Suporte para Comunicação Assíncrona com QoS em Aplicações CORBA

Nélio Cacho, Thaís Batista

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

cacho@lcc.ufrn.br, thais@ufrnet.br

Abstract. *In this article we present the CORBA support for asynchronous communication (event service and notification service) with Quality of Service (QoS), we discuss the complexities of using the event and notification services and we present a library to asynchronous communication – LOrbNotify – that eliminates such complexities. LOrbNotify offers an abstraction over the CORBA notification service that avoids that applications include many services invocations. Besides, in order to avoid inconsistence, LOrbNotify offer a mechanism to unify QoS properties of events suppliers and consumers. We also present a case study in order to exemplify the facility of using the library.*

Resumo. *Neste artigo apresentamos o suporte de CORBA para comunicação assíncrona (serviço de eventos e serviço de notificação) com Qualidade de Serviço (QoS), discutimos as complexidades de utilização dos serviços de eventos e de notificação e apresentamos uma biblioteca para comunicação assíncrona – LOrbNotify - que elimina tais complexidades. LOrbNotify oferece uma abstração sobre o serviço de notificação de CORBA que evita que as aplicações contenham as extensas chamadas ao serviço. Além disso, LOrbNotify inclui um mecanismo para unificação de propriedades de QoS entre fornecedores e consumidores de eventos de modo a evitar inconsistências. Para exemplificar a simplicidade de uso da biblioteca, apresentamos um estudo de caso.*

1. Introdução

A natureza assíncrona de diversas aplicações distribuídas promove a interação via eventos como uma forma natural de comunicação entre os componentes da aplicação [Carzaniga, Rosenblum and Wolf 2001]. Plataformas de Middleware [Bernstein 1996] tem sido amplamente utilizadas como infraestruturas de desenvolvimento de aplicações distribuídas. Em muitas delas, o mecanismo tradicional de comunicação é a chamada remota de procedimento (RPC), que estabelece um modelo de comunicação síncrona com emissores explicitamente comunicando-se com receptores. No entanto, tais plataformas, em geral, também oferecem mecanismos de comunicação assíncrona, permitindo um maior desacoplamento entre os elementos comunicantes. Na comunicação assíncrona via eventos, componentes publicam eventos e também registram interesse em receber eventos.

Para suporte a comunicação assíncrona, CORBA (Common Object Request Broker Architecture) [OMG 1998, Tari and Bukhres 2001] oferece os serviços de eventos e de notificação [OMG 1999]. O serviço de notificação é uma extensão do serviço de eventos que inclui suporte a Qualidade de Serviço (QoS) e Filtragem de eventos. Atualmente, Qualidade de Serviço (QoS) é um aspecto importante para muitas aplicações portanto é essencial que o serviço de comunicação assíncrona permita que essas aplicações estabeleçam os requisitos de QoS para os seus eventos, tais como: garantia a entrega dos eventos enviados, o estabelecimento de prioridades para os eventos, a hora de envio dos eventos, etc.

Apesar dos serviços de eventos e de notificação eliminarem o acoplamento direto entre componentes, há uma grande complexidade na sua utilização: é necessário fazer várias invocações as operações para iniciar o serviço e para criar os objetos que dão suporte ao funcionamento do serviço. Portanto, no código das aplicações que usam o serviço de eventos e de notificação de CORBA, as funções específicas das aplicações misturam-se com as diversas funções de invocação ao serviço, comprometendo a legibilidade do código e dificultando sua compreensão. Além disto, o programador divide a atenção entre a aplicação e as extensas chamadas aos serviços de comunicação assíncrona. Esta complexidade dificulta o uso dos serviços e desestimula programadores a explorar as vantagens do modelo de comunicação assíncrona. Para viabilizar e incentivar a utilização de tais serviços é necessário incluir uma camada de software que abstraia a complexidade de se usar tais serviços. Ou seja, um nível de abstração adicional deve intermediar o acesso das aplicações ao serviço de eventos e de notificação de CORBA tornando transparente as dificuldades existentes.

Este trabalho tem como objetivos: (1) apresentar os serviços de comunicação assíncrona de CORBA, detalhando o serviço de notificação, que oferece suporte a Qualidade de Serviço (QoS) e Filtragem de Eventos; (2) apresentar uma biblioteca que abstrai o uso do serviço de notificação de CORBA - LOrbNotify; (3) exemplificar o uso da biblioteca em uma aplicação exemplo que explora o suporte do serviço de notificação para envio/recebimento de eventos com QoS e critérios de filtragem de eventos.

LOrbNotify abstrai as dificuldades de se usar o serviço de notificação de CORBA, mantendo todas as suas funcionalidades 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. Este trabalho consiste na continuidade do trabalho publicado em [Batista, Cacho and Galvão 2002] cujo foco foi em uma abstração para uso do serviço de eventos, não incluía o serviço de notificação nem suporte a Qualidade de Serviço (QoS) e a filtragem de eventos a nível do serviço CORBA. A filtragem de eventos era implementada pela biblioteca.

LOrbNotify está inserida em um ambiente de desenvolvimento de aplicações CORBA com suporte a reconfiguração dinâmica - LuaSpace [Batista and Rodriguez 2000]. Em LuaSpace a aplicação é composta por componentes CORBA e sua configuração é escrita em Lua [Ierusalimschy, Figueiredo and Celes 1996] – uma linguagem interpretada e procedural. LuaSpace é composto pela linguagem Lua e por um conjunto de ferramentas baseadas em Lua com suporte para configuração e reconfiguração dinâmica das aplicações. Uma destas ferramentas é LuaOrb [Cerqueira, Cassino and Ierusalimschy 1999] – um binding entre Lua e CORBA 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 notificação 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 o suporte a comunicação assíncrona com garantia de Qualidade de Serviço oferecido por CORBA. A seção 3 discute detalhadamente o serviço de notificação de CORBA incluindo os tipos de eventos, a arquitetura do serviço e os passos que devem ser seguidos para se usar o serviço. A seção 4 apresenta a biblioteca LOrbNotify e ilustra o seu uso em uma aplicação exemplo. A seção 5 contém as conclusões.

2. Comunicação Assíncrona com QoS em CORBA

A comunicação assíncrona em CORBA é baseada na troca de eventos entre um conjunto de componentes de software que comunicam-se uns com os outros através da produção e da recepção de eventos. Nesse modelo, a ocorrência de algo representa um evento. Eventos são considerados elementos atômicos, podendo ser gerados e consumidos de forma assíncrona, não sendo necessário o bloqueio dos participantes da interação.

Uma das formas de se utilizar comunicação assíncrona em CORBA é através do serviço de eventos. A especificação desse serviço introduz alguns conceitos como a definição de papéis para os objetos envolvidos na comunicação assíncrona: *fornecedor* e *consumidor*. Os objetos fornecedores são os responsáveis pelo envio de eventos que não são entregues diretamente aos consumidores, mas sim a um elemento intermediário, chamado *canal de eventos.* O canal de eventos repassa os eventos para os consumidores registrados no canal.

A especificação do serviço de eventos oferece a comunicação assíncrona através de modelos para envio e recebimento de eventos: *push* e *pull*. No modelo *push*, o fornecedor é o elemento que toma a iniciativa da comunicação, isto o define como o 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 na chegada de uma nova notificação de eventos, essa função é chamada para tratar tal notificação.

O serviço de eventos opera sobre dois tipos de canais: *tipado* e o *sem tipo*. No canal sem tipo o evento é definido como do tipo *Any*, o que possibilita enviar, através dos métodos *Push* e *Pull*, valores de quaisquer tipos definidos na Linguagem de Definição de Interfaces (IDL) de CORBA. No canal tipado pode-se escolher como será a estrutura de dados do evento e quais os métodos que substituem *Push* e *Pull* no envio e recebimento de eventos. A Figura 1 ilustra como ocorre o fluxo de eventos, em um canal sem tipo, entre os elementos e também a seqüência em que os objetos da arquitetura são criados. O primeiro objeto a ser criado é o canal de eventos, sua função é desacoplar os fornecedores dos consumidores e administrar o fluxo de eventos. A partir

do canal são criados um, e apenas um, objeto *Consumidor Admin* e um *Fornecedor Admin*. Esses objetos disponibilizam os métodos para a criação dos vários *proxies Push* e *Pull* que permitem aos fornecedores e consumidores efetivamente enviar e receber eventos.

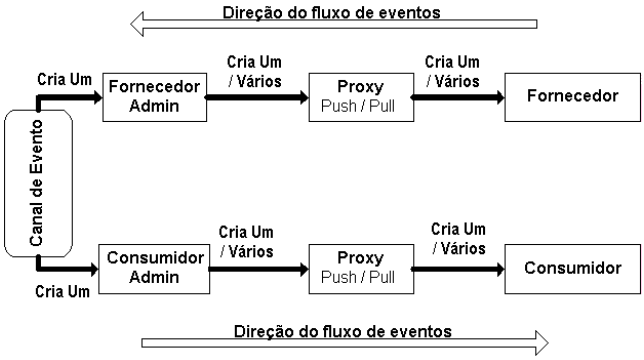

Figura 1 – Arquitetura do Serviço de Eventos

A especificação do serviço de eventos, apesar de oferecer compatibilidade com os conceitos naturais de comunicação assíncrona, não oferece suporte para Qualidade de Serviço. Isto significa que não há como determinar os requisitos de QoS da aplicação para o envio e recebimento dos eventos. O crescente número de aplicações que exigem garantia de Qualidade de Serviço, como controle de conferência, sistemas móveis e gerência de falhas em redes, está aproximando a comunicação assíncrona da Qualidade de serviço (QoS). As aplicações citadas caracterizam-se por realizarem suas ações baseadas em resposta a algumas mensagens e por terem como exigência primordial a garantia de que tais mensagens devem realmente chegar aos seus destinos e em um intervalo de tempo fixo. A união da comunicação assíncrona com QoS permite se estabelecer os requisitos não funcionais da aplicação em relação ao envio e recebimento de eventos, como exemplo: confiabilidade (garantia de entrega dos eventos), persistência (reconstituição exata dos objetos envolvidos na comunicação após o reinício do sistema), prioridade (envio de eventos com prioridade de disseminações diferentes), etc.

Como a especificação do serviço de eventos não define suporte para QoS, alguns trabalhos [Orvalho, Andrade and Boavida 1999, PeerLogic 1998] estenderam o serviço incluindo tratamento a QoS. No entanto, tais extensões são proprietárias e, portanto, as aplicações baseadas nestas extensões tem pouca portabilidade, uma vez que cada trabalho implementa suporte a propriedades de QoS específicas.

Como solução para esse problema, o OMG publicou uma extensão do serviço de eventos chamada de serviço de notificação [OMG 1999], um serviço que além de incorporar características de QoS também introduz filtragem de eventos.

3. Serviço de Notificação CORBA

O serviço de notificação surgiu como extensão do serviço de eventos com o objetivo de disponibilizar comunicação assíncrona baseada em eventos juntamente com suporte a filtragem de eventos e qualidade de serviço(QoS).

Os benefícios da Qualidade de Serviço numa aplicação são efetivos apenas quando as propriedades de QoS estão presentes em todos os níveis da arquitetura da

aplicação. O serviço de notificação define propriedades de QoS em cada um dos diversos níveis. A Figura 2 ilustra os diversos níveis definidos pelo serviço de notificação: Canal de eventos, Admin, Proxy e Clientes.

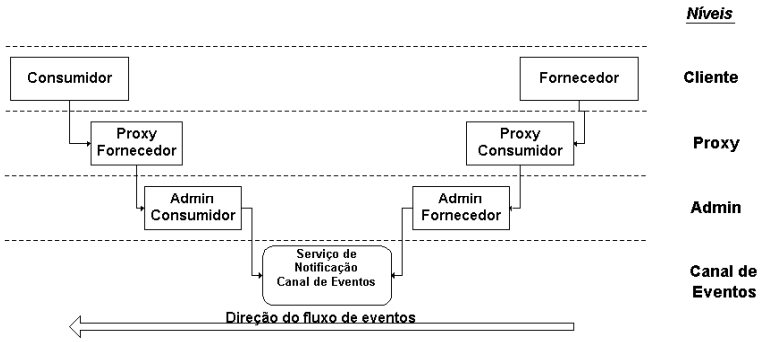

Figura 2 – Níveis do serviço de Notificação

Para dar suporte ao acréscimo dessas novas propriedades, em cada um dos níveis, o serviço de eventos foi reestruturado passando a definir novos tipos de eventos e a agrupar seus objetos componentes numa forma diferenciada da existente.

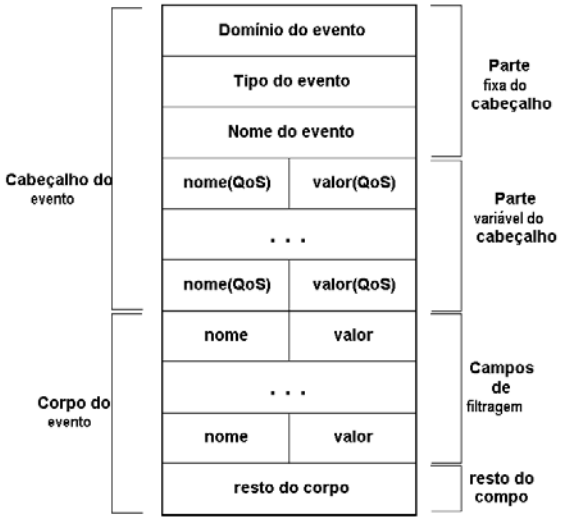

Figura 3 – Estrutura do evento estruturado

3.1 Tipos de eventos

A exigência de ter, no evento, características de QoS, determinou a criação de dois novos tipos de eventos para o serviço de notificação: eventos *estruturados* e *seqüenciais*. Os eventos seqüenciais são seqüências de eventos estruturados. Por motivo de compatibilidade com o serviço de eventos, o evento de tipo *Any* foi mantido.

A Figura 3 ilustra a composição do evento estruturado e sua divisão em quatro partes: cabeçalho fixo, cabeçalho variável, campos de filtragem e o restante do corpo. Na parte fixa do cabeçalho está o nome do evento e as informações relacionadas ao domínio da aplicação (telecomunicações, finanças, saúde, etc) e ao tipo de evento dentro do domínio (alarme de comunicação, falta de estoque, sinais vitais, etc.). A parte variável do cabeçalho contém os dados relacionados à propriedades de Qualidade de Serviço. Esses dados são opcionais mas, quando fornecidos, devem definir o par

nome/valor. A tabela 1 mostra as propriedades de QoS oferecidas pela especificação do serviço de notificação para um evento individual e suas respectivas faixas de valores.

Tabela 1. Propriedades de QoS suportadas por um evento individual

Nome da propriedade	Valores possíveis
EventReliability	0: não existe a garantia de entrega do evento.
	1: o evento é tratado como persistente e deve ser entregue mesmo em caso de falha.
Priority	Indica a ordem em que o evento será entregue. A mais baixa prioridade é –32.767 e a mais alta é 32.767.
StartTime	Determina o tempo absoluto (ex.: 11/06/2002 às 20:00) que o canal de eventos deve enviar o evento para seu destino.
StopTime	Determina o tempo absoluto (ex.: 11/06/2002 às 20:00) em que um evento será descartado.
Timeout	Determina o tempo relativo (ex.: 2 min depois de recebido) para descartar o evento.

A segunda parte variável do evento estruturado está relacionada ao processo de filtragem de eventos. Também introduzida pelo serviço de notificação, a filtragem de eventos é definida através da atribuição de pares nome/valor, onde *nome* recebe a propriedade que será avaliada pelo filtro e *valor* recebe o quantificador usado para comparação. Após o evento ter sido enviado com características de filtragem o consumidor pode, através de restrições, receber apenas os eventos de interesse baseado nos aspectos de filtragem estabelecidos. As restrições usadas no processo de filtragem são definidas usando-se a EXTENDED_TCL, onde TCL é *Trader Constraint Language.* A última parte que compõe o evento estruturado é responsável pelo armazenamento dos dados a serem enviados. Esses dados são definidos como do tipo *Any*, podendo assumir qualquer valor IDL válido.

3.2 Arquitetura do serviço de Notificação

A arquitetura do serviço de notificação mantém, em sua especificação, conceitos definidos pelo serviço de eventos: os mesmos tipos de papéis (fornecedor e consumidor), os métodos (*push* e *pull*) e os tipos de canal (tipado e sem tipo). No entanto, foram necessárias mudanças na estruturação dos objetos componentes do serviço de eventos. Essas mudanças podem ser observadas comparando-se a figura 4 com a figura 1.

As mudanças são as seguintes:

- Os objetos, ao criarem novos objetos descendentes, geram um identificador único e o armazenam numa lista de objetos criados. Portanto, um objeto Canal de eventos possui uma lista com todos os objetos *Admins*, que, por sua vez, também possui uma lista de todos os objetos *Proxies*. Essa seqüência é obedecida até o nível de Clientes (fornecedor e consumidor). Este aspecto permite que um cliente localize qualquer elemento presente no serviço de notificação - isso não é possível no serviço de eventos;

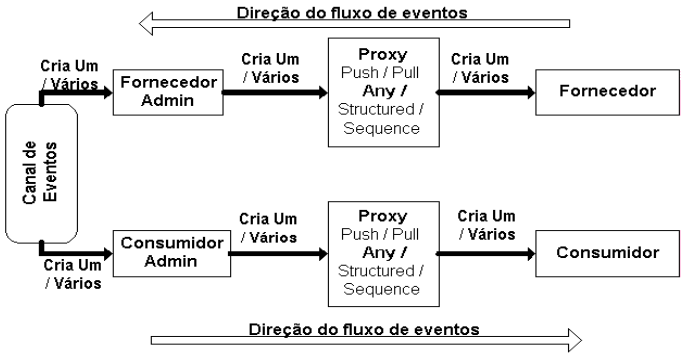

Figura 4 – Arquitetura do serviço de Notificação

- O canal de eventos pode criar mais de um objeto *Admin*, permitindo otimizar a manipulação de clientes com requisitos idênticos e tratar os objetos *Admins* como grupos de objetos *Proxies*.
- Os objetos *proxies* são divididos em três tipos: os que recebem e enviam eventos *Any*, os que recebem e enviam eventos estruturados e os que recebem e enviam eventos seqüenciais.

Tabela 2. Propriedades de QoS suportadas pelos objetos Canal de eventos, *Admins* e *Proxies*

Nome da propriedade	Valores possíveis
EventReliability	*BestEffort*: não existe a garantia de entrega do evento.
	Persistent: o evento é tratado como persistente e deve ser entregue mesmo em caso de falha. Deve ser usado com *ConnectionReliability= Persistent*
ConnectionReliability	*BestEffort*: após uma falha não há garantia que as conexões são restabelecidas.
	Persistent: garante que todas as conexões serão restabelecidas após uma falha.
MaxEventsPerConsumer	(Valor > 0): Determina o número máximo de eventos num determinando tempo que podem ser enviados para um consumidor.
OrderPolicy	Determina a ordem que os eventos serão entregues. Valores permitidos: *AnyOrder*: qualquer ordem, *FifoOrder*: ordem de chegada, *PriorityOrder*: por ordem de prioridade e *DeadlineOrder*: por ordem de expiração.
DiscardPolicy	Determina a ordem que os eventos serão descartados após o valor de *MaxEventsPerConsumer* ter sido ultrapassado. Valores permitidos: *AnyOrder*: qualquer ordem, *FifoOrder*: ordem de chegada, *PriorityOrder*: por ordem de prioridade e *DeadlineOrder*: por ordem de expiração.

As mudanças realizadas na arquitetura do serviço de eventos priorizam a separação explícita entre cada um dos níveis (Canal de eventos, Admins, Proxies e Clientes), adequando-se as necessidade de QoS e facilitando a administração do serviço de notificação pois, a medida que se atribui uma propriedade de QoS ou um requisito de filtragem a um objeto *Admin*, essa propriedade é passada também para os *proxies* criados por tal *Admin*. Nada impede que os objetos de níveis inferiores alterem individualmente suas propriedades para satisfazer necessidades específicas. No entanto isso cria um problema de inconsistência no caminho percorrido pelo evento pois dependendo da propriedade de QoS ela não poderá ser garantida pelo serviço. Por exemplo, quando um fornecedor envia um evento com a propriedade *EventReliability* =

1 e no caminho desse evento existe algum objeto (*Proxy, Admin* ou *Canal*) que não esteja configurado para trabalhar com eventos persistentes, no caso de uma falha do serviço, o mesmo não irá garantir a entrega desse evento quando retornar a atividade. A tabela 2 mostra as propriedades de QoS que podem ser usadas para os objetos Canal de eventos, Admins e Proxies e algumas combinações recomendadas.

3.3 Usando o serviço de Notificação de CORBA via Lua

```
1  if not LuaOrbCfg.getIFR() then
2    LuaOrbCfg.setIFR(readIOR("/home/nelio/orbevent/ir.ref"))
3  end
4
5  EventChannelFactory=lo_createproxy(readIOR("not.ref"),
           "CosNotifyChannelAdmin::EventChannelFactory")
6
7  QoSProperties = {{name = "Reliability",value = "Persistent"},
                    {name = "ConnectionReliability",value = "Persistent"},
                    {name = "OrderPolicy",value = "PriorityOrder"}}
8  AdminProperties = {}
9  EventChannel=EventChannelFactory:
                 create_channel(QoSProperties,AdminProperties,ChannelID)
10
11 SupplierAdmin=EventChannel:new_for_suppliers("OR_OP",AdminID)
12
13 SupplierAdmin:set_qos({{name = "OrderPolicy",value = "PriorityOrder"}})
14
15 ProxyConsumer=SupplierAdmin:
                  obtain_notification_push_consumer("SEQUENCE_EVENT",IDAdmin)
16
17 ProxyPushConsumer = lo_narrow(ProxyConsumer)
18
19 ProxyPushConsumer:set_qos({{name="OrderPolicy",value="PriorityOrder"}})
20 ProxyPushConsumer:connect_sequence_push_supplier(nil)
21
22 AvisoFalha = {
   header =
     {fixed_header =
         {event_type = {domain_name = "", type_name = "" },
          event_name = "" },
    variable_header = {{name = "EventReliability",value = 1},
                       {name = "Priority",value = 32767} } },
    filterable_data = {{name = "AdmRede",value = "redeA"}},
    remainder_of_body = "falha na redeA"
   }
23  AvisoDesligamento = {
    header =
    {fixed_header =
         {event_type = {domain_name = "", type_name = "" },
          event_name = "" },
     variable_header = {{name = "EventReliability",value = 1},
                        {name = "Priority",value = 32767}} },
     filterable_data = {{name = "AdmRede",value = "redeA"}},
     remainder_of_body = "Desligar Roteador"
    }

24  BatchEvent = {AvisoFalha, AvisoDesligamento}
25  ProxyPushConsumer:push_structured_events(BatchEvent)
```

Figura 5: Usando o serviço de notificação via Lua

A Figura 5 ilustra a utilização do serviço de notificação com especificação de propriedades de QoS. Este exemplo determina o envio de dois eventos definidos pelas variáveis *AvisoFalha* e *AvisoDesligamento,* empacotadas na variável *BatchEvent,* para

todos os consumidores que especificaram interesse em AdmRede="redeA". Estes eventos possuem duas propriedades de QoS: *Priority = 32767* e *EventReliability = 1*. O código da figura 5 foi escrito usando a linguagem Lua.

Inicialmente, da linha 1 a 3, LuaOrb é configurado para usar o Repositório de Interfaces (RI) de CORBA. Em seguida, na linha 5, é criado o *proxy* para a interface *EventChannelFactory,* que oferece a função *create_channel(),* usada com os parâmetros *QoSProperties* e *AdminProperties* para criar um novo canal com suporte a garantia de entrega de eventos e ordenação de eventos por prioridade. Com o novo canal definido, o método *new_for_suppliers()* é invocado e retorna um objeto *SupplierAdmin.* A partir desse objeto é possível atribuir características de QoS com o método *set_qos()* e ainda criar um *proxy Consumer* usando *obtain_notification_push_consumer().* Na seqüência, é utilizado o método *connect_sequence_push_supplier()* para conectar o fornecedor ao canal de eventos com suporte a envio de eventos seqüenciais (pacotes de eventos). Nas linhas 22 e 23 os eventos são definidos com propriedades de QoS *{name = "EventReliability", value = 1}* para garantia de entrega e *{name = "Priority",value = 32767}* para ordem de prioridade, além de possuírem um campo de filtragem com valor *{name = "AdmRede", value = " redeA "}*. Por fim, na linha 25, o método *push_structured_events()* envia o evento seqüencial, definido pela variável *BatchEvent,* para todos os consumidores interessados.

O código do exemplo mostrado na figura 5 caracteriza-se pelo elevado número de chamadas a métodos e pela definição excessiva de estruturas de dados complexas. Fica claro também, através das chamadas ao método set_qos(), a preocupação em se evitar os problemas de inconsistência das propriedades de QoS entre os níveis da aplicação.

4. Uma biblioteca sobre o serviço de Notificação

Como discutido na seção 3 e ilustrado no exemplo da figura 5, o serviço de notificação de CORBA é bastante flexível e satisfaz os requisitos de uma aplicação que utilize a comunicação assíncrona baseada em eventos. No entanto, sua utilização exige do programador da aplicação um profundo conhecimento das interfaces que compõem o serviço, da estrutura dos eventos, de como tais eventos são enviados e recebidos, de como os níveis são conectados e de como as inconsistências das propriedades de QoS podem afetar o objetivo final da aplicação. No sentido de diminuir as complexidades de utilização do serviço de notificação, especificamos e implementamos a biblioteca LOrbNotify, que visa facilitar a utilização do serviço de notificação e ainda minimizar os problemas relacionados as inconsistências de propriedades de QoS.

Na subseção 4.1 será apresentada a arquitetura de LuaSpace onde LOrbNotify está inserida. A subseção 4.2 comenta o funcionamento de LOrbNotify apresentando as funções que a biblioteca oferece. Na subseção 4.3 é discutido um estudo de caso que exemplifica o uso da biblioteca LOrbNotify.

4.1 Arquitetura

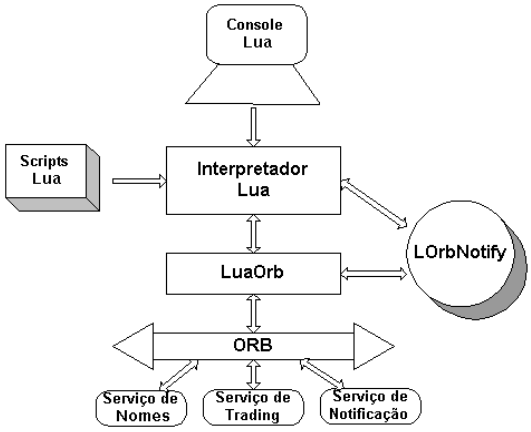

Figura 6: Arquitetura

A Figura 6 ilustra a parte da arquitetura do LuaSpace relevante para situar a biblioteca LOrbNotify. 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.

Quando o interpretador Lua executa uma chamada de funções da biblioteca LOrbNotify, ele invoca a implementação da função. As funções de LOrbNotify são escritas em Lua e realizam o mapeamento para as operações correspondentes do serviço de notificação de CORBA. Através de LuaOrb, LOrbNotify 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.

4.2 Operações

Tabela 3: Resumo das 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

Para facilitar a utilização do serviço de notificação a biblioteca LOrbNotify disponibiliza um conjunto de funções Lua descritas na tabela 3.

A função *new()* analisa os parâmetros recebidos e, a partir deles, inicia o serviço de notificação com canal tipado ou sem tipo e carrega no Repositório de Interfaces (RI) de CORBA as interfaces necessárias. 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 *supplier = LOrbNotify:new()*, estabelece o uso de um canal sem tipo. Em suma, a função *new* torna o serviço de notificação operacional e retorna a referência de um objeto que será usada na chamada a 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, LOrbNotify disponibiliza a função *sendevent*([assunto],any). Esta função permite o envio de eventos com suporte a todas as propriedades de QoS descritas nas tabelas 1 e 2. As propriedades de QoS juntamente com possíveis assuntos de interesse são estabelecidos 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 para definição do evento pois como o evento não tem um tipo específico pré-definido, ele pode ser representado por diferentes elementos inclusive por um conjunto de eventos. Por exemplo, a chamada *sendevent({AdmRede="redeA", Priority=32767, EventReliability=1 },"falha no roteador A")* envia o evento *"falha no roteador A"* (uma mensagem do tipo *strin*g) para os que registraram interesse no atributo *AdmRede* = "*redeA*" com garantia de entrega (propriedade *EventReliability* = 1) e alta prioridade (maior valor para a propriedade *Priority*). As propriedades de QoS definidas na função *sendevent* devem corresponder as mesmas presentes na tabela 1 e 2, caso contrário, a biblioteca LOrbNotify irá interpretar que não se trata de uma propriedade de QoS mas sim de um atributo de filtragem. Outro aspecto relacionado às propriedades de QoS é o problema da inconsistência no caminho do evento. No exemplo anterior, o evento possui *Priority=32767*, no entanto para que essa propriedade seja atendida é necessário que todos os elementos (Proxies, Admins e Canal) que compõem o caminho do evento possuam *OrderPolicy=PriorityOrder* pois, caso contrario, se algum desses elementos possuir um *OrderPolicy=FifoOrder*, o evento poderá entrar numa longa fila de espera, terminando por perder seu requisito de prioridade.

LOrbNotify minimiza esse problema com uma política de unificação das propriedades entre Fornecedores e Consumidores. Essa política é estabelecida quando um fornecedor envia um evento através da biblioteca LOrbNotify e seu evento requer atributos de QoS, como *EventReliability, Priority, StartTime* e *StopTime*, que exigem um caminho de eventos consistente. A biblioteca identifica essa necessidade e realiza duas ações antes de enviar os eventos: primeiro unifica as propriedades de QoS relativas ao lado da aplicação (Fornecedor Admin e Proxy Consumidor), em seguida envia um evento para localizar todos seus prováveis consumidores. Caso esses consumidores também usem a biblioteca, eles realizam as mesmas mudanças de propriedades de QoS nos seus objetos consumidores (Consumidor Admin e Proxy Fornecedor). A partir desse momento, o evento enviado pelo fornecedor poderá ser encaminhado com garantia das propriedades de QoS estabelecidas para o evento. Com este mecanismo, a biblioteca

LOrbNotify busca minimizar as prováveis inconsistências de propriedades de QoS no caminho do evento.

O método *sendevent* pode ser chamado ainda com um conjunto de eventos, como em *sendevent({AdmRede=“redeA”, Priority=32767},{“falha na redeA”, “Desligar Roteador”})* ou mesmo sem especificar assunto de interesse ou característica de QoS. Neste último caso apenas omite-se o parâmetro *assunto*, por exemplo, *sendevent(”falha na redeA”).* Uma grande flexibilidade oferecida pela biblioteca LOrbNotify para envio de eventos é permitir associar o evento à uma função. Por exemplo, na Figura 7 é definida uma função *function rotasdisp()* que retorna todas as rotas disponíveis. Na invocação *sendevent({AdmRede = “redeA”}, rotasdisp)* a biblioteca registra a função *rotasdisp*() para enviar eventos com assunto de interesse *AdmRede = “redeA”*.

```
1 function  rotasdisp()
2       rotas = {“200.147.15.21”,”200.234.19.230”,” 200.230.0.125”}
3       return rotas
4 end
5 supplier = LOrbNotify:new()
6 supplier:sendevent({AdmRede = “redeA”}, rotasdisp)
```

Figura 7 – Envio de Evento através de Função rotasdisp

Para a recepção de eventos, LOrbNotify oferece a função *getevent*([assunto],[any]). Esta função retorna um valor que é o evento recebido. Ou seja, o retorno pode ser um string, função ou uma tabela (caso o evento recebido seja um conjunto de eventos). Os eventos podem ser recebidos considerando-se assuntos de interesse ou não. Por exemplo, para estabelecer interesse no atributo AdmRede com valor *“RedeA*”, usa-se a função *getevent(“AdmRede==’redeA’ ”*). Para estabelecer vários assuntos de interesses, usa-se os operadores lógicos (AND, OR, NOT), por exemplo, *getevent(“AdmRede=’redeA’ or AdmRede=’redeB’ ”).* Pode-se também inserir nos atributos de interesse características de QoS, como: *getevent(“AdmRede == ’redeA’ and MaxEventsPerConsumer == 5 or AdmRede == ‘redeB’ ”).* Nesse caso a biblioteca, ao identificar *MaxEventsPerConsumer* como um atributo de QoS, remove-o avaliando como condições de filtragem apenas (“*AdmRede==’redeA’or AdmRede == ‘redeB’).* 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 8 é ilustrado um exemplo simples onde é definida a função *verificarErros(msg)* que muda a tabela de rotas (função *mudarota*) a medida que identifica-se um problema de operação na rede. A informação sobre o problema de operação é transmitida através de um evento com assunto de interesse *AdmRede = “redeA”*.

```
1 function verificaErros(msg)
2    if msg == “falha na redeA” then  mudarota(“redeB”) end
3 end
4 consumer = LOrbNotify:new()
5 consumer:getevent(“AdmRede == ‘redeA’ ”, verificaErros)
```

Figura 8 - Recebimento de Evento via função verificaErros

LOrbNotify 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 é automaticamente criado.

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 LOrbNotify, a aplicação ilustrada na Figura 5 pode ser implementada usando-se apenas dois comandos, como mostra a Figura 9.

```
1 Supplier = LOrbNotify:new()
2 sendevent({AdmRede=“redeA”, Priority=32767, EventReliability=1},{”falha na redeA”,” Desligar Roteador”})
```

Figura 9 - Uso do serviço de eventos via LOrbNotify

4.3. Estudo de Caso - um sistema de gerência de redes

Para ilustrar a utilização do LOrbNotify, será descrita a implementação de um sistema de gerência de redes simples inspirado no funcionamento do protocolo SNMP (Simple Network Management Protocol). O sistema é composto por dois elementos: *agentes* e *gerentes*. Os agentes são responsáveis pela captura dos dados em cada um dos recursos gerenciados (hosts, roteadores, impressoras) e pelo envio de eventos previamente programados. O gerente é responsável pelo monitoramento dos dados enviados pelos agentes além da geração de relatórios e tomada de decisões na ocorrência de problemas.

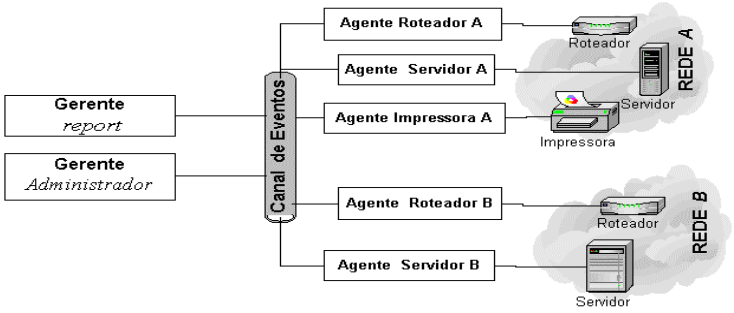

Figura 10 – Sistema de Gerência de Redes

Considerando duas redes simples, ilustrada na Figura 10, definimos como exemplo dois agentes e dois gerentes. O primeiro agente ilustrado na figura 11 é responsável por enviar dados para todos os gerentes que registraram interesse em *objetivo == 'report'*. Esse agente envia uma seqüência de eventos, onde o primeiro elemento indica a origem do evento("RoteadorA") e o segundo o número de datagramas que foram recebidos e descartados devido a erros no cabeçalho IP(ipInHdrErrors = 22541).

```
1 AgenteRoteadorA =  LOrbNotify:new()
2 AgenteRoteadorA:sendevent({objetivo = "report"},{"Roteador A", "ipInHdrErrors=22541"} )
```

Figura 11 – Código do agente roteador A

Para tratar os eventos enviados pelo *AgenteRoteadorA*, definimos através da figura 12 o *gerente report*. Esse gerente registra interesse em obter todos os eventos com *objetivo == 'report'*. Os eventos que chegarem serão tratados pela função *TratDados* que produz um relatório com estatísticas de datagramas recebidos e descartados.

```
1 function TratDados(msg)
2           ...
3  end
4 GerenteReport  =  LOrbNotify:new()
5 GerenteReport:getevent("objetivo == 'report' ",TratDados)
```

Figura 12 – Código do gerente report

O segundo agente ilustrado na figura 13 envia o evento *root@lcc.ufrn.br* com prioridade máxima(*Priority=32767*), garantia de entrega(*EventReliability=1*) e o atributo *tcpCurrentEstab* = 40, indicando um número de 40 conexões TCP que estão estabelecidas ou a espera de estabelecimento.

```
1 AgenteServidorB =  LOrbNotify:new()
2 AgenteServidorB:sendevent({Priority=32767,EventReliability=1, tcpCurrentEstab=40},"root@lcc.ufrn.br")
```

Figura 13 – Código do agente servidor B

```
1 function TratarExcessoConexoes (msg)
2       enviar_mail_to_root(msg)
3  end
4 GerenteAdm= LOrbNotify:new()
5 GerenteAdm:getevent("tcpCurrentEstab > 35",  TratarExcessoConexoes)
```

Figura 14 – Código do gerente administrador

Os eventos enviados pelo *Agente Servidor B* são tratados pelo *gerente Adm*, como mostra o código da figura 14. Neste código o gerente define que receberá todos os eventos com *tcpCurrentEstab > 35* e, para cada evento recebido, enviará através da função *TratarExcessoConexoes(msg)* um email para seu administrador.

5. Conclusões

Este trabalho apresentou o suporte de CORBA para comunicação assíncrona com Qualidade de Serviço (QoS), discutiu as complexidades do serviço de notificação de CORBA e apresentou uma biblioteca – LOrbNotify - que abstrai as complexidades de utilização do serviço. A biblioteca oferece um conjunto de funções que evita que o programador tenha de conhecer detalhes do serviço de notificação e que o código da aplicação possua diversas chamadas ao serviço que são alheias ao domínio da aplicação.

Além disso, LOrbNotify implementa uma política de unificação das propriedades de QoS entre Fornecedores e Consumidores que evita que incompatibilidade entre as propriedades no caminho do evento afetem a garantia de QoS exigida por uma das partes. Este aspecto não é contemplado pelo serviço de notificação.

Em termos de trabalhos correlatos, a maioria dos sistemas de notificação existentes [Cugola, Di Nitto and Fuggetta 2001, Orvalho, Andrade and Boavida 1999, PeerLogic 1998, Talarian 2001] implementam uma solução proprietária e poucos deles [Orvalho, Andrade and Boavida 1999, PeerLogic 1998] endereçam comunicação assíncrona em conjunto com QoS. Em contraste, LOrbNotify explora o suporte do padrão CORBA oferecendo uma API para facilitar o uso do serviço de notificação.

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 [Carzaniga, Di Nitto, Rosenblum and Wolf 1998]. Um outro aspecto de LOrbNotify 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.

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 notificação de CORBA, não se restringe simplesmente ao serviço de notificação mas também a qualquer outro serviço CORBA uma vez que a complexidade de usá-los é semelhante, com a ressalva de que o serviço de notificação envolve uma quantidade maior de objetos a serem habilitados. Em [Batista, Morais, Carvalho and Teixeira 2002] 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.

Referências Bibliográficas

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

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

Batista, T., Cacho, N. and Galvão, G. (2002). LOrbEvent: Uma Biblioteca para Viabilizar o Uso do Serviço de Eventos de CORBA. In Simpósio Brasileiro de Engenharia de Software (SBES 2002), Gramado – RS, Outubro.

Bernstein, P. "Middleware" (1996). Communications of the ACM, 39(2), February.

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

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

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

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

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

OMG (1999) CORBA Notification Service. Technical Report formal/99-07-01, OMG, August.

OMG. (1998) The Common Object Broker Architecture and Specification. Technical Report Revision 2.2, OMG, 1998.

Orvalho, J., Andrade, T. and Boavida, F. (1999) "Extension to Corba Event Service for a Conference Control System". Proceedings IEEE International Conference on Multimedia Computing and Systems ICMCS'99, Florence, Italy, June.

PeerLogic (1998) "DAIS Multicast Event Service", PerrLogic White Paper.

Talarian (2001) "Mission Critical Interprocess Communications – an Introduction to SmartSockets" – White Paper. Available at http://www.talarian.com/

Tari, Z. and Bukhres, O. (2001) Fundamentals of Distributed Object Systems – The CORBA perspective. John Wiley & Sons.