é feita através da interface IDL **ServerLua** que possui dois métodos:

- **Instance**: responsável pela criação de uma tabela Lua no servidor

- **InstallImplementation**: recebe como parâmetros a interface do objeto previamente criado por **Instance**, uma referência da descrição da operação **OperationDef** e o código Lua referente à implementação desta operação. Instala, na tabela Lua associada ao objeto CORBA em questão, a operação definida pela interface **OperationDef**. Isto é feito via a função **dostring** definida em Lua.

2.5 Meta-Interface

O mecanismo de meta-interface [20] tem a finalidade de permitir a incorporação de objetos e serviços Lua dinamicamente sem necessidade de alterar o repositório de interfaces.

A meta-interface é uma interface IDL que define um servidor genérico ("servidor de servidores") para acoplar vários objetos. O acesso a todos os objetos existentes no servidor é feito via meta-interface através de um conjunto de operações para chamada de funções, leitura e escrita de atributos.

Apesar de poder ser usada por qualquer cliente CORBA, para o cliente Lua o acesso a meta-interface é transparente e implementado através de fallbacks.

As facilidades principais oferecidas pela meta-interface são identificação dinâmica da interface dos objetos e instalação dinâmica de novos objetos.

A instalação dinâmica de novos objetos no servidor consiste em:

- criação de um objeto remoto realizada através da interface **Factory** que oferece o método **Object** para criação de um objeto e métodos para criar os tipos de dados que compẽm os objetos.

- associação do objeto remoto a um objeto servidor genérico através do método **Set** da interface **GenObject** que oferece mecanismos para acesso aos objetos do servidor.

Para identificação dinâmica da interface de objetos existentes no servidor usa-se o método **GetClassType** da interface **GenObject** que retorna uma estrutura contendo a descrição dos atributos e métodos do objeto.

3 Mecanismos para Interconexão dos Componentes

A linguagem de configuração Lua será o instrumento para definição da configuração da aplicação, ou seja, dos seus componentes integrantes e das conexões entre eles. Conexão é um caminho de comunicação entre o emissor e o receptor [19], portanto, descreveremos como este caminho de comunicação pode ser estabelecido via Lua.

Serão discutidas a seguir soluções para interconexão de componentes CORBA na configuração programada em Lua.

3.1 Interconexão via Lua como Mensageiro

Para ilustrar como atualmente pode-se especificar uma configuração em Lua e realizar a reconfiguração dinâmica, usaremos como exemplo o Sistema de Enfermaria, por se tratar de um exemplo clássico na literatura [18] sobre sistemas reconfiguravéis. Originalmente este sistema segue fortemente a estrutura estabelecida pelo paradigma da configuração e iremos modelá-lo a seguir como uma aplicação formada por componentes CORBA, o que nos permitirá analisar o poder expressivo da linguagem Lua e sua versatilidade em modelar aplicações de diferentes contextos.

Exemplo: O Sistema de Enfermaria consiste de uma central de enfermagem monitorando pacientes. Dados vitais dos pacientes são constantemente monitorados, e se algum dele atinge um valor

```
interface enfermaria{
   void alarme(in long id);
   string query(in long id);
}
```

Figure 3: Interface IDL da Enfermaria

```
void enfermaria::alarme(CORBA::Long id){
   cout <<''Alarme do Paciente:<<'':''<<id<<endl
}

char* enfermaria::query(CORBA::Long id){
   cout<<''Enviando solicitacao de status''<<endl;
   return(``OK'');
}

dados* paciente::status(CORBA::Long id){
   dados d;
   return(&d);
}

char* paciente::monitora(CORBA::Long id){
   dados d;
   char *resultado = ``Perigo'';
   if ((d.pressao_max <=12) || (d.pressao_min<=8) || (d.temperatura<=38) || (d.pulso<=120)){
       resultado = ``OK'';
   }
   cout<<''Paciente#''<<id<<''resultado=''<<resultado<<endl;
   return strdup(resultado);
}
```

Figure 4: Implementação dos Componentes na linguagem C++

crítico, é enviada uma mensagem de alarme à central de enfermagem que pode solicitar informação de status (dados vitais) do paciente. Este exemplo usa a interface IDL especificada na Figura 1, além da descrita Figura 3. A implementação dos componentes está ilustrada na figura 4.

Na solução da configuração ilustrada na Figura 5, usando a linguagem Lua para interconexão dos componentes é necessário que exista um agente configurador implementado a nível de programação da configuração que realiza o papel de um mensageiro, questionando os módulos envolvidos (no exemplo acima, **if (res = "OK") then)** se há mensagem a ser enviada para outro módulo, e enviando-a ao destinatário correspondente **(enf:alarme(numpac))**. Portanto, o mensageiro tem de conhecer o conteúdo da mensagem que um método está enviando para outro.

A solução para o mesmo problema usando linguagens de configuração tradicionais, que possuem construtores para ligação entre portas (*link* na linguagem CONIC [18] e *bind* na linguagem Durra [3]), estabelece a configuração a nível da linguagem, tornando transparente para a configuração as mensagens transmitidas via as interfaces. Por exemplo, a configuração do mesmo sistema usando o CONIC está na Figura 6.

A metáfora que se pode usar para analisar o comportamento dos sistemas acima é a de uma Central Telefônica que estabelece ligações entre objetos comunicantes. No caso do sistema CONIC a central não precisa tomar conhecimento da semântica da comunicação, ou seja, não precisa interpretar as mensagens que estão sendo enviadas depois que foi estabelecida uma ligação direta entre os métodos de cada objeto. Adotando a metáfora da central telefônica, o programador da configuração se limita a estabelecer os canais de comunicação, de maneira que quando determinado objeto envia uma mensagem, esta mensagem é recebida por outro objeto que esteja ligado no canal de comunicação. Este

```
enf=createproxy(``enfermaria'')
numpac=1
while numpac<5 do
   p=createproxy(``paciente'',''pac''..numpac)
   res=p:monitora(numpac)
   if (res~=''OK'') then
      enf:alarme(numpac) -- link entre o metodo monitora da interface paciente e o metodo
                            alarme da interface enfermaria
   else
      printf(``paciente'',numpac,''paciente OK'')
   end
   if enf:query(numpac)=''OK'' then
      s=p:status(numpac) -- link entre o metodo query da interface enfermeira e o metodo
                            status da interface paciente
   end
   numpac = numpac+1
end
```

Figure 5: Programa de Configuração em LUA - Exemplo 1

```
use enfermaria;paciente;
const numpac=4
create family k:[1..numpac]
pac[k]:paciente();
create enf:enfermaria;
link family k:[1..numpac]
     pac[k].monitora to enf.alarme[k];
     enf.query[k] to pac[k].status;
end.
```

Figure 6: Programa de Configuração no CONIC

```
interface enfermaria2{
   void alarme(in string nome, in string dado_critico)
}
```

Figure 7: Nova Interface IDL para a Enfermaria

é o caso mais fiel ao paradigma de orientação a objetos uma vez que não é necessário introduzir o objeto mensageiro.

Vale salientar que, apesar de não ser usual nos conceitos tradicionais de configuração, pode ser interessante a linguagem de configuração conhecer a semântica das mensagens trocadas entre os módulos para decidir qual serviço habilitar para atender à mensagem. Mensagens diferentes podem necessitar de serviços diferentes. Em linguagens de configuração tradicionais esta semântica tem de ser conhecida previamente para no momento da programação da configuração se estabelecer as ligações entre os componentes. Apesar da configuração poder ser modificada dinamicamente, esta estratégia passa a idéia de comportamento estático, por exemplo, se determinada mensagem dispara a instanciação de um novo componente, tem-se de estabelecer um canal só para a ligação entre eles. Quando um componente evolui é necessário remover o componente antigo e inserir o novo componente, o que, dependendo da linguagem, pode envolver vários comandos de reconfiguração. Com o LuaOrb e LuaDSI este procedimento pode ser mais automático que em linguagens de configuração tradicionais. Por exemplo, usando LuaDSI a instalação de novos serviços no componente é simples e não há necessidade de recompilação deste componente, o que constitui uma facilidade para se estender dinamicamente um objeto instalado sem afetar a configuração da aplicação. Em linguagens de configuração tradicionais não há infra-estrutura para provisão desta facilidade. Elas não incluem suporte para tratamento dos componentes, apenas da configuração, portanto a programação e extensão dos componentes fica a cargo de outra linguagem, o que implica em recompilação do componente e consequente atualização da configuração para troca do componente antigo por sua versão mais atualizada.

Outro aspecto importante é que o conhecimento da semântica da mensagem viabiliza a junção entre componentes com algum tipo de incompatibilidade nas suas interfaces. Em Lua esta compatibilização pode ser feita pela linguagem que age como elemento adaptador da interação entre componentes. Por exemplo, supondo que no Sistema de Enfermaria, seja utilizada uma nova interface IDL da enfermaria, conforme ilustra a figura 7. Neste caso, o método monitora do paciente não poderia ser ligado ao método alarme da enfermaria, uma vez que o método alarme exige parâmetros de entrada não disponibilizados pelo método monitora. A figura 8 ilustra a adaptação. Uma nova função (function get_dados_criticos (d)) foi implementada para retornar as variáveis críticas do paciente necessárias como parâmetro de entrada do método alarme. Em linguagens de configuração onde os componentes são ligados diretamente, não é possível ligar componentes originalmente incompatíveis pois elas não oferecem flexibilidade para adaptação destes componentes. O ambiente OLAN [2] e a linguagem Glish[28] realizam a compatibilização entre interfaces com assinaturas diferentes. No OLAN este serviço é realizado por Conectores e em Glish pelo interpretador Glish.

Usando Lua neste contexto dispomos de uma linguagem de programação completa, interpretada, com sintaxe simples, facilidades reflexivas e tratamento de eventos, aspectos que permitem estender as aplicações e componentes dinamicamente. As ferramentas LuaOrb e LuaDSI introduzem mais flexibilidade à linguagem tornando simples para o programador da aplicação o acesso a componentes CORBA e a introdução de novos serviços nos componentes. Desta forma, um ambiente para programação de configuração/reconfiguração constituído por esta linguagem e suas ferramentas confere grande liberdade ao programador, mostrando-se um poderoso ambiente para programação.

3.2 Interconexão via uma Superinterface

Uma opção para eliminar a necessidade do mensageiro em programas LUA seria modelar a interface do componente adicionando nos parÌ‚metros um objeto que chamaremos de *Objeto Genérico*. Este objeto genérico é definido através de uma interface que contém a descrição dos métodos que o componente possivelmente irá usar. Denominaremos de *Superinterface* a interface do componente que incorpora a referência para o Objeto Genérico.

```
enf = createproxy(``enfermaria2'');
numpac=1
while numpac<5 do
   p=createproxy(``paciente'',''pac''..numpac)
   res=p:monitora(numpac)
   if (res=''OK'') then
      dados_pac=p:status(numpac)   -- compatibilizacao entre pac e enf
      dados_criticos=get_dados_criticos(dados_pac) -- compatibilizacao entre pac e enf
      enf:alarme(dados_pac.nome, dados_criticos) -- link entre o metodo monitora e alarme
   end

   function get_dados_criticos(d)
      criticos=''''
      if (d.pressao_max>12) then
         criticos=''pressao_max=''..(d.pressao_max)
      end
       if (d.pressao_min>8) then
         criticos=criticos..''pressao_min=''..(d.pressao_min)
      end
      if (d.temperatura>38) then
         criticos=criticos..''temp=''..(d.temperatura)
      end
      if (d.pulso>120) then
         criticos=criticos..''pulso=''..(d.pulso)
      end
      return criticos
end
```

Figure 8: Compatibilização entre Componentes via Lua

```
interface enfermaria{
   void alarme(in long id);
   void query(in long id, in obj_gen_enf);
}

interface obj_gen_enf{
   dados status(in long id);
}

interface paciente{
   long adiciona_paciente(in dados_paciente dados);
   dados status(in long id);
   void monitora(in long id, in obj_gen_pac);
}

interface obj_gen_pac{
   void alarme(in long id);
}
```

Figure 9: Interface IDL dos Componentes CORBA

```
enf=createproxy(``enfermaria'')
numpac=1
while numpac<5 do
   pac=createproxy(``paciente'',''pac''..numpac)
   pac:monitora(numpac,enf)
   enf:query(numpac,pac)
   numpac=numpac+1
end
```

Figure 10: Programa de Configuração em LUA - Exemplo 2

O objeto genérico representa a especificação dos métodos que o componente poderá usar. Como todas as interfaces CORBA, a interface que especifica o objeto genérico é puramente declarativa, não descrevendo nenhum detalhe de implementação. A identificação da(s) implementação(ões) apropriada(s) é realizada na configuração da aplicação que pode escolher qualquer objeto que satisfaça a interface descrita. Nesta tarefa devem ser aproveitadas as facilidades reflexivas de Lua, seu mecanismo de tipagem dinâmica e o modelo de compatibilidade estrutural entre tipos empregado no LuaOrb.

Assim, usando Lua na configuração da aplicação pode-se fazer a ligação entre o componente e o(s) objeto(s) que irão satisfazer os parâmetros descritos na interface do objeto genérico. A interface IDL correspondente está na Figura 9. A aplicação seria então estruturada conforme ilustra a Figura 10.

3.3 Interconexão via Eventos Lua

Pode-se explorar o suporte a eventos utilizando as facilidades oferecidas por Lua e pelo LuaOrb para tratamento de eventos.

Um modelo de gerência de eventos baseado na invocação síncrona adiada disponível em Lua em combinação com a função **completion_event** é umas das maneiras de resolver o problema da interconexão entre os componentes. Na Figura 11 é apresentado o exemplo do sistema de enfermaria programado usando estes aspectos.

Neste exemplo, quando encerrar a execução do método **monitora**, especificado na chamada adiada, a função **trata_evento_pac** é invocada. Esta função chama o método **alarme** do objeto **enf** e novamente volta a esperar a finalização do método **monitora** que recursivamente invoca a função **trata_evento_pac**. Similarmente, ao finalizar a execução do método **query** do objeto **enf**, a função

```
enf=createproxy(``enfermaria'')
numpac=1
while numpac<5 do
   p=createproxy(``paciente'',''pac''..numpac)
   completion_event(p:deferred_monitora(numpac),trata_evento_pac)
   completion_event(enf:deferred_query(numpac),trata_evento_enf)
   numpac=numpac+1
end

function trata_evento_pac(id)
   enf:alarme(id)
   completion_event(p:deferred_monitora(id),trata_evento_pac)
end

function trata_evento_enf(id)
   p:status(id)
   completion_event(enf:deferred_query(id),trata_evento_enf)
end
```

Figure 11: Programa de Configuração em LUA - Exemplo 3

```
enf = createproxy(``enfermaria'')
numpac = 1
while numpac<5 do
   p=createproxy(``paciente'',''pac''..numpac)
   completion_event(p:deferred_monitora(numpac),enf:alarme)
   completion_event(enf:deferred_query(numpac),p:status)
   numpac=numpac+1
end
```

Figure 12: Programa de Configuração em LUA - Exemplo 4

trata_evento_enf é chamada. Esta invoca o método **status** do objeto **pac** e recursivamente volta a tratar o término da execução do método **query**.

Uma outra solução consiste em passar diretamente para a função **completion_event** um método CORBA em vez de um método Lua. Então o programa exibido na Figura 11 pode ser reescrito como mostra a Figura 12.

4 Tarefas de Reconfiguração

Nos principais sistemas que permitem reconfiguração dinâmica das aplicações, as possíveis ações de reconfiguração são remover e inserir componentes e suas instâncias, criar e destruir instâncias de componentes, remover e inserir links entre interfaces de componentes e alocar componentes a processadores [3, 8, 15, 16, 19, 27].

As modificações em sistemas podem ocorrer no nível dos componentes (mudança na implementação, adição de novas funções) e no nível de configuração (adição ou remoção de componentes ou conexões) [37]. Em geral, as linguagens de configuração endereçam apenas modificações na configuração. Em contraste, Lua oferece possibilidades de tratar modificações nos dois níveis.

Em relação à origem da ação de reconfiguração, ela pode partir da decisão do próprio programa (quando ele possui algum grau de reflexividade) ou pode partir do usuário [12]. O primeiro caso é chamado de mudança planejada pois a reconfiguração iniciada pelo programa requer que seu programador tenha previamente especificado a mudança. A habilidade de mudar conexões entre componentes em tempo de execução é um requisito fundamental da mudança estrutural, independente de sua origem.

```
CORBA::Object_ptr objLog =
          serverLua -> Instance(interfaceDefLog);
serverLua -> InstallImplementation(objLog, operationDefhistorico,
         ``function (number)''
         ``   dados = tabpaciente[number]''
          ``  return dados''
         ``end'');
```

Figure 13: Criação Dinâmica de Objetos

Como estamos usando componentes CORBA e uma linguagem de configuração com tipagem dinâmica, o problema de reconfiguração é, em alguns aspectos, diferente do abordado em outras linguagens de configuração e baseados no paradigma da configuração, onde tipicamente as interações entre os módulos são previsíveis e as interfaces especificadas em função da ligação entre portas de entrada e portas de saída. A linguagem de configuração Lua não se baseia em um modelo com operações de configuração explícitas no estilo Use, Inst, Bind das linguagens de configuração tradicionais [3, 18, 27]. Em Lua não há necessidade de declarar antecipadamente que vai-se usar um tipo de componente, pode-se diretamente usar ou criar a instância do componente (através do createproxy, como ilustrado nos exemplos anteriores), portanto, uma operação estilo *Use*, disponível em grande parte das linguagens de configuração, não é necessária em Lua. Discutiremos a seguir os pontos importantes em relação à reconfiguração no contexto deste trabalho, considerando as cinco tarefas de reconfiguração abordadas convencionalmente: adição de componentes, remoção de componentes, atualização de componentes, inserção/remoção de ligações entre componentes e mapeamento de componentes a processadores.

A linguagem permite que se crie código Lua e incorpore este código na configuração da aplicação. Pode-se posteriormente remover o código Lua incluído. Estas ações podem ser feitas meramente editando o script de configuração da aplicação que irá reconstruir a aplicação a partir da especificação editada.

4.1 Inserção de Componentes na Configuração

A inserção de componentes na configuração de uma aplicação deve abordar dois aspectos: incorporação de componentes já existentes e incorporação de componentes não existentes. Esta última tarefa envolve a criação do componente e posteriormente a sua inserção na configuração.

Para criação dinâmica de componentes pode-se usar os recursos oferecidos por LuaDSI. Após publicar a interface CORBA no repositório de interfaces, obtem-se as referências InterfaceDef e OperationDef e em seguida instancia-se o objeto Lua no servidor. O código C++ da instanciação está na figura 13.

Para inserção dinâmica de componentes pode-se lançar mão das facilidades de comunicação entre processos conferidas pela biblioteca **aLua** [36] ou explorar o comando condicional oferecido por Lua para programar quando a reconfiguração deve ocorrer, selecionando dinamicamente os componentes a serem inseridos na configuração.

Usando aLua, a reconfiguração é feita enviando-se para o script que corresponde à configuração da aplicação uma mensagem contendo um pedaço de código a ser executado. Este código contém as ações de reconfiguração do sistema. Como exemplo, descreveremos abaixo uma reconfiguração no Sistema de Enfermaria cujos componentes estão descritos na figura 9 e configuração na figura 10. A reconfiguração consiste em adicionar um componente de registro periódico de dados da monitoração dos pacientes, com a finalidade de guardar em arquivo o histórico dos pacientes. Este componente registra os dados fornecidos pelo método **status** da interface Paciente e oferece um método para consulta do histórico de determinado paciente. A interface IDL está descrita na figura 14. A figura 15 mostra o script de reconfiguração da aplicação gravado no arquivo "reconf.lua". Supondo que o processo A corresponda ao Sistema de Enfermaria da figura 10 em execucao, o código Lua da figura 16 envia a mensagem de reconfiguração ao processo A. Note que a reconfiguração descrita na figura 15 usa as variáveis numpac e pac definidas no ambiente global onde será executada, ou seja, definida no ambiente do processo A.

```
interface log{
   void get_dados(in long id, in obj_gen_enf); /* grava periodicamente os dados do paciente */
   seqdados historico(in long id); /* retorna o historico do paciente */
}
```

Figure 14: Interface IDL do componente Log

```
l=createproxy(``log'')
numpac=1
while numpac<5 do
   l:get_dados(numpac, pac)
   numpac=numpac+1
end
```

Figure 15: Script de Reconfiguração gravado no arquivo "reconf.lua"- Extensão do Sistema de Enfermaria

O comando condicional disponível em Lua possibilita que a reconfiguração do sistema faça parte do próprio script de configuração. A figura 17 exemplifica esta facilidade. Neste exemplo, o método query do componente enfermaria pode solicitar o status atual do paciente (retorna a mensagem "status") ou solicitar todo o histórico dele (retorna a mensagem "historico"). Dependendo da mensagem retornada será introduzido na configuração o componente log ou usado o componente paciente. Este exemplo ilustra também a importância de se conhecer a mensagem emitida por um módulo para decidir qual serviço utilizar.

4.2 Remoção de Componentes

A remoção de componentes deve considerar dois níveis: remoção dos proxies Lua para objetos CORBA e remoção dos próprios objetos CORBA.

A remoção dos proxies é feita automaticamente pelo mecanismo de coleta de lixo de Lua quando ele não é mais usado. Isto é similar a abordagem de Regis [21] onde instâncias de componentes são removidas via o mecanismo de coleta de lixo quando a referência para ele não é mais usada.

Para a remoção de objetos CORBA seria necessário um suporte que a infra-estrutura CORBA não oferece. Este suporte deveria manter a informação sobre a utilização do objeto e um mecanismo de coleta de lixo para remover os objetos que não estão sendo utilizados.

4.3 Inserção/Remoção de ligações entre componentes

Quanto à inserção/remoção de ligações entre componentes, como discutido na seção anterior, as interconexões entre os módulos são realizadas no nível de configuração mas não usamos construtores como link ou bind, portanto, na reconfiguração não necessitamos de construtores inversos como unlink ou unbind. Independente da estratégia usada para interconexão, o sistema deve permitir a mudança dinâmica das conexões. Nos casos de interconexão que discutimos neste trabalho a mudança consiste em retirar o componente origem e/ou destino da interconexão e, quando for o caso, introduzir o(s) novo(s) componentes e fazer a ligação. Uma modificação no Sistema de Enfermaria será exemplificada

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

Figure 16: Código Lua com a mensagem de Reconfiguração

```
enf=createproxy(``enfermaria'')
numpac=1
while numpac<5 do
   p=createproxy(``paciente'',''pac''..numpac)
   res=p:monitora(numpac)
   if (res~=''OK'') then
      enf:alarme(numpac) -- link entre o metodo monitora da interface paciente e o metodo
                            alarme da interface enfermaria
   else
      printf(``paciente'',numpac,''paciente OK'')
   end
   if enf:query(numpac)=''status'' then
      s=p:status(numpac) -- link entre o metodo query da interface enfermeira e o metodo
                            status da interface paciente
   else
     l=createproxy(``log'')
     s=l:historico(numpac) -- link entre o metodo query da interface enfermeira e o metodo
                              historico da interface log
   end
   numpac = numpac+1
end
```

Figure 17: Reconfiguração através de Condicionais - Inserção de Componente

trocando o módulo enfermaria por novaenf (Figura 18) para permitir que a enfermaria tenha acesso ao histórico do paciente fornecido pelo módulo log da figura 14.

```
interface novaenf{
   void alarme(in long id);
   string query(in long id);
   void historico(in long id);
}
```

Figure 18: Interface IDL do Componente NovaEnfermaria

Considerando a configuração descrita na figura 12, a reconfiguração de forma a considerar o componente novaenf pode ser feita editando o script de configuração ou enviando ao processo que corresponde a execução da configuração (chamamos de processoA), uma mensagem com os comandos de reconfiguração conforme mostra a figura 19. Esta figura ilustra duas tarefas de reconfiguração: troca do componente enfermaria por novaenf (a variável enf aponta para novaenf e não mais para enfermaria) e inserção da ligação entre o método histórico do componente enf e o método histórico do componente log.

4.4 Atualização de Componentes

Tradicionalmente a atualização de um componente determina na configuração a troca do componente antigo pelo novo componente. Usando Lua o tratamento pode ser diferente. Podemos considerar a atualização por duas vertentes:

- Atualização da Implementação de Métodos do Componente

  Em geral as linguagens de configuração tradicionais não endereçam este aspecto uma vez que a abstração divulgada pelo paradigma de orientação a objetos garante que a interface permanecendo a mesma, a implementação pode ser modificada sem haver necessidade de alteração na configuração. A abstração também é válida na configuração Lua que também oferece a flexibilidade de modificar a implementação dos métodos de componentes Lua. Esta alteração pode

```
msg=[[enf=createproxy{''novaenf'')
l=createproxy(''log'')
numpac=1
while numpac<5 do
   completion_event(enf:deferred_historico(numpac),l:historico)
   numpac=numpac+1
end]]
send('processoA',msg)
```

Figure 19: Reconfiguração do Sistema de Enfermaria - Troca de Componente e Inserção de Ligação

```
CORBA::Object_ptr objLog =
          serverLua -> Instance(interfaceDefLog);
serverLua -> InstallImplementation(objLog, operationDefhistorico,
        ''function (number)''
        ''   dados = tabpaciente[number]''
        ''   dados_pressao = dados[pressao]''
         ''  return dados_pressao''
        ''end'');
```

Figure 20: Atualização da Implementação de um Método do Componente

ser feita usando LuaDSI conservando inalterada a configuração da aplicação. Para isto, usa-se o método **InstallImplementation** da interface **serverLua** que irá instalar a nova implementação substituindo a antiga. O exemplo 20 ilustra a instalação de uma nova implementação para o método histórico da interface log que retorna apenas os registros de pressao de determinado paciente.

- Atualização da Interface do Componente (adição ou remoção de métodos ou dos parâmetros de métodos)

Um ponto importante a ser considerado na atualização de componentes se refere a guardar diferentes versões do mesmo componente e definir mecanismos que transparentemente para a configuração selecione uma das versões quando o componente é utilizado. Isto seria um mecanismo de reconfiguração automática. Pode-se disponibilizar este mecanismo de duas maneiras:

- Usando a Meta-Interface para englobar as diferentes versões de uma mesma interface. Exemplificando este aspecto,suponha que a configuração da aplicação inclui uma chamada a um objeto A. Existe uma meta-interface que engloba diferentes versões deste objeto: A1, A2, A3, A4. Ao ser chamado o objeto A, a meta-interface recebe a chamada, seleciona a versão mais apropriada de acordo com o filtro estabelecido para tal seleção (se é a última versão ou a versão mais antiga, etc). Para a configuração fica transparente qual objeto está sendo utilizado. A limitação desta abordagem é permitir apenas acesso a objetos Lua.

- Usando a informação sobre versões oferecida pelo Repositório de Implementações e definindo um conector cuja função é selecionar deste repositório versões diferentes do mesmo objeto. Este tratamento é diferente do oferecido pela meta-interface uma vez que o conector faz uma busca no repositório de interfaces enquanto que a meta-interface já incorpora as diferentes versões da mesma interface.

4.5 Mapeamento de Componentes a Processadores

A filosofia do CORBA é tornar transparente para o cliente detalhes como a máquina onde determinado servidor está sendo executado. Portanto não se tem acesso a esta informação a nível de configuração.

Uma possibilidade de controlar este aspecto é modelar um serviço CORBA onde o cliente determine em qual máquina um servidor vai executar e disponibilizar esta opção à nível de configuração.

5 Arquitetura do Sistema de Reconfiguração Dinâmica

A linguagem de configuração, no seu papel de facilitar o processo de descrição arquitetural da aplicação, pode oferecer suporte à interconexão dinâmica entre os componentes em dois níveis, que chamaremos de níveis de reconfiguração:

- **nível do usuário**: neste nível, o usuário é responsável por estabelecer as conexões necessárias entre os componentes, o que pode ser feito durante a etapa de configuração inicial do sistema e também durante a execução do mesmo. A linguagem Lua e o LuaOrb oferecem ferramentas para dar suporte ao usuário neste nível: o console Lua e o browser do Repositório de Interfaces. O console Lua disponibiliza uma interface textual interativa que pode ser usada para se especificar reconfigurações dinamicamente. Usando esta interface o usuário pode emitir comandos Lua ou determinar a execução de um arquivo contendo um script de reconfiguração. O browser do Repositório de Interfaces, por sua vez, exibe todos os componentes que a aplicação pode acessar via o LuaOrb, permitindo que o usuário encontre o componente que necessita e, usando o console Lua, o insira na sua aplicação. Além disso, a ferramenta LuaDSI permite que o usuário reconfigure o sistema instalando dinamicamente novos serviços nos componentes.

  O próprio script inicial de configuração escrito pelo usuário contém estruturas de decisão que na execução selecionam um componente ou outro dependendo de condições estabelecidas. Assim, é realizada reconfiguração dinâmica do sistema na própria execucão deste script.

- **nível do sistema**: neste nível de reconfiguração é realizada a automatização do processo de encontrar componentes que satisfazem as solicitações e de interconectá-los ao sistema. Neste caso a tarefa de reconfiguração é transparente para o usuário e portanto é necessário a intervenção de um *Gerenciador de Configuração* que é o elemento responsável por estas tarefas.

  Siegel [34] alerta que o serviço de Trader deverá ser o meio para se evoluir de aplicações estáticas para as aplicações configuradas dinamicamente. Seguindo esta previsão, uma forma de se encontrar o componente dinamicamente é usando o serviço de Trader. Dada a configuração inicial de uma aplicação, não é possível saber quais os componentes serão adicionados ao sistema para atender as necessidades de um componente já presente. A idéia é que durante a execução, quando um componente necessitar de um serviço para completar sua tarefa, o Gerenciador de Configuração é ativado. Este gerenciador dispara uma consulta ao serviço de Trader passando como parâmetro o serviço solicitado pelo componente. O Trader envia de volta a lista de componentes que fornece tal serviço e o gerenciador pode neste momento introduzir o novo componente (fornecedor do serviço procurado) na configuração, que consiste em introduzir comandos de configuração necessários para inserção do componente auxiliar e sua ligação com o componente que o solicita. Desta forma, a configuração vai sendo formada dinamicamente e o gerenciador é responsável pela instanciação das conexões não previsíveis.

Em suma, pode-se seguir dois caminhos para resolver o problema da interconexão dos objetos, um deles com intervenção do usuário que se responsabiliza por encontrar os novos componentes e inseri- los na configuração, o outro, controlado pelo sistema, onde o gerenciador de configuração usa o serviço de trader e o repositório de interfaces para encontrar componentes dinamicamente e inseri-los automaticamente na configuração. Eventualmente o gerenciador pode solicitar a ajuda do usuário para realizar esta tarefa.

6 Classificação de Atividades de Configuração

No modelo de reconfiguração com componentes CORBA em geral a tarefa de reconfiguração é mais intensa que nos demais sistemas onde as interconexões podem ser previamente definidas e a reconfiguração só ocorre quando modificações são necessárias. Neste sistema o mecanismo de reconfiguração vai ser disparado toda vez que um componente precisar de um serviço que não está disponível na configuração, e o usuário poderá optar pela intervenção do Gerenciador de Configuração ou por se responsabilizar por esta tarefa.

Bidan et al. [6] classificam a atividade de reconfiguração dinâmica em duas categorias:

- *reconfiguração programada* que é parte do design do sistema e é escalonada para acontecer quando certas condições forem satisfeitas durante a execução da aplicação.
- *reconfiguração evolutiva* que se refere às modificações na estrutura causadas por eventos independentes da especificação da aplicação. Esta reconfiguração é realizada pelo usuário visando uma evolução da sua aplicação através da modificação na estrutura existente, seja através da inserção de novos elementos ou remoção de elementos presentes. Por exemplo, a figura 19 mostra uma reconfiguração evolutiva determinada pelo usuário que deseja substituir o componente enf pelo componente novaenf que é uma versão mais atualizada.

Segundo esta classificação, na reconfiguração programada um conjunto fixo de eventos possíveis são previstos e para cada um deles é programada uma resposta. Considerando a aplicação formada por componentes CORBA, a reconfiguração programada pode ser estendida de forma a incluir especificamente o tratamento do evento que corresponde a falta de um componente que o sistema precisa para continuar. Neste caso podemos dizer que a configuração vai sendo adaptada automaticamente à medida que são necessários novos serviços para completar a tarefa de componentes presentes na configuração, e podemos denominar este aspecto de *reconfiguração programada adaptativa*.

Pode-se relacionar estas atividades de configuração com os níveis de reconfiguração comentados na seção 5. A Reconfiguração Programada é realizada a nível de usuário, já que é o usuário que determina os eventos de reconfiguração e as respostas à cada um deles. A Reconfiguração Evolutiva é também realizada pelo usuário que modifica a configuração. A Reconfiguração Programada Adaptativa pode corresponder a reconfiguração à nível do usuário ou do sistema, dependendo se é o usuário quem determina o componente a ser inserido na configuração, ou se é o sistema através de mecanismos automáticos como serviço de Trader.

7 Conclusão

Este trabalho apresentou mecanismos para reconfiguração dinâmica de aplicações CORBA concentrando-se no aspecto de interconexão entre os componentes através da linguagem interpretada Lua.

A maioria das propostas para reconfiguração dinâmica de aplicações não se adaptam às aplicações com componentes CORBA, uma vez que estes componentes não declaram os serviços que precisam usar. Além disso, a complexidade do mecanismo de acesso dinâmico do CORBA aos objetos é considerado um empecilho para os trabalhos nesta área. Este último problema é endereçado pelo LuaOrb que permite que chamadas aos métodos em objetos Corba sejam idênticas às chamadas a um objeto Lua.

Consideramos o enfoque na interconexão entre objetos via Lua um tema bastante interessante e oportuno dada a importância que atualmente vem sendo dedicada a interoperação entre objetos e linguagens e a ligação entre eles. Esta atenção não tem se limitado apenas a objetos CORBA. O trabalho descrito em [14] descreve como algumas linguagens foram modificadas para permitir que aplicações escritas nelas possam acessar objetos COM (Component Object Model). Neste trabalho está descrito como três diferentes linguagens implementam suporte para o COM. A abordagem que focamos neste trabalho também poderia ser aplicada a outros sistemas de componentes considerando as particularidades de cada um deles.

A originalidade deste trabalho está na estruturação das aplicações CORBA e no respectivo tratamento de reconfiguração, ambos aspectos apoiados por uma linguagem interpretada com um sistema dinâmico de tipos, a linguagem Lua, por um sistema acoplado à linguagem, o LuaOrb, que aproveita as funcionalidades dinâmicas oferecidas pelo CORBA e por uma ferramenta, LuaDSI, que permite a instalação dinâmica de novos serviços nos componentes. Além disso, a utilização da biblioteca aLua para envio de mensagens com códigos Lua, e o emprego do mecanismo de meta-interface para incorporação de objetos Lua dinamicamente sem alterar o repositório de interfaces, ampliam a gama de mecanismos de reconfiguração e interconexão entre componentes disponíveis via Lua.

Em [24] é mencionado que o estilo de invocação implícita baseada em eventos é mais propício a reconfiguração dinâmica uma vez que os componentes não são diretamente ligados à outros e um componente não conhece os componentes invocados implicitamente. Estas características nos fazem identificar que este estilo é bastante adequado a reconfiguração com componentes CORBA, e portanto iremos explora-lo através do suporte oferecido pelo serviço de eventos do CORBA. Também deveremos explorar a idéia de tratamento de eventos via objetos conectores cujo comportamento pode ser customizado dinamicamente. Possivelmente será feita uma integração de objetos conectores no mecanismo de configuração Lua. Desta maneira estaremos disponibilizando comunicação síncrona ou assíncrona não apenas entre objetos CORBA mas entre quaisquer outros componentes da configuração Lua.

Pode-se alegar que a linguagem Lua, no seu papel de linguagem de configuração que pode ser usada não apenas para interconexão de componentes CORBA mas para interconexão de quaisquer componentes, deve oferecer um stream como instrumento para ligar as interfaces dos componentes, introduzindo na sua sintaxe um construtor semelhante ao bind ou link. Este construtor seria uma forma de definir uma conexão implícita e, conforme ressaltado em [7], os módulos teriam de ser programados em função desta característica. O módulo referenciado do lado esquerdo do construtor bind (ou link) deve gerar uma saída que é observada no nível de de configuração, onde a linguagem irá efetuar a conexão com o método referenciado do lado direito do construtor. Foge ao escopo deste trabalho tratar deste aspecto já que não seria aplicado no contexto de aplicações CORBA.

References

[1] Adler,R. Emerging Standards for Component Software. IEEE Computer. March 1995

[2] Balter, R.; Bellissard, L; Boyer, F.; Riveill, M. & Vion-Dury, J. Architecturing and Configuring Distributed Application with Olan. http://sirac.imag.fr/PUB/98/98-middleware-olan-PUB.ps.gz

[3] Barbacci, M et al. Durra: a structure description language for developing distributed applications Software Engineering Journal, March 1993.

[4] Bellisard, L; Riveill, M.; Constructions des applications réparties. Ecole Placement Dynamique et Répartition de Charge - Juillet 1996.

[5] Bernstein, P. Middleware. Communications of the ACM, Vol. 39, N* 2, February 1996, pp 86-98.

[6] Bidan, C.; Issarny, V. ; Saridakis, T. & Zarras, A. A Dynamic Reconfiguration Service for CORBA. http://doubletap.cs.umd.edu/ purtilo/Papers/papers.html

[7] Bishop, J. & Faria, R. Connectors in Configuration Programming Languages: are they necessary? Third International Conference on Configurable Distributed Systems. May 6-8, 1996 - Annapolis, Maryland. pp 11- 18

[8] Bloom, T. & Day, M. Reconfiguration and module replacement in Argus: theory and practice. Software Engineering Jounal. March 1993.

[9] Cerqueira, R. Rodriguez, N. & Ierusalimschy, R. A Dynamic Approach for Composing CORBA Objects. Monografias em Ciência da Computação, PUC-Rio, Rio de Janeiro, Brazil, 1997. MCC43/97.

[10] Chambers, C. Towards Reusable, Extensible Components. ACM Computing Surveys, 28, December 1996. Available in http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a192-chambers/

[11] Crane,J.; Magee,J. & Pryce,N. Design Patterns for Binding in Distributed Systems. 1995

[12] Crane, J. Dynamic Binding for Distributed Systems. http://outoften.doc.ic.ac.uk/ jsc/phd/

[13] Merle, P. CorbaScript - CorbaWeb : Propositions pour l'acces à des objets et services distribués. Janvier 97. Université des Sciences et Technologies de Lille.

[14] Gray, D. et al. Modern Languages and Microsoft's Component Object Model. Communications of the ACM, Vol. 41, N. 5, pp 55-65, May 98

[15] Hofmeister, C. Dynamic Reconfiguration of Distributed Applications. http://www.cs.umd.edu/ purtilo

[16] Hofmeister, C.; White, El & Purtilo, J. Surgeon: a packager for dynamically reconfigurable distributed applications. Software Engineering Journal, March 95, pp 95-101

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

[18] Kramer, J. & Magee, J. Dynamic Configuration for Distributed Systems. IEEE Transactions on Software Engineering, Vol. SE-11, N. 4, April 1985.

[19] Kramer, J. & Magee, J. The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering, Vol. 16, N. 11, November 1990.

[20] Martins, M.A.C. Extensão Dinâmica de Agentes CORBA. Dissertação de Mestrado, Depto. de Informática - PUC-Rio. Setembro de 1998

[21] Magee, J.; Dulay, N. & Kramer, J. Regis: A Constructive Developement Environment for Distributed Programs. http://www.dse.doc.ic.ac.uk/

[22] Medvidovic, N. & Taylor, R. A Framework for Classifying and Comparing Architecture Description Languages.Available at http://www.ics.uci.edu/pub/arch/sw-and-pubs.shtml

[23] Nierstrasz, O.; Schneider, J. & Lumpe, M. Formalizing Composable Software Systems - A Research Agenda. In Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS'96, Paris, France, March 1996

[24] Oreizy, P. & Taylor, R. On the Role of Software Architectures in Runtime System Reconfiguration. Proceedings of the Fourth International Conference on Configurable Distributed Systems. May 4-6,1998, Annapolis, Maryland, pp 61-70

[25] Oreizy, P. Issues in the Runtime Modification of Software Architectures. http://www.ics.uci.edu/ peymano/

[26] Ousterhout, J. Scripting: Higher Level Programming for the 21st Century. http://www.scriptics.com/people/john.ousterhout

[27] Paula, V. C. C. de; Implementação de Linguagens de Configuração para Sistemas Distribuídos. Dissertação de Mestrado. Universidade Federal de Pernambuco. Janeiro 1994

[28] Paxson, V. Glish: A Software Bus for High-Level Control. http://www.cv.nrao.edu/pub/glish/

[29] Plásil, F.; Bálek, D. & Janecek, R. SOFA/DCUP: Archicture for Component Trading and Dynamic Updating. Proceedings of the Fourth International Conference on Configurable Distributed Systems. May 4-6, 1998, Annapolis, Maryland.

[30] Purtilo, H. The Polylith Software Bus. ACM Transations on Programming Languages and Systems. Vol. 18, N.6, November 1996

[31] Rodriguez, N.; Ierusalimschy, R. & Cerqueira, R. Dynamic Configuration with CORBA components. Proceedings of the Fourth International Conference on Configurable Distributed Systems. May 4-6,1998, Annapolis, Maryland, pp 27-34

[32] Rodriguez, N.; Ururahy, C; Ierusalimschy, R. & Cerqueira, R. The use of interpreted languages for implementing parallel algorithms on distributed systems. Monografias em Ciência da Computação, PUC-Rio, Rio de Janeiro, Brazil, 1996. MCC13/96.

[33] Shaw, M. & Garlan, D. Software Architecture - Perspectives on an Emerging Discipline. Ed. Prentice-Hall Inc., 1996.

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

[35] Siegel, J. OMG Overview: CORBA and the OMA in Enterprise Computing. Communications of the ACM, Outubro 98.

[36] Ururahy, C. Agentes Lua: Um Mecanismo de Comunicação em Lua. Dissertação de Mestrado. Departamento de Informática PUC/RJ. Setembro 1998.

[37] Wermelinger, M. A Hierarchic Architecture Model for Dynamic Reconfiguration. Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems, May 97, Boston, Massachusets, USA, pp 243-254.