4
Stubs Dinâmicos

Uma proposta alternativa na direção da adaptação dinâmica é o *upload* e o *download* de stubs, descritos neste capítulo. Eles constituem em uma aplicação específica de mobilidade de código, implementando alguns paradigmas descritos em [13]. O *download* de stubs pode ser classificado como uma variante de *código sob demanda*. O LuaProxy detecta a existência desses stubs e os obtém automaticamente, permitindo-os encapsular os métodos do proxy no lado do cliente. O paradigma de *avaliação remota*, por sua vez, consiste na transferência de código para ser executado em outro nó da rede. O *upload* de stubs usa essa estratégia para estender a funcionalidade do servidor ou modificar o comportamento de seus métodos.

O uso pretendido para esses stubs é o de auxiliar na tarefa da adaptação dinâmica, o que pode ser feito usando-os em conjunto com o LuaProxy ou independentemente. Por exemplo, uma implementação de QoS poderia fazer *upload* de código para compactar grandes seqüências de dados na escassez de largura de banda; ou um cliente que não usa o LuaProxy poderia baixar um stub simplesmente por ele implementar um mecanismo de *cache*.

De modo a facilitar a disponibilização e o uso desses recursos, algumas funções do LuaOrb foram redefinidas, como a *luaorb.createservant*. Conforme discutido mais adiante, às vezes é necessário que um servant implemente várias interfaces ao mesmo tempo, o que requer a criação de uma interface que herde de todas as demais. Isso agora é feito automaticamente, facilitando a disponibilização e o uso dos stubs.

```
module LuaProxy
{
   enum LangEnum {LUA, C, JAVA};
   typedef sequence<octet> OctetSeq;

   struct Stub
   {
      LangEnum Language;
      OctetSeq Encoding;
   };

   interface DStubProvider
   {
      Stub getStub (in LangEnum lang);
   };
};
```

Figura 4.1: IDL para disponibilizar *downloadable stubs*.

4.1 Download de Stubs

Apesar de o LuaProxy permitir o uso de procedimentos de adaptação personalizados (implementações de QoS), alguns servidores específicos podem oferecer stubs especializados, que são baixados pelos clientes (*download*) em tempo de execução. Eles podem implementar rotinas de adaptação próprias ou simplesmente versões modificadas dos métodos do servidor, tirando proveito de detalhes internos de implementação.

Para disponibilizar esses stubs, doravante denominados *dstubs* (em alusão ao termo em inglês *downloadable stubs*), o servant precisa implementar a interface *LuaProxy::DStubProvider*, parcialmente listada na Figura 4.1. Como nem todas as instâncias de um determinado serviço podem precisar desse recurso, a criação explícita de uma interface que herde da *LuaProxy::DStubProvider* pode não ser conveniente; nesses casos a nova versão da função *luaorb.createservant* oferece uma alternativa mais cômoda.

Conforme mostrado na Figura 4.1, os dstubs são estruturas simples, compostas de apenas dois campos: uma seqüência "crua" de octetos, contendo o código do dstub, e a linguagem na qual o dstub foi implementado. Em sua versão atual, este trabalho suporta apenas dstubs escritos em Lua, mas a possibilidade de comunicação de Lua com C e Java permitiria a utilização de dstubs escritos nessas linguagens no futuro.

A inicialização típica de dstubs consiste em criar um proxy para o objeto remoto, detectar se ele suporta dstubs, chamar o método *getStub* e integrar o código do dstub ao próprio proxy. Um objeto remoto suporta dstubs se, e somente se, sua interface deriva de *LuaProxy::DStubProvider*. Para verificar se um servant implementa essa interface, usa-se a pseudo-operação CORBA *_is_a*, passando como argumento o id *IDL:LuaProxy/DStubProvider:1.0*.

Uma vez obtido, o dstub pode ser combinado ao proxy pelo método *DownloadableStub.mergeProxy*, resultando em uma estrutura semelhante à da Figura 4.2. Baseado na linguagem em que o dstub foi implementado, seu código pode ser tratado como uma string de código Lua, gravado em arquivo e carregado como biblioteca dinâmica C, etc. Naturalmente, a execução desse código inspira cuidados relativos à segurança, como o uso de *sandboxing* [14], mas essa discussão foge do escopo desta dissertação.

A partir do momento em que for instalado, o dstub pode interceptar todos os métodos chamados sobre o proxy (como f e g na Figura 4.2) e optar por redirecionar a chamada de método diretamente ao objeto remoto (campo especial *_sproxy*) ou executar um processamento antes ou depois da chamada. Isso possibilita criptografar ou compactar a transmissão de dados, implementar algum mecanismo de cache ou participar da adaptação dinâmica, seja por conta própria ou interagindo com o LuaProxy (discutido na próxima seção).

De modo a facilitar o uso dos dstubs, as funções *luaorb.createproxy* e *luaorb.narrow* foram redefinidas para detectarem a presença deles e, em caso afirmativo, obtê-los e integrá-los automaticamente. O exemplo das Figuras 4.3 e 4.4 mostram como disponibilizar o dstub.

4.1.1

Interagindo com o LuaProxy

Apesar de os dstubs serem auto-suficientes para implementarem seus próprios mecanismos de adaptação, o LuaProxy foi projetado para poder interagir com o dstub quando este estiver disponível. Ao selecionar uma oferta de serviço no trader, o LuaProxy armazena uma referência para si mesmo (*self*) no campo *_sproxy* do proxy da oferta.

Baseado na existência ou não desse campo especial, o dstub pode determinar se está sendo usado por uma instância do LuaProxy. Nesse caso, ele pode combinar seus próprios mecanismos de adaptação com os do LuaProxy.

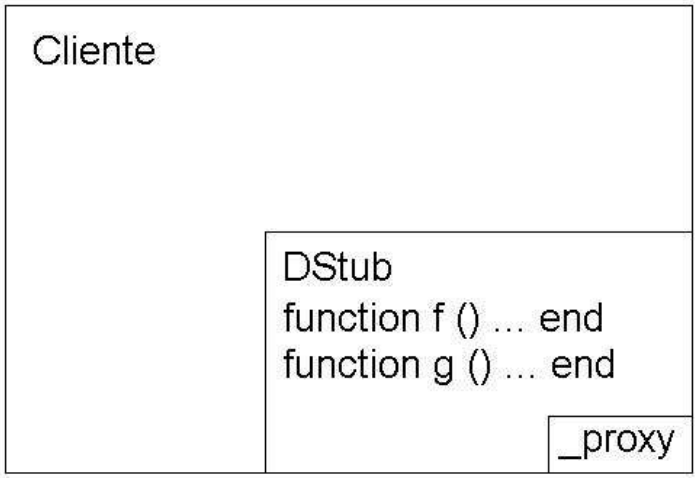

Figura 4.2: Proxy encapsulado por um dstub.


```
MyServant = {
   foo = function (self)
      print ("Dentro de foo ()")
   end,

   getStub = function (self, lang)
      return {
         Language = "LUA",
         Encoding = [[{
            foo = function (self)
               print ("Dentro do dstub.")
               self._proxy:foo ()
            end
         }]]
      }
   end
}
```

Figura 4.3: Exemplo de uso do dstub, lado do servidor.

```
MyProxy = luaorb.createproxy ("IOR:...")
MyProxy:foo ()
-- Cliente imprime: "Dentro do dstub"
-- Servidor imprime: "Dentro de foo ()"
```

Figura 4.4: Exemplo de uso do dstub, lado do cliente.

Para possibilitar isso, ao detectar que o proxy contém um dstub, o LuaProxy permite que ele redefina seus métodos internos.

Dentre as muitas possibilidades, o dstub pode redefinir o método *_adapt* para que seus mecanismos de adaptação tenham precedência sobre os da implementação de QoS do usuário. Se o dstub não conseguir realizar a adaptação por si só, ele ainda tem a opção de chamar a versão do método *_adapt* nativa do LuaProxy. Outra aplicação seria redefinir o método *_getOffer*, usado pelo LuaProxy para encontrar um novo servidor: quando o *_getOffer* do dstub fosse chamado, ele poderia usar a versão nativa do LuaProxy para fazer o trabalho pesado e, de posse do novo servidor, transferir o estado do processamento para ele antes de retorná-lo para o LuaProxy.

4.2 Upload de Stubs

No contexto de aplicações distribuídas, o retardo na comunicação cliente-servidor em si tipicamente constitui a maior parcela do tempo de execução de um método remoto. Em situações desfavoráveis, diversas chamadas de método a um mesmo objeto podem ser necessárias para se concluir uma tarefa. Por exemplo, uma consulta em um banco de dados remoto pode gerar diversas tabelas intermediárias que, após combinadas, poderiam produzir um resultado final de poucas linhas.

Uma possível solução para esse problema seria executar parte do processamento no próprio servidor. Para o exemplo da consulta ao banco de dados, o padrão SQL oferece as *stored procedures*. Elas são rotinas transferidas do cliente para o servidor, onde podem acessar as tabelas localmente e transferir pela rede somente o resultado final da consulta. Isso economiza a largura de banda e o tempo de transmissão relativos aos resultados parciais, nos quais o cliente normalmente não está interessado.

Inspirado em situações desse tipo, e explorando mais uma vez a natureza interpretada da linguagem Lua, esta seção apresenta os *uploadable stubs* (ou, simplesmente, *ustubs*). Além de proporcionarem ao programador os benefícios mencionados no parágrafo anterior, os ustubs permitem, sob um certo ponto de vista, estender a interface do servidor, disponibilizando suas funcionalidades extras na forma de métodos visíveis a outros clientes. Opcionalmente, o ustub também pode redefinir ou encapsular métodos do servidor, oferecendo uma

```
module LuaProxy
{
   interface ResultReceiver
   {
      oneway void receiveResult (in any result);
   };

   interface CallbackStub
   {
      oneway void invoke (in string opname,
         in ResultReceiver rec, in AnySeq args);
   };

   interface UStubHolder
   {
      any insertTempStub (in Stub stub);
      oneway void insertTempStubCB (in Stub stub,
         in ResultReceiver rec);

      long insertStub (in Stub stub, in string ifname,
         out Object extobj);

      void removeStub (in long id);
   };
};
```

Figura 4.5: IDL para *upload* de stubs.

abordagem adicional para o problema da adaptação dinâmica.

A diversidade das possíveis aplicações para os ustubs faz com que eles exibam características diferentes entre si. Especificamente, este trabalho leva em consideração dois critérios: *tempo de vida* e *modo de retorno*.

O tempo de vida de um ustub determina por quanto tempo ele será mantido no servidor. Quanto a esse critério, os ustubs podem ser *temporários* ou *persistentes*. Ustubs temporários são removidos do servidor assim que terminam de executar; caso o cliente precise do ustub novamente será necessário reenviá-lo. Os ustubs persistentes, por outro lado, permanecem no servidor até serem explicitamente removidos.

Em relação ao modo de retorno, ele pode ser *imediato* ou *por callback*. A única diferença é que no primeiro caso o cliente fica bloqueado enquanto o ustub executa no servidor; quando se obtém os resultados por *callbacks* o cliente pode conduzir outra tarefa em paralelo.

```
interface Matrix
{
   double getAij (in unsigned long i, in unsigned long j);
   long getn ();
};
```

Figura 4.6: Interface *Matrix*.

A Figura 4.5 mostra as interfaces relevantes ao *upload* de stubs. O servant interessado em disponibilizá-los deve implementar a interface *LuaProxy::UStubHolder*. A maneira mais fácil de fazer isso é usar o meta-método *__index* para herdar esses métodos da tabela *UploadableStub*, incluída no pacote LuaProxy. A criação do servant pode ser feita com a nova versão da função *luaorb.createservant*, de modo a evitar a criação manual de uma nova interface.

4.2.1
Ustubs Temporários

Ustubs temporários possuem um tempo de vida muito curto. Essencialmente eles consistem em um trecho de código que é transferido ao servidor, executado e logo após removido, retornando seu resultado ao cliente. Eles são muito úteis quando o código a ser executado acessa freqüentemente um mesmo objeto remoto. Ao se transferir esse código ao servidor em questão, elimina-se a maior parte dos custos da comunicação remota.

Para instalar um ustub temporário em um servidor, usa-se o método *insertTempStub*. O ustub tem a mesma estrutura discutida na Seção 4.1, mas seu código precisa definir pelo menos o método *_run*, cujo único argumento é a referência para si mesmo (*self*). Quando o ustub é instalado, o método *insertTempStub* faz com que ele herde do objeto remoto ao qual ele está ligado, de modo que o ustub possa acessar seus métodos. A seguir, o método *_run* é executado e os resultados retornados ao cliente.

Para ilustrar esse procedimento, consideremos um objeto que implemente a interface *Matrix* (mostrada na Figura 4.6) e que suporte ustubs. O método *getAij* permite ler da matriz o elemento da posição (i, j), mas se o cliente quiser obter a diagonal principal ele teria de chamar *getAij* n vezes. A Figura 4.7 mostra como instalar um ustub temporário para se obter essa diagonal com uma única chamada remota, onde p é um proxy para a matriz remota.

```
ustub = {
   Language = "LUA",
   Encoding = [[
      _run = function (self)
         local diag = {}
         for i = 1, self:getn () do
            table.insert (diag, self:getAij (i, i))
         end
         return diag
      end
   ]]
}

result = p:insertTempStub (ustub)
```

Figura 4.7: Instalando um ustub temporário

Uma vantagem do ustub temporário é sua versatilidade, pois o cliente não precisa se preocupar em removê-lo após usá-lo (isso é feito automaticamente). Se o ustub precisar ser reutilizado, é preferível instalar uma versão persistente dele (discutido na Seção 4.2.2) a instalar uma versão temporária a cada utilização. Os ustub temporários são apropriados para tarefas específicas, conhecidas *à priori*, pois seu método *_run* não aceita argumentos.

O método *insertTempStub* só retorna quando o ustub terminar seu processamento. Como isso pode demorar muito tempo, o usuário pode preferir receber os resultados via *callback*. Para isso, usa-se o método *insertTempStubCB*, que não é bloqueante, mas requer um objeto que implemente a interface *LuaProxy::ResultReceiver* (Figura 4.5). Essa interface define um único método, *receiveResult*, pelo qual o objeto recebe o valor retornado pelo ustub.

A Figura 4.8 mostra como reescrever o código da Figura 4.7 com o uso de *callback*. Nela, as variáveis *p* e *ustub* são idênticas às do exemplo anterior. Isso significa que o ustub não "sabe" se o seu resultado será retornado diretamente a quem o instalou ou por *callback*. Isso permite ao mesmo ustub ser usado de ambas as formas sem sofrer modificações.

4.2.2

Ustubs Persistentes

Ao contrário dos ustubs temporários, os persistentes permanecem no servidor até serem explicitamente removidos. Uma vez instalado, o ustub pode

```
cb = {
   resultado = nil,
   receiveResult = function (self, result)
      self.resultado = result
   end
}

p:insertTempStubCB (ustub, cb)

-- Espera o resultado
while not cb.resultado do
   -- Faz outra coisa enquanto isso
   ...
end
```

Figura 4.8: Instalando um ustub temporário com *callback*

interceptar as chamadas de métodos no servidor e executar qualquer pré ou pós-processamento necessário.

Existem várias opções de como se implementar ustubs persistentes. Uma delas seria a de usar servidores genéricos [15], que permitem ao programador tanto modificar a interface do objeto remoto como construir as chamadas de método manual e dinamicamente. Este trabalho adota uma abordagem mais simples, que consiste em usar o ustub persistente como uma simples porta de acesso alternativa, deixando o objeto remoto intacto.

É normal que ustubs persistentes incluam novos métodos ao objeto ao qual se acoplam, resultando em um *objeto estendido* (Figura 4.9). Para tornar essas extensões visíveis aos seus clientes, é necessário que o novo objeto implemente a interface que define os métodos adicionais. Ao mesmo tempo, ele precisa implementar a interface do objeto original, de modo a manter sua funcionalidade.

O método *insertStub* é responsável pela instalação de ustubs persistentes. Ele recebe o ustub e a interface das extensões como argumentos. Caso o cliente não deseje adicionar nenhuma extensão ao servidor (apenas encapsular seus métodos), este último argumento deve ser a string vazia (“”). Nesse caso sua interface será a mesma do objeto original. No lado do servidor, o código do ustub é traduzido em uma tabela cujo meta-método *__index* se refere ao objeto original. Com efeito, o objeto estendido consiste em uma referência ao original encapsulada pelo ustub.

O exemplo da obtenção da diagonal principal de uma matriz é revisto

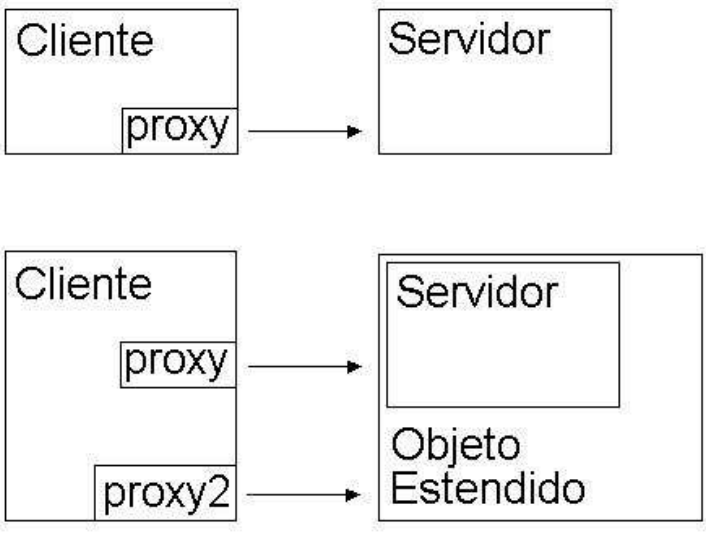

Figura 4.9: Servidor antes e depois da instalação do ustub.

```
interface NewMethods
{
   typedef sequence<double> DoubleSeq;
   DoubleSeq getMainDiagonal ();
};
```

Figura 4.10: Exemplo de extensão de interfaces.

aqui. A Figura 4.11 mostra como estender a interface *Matrix* (Figura 4.6) com o método *getDiagonal*, da interface *NewMethods* (Figura 4.10). Neste exemplo, *p* é um proxy para a matriz remota, e *obj*, retornado pela *insertStub*, é a referência para o objeto estendido criado no servidor. Ela precisa ser convertida (*narrowed*) para sua interface mais específica antes de ser usada, com a função *luaorb.narrow*.

Há alguns detalhes importantes sobre o procedimento descrito na Figura 4.11. Primeiro, assumiu-se que o objeto remoto suporta o *upload* de stubs; na prática seria necessário usar a pseudo-operação CORBA *_is_a* para detectar se o objeto remoto implementa a interface *LuaProxy::UStubHolder*. Segundo, o ustub usa o meta-método *__index* para acessar campos do objeto que ele encapsula, como nas expressões *self:getn* e *self:getAij*. Terceiro, a interface do objeto estendido herda da do servidor original, o que permite ao cliente usá-lo para chamar tanto o novo método *getMainDiagonal* quanto os do objeto original, como o *getAij*.

Como efeito colateral, o objeto estendido também implementaria a interface *LuaProxy::UStubHolder*. Isso permitiria ao cliente instalar outro ustub

```
ustub = {
   Language = "LUA",
   Encoding = [[
      getMainDiagonal = function (self)
         local diag = {}
         for i = 1, self:getn () do
            table.insert (diag, self:getAij (i, i))
         end
         return diag
      end
   ]]
}

id, obj = p:insertStub (ustub, "NewMethods")
obj = luaorb.narrow (obj)
diag = obj:getMainDiagonal ()
p:removeStub (id)
```

Figura 4.11: Usando o objeto estendido.

nele, fazendo um cascateamento. Nesse caso, o objeto estendido resultante do primeiro *upload* será usado como se fosse um objeto qualquer para a segunda chamada do método *insertStub*.

Além da referência para o objeto estendido, o método *insertStub* retorna o *id* do ustub no servidor. Quando o cliente não precisar mais do ustub, ele deve removê-lo do servidor com o método *removeStub*. O único argumento desse método é o id do ustub em questão.

Os ustubs persistentes também podem retornar seus resultados por *callback*, mas este trabalho não obriga o programador a adotar nenhum procedimento específico para isso. Como sugestão, pode-se fazer o ustub implementar a interface *LuaProxy::CallbackStub* (Figura 4.5), que define um método *invoke*. Ele recebe o nome do método a ser chamado, seus argumentos e o objeto de *callback* que deve receber os valores de retorno.

A principal aplicação pretendida para os ustubs é a de auxiliar na tarefa da adaptação dinâmica. As possibilidades incluem benefícios similares aos oferecidos pelos dstubs, mas, como a funcionalidade do objeto remoto pode ser estendida, mecanismos como compactação e criptografia podem ser empregados sem que o servidor sequer suspeite do que esteja sendo feito.

Para ilustrar a aplicação dessa técnica, consideremos uma implementação de QoS que adapte à escassez de largura de banda por compactação dos dados

transmitidos. Ao ser chamada, e ao detectar suporte a ustubs, essa rotina de adaptação encapsularia o proxy do cliente para que os dados enviados ao servidor fossem compactados primeiro. Adicionalmente, enviaria um ustub que descompactasse os dados recebidos antes que fossem repassados adiante. Dessa forma, é possível abordar a diminuição da largura de banda disponível sem depender apenas da implementação original do servidor.

4.2.3
Comparando os Ustubs

Uma vez apresentados os dois tipos de ustubs (temporário e persistente), convém cogitar qual deles seria mais apropriado para cada situação. A rigor, qualquer atividade desempenhada por um ustub temporário poderia ser executada por um ustub persistente. Para isso, basta colocar o código do ustub temporário dentro de um método de um ustub persistente, instalá-lo e chamar esse método. Finalmente, deve-se remover o ustub manualmente quando ele não for mais necessário.

O procedimento do parágrafo anterior, entretanto, requer pelo menos três chamadas remotas para concluir seu trabalho: uma para instalar o ustub, outra para chamar o método desejado e, finalmente, mais uma para desinstalar o ustub. Se o código transferido ao servidor precisar executar uma única vez, um ustub temporário é a opção mais eficiente. Além disso, um ustub persistente requer remoção explícita, sob pena de desperdício de recursos no servidor. Com ustubs temporários não existe o risco de se esquecer de desinstalá-los, pois isso é feito automaticamente.

Por outro lado, somente um ustub persistente pode redefinir métodos no servidor ou estender sua interface. Combinado com a implementação de QoS do lado do cliente, o ustub persistente constitui uma camada adicional na comunicação cliente-servidor, oferecendo uma opção adicional para se conseguir adaptação dinâmica.

4.3 Gerenciador de Interfaces

Conforme discutido nas seções anteriores, o objeto que deseja suportar o *download* ou o *upload* de stubs precisa implementar as respectivas interfaces. Nessas situações, é necessário criar uma nova interface que herde de várias outras, o que pode ser feito de duas maneiras. Uma delas seria especificá-la em um arquivo IDL e carregá-la no repositório de interfaces usando um utilitário próprio para isso, incluído na distribuição do ORB. Além de ser impraticável criar um arquivo IDL para cada possível combinação de interfaces, essa abordagem normalmente é usada antes de a aplicação executar, e depende do conhecimento prévio das interfaces que serão usadas.

A outra opção seria criar a interface diretamente no repositório, usando reflexividade. A complexidade dessa tarefa costuma desencorajar muitos programadores, o que motivou o desenvolvimento do LuaRep [4], uma camada de abstração em Lua para o acesso ao repositório de interfaces. Embora essa ferramenta extra cuide dos detalhes de mais baixo nível, a criação de uma nova interface para cada servant que implemente várias interfaces não seria aceitável.

Uma solução seria manter uma tabela das interfaces que já foram criadas, mas tal tabela à priori não seria visível a outros processos ou nós da rede. Para resolver esse problema, foi criado um objeto que centraliza e gerencia a criação de interfaces em tempo de execução, o *gerenciador de interfaces*. Seu único método, *createInterface*, recebe uma lista de interfaces, cria uma nova que herda de todas elas e retorna seu nome.

As interfaces criadas dessa maneira são inseridas dentro de um módulo reservado, de modo a não poluírem o espaço de nomes do repositório. Mesmo assim, o gerenciador de interfaces mantém em uma tabela as interfaces que já foram criadas. Caso o gerenciador seja solicitado a criar uma interface e já exista uma que implemente todas as interfaces especificadas, esta será retornada em vez de se criar uma nova.

A nova versão da função *luaorb.createservant* aceita várias interfaces, e simplesmente usa o gerenciador de interfaces para criar uma nova quando necessário. Por exemplo, para criar um servant que implemente as interfaces *i1* e *i2* usa-se o comando *luaorb.createservant (impl, i1, i2)*, onde *impl* é a implementação do servant.

4.4 Observações

Os stubs apresentados nesse capítulo oferecem uma abordagem alternativa para a interação cliente-servidor. A transferência de código ao servidor e o encapsulamento de seus métodos possibilita executar código conhecido em ambos os lados da comunicação remota. Sob um certo ângulo, esse código pode ser visto como uma nova camada sobre o *middleware*, capaz de pré ou pós processar os dados de cada método, estender a funcionalidade do servidor, etc.

Em contrapartida, a transferência de código ao cliente é um caminho mais simples e transparente. O código vindo do servidor é desenvolvido pelos autores do serviço, que são programadores experientes e familiarizados com os detalhes internos do projeto. Ao usuário restaria a tarefa de detectar a disponibilidade dos dstubs e incorporá-los aos seus proxies, mas a versão redefinida da função *luaorb.createproxy* já cuida desse detalhe automaticamente.

Contudo, a liberdade com a qual esses stubs podem executar, em suas versões atuais, requer que cuidados adicionais relativos à segurança sejam tomados em aplicações mais sérias.