Aplicações de gerência com comportamento dinâmico

Noemi Rodriguez
Ana Lúcia de Moura
Roberto Ierusalimschy
Renato Cerqueira
noemi,ana,roberto,rcerq@inf.puc-rio.br
Departamento de Informática - PUC-Rio

Resumo

Este trabalho apresenta um binding de CORBA com a linguagem de extensão Lua. Discute-se como a combinação das características de uma linguagem de extensão flexível com o acesso simples a objetos CORBA e a primitivas SNMP pode facilitar o desenvolvimento de aplicações de gerência dinamicamente extensíveis.

1 Introdução

O acesso a objetos CORBA [Obj95] vem se afirmando como uma importante alternativa na área de gerência de serviços e telecomunicações [KC96,Ban97]. Uma das vantagens dessa abordagem é o desenvolvimento de aplicações de gerência sob a forma de clientes CORBA, usufruindo das facilidades de verificação oferecidas pelo uso de uma IDL, ao invés de comunicação baseada unicamente em um protocolo de gerência.

O repositório de interfaces CORBA (IR) mantém descrições das interfaces disponíveis, e pode ser atualizado e consultado através de chamadas previstas no modelo. Uma vez conhecida a interface de um objeto servidor, um cliente CORBA pode acessar os seus métodos através de dois tipos de chamadas: estáticas e dinâmicas. Chamadas estáticas usam *stubs* gerados em tempo de compilação, a partir de descrições IDL dos objetos a serem invocados. Chamadas dinâmicas utilizam a *Dynamic Invocation Interface (DII)*, que permite a um cliente construir chamadas a servidores em tempo de execução.

Para que se possam usar chamadas construídas estaticamente, a aplicação de gerência deve incorporar os *stubs* gerados pelo compilador IDL. Em outras palavras, ao ser gerada, a aplicação de gerência deve ter conhecimento dos objetos aos quais ela fornecerá acesso. Isso pode ser bastante restritivo em um ambiente interredes, onde um administrador de sistemas está continuamente tomando conhecimento de novos problemas e de novos serviços. Nesse contexto, o uso da interface de invocação dinâmica permite que uma ferramenta de gerência possa incorporar acesso a novos objetos sem necessidade de se recompilar a ferramenta, ou mesmo interromper sua execução.

A grande maioria dos *bindings* oferecidos hoje para CORBA são baseados no uso da interface de invocação estática. Na realidade, com exceção do *binding* de Smalltalk, o uso dessa interface é natural pelo fato das linguagens em questão (C, C++, ou Java) serem tipadas estaticamente.

Usando-se chamadas construídas estaticamente, o benefício oferecido ao modelo de gerência pelo uso do repositório de interfaces é limitado. Pode-se acessar o repositório através de uma ferramenta estilo *browser*, e, baseando-se na informação coletada, gerar uma nova versão da aplicação de gerência. Isso exige, no entanto, um passo de recompilação e reinstalação da aplicação de gerência para cada mudança detectada pelo *browser*.

Este trabalho, em contraste, utiliza um *binding* de CORBA baseado no uso da interface de invocação dinâmica para uma linguagem interpretada, tipada dinamicamente. A linguagem em questão é Lua, uma linguagem de extensão desenvolvida na PUC-Rio. O uso de chamadas construídas dinamicamente permite que uma aplicação de gerência faça uso efetivo do repositório de interfaces CORBA.

O uso de Lua para acesso a servidores CORBA permite que trabalhemos com um console de gerência, a partir do qual o administrador pode interrogar o repositório e fazer acessos aos objetos encontrados. Além disso, também estamos disponibilizando para Lua, como parte do projeto GERENTE [S+96], uma biblioteca de funções de gerência SNMP [CFSD90]. Isso significa que o console de gerência pode tanto acessar servidores CORBA como agentes SNMP tradicionais. Essa abordagem permite que o administrador possa utilizar uma mesma aplicação para gerência de objetos CORBA e SNMP, e, o que é mais importante, que essa aplicação possa se adaptar dinamicamente a mudanças nos serviços e configuração da rede gerenciada.

2 Por que uma Linguagem Interpretada?

O termo *linguagem interpretada* não é muito preciso, por pelo menos duas razões: em primeiro lugar, normalmente uma mesma linguagem pode ser implementada tanto através de um compilador quanto através de um interpretador. Em segundo lugar, implementações normalmente usam uma combinação de ambas as técnicas. Por exemplo, linguagens como Pascal e Java, normalmente classificadas como compiladas, são muitas vezes pré-compiladas para um código intermediário (P-code, em Pascal, e bytecode, em Java) que é então interpretado. Vale lembrar também que mesmo um programa em linguagem de máquina é interpretado pela CPU, e também pode ser interpretado por um programa emulador em outras arquiteturas.

Apesar do dito acima, o termo *linguagem interpretada* em geral se apresenta como um termo útil para classificar uma classe de linguagens com características distintas das linguagens ditas convencionais, como FORTRAN ou C. Para tornar o termo um pouco mais preciso, vamos adotar a seguinte definição: uma linguagem será classificada como interpretada se ela tiver algum mecanismo para execução de trechos de código fonte criados dinamicamente; em outras palavras, se o interpretador for diretamente acessível pela linguagem. De acordo com a definição acima, linguagens como Lisp, Lua, Mumps e Tcl são interpretadas (independentemente de suas implementações específicas), enquanto Pascal, C, C++ e Java ficam fora da categoria.

Várias propriedades típicas de linguagens interpretadas seguem diretamente da definição acima, como *interatividade*, possibilidade de *programação pelo usuário* (*end user programming*) e *reflexividade*. Outras propriedades não são consequência direta da definição, mas facilitam o desenho de linguagens interpretadas, como, por exemplo, ausência de declarações e tipagem dinâmica. Finalmente, algumas características poderosas comumente encontradas em linguagens interpretadas, como coleta automática de lixo e estruturas dinâmicas, não têm nenhuma relação direta com a definição de linguagem interpretada, e de fato estão presentes em diversas linguagens compiladas.

Chamamos de *interatividade* em uma linguagem a possibilidade do programador entrar com comandos da linguagem um a um, e esses serem imediatamente executados, como em um console com uma linguagem de comandos (shell). Essa característica, por si só, já seria uma forte razão para o uso de uma linguagem interpretada no âmbito de gerência de redes. Através de um console interativo um operador tem acesso imediato a todas as facilidades e opções do sistema, independentemente da existência de ferramentas específicas. Um console também se mostra bastante útil durante as fases de desenvolvimento e prototipação, permitindo que as diversas partes de um sistema sejam acionadas e testadas sem a necessidade de desenvolvimento de programas específicos para testes.

Outra característica de linguagens interpretadas é a não necessidade de recompilação e religação do código após cada modificação. Essa propriedade é útil tanto na construção de pequenos programas (em especial para programação pelo usuário) quanto para prototipagem rápida, onde diversas opções de projeto podem ser implementadas e testadas em um curto espaço de tempo. Outra vantagem de linguagens interpretadas, para a construção de protótipos, é a existência de estruturas dinâmicas não limitadas e gerência automática de memória. Estruturas de dados como listas em Lisp ou tabelas em Lua tornam trivial a implementação de diversos tipos de dados, como conjuntos e tabelas de símbolos, aliviando o projetista do trabalho de implementar complexas estruturas de dados para colocar um protótipo em funcionamento.

No presente trabalho, o interesse particular é no subconjunto de linguagens interpretadas conhecidas como linguagens de *configuração* ou de *extensão*. O uso de uma linguagem de programação completa para configuração de aplicações permite que o arquivo de configuração deixe de ser um mero repositório de dados e opções, e passe a ser o código de uma rotina de inicialização, contando para isso com toda a expressividade de uma linguagem de programação (condicionais, repetições, abstrações, etc).

É interessante observar que linguagens de configuração, quando suficientemente expressivas, podem ser usadas por vários níveis de programadores/usuários. Tipicamente, pode-se classificar pelo menos dois níveis desses "usuários": o primeiro nível é formado por administradores do sistema e outros profissionais, que podem usar todo o poder da linguagem para adaptar um programa a um determinado ambiente. Com o uso de uma linguagem de configuração, tais profissionais podem programar políticas de segurança do sistema (por exemplo selecionando que funcionalidades do sistema estarão disponíveis em geral), adaptar o programa para a estrutura de diretórios local, etc. Um segundo nível de usuários são os chamados *expert users*, normalmente usuários finais do programa mas com interesse natural em programação. Tais usuários geralmente não usam toda a expressividade da linguagem de configuração, mas a utilizam para tarefas mais básicas, como configuração de um menu de opções. Mesmo nesse nível de uso, a expressividade dos construtores de uma linguagem de programação é relevante (por exemplo, um simples *if* permite que um menu tenha configurações distintas dependendo do estado corrente do sistema).

Dois pontos fracos normalmente associados com linguagens interpretadas são eficiência e robustez. De fato, linguagens interpretadas são normalmente significativamente mais lentas que linguagens compiladas (um fator de até 10 vezes não é incomum). Entretanto, em muitos domínios de aplicação, uma fatia substancial do tempo total de execução de um programa é gasta em operações primitivas, fora do domínio do programa, como operações sobre arquivos ou comunicação. Para esses programas, a eficiência de uma linguagem compilada é de pouca valia. De forma semelhante, a robustez de uma linguagem deve ser considerada no contexto em que a linguagem está sendo usada. Certamente, a verificação estática de tipos é um aliado importante na construção de grandes sistemas de software. Entretanto, em linguagens convencionais, a tipagem estática tem seu custo, na forma de perda de polimorfismo e flexibilidade pelos componentes do programa. Além disso, se por um lado linguagens interpretadas tem fraca verificação estática, seus sistemas de verificação de erros dinâmicos (como variáveis não inicializadas, referências pendentes, etc) são naturalmente robustos, e de muita valia na depuração de programas.

A Linguagem de Extensão Lua

Lua é uma linguagem de extensão de propósito geral que integra facilidades para descrição de dados, reflexividade e construções com sintaxe simples. Entre seus aspectos "convencionais", Lua é uma linguagem procedural com as estruturas de controle padrão (*while*, *if*, ...), definições de funções com parâmetros e variáveis locais, etc. Entre seus aspectos menos convencionais, Lua oferece funções como valores de primeira classe, e arrays associativos (*tabelas*) como um mecanismo geral de estruturação de dados.

Funções em Lua são valores de primeira classe, ou seja, podem ser manipuladas como qualquer outro valor da linguagem. Uma definição de função cria um valor do tipo *function* e atribui esse valor a uma variável global. Como qualquer outro valor, esse valor pode ser armazenado em variáveis, passado como parâmetro, ou retornado como resultado de uma outra função. Essa característica simplifica bastante a implementação de facilidades de orientação a objetos.

Tipos são verificados dinamicamente em Lua. Uma variável pode armazenar um valor de qualquer tipo. Sempre que uma operação é ativada, os tipos de seus argumentos são verificados. Além dos tipos básicos `number` (floats) e `string`, e do tipo `function`, Lua oferece mais três tipos de dados: `nil`, com um único valor também chamado `nil`, cuja propriedade básica é ser diferente de qualquer outro valor da linguagem; `userdata`, que representa ponteiros em C; e `table`.

O tipo `table` implementa arrays associativos, ou seja, arrays que podem ser indexados por qualquer valor da linguagem (com exceção de `nil`). Arrays associativos constituem uma ferramenta de programação bastante poderosa; muitos algoritmos se tornam triviais com seu uso [Ben88]. Estruturas de dados mais comuns, como arrays convencionais, conjuntos, bags e tabelas de símbolos, podem ser facilmente implementadas com tabelas. Estruturas também podem ser implementadas por tabelas, usando-se os nomes de campos como índices. Lua dá suporte a essa representação, tratando `a.nome` como açúcar sintático para `a["nome"]`.

Como funções são valores de primeira classe, campos de tabelas podem armazenar funções. Essa propriedade permite a implementação de alguns mecanismos interessantes de orientação a objetos, cujo uso é simplificado com um pouco de açúcar sintático. Por exemplo, a sintaxe:

```
receiver:foo(params)
```

é equivalente a

```
receiver.foo(receiver, params)
```

Ou seja, a função armazenada no campo `foo` do objeto `receiver` é chamada, tendo como primeiro parâmetro o próprio `receiver` (que faz assim o papel do `this` em C++). Em [IFC96] o leitor pode encontrar um tratamento mais completo das facilidades de orientação a objetos de Lua, que está fora do escopo deste trabalho.

Parte da natureza reflexiva de Lua é fornecida por *fallbacks*. Fallbacks permitem que o programador modifique o comportamento de certas construções da linguagem, em especial para situações de erro. Fallbacks em Lua têm implementação eficiente e muitas vezes são usadas para emular facilidades inexistentes na linguagem original, como herança e sobrecarga [IFC96]. Como um exemplo, a expressão apresentada acima,

```
receiver:foo(params)
```

geraria um erro de execução caso `receiver` não fosse um objeto Lua (uma tabela), ou não tivesse um método `foo`. Com o uso de fallbacks, o programa pode interceptar esse erro, e dar uma semântica apropriada para a expressão.

3 LuaOrb

3.1 A Interface de Invocação Dinâmica CORBA

No padrão CORBA [Obj95], toda a comunicação entre objetos é mediada pelo *Object Request Broker* (ORB) [Obj95,Sie96]. Um cliente pode interagir com o ORB através de *stubs IDL* ou através da interface de invocação dinâmica (DII - *Dynamic Invocation Interface*).

A interação através de *stubs* funciona de maneira análoga ao modelo de chamadas remotas de procedimento [Sun88,BN84]. A compilação de uma especificação escrita em IDL, a sintaxe definida por CORBA para descrição de interfaces, gera um *stub* cliente e um *stub* servidor, chamado, no modelo CORBA, de programa esqueleto. A partir daí, o programador pode criar o programa cliente e a implementação do servidor. Do ponto de vista do cliente, os serviços desejados são requisitados ao *stub* cliente, que se encarrega de repassá-los ao servidor, codificando (*marshalling*) os parâmetros de entrada, e de tratar a resposta, decodificando (*unmarshalling*) os resultados. O programa esqueleto se encarrega de receber o pedido do *stub* cliente e repassá-lo à implementação do servidor.

A interface de invocação dinâmica (DII) é uma facilidade genérica para invocação de qualquer operação CORBA; usando-se DII, a chamada da operação e a lista de parâmetros são construídas em tempo de execução. A interação através da DII envolve as etapas de 1) identificação do objeto servidor, 2) obtenção de sua interface, 3) construção da chamada (invocação) e 4) realização da chamada e obtenção de resultados.

A identificação do objeto a ser contactado pode ser obtida através de serviços disponíveis no ambiente CORBA, como o serviço de nome ou o *trading object* *server* [Sie96]. Também é possível que o usuário forneça a identificação de um novo objeto explicitamente, possivelmente depois de tomar conhecimento desse objeto através de algum serviço de divulgação. Uma vez de posse de uma referência para o servidor, o programa pode requisitar meta-dados que descrevem sua interface. O método `get_interface`, oferecido por todo objeto CORBA, pode ser usado para se ter acesso à descrição do objeto. Esse método retorna um objeto `InterfaceDef`, que funciona como um ponto de entrada no repositório de interfaces. Navegando no repositório de interface, pode-se obter informação detalhada sobre a interface de um objeto. Utilizando-se essa descrição, e uma estrutura de dados específica para descrição de listas de parâmetros, a *Named Value List*, pode-se construir a lista de argumentos a ser usada na chamada de um método.

A chamada `create_request` cria um novo pseudo-objeto CORBA, que é usado para armazenar o nome do método, a lista de argumentos, e, depois da chamada, o valor retornado. A realização da chamada propriamente dita é feita através da invocação de um método do objeto *Request* (diferentes semânticas de chamada são implementadas por diferentes métodos).

3.2 O binding entre CORBA e Lua

Para tornar o mapeamento entre Lua e CORBA compatível com o uso da linguagem, um objeto CORBA deve poder ser usado da mesma forma que outros objetos Lua. A linguagem se baseia em uso de objetos sem declaração prévia e com tipagem dinâmica. Para estender essa abordagem aos objetos CORBA, é natural usar a DII, uma vez que a geração de *stubs* implica na necessidade de declarações e de um passo de compilação.

O mapeamento é implementado pela biblioteca *LuaOrb*. Quando o programa realiza um acesso a um objeto CORBA, a biblioteca intercepta essa chamada, usando o mecanismo de fallbacks descrito anteriormente. Em seguida, LuaOrb consulta o repositório de interfaces para obter a interface do método sendo chamado. De posse dessa interface, os dados fornecidos na chamada em Lua são convertidos para os tipos descritos na interface. Após a construção da lista de parâmetros, LuaOrb executa a chamada CORBA, e então converte os resultados de volta para valores Lua.

A conversão entre os dados em Lua e os tipos descritos na interface é bastante geral, e é uma das chaves para a facilidade de uso de LuaOrb. Por exemplo, uma tabela Lua pode ser convertida para uma estrutura IDL, desde que tenha os campos descritos na estrutura com tipos apropriados. A ordem dos campos, ou a existência de outros campos na tabela, não afeta a compatibilidade.

4 Aplicações dinâmicas de gerência

Com o objetivo de oferecer uma plataforma de desenvolvimento de aplicações de gerência, foi construída uma outra biblioteca para Lua, a biblioteca *LuaMan* [ML97]. Essa biblioteca implementa, além de funções de acesso a MIB e de SNMP, várias outras facilidades que podem ser úteis em um ambiente de gerência de rede, como comunicação ICMP e acesso ao serviço de nomes DNS.

Combinando as facilidades oferecidas pelas bibliotecas LuaMan e LuaOrb, obtém-se uma console para administração de redes e serviços distribuídos. O administrador pode usar essa console para interativamente escrever comandos Lua e ver os resultados retornados. Além disso, como existem diversas bibliotecas gráficas disponíveis para Lua [LFG+96,Cel97], aplicações com interfaces mais sofisticadas podem ser desenvolvidas. A criação de uma aplicação de gerência com interface gráfica tipicamente não seria uma tarefa para um usuário iniciante, já que requer uma razoável formação de programação. Como discutido na seção 2, uma aplicação desse tipo pode ser construída por programadores mais avançados, como administradores de sistemas de grandes instituições

Para ilustrar o uso integrado de LuaOrb e LuaMan, será apresentado a seguir um exemplo de aplicação das bibliotecas. Este exemplo utiliza um cenário no qual um administrador é informado da disponibilização de um novo serviço de nomes em sua rede.

Utilizando uma base de dados de estrutura semelhante à do DNS, esse novo serviço, implementado como um servidor CORBA, disponibiliza, entre outras informações: a lista de serviços oferecidas por um host, seu endereço e o tipo de acesso a ser utilizado para a obtenção de informações mais detalhadas. A informação quanto ao tipo de acesso pode, por exemplo, indicar que o host incorpora um agente SNMP, acessível através da *community string* especificada. A figura 1 apresenta a descrição, em IDL, da interface oferecida pelo serviço de nomes.

No cenário proposto, o administrador da rede poderia utilizar esse novo serviço para, por exemplo, obter informações sobre os hosts que oferecem o serviço 'WWW' no subdomínio 'TeleMidia' de um domínio 'puc-rio'. Para isso, ele dispõe do método whoDoes, que retorna uma lista dos hosts em um subdomínio que implementam o serviço especificado. As informações desejadas (endereço e tipo de acesso aos hosts) podem ser obtidas através dos métodos `hostAccess` e `hostAddress`. De posse do nome e da descrição da interface oferecida pelo serviço de nomes, o administrador, num primeiro passo, utilizaria o console de gerenciamento para criar um objeto de acesso (*proxy*) ao servidor e, via esse objeto, invocar os métodos de sua interface. O código Lua correspondente é apresentado na figura 2, e um possível resultado para sua execução na figura 3.

```
interface nameServer
{
 typedef sequence<string> String_seq;

 String_seq domains ();
 String_seq subdomains (in string domain);
 String_seq hosts (in string domain, in string subdomain);
 String_seq hostServices(in string domain, in string subdomain,
                         in string host);
 String_seq whoDoes(in string service, in string domain,
                    in string subdomain);
 string hostDescription(in string domain, in string subdomain,
                        in string host);
 string hostAccess(in string domain, in string subdomain,
                   in string host);
 string hostAddress(in string domain, in string subdomain,
                    in string host);
};
```

Figura 1: Descrição IDL de um servidor de nomes.

```
 dom = "puc-rio" ; subd = "TeleMidia"
 ns = createproxy("nameServer")
 hosts = ns:whoDoes("WWW",dom,subd)
 if hosts then
   i = 1
   while hosts[i] do
     host = hosts[i]
     print ("Host: "..host)
     print(format(" access  - %s",ns:hostAccess(dom,subd,host)))
     print(format(" address - %s",ns:hostAddress(dom,subd,host)))
     i = i + 1
   end
 end
```

Figura 2: Acesso a um objeto CORBA

A função `createproxy`, implementada pela biblioteca LuaOrb, tem dois parâmetros. O primeiro, obrigatório, especifica o nome da interface do objeto a ser criado. O segundo, opcional, especifica o nome de uma intância específica de implementação da interface. Se, como no exemplo acima, este segundo parâmetro não for fornecido, qualquer servidor ativo que implemente a interface será contactado. Caso um nome de instância seja fornecido, a chamada a `createproxy` só retornará com sucesso se existir um servidor com o nome especificado. Uma vez criado o objeto de acesso, o mesmo pode ser usado diretamente para a realização das chamadas aos métodos. Observe que as diversas fases envolvidas em uma interação via DII são totalmente transparentes para o programador Lua. Isso pode ser contrastado com a provisão de acesso à interface dinâmica de CORBA em TclDii [ASGadVJ], onde o programador deve codificar explicitamente cada uma das etapas de acesso à DII descritas enteriormente.

```
Host: salgueiro
access  - unkown
address - 139.82.95.11
Host: imperio
access  - SNMP/public
address - 139.82.95.18
```

Figura 3: Resultado da execução do código da figura 2

Prosseguindo com o exemplo, com base no resultado obtido, o administrador de rede poderia obter maiores informações sobre o host 'imperio' acessando a MIB disponibilizada pelo agente SNMP incorporado a este host. Para isso ele dispõe das funções oferecidas pela biblioteca LuaMan. Um trecho de código Lua que realiza o acesso a uma variável SNMP é apresentado na figura 4.

```
session = snmp_open{peer="139.82.95.18",community="public"}
varBind = snmp_get(session,"sysDescr.0")
print(sprint_var(varBind))
```

Figura 4: Acesso a variável SNMP

Através da função `snmp_open`, o administrador de rede abre (cria) uma "sessão SNMP", e fornece os parâmetros de configuração (endereço, community string, entre outros) necessários à realização posterior de operações SNMP sobre o agente associado à sessão. A função `snmp_get` permite a obtenção do valor corrente de uma variável (ou conjunto de variáveis) da MIB disponibilizada pelo agente associado a uma sessão SNMP. O resultado de `snmp_get` é uma tabela do tipo *varbind*. A função auxiliar `sprint_var` oferecida pela biblioteca LuaMan converte uma tabela do tipo *varbind* em uma string no formato

<var> (<tipo>) = <valor>.

5 Considerações Finais

Acreditamos ter mostrado que a combinação das características de uma linguagem interpretada, como Lua, com o acesso simples a objetos CORBA e a primitivas SNMP pode facilitar bastante o desenvolvimento e configuração de aplicações de gerência.

Uma abordagem um pouco semelhante à deste trabalho é apresentada em [Ban97], onde se define uma linguagem interpretada, o GOMScript, para acesso a objetos gerenciados. No entanto, naquele trabalho é usado um modelo de objetos novo, o GOM, que cria uma camada de abstração em relação a CORBA. A idéia é que seja definido um novo tradutor entre cada modelo de gerência a ser integrado (CORBA, SNMP, CMIP, ...) e o ambiente de gerência. Ainda que possa ser interessante tornar transparente ao administrador os diferentes modelos de gerência subjacentes, acreditamos que também é importante que ele possa ter acesso direto, de forma simples, a cada um dos modelos usados. Assim, consideramos o console apresentado aqui, onde o administrador pode especificar diretamente acesso a objetos CORBA e SNMP, essencial. Possivelmente, em uma etapa posterior do projeto, será criada também uma API de nível mais abstrato, correspondendo, de certa forma, ao modelo GOM definido por Ban.

A construção de interfaces gráficas sobre as facilidades oferecidas por este trabalho é bastante simples, dada a existência de várias bibliotecas para desenvolvimento de interfaces gráficas em Lua. Nas interfaces gráficas em desenvolvimento no projeto GERENTE, Lua também pode ser utilizada para configuração dinâmica, permitindo a extensão da funcionalidade da ferramenta (novas telas, novas funções de filtragem, etc) em tempo de execução.

Agradecimentos

Este trabalho está inserido no contexto do projeto GERENTE, desenvolvido dentro da 3ª fase do ProTeM-CC, e contou com o apoio do laboratório TeleMídia/PUC-Rio. Os autores agradecem o apoio financeiro do CNPq (bolsas de pesquisa, doutorado e DTI).

Referências Bibliográficas

[Ban97] Bela Ban. *A Generic Management Model for CORBA, CMIP and SNMP*. PhD thesis, University of Zurich, 1997.

[Ben88] J. Bentley. *More programming pearls*. Addison-Wesley, 1988.

[BN84] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Trans. On Computer Systems*, 2(1), February, 1984.

[Cel97] W.Celes. Tklua binding, 1997. Descrição disponível por WWW em http://www.graphics.cornell.edu/~celes/tklua/tklua.html.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J.Davin. A simple network management protocol, 1990. RFC 1157.

[IFC96] R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.

[KC96] Q. Kong and G. Chen. Integrating corba and tmn environment. In *Network Operations and Management Symposium*. IEEE/IFIP, 1996.

[LFG+96] C. Levy, L. Figueiredo, M. Gattass, C. Lucena, and D. Cowan. IUP/LED: a portable user interface development tool. *Software Practice and Experience*, 26(7):737-762, 1996.

[ML97] A. Moura and M. Lima. A API de gerenciamento LuaMan, 1997. Descrição disponível por WWW em http://www.telemidia.puc-rio.br/~ana/luaman.html.

[Obj95] Object Management Group, Inc., Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.0*, jul 1995.

[S+96] Michael Stanton et al. Ambientes abertos para o gerenciamento de redes de computadores. In *Segundo Seminário Fluminense de Engenharia*, Niteroi, RJ, 1996. UFF.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[Sun88] Sun Microsystems. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, jun 1988. RFC 1057.