PUC-Rio - Certificação Digital Nº 0510971/CA

# 3
# OiL - ORB in Lua

O estudo feito neste trabalho se baseia na implementação do OiL, um middleware de distribuição escrito na linguagem dinâmica Lua e baseado no modelo de ORB. Na PUC-Rio, Lua foi utilizada na implementação de diferentes ferramentas de programação distribuída (Ururahy et al., 2002; Moura et al., 2002; Maia et al., 2004). Uma dessas ferramentas foi o LuaOrb (Cerqueira et al., 1999), um ORB para Lua cuja implementação se baseia em um ORB CORBA em C++. O LuaOrb permitiu obter resultados importantes, como a utilização das características de Lua para definir um mapeamento simples e flexível de CORBA para Lua. Entretanto, a utilização de um ORB em C++ impedia a avaliação das características de Lua na própria implementação do middleware, em particular, na configuração dos recursos oferecidos. Ao contrário do LuaOrb, o OiL foi projetado para ser inteiramente escrito em Lua e tirar proveito das características dessa linguagem em sua arquitetura e implementação interna.

Os dois objetivos originais do projeto do OiL eram investigar (a) como uma linguagem dinâmica como Lua pode contribuir para que o middleware apresente um modelo de programação e uma implementação que sejam flexíveis e fáceis de usar; e (b) técnicas que permitam reduzir o impacto que o uso dessa linguagem pode trazer ao desempenho do middleware. Esses objetivos foram decisivos na organização da implementação do OiL em componentes de software, pois através de diferentes montagens de componentes é possível modificar a implementação mais facilmente, como é apresentado na seção 3.2. Neste capítulo, é apresentada uma visão geral dos principais recursos oferecidos pelo OiL (seção 3.1), assim como a implementação dos seus principais componentes e suas diferentes montagens, divididas em três partes: camada de apresentação (seção 3.3), camada do protocolo LuDO (seção 3.4) e camada do protocolo CORBA (seção 3.5). A arquitetura de componentes do OiL é uma característica chave da sua flexibilidade, como será discutido em mais detalhes no capítulo 4. Ao final deste capítulo, também são apresentados alguns dos problemas encontrados no uso do OiL em diferentes aplicações, o que motivou a necessidade de maior flexibilidade da implementação para acomodar

diferentes configurações (seção 3.6).

## 3.1 Visão Geral

Uma das dificuldades do projeto de um middleware é encontrar uma forma em que ele possa se ajustar a uma aplicação específica de forma a facilitar ao máximo sua utilização. No caso de ORBs, isso se reflete na adequação do middleware às interfaces e tipos de dados manipulados pelos objetos distribuídos. Por exemplo, considere um objeto local que é registrado no ORB como um *servant*, ou seja, um objeto que pode ser acessado remotamente por outros objetos ou aplicações. O ORB tem que conhecer a interface do *servant* para poder despachar as chamadas remotas destinadas a ele, assim como os tipos de dados que são passados nas chamadas dessas operações. Um problema similar acontece com a criação de *proxies*, que são objetos locais que representam os objetos remotos. Os *proxies* são criados pelo ORB e devem oferecer as mesmas interfaces dos *servants* que representam.

Idealmente, o middleware deve se adaptar para acomodar objetos de diferentes interfaces de forma automática. A maior parte dos sistemas de middleware para objetos distribuídos, em particular os baseados no padrão CORBA, fazem isso gerando automaticamente uma porção de código que deve ser compilado junto com o código da aplicação. Isso impõe um processo de desenvolvimento que consiste em dois passos de compilação: (a) a geração automática do código que implementa parte do modelo de programação do middleware especificamente para a aplicação, o que é geralmente feito com base em descrições de interface fornecidas pela aplicação, e (b) compilação da aplicação juntamente com o código gerado na etapa anterior.

O OiL utiliza uma abordagem diferente. Ao invés de gerar código, ele utiliza os recursos de reflexão computacional de Lua para inspecionar e manipular os objetos da aplicação, assim como criar objetos que implementam as mesmas interfaces. Por exemplo, a figura 3.1(a) ilustra o código de um programa que registra um novo *servant* (linha 9) em um ORB que utiliza o protocolo LuDO (linha 2), de forma que ele possa ser acessado como um objeto distribuído. Após o registro do novo *servant*, uma representação textual de objeto é exibida na tela (linha 10). Essa informação é utilizada posteriormente para criar um *proxy* para esse objeto em uma aplicação remota. Por fim, a operação `run` do *broker* é executada e passa a receber e processar chamadas oriundas de clientes remotos. Quando uma chamada é recebida, o ORB inspeciona se o objeto tem aquela operação e a executa com os parâmetros fornecidos.

PUC-Rio - Certificação Digital Nº 0510971/CA

Algo similar acontece no programa da figura 3.1(b), onde um objeto remoto é acessado através de um *proxy* criado a partir de um *broker* (linha 5). O *proxy* é criado a partir da representação textual do objeto que é fornecida pela entrada padrão (linha 4). Nesse caso, o *proxy* criado é um objeto genérico criado pelo *broker*, que permite a chamada de qualquer operação. Quando uma operação é chamada no *proxy* (linha 6), o ORB envia a requisição dessa operação pela rede e aguarda pela resposta.

Figura 3.1: Criação de *servant* e acesso com *proxy*.

3.1(a): Programa servidor.

```
1 local broker = oil.init{
2   flavor = "ludo",
3 }
4 local hello = {}
5 function hello:say()
6   print("Hello, World!")
7 end
8 local servant =
9          broker:newservant(hello)
10 print(servant)
11 broker:run()
```

3.1(b): Programa cliente.

```
1 local broker = oil.init{
2   flavor = "ludo",
3 }
4 local ref = io.read()
5 local hello = broker:newproxy(ref)
6 hello:say()
```

O OiL permite interoperabilidade com ORBs CORBA através do protocolo IIOP (Object Management Group, 2002). No caso de CORBA, o uso do OiL é ligeiramente diferente. Para ilustrar isso, considere o código da figura 3.2(a). Uma aplicação CORBA necessariamente define um conjunto de interfaces, descritas na linguagem IDL (*Interface Definition Language*) de CORBA. Todo objeto distribuído da aplicação implementa uma dessas interfaces. Por isso, quando o ORB utiliza o protocolo IIOP de CORBA (linha 2), os *servants* devem ser registrados definindo a interface que eles implementam. Isso pode ser feito através do campo `__type` na implementação do objeto (linha 7). A operação `loadidl` do *broker* é usada para carregar informações de interface descritas em uma linguagem de descrição de interfaces, como IDL de CORBA (linha 4).

O acesso a objetos CORBA é feito de forma idêntica ao da figura 3.1(b), pois o ORB com suporte CORBA é capaz de importar a definição de interface do servidor, se ele oferecer esse suporte para isso através de um repositório de interfaces CORBA. Se isso não for possível, a interface do objeto remoto deve ser cadastrada no ORB, por exemplo, através da carga de uma IDL pela operação `loadidl`, como ilustrado na figura 3.2(b).

PUC-Rio - Certificação Digital Nº 0510971/CA

Figura 3.2: Criação de *servant* CORBA e acesso com *proxy* tipado.

3.2(a): Programa servidor.

```lua
local broker = oil.init{
  flavor = "corba",
}
local iface = broker:loadidl([[
  interface Hello { void say(); };
]])
local hello = { __type = iface }
function hello:say()
  print("Hello, World!")
end
local servant =
          broker:newservant(hello)
print(servant)
broker:run()
```

3.2(b): Programa cliente.

```lua
local broker = oil.init{
  flavor = "corba",
}
broker:loadidl([[
  interface Hello { void say(); };
]])
local ref = io.read()
local hello = broker:newproxy(ref)
hello:say()
```

## 3.2 Montagens do OiL

O OiL pode ser visto como um arcabouço de componentes de software que podem ser montados para composição de um middleware. Diferentes montagens desses componentes podem ser feitas para compor implementações de middleware com recursos e características distintas. Por exemplo, é possível montar ORBs que utilizem diferentes protocolos de RMI. Um protocolo de RMI define a forma como o ORB interage através da rede para realizar as chamadas em objetos remotos. O OiL apresentado neste trabalho oferece suporte a dois protocolos de RMI: o IIOP, definido pelo padrão CORBA, e o LuDO, que é um protocolo desenvolvido especificamente neste trabalho, como é visto na seção 3.4.

Os componentes do OiL podem ser vistos como unidades de composição de software desenvolvidas e utilizadas de forma independente, onde cada uma oferece um conjunto de serviços e define suas dependências através de interfaces bem definidas (Szyperski, 1998). A rigor, o OiL não impõe uma arquitetura específica para montagem de seus componentes ou um modelo de programação específico a ser oferecido pelo middleware. Entretanto, as implementações de componentes fornecidas atualmente pelo OiL assumem um modelo arquitetural geral, que pode ser conceitualmente dividido em duas camadas. Na camada de apresentação, encontram-se componentes que implementam o modelo de programação oferecido pelo middleware, como o suporte a *servants* e *proxies*. Na camada do protocolo, encontram-se componentes que implementam o protocolo de comunicação utilizado pelo middleware, como por exemplo o suporte ao protocolo LuDO.

Para facilitar a montagem e manipulação dos ORBs, o OiL oferece o módulo `oil` com funções para montagem de ORBs e uma interface simples para a manipulação desses ORBs. A manipulação dos ORBs é feita através dos

PUC-Rio - Certificação Digital Nº 0510971/CA

objetos *broker*, que definem uma fachada para acesso aos serviços oferecidos pelos componentes que formam o ORB (padrão *Façade* (Gamma et al., 1994)). ORBs são montados através da operação `oil.init([config])`, que devolve um *broker* contendo todos os componentes que formam o ORB. O parâmetro `config` é opcional e indica a tabela de configurações de criação do ORB. Os componentes a serem utilizados na montagem do ORB podem ser fornecidos através dessa tabela de configurações.

Alternativamente, pode-se utilizar uma das montagens de ORB pré-definidas no OiL, denominadas *flavors*. O campo `flavor` da tabela de configurações é utilizado para indicar o *flavor* do ORB a ser montado. Um *flavor* é uma seqüência de nomes separados por ponto-e-vírgula (`;`), onde cada um designa uma *montagem*. Cada montagem estabelece conexões entre componentes de uma parte do ORB. O trecho da figura 3.3 ilustra uma montagem da camada de apresentação do OiL. O OiL define um conjunto de montagens pré-definidas, que podem ser combinadas para formar os *flavors*. Através dos *flavors* é possível construir ORBs com diferentes funcionalidades, conforme será apresentado no restante deste capítulo.

Figura 3.3: Especificação da arquitetura básica de ORBs OiL.

```
1  local component = require "loop.component.base"
2  local port      = require "loop.component.port"
3  local arch      = require "oil.arch" -- funcoes auxiliares
4
5  module "oil.arch.base"
6
7  ProxyManager = component.Template{
8    proxies   = port.Facet,
9    requester = port.Receptacle,
10   referrer  = port.Receptacle,
11   servants  = port.Receptacle}
12 ServantManager = component.Template{
13   servants   = port.Facet,
14   dispatcher = port.Facet,
15   referrer   = port.Receptacle}
16 RequestReceiver = component.Template{
17   acceptor   = port.Facet,
18   dispatcher = port.Receptacle,
19   listener   = port.Receptacle}
20 BasicSystem = component.Template{
21   sockets    = port.Facet}
22
23 function assemble(orb)
24   orb.ProxyManager.requester      = orb.OperationRequester.requests
25   orb.ProxyManager.referrer       = orb.ObjectReferrer.references
26   orb.ProxyManager.servants       = orb.ServantManager.servants
27   orb.ServantManager.referrer     = orb.ObjectReferrer.references
28   orb.RequestReceiver.dispatcher  = orb.ServantManager.dispatcher
29   orb.RequestReceiver.listener    = orb.RequestListener.requests
30 end
```

Uma abordagem comum para o desenvolvimento baseado em componentes em linguagens orientadas a objetos é a aplicação de padrões de projeto que permitam expor suas dependências através de atributos públicos (Fowler,

PUC-Rio - Certificação Digital Nº 0510971/CA

2008). Em Lua, adotamos uma abordagem similar, onde um componente é uma tabela com campos que definem seus pontos de conexão, denominados *portas*. As portas podem ser *facetas*, onde o campo contém o objeto interno que implementa um serviço oferecido pelo componente; ou *receptáculos*, quando o campo é utilizado para armazenar um objeto externo que oferece um serviço utilizado pelo componente. Nessa abordagem, o componente é visto como um conglomerado de objetos intimamente relacionados que cooperam para oferecer um conjunto de serviços. Um exemplo disso é o componente `BasicSystem` do OiL, que é responsável por oferecer acesso às funcionalidades do software básico da plataforma, como *threads* e *sockets*. Tipicamente, esse componente é implementado por um escalonador de co-rotinas, que oferece suporte a *threads* cooperativas, e outro objeto que implementa uma API de *sockets*. Cada um desses objetos oferece seus serviços através de facetas diferentes, que podem ser conectadas a componentes distintos.

No desenvolvimento do OiL, a biblioteca LOOP (*Lua Object-Oriented Programming*)[1] foi criada para oferecer funções auxiliares para criação de componentes seguindo o modelo de componentes utilizado no OiL. Para tanto, o LOOP define o conceito de molde de componente ou *template*, que são objetos usados para auxiliar a criação de componentes. Contudo, o *template* impõe uma estrutura específica de implementação de componentes. Esse modelo visa facilitar a criação de componentes formados por diferentes objetos, chamados *segmentos*. Os *templates* também podem criar uma infra-estrutura especial para execução dos componentes. Por exemplo, o LOOP oferece suporte para *templates* que criam uma infra-estrutura para interceptação de operações nas portas dos componentes, de forma similar à abordagem baseada em *proxies* apresentada na seção 2.2.1.

Um *template* é criado a partir de uma tabela que define o conjunto de portas que o componentes deve oferecer, como ilustrado na linha 5 da figura 3.4. Depois disso, pode-se utilizar o *template* para criar uma fábrica de componentes que sigam o molde de portas definido na criação do *template* (linha 14). Na criação da fábrica, são fornecidos construtores dos objetos que implementam o componente. Tipicamente, esses construtores são classes LOOP ou funções que constroem objetos. É possível definir um construtor para cada porta do componente. Além disso, é possível definir um construtor principal no índice 1 da tabela. Esse construtor é responsável por criar o segmento principal do componente, mas também pode criar os demais segmentos que implementam as facetas. Sempre que uma faceta não possuir um construtor associado e o construtor principal não criar um segmento para aquela faceta, o *template*

[1] `http://loop.luaforge.net/`

PUC-Rio - Certificação Digital Nº 0510971/CA

considera que a faceta é implementada pelo segmento principal. Isso facilita a criação de componentes com múltiplas facetas que são implementados por um único objeto ou segmento. Tais componentes são denominados componentes *monolíticos*.

Figura 3.4: Criação de componentes usando o suporte oferecido pelo LOOP.

```
1  local component = require "loop.component.base"
2  local port      = require "loop.component.port"
3
4  -- criacao de template de componente
5  local TypeRepository = component.Template{
6    types     = port.Facet,
7    importer  = port.Facet,
8    indexer   = port.Facet,
9    compiler  = port.Facet,
10   delegated = port.Receptacle,
11 }
12
13 -- criacao de fabricas de componente
14 local Factory  = TypeRepository{ require "oil.corba.idl.Repository",
15                       importer = require "oil.corba.idl.Importer"  ,
16                       indexer  = require "oil.corba.idl.Indexer"   ,
17                       compiler = require "oil.corba.idl.Compiler"   }
18
19 -- criacao de instancias de componente
20 local myRepo = Factory()
21
22 -- utilizacao de portas de componente
23 myRepo.delegated = broker:newproxy(io.read())
24 myRepo.importer:lookup("MyInterface")
```

Conexões são estabelecidas entre os componentes ligando facetas aos receptáculos dos componentes. O código da figura 3.3 mostra a criação de *templates* de alguns componentes que fazem parte da arquitetura interna do OiL (linhas 7–21) como será descrito na seção 3.3. A função `assemble` (linha 23) define como componentes seguindo esses *templates* são interconectados, inclusive as conexões com outros componentes do ORB (linhas 24, 28 e 29). Todos os componentes criados para a montagem do ORB são fornecidos na tabela passada no parâmetro `orb`.

O acesso aos objetos conectados aos receptáculos é feito através de objetos de *contexto*. O contexto é uma tabela que contém todos os segmentos do componente e objetos conectados através dos receptáculos. Tipicamente, o contexto é fornecido como um campo `context` adicionado a cada segmento do componente na sua criação. Entretanto, alguns *templates* permitem que componentes monolíticos acessem os receptáculos diretamente como campos do segmento principal.

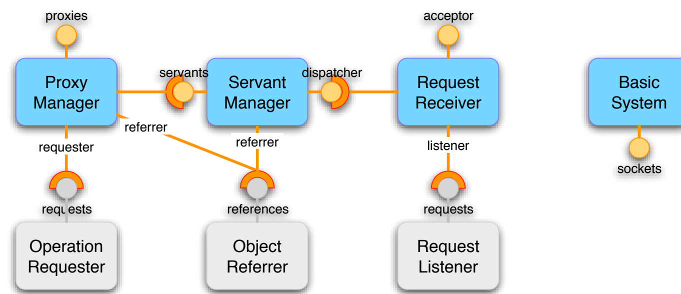
O uso de *templates* na criação de fábricas de componentes é apenas uma facilidade oferecida pela biblioteca LOOP e não a única forma de criação de componentes que sigam o modelo de componentes utilizado no OiL.

PUC-Rio - Certificação Digital Nº 0510971/CA

## 3.3
## Camada de Apresentação

A camada de apresentação inclui pelo menos quatro componentes: dois para suporte à criação de *servants* (lado servidor), um para suporte ao uso de *proxies* (lado cliente) e outro que oferece os recursos da plataforma de execução, como criação de *sockets* ou *threads*. A figura 3.5 ilustra a arquitetura da camada de apresentação. Essa arquitetura é a mesma definida pela função `assemble` da figura 3.3. Os componentes em cinza claro representam componentes da camada do protocolo e não são necessariamente distintos. Abaixo, é feita uma descrição dos componentes básicos dessa camada.

Figura 3.5: Arquitetura da camada de apresentação de ORBs OiL




`ServantManager` Componente que define a forma com que os *servants* são registrados e como as requisições endereçadas a eles são despachadas. A faceta `servants` permite registrar objetos locais como *servants* e recuperar sua implementação a partir de sua chave de objeto ou referência de objeto (*e.g.* IOR). Quando um novo *servant* é registrado, uma referência de objeto distribuído é criada através do receptáculo `referrer` com base em informações de acesso ao ORB previamente configuradas neste componente. Essas informações de acesso são obtidas do componente `RequestReceiver` durante a iniciação do ORB. A faceta `dispatcher` permite despachar as requisições para seus *servants*, que resulta na execução da sua respectiva implementação.

`RequestReceiver` Componente que define a forma como as requisições são recebidas e processadas, gerenciando o ciclo de vida dos canais de recebimento de requisições remotas. A faceta `acceptor` permite iniciar um ponto de acesso ao ORB e controlar a aceitação de requisições através desse ponto. Isso é feito através do receptáculo `listener` que recebe canais de requisição de ORBs remotos, e também permite receber requisições e envia respostas por esses canais. O `RequestReceiver` pode implementar diferentes abordagens para o processamento das requisições. Por exemplo, a leitura de cada canal pode ser

PUC-Rio - Certificação Digital Nº 0510971/CA

feita em uma *thread* independente, assim como cada nova requisição pode ser despachada em uma *thread* diferente.

`ProxyManager` São componentes que definem a forma com que os objetos remotos são acessíveis pela aplicação. Isso é feito através da criação de *proxies* a partir de referências textuais, como IOR textual (*stringfied IOR*) ou *corbaloc* do padrão CORBA. Para tanto, eles fazem uso de um decodificador conectado ao receptáculo `referrer` para extrair as informações das referências textuais. Tipicamente, cada `ProxyManager` implementa um único tipo de *proxy*. Entretanto, uma montagem de ORB pode incluir múltiplos `ProxyManager`. Os *proxies* não realizam efetivamente as chamadas; ao invés disso eles utilizam um componente externo conectado através do receptáculo `requester` do `ProxyManager`. O receptáculo `servants` é opcional, ou seja, não é necessário conectá-lo a um objeto para que o componente funcione. Quando disponível, ele é utilizado para identificar se uma referência utilizada na criação de um novo *proxy* é local, ou seja, referencia um *servant* local. Se este for o caso, o `ProxyManager` utiliza esse mesmo receptáculo para recuperar o objeto local que implementa o *servant*.

`BasicSystem` Componente que oferece acesso a recursos do software básico da plataforma. A principal motivação deste componente é definir interfaces de acesso às chamadas do sistema. Um exemplo isso é a faceta `sockets` que é utilizada pelos componentes da camada do protocolo para implementação de comunicação em rede através da criação e utilização de *sockets*.

A montagem `basic` constrói e monta esses quatro componentes, inclusive as conexões com os componentes da camada do protocolo, se estiverem disponíveis. Essa montagem também pode instanciar componentes `ProxyManager` adicionais, que implementam *proxies* alternativos. A implementação atual do OiL oferece suporte para três tipos de *proxies* alternativos que oferecem modelos de programação ligeiramente diferentes. A definição dos componentes `ProxyManager` a serem instanciados na montagem `basic` é dada pelos campos `proxykind` e `extraproxies` da tabela de configurações do ORB fornecida para operação `oil.init`.

### 3.3.1 Proxies Protegidos

Chamadas remotas estão mais sujeitas a erros que chamadas locais. Por exemplo, falhas de acesso a um objeto referenciado não ocorrem localmente,

PUC-Rio - Certificação Digital Nº 0510971/CA

mas pode ser um erro freqüente em um ambiente distribuído devido a problemas de comunicação de rede. Por isso, não é raro que um programa que faz chamadas remotas precise tratar muitos erros relativos a essas chamadas. Contudo, o lançamento e captura desses erros pode não ser eficiente em muitos sistemas, pois são projetados para situações adversas e não para ser usados corriqueiramente pelo programa. Em Lua, por exemplo, a captura de erros é feita através da função `pcall`, que implica em uma nova chamada de função em cada chamada protegida ou na criação de um fecho com todas as chamadas que podem gerar um erro. Finalmente, a utilização do `pcall` com operações de objetos impossibilita o uso do operador `:`. Como conseqüência disso, tem-se que o lançamento de erros em Lua nem sempre seja a melhor forma de sinalizar condições adversas em uma chamada remota.

Para contornar esse problema, o OiL oferece suporte para *proxies protegidos*, que são *proxies* cujas operações não lançam erros. Ao invés disso, as operações retornam um valor adicional que indica se a chamada foi feita com sucesso ou não, de forma idêntica à chamada da função `pcall` de Lua. A utilização desses *proxies* simplifica o tratamento minucioso de erros pelo programa. Para ilustrar isso, considere a implementação do serviço de notificação ilustrado na figura 3.6, onde um conjunto de objetos remotos são registrados como observadores (linha 8) que recebem notificações de eventos publicados através da operação `notify`. Sempre que o envio de uma notificação para um observador falhar, este deve ser removido da lista de notificação (linha 21), pois considera-se que ele se tornou inacessível permanentemente. Para tanto, o ORB utilizado é configurado para usar *proxies* protegidos. Isso é feito através do campo `proxykind`, que é fornecido na criação do ORB (linha 3). Com uso desse tipo de *proxy*, a chamada remota na linha 17 não lança erro caso o objeto remoto não esteja acessível, mas retorna uma indicação de erro que pode ser testada no programa (linha 18).

### 3.3.2 Proxies Assíncronos

Chamadas síncronas também podem não ser adequadas em algumas situações. Ainda no exemplo da figura 3.6, as notificações devem ser enviadas o mais rápido possível, independentemente da demora da confirmação de cada observador. Por isso, uma nova *thread* é iniciada para cada observador que envia a notificação e aguarda sua confirmação (linha 16). A dificuldade dessa abordagem é a sincronização das *threads* criadas e a chamada da função, pois a função `notify` deve informar o número notificações feitas com sucesso. Isso é feito da seguinte maneira. A variável `pending` é utilizada para contar o

PUC-Rio - Certificação Digital Nº 0510971/CA

Figura 3.6: Notificação de eventos com chamadas protegidas e concorrência.

```
1  local broker = oil.init{
2    flavor = "cooperative;corba",
3    proxykind = "protected",
4  }
5
6  Publisher = { observers = {} }
7  function Publisher:subscribe(observer)
8    self.observers[observer] = true
9  end
10 function Publisher:notify(event)
11   local suspended = false
12   local success = 0
13   local pending = 0
14   for observer in pairs(self.observers) do
15     pending = pending + 1
16     oil.tasks:start(function()
17       local ok, except = observer:push(event)
18       if ok then
19         success = success + 1
20       else
21         self.observers[observer] = nil
22       end
23       pending = pending - 1
24       if pending == 0 and suspended then
25         return oil.tasks:resume(suspended)
26       end
27     end)
28   end
29   if pending > 0 then
30     suspended = oil.tasks.current
31     oil.tasks:suspend()
32   end
33   return success
34 end
35
36 broker:loadidlfile("event_publisher.idl")
37 Publisher.__type = "Publisher"
38 print(broker:newservant(Publisher))
39 broker:run()
```

número de *threads* criadas para enviar notificações (linha 15). Posteriormente, quando uma dessas *threads* termina, essa variável é decrementada (linha 23). Dessa forma, quando o seu valor chega a zero, todas as *threads* criadas terão completado suas tarefas. Após todas as *threads* terem sido iniciadas, se alguma delas ainda estiver em execução (linha 29), a *thread* que executa a função `notify` se registra na variável `suspended` (linha 30) e suspende sua execução até que todas as *threads* de notificação tenham terminado (linhas 31). Quando uma *thread* de notificação completa sua tarefa, ela verifica se não há mais outras *threads* em execução (linha 24). Se esse for o caso, a *thread* da função `notify` é restaurada (linha 25) e a chamada da função retorna o seu resultado (linha 33).

Uma alternativa mais simples e tão eficiente quanto a apresentada anteriormente é possível através do uso de chamadas assíncronas. O código da figura 3.7 ilustra a implementação do serviço de notificação usando *proxies assíncronos*. Na criação do ORB, o campo `proxykind` é usado para determinar que todos os *proxies* criados por aquele ORB são assíncronos (linha 3).

PUC-Rio - Certificação Digital Nº 0510971/CA

Chamadas de operação em um *proxy* assíncrono apenas enviam a requisição, mas não esperam pelos resultados. Ao invés disso, a chamada retorna um *futuro*, que é um objeto que representa os resultados a serem recebidos. O futuro permite verificar a disponibilidade dos resultados através da operação `ready` (processo conhecido como *polling*), ou obter os resultados através da operação `evaluate`, que bloqueia até que os resultados estejam disponíveis. Uma alternativa à `evaluate` é a operação `results`, que obtém os resultados em modo protegido, ou seja, sem lançar erros ou exceções, mas retornando uma valor adicional que indica se a chamada resultou em um erro ou não.

Figura 3.7: Notificação de eventos com chamadas assíncronas.

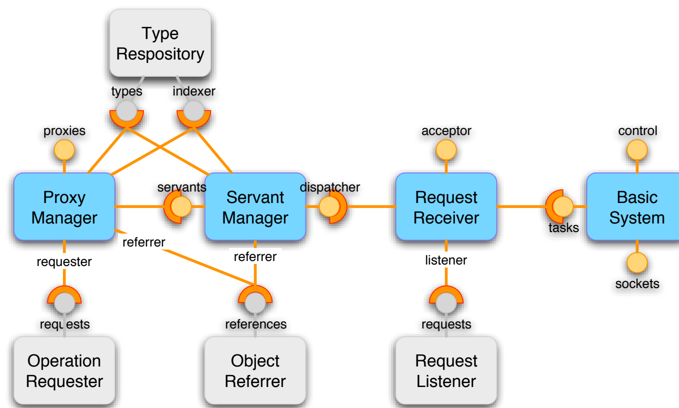
```
1 local broker = oil.init{
2   flavor = "corba",
3   proxykind = "asynchronous",
4 }
5
6 Publisher = { observers = {} }
7 function Publisher:subscribe(observer)
8   self.observers[observer] = true
9 end
10 function Publisher:notify(event)
11   local success, futures = 0, {}
12   for observer in pairs(self.observers) do
13     futures[observer] = observer:push(event)
14   end
15   for observer, future in pairs(futures) do
16     local ok, except = future:results()
17     if ok then
18       success = success + 1
19     else
20       self.observers[observer] = nil
21     end
22   end
23   return success
24 end
25
26 broker:loadidlfile("event_publisher.idl")
27 Publisher.__type = "Publisher"
28 print(broker:newservant(Publisher))
29 broker:run()
```

Com o uso de *proxies* assíncronos, a operação `notify` consiste apenas em enviar a notificação para todos os observadores cadastrados guardando os futuros na tabela `futures` (linha 13), e então obter a confirmação de todos os observadores em seqüência (linha 16). Note que a ordem em que as confirmações são recebidas não influencia significativamente no tempo de execução da função `notify`, pois ela só deve retornar quando todas as confirmações forem recebidas.

Adicionalmente, o OiL também define duas extensões da camada de apresentação, descritas nas seções seguintes. A figura 3.8 ilustra a arquitetura da camada de apresentação com essas duas extensões.

PUC-Rio - Certificação Digital Nº 0510971/CA

Figura 3.8: Arquitetura da camada de apresentação com extensões.




### 3.3.3
### Extensão para Suporte a Interfaces

Objetos em Lua não estão associados necessariamente a uma interface com operações pré-definidas. Ao invés disso, a interface de um objeto em Lua é definida exclusivamente pela sua implementação e pode inclusive mudar durante a execução do programa. Entretanto, muitas tecnologias de RMI se baseiam em declarações de interfaces tipadas para permitir chamadas de operações remotas. Ou seja, é necessário definir explicitamente as interfaces oferecidas pelos objetos distribuídos, assim como os tipos dos valores manipulados nessas interfaces. Para dar suporte a essas tecnologias, o OiL define uma extensão da camada de apresentação para introduzir suporte a declarações de interfaces tipadas.

Essa extensão define implementações alternativas dos seguintes componentes: `ProxyManager`, que implementa *proxies* que só oferecem as operações definidas na interface do objeto remoto; e `ServantManager`, que exige a definição da interface do *servant* na sua criação e só despacha operações que sejam definidas nessa interface. Ambas implementações utilizam as informações fornecidas por um componente adicional denominado `TypeRepository`.

O `TypeRepository` é um componente da camada do protocolo que oferece duas facetas `types` e `indexer` e ambas são conectadas aos componentes `ProxyManager` e `ServantManager` estendidos nessa camada. A faceta `types` é utilizada pelo componente `ProxyManager` para obter descritores das interfaces dos *proxies* criados através dele. A aplicação pode determinar essas interfaces de diferentes maneiras, como por exemplo um identificador da interface ou uma tabela Lua com uma estrutura específica que descreve a interface. Além disso, a faceta `types` também oferece operações para descobrir automaticamente a interface de um proxy; A faceta `indexer` permite obter um descritor de um membro da interface a partir do seu nome e o descritor da interface. Diferen-

PUC-Rio - Certificação Digital Nº 0510971/CA

temente da implementação do `ProxyManager` da camada de apresentação, que utiliza o nome da operação como identificação da operação sendo chamada, o componente dessa extensão utiliza esse descritor obtido através da faceta `indexer`. Isso permite que, quando uma chamada seja feita através do componente `OperationRequester` da camada do protocolo, ele não precise consultar o `TypeRepository` para obter o informações sobre operação, tais como tipos dos parâmetros e valores de retorno. De forma similar, as duas facetas também são utilizadas pelo componente `ServantManager`, tanto para obter os descritores das interfaces dos *servants* quando eles são registrados, como para determinar se as operações sendo despachadas são válidas.

A utilização dessa extensão é geralmente definida pelos componentes da camada do protocolo. Por exemplo, a implementação da camada do protocolo que oferece suporte CORBA exige a utilização dessa extensão, pois as chamadas só podem ser feitas com base em informações de interface fornecidas pela aplicação. Por outro lado, a implementação da camada do protocolo LuDO não funciona com essa extensão, pois ela não implementa o componente `TypeRepository`. Por essas razões, não é necessário definir essa extensão na criação do ORB, pois isso é feito automaticamente de acordo com a camada de protocolo escolhida.

PUC-Rio - Certificação Digital Nº 0510971/CA

### 3.3.4 Extensão para Concorrência

Aplicações distribuídas são inerentemente concorrentes, pois cada parte da aplicação é um processo que executa em uma unidade de distribuição (processador ou computador) de forma independente das demais. É possível ocultar essa concorrência sincronizando esses processos de forma que apenas um execute por vez. Entretanto, isso representaria um enorme desperdício de processamento em muitas aplicações. Por essa razão, uma parte importante do desenvolvimento de aplicações distribuídas é permitir que cada processo da aplicação possa interagir simultaneamente com múltiplos pares que executam independentemente, de forma a maximizar o seu desempenho. Uma forma comum de fazer isso é através de concorrência, que é o suporte a múltiplas linhas de execução independentes, ou *threads*, em um mesmo programa. Cada parte da aplicação distribuída pode criar uma *thread* diferente para interagir com as demais partes remotas, de forma que a iteração de uma parte da aplicação não interfira na outra.

O OiL oferece suporte para concorrência através de um modelo cooperativo. Neste modelo, uma *thread* em execução só deixa de executar, passando a vez para outra *thread*, em pontos pré-determinados da sua execução, denomina-

dos de *pontos de escalonamento*. Uma das maiores dificuldades da programação com concorrência cooperativa é a definição dos pontos de escalonamento de forma adequada para garantir que a execução de todas as *threads* cooperem entre si para que possam prosseguir sua execução de forma igualitária (denominado *fairness*) ou, pelo menos, de forma adequada com as necessidades particulares da aplicação. Por outro lado, a execução das *threads* é tipicamente determinística e problemas relacionados a condições de corrida e sincronização de *threads* (Tanenbaum, 2001) são mais facilmente tratados do que em modelos baseados em concorrência preemptiva (Moura e Ierusalimschy, 2004).

O suporte a concorrência do OiL é baseado no suporte de co-rotinas de Lua através de um *escalonador*. A principal função do escalonador é auxiliar a aplicação na gerência da execução de suas várias co-rotinas, que são chamadas neste trabalho de *threads*. O escalonador faz isso através de uma escala de execução das *threads*, ou seja uma ordem em que elas são executadas. Tipicamente, o escalonador fica ativo durante toda aplicação, executando todas as *threads* escalonadas continuamente. O escalonador oferece as operações de controle `run`, que executa todas as *threads* escalonadas continuamente até que não haja mais nenhuma, e `step`, que executa cada *thread* escalonada uma única vez. As *threads* são registradas para escalonamento através da operação `register` do escalonador. A qualquer momento a aplicação pode remover *threads* do escalonamento através da operação `remove`.

O escalonador é na verdade um componente do ORB. O OiL permite que cada ORB possa utilizar um escalonador diferente, com implementações, funcionalidades e políticas diferentes. Apesar do uso de múltiplos escalonadores em uma mesma aplicação permitir a criação de esquemas de escalonamento elaborados, como a separação das *threads* em grupos com prioridades de escalonamento, isso é incomum. Por essa razão, o módulo `oil` cria um escalonador *default*, que é utilizado na criação de todos os ORBs, de forma que todos eles compartilhem o mesmo escalonador. O escalonador *default* fica disponível através do campo `oil.tasks`.

Todas implementações de componentes do OiL são feitas de forma a funcionar corretamente no contexto de uma *thread* cooperativa (*thread-safe*), como é apresentado na seção 4.4.3. Apesar da montagem básica do ORB funcionar corretamente em aplicações que usam concorrência, o ORB não faz uso do recurso de *threads*, como por exemplo, para despachar chamadas de forma independente uma das outras. Para ativar isso, é necessário utilizar a montagem `cooperative`, que define uma extensão da camada de apresentação com implementações alternativas de dois componentes. O primeiro é o `BasicSystem`, que implementa o escalonador de *threads* cooperativas do OiL, através de uma

PUC-Rio - Certificação Digital Nº 0510971/CA

faceta adicional denominada `tasks`. A faceta *control* oferece as operações de controle do escalonador como `run` e `step`. O segundo componente é o `RequestReceiver`, que passa a apresentar um receptáculo para acessar o suporte a *threads* oferecido pelo `BasicSystem`. Essa implementação alternativa do componente `RequestReceiver` cria *threads* para receber requisições oriundas de diferentes canais, minimizando a interferência entre eles, e também para despachar cada requisição em uma *thread* independente, permitindo tratar requisições concorrentemente.

Essa extensão pode ser utilizada com qualquer camada de protocolo. Por isso, para utilizar essa extensão, é necessário definir explicitamente o nome `cooperative` no campo `flavor` na criação do ORB.

### 3.3.5 Interação com a Camada do Protocolo

Apesar de não ser necessário definir explicitamente as interfaces dos componentes, é necessário que haja uma interface em comum para que um componente possa utilizar o serviço oferecido por outro. Essas interfaces não são definidas *a priori* pelo OiL, pois elas dependem exclusivamente das implementações específicas dos componentes que interagem entre si. Entretanto, para permitir a interoperabilidade entre os componentes da camada de apresentação e diferentes implementações da camada do protocolo, é necessário definir as interfaces utilizadas nessas interações, assim como os elementos manipulados. Quatro elementos básicos são utilizados na comunicação entre os componentes das duas camadas, conforme descritos a seguir.

**Ponto de Acesso** Um ponto de acesso é uma tabela com informações que definem a forma de acesso remoto ao ORB. Esses valores podem ser definidos com base em informações oferecidas pela aplicação ou pelos seus componentes internos (*e.g.* valores *default*). A importância dos pontos de acesso é permitir que cada ORB possa ter características de acesso específicas e elas sejam disponíveis para todos os seus componentes internos. Um exemplo disso são informações como a interface de rede e a porta a serem utilizadas nas comunicações sobre TCP (*Transmission Control Protocol*). Tipicamente, o ponto de acesso é criado pelo componente `RequestListener` da camada do protocolo com base nas informações da tabela de configurações fornecida na chamada da operação `oil.init`, que cria o ORB. O ponto de acesso é armazenado nos componentes `RequestReceiver`, para controlar a aceitação de canais de requisição, e também no `ServantManager`, que utiliza essas informações na criação de referências de objeto dos *servants* através do

PUC-Rio - Certificação Digital Nº 0510971/CA

componente ObjectReferrer da camada do protocolo.

**Referências de Objeto** Uma referência de objeto é um conjunto de informações sobre um objeto distribuído. Uma referência deve conter pelo menos as informações necessárias para se comunicar remotamente com esse objeto. A referência também pode conter outras informações adicionais sobre esse objeto, tais como a interface que ele oferece, políticas que regem a forma de comunicação com o ele, entre outras. As referências são tabelas Lua, que podem conter um conjunto arbitrário de informações sobre o objeto referenciado. Inclusive, o uso de tabelas permite associar informações temporárias utilizadas para otimização de algumas tarefas. Um exemplo desse tipo de otimização é armazenar o canal de comunicação estabelecido na primeira chamada de operação feita através daquela referência, de forma que ele possa ser reutilizado em chamadas futuras. A importância das referências é permitir identificar objetos distribuídos. As referências são criadas pelo componente ObjectReferrer da camada do protocolo, sempre que um novo *servant* é registrado no ServantManager ou quando um novo *proxy* é criado a partir de uma referência de objeto textual que deve ser primeiramente decodificada pelo ObjectReferrer.

**Canais de Requisição** Um canal de requisição é uma representação de um meio de recebimento de requisições estabelecido através de um ponto de acesso do ORB. Requisições de operação são recebidas seqüencialmente através do canal, ou seja, enquanto uma requisição estiver sendo lida nenhuma outra pode ser obtida através do mesmo canal. Por outro lado, requisições oriundas de diferentes canais podem ser lidas independentemente, ou seja, múltiplas requisições podem ser lidas de diferentes canais simultaneamente. A importância dos canais é fazer com que a criação ou aceitação das conexões com clientes seja feita pelo componente RequesterListener da camada do protocolo, mas a forma com que esses canais são lidos possa ser implementada pelo componente RequestReceiver da camada de apresentação, simplificando a implementação da camada do protocolo.

**Requisições de Operação** Uma requisição de operação é uma tabela Lua que é criada sempre que uma nova chamada de operação é iniciada ou recebida. Cada requisição identifica uma chamada pendente e armazena informações como o nome da operação sendo chamada, a chave de objeto do *servant* de destino, seus parâmetros, entre outras informações. A requisição também armazena os resultados da chamada quando estiverem disponíveis e permite con-

PUC-Rio - Certificação Digital Nº 0510971/CA

sultá-los. Como a requisição é uma tabela, ela pode armazenar informações adicionais sobre a chamada, tais como o componente que realizou a chamada, o canal de requisição utilizado, etc. A importância das requisições é permitir identificar cada chamada pendente e facilitar a implementação de chamadas assíncronas e concorrentes. As requisições de operação são criadas pelos componentes `OperationRequester` e `RequestListener` da camada do protocolo.

Esses quatro elementos são criados pelos componentes da camada do protocolo e são opacos para os componentes da camada de apresentação. A exceção dessa regra são as requisições de operação, que devem apresentar uma estrutura específica que permite a consulta de alguns de seus dados internos, como nome da operação, valores dos parâmetros e resultados.

Com base nesses quatro elementos, são definidas as três interfaces que devem ser oferecidas pelos componentes da camada do protocolo, conforme descritas a seguir. A figura 3.9 ilustra um pseudo-código que descreve essas três interfaces. Todas as operações dessas interfaces seguem um padrão comum em Lua para indicação de exceções, que consiste em caso de sucesso devolver os valores de resultado, porém em caso de falha o valor devolvido é `nil` seguido de um outro valor descrevendo a exceção.

Figura 3.9: Interfaces de integração com a camada do protocolo.

```
accesspt  = RequestListener.acceptor:setupaccess(configs)
channel   = RequestListener.acceptor:getchannel(accesspt)
request   = RequestListener.acceptor:getrequest(channel)
success   = RequestListener.acceptor:sendreply(request, success, ...)
success   = RequestListener.acceptor:freeaccess(access)
success   = RequestListener.acceptor:freechannel(channel)

reference = ObjectReferrer.references:newreference(servantid, accesspt)
stringref = ObjectReferrer.references:encode(reference)
reference = ObjectReferrer.references:decode(stringref)

request = OperationRequester.requests:newrequest(reference, operation, ...)
success = OperationRequester.requests:getreply(request)
```

**Request Listener** A iniciação do ORB para processar requisições remotas inclui a criação de um ponto de acesso através da operação `setupaccess`. Essa operação utiliza um conjunto de configurações fornecido pela aplicação em uma tabela Lua que mapeia nomes de propriedades para seus respectivos valores. O conjunto de propriedades considerado é exclusivamente dependente dos componentes da camada do protocolo. Uma vez criado o ponto de acesso, ele pode ser usado na criação de referências de objeto como será discutido a seguir. O ponto de acesso também é utilizado para aceitar canais de requisição através da operação `getchannel`. A operação `getrequest` é utilizada para

PUC-Rio - Certificação Digital Nº 0510971/CA

obter requisições a partir de um dado canal. Tanto a operação `getchannel` como a `getrequest` podem bloquear até que um novo canal ou requisição esteja disponível. Isso é determinado por um parâmetro adicional que indicia se a operação deve ser bloqueante ou não. Uma vez que os resultados de uma chamada estejam disponíveis, a operação `sendreply` é utilizada para devolver os resultados. Por fim, quando o processamento de requisições do ORB é encerrado, as operações `freeaccess` e `freechannel` são chamadas para liberar quaisquer recursos associados ao ponto de acesso ou a seus canais.

**Object Referrer** A principal operação definida por essa interface é a operação `newreference`, que é utilizada para criar referências de objeto a partir de uma chave do objeto e um conjunto de pontos de acesso utilizados pelo ORB. A chave do objeto é tipicamente um texto em uma *string* que o identifica univocamente no ORB. A estrutura dos pontos de acesso só é conhecida pelos componentes da camada do protocolo, que a utilizam para extrair informações necessárias para criação de canais de requisição. De forma similar, a estrutura da referência criada também é opaca e sua estrutura só é conhecida pelos componentes da camada do protocolo, como discutido no próximo parágrafo. Duas outras operações adicionais devem ser fornecidas para codificar referências de objeto em texto e vice versa. Isso é importante para permitir passar referências como parâmetros de linha de comando ou gravar essas informações em arquivo.

**Operation Requester** A operação `newrequest` pode ser utilizada para criar uma nova requisição, dado uma referência de objeto, uma identificação da operação sendo chamada e um conjunto de valores de parâmetros dessa operação. A referência é utilizada para criar um canal de envio de requisição usando os dados sobre o ponto de acesso do ORB e também a chave do objeto. A identificação da operação é tipicamente um texto indicando o nome da operação, mas em alguns casos pode ser algo mais complexo, como uma estrutura contendo diversas informações sobre a operação a ser chamada. Os valores dos parâmetros são arbitrários e fornecidos diretamente pela aplicação. Uma vez que uma nova requisição seja criada, é possível consultar se seus resultados estão disponíveis através do campo `success` que deve conter um valor boleano indicando se a chamada foi feita com sucesso. Caso esses resultados não estejam disponíveis o campo `success` é `nil` e é possível consultar sua disponibilidade através da operação `getreply`. Essa operação pode bloquear até que a resposta esteja disponível, dependendo do valor de um parâmetro opcional.

PUC-Rio - Certificação Digital Nº 0510971/CA

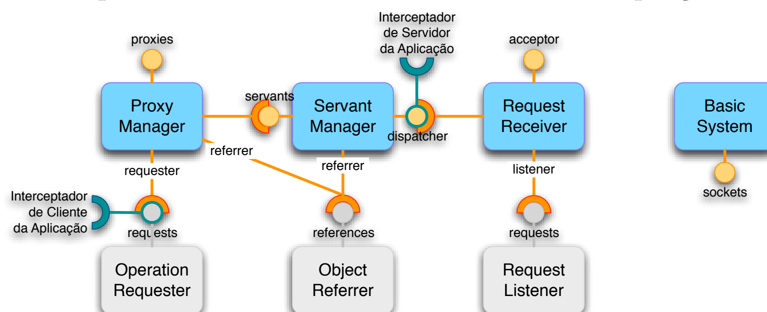
### 3.3.6 Interceptação de Chamadas

A arquitetura de componentes dos ORBs do OiL permite a implementação de um mecanismo de interceptação de chamadas através da interceptação de algumas portas de componentes. Todas as chamadas iniciadas pelo ORB são feitas através da porta `requests` do componente `OperationRequester`, portanto a interceptação dessa porta permite interceptar todas essas chamadas. De forma similar, todas as chamadas recebidas pelo ORB são despachadas através da porta `dispatcher` do componente `ServantManager`, que representa o ponto para interceptação de chamadas recebidas.

A interceptação de portas pode ser feita através da inserção de um componente adicional entre as duas portas conectadas. Neste caso, o componente repassa as chamadas de uma porta para outra tendo a oportunidade de inspecioná-las e alterá-las. Contudo, a inserção de um componente adicional pode ser trabalhosa em conexões de cardinalidade maior que um. Por exemplo, em uma montagem de ORB, é possível haver vários componentes `ProxyManager` conectados ao `OperationRequester` para iniciar chamadas. Por isso, seria necessário rastrear todas essas conexões para inserir o componente de interceptação em cada uma delas.

Alternativamente, o OiL utiliza o suporte a interceptação de portas de componentes oferecido pelo LOOP. Nesse caso, é possível registrar um interceptador em uma porta específica do componente e inspecionar ou alterar qualquer chamada feita através de conexões que envolvam aquela porta. A interceptação de chamadas nos ORBs do OiL então se resume a registrar uma interceptador especial nas portas dos componentes `OperationRequester` e `ServantManager`, que repassa a interceptação para um interceptador da aplicação, como ilustrado na figura 3.10.

Figura 3.10: Arquitetura da camada básica com interceptação de chamadas.

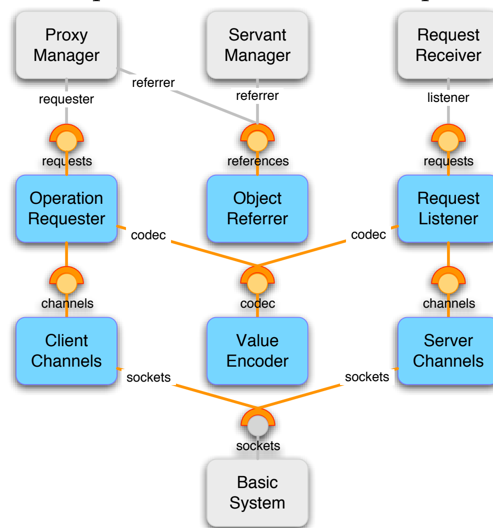



PUC-Rio - Certificação Digital Nº 0510971/CA

## 3.4
## Camada do Protocolo LuDO

O LuDO (*Lua Distributed Objects*) é um protocolo de RMI baseado em Lua desenvolvido especificamente neste trabalho. Lua influencia o LuDO em vários aspectos. Primeiramente, as mensagens e valores enviados pela rede são codificados como trechos de código Lua, que quando executados decodificam os valores. Por essa razão, as chamadas feitas através do LuDO podem utilizar quaisquer valores que possam ser integralmente inspecionados e reconstruídos a partir de Lua, o que inclui, além de valores simples, tabelas e funções Lua cujo fecho não compartilhe variáveis com outras. Portanto, é possível passar algumas funções como parâmetro em chamadas de operação através do LuDO. Inclusive, esses valores também podem ser lançados como erro nas chamadas através do LuDO. Assim como em Lua, os objetos acessíveis pelo LuDO não definem interfaces e suas operações podem ter um número arbitrário de parâmetros e valores de retorno.

Figura 3.11: Arquitetura da camada do protocolo LuDO.




PUC-Rio - Certificação Digital Nº 0510971/CA

A camada de protocolo do LuDO é composta por seis componentes, descritos a seguir. A figura 3.11 ilustra a arquitetura da camada do protocolo LuDO.

`ClientChannels` Componente utilizado pelo `OperationRequester` para obter canais de comunicação para destinos remotos. Os canais de comunicação são *sockets* criados através do receptáculo `sockets`, que é conectado à faceta de mesmo nome do componente `BasicSystem` da camada de apresentação. O `ClientChannels` é responsável pela gerência das conexões estabelecidas com destinos remotos, recuperando-as sempre que requisitadas pelo componente `OperationRequester`.

`ServerChannels` Componente utilizado pelo `RequesterListener` para estabelecer um ponto de acesso ao ORB e obter canais de comunicação estabelecidos através dele. De forma similar ao `ClientChannels`, os canais são *sockets* criados através de um receptáculo conectado à faceta `sockets` do componente `BasicSystem`. Entretanto, o `ServerChannels` não gerencia o ciclo de vida das conexões. Ao invés disso, oferece funções que permitem descartar as conexões.

`ValueEncoder` Componente que implementa o mecanismo de codificação de valores em código Lua e a posterior restauração desses valores através da execução do código gerado. Todos os valores são enviados por cópia, inclusive tabelas e funções. Para enviar uma referência de objeto, é necessário criar um *proxy* para o objeto e enviá-lo por cópia. Ele é utilizado pelos componentes `OperationRequester` e `RequestListener` para codificação e decodificação das requisições transmitidas pela rede.

`ObjectReferrer` Componente utilizado pela camada de apresentação para criar referências de objeto LuDO com base nas informações de acesso do ORB. Essas referências são tabelas com os campos: `host`, que é uma *string* indicando o nome ou endereço IP da maquina onde reside o objeto remoto; `port`, que é o número da porta do ORB em que esse objeto reside; e `object`, que é uma *string* com a chave do objeto remoto. Esse componente também permite codificar e decodificar essas informações em *strings*.

`OperationRequester` Componente utilizado pela camada de apresentação para enviar requisições a objetos remotos e obter os resultados correspondentes. O receptáculo `codec` é utilizado para codificação das requisições a serem transmitidas pela rede, assim como a decodificação das respostas recebidas.

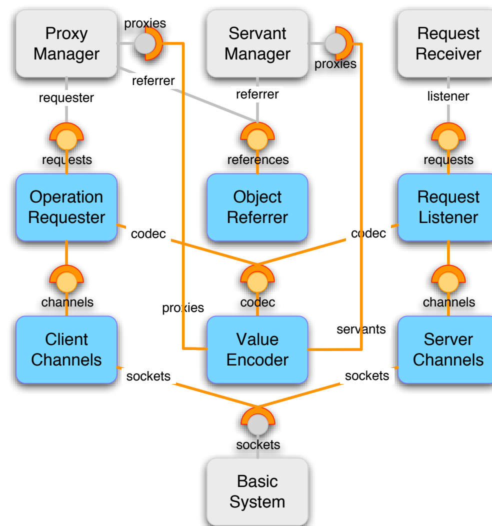
PUC-Rio - Certificação Digital Nº 0510971/CA

`RequestListener` Componente utilizado pela camada de apresentação para iniciar pontos de acesso, aceitar canais de requisição estabelecidos a partir desses pontos, receber requisições a *servants* locais e enviar os resultados dessas chamadas. O receptáculo `codec` é utilizado para decodificação das requisições recebidas pelos canais, assim como a codificação das respostas a serem transmitidas.

### 3.4.1
### Extensão para Passagem por Referência

O OiL também define uma extensão da camada do protocolo LuDO, denominada `ludo.byref`. Essa extensão define uma implementação alternativa do componente `ValueEncoder` que permite o envio de valores por referência. Para tanto, dois novos receptáculos são definidos nesse componente, denominados `servants` e `proxies`. Esses receptáculos são conectados às facetas de mesmo nome dos componentes `ServantManager` e `ProxyManager`, respectivamente. A figura 3.12 ilustra a arquitetura da chamada LuDO com a extensão para passagem por referência.

Figura 3.12: Arquitetura estendida da camada do protocolo LuDO.




Sempre que uma tabela, função, *userdata* ou *thread* é codificado, esse valor é registrado implicitamente como um *servant* local através da faceta `servants`. Após o registro, referência de objeto desse *servant* é codificada e enviada pela rede ao invés de uma cópia do valor original. Quando a referência é decodificada (*i.e.* executada), um *proxy* do objeto remoto referenciado é criado automaticamente através do receptáculo `proxies`. Se a referência for para um *servant* local, o componente `ProxyManager` é responsável por recuperar sua implementação no momento da criação do *proxy*.

PUC-Rio - Certificação Digital Nº 0510971/CA

## 3.5 Camada do Protocolo CORBA

O OiL oferece uma segunda implementação da camada de protocolo que oferece suporte para os protocolos de RMI definidos pelo padrão CORBA, denominados genericamente de GIOP (*General Inter-ORB Protocol*) (Object Management Group, 2002). O GIOP é significativamente diferente do LuDO. Ele é um protocolo de alto desempenho projetado para chamadas de operações tipadas, ou seja, que estão associadas a uma quantidade fixa de parâmetros e valores de resultado com tipos específicos. Por essa razão, do GIOP utiliza um formado binário compacto para codificação de valores enviados nas requisições. Para decodificar corretamente esses valores, é necessário conhecer os seus tipos, que devem ser inferidos a partir das informações de tipo associadas à operação sendo chamada. Como conseqüência, o suporte CORBA necessita da extensão da camada de apresentação que introduz suporte para definição das interfaces dos objetos.

Outra diferença importante do protocolo GIOP são os seus múltiplos mapeamentos para diferentes tecnologias de rede. Apesar do OiL atualmente só oferecer suporte para o mapeamento do GIOP para TCP/IP, denominado IIOP (*Internet Inter-ORB Protocol*), a camada do protocolo CORBA é projetada para ser estendida com o suporte para outros mapeamentos.

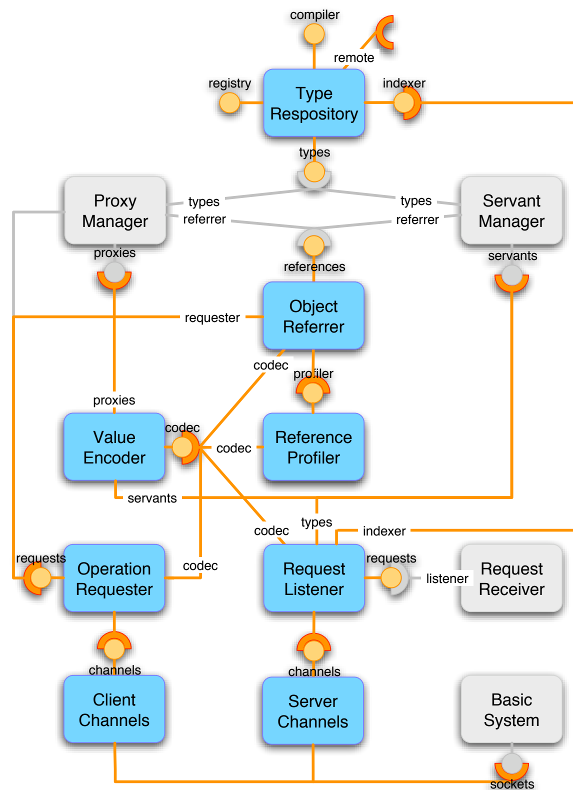
A figura 3.13 ilustra os componentes que compõem esta camada. As mesmas implementações dos componentes `ClientChannels` e `ServerChannels` da camada do protocolo LuDO são utilizadas pela camada do protocolo GIOP. A seguir, são apresentados os demais componentes desta camada.

`TypeRepository` Componente utilizado pela aplicação para definir as interfaces dos objetos do sistema, e também é utilizado pela camada de apresentação para associar informações de interface aos *servants* e *proxies* criados. Esse componente oferece quatro facetas: `compiler` permite registrar no repositório definições de interfaces descritas na linguagem IDL de CORBA; `registry` permite obter descritores das interfaces registradas, que são tabelas contendo informações sobre as interfaces; `types` similar a `registry`, mas permite importar automaticamente as interfaces não encontradas localmente do repositório de interfaces CORBA conectado ao receptáculo `remote`; `indexer` permite obter uma tabela contendo toda descrição de uma operação a partir do nome da operação e o descritor da interface.

`ValueEncoder` Componente que implementa o CDR (*Common Data Representation*), que é o formado binário de codificação de valores utilizado pelo

PUC-Rio - Certificação Digital Nº 0510971/CA

Figura 3.13: Arquitetura da camada do protocolo CORBA.




protocolo GIOP. Esse formato é utilizado para codificação tanto das mensagens trafegadas pela rede, como também para codificação de referências de objeto em *strings*. Por isso, este componente é utilizado por vários componentes desta camada. De forma similar ao componente `ValueEncoder` da extensão `ludo.byref` da camada do protocolo LuDO, este componente define dois receptáculos para acessar as facetas `servants` e `proxies` dos componentes `ServantManager` e `ProxyManager`, que são usadas para criar *servants* implicitamente e automaticamente criar *proxies* para referências de objeto obtidas pela rede. Neste componente, as conexões desses receptáculos é opcional. O receptáculo `servants` deve ser desconectado para desabilitar o suporte a criação implícita de *servants*. Quando o receptáculo `proxies` é desconectado, não há criação implícita de *proxies* e todas as referências de objeto recebidas como parâmetros de chamadas ou valores de retorno são objetos IOR criados pelo componente `ObjectReferrer`.

`ObjectReferrer` Componente utilizado pela camada de apresentação para criar referências de objeto para *servants* locais, assim como codificá-las e decodificá-las em *strings* segundo os formatos de referência textuais definidos no padrão CORBA. As referencias de objeto CORBA são denominadas IOR (*Interoperable Object Reference*) e são compostas por diferentes componentes

PUC-Rio - Certificação Digital Nº 0510971/CA

denominados *profiles*. Os *profiles* são informações codificadas uma seqüência de bytes usando o formato CDR. Cada *profile* possui um número de identificação denominado *tag*, que indica o tipo das informações contidas nele. O IOR também contém uma indicação da interface do objeto referenciado. Para criar IORs, este componente deve ser primeiramente configurado com as informações do ponto de acesso criado pelo componente `RequestListener` na iniciação do ORB. Com base nas informações desse ponto de acesso, o `ObjectReferrer` utiliza os componentes conectados ao receptáculo `profilers` para criar os *profiles* que vão compor o IOR. Os IOR são objetos que oferecem operações para decodificação dos *profiles*, que é feita através dos componentes conectados ao receptáculo `profilers`.

`ReferenceProfiler` São componentes utilizados pelo `ObjectReferrer` para criar e decodificar as informações nos *profiles* IOR. Tipicamente, cada `ReferenceProfiler` é capaz de criar e decodificar *profiles* de um mapeamento GIOP diferente. O OiL oferece apenas uma implementação desse componente, que suporta *profiles* IIOP.

`OperationRequester` Desempenha o mesmo papel do componente de mesmo nome da camada do protocolo LuDO, porém utilizando o protocolo GIOP. O receptáculo `codec` é utilizado para codificação e decodificação das mensagens GIOP transmitidas pela rede. Nas chamadas feitas através desse componente, a identificação da operação sendo chamada deve ser uma tabela contendo informações sobre a operação, como nome, tipo dos parâmetros e valores de retorno, etc., como fornecido pela faceta `indexer` do componente `TypeRepository`. Essas informações são utilizadas para converter os valores fornecidos na chamada nos tipos dos parâmetros da função sendo chamada e codificá-los adequadamente. A assinatura da operação também é necessária para decodificação dos valores codificados na resposta da requisição. O receptáculo `channels` é um receptáculo que aceita múltiplas conexões simultâneas. Cada objeto conectado é identificado por um número, que indica o *tag* do *profile* IOR a partir do qual ele é capaz de criar canais. Dessa forma, sempre que a primeira requisição é feita através de uma referência, o `OperationRequester` consulta os *profiles* disponíveis na referência e as fábricas de canais disponíveis através desse receptáculo, para decidir qual tipo de canal de comunicação será estabelecido para realização da chamada.

`RequestListener` Desempenha o mesmo papel do componente de mesmo nome da camada do protocolo LuDO. O receptáculo `codec` é utilizado para

PUC-Rio - Certificação Digital Nº 0510971/CA

codificação e decodificação das mensagems GIOP transmitidas pela rede. O receptáculo `servants` deste componente é conectado à faceta de mesmo nome do componente `ServantManager`. Quando uma requisição é recebida da rede, esse receptáculo é usado para obter informações dos *servants* registrados localmente, em particular, o descritor da interface associada. De posse do descritor da interface, o `RequestListener` utiliza a faceta `indexer` do componente `TypeRepository` para obter a descrição da operação sendo chamada. Com base nessa descrição, é possível obter o tipo dos valores codificados na mensagem e decodificá-los corretamente. Assim como também é possível converter os valores Lua do resultado da chamada nos tipos dos valores de resultado definidos na assinatura da operação, de forma que sejam codificados adequadamente. De forma similar ao componente `OperationRequester`, o receptáculo `channels` do `RequestListener` aceita a conexão com múltiplas fábricas de canais, onde cada conexão é identificada por um *tag* que indica o mapeamento do GIOP que a fábrica dá suporte. Dessa forma, esse componente é capaz de criar pontos de acesso, e aceitar conexões e requisições usando o suporte de canais de diferentes mapeamentos do GIOP.

PUC-Rio - Certificação Digital Nº 0510971/CA

### 3.5.1 Extensão para Interceptação

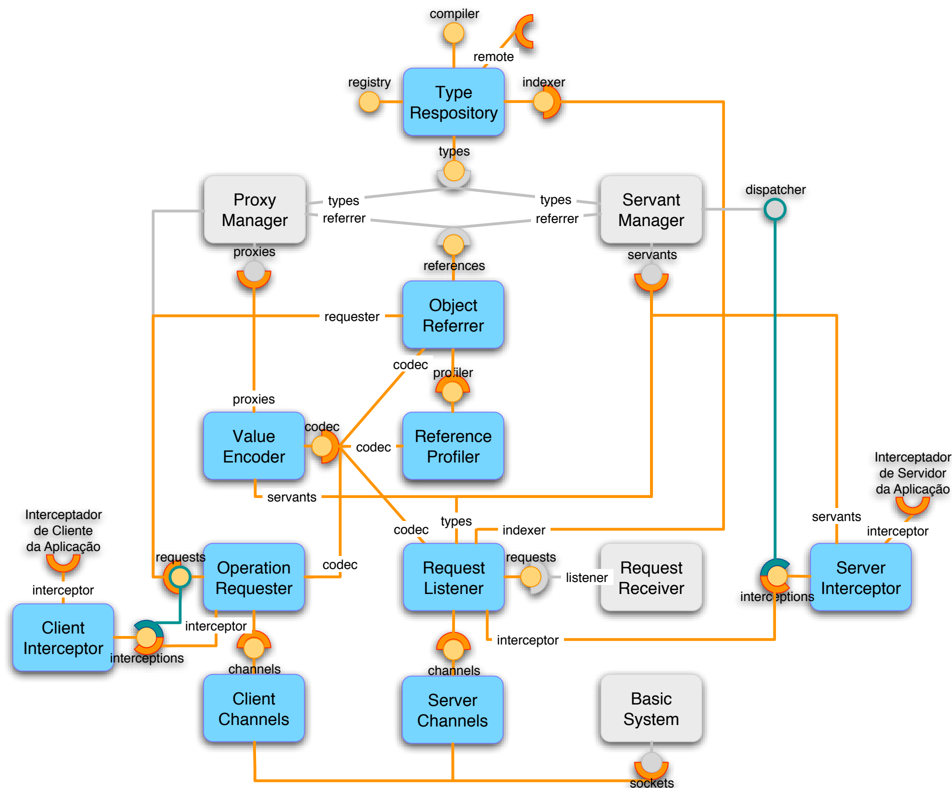
As mensagens do GIOP oferecem campos especiais para inclusão de informações adicionais nas requisições. Tipicamente, esses campos podem ser acessados através de interceptadores de chamada. Contudo, a implementação de interceptação de chamadas do OiL não oferece acesso a esses campos porque se baseia na interceptação de interações de componentes da camada básica, onde detalhes do protocolo de RMI não são acessíveis.

Para ativar o suporte aos campos das mensagens GIOP nos interceptadores de chamada, o OiL estende a camada do GIOP de forma que a codificação e decodificação das mensagens do GIOP possam ser interceptadas também. Entretanto, isso não pode ser feito através de interceptação de portas de componentes como é feito no suporte na camada de apresentação, pois toda a codificação e decodificação de mensagens é feita internamente nos componentes `OperationRequester` e `RequestListener`. Para permitir essa interceptação, implementações alternativas desses dois componentes são utilizadas, sendo que elas disponibilizam um receptáculo adicional onde um interceptador deve ser registrado para interferir na codificação e decodificação das mensagens. Para evitar que o desenvolvedor da aplicação tenha que lidar com dois mecanismos de interceptação (*i.e.* chamadas e codificação de mensagens), o OiL implementa um componente adicional que integra os dois mecanismos em um só como ilus-

trado na figura 3.14. Esse mecanismo integrado de interceptação permite inspecionar e interceder tanto na realização e despacho de chamadas, assim como a codificação das mensagens GIOP, como apresentado na seção 3.5.1.

O uso de um componente para realizar a interceptação também permite que ele se conecte a outros componentes para obter informações adicionais a serem fornecidas à aplicação no momento da interceptação. Por exemplo, o interceptador de mensagens GIOP se conecta à faceta `servants` do `ServantManager` para obter a interface e a implementação do *servant* ao qual a requisição se destina.

Figura 3.14: Arquitetura da chamada CORBA com interceptação de chamadas.




### 3.5.2
### Extensão para Marshaling Otimizado

Parte da sobrecarga de processamento nas chamadas de operações remotas feitas pelo OiL está relacionada à inspeção dinâmica das informações de tipo das interfaces descritas em IDL. Ou seja, sempre que uma chamada de operação é feita, o ORB consulta a descrição dos tipos de cada parâmetro e valor de retorno da operação para poder codificar e decodificar os valores adequadamente. Como essas descrições são estruturas dinâmicas (tabelas), essa inspeção implica em uma sobrecarga de processamento, mesmo que esses tipos raramente mudem durante a execução do programa. A figura 3.15 mostra a

PUC-Rio - Certificação Digital Nº 0510971/CA

a implementação da decodificação de uma `struct` de CORBA utilizada OiL, onde a descrição da estrutura da *struct* é dada pelo parâmetro `idltype` (linha 1). Esse parâmetro é inspecionado para consultar os nomes de cada campo da `struct`, assim como o tipo dos seus valores (linhas 3–4). Note, que mesmo que a estrutura da `struct` nunca mude, essa inspeção é feita em cada codificação de valor desse tipo.

Figura 3.15: Decodificação iterativa de uma `struct` de CORBA.

```
1 function Decoder:struct(idltype)
2   local value = {}
3   for _, field in ipairs(idltype.fields) do
4     value[field.name] = self:get(field.type)
5   end
6   return value
7 end
```

Uma alternativa a essa abordagem é utilizar uma técnica similar à utilizada por implementações de CORBA em linguagens estáticas, em que a implementação da codificação dos valores é feita por um código gerado especificamente para uma dada definição da interface. O suporte à interpretação de código das linguagens dinâmicas permite que esse código seja gerado dinamicamente, a medida que valores com tipos diferentes sejam codificados, similar ao que é proposto em (Aktemur et al., 2005). Para avaliar o possível impacto no desempenho que essa otimização poderia trazer, desenvolvemos um protótipo desse suporte através de uma extensão da camada do protocolo GIOP, denominada `gencode`. Essa extensão basicamente substitui a implementação do componente `ValueEncoder` por uma que gera código de *marshaling* especializado para cada novo tipo de dado codificado pelo ORB.

A figura 3.16 ilustra parte da implementação do decodificador de valores baseado em geração dinâmica de código utilizado pela extensão `gencode`. A classe `DecoderGen` é utilizada para gerar o código do decodificador e compilá-lo. A operação `struct` (linha 3) ilustra a geração do código de decodificação de uma `struct`. A estrutura da `struct` é inspecionada (linha 6) para que o código gerado, quando executado, decodifique os campos da ordem correta. A decodificação de cada campo da `struct` é incluída no código gerado, recursivamente (linha 8). Quando um valor é decodificado pela operação `get`, o tipo do valor é informado pelo parâmetro `type` (linha 16). Se o tipo ainda não definir uma função de decodificação (linha 18) então uma nova função deve ser gerada para aquele tipo específico. Se a classe `DecoderGen` oferecer suporte para geração de decodificadores daquele tipo (linha 20), então uma nova instância da classe é criada para gerar o decodificador otimizado (linhas 21–23). Caso contrário, o decodificador iterativo é usado, que é herdado

PUC-Rio - Certificação Digital Nº 0510971/CA

da classe original que implementa o suporte de decodificação de CORBA no OiL (linha 25). Por fim, a função de decodificação escolhida é associada ao tipo para que os próximos valores daquele mesmo tipo sejam decodificados usando a mesma função (linha 27).

Figura 3.16: Geração de decodificador de uma `struct` de CORBA.

```
1  -- TRECHO OMITIDO
2
3  function DecoderGen:struct(type)
4    self:add("{\n")
5    local fields = type.fields
6    for _,field in ipairs(fields) do
7      self:add(field.name,"=")
8      self:generate(field.type)
9      self:add(",\n")
10   end
11   self:add("}\n")
12 end
13
14 -- TRECHO OMITIDO
15
16 function Decoder:get(type)
17   local decode = type.decode
18   if decode == nil then
19     local type = type._type
20     if DecoderGen[type] then
21       local gen = DecoderGen()
22       gen:generate(type)
23       decode = gen:compile(type)
24     else
25       decode = self[type]
26     end
27     type.decode = decode
28   end
29   return decode(self, type)
30 end
```

A figura 3.17 ilustra o código equivalente ao decodificador gerado para a `struct Person` ilustrada na figura 3.18. A implementação atual da geração de código de *marshaling* ainda é um protótipo e apresenta várias limitações importantes. Em particular, não é possível gerar código para estruturas de dados recursivas. Outras otimizações são possíveis de serem implementadas na geração de código, como a redução do número de chamadas de função através da substituição da chamada pelo corpo da função apropriadamente expandido.

Figura 3.17: Decodificador especializado de uma `struct` de CORBA.

```
1  function DecodePerson(self)
2    local data = self.data
3    return {
4      name = data:sub(
5        self:jump(unpack('L',data,self:alignedjump(4))),
6        self.cursor-2),
7      age = unpack('L',data,self:alignedjump(4)),
8      address = {
9        street = data:sub(
10         self:jump(unpack('L',data,self:alignedjump(4))),
11         self.cursor-2),
12       number = unpack('L',data,self:alignedjump(4)),
13       postalcode = unpack('L',data,self:alignedjump(4)),
14       city = data:sub(
15         self:jump(unpack('L',data,self:alignedjump(4))),
16         self.cursor-2),
17       country = data:sub(
18         self:jump(unpack('L',data,self:alignedjump(4))),
19         self.cursor-2),
20     },
21   }
22 end
```

Lua também permite a geração de código executável a partir de *opcodes* de Lua, que são as instruções da máquina virtual de Lua. Contudo, Lua

PUC-Rio - Certificação Digital Nº 0510971/CA

Figura 3.18: IDL utilizada para ilustrar a geração de código de *marshaling*.

```
1 struct Address {
2   string street;
3   unsigned long number;
4   unsigned long postalcode;
5   string city;
6   string country;
7 };
8 struct Person {
9   string name;
10  unsigned long age;
11  Address address;
12 };
```

não oferece mecanismos de suporte para geração e manipulação de *opcodes*, inclusive o formato dos *opcodes* não é padronizado, sendo um detalhe de implementação que pode mudar entre versões de lançamento de Lua.

## 3.6 Problemas Encontrados

As principais motivações para a organização do OiL em componentes foi facilitar a introdução de novos recursos e a modificação de recursos já oferecidos, uma vez que a modificação do OiL em versões anteriores se mostrou difícil em algumas situações. Dois casos podem ser citados nesse sentido, a introdução de mecanismos de travessia de *firewall*/NAT usando o protocolo GIOP (Theophilo et al., 2005) e a introdução de suporte a interceptação de chamadas (Oliveira Neto, 2008). Em ambos os casos, a introdução desses recursos exigiu diferentes modificações no código do OiL, que resultaram em implementações mais complexas, incompatíveis e com funcionalidades que são desnecessárias em algumas aplicações. A modificação de recursos, como o protocolo de RMI utilizado nas chamadas ou a forma de realização de uma chamada de operação para permitir chamadas assíncronas e protegidas, também se mostrou difícil de ser acomodada nas versões iniciais do OiL.

Como resultado disso, foi realizado um estudo para a organização da implementação do OiL de forma que pudesse acomodar modificações mais facilmente (Nogara, 2006), que posteriormente resultou na arquitetura baseada em componentes do OiL apresentada neste trabalho. O uso de componentes de software se mostrou efetivo no sentido de propiciar uma melhor modularização do software e facilitar modificações, visto que a substituição de partes do software ou a mudança nas interações entre essas partes pode ser feita em poucos passos através de modificações na montagem do sistema e do reuso de componentes. Um exemplo disso foi a reimplementação de um dos mecanismos de travessia de *firewall*/NAT na versão baseada em componentes do OiL (Fusco, 2007), que pôde então ser feita através da implementação de novos componentes adicionados à montagem do ORB de acordo com a aplicação, sem necessidades de alterações no resto da implementação do OiL. Outros exemplos incluem o suporte a *proxies* alternativos, diferentes protocolos e interceptação

PUC-Rio - Certificação Digital Nº 0510971/CA

de chamadas apresentados neste capítulo.

Por outro lado, o uso de componentes também trouxe alguns problemas, em particular, a dificuldade de navegação na implementação, pois os relacionamentos entre os componentes não são claros, uma vez que eles só são definidos na criação do ORB pela aplicação. Essa dificuldade foi particularmente problemática na utilização do OiL por alunos que introduziram modificações na implementação. Outro problema do uso de componentes é a necessidade da aplicação montar os componentes que formam o ORB de forma correta. As montagens pré-definidas no OiL permitem contornar esse problema, pois através delas é possível montar automaticamente configurações de ORB recorrentes. Além disso, outras montagens dos componentes podem ser feitas diretamente pela aplicação quando necessário.

Um problema particular da implementação de componentes do OiL, é a complexidade da estrutura imposta pelos *templates* do LOOP. Esses *templates* facilitam a construção de componentes formados por múltiplos objetos, que podem ser tabelas ou *userdata*, e também oferecem funcionalidades como interceptação de portas e outros mecanismos de intercessão nos componentes. Contudo, essas funcionalidades foram pouco utilizadas na implementação dos componentes do OiL. Além disso, a utilização dos *templates* requer um aprendizado do seu funcionamento e do modelo de implementação de componentes que eles impõem. Portanto, apesar de ser um detalhe de implementação, o uso de *templates* dificulta o entendimento da implementação e oferece poucos ganhos na implementação do OiL. Uma linha de investigação futura deste trabalho consiste em definir modelos de componentes mais simples e que facilitem o entendimento da implementação do OiL.

Como resultado disso, temos que, em experimentos isolados, a implementação baseada em componentes do OiL apresentou algumas facilidades para introdução de modificações, mas também apresentou algumas dificuldades. Com base nisso, decidimos realizar um estudo mais detalhado dessa implementação para identificar mais claramente as características que contribuem ou comprometem sua flexibilidade, e particularmente como as características de uma linguagem dinâmica como Lua influenciam nisso. O próximo capítulo apresenta uma avaliação da flexibilidade do OiL com base nesse estudo, onde alguns aspectos da implementação do OiL são examinados em maior detalhe, tomando por base a implementação de algumas funcionalidades específicas e identificando aspectos do OiL que facilitam ou dificultam essa tarefa.

PUC-Rio - Certificação Digital Nº 0510971/CA