3
LuaProxy

Conforme explicado na seção 2.2, a função *luaorb.createproxy* cria um *proxy* convencional, responsável pela comunicação com o objeto determinado em seus argumentos. O *LuaProxy*, por sua vez, é uma alternativa mais sofisticada que visa prover alguns mecanismos básicos para a adaptação relativa às necessidades de QoS das aplicações.

Para tanto, faz-se necessário o uso de uma linguagem de especificação de QoS. Apesar de já existirem diversas propostas nesse sentido [11, 12], as características da linguagem interpretada Lua possibilitam uma alternativa mais simples para se especificarem os níveis desejados de QoS. Apresentamos na Seção 3.1 a SQDL (*Simple QoS Description Language*), que, mesmo não sendo muito expressiva, satisfaz os requisitos das especificações de QoS mais usuais.

Baseado em uma especificação SQDL, o LuaProxy pode detectar quando a QoS solicitada foi violada (usando monitores de evento), disparando, assim, o método de adaptação. O usuário pode, opcionalmente, personalizar o tratamento dos eventos de violação de QoS, conforme discutido na Seção 3.2.2. Caso isso não seja feito, o LuaProxy simplesmente busca um novo servidor compatível com a especificação SQDL em uso; a Seção 3.2.1 descreve os detalhes desse procedimento de seleção.

3.1
SQDL — Simple QoS Description Language

A descrição de requisitos de QoS pode ser facilmente codificada em tabelas Lua. Essas tabelas, chamadas *descritores de QoS*, são organizadas como instâncias da classe *QoSDescriptor*. Elas consistem em uma lista de *atributos* e

```
d3 = QoSDescriptor (d1, d2, "delay < 10")
```

Figura 3.1: Criação de um descritor de QoS

podem herdar de outras instâncias de maneira similar à encontrada em algumas linguagens orientadas a objeto com suporte a herança múltipla. Nesse contexto, quando um descritor herda de outros, estes se chamam *descritores-pais* ou *super-descritores*, enquanto aquele se denomina *descritor-filho* ou *descritor derivado*.

Para se criar um descritor de QoS, basta invocar o construtor *QoSDescriptor*; os argumentos são os descritores dos quais se deve herdar e os atributos do novo descritor. Esses atributos, por sua vez, são strings que definem restrições sobre os *parâmetros* de QoS desejados, como "delay $<$ 10", que diz que o retardo de uma transmissão não deve ultrapassar 10 ms. Finalmente, um atributo pode redefinir os dos super-descritores ou ser totalmente novo.

A título de exemplo, supondo que `d1` e `d2` sejam descritores válidos, pode-se criar um novo descritor `d3` que herde deles e defina a restrição de retardo do parágrafo anterior como na Figura 3.1.

3.1.1 Atributos

Formalmente, um atributo é uma string que define uma expressão relacional sobre um *parâmetro* de QoS. Essas expressões usam os operadores $<$ e $>$ para limitarem os valores que o parâmetro em questão pode assumir. Os parâmetros aceitos são discutidos na Seção 3.1.3.

Sejam `par` um parâmetro qualquer e `v` um possível valor para esse parâmetro. Um atributo deve, então, assumir uma das seguintes formas:

- `par > v`
- `par < v`

Assim, pode-se especificar uma restrição de retardo máximo de 20 ms com a string "delay $<$ 20".

Em condições ideais, quando o *LuaProxy* escolhe no *trader* uma oferta de serviço, esta satisfaz todos os atributos do descritor de QoS em uso. Quando

isso não é possível, torna-se necessário restringir a busca, considerando somente os atributos mais relevantes à aplicação.

De modo a expressar essa relevância, os atributos podem ter uma prioridade associada a eles. Para tanto, escreve-se o atributo normalmente, acrescentando-se uma vírgula e a prioridade desejada. Por exemplo, um atributo de alta prioridade que especifica um retardo de no máximo 10 ms é escrito como: "delay $<$ 10, high". As prioridades aceitas são "low", "normal" e "high". Caso a prioridade de um atributo não seja especificada, o descritor de QoS usa "normal" como *default*.

3.1.2 Mecanismo de Herança

Conforme mencionado na Seção 3.1, um descritor pode herdar de nenhum, um ou vários outros. Quando se tenta acessar um atributo inexistente em um descritor, o mecanismo de meta-tabelas de Lua chama automaticamente o meta-método *__index*, que busca o atributo em questão recursivamente na lista dos super-descritores.

Essa busca se dá na mesma ordem em que os super-descritores foram especificados no construtor. Como exemplo, sejam `d1` e `d2` dois descritores de QoS e `d3` um descritor que herda deles, como mostrado na Figura 3.1. Ao se buscar um atributo em `d3`, o mecanismo de herança consultará primeiro `d1`. Somente se o atributo não for encontrado em `d1` a busca prosseguirá para `d2`. Caso a ordem dos argumentos do construtor tivesse sido invertida, a ordem da busca pelo atributo também o teria sido.

Esse algoritmo simples de busca possui alguns efeitos colaterais indesejados. A título de ilustração, suponham-se `d1`, `d2` e `d3` definidos como na Figura 3.2. Ao se consultar o parâmetro *delay* em `d3`, o algoritmo de busca retornaria "delay $<$ 80, low", apesar de `d2` requerer, com alta prioridade, um retardo máximo de apenas 10 ms. Outra questão a ser pensada seria a de como avaliar a importância de um atributo caso ele seja definido em um descritor muito distante (i.e. em um nível mais alto) na hierarquia da herança. Finalmente, restaria o problema de como resolver conflitos entre atributos mutuamente exclusivos, como "delay $<$ 10" e "delay $>$ 10".

Em situações práticas, no entanto, os grafos de herança de descritores são pequenos, e seus atributos não conflitam entre si. Mesmo quando isso ocorre,

```
d1 = QoSDescriptor ("delay < 80, low")
d2 = QoSDescriptor ("delay < 10, high")
d3 = QoSDescriptor (d1, d2)
```

Figura 3.2: Efeito indesejado do algoritmo de busca.

normalmente um deles consiste em uma versão mais restritiva dos demais. O resultado da resolução do conflito poderia ser, então, o atributo mais restritivo. Nessas condições, a decisão de projeto escolhida foi a de manter a simplicidade da especificação de QoS, mesmo em renúncia de sua expressividade em casos mais complexos.

3.1.3
Parâmetros de QoS

Do ponto de vista dos descritores de QoS, seus parâmetros são simples strings, como “delay”, “bandwidth”, etc. Via de regra, baseado apenas nos seus atributos, um descritor de QoS não pode concluir, por exemplo, se uma oferta de serviço é mais vantajosa do que outra. Se uma delas oferecer um retardo maior em troca de mais largura de banda disponível, é necessário traduzir a importância de cada um desses parâmetros numericamente para que essas ofertas possam ser comparadas.

Para auxiliar os descritores de QoS em tarefas que dependam da interpretação de parâmetros de QoS, o usuário pode utilizar instâncias da classe *QoSParameters*. Elas contêm, para cada parâmetro de QoS, um campo com as informações sobre o parâmetro em questão.

Um dos métodos definidos nesses campos é *getMonitor*. Quando o LuaProxy importa uma oferta de serviço, ele precisa se registrar como observador nos monitores dos parâmetros de QoS de interesse. A referência para esses monitores, por *default*, é obtida da própria oferta de serviço. Se *par* é um parâmetro de QoS, então a referência para o monitor associado a ele se encontra na propriedade *par_mon* da oferta. Em alguns casos o programador pode precisar redefinir o método *getMonitor* de algum parâmetro de QoS. Por exemplo, não faria sentido obter da oferta de serviço o monitor de número de processos executando no lado do cliente, que seria um objeto local.

O método *getQualityExpression*, por sua vez, retorna uma string, uma

```
p3 = QoSParameters (p1, p2)
```

Figura 3.3: Herança de tabelas de parâmetros de QoS.

```
local delay_desc = {}
function delay_desc.getQualityExpression (self, prop, op, val)
   return "2000 / delay"
end

p3:addParameter ("delay", delay_desc)
```

Figura 3.4: Definindo uma expressão de qualidade para o retardo.

expressão[1] que avalia numericamente a qualidade do valor de uma propriedade (*expressão de qualidade*). Uma possível expressão seria "1000 / delay", que retorna valores altos para os menores retardos, que implicam melhor qualidade de serviço.

Apesar de o usuário já poder especificar quais parâmetros lhe são mais importantes com as prioridades "low", "normal" e "high" dos descritores de QoS, o método *getQualityExpression* permite fazer ajustes mais finos. Para isso, o usuário pode criar suas próprias tabelas de parâmetros de QoS. Estas podem, opcionalmente, herdar de outras tabelas, de acordo com as mesmas regras descritas na Seção 3.1.2. Normalmente esse procedimento não é necessário, pois a expressão de qualidade *default* é adequada à maioria das situações.

A título de exemplo, sejam `p1` e `p2` duas tabelas de parâmetros válidas. Pode-se criar uma nova tabela que herde dessas duas como na Figura 3.3 e, então, adicionar o parâmetro *delay* conforme mostrado na Figura 3.4.

Nota-se que os argumentos do método *getQualityExpression* não estão sendo usados na Figura 3.4. O argumento *prop* contém a propriedade sendo avaliada; *op* pode ser "<" ou ">" e *val* é o valor limite para a propriedade, conforme especificado no descritor de QoS. Assim, se o usuário solicitou um retardo menor que 50 ms (com um atributo "delay < 50" no descritor de QoS), o método *getQualityExpression* será chamado com os argumentos "*delay*", "<" e 50. O uso desses argumentos permite a reutilização da implementação do método *getQualityExpression* para outros parâmetros de QoS.

Especificamente, a implementação *default* (Figura 3.5) usa o argumento

[1]Como essas expressões são avaliadas pelo *trader*, é necessário que elas obedeçam às restrições impostas pela especificação da OMG para o serviço de *trading*.

```
function defaultQualityExpression (self, prop, op, val)
   if op == ">" then
      -- Para o caso de "prop > val"
      return self.Multiplier ..
         " * (" .. prop .. " - " .. val .. ")"
   else
      -- Para o caso de "prop < val"
      return self.Multiplier .. " / " .. prop
   end
end
```

Figura 3.5: Implementação da expressão de qualidade *default*.

```
local delay_desc = {Multiplier = 2000}
p3:addParameter ("delay", delay_desc)
```

Figura 3.6: Ajustando a expressão de qualidade *default* para o retardo.

op para decidir se a expressão de qualidade deve ser crescente ou decrescente. Se o descritor de QoS em uso requer que um parâmetro seja superior a um certo patamar, então o aumento do seu valor provavelmente torna a oferta mais atraente; logo a expressão *default* é uma função crescente desse parâmetro. Analogamente, quando o parâmetro é restrito a um dado limite, as melhores ofertas tendem a ser aquelas nas quais seu valor é menor. Nesse caso, a expressão *default* é função decrescente desse parâmetro.

Essa política de avaliação dos parâmetros possui um ponto de flexibilização. Como a Figura 3.5 mostra, as expressões *default* são multiplicadas pelo valor armazenado no campo *Multiplier* do parâmetro em questão. Assim, para ajustar a expressão de qualidade de um parâmetro de QoS específico, basta definir esse campo com o valor desejado. O exemplo da Figura 3.4 poderia ser reescrito simplesmente como na Figura 3.6.

3.2 Adaptação Automática

O funcionamento básico do *LuaProxy* consiste em redirecionar as chamadas de métodos para objetos que satisfaçam as propriedades estipuladas. Esses objetos são obtidos por uma pesquisa (*query*) ao serviço de *trading*, descrita na próxima seção.

Após selecionar o objeto que vai usar, o *LuaProxy* associa a ele um monitor que verifica, periodicamente, os atributos de QoS relevantes. Caso eles não mais satisfaçam os limites desejados, o monitor notifica esse evento ao observador correspondente, para que ele seja tratado. Esse procedimento é discutido na Seção 3.2.2.

3.2.1 Seleção de Servidores

Para encontrar servidores apropriados, o *LuaProxy* utiliza o serviço de *trading*, usando como critério de busca (*query constraint*) a conjunção dos atributos do descritor de QoS. Essa string é obtida automaticamente pelo método *getConstraint* do descritor de QoS. Ele recebe uma prioridade como argumento ("low", "normal" ou "high") e considera somente os atributos que possuam prioridade maior ou igual à que foi especificada.

Isso é muito útil para o caso de a busca no *trader* não retornar resultado algum. Nesse caso, o critério de busca é redefinido, levando-se em conta somente atributos mais prioritários. Isso torna a busca menos restritiva, o que proporciona uma chance maior de a nova consulta retornar pelo menos uma oferta de serviço, e que atenda pelo menos aos atributos mais urgentes.

Independentemente do critério de busca, as ofertas são ordenadas de modo que as primeiras sejam as mais adequadas aos atributos do descritor em uso. Isso é feito usando o parâmetro de preferência de ordenação do método *query* do *trader*. Especificamente, a preferência consiste em maximizar a *qualidade* das ofertas, definida como o somatório ponderado das qualidades de todas as propriedades, ou seja:

$$Q = \sum_{i=1}^{\#props} (p_i \times q_i), \tag{3-1}$$

onde Q é a qualidade da oferta em questão, p_i é o peso da propriedade i e q_i é a expressão que avalia sua qualidade. O peso de cada propriedade depende da prioridade do seu atributo no descritor de QoS: 1 para as prioridades baixas (*low*), 2 para as normais e 4 para as altas (*high*).

A construção dessa string de preferência pode ser ilustrada com o descritor da Figura 3.7. Sejam os parâmetros *delay* e *bandwidth* definidos de

```
d = QoSDescriptor ("delay < 80, low"; "bandwidth > 50, high")
```

Figura 3.7: Descritor para exemplo de string de preferência.

modo tal que as expressões de qualidade (Seção 3.1.3) sejam "1000 / delay" e "bandwidth - 50". Assim, pela equação 3-1, a qualidade de uma oferta de serviço será:

$$Q = (1 \times \frac{1000}{delay}) + (4 \times (bandwidth - 50)) \quad (3\text{-}2)$$

Essa expressão será, então, usada pelo *trader* para ordenar as ofertas encontradas em ordem decrescente de qualidade. A melhor oferta será a primeira a ser retornada pelo *trader* e, consequentemente, selecionada.

3.2.2 Tratamento de Eventos

Quando o *LuaProxy* é notificado de um evento de violação de QoS, seu comportamento é determinado pela *implementação de QoS*. Trata-se de uma tabela onde os índices são os eventos que a tabela trata e os valores associados são tabelas que definem o comportamento a ser adotado.

Atualmente, o único campo que cada evento pode definir é o método *adapt*. Esse método recebe como argumento o proxy inteligente sendo adaptado; o valor de retorno indica se a adaptação teve sucesso ou não. Caso um evento não tenha um método de adaptação associado a ele ou o procedimento de adaptação falhe, a implementação de QoS retornará *false*, fazendo o LuaProxy selecionar um novo servidor.

Por outro lado, se a tentativa de adaptação for bem-sucedida, pode ser necessário alterar os atributos do descritor de QoS vigente (relaxamento dos atributos), de modo a refletir as novas necessidades da aplicação em seu estado pós-adaptação. Se isso não for feito, o LuaProxy será notificado mais de uma vez a respeito de um evento que já foi tratado. Analogamente, convém instalar um observador para detectar o retorno dos parâmetros de QoS aos seus níveis anteriores. Nesse momento, procedimentos como redução da densidade de cores ou taxa de quadros por segundo devem ser suspensos, sob pena de o LuaProxy continuar em seu estado adaptado, deixando de usufruir dos recursos disponíveis.

O método *adapt* admite dois valores de retorno adicionais (ambos opcionais) em caso de a adaptação ser bem-sucedida: o novo limite do parâmetro de QoS no estado pós-adaptação e o valor onde o sistema pode ser considerado recuperado. Caso esses valores sejam especificados, o LuaProxy monitora sua violação automaticamente, como demonstrado no Capítulo 5.

Com o objetivo de tornar o mecanismo de adaptação mais flexível, este trabalho suporta a criação de implementações de QoS personalizadas, através do construtor *QoSImplementation*. Essas estruturas podem herdar de outras de maneira análoga à que foi descrita na Seção 3.1.2, e podem ter novos eventos incluídos pelo método *addEvent*, que recebe como argumentos o nome do evento e uma tabela que contenha uma função *adapt*.

Implementações de QoS permitem tirar proveito de características específicas de alguns serviços. Por exemplo, se a aplicação for de vídeo sob demanda, o cliente pode usar uma implementação de QoS, possivelmente projetada pelo próprio fornecedor do serviço, capaz de tratar o evento *bandwidth_decrease* pela redução da taxa de quadros por segundo ou da densidade de cores. Situações mais complexas podem ser abordadas pelo uso de implementações que herdam de instâncias especializadas em lidar com cada propriedade separadamente.

3.3 Usando o LuaProxy

Um dos principais objetivos do LuaProxy é a facilidade de uso. De fato, seu construtor tem apenas um argumento obrigatório, o primeiro, que especifica o tipo de serviço a ser pesquisado no *trader*. Os demais são instâncias de descritores, implementações e parâmetros de QoS, ou ainda strings, que são interpretadas como atributos de descritores de QoS. O LuaProxy então cria um descritor que herda de todos os que foram especificados, e adiciona todas as strings fornecidas como atributos. Procedimento análogo é feito para os parâmetros e implementações de QoS. A Figura 3.8 mostra um exemplo de chamada de construtor com todos os parâmetros possíveis.

Caso nenhum argumento especifique descritores, implementações ou parâmetros, o LuaProxy usará instâncias globais desses objetos, que por *default* não executam tarefa alguma. Especificamente, a falta de descritores de QoS (e strings com atributos) inibe o monitoramento de todas as propriedades

```
-- Descritor de QoS
desc = QoSDescriptor ("bandwidth > 70", "delay < 10")

-- Tabela de parametros
params = QoSParameters ()
delay = {Multiplier = 5000}
params:addParameter ("delay", delay)

-- Define funcao de adaptacao especifica
impl = QoSImplementation ()
bw_dec = {}
function bw_dec.adapt = function (luaprx)
   ...
end
impl:addEvent ("bandwidth_decrease", bw_dec)

p = LuaProxy ("Multimedia::Video", desc, params, impl)
```

Figura 3.8: Exemplo de chamada de construtor do LuaProxy.

das ofertas de serviço; nesse caso somente uma falha do servidor (detectada pela pseudo-operação CORBA *_non_existent*) provocaria a adaptação, que consistiria, necessariamente, na troca do servidor. Essa também seria a única adaptação possível caso nenhuma implementação de QoS seja usada. Finalmente, a ausência de uma tabela de parâmetros de QoS faria o LuaProxy usar expressões de qualidade *defaults*, descritas na Seção 3.1.3.

3.4 Observações

Conforme discutido nesse capítulo, o LuaProxy e seus descritores são criados e manipulados com facilidade. Acreditamos que seu comportamento *default* é aceitável em várias aplicações, o que permite ao programador inexperiente tirar proveito dessa arquitetura de imediato. À medida que o usuário se tornar mais familiarizado, ele poderá usar rotinas de adaptação fornecidas por terceiros; os mais avançados também teriam a opção de desenvolver suas próprias implementações de QoS, ajustar ou redefinir as expressões de qualidade, etc.

Desenvolvedores de produtos para usuários finais poderiam explorar as opções avançadas do LuaProxy à vontade, protegendo seus clientes dos detalhes

de baixo nível com uma interface gráfica apropriada. Essa interface traduziria argumentos complicados em menus com opções de alto nível, como “qualidade alta”, “qualidade econômica”, etc. Em ambientes desse tipo, as opções mais avançadas da interface gráfica ficariam agrupadas em uma seção própria, de modo a não confundir usuários leigos.

Em aplicações mais complexas, pode ser necessário criar descritores e implementações de QoS mais elaborados, o que requer mais trabalho. Contudo, o reuso dessas estruturas por meio da herança tende a reduzir esse fardo a longo prazo. Para se criarem os descritores e implementações para uma nova aplicação, bastaria ao cliente herdar dos existentes e fazer algumas modificações.