Processamento Distribuído através da Linguagem Python

22/01/2021

Autores:
Antonio Maganhotte Junior - PUC-PR
Henrique Faria - PUC-PR
Luciana Murari Wenceslau - Estagiária da GPT

1 Introdução ao Python

Python é uma linguagem de programação portátil, interpretada, orientada a objetos que possui influências de uma variedade de outras linguagens, como ABC, C, Modula-3 e Icon. A linguagem Python recebeu esse nome para homenagear o grupo humorístico inglês Monty Python, adorado por geeks de todo o mundo. Apesar da associação cômica, Python vem sendo usada em projetos sérios por entidades como Yahoo, NASA, InfoSeek, MCI Worldcom, IBM e Hiway, a maior empresa de hospedagem de web-sites do mundo. É também a base do Zope, a mais sofisticada plataforma para construção de *web-applications* disponível hoje como *open-source*.

Com uma sintaxe elegante e tipos poderosos dos dados, é fácil aprender e ideal para *scripts* CGI, administração de sistemas, e muitas outras tarefas de extensão e de integração.

A potência real de Python, entretanto, encontra-se em sua capacidade de extensão. A linguagem pode ser estendida pela escrita de módulos em Python ou linguagens compiladas, tais como C e C++. Estes módulos podem definir variáveis, métodos, tipos novos de dados e seus métodos, ou fornece simplesmente uma ligação às bibliotecas existentes de código. É também possível encaixar o interpretador Python em uma outra aplicação para o uso como uma linguagem de extensão. A biblioteca Python padrão, inclui os módulos para uma larga escala de tarefas, de *debuggers* e *profilers* aos serviços de Internet/Web, e relações de interfaces gráficas. Python funciona sob muitos ambientes incluindo a maioria dos tipos de Unix, Windows 3.x, Windows 95 e NT, Macintosh, e OS/2. É de distribuição gratuita e pode ser usado sem taxa em produtos comerciais.

Apesar de sua sintaxe simples e clara, Python oferece os seguintes recursos,

disponíveis também em linguagens mais complicadas como Java e C++:

- programação orientada a objetos (incluindo herança múltipla, conceito apenas parcialmente presente em Java);
- exceções, um moderno mecanismo para o tratamento de erros;
- módulos, uma forma inteligente de acessar e organizar código a ser reutilizado;
- coleta de lixo automática, sistema que elimina os erros causados pelo acúmulo de dados inúteis na memória do computador (característica presente também em Java, mas não em C++);
- recursos avançados de manipulação de textos, listas e outras estruturas de dados;
- possibilidade de executar o mesmo programa sem modificações em várias plataformas de hardware e sistemas operacionais (uma virtude de Java, mas difícil de se conseguir em C++)

Em resumo, Python nos oferece uma sintaxe simples, mas ao mesmo tempo suporta a maior parte das características importantes de linguagens modernas e amplamente utilizadas como Java, C++, Perl e VBScript.

Python pode ser considerado com um pseudo-código. As variáveis não têm tipos, assim não precisam ser declaradas. Elas são criadas quando um valor é atribuído a elas, e são destruídas quando não forem mais utilizadas. A atribuição é feita pelo operador =. A igualdade é testada com o operador ==. Pode ser atribuída mais de uma variável ao mesmo tempo:

```
x, y, z = 1, 2, 3

primeiro, segundo = segundo, primeiro

a = b = 123
```

Em Python, os blocos são indicados somente pela identação (nada de *BEGIN/END* ou chaves). Algumas estruturas de controle são:

```
if x < 5 or 10 < x < 20:
    print "O valor está correto."
for i in [1, 2, 3, 4, 5]:
    print "Esta é a iteração número", i
```

```
x = 10

while x >= 0:

    print "x ainda não é negativo."

    x = x – 1
```

A variável de índice no laço for varia de acordo com os elementos de uma lista (escrita como no exemplo). Para fazer um laço *for* comum (isto é, um laço de contagem), usa-se a função *range*().

```
# Mostra os valores de 0 a 99 inclusive.

for valor in range(100):

print valor
```

A linha iniciada pelo caracter "#" é um comentário, sendo ignorada pelo interpretador. Mais importante ainda é sua sustentação para prototipagem rápida, e programação orientada a objetos, que fazem dela uma ferramenta valiosa para a tecnologia de programação séria e o desenvolvimento de produto. O pequeno núcleo fornece as listas básicas usuais, os tipos de dados e indicações de controle do fluxo junto com tipos de alto nível tais como strings, tuplas, e disposições associativas. A programação orientada a objetos é suportada por um mecanismo de classes com um modelo de herança múltipla.

Definir classes em Python, também é muito simples:

```
class Quadrado:

    pass
```

Todos os métodos recebem um argumento adicional no início da lista de argumentos, contendo o próprio objeto. O modo de chamar um método é:

```
objeto.método(argumentos)
```

Alguns nomes de métodos, como *__init__* são pré-definidos, e têm significado especial. *__init__* é o nome do construtor da classe, isto é, esta é a função que é chamada quando uma instância é criada.

Nenhum método ou variável membro é protegido (nem privado) em Python.

A maioria das funções e classes mais úteis, são colocadas em módulos, os quais são, na verdade, arquivos-texto contendo códigos Python. Esses módulos podem ser importados e utilizados em qualquer programa. Por exemplo, para usar o método *split* do módulo padrão *string*, podem ser utilizadas essas duas formas:

import string

x = string.split(y)

ou

from string import split

x = split(y)

Mais informação (incluindo código fonte) pode ser obtida em http://www.python.org/.

2 Razões para se Usar Python na Programação Distribuída

Python é uma linguagem de propósito geral que traz o poder e a flexibilidade para desenvolver *scripts* de um sistema. Isto satisfaz os três requisitos mais importantes da programação: (1) Um mecanismo de persistência de objetos baseado em chaves (TRIPcode), (2) Suporte a CORBA graças ao mapeamento CORBA Python, um exemplo é o pacote Fnorb a ser descrito na sequência deste trabalho e (3) um mecanismo de notificação assíncrona de acordo com os Serviços de Evento CORBA. Como alternativa, uma linguagem como C++ ou Java pode ser adotada devido a uma melhor performance, especialmente se considerarmos o fato de que o Serviço de Monitoração TRIP, CORBA e C++ tem sido largamente usado. Entretanto, em boa parte dos casos, Python deve ser escolhido pela rapidez no desenvolvimento destas aplicações.

O módulo *shelve* do Python, provê um mecanismo baseado em chaves que possibilita a criação de *arrays* associativas de persistência de objetos. Criando ou acessando instâncias do tipo *shelve*, é tão fácil quanto manipular dicionários Python. *Shelves* facilita a criação de base de dados de objetos nativos Python, porque não existe a necessidade de usar outras API, manusear base de dados, especificar estruturas de dados, ou conversões de um para outro.

A já existente IDL OMG para Python, é mapeada por ambas implementações: o Xerox´s ILU ou o Fnorb da universidade de Queensland, que provêem a

capacidade de programação distribuída CORBA. A simplicidade de mapeamento, comparado com C++ ou Java, faz destas implementações CORBA uma ferramenta ideal para a rápida prototipagem.

Python, com suas características orientadas a objetos, integra-se com o modelo de objetos CORBA. Python livra os programadores da complicada gerência de memória como visto no mapeamento CORBA C++. As características dinâmicas removem a tediosa necessidade de *type-casting* e declaração de longas variáveis, como ocorrem em C++ e no mapeamento Java CORBA. São suportados ainda todos os tipos de dados CORBA 2.0 e provê uma completa implementação do IIOP.

3 Hector: Objetos Distribuídos em Python

Esse item é a análise do artigo Hector: Distributed Objects in Python[1].

Hector é um sistema de objetos distribuídos, primariamente escrito em Python. Ele provê uma camada transparente de comunicação, permitindo um protocolo de comunicação e negociação de informações.

3.1 Motivação

Projetos orientados a objetos se tornaram uma abordagem largamente aceita em computação. É ótimo para ambientes distribuídos onde a encapsulação e subseqüentemente a independência dos objetos, permite a distribuição de uma aplicação. Entretanto, a programação de sistemas objeto distribuído é difícil, complicada pelas possibilidades de arquiteturas de máquinas heterogêneas, distâncias físicas e falhas em componentes independentes.

3.2 CORBA

O CORBA (Common Object Request Broker Architecture), é uma especificação da OMG, agora largamente adotada pela indústria, e apesar da sua imaturidade em algumas áreas, é adotado como uma arquitetura inteiramente orientada a objetos.

3.3 COM

Somente para citar uma especificação concorrente ao CORBA, temos o Microsoft Common Object Model, que é uma proposta de extensão distribuída da já existente tecnologia Object Linking and Embedding (OLE), agora conhecida como ActiveX. Neste ponto, COM é mais especificação do que implementação, mas é destinado a se tornar uma significante influência.

3.4 HECTOR

Temos então o Hector, uma estrutura que se situa acima de outros ambientes distribuídos, fornecendo a negociação e a interoperabilidade de protocolos de comunicação, a descrição aberta do nível elevado de serviços componentes e de suas exigências, um jogo rico de serviços de sustentação para objetos e uma estrutura da interação que permite a descrição *workflow*, como de interações entre objetos autônomos. O protótipo é executado quase inteiramente em Python, fazendo uma análise das vantagens da larga sustentação da linguagem para outros serviços e a sintaxe poderosa, permitindo o desenvolvimento rápido.

3.5 Programação Distribuída

Uma comunicação entre objetos distribuídos é modelada tradicionalmente como a invocação remota dos métodos. Isto pode ser feito de duas maneiras: RPC e mensagem. A escolha depende se o método tem um valor de retorno, ou não. Os métodos que um objeto remoto exporta são chamados coletivamente com sua interface. Tipicamente, selecionando um objeto remoto identificado normalmente por algum tipo opaco de endereço, o programador pode invocar um método na interface remota.

O Hector fornece uma classe obrigatória abstrata que deve ser especializada por programadores da aplicação. O Kernel pode instanciar estas classes derivadas. O processo de instanciar estabelece os canais de comunicação descritos pelo tipo obrigatório (binding).

Por exemplo, as exigências de um *scheduler* de encontro distribuído podem ser especificadas no tempo de programação como calendários de alguns indivíduos, abrangendo o tipo obrigatório (binding), serviços de reserva do quarto e de equipamento, notificação a um serviço de notícia e o componente próprio do *scheduler*. As construções dos canais de comunicação são requeridas par-a-par no *runtime* e é uma saída para uma melhor infra-estrutura explicitamente programada.

3.6 Serviços

Os objetos requerem serviços, e em um ambiente distribuído, alguns serviços devem estar disponíveis através da rede. Alguns daqueles serviços podem ser julgados fundamentais, por razões de dependência ou como meios de assegurar um ambiente de programação mais rico. O Hector garante o fornecimento de um jogo fixo de serviços fundamentais em cada posição. A execução real dos

serviços pode variar, mas a relação do serviço fornecida aos objetos é padronizada e garantida para estarem disponíveis.

Foram selecionados nove serviços fundamentais, com uma ênfase particular em fornecer um ambiente rico para objetos. Estes serviços são:

- Notificação:

Geração da notificação da informação e da subscrição do evento para receber a notificação dos eventos usando expressões sobre a informação do evento.

- Autenticação:

Autenticação da identidade do objeto. Estas credenciais são usadas para verificações de autorização durante uma ligação (bind).

- Privacidade:

Ambas, comunicação entre objetos e o armazenamento de informações podem requerer privacidade. O serviço de privacidade provê um mecanismo para encriptar informações.

- Banco:

Os objetos precisam pagar para usar alguns serviços de forma a manter seus "créditos" através de invocações. Objetos podem abrir contas e acessá-las usando tokens de identificação.

- Nomenclatura:

O serviço de nomenclatura provê um mecanismo para associar informações com uma string nome.

- Gerência de Tipos:

É um repositório de descrição de tipos. Um simples tipo pode ter sua descrição

em várias linguagens de comentários em inglês, passando por especificações formais, até especificações executáveis.

- Relacionamento:

Objetos freqüentemente precisam manter registro de seu relacionamento com outros objetos. Este serviço provê um repositório de relacionamento, fortemente tipificado, o qual pode ser solicitado por objetos para reaver estas informações.

- Negociação:

Quando um objeto requer o uso de um serviço, ele pode localizar instâncias daquele tipo de serviço de várias formas. O serviço de negociação provê um sistema baseado em conteúdo do tipo "Páginas Amarelas", parametrizadas pelo tipo de serviço e qualidade dos atributos.

Provedores de serviço avisam sobre seus serviços, descrevendo seus atributos. Consumidores de serviço questionam um Negociador, o qual retorna uma lista de possíveis provedores.

4 Mapeamento CORBA Python Fnorb

Fnorb é um Object Request Broker (ORB) compatível com a especificação de CORBA 2.0 do grupo de gerência de objetos Object Management Group (OMG). Fnorb é escrito quase inteiramente na língua de programação Python e executa uma única linguagem que traça de OMG IDL a Python. Por causa da natureza interpretada e interativa de Python, e a simplicidade de mapear (comparado a C++ e Java), Fnorb é ideal como uma ferramenta para prototipagem rápida, testes e para descrição de sistemas e de arquiteturas CORBA.

Para exemplificar como funciona o Fnorb, usaremos como exemplo a implementação de "Olá Mundo!". Neste caso será executado como um sistema cliente/servidor, com um objeto do servidor em um processo que oferece a operação do helloWorld(), e um cliente com um outro processo que encontre o objeto do servidor e invoque a operação nela.

4.1 Definindo a interface em OMG IDL

A primeira etapa em desenvolver um sistema CORBA é definir a(s) interface(s) em OMG IDL. No exemplo "Olá Mundo!", isto é trivial porque nós temos uma

única relação, contendo uma única operação. Entretanto, vale a pena notar o uso da diretriz orientadora do prefixo do #pragma, e a definição do módulo OlaMundo contendo nossa relação. Estes mecanismos asseguram-se de que não poluímos o namespace global e devamos ser adotados por cada bom cidadão CORBA! O IDL mostrado abaixo pode ser encontrado no arquivo: "Fnorb/examples/hello-world/HelloWorld.idl"

```
    #pragma prefix "dstc.edu.au"

    //

    // "Hello World" example!

    //

    module HelloWorld {

        const string Message = "Hello CORBA World!";

        interface HelloWorldIF {

        string hello_world();
    };

};
```

4.2 Gerando os módulos Stub e Skeleton

Em seguida certifica-se de que as definições de IDL estejam válidas, e cria-se o código Stub e do esqueleto usando o *fnidl* do compilador de IDL. Isto é feito na linha de comando, com o simples comando: $ do *fnidl* HelloWorld.idl quando o comando termina (esperançosamente) vê-se que dois pacotes Python foram gerados; HelloWorld que contém todas as definições do tipo que o código Stub requer do cliente, e HelloWorld_skel que contém o código de esqueleto requerido pelo servidor.

4.3 Escrevendo o cliente "Olá Mundo!"

Agora, vamos dar uma olhada no código do cliente "Olá Mundo!", que pode ser encontrado em "Fnorb/examples/hello-world/client.py". Em um típico código Python, começamos importando os módulos e pacotes necessários para a aplicação. Em todo programa Fnorb, ambos, cliente e servidor devem importar o

módulo CORBA do pacote Fnorb.orb. O cliente deve, ainda, importar o código Stub para qualquer interface que ela precisar usar. O código Stub é gerado pelo compilador fnidl e, neste exemplo, o código que precisamos pode ser encontrado no pacote HelloWorld.

```
#!/usr/bin/env python

    """ Client for the "Hello World" example. """

# Standard/built-in modules.

import sys

# Fnorb modules.

from Fnorb.orb import CORBA

# Stubs generated by `fnidl'.

import HelloWorld
```

A primeira coisa que todo o programa Fnorb deve fazer é inicializar o ORB. Isto se consegue chamando o método ORB_init() encontrado no módulo CORBA. ORB_init() faz exame de dois parâmetros. O primeiro é uma lista das opções da linha de comando que podem ser usadas para configurar o ORB, e o segundo é a identidade especificada pelo desenvolvedor ORB, que em Fnorb deve sempre ser o valor do CORBA.ORB_ID variável. O método ORB_init() pode ser chamado tantas vezes quantas você precisar e retorna sempre uma referência ao ORB previamente inicializado (embora você deva estar ciente de que os argumentos da linha de comando são processados somente no primeiro atendimento). Para a conveniência, ORB_init() pode também ser chamado sem parâmetros.

```
def main(argv):

    """ Do it! """

    print 'Initialising the ORB...'

    # Initialise the ORB.

    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
```

Inicializado o ORB, o cliente deve agora encontrar o objeto do servidor. Neste caso o servidor começa escrevendo um campo string com a versão deste objeto

como referência ao arquivo "server.ref". Obviamente, isto não é uma solução ideal como requerem o cliente e o servidor, que compartilham o mesmo sistema de arquivo, mas servirá para este exemplo. Na prática, o cliente deve localizar o servidor usando um mecanismo mais flexível como os serviços de nome e/ou negociação do CORBA.

stringified_ior = open('server.ref', 'r').read()

A próxima chamada é do método ORB string_to_object(). Este método cria um objeto "proxy" que possui os mesmos métodos que o objeto servidor. Quando métodos são invocados no proxy, este redireciona o pedido para o objeto servidor em qualquer processo, em qualquer máquina que acontecer de estar rodando naquele instante. Desta forma, o cliente fica isolado dos detalhes de onde o objeto servidor está fisicamente localizado, e como os pedidos de operações chegam até ele.

server = orb.string_to_object(stringified_ior)

Só porque string_to_object foi chamada, isto não significa que possamos utilizar o servidor diretamente. Primeiro de tudo, a referência pode ser uma "referência nula a um objeto". Referências nulas a um objeto, é como CORBA representa referência a nada, um ponteiro no espaço. Em Fnorb, referência a objetos nulos são representadas pelo valor Python None, logo, a primeira verificação que temos de fazer é a seguinte:

if server is None:

    raise 'Nil object reference!'

Embora agora nós saibamos que a referência ao objeto não é nula, isto ainda não significa que o servidor está realmente rodando. O único meio de termos certeza disto, é na tentativa de invocar uma operação (via objeto proxy). Uma boa escolha é a operação _is_a() a qual está disponível em todos objetos CORBA. O método _is_a() pega um identificador no repositório de identificadores e pergunta ao objeto se ele implementa uma interface daquele tipo. Se a chamada for bem sucedida, então o cliente pode ter certeza que:

a. o objeto servidor está vivo, e
b. o objeto servidor realmente implementa a interface esperada.

if not server._is_a('IDL:dstc.edu.au/ HelloWorld/HelloWorldIF:1.0'):

raise 'This is not a "HelloWorldIF" server!'

Agora que o cliente tem certeza que o servidor diz ser quem é, ele pode fazer todas as chamadas, como se fosse uma instância Python local.

print server.hello_world()

return 0

if __name__ == '__main__':

 # Do it!

sys.exit(main(sys.argv))

Escrever um servidor para o nosso exemplo, requer um pouco mais de trabalho. Nesta sessão vamos discutir o código do servidor, encontrado no exemplo "Fnorb/examples/ hello-world/server.py'". Conforme mencionado anteriormente, todos os programas Fnorb, ambos, cliente e servidor, devem importar o módulo CORBA do pacote Fnorb.orb. Os servidores Fnorb devem, ainda, importar um adaptador básico de objetos no módulo BOA. O BOA é uma implementação da interface de adaptador básico de objetos definidos nas especificações CORBA 2.0 e é usado para conectar implementação de interfaces para o ORB. Infelizmente, a especificação estava um pouco obscura para permitir todos os desenvolvedores ORB implementarem o BOA de maneira própria, e para endereçar este problema, a OMG aprovou a especificação de um bem definido e mais portável adaptador de objeto conhecido como POA. Versões futuras do Fnorb irão conter uma implementação do POA, e o uso do Fnorb BOA deve, eventualmente, tornar-se deprecado. Como no cliente, o servidor importa o código Stub gerado do pacote HelloWorld. Estritamente falando, isto não é necessário se o servidor não utilizar nenhum tipo de usuário definido IDL. Entretanto, na prática, quase todas as interfaces irão usar tipos definidos pelo usuário, e é uma boa prática importar os pacotes Stubs (no nosso exemplo, o servidor usa a constante HelloWorld.Message e o módulo Stub é necessário). Mais importante, o servidor deve ainda incluir códigos Skeletons para cada interface que implementa isto. Como o código Stub no cliente, o código Skeleton é gerado pelo compilador Fnorb IDL fnidl, e neste caso, é encontrado no pacote HelloWorld_skel.

#!/usr/bin/env python

```
""" Implementation of the HelloWorldIF interface. """

import sys

from Fnorb.orb import BOA, CORBA

import HelloWorld_skel
```

Agora, vamos dar uma olhada na classe Python que implementa a interface que definimos na IDL. A única diferença entre definir uma classe servidor Fnorb em oposição a uma classe padrão Python, é que esta precisa herdar o Skeleton gerado pelo compilador IDL. A classe Skeleton é encontrada no pacote skeleton e tem o mesmo nome da interface IDL com o sufixo "_skel". A classe servidor deve conter implementação para todas as operações definidas na IDL, que neste caso significa que nós temos que implementar uma simples operação hello_world(). Note que na IDL a operação é definida sem parâmetros, mas, na sua implementação Python, nós temos que explicitamente adicionar o parâmetro da instância corrente conhecida como self.

```
class HelloWorldServer(HelloWorld_skel.HelloWorldIF_skel):
    """ Implementation of the 'HelloWorldIF' interface. """

def hello_world(self):
    print HelloWorld.Message

    return HelloWorld.Message
```

Agora vamos dar uma olhada em como criamos uma instância do servidor, e tornarmos ele avaliável para os clientes. Como no cliente, a primeira coisa que um servidor Fnorb deve fazer é inicializar o ORB usando o método ORBinit().

```
def main(argv):

    """ Do it! """

    print 'Initialising the ORB...'

    # Initialise the ORB.

    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
```

Agora vem um passo adicional, necessário apenas pelos servidores Fnorb: a inicialização do BOA que será usado para "conectar" implementações da interface ao ORB. O método de inicialização do BOA é chamado BOA_init(), e assim como o método ORB_init(), ele pode ser chamado múltiplas vezes, com a segunda e subseqüentes chamadas, simplesmente retornando uma referência para a instância inicial BOA.

```
print 'Initialising the BOA...'

# Initialise the BOA.

boa = BOA.BOA_init(argv, BOA.BOA_ID)
```

No mundo CORBA, objetos são identificados por uma opaca estrutura de dados conhecida como referência de objetos. Para criar uma referência a objeto para nossa implementação será usado o método create() no BOA. O Método create() pega dois argumentos, a chave do objeto, e o tipo de interface.

```
print 'Creating object reference...'

# Create an object reference ('fred' is the object key).

obj = boa.create('fred', HelloWorldServer._FNORB_ID)
```

Agora nós simplesmente criamos uma instância da nossa classe implementada, como uma classe Python qualquer.

```
print 'Creating implementation...'

impl = HelloWorldServer()
```

Antes do ORB conseguir direcionar qualquer pedido de operação para nossa implementação, nós temos de "conectar" isto à referência do objeto que criamos anteriormente. Note que embora após este passo esteja completo, a implementação não irá receber nenhum pedido enquanto não for inicializado o laço de eventos Fnorb.

```
print 'Activating the implementation...'

boa.obj_is_ready(obj, impl)
```

Como mencionado, quando nós criamos o cliente, este exemplo usa um arquivo

para advertir a referência do objeto servidor. Aqui nós usamos um método ORB string_to_object() para mudar a referência do objeto em um campo formato string e então escrever isto no arquivo "server.ref"

print 'Server created and accepting requests...'

O passo final é inicializar o loop de evento do BOA. Ou seja, dizer ao BOA para escutar requisições de operações e para passar então para o objeto apropriado. Note que o método _fnorb_mainloop() nunca irá retornar, a não ser que uma exceção seja disparada, ou se o método _fnorb_quit() for chamado.

boa._fnorb_mainloop()

return 0

##############################

if __name__ == '__main__':

 # Do it!

sys.exit(main(sys.argv))

##############################

A maneira mais fácil de testar o exemplo é rodar o cliente e o servidor em janelas separadas. Em uma janela inicialize o servidor:

python server.py

Se o servidor funcionar como esperado, deve-se observar a seguinte saída:

$ python server.py

Initialising the ORB...

Initialising the BOA...

Creating object reference...

Creating implementation...

Activating the implementation...

Server created and accepting requests...

Em outra janela, rodamos o cliente usando:

python client.py

Novamente, se tudo ocorrer como planejado, devemos observar a seguinte saída:

$ python client.py

Initialising the ORB...

Hello CORBA World!

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ARNOLD, D. et al. Hector: distributed objects in Python. Australia: University of Queensland. Disponível em: <http://www.python.org/workshops/1996-06/papers/d.arnold/paper.html>. (CRC for Distributef Systems Technology).

[2] BEAZLEY, D. M. Python essential reference. USA: New Rider, 1999.

[3] FNORB tutorial. Disponível em: <http://www.fnorb.org>.

[4] PYTHON. Disponível em: <http://www.python.org>.