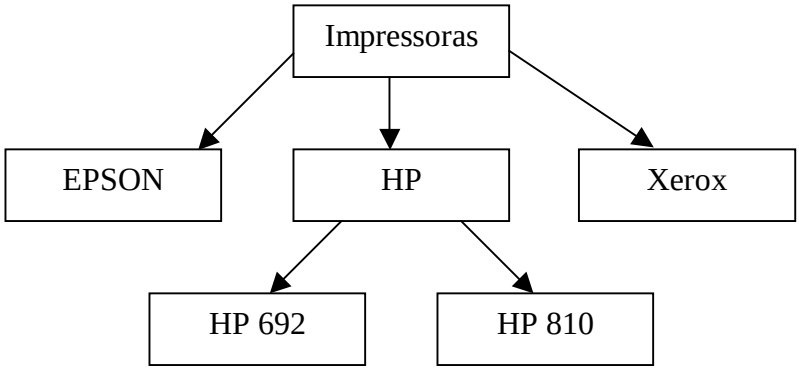
1. Serviço de nomes

O serviço de nomes é utilizado para mapear a ligação de um nome a um objeto. Para realizar alguns experimentos explorando as funcionalidades deste serviço, utilizamos ORBACUS 4.0.5, lua 3.2 e luaorb 1.5.

No ambiente CORBA tudo é representado por objetos. Normalmente estes objetos encapsulam os métodos implementados na aplicação. Caso seja necessário realizar a chamada de algum método de uma aplicação, é necessário que se tenha uma referência do objeto que encapsula o método a ser chamado.

Uma associação nome-objeto é chamada ligação de nomes. O mapeamento de um nome a um objeto sempre é definido relativo ao nome de contexto que pertence. Um contexto é um objeto que contém um conjunto de nomes de ligação (nome/objeto) com a restrição de que cada nome deve ser único, ou seja, em um contexto podemos encontrar um conjunto de nomes que mapeia a ligação a determinados objetos. Um nome de contexto pode ser considerado uma forma de organizar a estrutura de ligação entre os nomes e objetos. Esta organização é definida hierarquicamente. Como exemplo podemos citar uma estrutura organizacional de um conjunto de impressoras. Temos um contexto chamado impressoras e dentro deste contexto encontramos diferentes nomes que especificam categorias de impressoras. Graficamente temos a seguinte situação:



A estrutura hierárquica facilita a organização e a navegação no processo de resolução de nome de um determinado objeto.

1.1 Características do serviço de nomes

- Dar suporte a uma implementação distribuída, heterogênea e administração de nomes e contextos;
- Oferece transparência para clientes que necessitem localizar objetos distribuídos pela rede, abstraindo aspectos da localização física do objeto no ambiente distribuído, e abstraindo o modo de como os servidores implementam os objetos;
- É um serviço de grande importância no processo de gerenciamento de objetos em um ambiente distribuído, que não tem nenhuma dependência de outras interfaces;

- Provê transparência e facilidade de gerenciamento da estrutura organizacional de objetos pertencentes a diferentes contextos através do encapsulamento de contexto.

1.2 Registrando e consultando nomes

Para registrar, listar ou remover a referência de um objeto da base de dados do serviço de nomes deve ser utilizado os métodos definidos nas interfaces do módulo **CosNaming** implementado no arquivo: **CosNaming.idl**. O módulo **CosNaming** contém um conjunto de interfaces que juntas definem o serviço de nomes. O arquivo CosNaming.idl é mostrado nos anexos.

Para registrar e consultar um objeto na base de dados do serviço de nomes deve ser utilizado os métodos “bind” e “resolv” implementados na interface **NamingContext** no módulo **CosNaming**. Como parâmetro estes métodos recebem uma variável do tipo “name”. Essa variável tem a seguinte estrutura:

```
struct NameComponent
{
      Istring id;
      Istring kind;
};
typedef sequence<NameComponent> Name;
```

A variável (*name*) é uma seqüência de estruturas ***NameComponet***. A estrutura (*NameComponet*) possui dois campos do tipo string. O campo id representa o nome do objeto a ser gerenciado pelo serviço de nomes. O campo *kind* pode ser utilizado para acrescentar algum comentário sob o objeto especificado no campo id. O código abaixo mostra um exemplo de como utilizar o método *bind* para registrar um objeto na base de dados do serviço de nomes.

```
name_service:bind({{id = "HelloServant", kind =".cpp"}}, hello_servant)
```

Neste exemplo estamos informando através do campo *kind* que o objeto identificado pelo nome HelloServant foi implementado em c++.

Para adquirir a referência de um objeto no serviço de nomes, infelizmente temos que saber o nome na qual ele foi registrado na base de dados. O código abaixo demonstra a recuperação da referência de um objeto. O processo de resolução do nome de um objeto para sua respectiva referência é realizado através do método ***resolve***.

```
hello = name_service:resolve({{id = "HelloServant", kind =".cpp"}})
```

Como podemos perceber no exemplo acima, registramos um objeto com o nome "HelloServant" utilizando o método *bind*, e para recuperar sua referência deve ser utilizado o mesmo nome ("HelloServant") no método resolve. Essa é uma das grandes limitações do serviço de nomes. Imaginemos um ambiente distribuído onde temos milhares

de objetos em máquinas espalhadas pela rede, que possuem n métodos diferentes. Por exigência do serviço de nomes cada objeto tem que ter um nome único. Sendo assim, se torna praticamente impossível sabermos com qual nome um determinado objeto foi registrado na base de dados do serviço de nomes. Para amenizar este problema de localização dos objetos temos o “serviço de Training”.

A referência de um objeto pode ser armazenada de duas formas. A primeira delas é guardar a referência do objeto em um arquivo local. A outra é armazenar a referência do objeto em uma base dados gerenciada pelo serviço de nomes.

1.3 Armazenando e recuperando a referência de um objeto a partir de um arquivo local

O exemplo abaixo demonstra a implementação de um “hello world” armazenando a referência do objeto em um arquivo local sem utilizar o serviço de nomes.

Primeiro temos a declaração da interface .idl.

Arquivo: hello.idl

```
#ifndef HELLO_IDL
#define HELLO_IDL

module Tst_Hello
{
  interface Hello
  {
    void imprimir_msg(in string msg);
  };
};
#endif
```

Nesta interface, simplesmente declaramos a existência do método “imprimir_msg”. Logo abaixo demonstramos o código da aplicação servidor que contém a implementação do(s) método(s) declarado(s) na interface.

Arquivo: server_hello.lua

```
$debug

if not LuaOrb_cfg.irep then
  LuaOrb_cfg.irep = createproxy{interface="CORBA::Repository", ior_file="/tmp/ir.ref"}
End

-- (o símbolo “--“ indica um comentário no luaorb) Declarando o objeto “hello_impl” que contém o método imprimir_msg()
```

```
hello_impl = {
  imprimir_msg = function (self, msg)
    print(msg)
  end
}

-- Criando e armazenando uma referência do objeto hello_imp em hello_servant.
hello_servant = CreateDSIServer("Tst_Hello::Hello",hello_impl)

-- Escrevendo a referencia do objeto “hello_impl” armazenada em hello_servant no arquivo
server.ref

writeto("server.ref")
write(hello_servant:_get_ior())
writeto()
```

A referência do objeto hello_impl armazenada em “server.ref” possui uma identificação conhecida por IOR com este formato:

```
IOR:01e9ffbf040000004167650001000000000000008800000001010208110000006c656f6
12e6e6174616c6e65742e6272000f02042e000000abacab31393737323231303732005f526f
6f74504f41004c75614f7262504f410000cafebabe3a3f35d0000000000000001000000010000
002c000000010f0b080100010002000000010001052000010009010100040000000010100
010001000100010520000100
```

Agora iremos mostrar uma aplicação cliente que recupera a referência do objeto armazenada no arquivo “server.ref” e realiza uma chamada ao método “imprimir_msg” implementado pelo servidor.

Arquivo: client_hello.lua

```
$debug

  if not LuaOrb_cfg.irep then
    LuaOrb_cfg.irep = createproxy{interface = "CORBA::Repository",
                          ior_file = "/etc/ir.ref"}
  end

-- Recuperando e armazenando a referência do objeto na variável “hello”.
hello = createproxy{interface="Tst_Hello::Hello",ior_file="server.ref"}

-- Fazendo chamada do método implementado pelo servidor
hello:imprimir_msg("Hello World !!!")
```

Através deste exemplo podemos perceber as facilidades de se recuperar a referência de um objeto localmente a partir de um arquivo. Porém, desta forma não é possível recuperar a referência de um objeto em um ambiente distribuído.

Para executar este programa utilizando o ***luaorb*** tem que ser realizado os seguintes passos:

Passo 1: Executar o servidor de repositório de interfaces.
irser –ior > /tmp/ir.ref & -- Este arquivo pode ser gerado em qualquer área do sistema. Desde que seja especificado o caminho exato da sua localização para uma leitura posterior.

Passo 2: luaorb –i console.lua

Chamar o luaorb passando como parâmetro as opções –i console.lua. O arquivo console.lua tem algumas funções que serão utilizadas durante os experimentos.

Passo 3: após executar o luaorb, o repositório de interfaces deverá ser alimentado.

>irfeed(“hello.idl”)

Passo 4: digitar seqüencialmente os comandos dos arquivos server_hello.lua e client_hello.lua ou utilizar o comando dofile(“arquivo”).

>dofile(“server_hello.lua”)
>dofile(“client_hello.lua”)

1.4 Armazenando e recuperando a referência de um objeto a partir de um arquivo local utilizando o serviço de nomes.

O arquivo da interface .idl (hello.idl) não muda nada. Permanece o mesmo arquivo mostrado no exemplo anterior.

O código do servidor (server_naming.lua) que irá registrar um objeto no serviço de nomes é mostrado e comentado logo abaixo.

```
$debug

if not LuaOrb_cfg.irep then
  LuaOrb_cfg.irep = createproxy{interface="CORBA::Repository", ior_file="/tmp/ir.ref"}
end

function unbind()
  name_service:unbind({{id = "HelloServant", kind =""}})
end

-- Declarando o objeto que contém o(s) método(s).

hello_impl = {
  imprimir_msg = function (self, msg)
    print(msg)
```

```
  end
}

-- Criando e armazenando uma referência do objeto hello_imp em hello_servant.

hello_servant = CreateDSIServer("Tst_Hello::Hello",hello_impl)

-- Adquirindo a referência do objeto que contém os métodos bind/resolve/unbind do serviço
de nomes. Este objeto está armazenado no módulo::interface CosNaming::NamingContext
conforme explicado anteriormente.

name_service=createproxy{ior_file="/tmp/naming.ref",interface="CosNaming
::NamingContext"}

-- registrando um objeto localmente no serviço de nomes
name_service:bind({{id = "HelloServant", kind =""}}, hello_servant)

-- END
```

O código do cliente que irá recuperar a referência do objeto hello_impl armazenada no serviço de nomes com o nome "**HelloServant**" é mostrado e comentado abaixo:

Para executar o cliente deverá ser aberto um outro console.lua. Ex. luaorb –i console.lua

```
$debug

if not LuaOrb_cfg.irep then
  LuaOrb_cfg.irep = createproxy{interface="CORBA::Repository",ior_file="/tmp/ir.ref"}
end

-- Adquirindo a referência do objeto que contém os métodos bind/resolve/unbind do serviço
de nomes.
name_service =createproxy{ior_file="/tmp/naming.ref",interface="CosNaming ::
NamingContext"}

-- Resolvendo o nome "HelloServant" para sua respectiva referência. A referência do
objeto registrado no serviço de nomes com o nome "HelloServant" será armazenada na
variável hello_proxy

hello_proxy = name_service:resolve({{id = "HelloServant", kind =""}})

-- o cliente deverá chamar a função narrow para fazer a conversão da referência adquirida
de string_to_object CORBA.
hello_proxy = narrow(hello_proxy)

-- Fazendo chamada do método implementado pelo servidor.
hello_proxy:imprimir_msg("Hello World")
```

Para executar este programa utilizando o ***luaorb*** tem que ser realizado os seguintes passos:

Passo 1: Executar o servidor de repositório de interfaces.
irser –ior > /tmp/ir.ref & -- Este arquivo pode ser gerado em qualquer área do sistema. Desde que seja especificado o caminho exato da sua localização para uma leitura posterior.

Passo 2: Executar a aplicação nameserv . Esta aplicação desempenha as funcionalidades do serviço de nomes. O ORBACUS 4.0.5 já vem com suporte para o serviço de nomes.

nameserv –ior > /tmp/naming.ref&

Passo 3: luaorb –i console.lua

Passo 4: após executar o luaorb, o repositório de interfaces deverá ser alimentado. Devido estarmos utilizando os métodos bind/resolv do serviço de nomes temos que alimentar o repositório com a interface CosNaming.idl

>irfeed(“hello.idl”)
>irfeed(“CosNaming.idl”)

Passo 5: digitar seqüencialmente os comandos dos arquivos server_hello.lua e client_hello.lua ou utilizar o comando dofile(“arquivo”).

>dofile(“server_hello.lua”)

Passo 6: Abrir um outro console.lua para executar o cliente. O cliente pode ser executado no mesmo console do servidor, porém, a função narrow não deverá ser utilizada. Caso seja necessário executar o cliente em console diferente é obrigatório o uso da função narrow para retornar uma referência do tipo CORBA do objeto de um outro console.

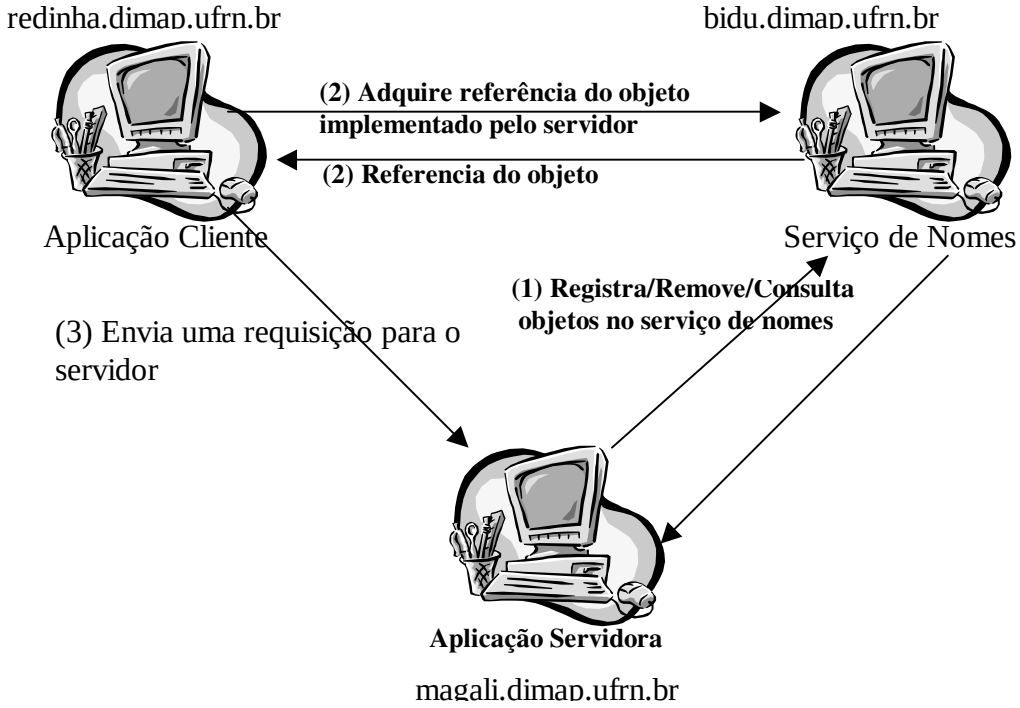
luaorb –i console.lua
>dofile(“client_hello.lua”)

1.5 Armazenando e recuperando a referência de um objeto no serviço de nomes em um ambiente distribuído.

Este exemplo é similar ao mostrado na seção anterior. A diferença é que o servidor poderá registrar um objeto no serviço de nomes que está executando em uma máquina remota e o cliente poderá recuperar a referência deste objeto e utilizar o(s) método(s) implementado(s) pelo servidor.

Para que se torne possível à chamada de qualquer objeto armazenado em uma máquina distribuída deve ser configurado o ORB tanto no servidor para atender requisições dos cientes quanto nos clientes para enviarem/receberem as requisições.

A interação entre cliente/servidor e servidor de nomes é mostrada na figura 1.



Para identificar e localizar a máquina em que o serviço de nomes está executando, pode ser utilizada a facilidade de resolução de nomes realizada pelos servidores DNS. As máquinas na internet possuem uma identificação única que é o seu nome e/ou endereço IP.

Para configurarmos o ORB da aplicação ORBACUS poderemos utilizar os parâmetros CORBALOC e CORBANAME para referenciarmos uma URL.

Ex. corbaloc::bidu.dimap.ufrn.br:5000/NameService

Uma URL corbaloc contém:

- Um endereço que identifica o *host*. Ex. **bidu.dimap.ufrn.br**
- A porta utilizada para atender requisições. Ex. **:5000**
- Uma string que especifica o serviço a ser acessado. Ex. **/NameService**

Existem diversas formas de configurar o ORB. Duas delas utilizam o comando ORBInitRef e ORBconfig. Maiores detalhes sobre a configuração do ORB podem ser encontrados na seção 4 do manual do ORBACUS 4.0.5.

O comando ORBInitRef permite realizar a configuração do ORB através de uma linha de comando.
Ex. nameserv –ORBInitRef NameService = corbaloc:iiop:bidu.dimap.ufrn.br:5000 /NameService &

Ou

Ex. nameserv –ORBInitRef NameService = corbaloc::bidu.dimap.ufrn.br:5000 /NameService &

O identificador "**iiop**" pode ser omitido.

A partir do comando acima estamos definindo no ORB que a aplicação do serviço de nomes está atendendo requisições na máquina bidu.dimap.ufrn.br na porta 5000.

O comando ORBconfig é utilizado para carregar os parâmetros de configuração do ORB a partir de um arquivo.
Ex. nameserv –ORBconfig config.txt &

O arquivo config.txt pode conter o seguinte conteúdo:

ooc.iiop.host=bidu.dimap.ufrn.br
ooc.naming.port=5000
ooc.orb.service.NameService=corbaloc::bidu.dimap.ufrn.br:5000/NameService

Todos possíveis parâmetros que podem ser utilizados na configuração do ORB utilizando o ORBconfig são descritos no manual do ORBACUS.

Se necessário especificar o arquivo que será à base de dados do serviço de nomes o seguinte comando pode ser utilizado. (maiores detalhes podem ser obtidos com as *man pages* do Linux: man nameserv)

Ex. nameserv –s –d /tmp/datanaming –ORBconfig config.txt &

Configurando o Servidor de Nomes

Servidor de Nomes executando na máquina bidu.dimap.ufrn.br

nameserv –s –d /tmp/datanaming –ORBconfig config.txt &

Agora descreveremos um exemplo básico que utiliza o serviço de nomes em um ambiente distribuído.

Configurando e implementando o Servidor

A aplicação que desempenha as funcionalidades de servidor foi executada na máquina magali.dimap.ufrn.br.

O ORB do servidor tem que ser configurado, de modo que ele possa localizar a máquina que está executando o serviço de nomes. O comando ORBconfig é utilizado para carregar os parâmetros de configuração do ORB a partir de um arquivo.
Ex. nameserv –ORBconfig config.txt &

O conteúdo do arquivo config.txt foi mostrado anteriormente.

O arquivo da interface .idl (hello.idl) não muda nada. Permanece o mesmo arquivo mostrado no exemplo (“Hello World”) anterior.

O código do servidor (server_naming.lua) que irá registrar um objeto no serviço de nomes é mostrado e comentado logo abaixo.

```
$debug

if not LuaOrb_cfg.irep then
  LuaOrb_cfg.irep = createproxy{interface="CORBA::Repository", ior_file="/tmp/ir.ref"}
end

-- Declarando o objeto que contém o(s) método(s).

hello_impl = {
  imprimir_msg = function (self, msg)
    print(msg)
  end
}

-- Criando e armazenando uma referência do objeto hello_imp em hello_servant.

hello_servant = CreateDSIServer("Tst_Hello::Hello",hello_impl)

-- Adquirindo a referência do objeto que contém os métodos bind/resolve/unbind do serviço de nomes. Este objeto está armazenado no módulo::interface CosNaming::NamingContext. Ao invés de referenciarmos um arquivo, especificamos uma URL através da opção ior= “URL”

name_service= createproxy{ior="corbaloc::bidu.dimap.ufrn.br:5000/
NameService",interface="CosNaming ::NamingContext"}

-- Esta função pode ser utilizada para remover uma identificação de um objeto da base de dados do serviço de nomes
--  name_service:unbind({{id = "HelloServant", kind =""}})

-- registrando um objeto no serviço de nomes localizado na máquina bidu.dimap.ufrn.br
name_service:bind({{id = "HelloServant", kind =""}}, hello_servant)

-- END
```

Para executar este programa utilizando o ***luaorb*** tem que ser realizado os seguintes passos:

Passo 1: Executar o servidor de repositório de interfaces.
irser –ior > /tmp/ir.ref & -- Este arquivo pode ser gerado em qualquer área do sistema. Desde que seja especificado o caminho exato da sua localização para uma leitura posterior.

Passo 2: luaorb –i console.lua

Passo 3: após executar o luaorb, o repositório de interfaces deverá ser alimentado. Devido estarmos utilizando os métodos bind/resolv do serviço de nomes temos que alimentar o repositório com a interface CosNaming.idl

>irfeed(“hello.idl”)
>irfeed(“CosNaming.idl”)

Passo 4: digitar seqüencialmente os comandos dos arquivos server_hello.lua e client_hello.lua ou utilizar o comando dofile(“arquivo”).

>dofile(“server_hello.lua”)

Configurando e implementando o Cliente

O cliente foi executado na máquina redinha.dimap.ufrn.Br. O código do cliente que irá recuperar a referência do objeto hello_impl armazenada no serviço de nomes com o nome “**HelloServant**” é mostrado e comentado abaixo:

O ORB do cliente também deverá ser configurado, de modo que ele possa localizar a máquina que está executando o serviço de nomes, para que ele possa recuperar a referência do objeto implementado pelo servidor. O comando ORBconfig é utilizado para carregar os parâmetros de configuração do ORB a partir de um arquivo.
Ex. nameserv –ORBconfig config.txt &

O conteúdo do arquivo config.txt foi mostrado anteriormente.

O arquivo da interface .idl (hello.idl) não muda nada. Permanece o mesmo arquivo mostrado no exemplo (“Hello World”) anterior.

```
$debug

if not LuaOrb_cfg.irep then
  LuaOrb_cfg.irep = createproxy{interface="CORBA::Repository",ior_file="/tmp/ir.ref"}
end

-- Adquirindo a referência do objeto que contém os métodos bind/resolve/unbind do serviço
de nomes.
```

```
name_service =createproxy{ ior="corbaloc::bidu.dimap.ufrn.br:5000/
NameService",interface="CosNaming :: NamingContext"}
```

-- Resolvendo o nome “HelloServant” para sua respectiva referência. A referência do objeto registrado no serviço de nomes com o nome “HelloServant” será armazenada na variável hello_proxy

```
hello_proxy = name_service:resolve({{id = "HelloServant", kind =""}})
```

-- o cliente deverá chamar a função narrow para adquirir o tipo apropriado do objeto no formato CORBA. O resolve retorna uma string para hello_proxy, esta string será passada para um tipo object corba através da função narrow.

```
hello_proxy = narrow(hello_proxy)
```

```
-- Fazendo chamada do método implementado pelo servidor.
hello_proxy:imprimir_msg("Hello World")
```

Configurações para executar o cliente em uma máquina remota:

Passo 1: Executar o servidor de repositório de interfaces.
irser –ior > /tmp/ir.ref & -- Este arquivo pode ser gerado em qualquer área do sistema. Desde que seja especificado o caminho exato da sua localização para uma leitura posterior.

Passo 2: Abrir um console.lua para executar o cliente.
Luaorb –i console.lua
>dofile(“client_hello.lua”)

Passo 3: após executar o luaorb, o repositório de interfaces deverá ser alimentado. Devido estarmos utilizando os métodos bind/resolv do serviço de nomes temos que alimentar o repositório com a interface CosNaming.idl

>irfeed(“hello.idl”)
>irfeed(“CosNaming.idl”)

Passo 4: dofile(“client.lua”)

Bibliografia

1 - www.ooc.com
2 - www.omg.org\
3 – www.tecgraf.puc-rio.br/lua
4 – www.tecgraf.puc-rio.br/luaorb
5 - Object Management Group 1998. CorbaServices: Commom object services specification. ftp://ftp.omg.org/pub/docs/formal/98-12-09.pdf. Framing, MA: object Managemente Group.
6 - Object Management Group 1999. Naming Service Specification. ftp://ftp.omg.org/pub/ptc/99-12-03.pdf. Framing, MA: object Managemente Group.
7 - Manual Orbacus. ftp://ftp.orbacus.com/pub/OB/4.0/OB-4.0.5.pdf

ANEXOS

```
// ***************************************************************************
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// ***************************************************************************

#ifndef COS_NAMING_IDL
#define COS_NAMING_IDL

//
// The Naming Service
//
#pragma prefix "omg.org"

module CosNaming
{
   typedef string Istring;

   struct NameComponent
   {
         Istring id;
         Istring kind;
   };

   typedef sequence<NameComponent> Name;

   enum BindingType {nobject, ncontext};

   struct Binding
   {
         Name binding_name;
         BindingType binding_type;
   };

   //
   // Note: in struct Binding, binding_name is incorrectly defined as
   // a Name instead of a NameComponent. This definition is unchanged
   // for compatibility reasons.
   //
   typedef sequence<Binding> BindingList;
```

```
interface BindingIterator;

interface NamingContext
{
      enum NotFoundReason
      {
         missing_node,
         not_context,
         not_object
      };

      exception NotFound
      {
         NotFoundReason why;
         Name rest_of_name;
      };

      exception CannotProceed
      {
         NamingContext cxt;
         Name rest_of_name;
      };

      exception InvalidName { };
      exception AlreadyBound { };
      exception NotEmpty { };

      void bind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

      void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);

      void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

      void rebind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName);

      Object resolve (in Name n)
         raises(NotFound, CannotProceed, InvalidName);

      void unbind(in Name n)
         raises(NotFound, CannotProceed, InvalidName);

      NamingContext new_context();
```

```
        NamingContext bind_new_context(in Name n)
            raises(NotFound, AlreadyBound, CannotProceed, InvalidName);

        void destroy()
            raises(NotEmpty);

        void list(in unsigned long how_many,
                  out BindingList bl, out BindingIterator bi);
    };

    interface BindingIterator
    {
        boolean next_one(out Binding b);

        boolean next_n(in unsigned long how_many,
                       out BindingList bl);

        void destroy();
    };

    interface NamingContextExt : NamingContext
    {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName to_string(in Name n)
            raises(InvalidName);

        Name to_name(in StringName sn)
            raises(InvalidName);

        exception InvalidAddress { };

        URLString to_url(in Address addr, in StringName sn)
            raises(InvalidAddress, InvalidName);

        Object resolve_str(in StringName sn)
            raises(NotFound, CannotProceed, InvalidName);
    };
};

#endif
```