2
Ferramentas Utilizadas

Esta dissertação utiliza vários outros trabalhos para implementar os mecanismos de adaptação abordados. Essas ferramentas são descritas nas seções seguintes.

2.1
Lua

Lua [7, 8] é uma linguagem interpretada desenvolvida na PUC-Rio. Seus principais focos são o da simplicidade e o da portabilidade; apesar disso, sua eficiência é muito boa se comparada à de outras linguagens similares.

Uma característica da linguagem consiste em as variáveis não possuirem tipos, apesar de os valores que elas assumem serem tipados dinamicamente em números, strings, tabelas, funções (em Lua, funções são valores de primeira ordem, o que permite ao programador passar funções como argumentos e valores de retorno de outras funções), etc. Dessa forma, uma mesma variável pode assumir, em momentos diferentes, valores de tipos diferentes.

Dentre esses tipos, a *tabela* é de especial importância. Ela implementa um vetor associativo, ou seja, uma estrutura de dados que armazena valores quaisquer (também chamados de *campos* da tabela) indexados por chaves de tipo arbitrário (exceto nil). Assim sendo, uma mesma tabela pode conter uma função indexada por uma string e uma outra tabela indexada por um número.

A biblioteca-padrão de Lua define algumas funções que manipulam tabelas e facilitam a implementação de filas, pilhas, *heaps*, etc. Além disso, tabelas podem ter *meta-tabelas* associadas a elas, o que permite a redefinição dos seus comportamentos em determinados eventos.

Para tanto, as *meta-tabelas* definem certos campos chamados *meta-métodos*. Eles contêm as funções a serem chamadas quando o evento correspondente ocorrer. Por exemplo, quando se tenta acessar um campo inexistente em uma tabela, o meta-método *__index* é chamado, recebendo como argumentos a tabela em questão e o campo procurado.

Uma possível aplicação desse meta-método é a emulação do mecanismo de herança encontrado nas linguagens de programação orientadas a objeto: escreve-se uma função *__index* que busque o campo solicitado na tabela da qual se está herdando. Naturalmente, caso esta também não possua o campo em questão, o meta-método *__index* dessa tabela será chamado, e assim sucessivamente.

Outra utilidade consiste em retornar valores *default* para campos inexistentes. Por exemplo, uma implementação de matriz esparsa pode retornar zero em vez de nil para campos que não foram inicializados explicitamente. Como último exemplo, o LuaOrb (Seção 2.2) usa esse meta-método para interceptar o acesso aos campos de um objeto remoto, de modo a permitir sua utilização como se fosse uma tabela local.

2.2 LuaOrb

O LuaOrb [9] é um *binding* de CORBA em Lua, oferecendo ao usuário o poder de CORBA com a simplicidade de Lua. Em sua versão atual (3.0), o LuaOrb não é um ORB por si só: ele foi projetado para rodar sobre outro ORB, como um intermediário. Nessa dissertação foi utilizado o MICO [10] versão 2.3.8, um ORB gratuito, de código aberto, implementado em C++.

Com o LuaOrb, a tarefa de escrever um *servant* (objetos que implementam uma interface IDL) se reduz a definir uma tabela que contenha os campos correspondentes aos atributos e métodos da interface sendo usada. Em outras palavras, observadas as regras de mapeamento CORBA-Lua, para cada atributo da interface em IDL deve haver um campo com o mesmo nome e tipo correspondente na tabela que representa o servant; analogamente, ela também deve conter um campo para cada método da interface, com o mesmo nome e contendo uma função com assinatura compatível.

Servants normalmente são criados pela função *luaorb.createservant*, cujos argumentos são a tabela que implementa o servant e a sua interface. Esse

procedimento também pode ser feito implicitamente, quando um método retorna um valor ou recebe um argumento que não é um servant, mas uma tabela comum. Nesse caso, o LuaOrb tenta criar o servant baseado nessa tabela e na interface esperada para ele. Essa informação é obtida do repositório de interfaces em tempo de execução, por reflexividade.

Do lado cliente, para se criar um *proxy*, usa-se a função *luaorb.createproxy*, que recebe como argumentos uma referência para o servant (tipicamente uma string com seu IOR) e o nome da interface sendo usada.

Um proxy nada mais é do que uma tabela que usa o meta-método *__index* (descrito na Seção 2.1) para abstrair a comunicação com o objeto remoto. Especificamente, quando se acessa um campo ou método em um proxy, o meta-método busca esse dado no objeto remoto, retornando-o ao cliente de maneira transparente.

2.3 LuaTrading

Dentre os serviços CORBA existentes, o de *trading* desempenha papel importante neste trabalho, pois permite a um cliente encontrar um servidor adequado às suas necessidades. Para tanto, esse cliente solicita ao trader uma lista de *ofertas de serviço* que lhe interessem. Esse procedimento se chama *importação*.

Uma oferta de serviço, por sua vez, consiste em uma referência para o servidor que a implementa, bem como um conjunto de *propriedades* que descrevem as características desse servidor. Tais propriedades podem especificar, por exemplo, a quantidade de recursos computacionais disponíveis (como CPU, espaço em disco, etc.), de modo que um cliente possa optar por servidores menos carregados.

Para especificar as ofertas que lhe interessam, o cliente usa uma string que descreve as restrições sobre as propriedades em questão. Por exemplo, se um cliente quer importar ofertas cuja propriedade *custo* seja inferior a 30, ele passaria a string "custo $<$ 30" como argumento do método *query*, responsável pela importação.

Em contrapartida, pelo procedimento de *exportação*, um servidor pode anunciar suas ofertas de serviço ao *trader*, passando a lista de propriedades

```
ServiceTypesRep.servicetypes ["NovoTipo"] = {
   interface = "Demo::Pessoa",
   props = {nome = "string", idade = "short"}
}
```

Figura 2.1: Criando um tipo de serviço com o LuaTrading.

```
traderrep:add_type ("NovoTipo", "Demo::Pessoa", {
   {
      name = "nome",
      value_type = luaorb.createtypecode ("tk_string"),
      mode = "PROP_NORMAL"
   },
   {
      name = "idade",
      value_type = luaorb.createtypecode ("tk_short"),
      mode = "PROP_NORMAL"
   }
}, {})
```

Figura 2.2: Criando um tipo de serviço sem o LuaTrading.

como argumento. Todas as ofertas de serviço pertencem a um determinado *tipo de serviço*. A descrição de um tipo de serviço define se suas propriedades são *somente-leitura*, *opcionais*, etc. e os tipos das mesmas (*string, long, float*, etc.). Assim, antes de exportar uma oferta, é necessário criar o tipo de serviço correspondente caso ele ainda não exista.

Apesar de aqui expostas de maneira simplificada, as tarefas de importar e exportar ofertas de serviço envolvem o uso de vários métodos e estruturas de dados. O *LuaTrading* [5] é uma camada de abstração desenvolvida em Lua que facilita consideravelmente o acesso ao serviço de *trading*.

Uma dessas facilidades consiste em reduzir o fardo da criação de tipos de serviço a simples atribuições. Especificamente, para definir um tipo de serviço, basta escrever na tabela global *ServiceTypesRep* um campo contendo a descrição do tipo a ser criado; o nome do campo precisa ser igual ao nome do tipo a ser criado. As Figuras 2.1 e 2.2 ilustram esse procedimento, com e sem o uso do LuaTrading, respectivamente.

Nesse momento, um meta-método intercepta a operação e se encarrega de repassar a requisição ao *trader*. Conseqüentemente, a criação do tipo de serviço é feita de maneira totalmente transparente. A funcionalidade correspondente para exportar *ofertas de serviço* é oferecida pela tabela global

ServiceOffersRep.

Finalmente, para simplificar a importação de ofertas, o LuaTrading define a função *importServiceOffers*, que recebe seus argumentos por meio de uma tabela. Caso essa tabela não defina algum argumento, o LuaTrading usará um *default* aceitável para a maioria das situações.

2.4 LuaMonitor

O LuaMonitor [6] provê um conjunto de ferramentas para monitorar variáveis de interesse do programador. Seu principal componente é o *monitor de eventos* (*EventMonitor*), que observa uma variável em intervalos de tempo regulares. Para se criar um, chama-se a função *EventMonitor.new*, passando-se como argumentos o período com que a variável será monitorada, uma função para obter seu valor (*get*) e, opcionalmente, outra para modificá-lo (*set*).

O resultado é um monitor de eventos parametrizado de acordo com a necessidade, pois cada monitor tem períodos de verificação independentes. Assim, variáveis que mudam raramente podem ser monitoradas com um período maior, enquanto as que mudam constantemente podem precisar de períodos mais curtos.

Os monitores de evento também permitem a definição de *aspectos*, que são valores auxiliares derivados da variável sendo monitorada. Exemplos típicos de aspectos são funções estatísticas como média e variância. Os aspectos de um monitor também são atualizados periodicamente.

Talvez a maior utilidade de um monitor de eventos seja o de suportar *observadores de eventos*. São objetos interessados em condições específicas sobre a variável sendo observada e/ou seus aspectos. Caso tais condições sejam satisfeitas, a função que as avalia sinaliza isso ao monitor, que por sua vez chama a função *notifyEvent* do observador.

Por exemplo, suponhamos que um monitor esteja associado à saída de um termômetro digital. Se uma aplicação estiver interessada em ser notificada sobre um aumento excessivo de temperatura, ela pode registrar um observador no monitor, juntamente com uma função que verifica se a temperatura ultrapassou o limite desejado. No momento em que isso acontecer, o observador será notificado e poderá, por exemplo, acionar um mecanismo de resfriamento.

Alternativamente, a aplicação descrita acima poderia monitorar a temperatura por si mesma. Uma possibilidade seria criar uma *thread* para ler a temperatura continuamente, e reagir de acordo com o valor obtido. Além de tornar a aplicação mais complexa, essa proposta consumiria recursos da rede com as mensagens de verificação periódica.

Em contrapartida, a arquitetura do LuaMonitor testa as condições dos observadores localmente ao monitor, o que simplifica o cliente e não requer mensagens adicionais pela rede. Isso é possível graças à natureza interpretada da linguagem Lua, pois o código que testa as condições é passado para o monitor na forma de uma string.

Esta dissertação aplica esse mecanismo na implementação de um *proxy inteligente*, descrito no Capítulo 3. Especificamente, esse *proxy* monitora o valor de uma lista de parâmetros de QoS e, quando esses parâmetros extrapolam certos limites, o proxy é notificado para se adaptar em resposta a tais mudanças.