LOrbEvent: Uma Biblioteca para Viabilizar o Uso do Serviço de Eventos de CORBA

Thaís Batista
thais@ufrnet.br

Nélio Cacho
neliocacho@ig.com.br

Gabriel Galvão
gabriel_galvao@yahoo.com

Departamento de Informática e Matemática Aplicada - DIMAp
Universidade Federal do Rio Grande do Norte - UFRN
Campus Universitário – Lagoa Nova – 59.072-970 - Natal – RN

Resumo

Este trabalho tem como objetivo apresentar LOrbEvent, uma biblioteca que abstrai as dificuldades de se usar o serviço de eventos de CORBA e torna viável que programadores de aplicações usem tal serviço sem precisar conhecer os detalhes da especificação CORBA nem emitir diversos comandos para habilitar o uso do serviço. Com o suporte de LOrbEvent, as aplicações podem ser estruturadas seguindo o modelo de comunicação orientada a eventos sem necessidade de incluir extensas chamadas ao serviço de eventos de CORBA. Além disso, LOrbEvent adiciona facilidades que originalmente o serviço de eventos de CORBA não fornece: envio e recebimento de mensagens considerando o assunto de interesse.

Abstract

The main purpose of this work is to present LOrbEvent, a library that abstracts the difficulties of using the CORBA event service and makes it feasible for the programmers to use such service without knowing the details about the CORBA specification and without having to use many commands to initialize the service. With the LorbEvent support, applications can be structured following the event based communication model without including long invocations to the CORBA event service. Moreover, LorbEvent includes features that the CORBA event service does not offer, originally: sending and receiving messages considering the subject of interest.

1 . Introdução

Plataformas de middleware [7] estão se tornando uma parte importante de muitos ambientes de desenvolvimento de software distribuído. Nas plataformas de middleware o estilo mais usual de interação entre os componentes é a chamada remota de procedimentos (RPC) que oferece um modelo de comunicação síncrona onde os elementos comunicantes (emissor e receptor) devem estar disponíveis para a comunicação no mesmo momento. Um outro modelo de comunicação, baseada em eventos, vem ganhando popularidade principalmente no contexto de ambientes distribuídos [8,9]. Na comunicação orientada a eventos, componentes comunicam-se gerando ou produzindo eventos e recebendo notificações de eventos. Quando um evento é gerado por um componente, ele é propagado para os componentes que declararam interesse em receber o evento. A geração do evento e a sua propagação são realizadas de forma assíncrona. Um elemento mediador, em alguns contextos denominado *canal de eventos*, é responsável pela propagação do evento. Na orientação a eventos a comunicação não é limitada a pares específicos de componentes. Um serviço de eventos implementa um mecanismo de *multicasting* que desacopla emissores de receptores de eventos [4].

Para comunicação baseada em eventos a OMG [2] especifica o serviço de eventos de CORBA [1]. O uso deste serviço envolve diversas chamadas para habilitar a comunicação e criar os objetos que dão suporte à sua operação. Desta forma, no código das aplicações que usam o serviço de eventos de CORBA, as funções específicas das aplicações misturam-se com as diversas funções de invocação ao serviço de eventos, comprometendo a legibilidade do código e dificultando sua compreensão. Além disto, o programador divide a atenção entre a aplicação e as extensas chamadas do serviço de eventos. Esta complexidade dificulta o uso do serviço e desestimula programadores a explorar as vantagens do modelo de comunicação assíncrona. Para viabilizar e incentivar a utilização do serviço de eventos é necessário incluir uma camada de software que abstraia a complexidade de se usar tal serviço. Ou seja, um nível de abstração adicional deve intermediar o acesso das aplicações ao serviço de eventos de CORBA mascarando as dificuldades existentes.

Este trabalho tem como objetivo apresentar uma biblioteca, *LOrbEvent*, que facilita o acesso ao serviço de Eventos de CORBA [1] através de uma linguagem interpretada e procedural – a linguagem Lua [3]. LOrbEvent abstrai as dificuldades de se usar o serviço de eventos de CORBA e torna viável que aplicações usem tal serviço sem precisar conhecer os detalhes da especificação CORBA nem emitir diversos comandos para habilitar o uso do serviço. Com o suporte de LOrbEvent, as aplicações podem ser estruturadas seguindo o modelo de comunicação orientada a eventos sem necessidade de incluir extensas chamadas ao serviço de eventos CORBA. Além disso, LOrbEvent adiciona facilidades que originalmente o serviço de eventos CORBA não fornece: envio e recebimento de mensagens considerando o assunto de interesse.

LOrbEvent está inserida no contexto de um ambiente de desenvolvimento de aplicações distribuídas baseadas em componentes CORBA com suporte à reconfiguração dinâmica – o ambiente LuaSpace [5,6]. LuaSpace é composto pela linguagem Lua e um conjunto de ferramentas baseadas nesta linguagem que visam facilitar a configuração e reconfiguração dinâmica de aplicações. Em LuaSpace a configuração de uma aplicação é escrita em Lua e composta por componentes CORBA. Uma das ferramentas de LuaSpace é um *binding* entre Lua e CORBA – LuaOrb [10] - que permite o uso de componentes CORBA como se fossem objetos Lua. Neste ambiente, a integração de uma biblioteca que viabiliza o uso do serviço de eventos agrega um mecanismo adicional para composição de aplicações, para interação entre componentes e para reconfiguração dinâmica uma vez que a ocorrência de um evento pode determinar a inclusão/remoção de componentes em uma aplicação. Além disso, a possibilidade de comunicação assíncrona entre componentes confere uma maior flexibilidade para a composição das aplicações.

Este artigo está estruturado da seguinte forma. A seção 2 apresenta brevemente conceitos básicos de CORBA, Lua e LuaSpace. A seção 3 apresenta o serviço de eventos de CORBA e os passos que devem ser seguidos para se usar o serviço. A seção 4 apresenta a biblioteca LOrbEvent e ilustra o seu uso em uma aplicação exemplo. A seção 5 apresenta os trabalhos relacionados. A seção 6 contém as conclusões.

2. Conceitos Básicos

2.1. CORBA

CORBA (*Common Object Request Broker Architecture*) [2,16] é um padrão proposto pelo *Object Management Group* (OMG) cujo propósito é permitir interoperabilidade entre aplicações em ambientes distribuídos e heterogêneos. Este padrão estabelece a separação

entre a interface de um objeto e sua implementação. Para descrição da interface do objeto, CORBA oferece a *linguagem para definição de interfaces (IDL)*. Para implementação do objeto CORBA, pode ser utilizada qualquer linguagem de programação que tenha o *binding* para CORBA.

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

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

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

CORBA oferece um conjunto de serviços para dar suporte ao desenvolvimento de aplicações distribuídas. Cada serviço é descrito por interfaces IDL e composto por objetos CORBA.

2.2. Lua

Lua [3] é uma linguagem de configuração interpretada e procedural com um sistema de tipos dinâmico: variáveis não têm tipos apenas valores é que estão associados a um tipo. Lua integra facilidades para descrição de dados e reflexividade em uma sintaxe simples. Além disso, inclui aspectos convencionais, como estruturas de controle(*if*, *while*,etc), variáveis locais, e características não convencionais como: funções são valores de primeira classe; *arrays associativos* (chamados de *tabelas*) são a única facilidade de estruturação de dados; *tag methods* é o mecanismo mais genérico para reflexão. *Tag methods* podem ser especificados para serem chamados em situações nas quais o interpretador Lua não sabe como proceder.

2.3. LuaSpace

LuaSpace [5] é um ambiente para 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. LuaSpace integra a plataforma CORBA com a linguagem Lua e oferece um conjunto de ferramentas baseadas em Lua com funções estratégicas para facilitar o desenvolvimento de aplicações baseadas em componentes e para promover reconfiguração dinâmica. Em LuaSpace a aplicação é escrita em Lua e pode ser composta por componentes implementados em qualquer linguagem que tenha o *binding* para CORBA. Componentes, scripts e *glue code* são os elementos que formam uma aplicação em LuaSpace.

Uma das principais ferramentas de LuaSpace é LuaOrb [10], um binding entre Lua e CORBA baseado na Interface de Invocação Dinâmica de CORBA (DII) que oferece acesso dinâmico a componentes CORBA da mesma forma que se faz acesso a objetos Lua. O acesso a objetos CORBA é realizado de forma transparente. LuaOrb também usa a Interface de Esqueleto Dinâmico (DSI) de CORBA para permitir instalação dinâmica de novos objetos em um servidor em execução.

Para usar um componente CORBA é necessário primeiro criar um *proxy* Lua usando a função ***createproxy*** de LuaOrb. Esta função cria um objeto Lua que representa um objeto CORBA.

Através do mecanismo de *tag methods*, as operações aplicadas sobre o *proxy* são tratadas por LuaOrb que as transforma em operações sobre o componente CORBA.

As demais ferramentas de LuaSpace não serão apresentadas neste artigo pois não são relevantes no contexto deste trabalho.

3. Serviço de Eventos de CORBA

A especificação do serviço de Eventos de CORBA [1] atribui aos servidores o nome de fornecedor de eventos (*Supplier*) e ao cliente o nome de consumidor de eventos (*Consumer*). Fornecedores (servidores) não emitem seus eventos diretamente para os consumidores (clientes). A comunicação entre estes objetos é intermediada por um objeto, chamado ***Canal de Eventos*** (*Event Channel*), definido pela especificação do serviço de eventos. O canal de eventos viabiliza a troca de eventos entre dois ou mais objetos, sem que seja necessário que os objetos conheçam-se mutuamente. Sua principal função é registrar os fornecedores e consumidores além de receber e emitir os eventos. A Figura 1 ilustra os elementos do serviço de eventos de CORBA. A especificação define dois modelos de canal de eventos: não tipado e tipado. No canal não tipado usa-se funções oferecidas pelo serviço (funções push() ou pull()) para enviar e receber eventos que não possuem um tipo definido e todos os consumidores que estão conectados neste canal recebem todos os eventos emitidos pelos fornecedores. No canal tipado, os fornecedores e consumidores definem, através de uma IDL, uma função que invocarão quando estiverem interessados em receber e enviar eventos.

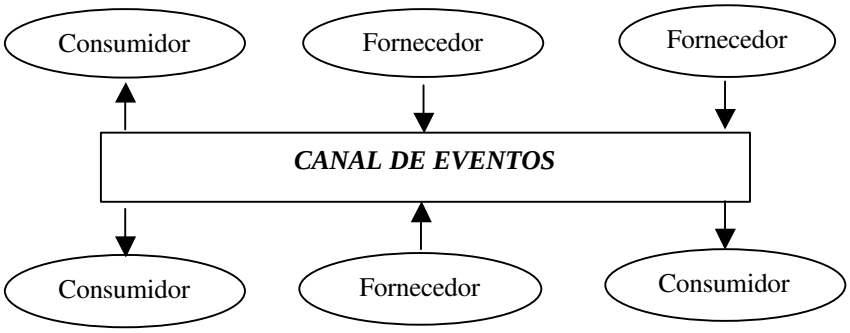

Figura 1 - Elementos do Serviço de Eventos CORBA

A especificação define também dois tipos de modelos de funcionamento: *push* e *pull*. No modelo *push,* o fornecedor é o elemento que toma iniciativa da comunicação. Este aspecto o define como um objeto ativo para o modelo. No modelo *pull* a iniciativa é tomada pelo consumidor, sendo este o objeto ativo neste modelo. Tal distinção é importante pois os objetos passivos devem implementar uma função (*callback*) e a registrá-la de modo que a partir do reconhecimento da chegada de uma nova notificação de eventos, essa função deve ser chamada para tratar tal notificação.

3.1. Usando o serviço de eventos de CORBA

Como todas as especificações CORBA, a especificação do serviço de eventos é descrita em IDL. Duas interfaces, a *CosEventChannelAdmin.IDL* e a *CosEventComm.IDL,* descrevem o serviço de eventos. A primeira interface define funções para conexão com o canal e a segunda, para envio e recebimento de eventos.

A Figura 2 ilustra as interfaces IDL do serviço de eventos CORBA e os passos necessários para uso de tal serviço. Para identificar os passos a serem seguidos é necessário

primeiramente definir o tipo do objeto a ser construído (objeto fornecedor ou consumidor) e o modelo a ser usado (push ou pull).

Inicialmente, deve-se criar o canal de eventos (usando as funções disponibilizadas na interface *EventChannel* ilustrada na Figura 2) e prepará-lo para trabalhar como fornecedor ou consumidor. Em seguida, a partir do modelo escolhido (push ou pull), deve-se criar uma representação do objeto (consumidor ou fornecedor) e a partir desta, conectar-se ao canal de eventos para que as notificações sejam enviadas e recebidas.

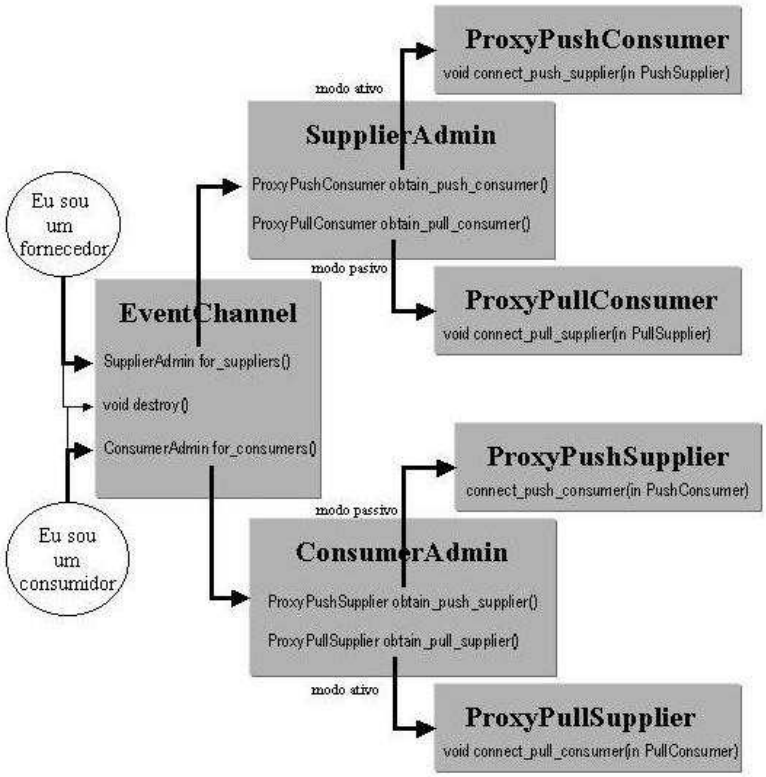

Figura 2 - Interfaces do Serviço de Eventos CORBA com seqüência de chamadas

3.2. Usando o serviço de eventos de CORBA via LUA

Usando *proxies* criados em Lua, através do LuaOrb, é possível estabelecer uma conexão com o *canal de eventos* e enviar ou receber eventos do canal. A Figura 3 apresenta o código de uma aplicação simples usando o modelo *push* onde fornecedores enviam a hora para os consumidores.

```
-------------------------------------------------------------------------------------------------------
1 LuaOrb_cfg.irep = createproxy{interface = "CORBA::Repository",
                          ior_file = "./ir.ref"}
2 proxy1 = createproxy{interface = "CosEventChannelAdmin::EventChannel",
                          ior_file = "./se.ref"}
3 canal = proxy1:for_suppliers()
4 proxy2 = createproxy{interface = "CosEventChannelAdmin::SupplierAdmin",
                          ior_file="./se.ref"}
5 fornecedor = canal:obtain_push_consumer()
6 proxy3 = createproxy{interface = "CosEventChannelAdmin::ProxyPushConsumer",
                          ior_file = "./se.ref"}
7 fornecedor:connect_push_supplier(nil)
8 fornecedor:push(clock())
-------------------------------------------------------------------------------------------------------
```

Figura 3 - Usando o serviço de eventos via Lua

Para que essa aplicação seja executada é necessário iniciar o serviço de eventos, iniciar o repositório de interfaces (RI) e carregar as IDL que serão usadas. No código da Figura 3 as funções que realizam estas tarefas são invocadas antes do fornecedor enviar a hora para o canal.

Inicialmente, na linha 1, LuaOrb é configurado para usar o RI. Em seguida, na linha 2, é criado o *proxy* para a interface E*ventChannel,* que oferece a função *for_suppliers(),* usada na linha 3 para obter a referência do canal de eventos. Na seqüência, é criado o *proxy* para a interface *SupplierAdmin* que fornece o método *obtain_push_consumer()*, usado para definir o *push* como modelo. A interface *ProxyPushConsumer,* cujo *proxy* é criado na linha 6, oferece o método *connect_push_supplier()* que é usado para conectar o objeto fornecedor ao canal de eventos. Enfim, na linha 8 estabelece-se o envio do evento.

Para a aplicação simples exemplificada na Figura 3 foram necessárias oito linhas de código, a criação de três *proxies* e ainda a preocupação com a seqüência de interfaces que devem ser referenciadas pois, como ilustrado na Figura 2, para cada modelo push/pull e consumidor/fornecedor tem-se uma seqüência diferente. Portanto, é inviável a utilização do serviço de eventos de CORBA em aplicações complexas que façam muito uso do serviço.

4. LOrbEvent

A biblioteca LOrbEvent, descrita nesta seção, foi desenvolvida com o objetivo de dar suporte às aplicações que necessitam usar o serviço de eventos de CORBA, fornecendo uma abstração para acesso a este serviço que facilita consideravelmente o seu uso. LOrbEvent habilita o serviço de eventos e o RI e cria automaticamente, e de forma transparente para o programador, os *proxies* para as interfaces IDL que o programa utiliza. Na subseção 4.1 será apresentada a arquitetura de LuaSpace onde LOrbEvent está inserida. A subseção 4.2 comenta o funcionamento de LOrbEvent apresentando as funções que a biblioteca oferece. Na subseção 4.3 é discutido um estudo de caso que exemplifica o uso da biblioteca LOrbEvent.

4.1. Arquitetura

A Figura 4 ilustra a parte da arquitetura do LuaSpace relevante para situar a biblioteca LOrbEvent. Em LuaSpace o script de configuração de uma aplicação pode ser escrito diretamente no console Lua ou armazenado em um arquivo. O script é submetido ao interpretador Lua que o executa fazendo as chamadas necessárias a cada diferente componente da arquitetura para tratar comandos.

LuaOrb é acionado nos casos de chamadas que fazem acesso a objetos CORBA. LuaOrb faz o mapeamento entre a chamada Lua e a chamada correspondente na plataforma CORBA. LuaOrb também é invocado quando são feitas chamadas para instalação dinâmica de objetos Lua em um servidor remoto.

Quando o interpretador Lua executa uma chamada de funções da LOrbEvent, ele invoca a implementação da função, disponível na biblioteca LOrbEvent. As funções de LOrbEvent são escritas em Lua e realizam o mapeamento para as operações correspondentes do serviço de eventos de CORBA. Através de LuaOrb, LOrbEvent faz chamadas à objetos, serviços e repositórios CORBA.

O ambiente LuaSpace também possibilita que o programador use diretamente as operações oferecidas pelos serviços CORBA. Para facilitar o uso desta opção, LuaSpace habilita automaticamente os serviços.

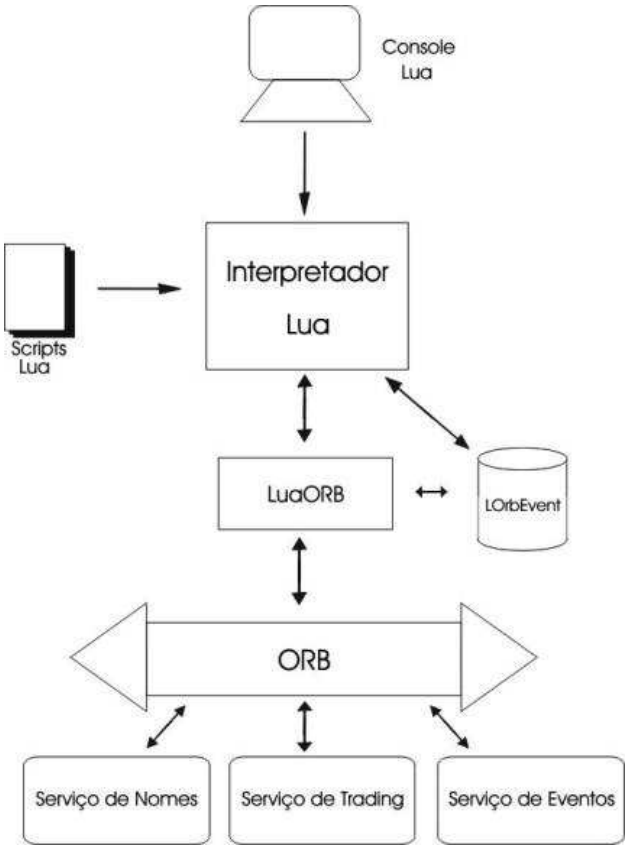

Figura 4 - Arquitetura de LuaSpace

4.2. Funcionamento

A biblioteca LOrbEvent oferece um conjunto de funções Lua que encapsulam a habilitação do serviço de eventos e as invocações às interfaces IDL, facilitando o uso deste serviço. A tabela 1 resume tais funções.

FUNÇÕES	DESCRIÇÃO
new(string)	Construtor do objeto. Retorna a referência do objeto que será usada para chamar os demais métodos.
sendevent([assunto],any)	Envia eventos. Retorna uma tabela.
getevent([assunto],[any])	Recebe eventos. Retorna any.
change_channel(string)	Muda o nome do canal se o mesmo existir e, caso não exista, cria o canal. Retorna uma tabela.
disconnect_send()	Desconecta fornecedor
disconnect_get()	Desconecta consumidor
disconnectAll()	Desconecta fornecedor e consumidor

Tabela 1 – Funções de LOrbEvent

O construtor new() é uma função que abstrai toda a dificuldade de criação de *proxies* na implementação de uma aplicação que utiliza o serviço de eventos. Esta função analisa os parâmetros passados a ela e, a partir deles, cria os *proxies* adequados para o funcionamento do serviço de eventos CORBA. Para trabalhar com canal sem tipo, utiliza-se a chamada padrão

new() ou *new("Untyped"),* e para trabalhar com um canal tipado, utiliza-se *new("Typed").* Por exemplo, a invocação *fornecedor = LOrbEvent:new()*, estabelece o uso de um canal não tipado. Em suma, a função *new* torna o serviço de eventos operacional e retorna a referência de um objeto que poderá usar as funções de envio (*sendevent()*) e recebimento de eventos (*getevent()*), além das funções auxiliares para criar novos canais (*change_channel*) e desconectar-se dos mesmos.

Para enviar eventos, LOrbEvent disponibiliza a função *sendevent*([assunto],any). Usando esta função pode-se enviar eventos para um assunto específico, para vários assuntos ou ainda sem especificar o assunto. O assunto de interesse é estabelecido no parâmetro opcional *assunto*. O segundo parâmetro desta função é o elemento a ser enviado, que pode assumir qualquer valor: string, função, número, etc. Este aspecto confere uma grande flexibilidade uma vez que como o evento não tem um tipo específico pré-definido, ele pode ser representado por diferentes elementos. Por exemplo, a chamada s*endevent("POO","Inicio da conferência")* envia o evento *"Inicio da conferência"* (uma mensagem do tipo *string*) para os que registraram interesse no assunto "POO". De forma semelhante, a chamada s*endevent({"POO","BD"},"Inicio da conferência")* envia a mensagem *"Inicio da conferência"* para os que registraram interesse nos assuntos "POO" e "BD". Para enviar uma mensagem sem especificar assunto de interesse, apenas omite-se o parâmetro *assunto*, por exemplo s*endevent("Inicio da conferência").* Uma grande flexibilidade oferecida pela biblioteca LOrbEvent para envio de eventos é permitir associar o evento à uma função. Por exemplo, na Figura 5 é definida uma função *function hora()* que retorna a hora atual. Na invocação s*endevent("POO", hora)* tal função é o evento da chamada e na execução ela retorna a hora para os objetos ligados ao canal que tenham especificado *"POO"* como assunto de interesse.

```
1 function hora()
2    return clock()
3   end
4 fornecedor = LOrbEvent:new()
5 fornecedor:sendevent("POO", hora)
```

Figura 5 – Envio de Evento através de Função Hora

Para a recepção de eventos, LOrbEvent oferece a função g*etevent*([assunto],[any]). Esta função retorna um valor que é o evento recebido. Ou seja, o retorno pode ser um string ou uma função. Os eventos podem ser recebidos considerando-se assuntos de interesse ou não. Por exemplo, para estabelecer interesse no assunto *"POO"*, usa-se a função get*event("POO").* Para estabelecer vários assuntos de interesses ("*POO"* e "*BD"*), usa-se a função get*event({"POO","BD"}).* Para não considerar interesse em um assunto específico usa-se apenas *getevent().* Da mesma forma que na chamada *sendevent*, na *getevent* pode-se também registrar uma função que é invocada a cada ocorrência de um evento. Esta função pode conter quaisquer comandos Lua inclusive comandos de reconfiguração da aplicação. Na Figura 6 é ilustrado um exemplo simples onde é definida a função *Imprime(msg)* que imprime o evento recebido na saída padrão. Neste exemplo, quando chegar um evento cujo assunto de interesse é *"POO"* , o evento será impresso na saída padrão.

```
1 function Imprime(msg)
2         print(msg)
3 end
4 consumidor = LOrbEvent:new()
5 consumidor:getevent("POO", Imprime)
```

Figura 6 - Recebimento de Evento via Função Imprime

LOrbEvent permite que uma aplicação, usando a função *change_channel(string),* mude o canal em que está conectada. Esta função muda o canal atual para o canal informado como parâmetro. Caso o canal de destino não exista, ele será criado automaticamente. Como forma de assegurar o fim do envio e do recebimento de eventos usa-se as funções *disconnect_send()* para desconectar o fornecedor, *disconnect_get()* para desconectar o consumidor e *disconnectAll()* para desconectar tanto o fornecedor como o consumidor.

Com as funções oferecidas por LOrbEvent, a aplicação ilustrada na Figura 3 pode ser implementada usando-se apenas dois comandos, como mostra a Figura 7.

```
1 fornecedor = LOrbEvent:new()
2 fornecedor:setevent(clock())
```

Figura 7 - Uso do serviço de eventos via LOrbEvent

4.3. Estudo de Caso

Para exemplificar a facilidade conferida por LOrbEvent, será descrita a implementação de uma multiplicação paralela de duas matrizes, ilustradas na Figura 8. Nesta aplicação dadas duas matrizes 2x2, o retorno deve ser o produto entre elas calculando-o de forma paralela.

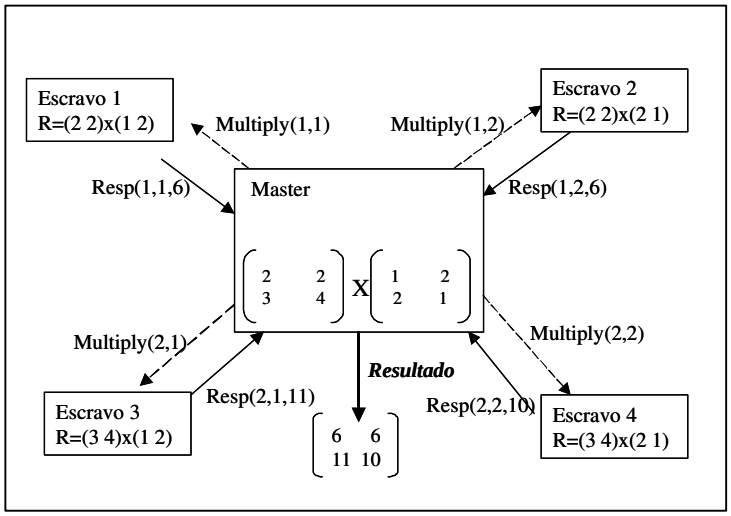

Figura 8 - Multiplicação Paralela de Matrizes

Para implementar tal paralelismo, LOrbEvent é usado para enviar a partir de um componente chamado "master" os dados e código para os outros componentes chamados "escravos" que executarão esse código para multiplicar uma linha por uma coluna e enviar a resposta ao "master".

```
-------------------------------------------------------------------------------------------------------
dofile(“LOrbEvent.lua”)
function callbackSlave(msg)
   dostring(msg)
end
fslave = LOrbEvent:new()
fslave:getevent(“Escravo1”, callbackSlave)
-------------------------------------------------------------------------------------------------------
```

Figura 9 - Código do Escravo1.lua

Inicialmente deve-se executar os “escravos” em máquinas distintas para ser possível o paralelismo. A Figura 9 mostra o código dos escravos. Nesse código, o primeiro escravo ou “Escravo1” registra interesse, através da função *getevent,* em todos os eventos direcionados ao “Escravo1”, e determina no segundo parâmetro a função (*callbackSlave)* que será chamada quando tal evento ocorrer. Essa função tem como único objetivo executar todos os eventos (mensagens) que forem enviadas pelo “master”. O código dos demais escravos é semelhante ao apresentado, variando apenas o registro de interesse, que assume o valor “Escravo2” para o segundo escravo, e assim por diante.

A Figura 10 ilustra o código do “master”. Inicialmente o “master” registra o interesse, através da função *getevent,* em todos os eventos direcionados ao “master” e determina, através do segundo parâmetro, a função (*CallbackMaster)* que será chamada quando tal evento ocorrer. Essa função tem como único objetivo executar todos os eventos (códigos) que forem enviadas pelos escravos.

Em seguida o “master” envia a todos os escravos a função *Multiply(i,j)*. O passo seguinte é enviar as matrizes com seus parâmetros para os respectivos escravos de modo que eles possam executar a função *Multiply.* É importante ressaltar a parte do código onde o “master” solicita que o “escravo” execute tal função:

*master:sendevent("Escravo"..tostring(((i-1)*2)+j),"multiply("..i..","..j..")")*

Neste momento, o escravo selecionado pelos parâmetros *i* e *j,* recebe o evento e executa a função *Multiply* previamente declarada. No escopo dessa função o escravo, ao ter o resultado armazenado na variável *result,* envia-o ao master invocando a função *Resp(i,j,x)*, definida no contexto do “master”. Esta função irá tratar o resultado. Quando todos os escravos enviarem suas respostas, a função *Resp* chama as funções responsáveis pela exibição do resultado.

```
-------------------------------------------------------------------------------------------------------
-- FUNÇÕES AUXILIARES --
Multdef = [[function multiply(i,j)
               local k, result = 1, 0
               while line[k] ~= nil do
                   result = result + (line[k]*column[k])
                   k = k + 1
               end
               slave = LOrbEvent:new()
               slave:sendevent("Master","Resp("..i..","..j..",
                                              "..tostring(result)..")")
               end]]

function show_matriz(M)
    ...
end
function show_result()
   ...
end
function Resp(i,j,x)
   h = h + 1
   R[i][j] = x
   if h == 4 then
      show_result()
   end
end
function tabtostring(tab)
   local text = "{"..tostring(tab[1])..","..tostring(tab[2])..."}"
   return text
end

function CallbackMaster(msg)
   dostring(msg)
end

-- INÍCIO DO PROGRAMA --
dofile("LOrbEvent.lua")
h = 0
R = {{},{}}
master = LOrbEvent:new()
master:getevent("Master", CallbackMaster)
master:sendevent({"Escravo1","Escravo2","Escravo3","Escravo4"},Multdef)
i = 1
A = {{2,2},{3,4}}
B = {{1,2},{2,3}}
while i <= 2 do
   local j = 1
   while j <= 2 do
      master:sendevent("Escravo"..tostring(((i-1)*2)+j),"line ="
                                                  ..tabtostring(A[i]))
      master:sendevent("Escravo"..tostring(((i-1)*2)+j),"column ="
                                                  ..tabtostring(B[j]))
      master:sendevent("Escravo"..tostring(((i-1)*2)+j),"multiply("
                                                    ..i..","..j..")")
      j = j + 1
   end
   i = i + 1
end
-------------------------------------------------------------------------------------------------------
```

Figura 10 - Código do master.lua

5. Trabalhos Relacionados

SmartSockets [12] é um *middleware* comercial desenvolvido pela Talarian que oferece opção de comunicação síncrona e assíncrona. Para comunicação assíncrona, SmartSockets usa um modelo orientado a eventos nos quais programas registram interesse em um assunto e publicam mensagens sobre um determinado assunto. Os programas desenvolvidos na plataforma SmartSockets podem registrar interesse em um assunto específico ou em múltiplos assuntos. Da mesma forma, LOrbEvent permite associar a um evento os assuntos de interesse. SmartSockets pré-define um conjunto de tipos de eventos comumente utilizados. Os programadores podem criar seus eventos como instâncias destes tipos ou pode definir outros tipos específicos da aplicação. As aplicações são escritas em C. Por ser um produto comercial, não foram publicados detalhes sobre sua implementação. Enquanto o SmartSockets desenvolveu um *middleware* proprietário e incluiu o suporte à comunicação assíncrona, LOrbEvent explora o suporte do padrão CORBA oferecendo uma API para facilitar o uso do serviço de eventos de CORBA. Além disso, LOrbEvent também inclui o tratamento a assuntos de interesse. Com LOrbEvent, LuaSpace passou a agregar suporte tanto a comunicação síncrona como a assíncrona, igualmente ao SmartSockets.

JEDI [13] (*Java Event-based Distributed Infrastructure*) é uma infraestrututa de suporte ao desenvolvimento e operação de sistemas baseados em eventos. Na arquitetura do JEDI o ED (*Event Dispatcher*) é o objeto equivalente ao canal de eventos de CORBA, que media o envio e recebimento de eventos. Em JEDI um evento é um conjunto ordenado de strings: o primeiro string é o *nome do evento* e os demais strings são parâmetros do evento. Um evento é representado usando a notação similar a chamada de funções em linguagens de programação tradicionais. Por exemplo, no evento *published(computing reviews, middleware), published* é o nome do evento e *computing reviews* e *middleware* são os parâmetros. Em JEDI objetos podem registrar interesse em um evento específico ou em um *padrão de evento*. O padrão de evento é uma forma simples de expressão regular, por exemplo, o padrão *published (computing *, *)* é usado para declarar interesse em todos os eventos com nome *published* e tendo dois parâmetros onde o primeiro começa com *computing* e o segundo pode ser qualquer string. Diferentemente de JEDI, o serviço de eventos de CORBA não fornece suporte para a especificação de assuntos de interesse, no entanto LOrbEvent incorporou esta facilidade e, como em JEDI, permite o registro de interesse em um assunto específico ou em vários assuntos. O suporte para o uso de expressões regulares está sendo implementado em LOrbEvent. JEDI define um tipo especial de objeto, chamado *objeto reativo*, que possui um método abstrato (chamado *processMessage*) que tem de ser especificado pelo programador e é automaticamente invocado toda vez o objeto reativo recebe um evento. Da mesma forma, LOrbEvent permite a associação de uma função à um evento, de forma que tal função é invocada quando um evento é recebido. Adicionalmente, LOrbEvent também permite que eventos sejam representados por funções e objetos.

Os dois trabalhos analisados oferecem soluções proprietárias para comunicação assíncrona enquanto que no LOrbEvent a opção foi usar o padrão CORBA devido ao seu suporte à interoperabilidade. SmartSockets e JEDI influenciaram LOrbEvent com a idéia de permitir se estabelecer assuntos de interesse – facilidade que o serviço de eventos de CORBA não oferece. Além disso, o uso de expressões regulares no parâmetro *assunto,* característica de JEDI, está sendo incorporada em LOrbEvent. Um outro diferencial de LOrbEvent em relação a JEDI e SmartSockets está no fato de LOrbEvent flexibilizar a estrutura do evento, permitido que um evento seja representado por uma função, o que enriquece a semântica do evento.

6. Conclusões

Este trabalho apresentou uma biblioteca de software que implementa uma camada de abstração sobre o serviço de eventos de CORBA, viabilizando a sua utilização no desenvolvimento de aplicações distribuídas baseadas em componentes. Além de facilitar o uso do serviço de eventos, LOrbEvent estende a funcionalidade de tal serviço, oferecendo facilidades que a especificação do serviço de eventos de CORBA não oferece: envio e recebimento de eventos considerando o(s) assunto(s) de interesse. De acordo com a especificação do serviço de eventos, todos os componentes ligados a um canal recebem todos os eventos emitidos para o canal. A vantagem de se especificar assuntos de interesse é poder selecionar, para cada evento que chega no canal, os componentes que estão interessados em tal evento. Portanto, LOrbEvent dispõe de um poder expressivo adicional que evita que componentes sejam notificados da ocorrência eventos que não lhe interessam.

A importância de se permitir que o evento possa ser um string, função ou outro elemento da linguagem é facilitar a interpretação semântica do evento. Se o evento não disponibiliza informações suficientes para sua interpretação, o componente receptor do evento pode necessitar envolver-se em uma complexa interação com o emissor do evento de forma a obter informações adicionais [4]. Neste caso, tem-se também o problema da dificuldade de descobrir o emissor do evento pois uma das características do modelo de eventos é o anonimato da origem e do destino da comunicação. Como LOrbEvent flexibiliza a estrutura dos eventos que são fornecidos e consumidos por componentes, pode-se incluir no evento informações completas para que o receptor possa interpretá-lo adequadamente.

Com LOrbEvent o programador dispõe de funções simples para envio e recebimento de eventos e pode utilizar comunicação assíncrona usando a mesma linguagem empregada na configuração da aplicação. Desta forma, o programa de configuração da aplicação é simples e compreensível pois não inclui uma série de comandos alheios ao domínio da aplicação para iniciar o serviço de eventos e seus elementos. Além disso, o programador não precisa conhecer as particularidades do serviço de eventos de CORBA para utilizar o suporte que este serviço oferece para comunicação assíncrona.

O modelo de orientação a eventos enquadra-se bem com a idéia de desenvolvimento baseado em componentes pois favorece a composição da aplicação uma vez que componentes não precisam estar diretamente ligados à outros componentes como no tradicional modelo de chamada remota de procedimento. Com a comunicação via eventos, um componente pode estar em operação sem tomar conhecimento da existência de outros componentes com os quais ele interage via o canal de eventos. Este aspecto beneficia a reconfiguração dinâmica uma vez que possibilita a inserção/remoção de componentes em uma aplicação sem afetar diretamente os outros componentes [4]. Um outro aspecto de LOrbEvent que também favorece a reconfiguração dinâmica é o fato de poder se definir uma função para tratar um evento. Tal função pode conter comandos de reconfiguração da aplicação que serão disparados quando o evento ocorrer.

Com a incorporação de LOrbEvent no ambiente LuaSpace, o programador passou a dispor de duas formas de comunicação entre componentes: a tradicional comunicação síncrona baseada em RPC oferecida por CORBA e também as funções para comunicação assíncrona fornecida por LOrbEvent que abstrai o uso do serviço de eventos. Portanto, nas aplicações desenvolvidas neste ambiente é possível utilizar qualquer uma das formas de comunicação como também, se necessário, as duas em um mesmo programa. A disponibilidade da comunicação assíncrona e da síncrona em um ambiente de desenvolvimento de aplicações é

primordial pois cada um dos tipos de comunicação é adequado para diferentes necessidades. Os dois tipos de comunicação não são alternativos, portanto devem co-existir em um mesmo ambiente. Por exemplo, em JEDI, uma infra-estrutura tipicamente orientada a eventos, foi identificada a necessidade de existir uma operação síncrona para os casos onde é necessário que o receptor de uma mensagem envie um valor de retorno.

A idéia apresentada neste trabalho, de provê uma camada de software em um nível mais alto de abstração para tornar viável o uso do serviço de eventos de CORBA, não se restringe simplesmente ao serviço de eventos mas também a qualquer outro serviço CORBA uma vez que a complexidade de usar tais serviços é praticamente equivalente, com a ressalva de que o serviço de eventos envolve uma quantidade maior de objetos a serem habilitados: canal de eventos, consumidores e fornecedores. Em [14] são apresentados mecanismos para seleção dinâmica de objetos distribuídos que exploram uma camada de abstração sobre os serviços de Nomes e de Trading de CORBA.

O OMG oferece serviço de notificação que estende o serviço de eventos com mecanismos para dar suporte a expressões compostas e combinação de eventos. LOrbEvent pode ser facilmente adaptada para incluir as extensões propostas por tal serviço e este será o próximo passo deste trabalho.

O ambiente apresentado está operacional em uma plataforma Linux 2.2.17 (Conectiva 6), usando Lua versão 3.2.2, LuaOrb versão 1.5 e a implementação CORBA fornecida pela Iona Technologies - ORBacus [17].

Referências Bibliográficas

[1] Object Management Group-OMG (1997): "CORBA services:Common Object Services Specification",formal/97-07-04, July 1997

[2] OMG. The Common Object Broker Architecture and Specification. Technical Report Revision 2.2, OMG, 1998.

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

[4] A. Carzaniga, E. Di Nitto, D. Rosenblum and A. Wolf. Issues in Supporting Event-based Architectural Styles. In *Proceedings of the Third International Workshop on Software Architectures (ISAW-3),* Orlando, USA, Nov. 1998.

[5] T. Batista and N. Rodriguez. Configuração de Aplicações no LuaSpace. In *Anais do 18° Simpósio Brasileiro de Redes de Computadores (SBRC*), Belo Horizonte – MG, Maio 2000.

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

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

[8] D. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717-734, September 1995.

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

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

[11] A. Carzaniga, D. Rosenblum and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, Vol. 19, No. 3, August 2001, 332-383.

[12] Talarian. "Mission Critical Interprocess Communications – an Introduction to SmartSockets" – White Paper. Available at http://www.talarian.com/

[13] G. Cugola, E. Di Nitto and A. Fuggetta. The JEDI event-based infrastrucuture and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering (TSE)*, 27(9):827-850, September 2001.

[14] T. Batista, J. N. Morais, M. Carvalho and W. Teixeira. Seleção Dinâmica de Objetos Distribuídos no Ambiente LuaSpace. A ser apresentado no *20° Simpósio Brasileiro de Redes de Computadores (SBRC 2002)*, Búzios – RJ, Maio 2002.

[15] Talarian: Everything You Need to Know About Middleware – White Paper. Available at http://www.talarian.com/industry/middleware/whitepaper.pdf

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

[17] ORBacus Home Page – http://www.ooc.com/