6
Trabalhos Relacionados

Outros trabalhos têm sido realizados com o intuito de oferecer implementações flexíveis de middleware, assim como oferecer implementações de middleware em linguagens dinâmicas. Este capítulo relaciona alguns desses trabalhos com o OiL, identificando similaridades e diferenças, assim como possíveis contribuições que a abordagem adotada nesses diferentes trabalhos podem trazer ao OiL.

6.1
Middleware Flexíveis

A pesquisa por implementações de middleware flexíveis não é recente (Blair et al., 1998). Diferentes implementações de middleware flexíveis estão disponíveis e sendo utilizadas tanto comercialmente como em projetos acadêmicos. Nesta seção, são apresentadas alguns dos principais trabalhos realizados na linha de middleware flexíveis.

6.1.1
Mico

Como visto na seção 4.1, o Mico[1] é uma implementação aberta de CORBA em C++. Seu objetivo original era ser uma ferramenta de ensino sobre implementação de infra-estruturas para sistemas distribuídos. Essa motivação foi inspirada no sistema operacional Minix[2], projetado inicialmente como ferramenta pedagógica sobre a implementação de sistemas operacionais. Inclusive, o nome Mico originalmente significava *Mini-CORBA*, uma alusão ao nome Minix (*Mini-Unix*). Entretanto, o Mico recebeu a atenção da comunidade de software livre, resultando em uma crescente evolução de sua implementação que culminou quando tornou-se a implementação de referência da especificação CORBA na linguagem C++. Como resultado disso, o significado do seu nome mudou para a sigla recursiva *Mico Is CORBA*, em homenagem ao sistema operacional de software livre GNU (*GNU's Not Unix*). Apesar do crescente número de contribuições, que resultou em uma implementação maior e mais

[1] `http://www.mico.org/`
[2] `http://www.minix3.org/`

complexa, o propósito inicial de funcionar como uma ferramenta de ensino acessível a alunos de Ciência da Computação não foi totalmente descartado. Mais recentemente, essa função foi resgatada com a publicação de um livro didático sobre sua implementação (Puder et al., 2006).

Como forma de conciliar a evolução da implementação com o propósito de ser uma ferramenta de ensino, o projeto do Mico tem como foco a flexibilidade da implementação. O objeto disso é permitir adaptar a sua implementação para que possa ser estendida por alunos, mas também possa ser utilizada como uma implementação rica em funcionalidades e recursos, que possa ser utilizada em aplicações e também sirva de base para outros projetos de implementação de middleware. O conceito chave na flexibilidade da implementação do Mico é sua arquitetura baseada em *micro-kernel*, que oferece funcionalidades mínimas e pode ser estendido com novas funcionalidades através do acoplamento de componentes adicionais, como visto na seção 4.1.

Além de ser uma boa base de comparação com o OiL, pois apresenta objetivos similares, o Mico também sugere possíveis linhas de investição futura do OiL. Em particular, a construção de um *micro-kernel* similar ao do Mico utilizando os componentes do OiL pode ser uma forma de facilitar a montagem de ORBs oferecendo uma montagem de núcleo mínimo pré-definido.

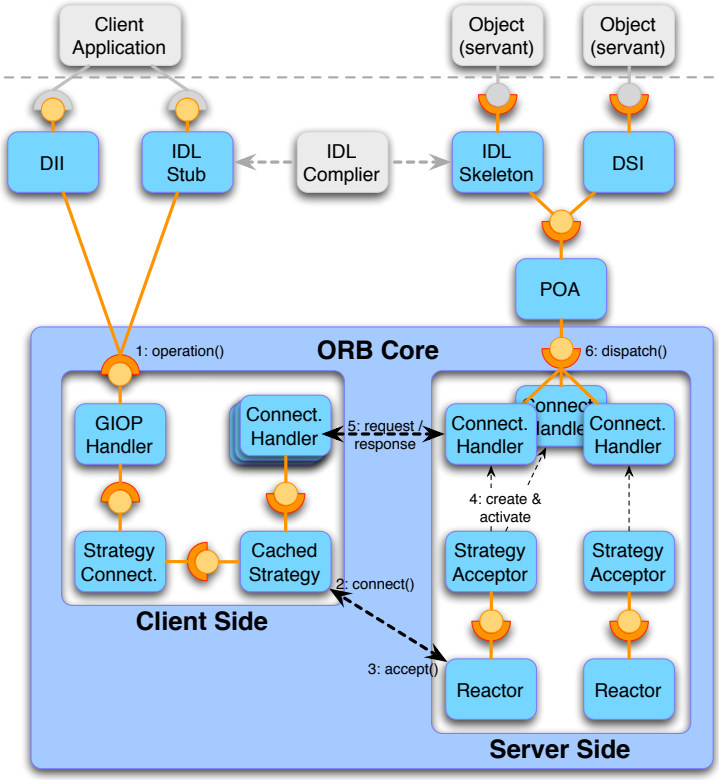
6.1.2 TAO

TAO (Schmidt, 2007) (*The ACE ORB*) é uma implementação de alto desempenho de CORBA para aplicações embarcadas de tempo real em C++. Apesar do foco principal do projeto ser alto desempenho e predição de execução, também foram realizadas pesquisas no sentido de prover mecanismos de adaptação do ORB para configuração dos recursos do middleware. Assim como o OiL, o TAO oferece suporte para adaptação do protocolo CORBA utilizado (O'Ryan et al., 2000) e vários outros aspectos do middleware, como escalonamento de tarefas e utilização de conexões. Contudo, ao contrário do OiL, o TAO utiliza padrões de projeto de software orientado a objetos para permitir maior flexibilidade (Schmidt et al., 1998), que são implementados no arcabouço ACE (Schmidt, 1994) (*ADAPTIVE Communication Environment*). Muitos desses padrões definem como organizar a implementação orientada a objetos do TAO de forma que seja possível modificar as ligações e interações entre esses objetos mais facilmente.

Os padrões de projeto utilizados no TAO facilitam a modificação do código do middleware para acomodar alterações que modifiquem os recursos oferecidos. A figura 6.1 ilustra os componentes internos do núcleo do ORB

do TAO, onde é possível ter implementações de interfaces internas que implementam diferentes estratégias de gerência de conexão (`CachedStrategy`). Inclusive, alguns padrões são utilizados para permitir a configuração do middleware na sua carga (Jain e Schmidt, 1997). O projeto dynamicTAO (Kon et al., 2002) introduz mecanismos de reflexão computacional ao TAO, de forma que seja possível inspecionar conexões entre componentes internos e modificá-los, assemelhando-se a um sistema baseado em componentes, como apresentado em (Kon e Campbell, 2000). Esse tipo de funcionalidade é visto como recurso chave para a área de middleware para sistemas embarcados de tempo real (Gill et al., 1999).

Figura 6.1: Arquitetura do núcleo do TAO.



Assim como o OiL, o TAO objetiva ser uma implementação flexível, ou seja, fácil de ser modificada. Inclusive, o TAO é utilizado como base para implementação e avaliação de diversos mecanismos em middleware (Schmidt, 2006). Contudo, os dois projetos seguem linhas muito diferentes. O foco principal do TAO é alto desempenho, predição de execução e garantias de qualidade de serviço (QoS - *Quality of Service*), que são características essenciais no desenvolvimento de software de tempo-real. Por isso, o TAO se baseia em uma linguagem que permite maior controle do programa, como C++, assim como suporte especializado do sistema operacional para prover garantias de QoS. O projeto OiL, por outro lado, tem por objetivo primordial desenvolver um mid-

dleware de fácil de utilização e boa flexibilidade, fazendo uso das características específicas de linguagens dinâmicas.

Apesar de diferentes, as duas abordagens não são antagônicas. O TAO apresenta diversas implementações e técnicas para melhoria do desempenho de middleware e garantia de QoS que podem ser aplicadas em linguagens dinâmicas. Além disso, linguagens dinâmicas não são necessariamente inadequadas a aplicações de tempo real. Em Lua, por exemplo, é possível impedir que o coletor automático de lixo execute em momentos inadequados que possam comprometer garantias de tempo de execução. Além disso, o suporte a concorrência cooperativa usado no OiL permite maior controle da execução das diferentes *threads* pela aplicação. Uma possível linha de investigação futura deste trabalho é investigar a adequação de middleware em linguagens dinâmicas como o OiL no contexto específico de aplicações de tempo real.

6.1.3 OpenORB

OpenORB é originalmente um conjunto de implementações de middleware adaptáveis construídas utilizando componentes de software reflexivos. A primeira implementação, denominada OpenORB v1, foi um protótipo de middleware com suporte a CORBA escrito na linguagem dinâmica Python (Costa et al., 2000). Esse protótipo foi utilizado como prova de conceito dos meta-modelos que formam a base do OpenCOM, o modelo de componentes reflexivo utilizado nas versões posteriores do OpenORB. O foco principal do OpenORB v1 era prover o maior grau de flexibilidade possível (Parlavantzas et al., 2000). Porém o uso da linguagem dinâmica foi considerado como meio de agilizar a implementação do protótipo, não sendo considerada como uma alternativa real para implementação de sistemas de middleware. O OpenORB v1 foi descontinuado, sendo substituído por uma segunda implementação denominada OpenORB v2, que é escrita na linguagem C++, considerada pelos autores um ambiente mais pragmático e realista (Coulson et al., 2002).

Similarmente ao seu predecessor, o OpenORB v2 (Coulson et al., 2002) também busca por flexibilidade, porém está mais focado em alto desempenho e garantias de integridade na montagem, evitando configurações inválidas do ORB. Ele é implementado usando o modelo de componentes reflexivo do OpenCOM, que é baseado no modelo de componentes binário COM (Box, 1997), proposto inicialmente pela Microsoft. De forma similar ao modelo de componentes utilizado no OiL, os componentes OpenCOM definem suas dependências explicitamente através de receptáculos e permitem interceptação de chamadas de métodos nas conexões estabelecidas entre os componentes,

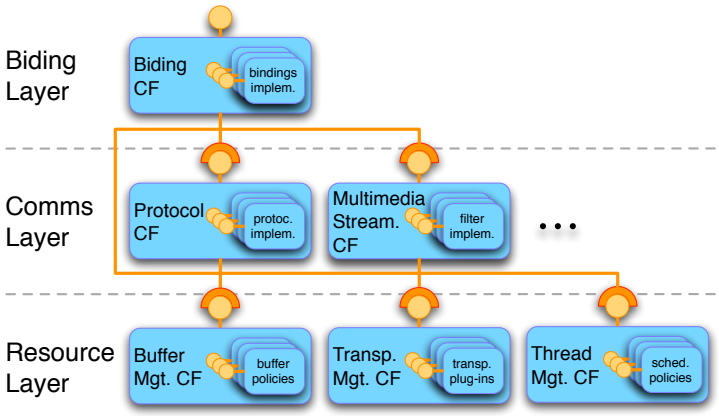
porém com menor sobrecarga no desempenho final da chamada. Ao contrário dos componentes do OiL, as conexões entre componentes do OpenCOM oferecem suporte a interfaces, ou seja, os pontos de conexão entre componentes definem uma interface explícita. Contudo, a verificação das interfaces é feita dinamicamente no momento em que a conexão é estabelecida, portanto não há verificação em tempo de compilação que exclua possíveis erros nas conexões entre os componentes. Por essa razão, o desenvolvedor é responsável por garantir a compatibilidade de interfaces nas conexões, o que pode ser feito por exemplo através dos mecanismos de introspecção de interfaces oferecido pela infra-estrutura do OpenCOM.

Outra diferença importante do OpenCOM é que todas as conexões entre os componentes são mantidas no componente central do modelo do OpenCOM, que permite inspecionar todas as conexões feitas pelo componente. No modelo de componentes do OiL, é possível iterar sobre todas as conexões estabelecidas por um dado receptáculo, contudo, para identificar todas as conexões estabelecidas por uma faceta, é necessário iterar sobre todos os componentes da arquitetura e verificar quais receptáculos estão conectados a interface. Contudo, é possível estender facilmente o suporte oferecido pelo LOOP de forma que quando uma faceta seja conectada a um receptáculo, a informação do receptáculo e o seu respectivo componente seja associado a faceta, permitindo o rastreamento das dependências da faceta. O OpenCOM também oferece suporte para travas de exclusão mútua, que permitem que *threads* preemptivas acessem concorrentemente as conexões de um mesmo componente sem risco de resultar em um estado inválido. Contudo, como o OiL utiliza concorrência cooperativa, tal mecanismo é desnecessário.

A arquitetura interna do OpenORB v2 apresenta muitas similaridades com a do OiL. Em particular, a arquitetura é dividida em três camadas, como ilustrado na figura 6.2. A primeira camada, denominada *Binding Layer*, é responsável por implementar o modelo de programação oferecido para a aplicação, que no caso é feito através de conexões entre os objetos distribuídos, de forma similar às conexões entre componentes do OpenCOM. A grosso modo, essa camada é comparável aos componentes `ProxyManager` e `ServantManager` da camada de apresentação do OiL. A segunda chamada, denominada *Comms Layer*, é responsável por implementar protocolos de RMI e outros mecanismos de comunicação como fluxos de dados multimídia. Essa segunda camada é comparável à camada do protocolo do OiL. A última camada, denominada camada de recursos, é responsável por oferecer acesso aos recursos da plataforma e implementar políticas de uso desses recursos, como criação de canais de comunicação como *sockets*, a forma de escalonamento de *threads*, políticas de

cópia de memória entre *buffers*, etc. Essa camada é comparável ao componente `BasicSystem` do OiL.

Figura 6.2: Arquitetura em camadas do OpenORB v2.



Uma diferença importante na arquitetura do OpenORB em relação ao OiL é a organização em *component frameworks* (CF), que são componentes aninhados, ou seja, componentes formados por sub-componentes. Os componentes dentro de um CF só podem ser configurados através de interfaces específicas oferecidas pelo CF. Isso permite que o desenvolvedor possa impor certas regras na composição interna dos CF, evitando assim a possibilidade de composições inválidas ou indesejadas. No OiL, essas regras de composição são impostas através do uso de *scripts* de montagem. Contudo, tanto a elaboração dos *scripts* de montagem do OiL como a implementação das operações de reconfiguração dos CF do OpenORB — de forma a evitar construções inválidas sem limitar as possibilidades de construções desejáveis — são tarefas complexas que são deixadas inteiramente na mão do configurador ou montador do ORB.

Atualmente, o OpenORB não é mais visto como uma implementação específica de middleware, mas como uma abordagem para construção de middleware baseado em componentes reflexivos. O foco atual do projeto é utilizar essa abordagem no desenvolvimento de middlweare para domínios específicos (Coulson et al., 2006; Grace et al., 2005; Costa et al., 2005) e trabalhar na evolução do modelo do OpenCOM, que constitui a base dessa abordagem (Coulson et al., 2004; Coulson et al., 2008). Atualmente, o OpenCOM já está disponível em linguagens mais dinâmicas como Java, que é considerada pelos usuários do OpenCOM mais fácil de utilizar do que C++ (Group, 2007). Inclusive, há uma implementação orientada a aspectos do OpenORB escrita em Lua (Cacho et al., 2007) utilizada na comparação de diferentes abordagens de programação orientada a aspectos em Lua e Java.

O middleware OpenORB representa uma forma interessante de uso de linguagens dinâmicas no desenvolvimento de middleware devido ao recente interesse de seus autores por meios de simplificar a utilização do modelo OpenCOM e a implementação dos sistemas de middleware. Em particular, a dinamicidade e suporte à reflexão dos componentes OpenCOM casam bem com linguagens dinâmicas. Uma possível linha de investigação futura é avaliar e estender o modelo de componentes do OiL para atender as funcionalidades e ao modelo do OpenCOM, identificando as possíveis facilidades introduzidas pelo uso da linguagem dinâmica. Outro aspecto importante dessa comparação seria avaliar o impacto no desempenho do modelo de componentes e avaliar sua viabilidade nos contextos onde o OpenCOM é utilizado atualmente. Tal estudo poderia incluir também uma revisão da implementação original do OpenORB v1, contudo avaliando o uso de Python como linguagem de implementação final e não apenas uma linguagem de prototipação.

6.1.4 LegORB e UIC

LegORB (Román et al., 2000) é uma implementação de middleware baseado em componentes reflexivos em C++ com suporte para CORBA. O objetivo principal do LegORB é permitir a montagem de ORBs pequenos de baixo consumo de memória para dispositivos embarcados. Esses ORBs são compostos por um número mínimo de componentes necessários para implementar os recursos utilizados pela aplicação. É possível realizar montagens estáticas, que são definidas durante a compilação e possuem tamanho menor, e montagens dinâmicas, que podem ser modificadas em tempo de execução. O UIC (*Universally Interoperable Core*) (Román et al., 2001) é um segundo middleware dos mesmos autores do LegORB que apresenta características muito similares, porém oferece suporte para outras tecnologias de RMI além de CORBA. Tanto o LegORB como o UIC utilizam o mesmo princípio de gerência de dependências entre componentes utilizado no dynamicTAO, que permite inspecionar a montagem dos componentes e atuar sobre ela em tempo de execução. Essa abordagem também é similar ao suporte de reflexão oferecido mais recentemente pelo OpenCOM.

Com objetivo de manter o ORB mínimo, tanto o LegORB quanto o UIC implementam apenas o suporte de interoperabilidade de CORBA. Além disso, eles não oferecem um modelo de programação de alto nível baseado em *proxies* e objetos distribuídos. Ao invés disso, o ORB oferece interfaces de invocação dinâmica que são utilizadas diretamente pela aplicação para realizar chamadas remotas. O ORB também não implementa suporte para codificação e

decodificação de tipos de dados compostos como seqüências, estruturas e uniões de CORBA. Contudo, os autores argumentam que tais limitações podem ser contornadas através da adição de novos componentes a montagens do ORB que implementem essas funcionalidades sobre os componentes mínimos da implementação padrão (Román et al., 2001).

O LegORB e UIC são funcionalmente equivalentes à camada do protocolo do OiL. Contudo, a camada do protocolo do OiL permite a codificação de valores compostos. Outra diferença importante é o tamanho reduzido do LegORB e UIC, que podem chegar até 100 KB, incluindo suporte completo para cliente e servidor, enquanto que apenas a máquina virtual de Lua padrão, que é necessária para executar o OiL, pode chegar a 120 KB. Por outro lado, as limitações de memória em muitos dispositivos têm reduzido gradualmente com a evolução da tecnologia. Por essa razão, em muitos domínios de aplicação, como em aplicações para celulares, o uso de implementações maiores, com mais funcionalidades e mais fáceis de utilizar é preferível.

A eficiência de Lua, tanto em consumo de memória quanto em velocidade de processamento, a torna uma linguagem dinâmica ideal para dispositivos embarcados e pequenos. Além disso, a implementação padrão de Lua é facilmente modificável, pois sua implementação é pequena e estruturada de forma a facilitar remover partes, como o compilador disponível em tempo de execução. Isso permite remover partes da máquina virtual de forma a diminuir seu tamanho. Essa flexibilidade aliada à arquitetura baseada em componentes do OiL permitem adotar uma abordagem de composição de ORBs mínimos, similar à proposta do LegORB e UIC, possibilitando a implementação de sistemas de middleware baseados no OiL para dispositivos com restrições de memória.

6.2 Middleware para Linguagens Dinâmicas

Com a popularização das linguagens dinâmicas, a necessidade do suporte de middleware nessas linguagens se tornou evidente. Como resultado, algumas implementações de middleware passaram a oferecer suporte para que suas funcionalidades também sejam acessíveis em linguagens dinâmicas através de *bindings*. Similarmente, surgiram novas implementações inteiramente escritas em linguagens dinâmicas. Nesta seção, são apresentados alguns desses trabalhos de implementação de middleware para linguagens dinâmicas.

6.2.1 CORBA em Linguagens Dinâmicas

CORBA define um mapeamento oficial para a linguagem dinâmica Python (Object Management Group, 2002) e um mapeamento para a linguagem dinâmica Ruby está sendo proposto (Object Management Group, 2009). Há algumas implementações desses mapeamentos oficiais. Por exemplo, há mapeamentos do omniORB, um ORB C++, e ORBit, um ORB em C, para que possam ser utilizados a partir de Python (Grisby, 2008; Tackaberry, 2000). Além disso, há um projeto, atualmente conduzido pela comunidade de software livre, que oferece uma implementação de CORBA inteiramente em Python (Thomson e Smith, 2005). Existem também outros mapeamentos de CORBA não oficiais para linguagens dinâmicas como Perl (Elliott, 2008).

Diferentemente do OiL, muitos desses mapeamentos oferecem as interfaces definidas por CORBA para o acesso aos serviços do ORB. Em alguns casos, essas interfaces tornam a utilização do ORB mais prolixa, porém são mais facilmente assimiladas por desenvolvedores familiarizados com outros mapeamentos de CORBA. Outra característica comum nos mapeamentos de CORBA em linguagens dinâmicas é a utilização de geração automática de código estático para adaptar o ORB às interfaces definidas pela aplicação (geração de *stubs* e *skeletons* de CORBA), mesmo quando o suporte à reflexão computacional da linguagem dinâmica permite fazer isso em tempo de execução. Uma exceção disso é o suporte de Python oferecido pelo ORBit, que permite criar *proxies* dinâmicos para objetos cuja interface é fornecida em tempo de execução, de forma muito similar ao que é feito no OiL. A figura 6.3(a) ilustra a implementação de um servidor que cria um objeto CORBA usando o ORBit2 para Python. E a figura 6.3(b) mostra a implementação de uma aplicação cliente que acessa esse objeto remoto. Outro exemplo disso é o LuaOrb (Cerqueira et al., 1999), que é precursor do OiL. O LuaOrb permite criar *proxies* e *servants* dinamicamente em Lua utilizando uma implementação de CORBA em C++. Para tanto, ele utiliza os mecanismos de reflexão computacional de Lua, de forma similar ao que é feito no OiL, porém envia e recebe chamadas e através das interfaces DII e DSI do ORB C++.

Diferentemente dos mapeamentos de CORBA para linguagens dinâmicas, o OiL faz uso dos recursos particulares das linguagens dinâmicas para oferecer novas funcionalidades, como a descoberta dinâmica de interfaces de *servants* remotos vista na seção 3.1. Outros recursos particulares do OiL incluem o suporte a concorrência cooperativa e adaptação dinâmica do ORB em virtude de modificações das interfaces armazenadas no repositório de interfaces. Um outro diferencial é a implementação aberta e modificável através da

Figura 6.3: Criação de *servant* e acesso com *proxy* usando ORBit2 em Python.

6.3(a): Programa servidor.

```
1  import CORBA
2  import sys
3
4  class Hello:
5      def say(self):
6          print "Hello, World!"
7
8  CORBA.load_idl("Hello.idl")
9  orb = CORBA.ORB_init(sys.argv,
10                       CORBA.ORB_ID)
11 poa =
12   orb.resolve_initial_references(
13                         "RootPOA")
14
15 servant = POA.Hello(Hello())
16 poa.activate_object(servant)
17 ref = poa.servant_to_reference(
18                          servant)
19 print orb.object_to_string(ref)
20
21 poa.the_POAManager.activate()
22 orb.run()
```

6.3(b): Programa cliente.

```
1  import CORBA
2  import sys
3
4  CORBA.load_idl("Hello.idl")
5  orb = CORBA.ORB_init(sys.argv,
6                       CORBA.ORB_ID)
7  ior = sys.stdin.readline()[:-1]
8  hello = orb.string_to_object(ior)
9  hello.say()
```

reconfiguração dos componentes internos. Essa facilidade de reconfiguração também permite que o OiL ofereça suporte para outras tecnologias de RMI como é exemplificado pelo suporte ao protocolo LuDO.

O OiL oferece interfaces próprias para acesso aos serviços do ORB, que são distintas das definidas pelo padrão CORBA. Por um lado, isso dá oportunidade para utilização de interfaces mais adequadas à linguagem de programação. Por exemplo, no OiL, as chamadas de função nas linhas 17–19 da figura 6.3(a) são substituídas pela função `tostring` da biblioteca padrão Lua. Além disso, como o OiL não se compromete a seguir o padrão CORBA, é possível experimentar com modelos de programação alternativos ao definidos por CORBA. Por outro lado, essas interfaces podem ser confusas para programadores familiarizados com os demais mapeamentos de CORBA. Isso pode ser especialmente problemático em um ambiente onde há intenso uso de múltiplas linguagens. Por isso, um possível trabalho futuro é estender a implementação do OiL para oferecer opcionalmente interfaces similares às definidas pelos mapeamentos de CORBA para outras linguagens dinâmicas.

6.2.2 Pyro

Pyro (Jong, 2007) (*Python Remote Objects*) é uma tecnologia para desenvolvimento de sistemas baseados em objetos distribuídos totalmente desenvolvida na linguagem Python. Diferentemente dos trabalhos relacionados anteriormente, o Pyro define um ORB que utiliza um protocolo próprio, denominado *PYRO*. O protocolo PYRO permite realizar chamadas usando o

modelo de chamadas de Python, ou seja, chamadas com tipagem dinâmica dos parâmetros. Assim como o protocolo LuDO implementado no OiL, o PYRO envia todos os valores por cópia. Para tanto ele utiliza a biblioteca de serialização de objetos de Python denominada `pickle`. A passagem de objetos por referência é feita serializando *proxies*, de forma similar ao LuDO sem a extensão `ludo.byref`. O Pyro também oferece um suporte especial, denominado de *migração de código*, que permite enviar pela rede a implementação das classes de objetos serializados que não são conhecidas no destino. No LuDO tal funcionalidade é transparente, pois as classes de objeto em Lua, representadas por meta-tabelas, são codificadas como os tabelas normais.

Assim como o OiL, o Pyro é especificamente projetado para prover um modelo de programação simples e fácil de usar, tentando ocultar algumas das complexidades inerentes da programação distribuída. Por exemplo, o Pyro implementa um mecanismo de busca automática de um serviço de nomes disponível na rede utilizando *broadcast* UDP. Dessa forma, a aplicação pode registrar objeto distribuídos que possam ser facilmente obtidos pelas diferentes partes do sistema.

Figura 6.4: Criação de *servant* e acesso com *proxy* usando Pyro.

6.4(a): Programa servidor.

```
1  import Pyro.core
2  import Pyro.naming
3
4  class Hello(Pyro.core.ObjBase):
5    def say(self):
6      print "Hello, World!"
7
8  Pyro.core.initServer()
9  l=Pyro.naming.NameServerLocator()
10 ns=l.getNS()
11 daemon=Pyro.core.Daemon()
12 daemon.useNameServer(ns)
13 uri=daemon.connect(Hello(),
14                     "myHelloWorld")
15 daemon.requestLoop()
```

6.4(b): Programa cliente.

```
1 import Pyro.core
2
3 hello = Pyro.core.getProxyForURI(
4           "PYRONAME://myHelloWorld")
5 hello.say()
```

A figura 6.4(a) ilustra o código de um servidor que cria um *servant* no Pyro. Na linha 4, é criada a classe que implementa o *servant* a ser criado. Essa classe deve estender a classe `Pyro.core.ObjBase`, que é definida pela implementação do Pyro. Alternativamente, é possível criar uma instância da classe `Pyro.core.ObjBase` a ser usada como implementação do *servant*, mas que delega todas as chamadas para um outro objeto. Para criar novos *servants*, é necessário iniciar o processo como um servidor (linha 8), caso contrário o Pyro é iniciado apenas com suporte de realização de chamadas. Em seguida, o Pyro obtém uma referência para um serviço de nomes acessível pela rede (linhas 9–10). Na linha 11, um *daemon* é criado, que corresponde a um *broker* do OiL.

O servidor de nomes a ser utilizado é registrado no *daemon* (linha 12) e o novo *servant* é criado e registrado no serviço de nomes em uma única operação (linha 14). Por fim, o *daemon* passa a executar continuamente processando requisições remotas (linha 15). No caso da aplicação cliente, ilustrada na figura 6.4(b), o Pyro ainda permite mais ações implícitas, como a localização do serviço de nomes, que é feita automaticamente pela infra-estrutura (linha 3).

Ao contrário do LuDO, o protocolo PYRO é um protocolo binário, que permite uma implementação mais eficiente, inclusive há uma implementação da codificação inteiramente em C (`cpickle`), que é mais rápida que a implementação em Python (`pickle`). Por outro lado, a codificação do LuDO é mais flexível, pois o formado codificado é código executável Lua, que pode incluir ações especiais a serem realizadas na decodificação, sem modificar a implementação do decodificador, que consiste basicamente do interpretador de Lua. Isso permite implementar variações da codificação do LuDO, como por exemplo, a passagem de valores por referência através da criação implícita de *servants*. Outras funcionalidades particulares do OiL incluem o suporte a *proxies* alternativos, que permitem chamadas assíncronas através de futuros, e *proxies* genéricos, que permitem acessar remotamente outros valores de Lua além de objetos convencionais. O OiL também oferece interoperabilidade com ORBs CORBA sob o mesmo modelo de programação do LuDO, tornando a escolha do protocolo de RMI utilizado transparente para aplicação em alguns casos. Além disso, o OiL pode ser estendido e modificado através de novos componentes ou novas formas de montagem. Contudo, o Pyro também foi projetado para facilitar a implementação de outros protocolos de RMI.

O Pyro sugere algumas linhas interessantes de evolução do OiL. Em particular, a investigação de protocolos de RMI que permitam chamadas dinâmicas de forma mais eficiente do que é feito no LuDO. Muitas das limitações atuais do Pyro[3] estão relacionadas ao fato dos valores transmitidos pela rede serem passados por cópia. Essa limitação pode ser contornada com as ações implícitas implementadas no OiL. As facilidades da interface de programação oferecida pelo OiL atualmente se restringem principalmente ao ORB, enquanto que o Pyro implementa algumas ações implícitas para acesso a serviços do middleware, como a descoberta do serviço de nomes. Por isso, é interessante investigar como os recursos particulares do OiL podem facilitar a utilização de outros aspectos do middleware além do ORB.

[3] `http://pyro.sourceforge.net/manual/7-features.html#rules`