Arquitetura de Software para Aplicações Distribuídas

Marco Túlio de Oliveira Valente

Middleware

Marco Túlio de Oliveira Valente

Middleware

- Desenvolver uma aplicação distribuída é mais difícil do que desenvolver uma aplicação centralizada
- Problemas típicos: comunicação, heterogeneidade, falhas, concorrência, segurança, escalabilidade etc

- Middleware: infra-estrutura de software que fica entre o sistema operacional e uma aplicação distribuída
- Objetivo: tornar mais simples e produtivo o desenvolvimento de uma aplicação distribuída

- Idéia: oferecer abstrações/recursos de mais alto nível que tornem transparente ao programador detalhes de programação em redes
- Fazer com que programação distribuída seja o mais semelhante a programação centralizada

- Wolfgang Emmerich. Software engineering and middleware: a roadmap. ICSE 2000: 117-129.

Middleware

- Middleware e linguagens de programação:
  - LP: escondem o hardware (registradores, instruções de máquina, modos de endereçamento, periféricos etc)
  - Middleware: escondem a rede (endereços IP, portas, sockets, formato de mensagens, conexões, falhas etc)
  - Veja que hoje “a rede é o computador”

- Classificação de sistemas de middleware:
  - De acordo com o nível de abstração provido
  - De acordo com o tipo de abstração provida
  - De acordo com o domínio de aplicação ou tecnologia de rede

Middleware: Níveis de Abstração

- Sistemas de middleware que fornecem uma infra-estrutura para execução de aplicações (*host infrastructure middleware*)
  - Abstraem e unificam em uma API particularidades de um hardware ou sistema operacional
  - Fornecem primitivas para criação de threads, sincronização, comunicação via TCP ou UDP etc
  - Exemplos: Sun JVM, .NET CLR

- Sistemas de middleware que fornecem uma infra-estrutura de comunicação (*communication ou distribution middleware*)
  - Exemplos: Java RMI, CORBA, DCOM, SOAP, .NET Remoting Framework, JavaSpaces, Serviços Web etc
  - Uso mais comum do termo middleware

Middleware: Níveis de Abstração

- Sistemas de middleware que fornecem outros serviços, além de comunicação (*common based middleware*)
  - Serviços: persistência, transações, nomes, autenticação etc
  - Exemplos: EJB, CCM (Corba Component Model)
  - Conhecidos como middleware baseados em componentes

- Requisitos funcionais: “o que” o sistema faz (ou seja, as funcionalidades do sistema)
- Requisitos não-funcionais: “como” o sistema funciona (atributos de qualidade)
  - Exemplos: distribuição, persistência, desempenho, segurança, concorrência, confiabilidade, tolerância a falhas

Middleware: Abstrações de Comunicação

- Classificação baseada nas primitivas de comunicação entre componentes fornecidas pela plataforma de middleware
- Exemplos:
  - Chamada remota de procedimentos (Sun RPC, DCE RPC etc)
  - Chamada remota de métodos (Java RMI, CORBA, DCOM etc)
  - Mensagens/eventos (JMS, IBM MQSeries etc)
  - Espaços de tuplas (TSpaces, Java Spaces etc)
  - Transações (CICS, MTS, JTS, Encina, Tuxedo etc)

Middleware: Domínios de Aplicação

- Middleware para domínios de aplicação específicos (*domain specific middleware*):
  - Exemplos: telemedicina, ensino a distância, comércio eletrônico, jogos digitais, multimídia, vídeo sob demanda etc
  - Nestes casos, o middleware é um *framework* (esqueleto de uma aplicação que vai ser “terminada” pelo usuário do middleware)

- Middleware para tecnologias e ambientes de rede específicos:
  - Exemplos: computação móvel, grades computacionais (*grids*), sistemas embutidos, TV Digital etc

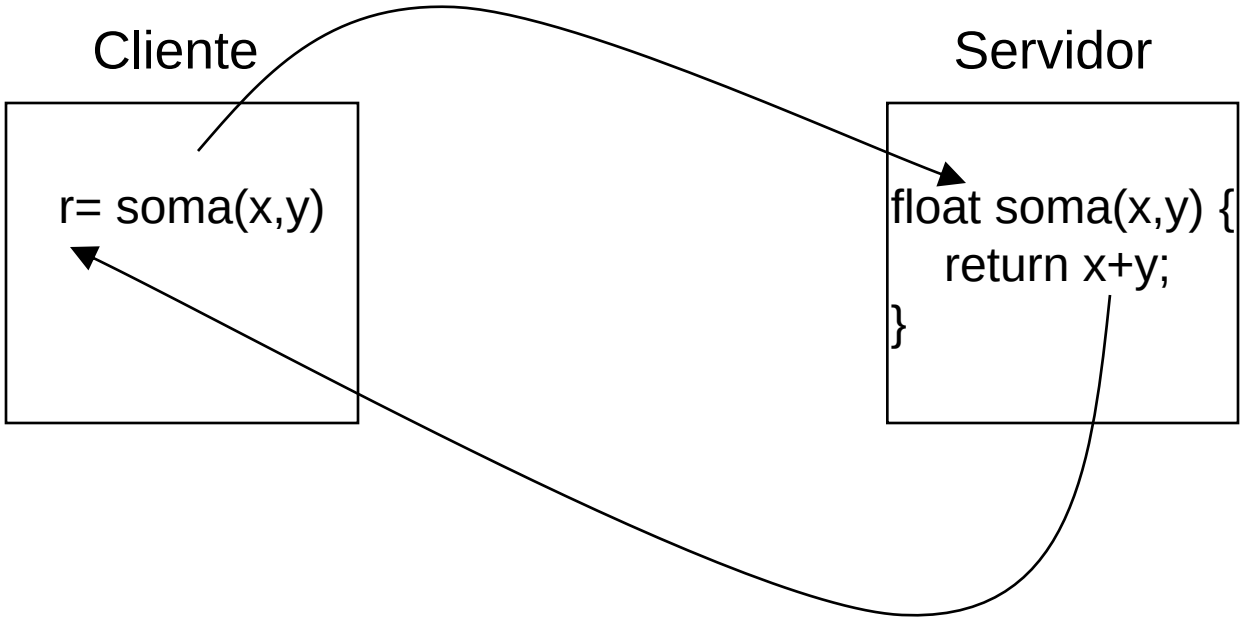
Próximos Capítulos

- RPC
- Middleware Orientado por Objetos: Visão Geral
  - Java RMI
  - CORBA
- Middleware Orientado por Objetos: Recursos Avançados
  - Chamadas assíncronas
  - Reflexão computacional
- Web Services, Estilos Arquiteturais e REST
- Arquiteturas orientadas a serviços

RPC

- Birrell, A.D. & Nelson, B.J. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems 2,* 1 (Feb. 1984): 39-59.
- “The primary purpose of our RPC project was to make distributed computation easy. Previously, it was observed within our research community that the construction of communicating programs was a difficult task, undertaken only by members of a select group of communication experts. Even researchers with substantial systems experience found it difficult to acquire the specialized expertise required to build distributed systems with existing tools. This seemed undesirable ....”
- “The existing communication mechanisms appeared to be a major factor constraining further development of distributed computing. Our hope is that by providing communication with almost as much ease as local procedure calls, people will be encouraged to build and experiment with distributed applications”.

RPC: Idéia Básica



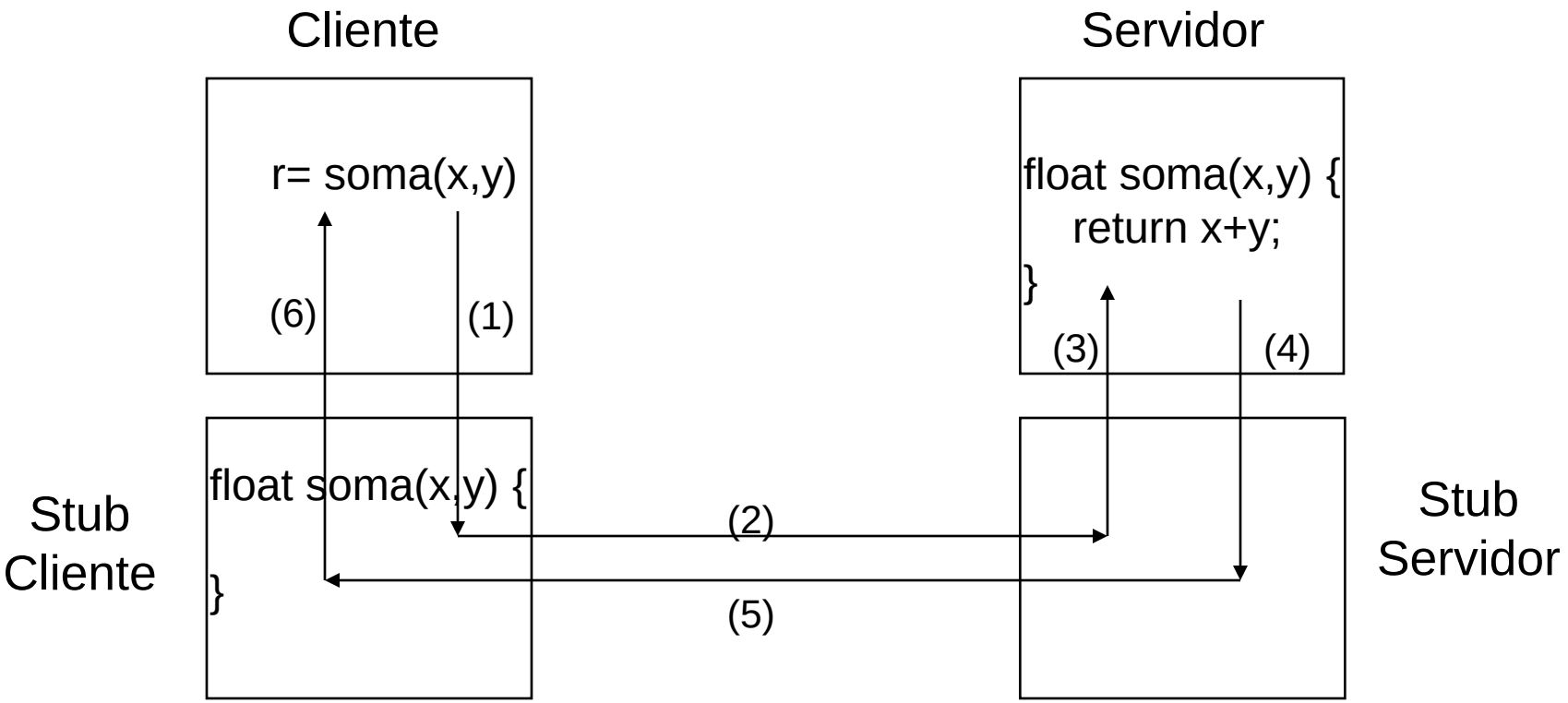
RPC: Marshalling e Unmarshalling

- Quando cliente A chama procedimentos de um servidor B:
  - Pode ser necessário passar como parâmetro (e receber como resultado) estruturas mais complexas (objetos, vetores etc)
- Cliente A:
  - Deve converter os parâmetros para um formato que possa ser transmitido por um protocolo de transporte (TCP, por exemplo)
  - Este processo é chamado de *marshalling*
- Servidor B:
  - Deve converter mensagem recebida em uma estrutura de dados
  - Este processo é chamado de *unmarhalling*
- Processo de marshalling/unmarshalling é tedioso e sujeito a erros
- Portanto, deve ser automatizado pela plataforma de middleware

RPC: Stubs

- Funcionamento interno baseado em módulos chamados de stubs
- Stubs: encapsulam detalhes de comunicação no cliente e servidor
  - Automatizam processo de marshalling e unmarshalling
  - Automatizam comunicação com o processo remoto (interagindo com o protocolo de transporte)
- Marshalling/unmarshalling realizado por stubs é chamado de estático

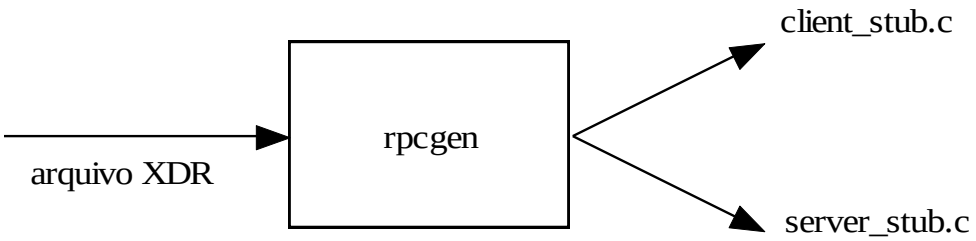
RPC: Arquitetura Interna



- Questão fundamental: Quem gera os stubs?
- Evidentemente, geração manual não faz sentido

RPC: Arquitetura Interna

- Stubs são gerados automaticamente, por um compilador de stubs
- Entrada deste compilador: assinatura dos procedimentos remotos, em uma linguagem chamada XDR (External Data Representation)
  - Genericamente, conhecida como linguagem para definição de interfaces
- Saída deste compilador: código fonte dos stubs

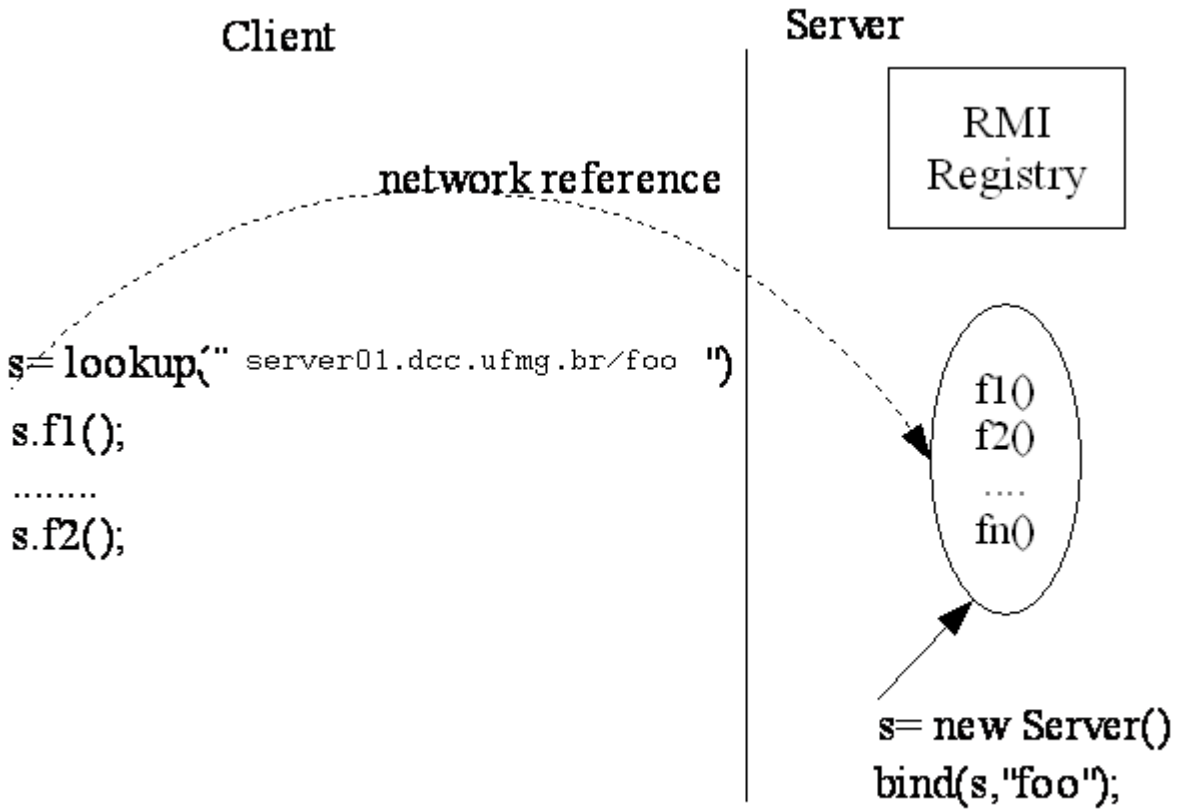


Sistemas de Middleware Orientados por Objetos

- Extensão de RPC para o paradigma orientado por objetos
- Idéia básica: objetos clientes podem chamar métodos de objetos remotos com transparência de acesso (isto é, com a mesma sintaxe de chamadas locais)
- Principal abstração: chamada remota de métodos (RMI)
- Objetos remotos são manipulados usando-se referências de rede
- Sistemas: CORBA, Java RMI, DCOM, .NET Remoting etc
- Servidor (de aplicação): instancia objeto remoto e o registra no servidor de nomes (registrar= dar um nome para o objeto)
- Servidor de nomes: tabela (nome, referência de rede)
- Cliente: consulta servidor de nomes (fornecendo o nome do objeto), obtém uma referência de rede para o objeto remoto), usa essa referência para realizar chamadas remotas

Java RMI

- Middleware para programação distribuída em Java
- Implementado como um pacote da linguagem
- Ann Wollrath, Roger Riggs, Jim Waldo. A Distributed Object Model for the Java System. COOTS 1996



Java RMI: Exemplo de Aplicação

- Codificar um servidor com um objeto da seguinte classe:

```
class ContaImpl .... { ......
   float obterSaldo ();
   ........
}
```

- Codificar um cliente que chame remotamente o método obterSaldo() deste objeto

Java RMI: Exemplo de Aplicação

- Passo 1: Definição da interface remota (assinatura dos métodos que serão invocados remotamente)

```
// arquivo Conta.java
 import java.rmi.Remote;
 import java.rmi.RemoteException;

 public interface Conta extends Remote {  ......
   float obterSaldo () throws RemoteException;  ......
 }
```

Java RMI: Exemplo de Aplicação

- Passo 2: Implementação da Interface Remota (servidor)

```
// arquivo ContaImpl.java
import java.rmi.*;
import java.rmi.server.*;
class ContaImpl extends UnicastRemoteObject implements Conta {
  private int numero;  private float saldo= 0.0;
  public ContaImpl (int n) throws RemoteException {
    numero= n;
  }
  public float obterSaldo  () throws RemoteException {
    return saldo;
  }
  public static void main (String[] args) {  ......
    Conta c= new ContaImpl (804);
    Naming.rebind ("Conta804", c);   // registra este objeto
  }
}
```

Java RMI: Exemplo de Aplicação

- Passo 3: Implementação do Cliente

```
 // arquivo ClienteImpl.java
import java.rmi.*;
import java.rmi.server.*;

public class ClienteImpl {
    public static void main (String[] args) {  ......
        Conta c= (Conta) Naming.lookup ( "//" + args[0]
                                          +"/Conta804");
        float s= c.obterSaldo(); // chamada remota
                                 // mesma sintaxe chamada local
        ......
     }
```

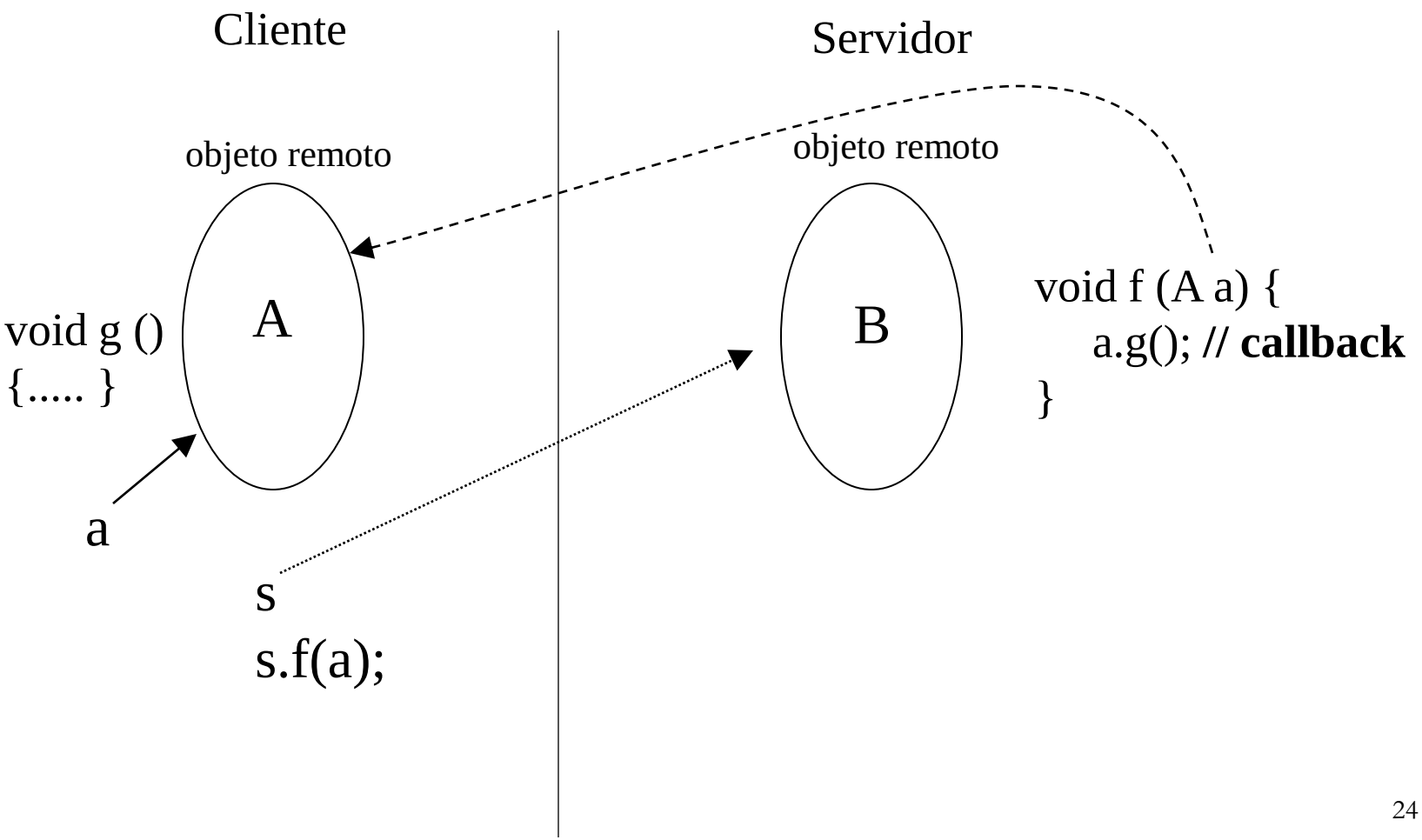
Java RMI: Exemplo de Aplicação

- Passo 4: Compilação
  - Compilação do servidor e do cliente (javac)
  - Geração de stubs:
    - rmic ContaImpl
  - rmic gera os seguintes arquivos:
    - ComtaImpl_Stub.class: stub do cliente
    - ContaImpl_Skel.class: skeleton (apenas RMI < JDK 1.4)
- Passo 5: Execução
  - Execução do Servidor:
    - rmiregistry // registry (permanece em execução)
    - java ContaImpl // servidor (outro processo)
  - Execução do Cliente:
    - java clienteImpl server01.dcc.ufmg.br

Java RMI: Passagem de Parâmetros

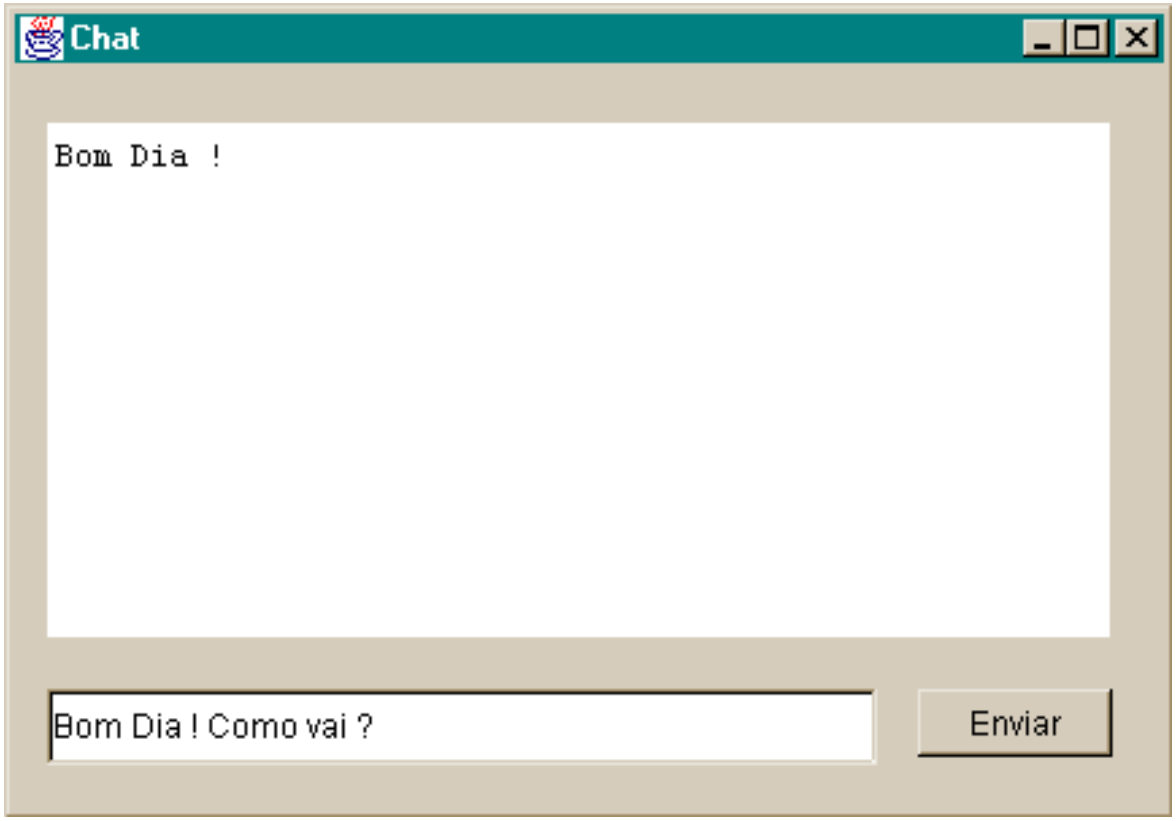
- Objetos serializáveis são passador por valor:
  - Classe do objeto deve implementar a interface Serializable
  - Passa-se uma cópia do objeto (atributos + métodos)
- Objetos remotos são passadas por referência:
  - Passa-se a referência (e não o objeto), a qual pode ser utilizada para realizar um *callback* no cliente
  - *Callback*: servidor chama método do cliente
  - Usado, por exemplo, quando o servidor deseja notificar o cliente sobre a ocorrência de algum evento

Exemplo de Callback

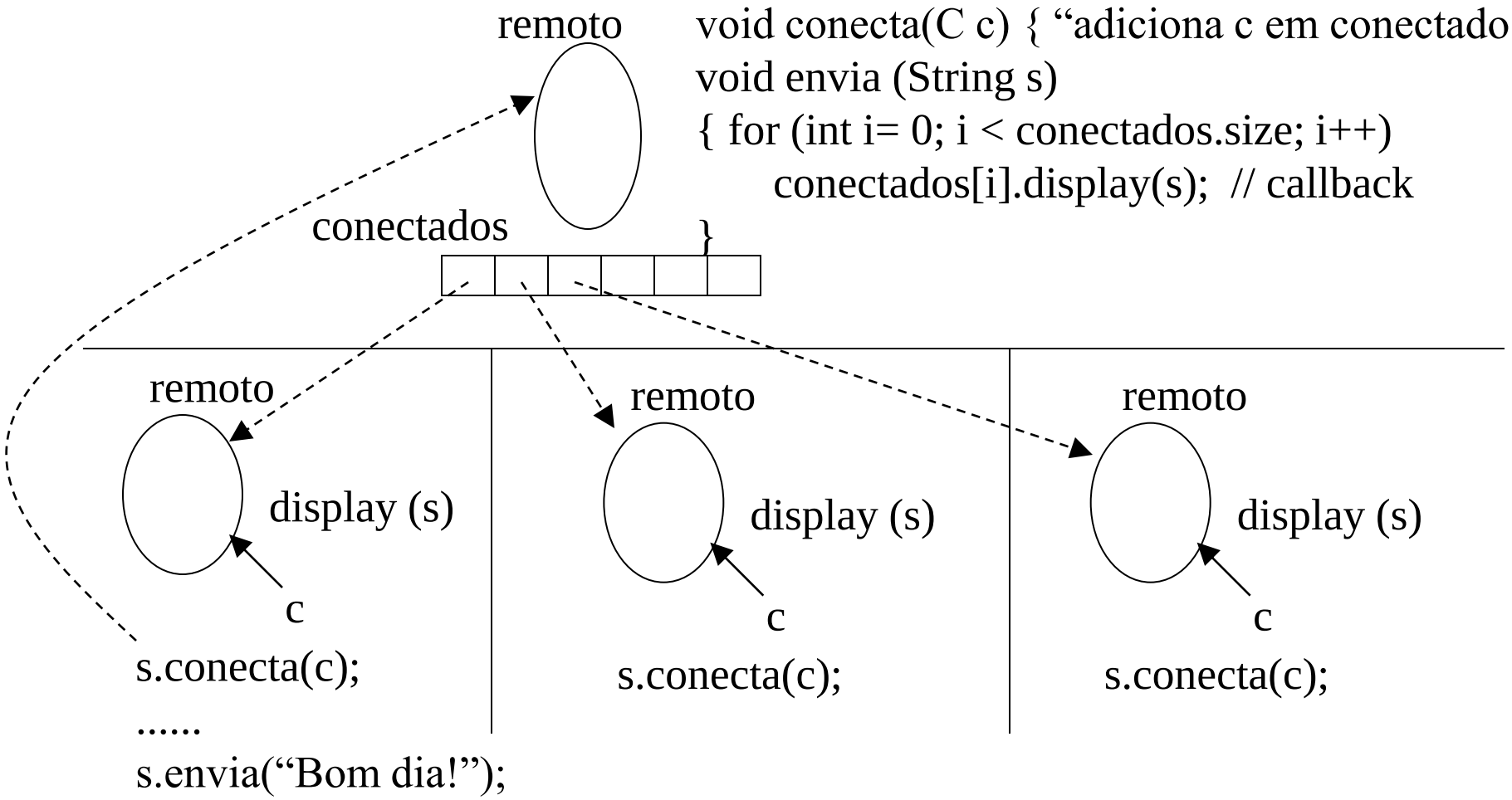


Java RMI - Chat

- Aplicação de *chat* usando Java RMI, com a seguinte interface:



Sistema de Chat em Java RMI



CORBA

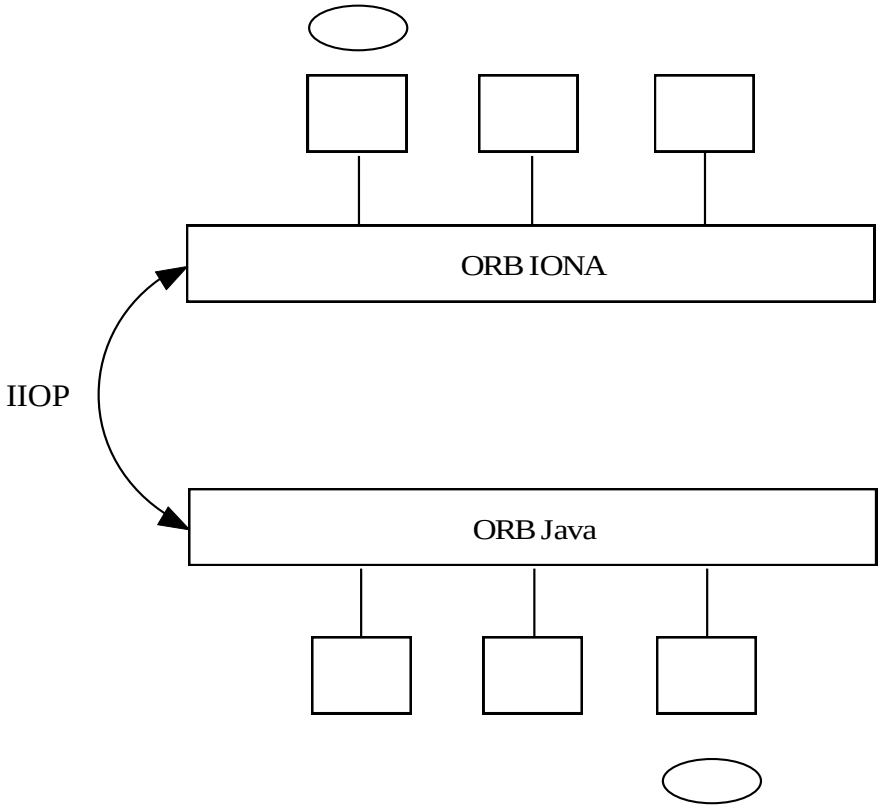
- CORBA: *Common Object Request Broker Architecture*
  - Padrão para desenvolvimento de aplicações distribuídas para sistemas heterogêneos usando orientação por objetos
- OMG: *Object Management Group* (http://www.omg.org)
  - Consórcio de empresas responsável pela proposição e manutenção do padrão CORBA
  - Criado em 1989, possui atualmente diversas empresas
- OMA: Object Management Architecture
  - Arquitetura para construção de aplicações distribuídas
  - Primeira especificação importante da OMG
- Objetivo da OMG: propor uma arquitetura e, em seguida, propor padrões para os componentes da mesma

CORBA

- Steve Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, IEEE Communications Magazine, vol. 14, no. 2, February 1997.
- Steve Vinoski: New Features for CORBA 3.0. Commun. ACM 41(10): 44-52 (1998)

IIOP: Internet Inter-ORB Protocol

- Protocolo, baseado em TCP/IP, para troca de mensagens entre ORBs de diferentes fabricantes
  - Objetivo: interoperabilidade



IDL: Interface Definition Language

- Linguagem para definir a interface de um objeto remoto
  - Especifica a assinatura das operações que serão implementadas por um objeto remoto
- Linguagem declarativa (sem código)
  - Tipos pré-definidos: long, short, float, double, char, boolean, enum
  - Tipos estruturados: struct, union, string, sequence
  - Tratamento de exceções
  - Modos de passagem de parâmetros: in, out e in out
  - Herança múltipla de interfaces
- Idéia: especificar interface em uma linguagem neutra

IDL: Interface Definition Language

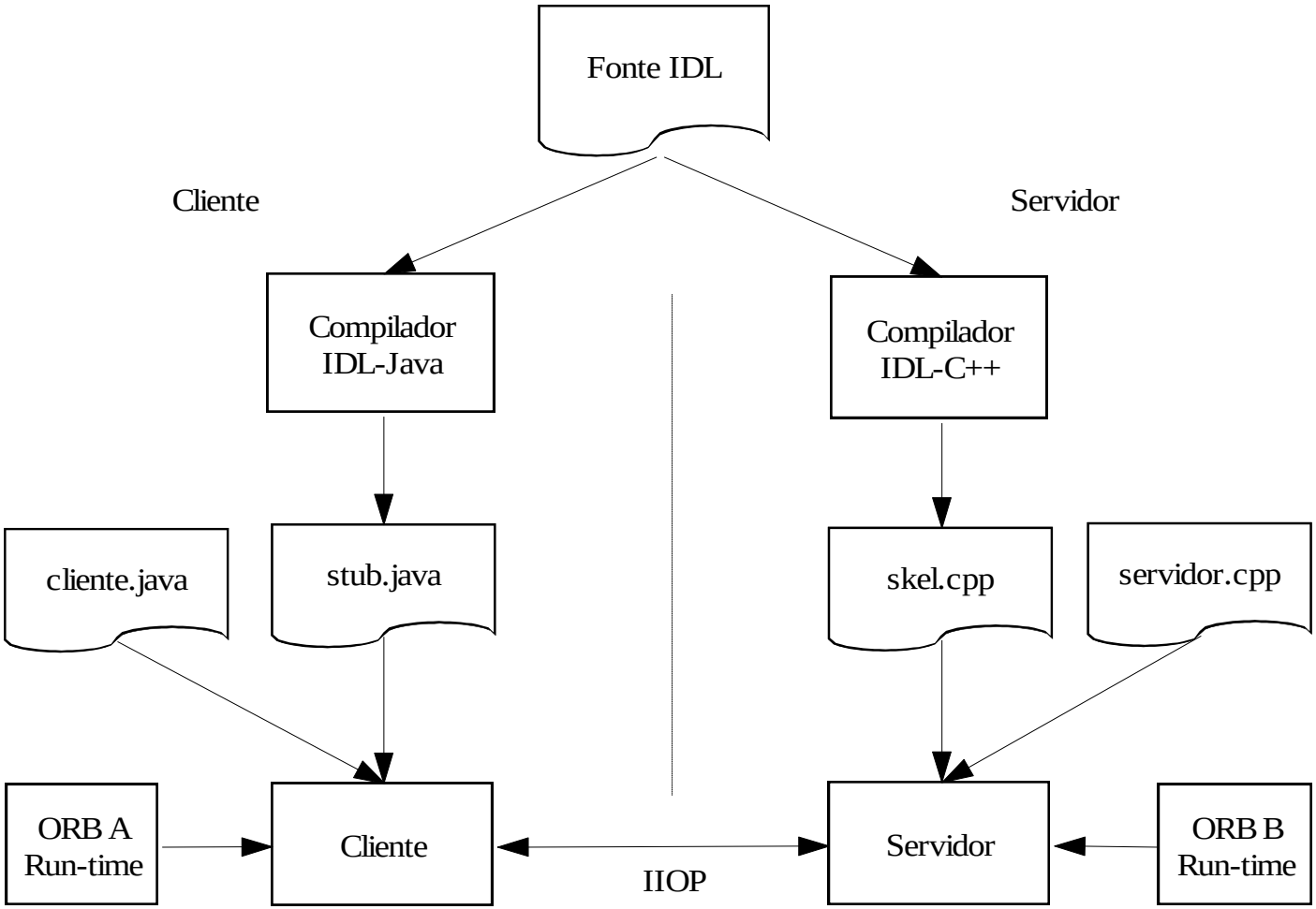
- Exemplo:

```
typedef sequence<string> extrato;

struct Data {
    short dia, mes,ano;
};

interface conta {
   float obterSaldo ();
   extrato obterExtrato (in Data inicio, in Data fim);
   .......
};
```

IDL: Interface Definition Language

- Compiladores de IDL geram automaticamente:
  - *stub* (clientes)
  - *skeleton* (servidor)
- Independência de linguagem:
  - Compiladores IDL implementam geração de código para C, C++, Smalltalk, Ada, COBOL, Java etc
  - CORBA especifica como deve ser o mapeamento (*binding*) de IDL para cada uma das linguagem acima

Desenvolvimento de Aplicações em CORBA: Visão Geral



CORBA: Vantagens e Desvantagens

- Vantagens:
  - Padrão (independência de LP, SO, fornecedor)
  - Middleware bastante completo (diversos serviços)
- Desvantagens:
  - Padrão “imposto” por um grande comitê: soluções extensas, duplicadas, sobrecarregadas, para atender a todos os gostos etc
    - Vide linguagens propostas por comitês (ADA, PL/I etc)
    - Ditado: “Um camelo é um cavalo projetado por um comitê”
    - “The only way to reach agreement during the submission process for a CORBA specification is to take the grand union of the feature sets of all the pre-existing proprietary implementations” (ICE home page)
  - Solução fechada, monolítica, inflexível, com poucos pontos de configuração, customização, personalização etc
    - Caixa-preta, “one size fits all”

Transparência

- Idéia: distribuição deve ser “transparente” aos programadores
- Principais tipos de transparência (segundo ISO RM-ODP):
  - Transparência de acesso: a maneira por meio da qual um componente A acessa serviços de um componente B independe do fato de B ser local ou remoto
  - Transparência de localização: para que um componente A acesse serviços de um componente B, ele não precisa conhecer o endereço físico de B (mas apenas seu nome lógico)
  - Transparência de migração: componente B pode migrar de uma máquina para outra sem impactar componente A
  - Transparência de replicação: componente A pode acessar o componente B ou uma de suas réplicas
- Transparência é fundamental para aumentar o nível de abstração, bem como para prover escalabilidade (no caso de migração e replicação)

Transparência em CORBA e RMI

- Transparência de acesso: métodos remotos são chamados com a mesma sintaxe de métodos locais
  - Java RMI: Sim
  - CORBA: Sim

- Transparência de localização: para efetuar uma chamada remota, clientes não precisam conhecer o nome da máquina onde localiza-se o objeto servidor
  - Java RMI: não, pois método lookup tem como parâmetro o nome da máquina do objeto servidor
  - CORBA: sim, pois clientes precisam conhecer a localização de uma única máquina (o servidor de nomes da rede)

Críticas a Transparência

- Jim Waldo, Geoff Wyant, Ann Wollrath, Samuel C. Kendall: A Note on Distributed Computing. Mobile Object Systems 1996: 49-64
- “There is an overall vision of distributed object-oriented computing in which, from the programmer’s point of view, there is no essential distinction between objects that share an address space and objects that are on two machines with different architectures located on different continents.”
- “It is the thesis of this note that this unified view of objects is mistaken. There are fundamental differences between tthe interactions of distributed objects and the interactions of non-distributed objects. Further, work in distributed object-oriented systems that is based on a model that ignores or denies these differences is doomed to failure, and could easily lead to an industry-wide rejection of the notion of distributed object-based systems.”

Computação Local x Remota

- Latência:
  - “Ignoring the difference between the performance of local and remote invocations can lead to designs whose implementations are virtually assured of having performance problems because the design requires a large amount of communication between components that are in different address spaces and on different machines.”
- Acesso a Memória:
  - “A more fundamental difference between local and remote computing concerns the access to memory in the two cases—specifically in the use of pointers. Simply put, pointers in a local address space are not valid in another (remote) address space. The system can paper over this difference ... Two choices exist: either all memory access must be controlled by the underlying system, or the programmer must be aware of the different types of access—local and remote.”

Computação Local x Remota

- Falhas:
  - “Being robust in the face of partial failure requires some expression at the interface level. Merely improving the implementation of one component is not sufficient. The interfaces that connect the components must be able to state whenever possible the cause of failure”
- Concorrência:
  - “Distributed objects by their nature must handle concurrent method invocations”

Reflexão Computacional

Marco Túlio de Oliveira Valente

Reflexão Computacional

- Capacidade de um sistema se auto-inspecionar e auto-manipular
- Um sistema reflexivo possui dois níveis:
  - Nível básico: contém objetos do domínio da aplicação
  - Meta-nível: contém objetos que permitem acessar propriedades e manipular objetos do nível básico
- Objeto do meta-nível são chamados de meta-objetos
- Protocolo de meta-objetos: protocolo de interação com meta-objetos
- Reificação: ação de transformar em meta-objetos informações do nível base
- Reflexão envolve duas atividades:
  - Inspeção: quando meta-nível inspeciona informações do nível básico
  - Interseção: quando meta-nível interfere no comportamento ou na estrutura do nível básico

Reflexão Computacional

- A atividade de intercessão pode ser:
  - Estrutural
  - Comportamental
- Reflexão estrutural: permite alterar estruturas de dados do programa (criar classes, criar objetos, adicionar métodos em classes, adicionar campos em classes etc)
- Reflexão comportamental: permite interferir na execução do programa, interceptando alguns eventos (chamadas de métodos, criação de objetos etc)
- Reflexão desde o início foi usada para implementar diversos requisitos não funcionais típicos de sistemas distribuídos:
  - Transações, segurança, tolerância a falhas, balanceamento de carga, logging, persistência etc
- Vantagem: separação de interesses

Reflexão em Java

- Tudo começa com um meta-objeto do tipo Class
- Exemplo: Class c= Class.forName(“java.util.Stack”);
- Pode-se usar métodos do tipo Class para inspeção de informações:
  - Obter a super-classe do tipo base
  - Obter interfaces que são implementadas pelo tipo base
  - Obter informações sobre os construtores do tipo base
  - Obter informações sobre os métodos do tipo base (modificadores, tipo de retorno, tipo dos parâmetros)
- Pode-se ainda interceder no nível básico do programa para:
  - Criar objetos de tipos cujos nomes somente serão conhecidos em tempo de execução
  - Obter/definir valores de atributos cujos nomes somente serão conhecidos em tempo de execução
  - Invocar métodos cujos nomes somente serão conhecidos em tempo de execução

Reflexão em Java

- Programa que lista informações sobre uma interface cujo nome é informado pelo usuário:

```
Console.readString(s);  // le nome da interface do teclado
Class c= Class.forName(s);
Method[] theMethods= c.getMethods();
for (int i = 0; i < theMethods.length; i++) {
  String methodString= theMethods[i].getName();
  System.out.println("Name: " + methodString);
  String returnString= theMethods[i].getReturnType().getName();
  System.out.println(" Return Type: " + returnString);
  Class[] parameterTypes= theMethods[i].getParameterTypes();
  System.out.print(" Parameter Types:");
  for (int k= 0; k < parameterTypes.length; k ++) {
    String parameterString= parameterTypes[k].getName();
    System.out.print(" " + parameterString);
  }
  System.out.println();
}
```

Reflexão em Java

- Permite inspeção do nível base
- Permite forma limitada de reflexão estrutural:
  - Pode-se criar objetos e chamar métodos de forma reflexiva
  - Não se pode adicionar atributos/métodos em classes, mudar o nome de um método, mudar modificadores etc
- Permite forma limitada de reflexão comportamental:
  - Interceptação de chamadas de métodos via classes proxy dinâmicas
- Soluções mais poderosas:
  - Sistemas como Javassist (reflexão estrutural) e Kava (reflexão comportamental).
  - Programação orientada por aspectos
- Em linhas gerais,
  - AOP= reflexão sem reificação

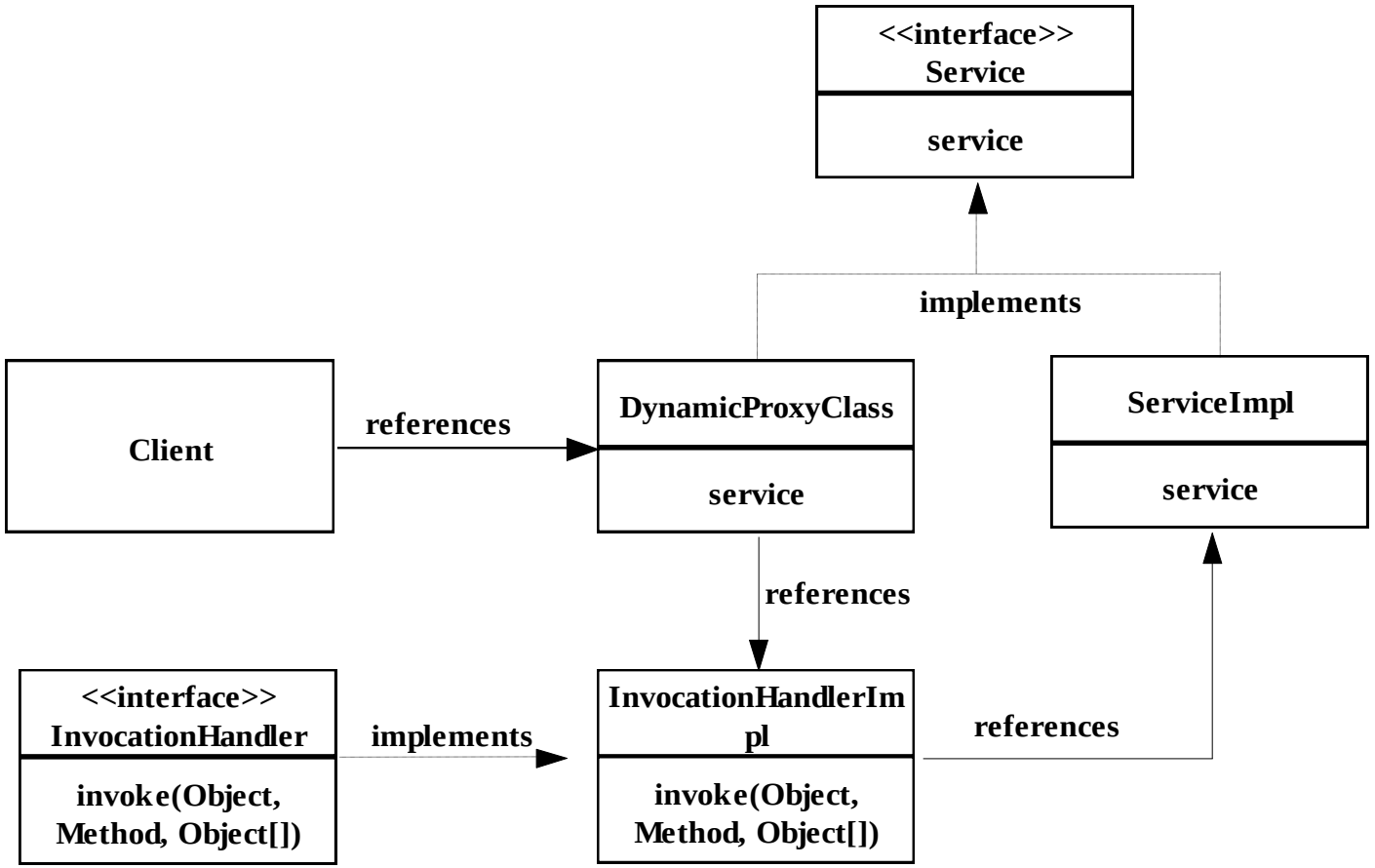
Classes Proxy Dinâmicas

- Recurso de reflexão de Java para criação dinâmica de proxies
- Proxy: padrão de projeto comum em aplicações distribuídas
- Proxy: objeto que implementa a mesma interface de um objeto base
  - Age como um intermediário entre objeto base e seus clientes
  - Normalmente, possuem uma referência para objeto base e interceptam todas as chamadas destinadas ao mesmo
- Exemplo: stubs são exemplos de proxy
- Em Java, classes proxy dinâmicas têm duas propriedades:
  - Seu código é criado em tempo de execução
  - Implementam interfaces definidas em tempo de criação
- Instâncias de classes proxy dinâmicas têm ainda um objeto manipulador de invocação (invocation handler), o qual é definido pelo cliente que solicitou a criação das mesmas.

Classes Proxy Dinâmicas

- Toda a invocação de método sobre uma instância de uma classe proxy dinâmica é enviada automaticamente para o método invoke do manipulador desta instância, passando os seguintes parâmetros:
  - a instância da classe proxy sobre a qual a invocação foi realizada;
  - um objeto da classe java.lang.reflect.Method, o qual reifica o método que está sendo chamado;
  - um vetor do tipo Object que contém os argumentos deste método.

Classes Proxy Dinâmicas



Exemplo de Classe Proxy

```
class DebugHandler implements java.lang.reflect.InvocationHandler {
    private Object base;
    private DebugHandler(Object base) {
        this.base = base;
    }
    public Object invoke(Object proxy, Method m, Object[] args)
       throws Throwable {
       Object result;
       try {
         System.out.println("before method " + m.getName());
         result = m.invoke(base, args);   // forward to base object
       } catch (...) {
         ...
       } finally {
         System.out.println("after method " + m.getName());
       }
       return result;
    }
} // class DebugHandler
```

Exemplo de Classe Proxy

- Exemplo de Uso:

```
public interface Foo {
    Object bar(Object obj) throws BazException;
}
public class FooImpl implements Foo {
    Object bar(Object obj) throws BazException { ... }
}
.....
Foo base= new FooImpl();
DebugHandler handler= new DebugHandler(base);
Class c= base.getClass();
Foo proxy= Proxy.newProxyInstance(c.getClassLoader(),
   c.getInterfaces(), handler);
.....
proxy.bar(....);
```

Outros Recursos de Sistemas de Middleware

Marco Túlio de Oliveira Valente

Sincronização de Requisições

- Normalmente, chamadas remotas são síncronas
- Cliente fica bloqueado, esperando resultado da chamada remota

- No entanto, dependendo da duração de uma chamada remota, pode não ser interessante que o cliente fique bloqueado

- Chamadas oneway:
  - Controle retorna ao cliente quando middleware recebe requisição
  - Despacho/execução da chamada e execução do cliente ocorrem de forma assíncrona
  - Somente podem ser aplicadas a métodos com retorno void
  - Não ativam uma exceção em caso de falha na execução da chamada remota; recomendadas quando o cliente não necessita do resultado da chamada remota

Chamadas Assíncronas

- Primeira solução: baseada em polling (também chamada de deferred synchronous)
- Quando cliente realiza uma chamada assíncrona, a mesma retorna um objeto de um tipo Future (ou Promises ou outro nome similar)

```
Future x= s.f(....);
```

- Cliente então prossegue sua execução
- Middleware despacha a chamada de forma assíncrona (em uma thread independente). Quando o resultado chega, o mesmo é associado ao objeto do tipo Future.
- Em um determinado momento, o cliente pode precisar do resultado da chamada remota.
- Para isso, basta realizar um polling no Future:

```
Object o= x.getResult();
```

Chamadas Assíncronas

- Segunda solução: usando callback
- Chamadas remotas possuem sempre um primeiro parâmetro que é um objeto sobre o qual será realizado o *callback*
  - Quando resultado estiver disponível, chama-se método deste objeto
- Interface provida pelo sistema:

```
interface Listener { void setResult (Object x); }
```

- Interface criada pelo programador e implementada pelo obj. remoto:

```
interface A { String f (int x); }
```

- Interface gerada pelo sistema

```
interface A_Async extends A {void async_f(Listener x, int y);}
```

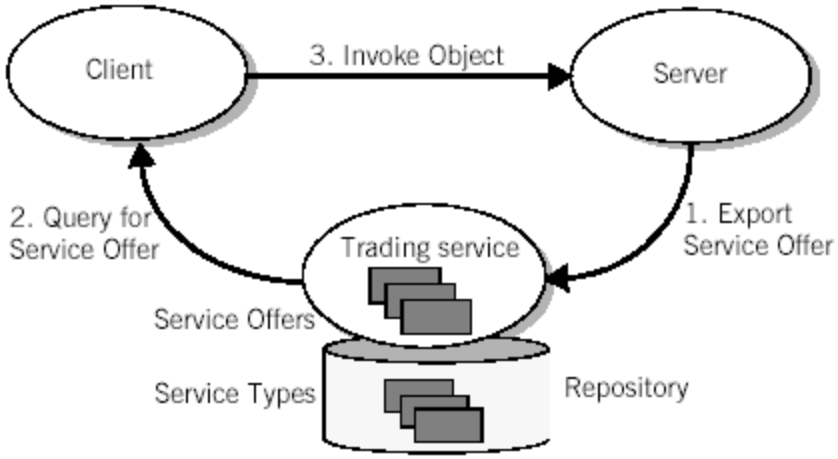
- Chamada Assíncrona:

```
Async_A a= Naming.lookup(....);
Listener x= new MyListener();
a.async_f(x, 10);
```

Trader: Serviço de Negociação

- Serviço de nome mais avançado existente em CORBA
- Serviços de nomes tradicionais funcionam como as “páginas brancas” de uma lista telefônica
- Requerem que clientes conheçam previamente os nomes dos objetos remotos que acessam (ou pelo menos do primeiro objeto remoto acessado)
- Este fato pode ser uma limitação importante, principalmente em ambientes mais dinâmicos
- Trader: objeto remoto é pesquisado com base em suas propriedades, características, qualidade de serviço etc
- Semelhante às “páginas amarelas” de uma lista telefônica
- Requer o uso de uma “Trader Constraint Language” para consultas
- Exemplo: (page_min > 5) and ((page_type=A4) or color)

CORBA Trader



- Objetivo: maior independência entre clientes e servidores
- Clientes não precisam conhecer nomes de objetos remotos
- No entanto, clientes ainda precisam conhecer:
  - o nome da interface dos objetos remotos
  - os nomes das propriedades dos objetos remotos
- Reflexão pode ajudar a obter estas informações, mas código reflexivo é extenso, sujeito a erros etc

Coleta de Lixo Distribuída

- Suponha as seguintes interfaces remotas:

```
interface Srv extends Remote { Msg getMsg(); }
interface Msg extends Remote { ... }
```

- Suponha um servidor com um objeto remoto que implementa Srv:

```
Msg getMsg() { return new MsgImpl(); }
```

- Suponha ainda uma classe remota MsgImpl, que implementa Msg
- Código do cliente:

```
void foo() {
  Srv s= "referência para objeto remoto que implementa Srv"
   Msg msg;
   for (int i= 0; i<100; i++)
      msg= s.getMsd();  // cria 100 objetos remotos no servidor
}
```

- Quando estes objetos são destruídos?
- Resposta: Quando o cliente deixa de utilizá-los.
- Mas como o servidor fica sabendo deste fato?

Coleta de Lixo Distribuída

- Em sistemas distribuídos, o coletor de lixo necessita remover aqueles objetos remotos que:
  - Não são referenciados localmente (como usual)
  - E que também não são referenciados por clientes localizados em um espaço de endereçamento remoto

- Java RMI usa um algoritmo de coleta de lixo distribuída baseado na técnica de contagem de referências
- Técnica usada pela primeira vez em Modula-3 (Network Objects)
- Idéia básica:
  - Objetos remotos possuem dois métodos: dirty() e clean()
  - Objetos remotos possuem um contador de referências externas para os mesmos

Coleta de Lixo Distribuída

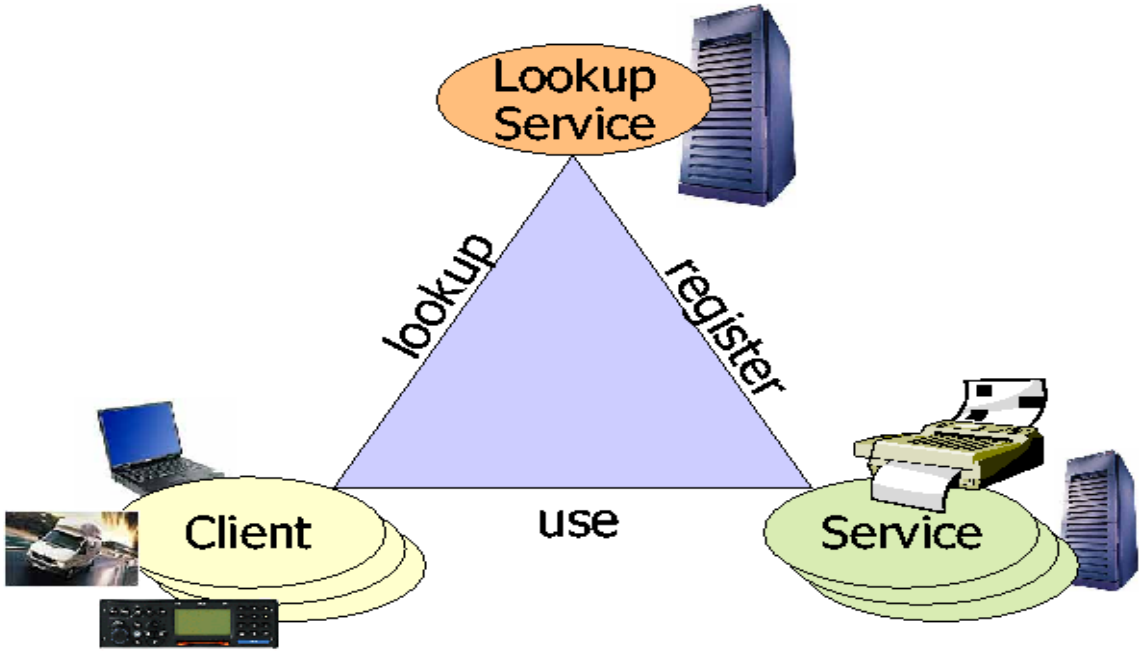
- Cliente:
  - Ao instanciar um stub de um objeto remoto: chama obj.dirty()
  - Ao destruir um stub de um objeto remoto: chama obj.clean()
- Servidor:
  - obj.dirty(): incrementa contador de referências externas
  - obj.clean(): decrementa contador de referências externas
  - Núcleo de RMI: deixa de referenciar obj quando contador == 0
- Problemas: falhas no cliente ou na rede podem impedir o envio de uma mensagem obj.clean()
- Solução: lease
  - Cliente reenvia object.dirty() de tempos em tempos
  - Servidor armazena não só um contador, mas também a identificação de cada cliente que possui uma referência para ele
  - Servidor decrementa contador quando lease não é renovado

Jini

- Jim Waldo: The Jini Architecture for Network-Centric Computing. CACM 42(7): 76-82 (1999)
- Sistema para construção de aplicações distribuídas em ambientes de computação móvel, ubíqua, pervasiva etc
- Redes sujeitas a constantes reconfigurações, redes espontâneas
- Utiliza Java RMI como infra-estrutura básica de comunicação
- Possui um serviço de nomes próprio, chamado lookup service
- Utiliza um protocolo mais flexível para registro, pesquisa e remoção de informações do serviço de nomes
- Consultas ao servidor de nomes são realizadas usando interfaces e/ou propriedades do serviço que está sendo solicitado
- Servidores e clientes não precisam conhecer previamente o endereço do servidor de nomes da rede à qual estão conectados

Jini

- Exemplo: conexão de um novo serviço de impressão em uma rede
- Ao se conectar, servidor de impressão difunde uma mensagem procurando pelo servidor de nomes
- Clientes procurando por um serviço de impressão também localizam o serviço de nomes difundindo uma mensagem
- Comunicação entre cliente e servidor de impressão é via RMI



Jini

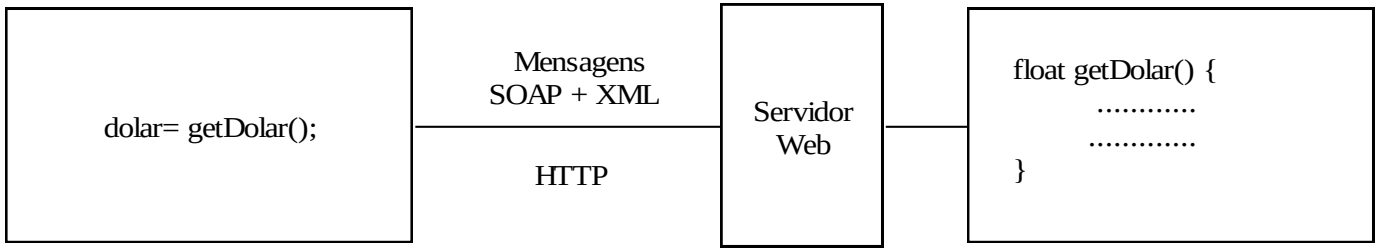
- Jini possibilita que novos serviços possam ser acrescentados transparentemente em um sistema distribuído, sem necessidade de suporte por parte dos administradores do mesmo
- Permite que serviços possam ser também automaticamente desconectados de uma rede
- Para isso, associa-se a toda referência de rede registrada no serviço de nomes um lease
- Isto é, um período de tempo durante o qual este registro permanece válido.
- Servidores devem renovar o lease de um serviço, caso pretendam continuar disponibilizando o mesmo
- Por outro lado, caso um servidor se desconecte da rede, mesmo que de forma inesperada, seus serviços serão automaticamente removidos do serviço de nomes assim que expire seu lease
- Outros recursos de Jini: notificação de eventos e transações

Serviços Web

Marco Túlio de Oliveira Valente

Serviços Web

- Extensão e adaptação dos conceito de chamada remota de métodos para a *Web*
- Informações estruturadas, com interfaces bem definidas
- Principais padrões: HTTP, XML, SOAP, WSDL e UDDI
- Exemplo: serviço *Web* para obter cotação do dólar

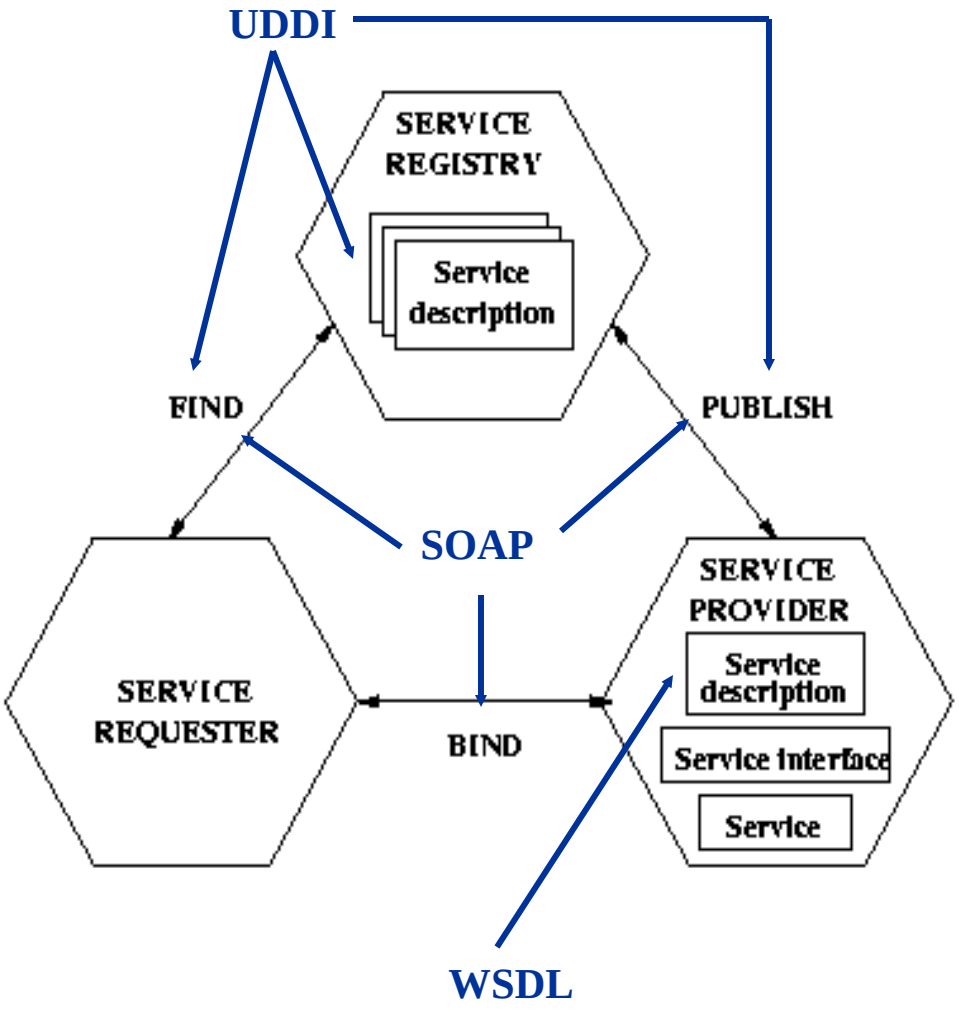


- Principal aplicação: integração de aplicações

Serviços Web

- Serviços Web são aplicações distribuídas que se comunicam por meio de mensagens XML, formatadas e encapsuladas segundo o protocolo SOAP e transportadas via HTTP.
- Serviços Web têm suas interfaces definidas em uma linguagem denominada WSDL e são registrados em servidores UDDI
- Para quem conhece CORBA:
  - HTTP e TCP fazem o papel do TCP em CORBA
  - XML e SOAP fazem o papel do IIOP em CORBA
  - WSDL faz o papel de IDL em CORBA
  - UDDI faz o papel de um serviço de nomes em CORBA
- Simplificando, serviços Web são semelhantes a CORBA, porém usando protocolos da Web

Arquiteturas Orientadas por Serviços



Fonte: Web Services: Concepts, Architecture and Applications. Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. Springer Verlag, 2004

SOAP

- SOAP: Simple Object Access Protocol
- Protocolo W3C para troca de informações na Internet
- Utiliza XML para codificação de mensagens e HTTP para transporte de mensagens de um nodo a outro
- Permite adicionar serviços a um servidor Web e então acessar estes serviços a partir de programas comuns
  - Serviços: objetos, com métodos chamados remotamente
- SOAP especifica como representar em XML todas informações necessárias para se executar uma chamada remota
  - Define o formato da mensagem de ida (com os argumentos)
  - Define o formato da mensagem de volta (com o resultado)
- Independente de linguagem de programação

SOAP: Mensagens

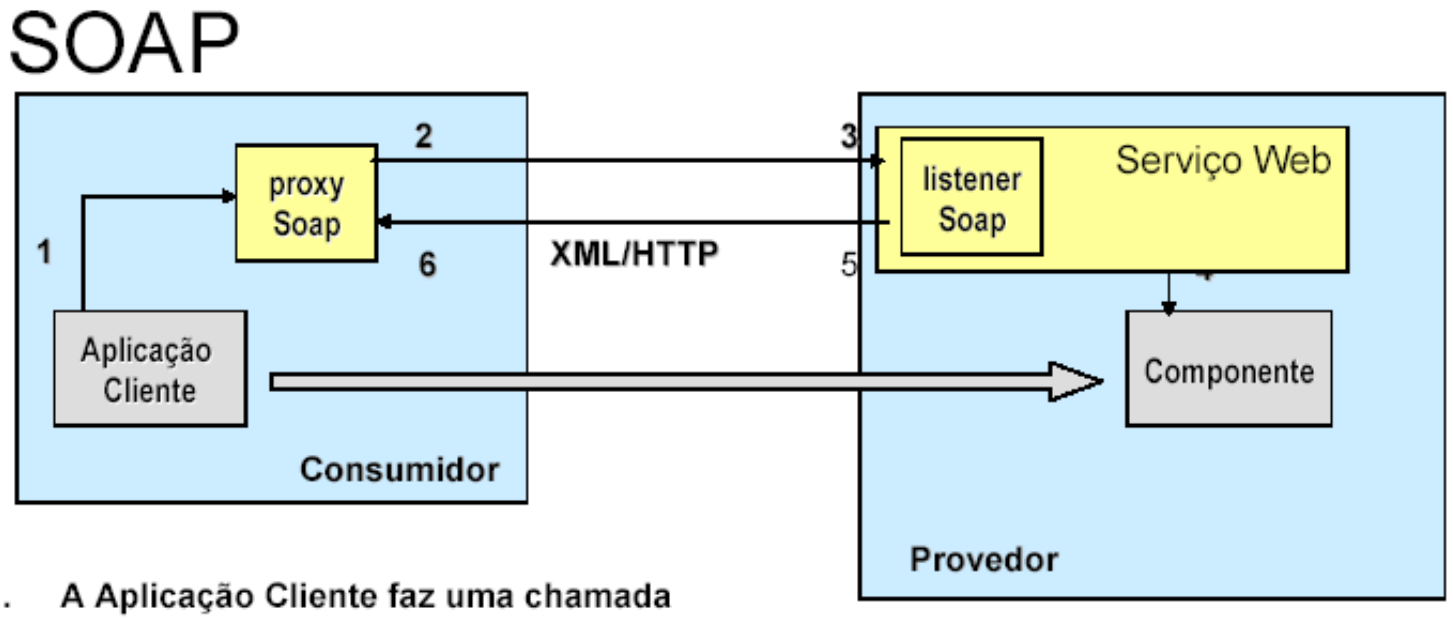
- Suponha a seguinte implementação de um serviço Web:

```
String sayHello(String name) {
  return "Hello, " + name;
}
```

- Quando um cliente invocar o método acima, passando Bob como parâmetro, será gerado e transmitido o seguinte envelope SOAP:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Header> </SOAP-ENV:Header>
  <SOAP-ENV:Body>
     <ns1:sayHello xmlns:ns1="Hello"
      SOAP-NV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <name type="xsd:string">Bob</name>
     </ns1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP: Arquitetura



1. A Aplicação Cliente faz uma chamada
2. O Proxy intercepta a chamada, constrói e transmite a mensagem de requisição XML
3. O listener Soap recebe, analisa gramaticalmente e valida a requisição
4. O Listener chama a mensagem do componente
5. O Listener pega o resultado da chamada, constrói e transmite a msg. XML de resposta
6. O Proxy recebe, analisa gramaticalmente a resposta, e retorna o resultado ao cliente

Este processo é transparente para o cliente e o componente

Fonte: Paulo Pires e Marta Matoso. Tecnologia de Serviços Web. Mini-curso SBES 2005.

WSDL

- WSDL: Web Service Description Language
- Linguagem para descrever a localização e a interface de um serviço Web, isto é:
  - Nome e URL do serviço (“binding information”)
  - Assinatura dos métodos disponíveis para chamada remota (“abstract interface”)
- Semelhante a IDL em CORBA, porém com uma sintaxe baseada em XML
- Existem ferramentas que geram especificações WSDL a partir, por exemplo, de interfaces Java

Exemplo de Arquivo WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:hello"
   ......
   <wsdl:message name="sayHelloResponse">
      <wsdl:part name="sayHelloReturn" type="xsd:string"/>
   </wsdl:message>
   <wsdl:message name="sayHelloRequest">
      <wsdl:part name="in0" type="xsd:string"/>
   </wsdl:message>
   <wsdl:portType name="HelloServer">
      <wsdl:operation name="sayHello" parameterOrder="in0">
         <wsdl:input message="intf:sayHelloRequest" name="sayHelloRequest"/>
         <wsdl:output message="intf:sayHelloResponse" name="sayHelloResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   ..........
   </wsdl:binding>
   <wsdl:service name="HelloServerService">
      <wsdl:port binding="intf:rpcrouterSoapBinding" name="rpcrouter">
         <wsdlsoap:address location="http://localhost8080/soap/servlet/rpcrouter"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

UDDI

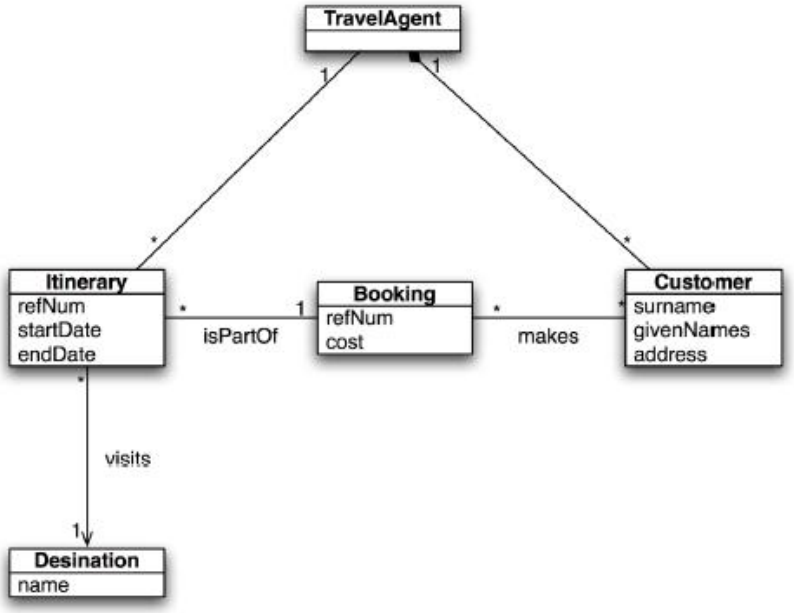
- UDDI: Universal Description, Discovery and Integration
- Especificação em WSDL: permite a um cliente obter todas as informações para interagir com um serviço Web
- Mas com obter obter informações sobre um serviço Web ?
  - Resposta clássica: junto a um servidor de nomes
- UDDI: conjunto de especificações para:
  - Registrar um serviço Web
  - Pesquisar por serviço Web, via interface Web ou SOAP
- Pesquisas por um serviço podem ser realizadas em:
  - Páginas brancas: contém nome e informações de contato
  - Páginas amarelas: serviços organizados em “categorias”
- Permite a criação de uma hierarquia de servidores UDDI

SOA vs DOA

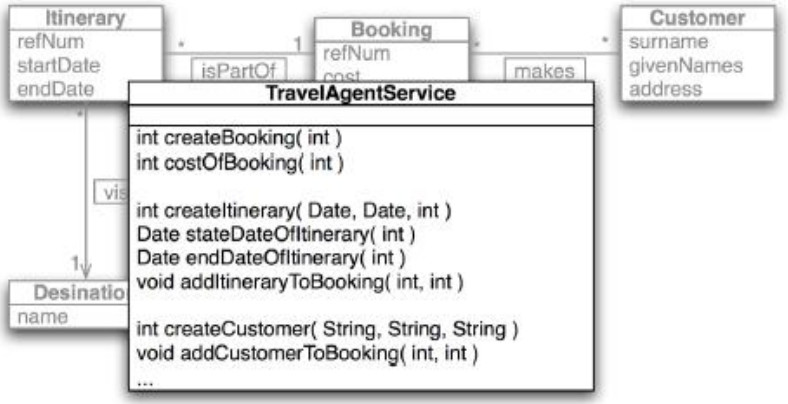
- *Comparing Service-Oriented and Distributed Object Architectures*. Seán Baker and Simon Dobson. DOA 2005
- DOA: Distributed Object Architectures (CORBA, RMI etc)
  - Bastante estável
  - Integração de sistemas de informação corporativos
- SOA: Service Oriented Architectures (Web Services)
  - Extensão de DOA para além das fronteiras de uma corporação
  - Integração de “sistemas fracamente acoplados”
  - Tecnologicamente mais neutro do que CORBA
- Questões que sempre surgem?
  - SOA é uma reinvenção de DOA?
  - SOA é uma readaptação de DOA para HTTP/XML?
- Principais semelhanças:
  - Serviços = objetos, WSDL = IDL, SOAP = IIOP

SOA vs DOA: Diferenças

- Granularidade:
  - Interfaces DOA possuem um menor nível de granularidade
- Exemplo: agência de viagem



(a) DOA



(b) SOA

SOA vs DOA: Diferenças

- Em DOA:
  - Interfaces remotas usam tipos Itinerário, Reserva, Cliente etc
  - Exemplos de métodos:
    - `void createCustomer(Customer)` `(interface Cliente)`
    - `void addItinerary(Itinerary)` `(interface Booking)`
    - `float getCost()` `(interface Booking)`
- Em SOA:
  - Métodos são agrupados em uma única interface (TravelAgencyService)
  - Exemplos de métodos:
    - `int createCustomer(String, String, String)`
    - `void addItineraryToBooking(int, int)`
    - `float getCost(int)`
- Em SOA, tipos Itinerário, Reserva, Cliente etc podem existir internamente, mas não são expostos nas interfaces remotas
- Modelo de objetos interno de SOA pode ser idêntico ao de DOA

SOA vs DOA: Diferenças

- SOA: interfaces seguem uma perspectiva de negócio
  - Um agente de turismo é uma entidade que merece uma interface
  - O mesmo não ocorre com Itinerários, Reservas etc

- DOA: interfaces seguem uma perspectiva de sistemas
  - Logo, Itinerários, Reservas, Clientes etc são entidades relevantes do domínio que está sendo modelado
  - Merecem ser “expostos” nas interfaces como objetos

SOA vs DOA: Diferenças

- “There are significant advantages to the SOA view. It reduces the “surface area” of interfaces, and hence the learning curve for client programmers. It also tends to produce interfaces that perform operations in fewer interactions.”

- “The DOA view may lead to interactions that are too “chatty”, in the sense of requiring a number of interactions to accomplish the same effect. In many systems this will increase the number of network operations to accomplish a single business level task, reducing system throughput.”

BPEL

- Business Process Execution Language (BPEL)
- Desenvolvida pela IBM, Microsoft, BEA, SAP etc
- Linguagem baseada em XML usada para descrever (ou orquestrar) o “fluxo de execução de um processo de um negócio”
- Exemplo: habilitação de um terminal telefônico
  - Verificar se usuário tem o “nome limpo” (sistema de crédito)
  - Verificar se há disponibilidade física para instalação do ramal telefônico (sistema de engenharia)
  - Cadastrar usuário (sistema de cobrança)
  - Cadastrar terminal (sistema de engenharia)
- BPEL poderia ser usada para modelar o fluxo de um processo de negócio como o descrito acima
- Sistemas devem possuir interfaces de comunicação baseadas em serviços Web

BPEL: Exemplo 1

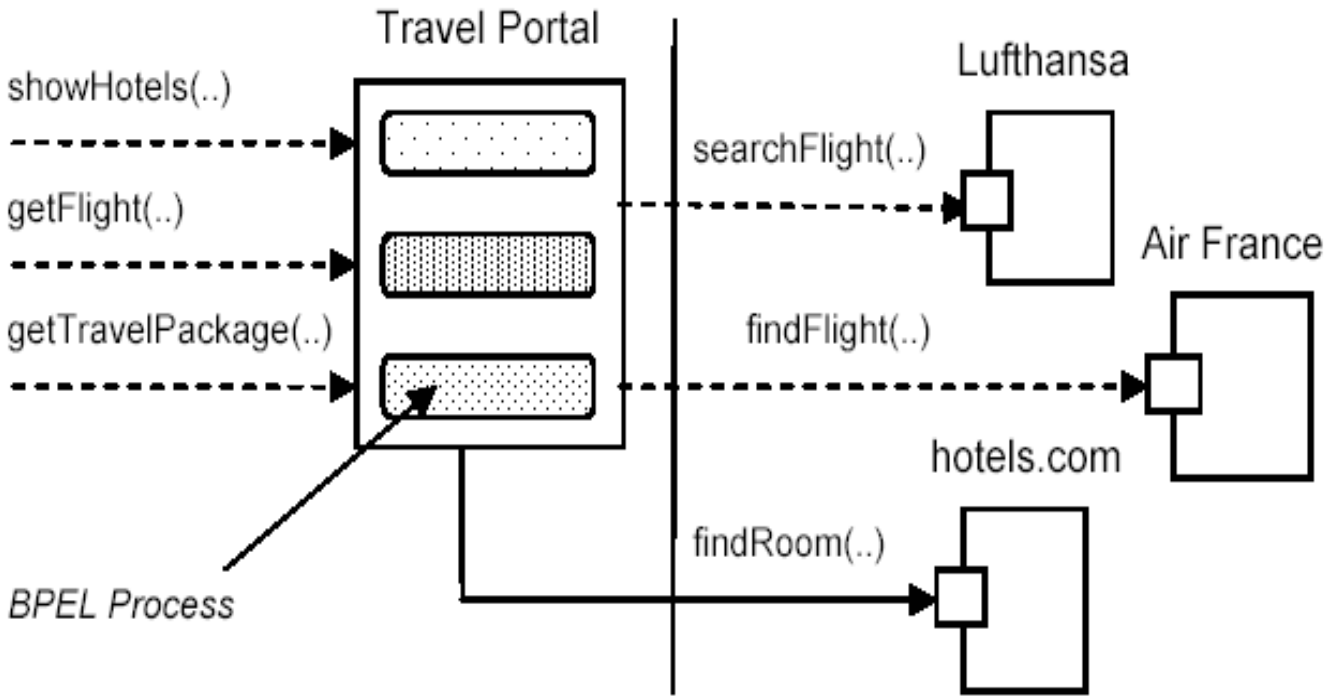


Fig. 2. A Travel portal as a composite web service

BPEL: Exemplo 2

- Processo que:
  - Dada uma string, realiza uma consulta ao Google
  - Para cada item da resposta, retorna as páginas armazenadas no cache do Google (para isso, volta a consultar o Google)

```
<variables>
  <variable name="search" messageType="goo:doGoogleSearch"/>
  <variable name="searchResponse"
   messageType="goo:doGoogleSearchResponse"/>
  <variable name="cache" messageType="goo:doGetCachedPage"/>
  <variable name="cacheResponse"
   messageType="goo:doGetCachedPageResponse"/>
  <variable name="i" type="xsi:int"/>
  <variable name="query" type="xsi:string"/>
  <variable name="itemNb" type="xsi:int"/>
</variables>
```

BPEL: Exemplo 2

```
<sequence>
  <assign>
     <copy>
       <from>
         <search>
           <key>0123456789</key>
           <q>ucl computer science</q>
           <start>1</start>
           <maxResults>5</maxResults>
           <filter>true</filter>
           <restrict/>
           <safeSearch>true</safeSearch>
           <lr/>
           <ie>latin1</ie>
           <oe>latin1</oe>
         </search>
       </from>
       <to variable="search"/>
     </copy>
  </assign>
```

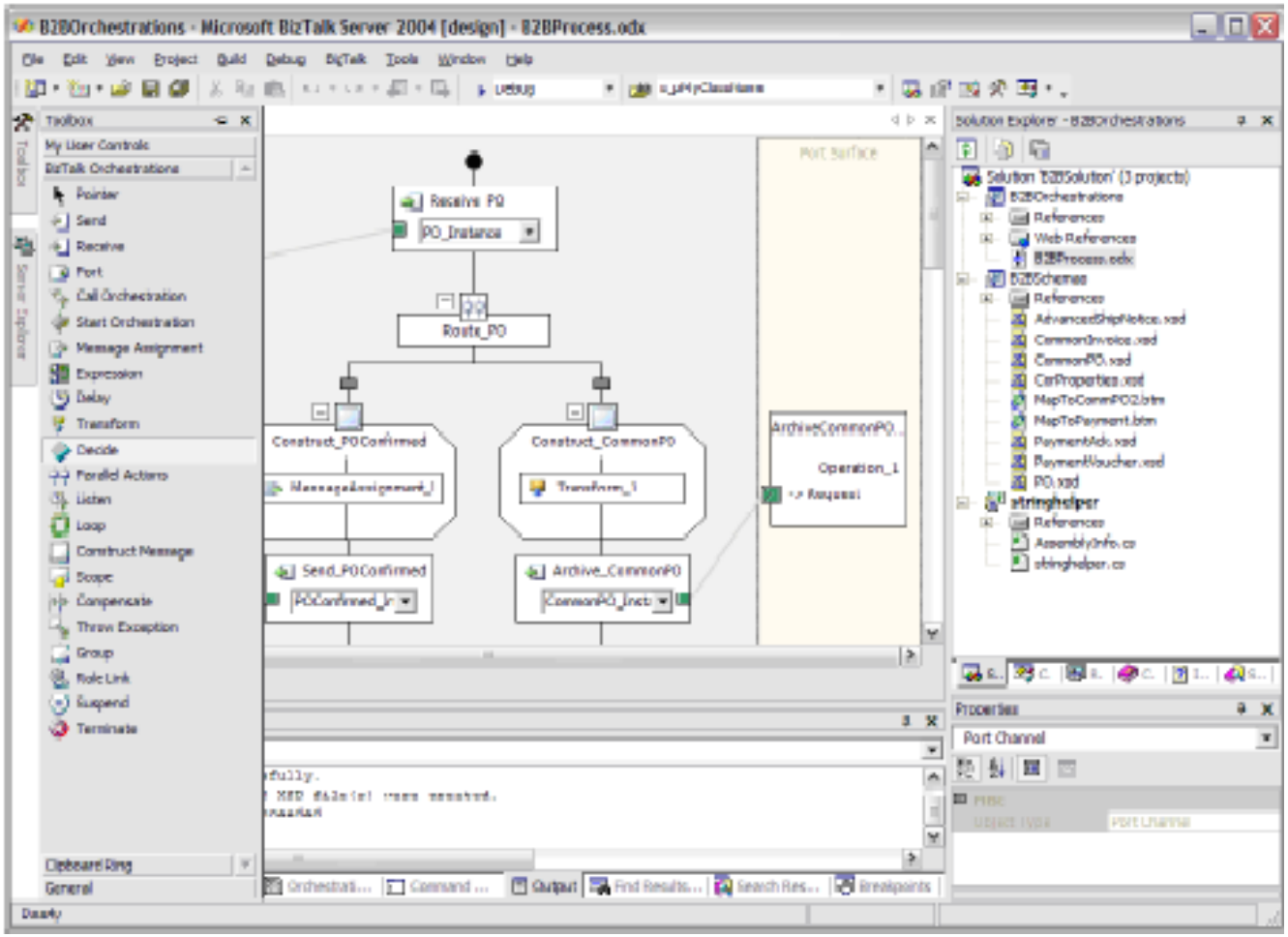
BPEL: Exemplo 2

```
<invoke partnerLink="goo:GoogleSearchService"
     portType="goo:GoogleSearchPort"
     operation="goo:doGoogleSearch"
     inputVariable="search"
     outputVariable="searchResponse"/>
  <assign>
  <copy>
     <from expression="1"/>
     <to variable="i"/>
  </copy>
  <copy>
     <from variable="searchResponse" part="return"
        query="count(/return/resultElements/item)"/>
     <to variable="itemNb"/>
  </copy>
  <copy>
    <from variable="search" part="key"/>
    <to variable="cache" part="key"/>
  </copy>
  </assign>
```

BPEL: Exemplo 2

```
<while condition="bpws:getVariableData('i') &lt;=
    bpws:getVariableData('itemNb')">
    <sequence>
       <assign>
         <copy>
           <from expression="concat(
              '/return/resultElements/item[',
              bpws:getVariableData('i'), ']/URL')"/>
           <to variable="cache" part="url"/>
         </copy>
         <copy>
            <from expression="bpws:getVariableData('i')+1"/>
            <to variable="i"/>
         </copy>
       </assign>
       <invoke partnerLink="goo:GoogleSearchService"
          portType="goo:GoogleSearchPort" operation="goo:doGetCachedPage"
          inputVariable="cache" outputVariable="cacheResponse"/>
    </sequence>
  </while>
</sequence>
```

Ferramentas para Programação com BPEL



Outros Paradigmas para Computação Distribuída (diferentes Cliente/Servidor)

Marco Túlio de Oliveira Valente

Espaços de Tuplas e Linda

Marco Túlio de Oliveira Valente

Espaços de Tuplas e Linda

- Linda: modelo de coordenação baseado na idéia de espaço de tuplas (David Gelernter, 1985)
- Diversas implementações: TSpaces (IBM), JavaSpaces, LightTS etc
- Modelo de coordenação: comunicação e sincronização de processos distribuídos
- Espaço de tuplas
  - Memória compartilhada, persistente, associativa e global
  - Processos distribuídos podem inserir, ler e retirar tuplas desta memória
- Tupla: sequência ordenada de valores
- Exemplos de tuplas:
  - ("João", "Rua Cinco", 125) ("João", "pisd") ("Maria", "pisd")
  - ("A", 1,1, 15) ("A", 1,2, 7) ("A", 2,1, 10) ("A", 2,2, 22)

Operações sobre Espaço de Tuplas

- Três operações básicas
  - out t: deposita uma tupla t no espaço
  - in p: retira uma tupla do espaço que "case" com o padrão p; se não existir, fica bloqueado até que exista
  - rd p: lê (sem retirar) uma tupla do espaço que que "case" com o padrão p; se não existir, fica bloqueado até que exista
- No caso de in e rd, se existir mais de uma tupla que “case” com o padrão, uma delas é não-deterministicamente escolhida
- Operações são atômicas e podem ser chamadas remotamente
- Existem ainda operações do tipo "probe":
  - inp p: retira uma tupla do espaço que "case" com o padrão p e retorna true; se não existir, retorna FALSE
  - rdp p: lê (sem retirar) uma tupla do espaço que que "case" com o padrão p e retorna true; se não existir, retorna false

Casamento de Padrões

- Se x é um variável do tipo t, ?x casa com qualquer valor do tipo t
- Exemplo 1:

```
int i;
out(1, 2, "xyz")
rd(?i, 2, "xyz”) é bem sucedido (i= 1)
```

- Exemplo 2:

```
int i, j;
out(1, 2, "xyz")
rd(?i, ?j, "xyz”) é bem sucedido (i= 1 e j= 2)
```

- Exemplo 3:

```
int i, j; string s;
out(1, 2, "xyz")
rd(?i, ?j, ?s) é bem sucedido (i= 1, j= 2, s= “xyz”)
```

Casamento de Padrões

- Exemplo 4:

```
int i, j=2; string s;
out(1, 2, "xyz")
rd(?i, j, ?s) é bem sucedido (i= 1, s= “xyz”)
```

- Exemplo 5:

```
int i, j=3; string s;
out(1, 2, "xyz")
rd(?i, j, ?s) não é bem sucedido (e permanece suspenso)
```

- Exemplo 6:

```
int i, j;
out(1, 2, "xyz")
rd (?i, ?j, "abc”) não é bem sucedido
```

- Exemplo 7:

```
out("string", 10.1, 21, "another string")
real f; int i;
rd("string", ?f, ?i, "another string") bem sucedido (f=10.1 e i=21)
in("string", ?f, ?i, "another string") bem sucedido (f=10.1 e i=21)
rd("string", ?f, ?i, "another string") permance suspenso
```

Espaço de Tuplas

- Espaço de Tuplas é uma memória:
  - Compartilhada: por diversos processos distribuídos
  - Persistente
  - Global: um espaço é uitlizado por diversos processos
  - Associativa: valores armazenados na memória (tuplas) são recuperados (operações in e rd) usando-se “padrões” (e não por meio de endereços, como ocorre em uma memória tradicional).
- Linda foi originalmente proposto para implementação de sistemas paralelos
- Periodicamente, interesse por Linda é renovado
- Razão: modelo de programação assíncrono e desacoplado de Linda é interessante em redes abertas, reconfiguráveis etc

Jantar dos Filósofos

```
process phil(i: 0 to n-1) { // cria n processos
  int i;
  while(true) {
    think();
    in("fork", i);
    in("fork", (i+1)%n);
    eat();
    out("fork", i);
    out("fork", (i+1)%n);
  }
}

process main() {
  int i;
  for (i=0, i<n, i++)
    out("fork", i);
}
```

Chat

- Tupla que identifica a próxima mensagem

```
out("nextmsg", 0)  // identificador da próxima mensagem
```

- Para enviar uma mensagem s:

```
int i;
in("nextmsg", ?i);
i++;
out("nextmsg", i);
out("msg", i, s);
```

- Para receber uma mensagem s:

```
i= 1;
while(true) {
  rd("msg", i, ?s); print s; i++;
}
```

- Deve existir ainda um processo coletor de lixo (mensagens antigas)
- Em TSpaces, pode-se associar um tempo de vida a uma tupla

RPC/RMI

- Suponha um servidor com métodos soma(x,y) e mult(x,y)
- Chamada remota realizada por um cliente c1:

```
int r;
out(c1,"soma",2,3)
in(c1,"somares",?r);
```

- Servidor:

```
int c,x,y,r; string m;
while(true) {
   in(?c,?m,?x,?y)
   if (m == "soma") r= soma(x,y);
   if (m == "mult") r= mult(x,y);
   out(c,m+"res",r)
}
```

- E se os métodos tivessem número e tipo de parâmetros diferentes
- Vantagens: transparência de localização, múltiplos servidores (tolerância falhas e balanceamento de carga), comunicação multicast, "chamadas assíncronas" etc

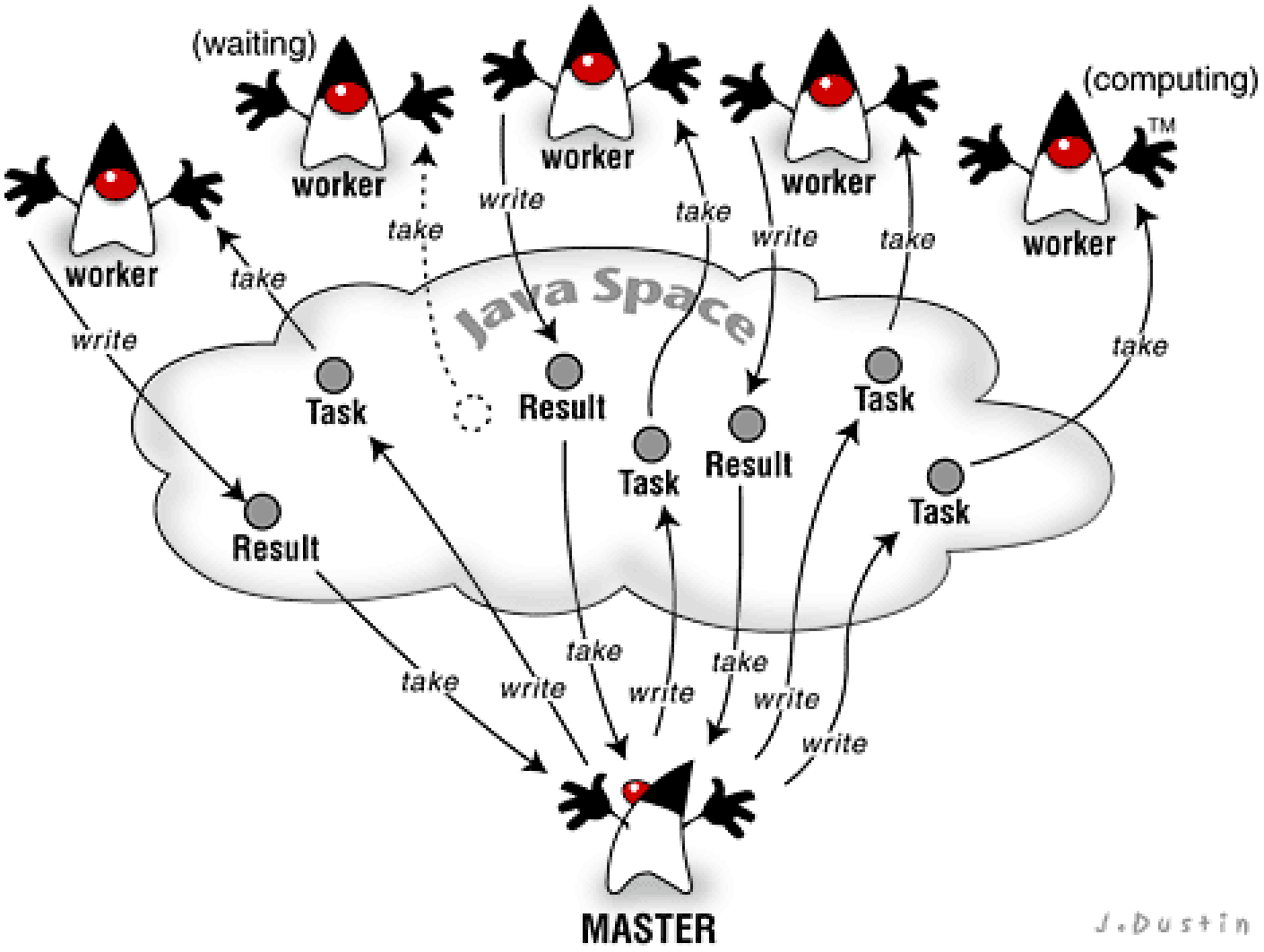
Sistema de Leilões

- Vendedor coloca um produto à venda:
  `out("pda", "palm", "lifedrive", 1500, id)`
- Comprador interessado em saber se existe um palm à venda:
  `rdp("pda", "palm", ?modelo, ?preco, ?id)`
- Comprador interessado em saber se existe um palm à venda ou em ser notificado quando um palm for colocado à venda :
  `rd("pda", "palm", ?modelo, ?preco, ?id)`
- Existem implementações de Linda que possuem operações que recuperam todas as tuplas que são compatíveis com um padrão
- Útil quando o comprador quer conhecer todos os palms à venda
- Existem ainda implementações que possuem estratégias de casamento de padrões mais sofisticadas.
- Exemplo: `rdp("pda", "palm", ?modelo, ?preco < 1000, ?id)`

Linda: Características Interessantes

- Comunicação em Linda é distribuída no espaço e no tempo.
- Distribuída no tempo porque um processo produtor pode inserir uma mensagem no espaço persistente, a qual somente será recuperada mais tarde por um processo consumidor. Assim, não é necessária a existência de uma conexão temporal entre processos para que eles possam interagir.
- Já distribuição no espaço é possível graças ao fato de que tuplas depositadas no repositório compartilhado são visíveis aos diversos nodos de um sistema distribuído. Além disso, graças ao uso de padrões para ler e remover tuplas, processos não precisam conhecer mutuamente seus identificadores para trocar mensagens. Por exemplo, processos consumidores são capazes de remover mensagens baseados no conteúdo das mesmas, sem que necessariamente tenham conhecimento da identidade dos processos produtores.

Mestre/Escravos



Números Primos

```
Process main(int argc, char *argv[]) {
  int primes[LIMIT] = {2,3}; /* my table of primes */
  int limit, i, isprime;
  int numPrimes = 2, value = 5;
  limit = atoi(argv[1]);    /* number of primes to generate */
  /* put first candidate in bag */
  OUT("candidate", value);
  /* get results from workers in increasing order */
  while (numPrimes < limit) {
    IN("result", value, ?isprime);
    if (isprime) { /* put value in table and TS */
       primes[numPrimes] = value;
       numPrimes++;
       OUT("prime", numPrimes, value);
    }
   value= value + 2;
  }
  /* tell workers to quit, then print the primes */
  OUT("stop");
  for (i = 0; i < limit; i++) printf("%d\n", primes[i]);
```

Números Primos

```
process worker(1 to numWorkers) { // create numWorkers processes
  int primes[LIMIT] = {2,3}; /* table of primes */
  int numPrimes = 1, i, candidate, isprime;
  /* repeatedly get candidates and check them */
  while(true) {
    if (RDP("stop")) /* check for termination */
       return;
    IN("candidate", ?candidate);     /* get candidate */
    OUT("candidate", candidate+2);   /* output next one */
    i = 0; isprime = 1;
    while (primes[i]*primes[i] <= candidate) {
      if (candidate%primes[i] == 0) /* not prime */
         { isprime = 0; break; }
      i++;
      if (i > numPrimes) /* need another prime */
         { numPrimes++; RD("prime", numPrimes, ?primes[numPrimes]); }
    }
    /* tell manager the result */
   OUT("result", candidate, isprime);
  }
}
```

Características Interessantes

- Comunicação "desacoplada no tempo”
  - Produtor e consumidor de mensagens não precisam estar temporalmente conectados para que ocorra uma comunicação
  - Não se precisa estabelecer uma "conexão" para que processos possam se comunicar
  - Importante, por exemplo, em sistemas para computação móvel
- Comunicação "desacoplada no espaço”
  - Cliente não precisa conhecer localização do servidor (e vice-versa)
  - Comunicação baseada em "casamento de padrões" dispensa o uso de endereços físicos para identificar mensagens
  - Tuplas não são recuperadas com base em "endereços de destino", mas no conteúdo das mesmas
  - Facilita estratégias de tolerância a falhas, balanceamento de carga, replicação etc

Características Interessantes

- Comunicação unicast e multicast
  - Unicast: "recepção" usando in
  - Multicast: "recepção" usando rd
- Sincronização é inerente ao modelo
  - in e rd possuem uma semântica de bloqueio
  - "Consumidor" fica bloqueado se tentar consumir uma mensagem que ainda não foi gerada por um "produtor”
- Modelo de comunicação “generativo” (generative)
  - Para se comunicar, processos não trocam mensagens (no sentido tradicional, via canais) nem compartilham variáveis
  - Em vez, mensagens são geradas, depositadas e recuperadas de um espaço de tuplas

Sistemas Publish/Subscribe

Sistemas Publish/Subscribe

- Eugster, Felber, Guerraoui, Kermarrec: The many faces of publish/subscribe. ACM Computing Surveys 35(2): 114-131 (2003)
- Motivação: modelo de comunicação ponto-a-ponto e síncrono de sistemas baseados em RPC é por demais rígido e estático quando aplicado em sistemas distribuídos de larga escala (Internet) ou dinâmicos (redes móveis)
- Nestes ambientes, modelos de comunicação fracamente acoplados são mais interessantes
- Sistemas Publish/Subscribe:
  - Assinantes manifestam interesse em um determinado evento ou em um conjunto de eventos
  - Produtores publicam eventos
  - Eventos são assincronamente publicados para seus assinantes

Sistemas Publish/Subscribe

- Serviço de eventos:
  - Mediador entre produtores e assinantes
  - Gerencia e armazena assinaturas
  - Despacha eventos para respectivos assinantes
- Inclusão e remoção de produtores e assinantes é simples
- Oferecem desacoplamento no espaço, no tempo e de sincronização

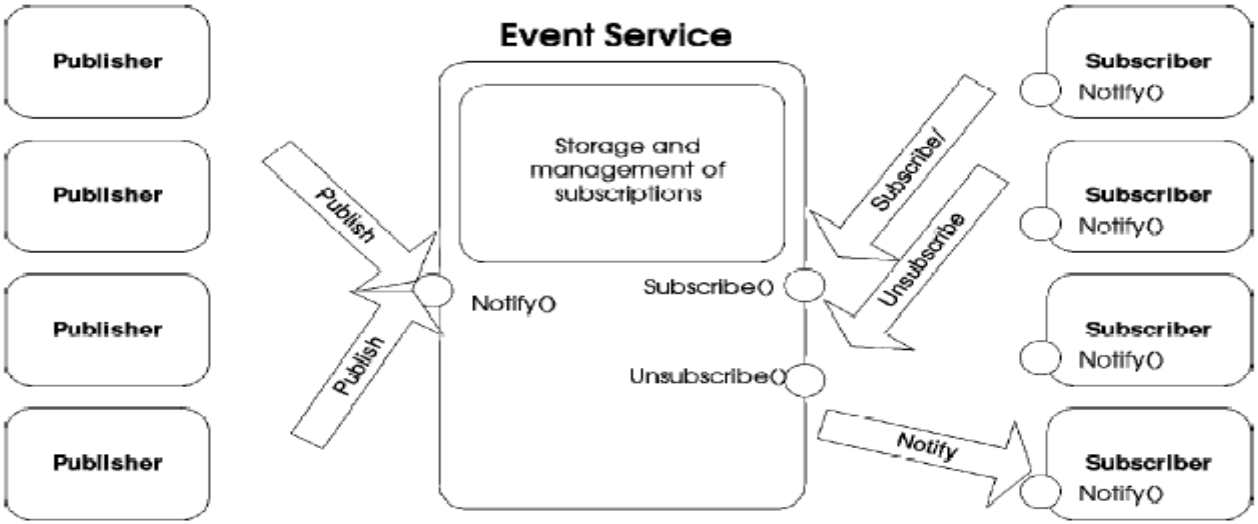
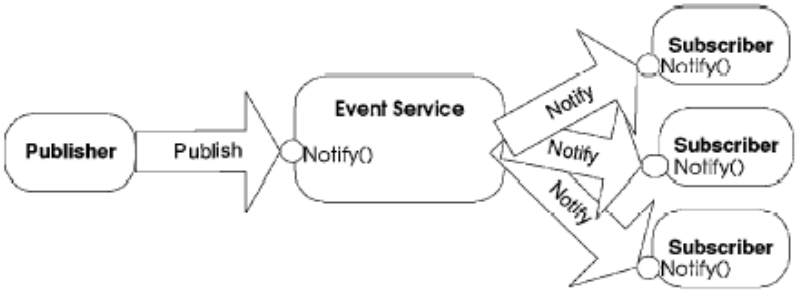


Fig. 1. A simple object-based publish/subscribe system.

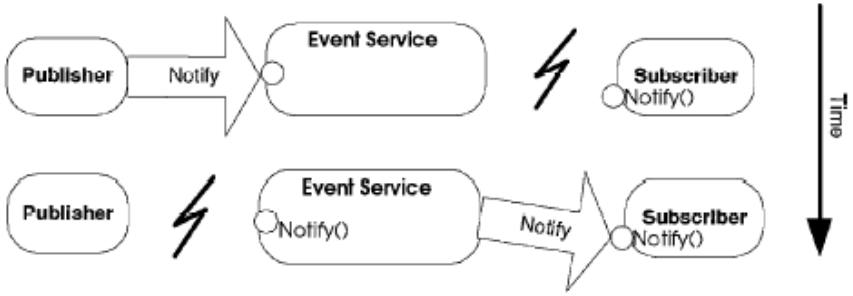
Desacoplamento

- Desacoplamento no espaço:
  - Produtores não conhecem assinantes (e vice-versa)
  - Produtores não tem referências para assinantes (e vice-versa)
  - Participantes não se conhecem mutuamente
- Desacoplamento no tempo:
  - Produtor pode gerar evento quando assinante desconectado
  - Assinante pode receber evento quando produtor desconectado
  - Não é necessário estabelecer uma conexão entre participantes
- Desacoplamento de sincronização:
  - Produção/consumo não acontece no fluxo principal de execução
  - Produtores não ficam bloqueados enquanto evento é enviado
  - Assinantes são notificados assincronamento (via um callback)
- Importância em sistemas de larga escala e dinâmicos: remover “amarras” ou “ligações” entre participantes

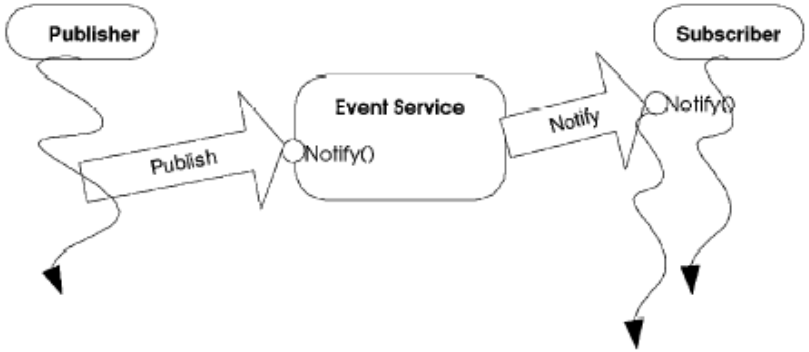
Desacoplamento



Space decoupling



Time decoupling



Synchronization decoupling

Alternativas

- Troca de mensagens
- RPC
- Espaços de Tuplas
- Filas de Mensagens

Troca de Mensagens

- Forma de comunicação de mais “baixo nível”
- Duas primitivas: send (assíncrono) e receive (síncrono)
- Exemplo: sockets
- Hoje, raramente utilizada pois torna visível à camada de aplicação:
  - Marshalling/unmarshalling, falhas, retransmissões etc
- Não provê desacoplamento no tempo e no espaço

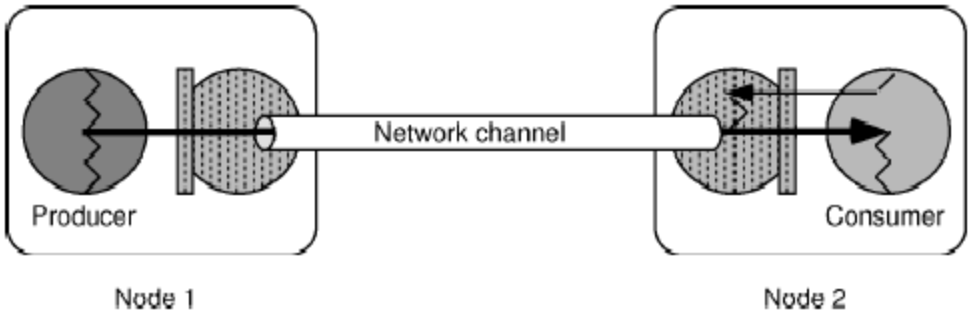
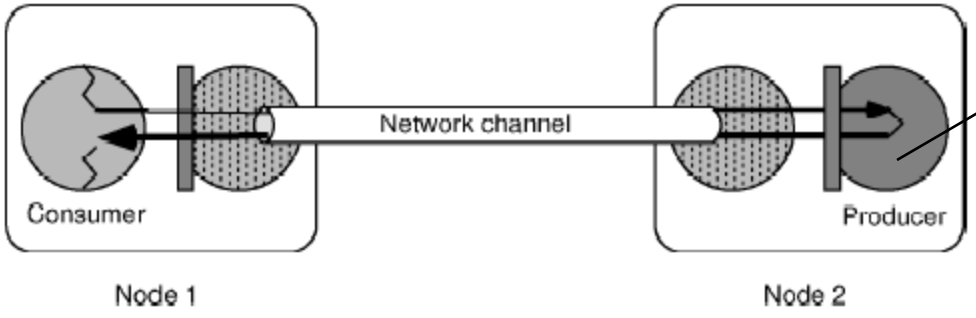


Fig. 3. Message passing interaction: the producer sends messages asynchronously through a communication channel (previously set up for that purpose). The consumer receives messages by listening synchronously on that channel.

RPC e derivados

- Torna programação distribuída “muito fácil” (MT: exagero ??)
- Não provê desacoplamento no tempo
- Não provê desacoplamento no espaço (clientes devem possuir uma referência para servidores)



Produtor: emite reply

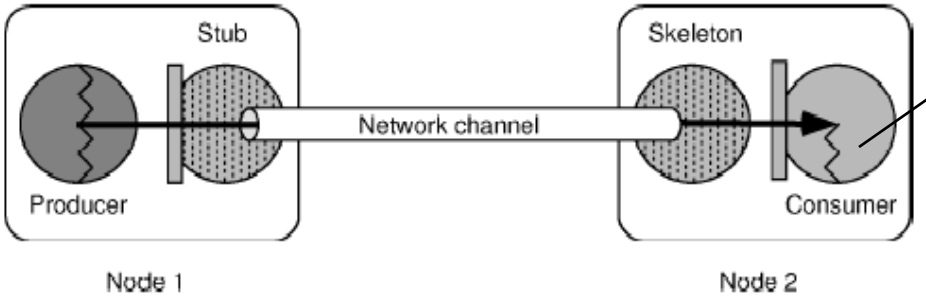
Legenda, no entanto, continua parecendo errada

Fig. 4. RPC and derivatives: the producer performs a synchronous call, which is processed asynchronously by the consumer.

- Nota de rodapé: “the distinction of consumer/producer roles is not straightforward in RPC. We assume here that an RPC which yields a reply attributes a consumer role to the invoker, while the invokee acts as producer. As we will point out, the roles are inversed with asynchronous invocations (no reply).”

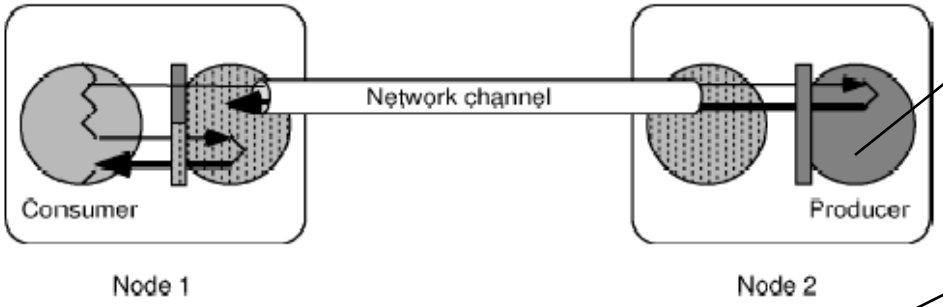
RPC e derivados

- Tentativas de diminuir desacoplamento de sincronização:
  - Chamadas one-way e chamadas assíncronas



Como não há reply, Neste caso, o servidor é o consumidor

Fig. 5. Decoupling synchronization with asynchronous remote invocation: the producer does not expect a reply.



Como há reply, o servidor é o produtor

Legenda, no entanto, continua parecendo errada

Fig. 6. Decoupling synchronization with future remote invocation: the producer is not blocked and can access the reply later when it becomes available.

Espaços de Tuplas

- Provê desacoplamento no tempo e no espaço
- Desvantagem: consumidores retiram tuplas de forma síncrona, o que limita escalabilidade de clientes (MT: basta usar inp ou usar in em uma thread independente)

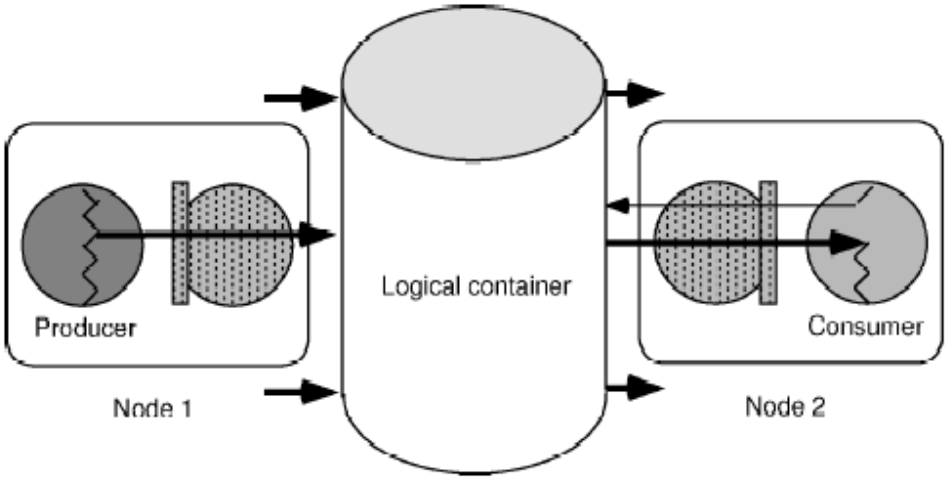


Fig. 8. Shared space: producers insert data asynchronously into the shared space, while consumers read data synchronously.

Fila de Messagens

- Sistemas de middleware orientados por mensagens (MOM)
- Exemplos: JMS, IBM MQSeries, Oracle AQ, BEA Tuxedo
- Desvantagem: consumidores são síncronos, isto é, não provê desacoplamento de sincronização

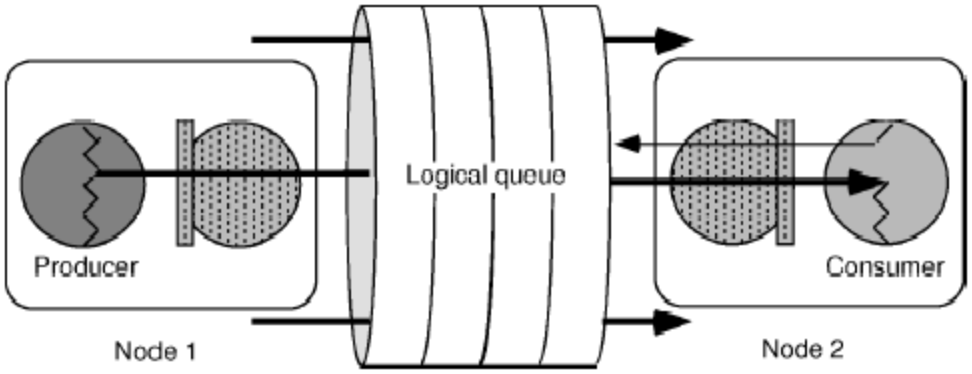


Fig. 9. Message queuing: messages are stored in a FIFO queue. Producers append messages asynchronously at the end of the queue, while consumers dequeue them synchronously at the front of the queue.

Resumo

Table 1. Decoupling Abilities of Interaction Paradigms

Abstraction	Space decoupling	Time decoupling	Synchronization decoupling
Message passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (observer pattern)	No	No	Yes
Tuple spaces	Yes	Yes	Producer-side
Message queuing (Pull)	Yes	Yes	Producer-side
Publish/subscribe	Yes	Yes	Yes

- Característica distintiva de P/S:
  - Desacoplamento no espaço
  - Desacoplamento no tempo
  - Desacoplamento de sincronização

Sistemas Publish/Subscribe

- Esquemas de assinatura de eventos:
  - Baseada em Tópicos
  - Baseada em Conteúdo
  - Baseada em Tipo

Assinatura baseada em Tópicos

- Eventos publicados e assinados são identificados por tópicos
- Tópico: conjunto de palavras chave, normalmente hierarquizadas
- Cliente assina eventos sobre um tópico especificamente
- Exemplos: LondonStockMarket/Stock/StockQuotes

```
public class StockQuote implements Serializable {
   public String id, company, trader;
   public float price;
   public int amount;
}
public class StockQuoteSubscriber implements Subscriber {
   public void notify(Object o) {
      if (((StockQuote)o).company == 'TELCO' && ((StockQuote)o).price < 100)
         buy();
   }
}
// ...
Topic quotes = EventService.connect("/LondonStockMarket/Stock/StockQuotes");
Subscriber sub = new StockQuoteSubscriber();
quotes.subscribe(sub);
```

Assinatura baseada em Conteúdo

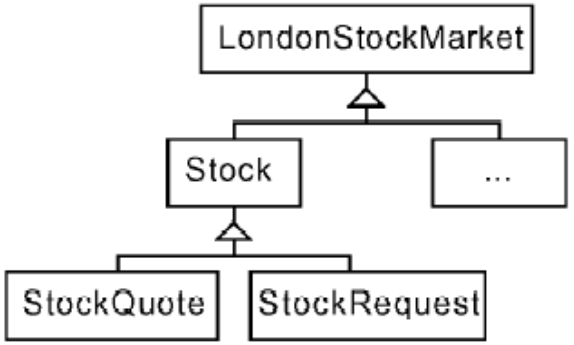
- Esquema de assinatura mais flexível e dinâmico, baseado no conteúdo real dos eventos gerados
- Assinantes selecionam eventos usando filtros especificados em uma “linguagem de assinatura”

```
public class StockQuote implements Serializable {
   public String id, company, trader;
   public float price;
   public int amount;
}
public class StockQuoteSubscriber implements Subscriber {
   public void notify(Object o) {
      buy();       // company == 'TELCO' and price < 100
   }
}
// ...
String criteria = ("company == 'TELCO' and price < 100");
Subscriber sub = new StockQuoteSubscriber();
EventService.subscribe(sub, criteria);
```

Assinatura baseada em Tipo

- Assinante manifesta interesse em eventos do tipo t implementando método notify que espera como parâmetro um objeto deste tipo

```
public class LondonStockMarket implements Serializable {
   public String getId() {...}
}
public class Stock extends LondonStockMarket {
   public String getCompany() {...}
   public String getTrader() {...}
   public int getAmount() {...}
}
public class StockQuote extends Stock {
   public float getPrice() {...}
}
public class StockRequest extends Stock {
   public float getMinPrice() {...}
   public float getMaxPrice() {...}
}
public class StockSubscriber implements Subscriber<StockQuote> {
   public void notify(StockQuote s) {
      if (s.getCompany() == 'TELCO' && s.getPrice() < 100)
         buy();
   }
}
// ...
Subscriber<StockQuote> sub = new StockSubscriber();
EventService.subscribe<StockQuote>(sub);
```




```
interface Subscriber<S> {
   notify(S s);
}
```

Mobilidade de Código e Agentes Móveis

Mobilidade de Código

- Estilo mais difundido de mobilidade
- Também chamado de mobilidade fraca
- Mobilidade de código: designa programas capazes de trafegar por uma rede heterogênea, cruzando diferentes domínios administrativos, e que são automaticamente executados ao atingirem uma estação de destino
- Mobilidade apenas do código da aplicação (fonte ou intermediário)
- Não é um conceito novo: linguagem Postscript (Adobe, 1985)
- Vantagens de mobilidade de código:
  - Viabiliza a execução na diversidade de arquiteturas da Internet
  - Viabiliza a construção de aplicações Internet interativas
  - Simplifica a administração e manutenção de redes

Mobilidade de Agentes

- Agente:
  - Programa que realiza uma tarefa para a qual recebeu delegação por parte de um usuário
  - Usados em Robótica, Inteligência Artificial, Bancos de Dados, Recuperação de Informação etc
- Agente Móvel:
  - Agente cuja execução não se restringe a uma única máquina
  - Programa que autonomamente se desloca pelas diversas máquinas de uma rede a fim de melhor realizar a tarefa que lhe foi delegada.
- Mobilidade: código + dados + controle
  - Mobilidade de estado ou mobilidade de código forte
- Agentes móveis: modelo alternativo para construção de aplicações distribuídas na Internet
- Idéia: mover computação para junto de seus dados

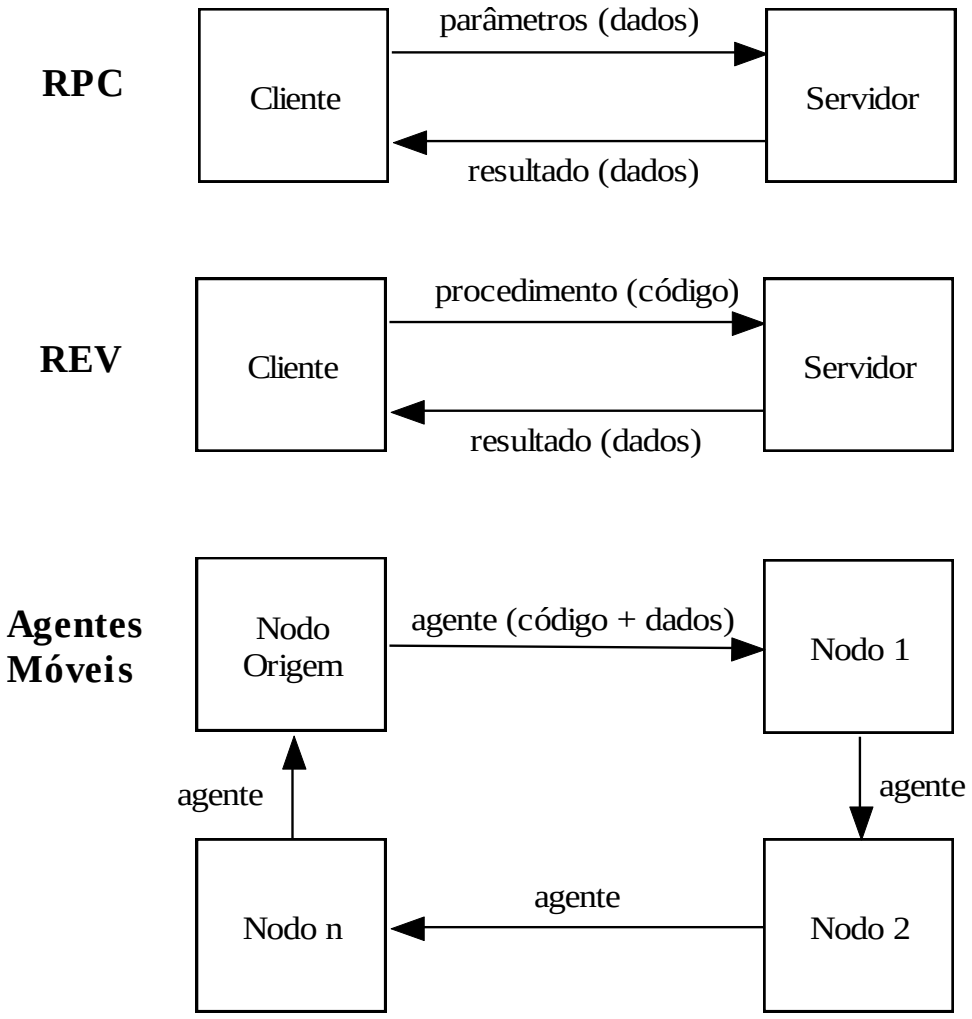
Modelos de Aplicações Distribuídas

- Principais modelos para implementação de aplicações distribuídas:
  - Modelo Cliente/Servidor
  - Avaliação Remota
  - Agentes Móveis
- Avaliação Remota (REV):
  - Generalização do modelo de RPC, permitindo a passagem de procedimentos como parâmetros
- Exemplo de REV:

```
float integral (float (*fun)(float), float inferior, float superior)
```

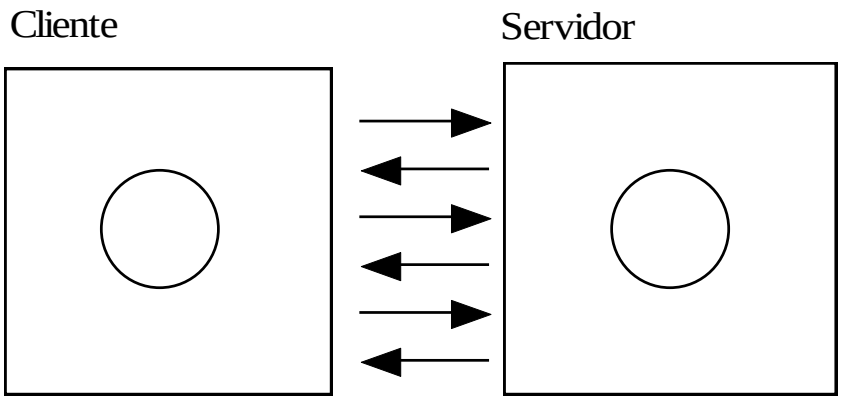
- Principais vantagens de REV:
  - Flexibilidade: não fixa conjunto de serviços que podem ser invocados remotamente
  - Redução do montante de comunicação: envia-se o programa para junto dos dados que irá manipular

RPC x REV x Agentes Móveis

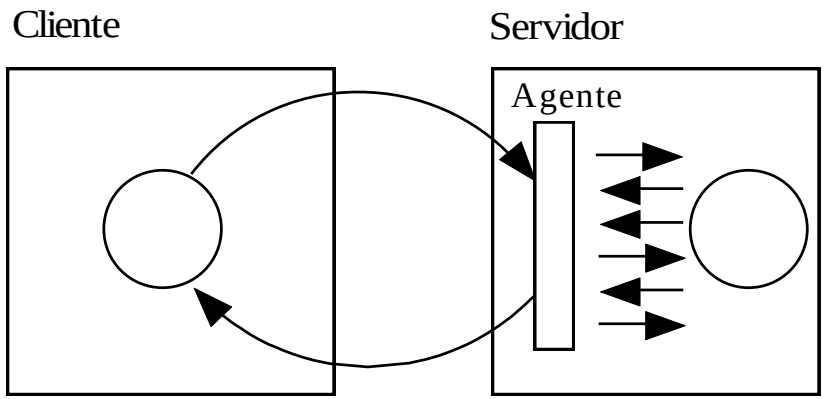


Vantagens de Agentes Móveis

- Principais vantagens:
  - Redução de tráfego na rede
  - Eliminação da latência da rede
- Aplicações que podem se beneficiar do modelo de agentes móveis:
  - Comércio eletrônico, gerência de redes de telecomunicações, aplicações de *workflow* e *groupware,* recuperação de informação



Modelo Cliente/Servidor



Modelo de Agentes Móveis

Linguagens com Suporte a Mobilidade de Agentes

- Principais linguagens com suporte a mobilidade de agentes:
  - Obliq
  - Telescript
  - Linguagens baseadas em Java:
    - Aglets
    - Voyager
    - JavaSeal
    - etc

Dynamically Programmable and Reconfigurable Middleware Services

Manuel Román, Nayeem Islam

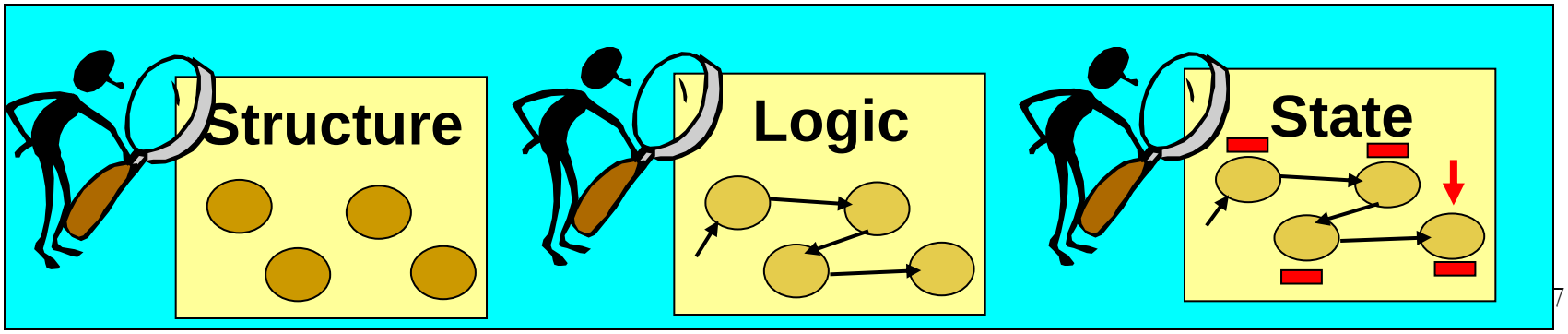
Middleware Conference, 2004

Introdução

- Manuel Román, Nayeem Islam: Dynamically Programmable and Reconfigurable Middleware Services. Middleware 2004: 372-396
- Desenvolvido no NTT DoCoMo Labs (USA)
- Middleware para telefones celulares deve atender aos requisitos:
  - Configuração estática e dinâmica: cliente ou cliente/servidor, com ou sem interceptadores, protocolo (IIOP, SOAP etc) etc
  - Correção dinâmica (updateability): 10% dos celulares possuem erros de software que implicam em retorno aos fabricantes
  - Atualização dinâmica: novas funcionalidades, protocolos, interfaces etc
- Proposta: infra-estrutura de middleware para celular que permite:
  - Corrigir, configurar e atualizar o comportamento do sistema de de forma dinâmica (sem interromper a execução de aplicações) e sem requerer intervenção por parte de usuários finais.

Introdução

- Baseado em uma técnica chamada de externalização
- Idéia: externalizar o estado, a lógica e a estrutura de componentes internos do middleware
- Externalizar: tornar visível e sujeito a inspeção e modificação
- Objetivo: permitir que desenvolvedores (e pessoal de suporte) possuam controle sobre serviços providos pelo middleware
- Externalização:
  - Estrutura: componentes que compõem o middleware
  - Lógica: regras de interação entre componentes
  - Estado: atributos internos do middleware



DPRS

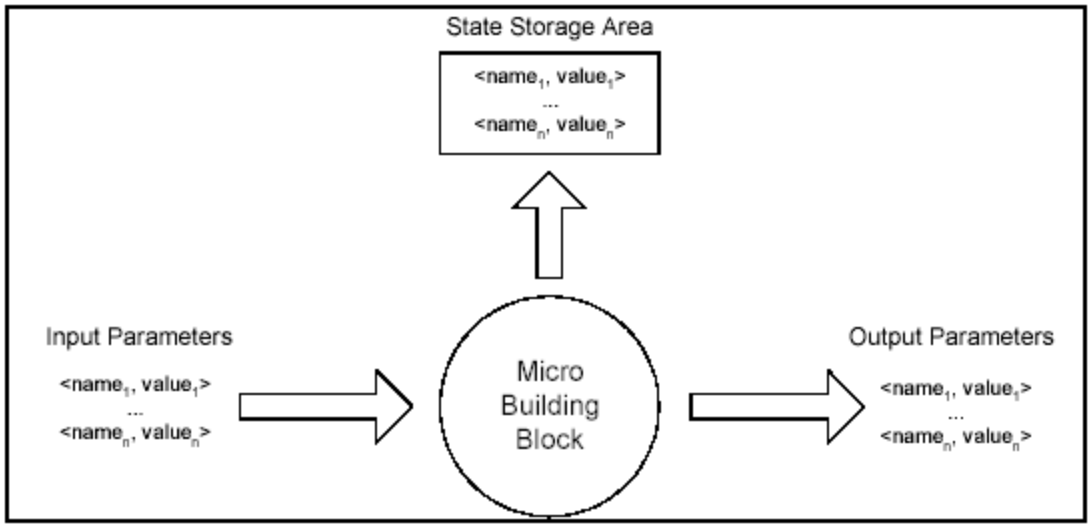
- Dynamically Programmable and Reconfigurable Middleware Services
- Arquitetura para construção de sistemas de middleware com suporte a externalização
- Vantagens:
  - Configurability
  - Updateability
  - Upgrateability
- Três abstrações principais:
  - Micro Building Blocks (MBBs)
  - Ação
  - Domínio

MBB

- Menor unidade funcional do sistema (componente mínimo)
- Em DPRS, um middleware é formado por um conjunto de MBBs
- MBB:
  - Recebe um conjunto de parâmetros de entrada
  - Executa uma ação (que pode atualizar seu estado)
  - Produz um conjunto de parâmetros de saída
- Exemplo: registerObject
  - Entrada: nome do objeto e referência de rede para o mesmo
  - Ação: atualizar tabela interna de objetos registrados
  - Saída: void

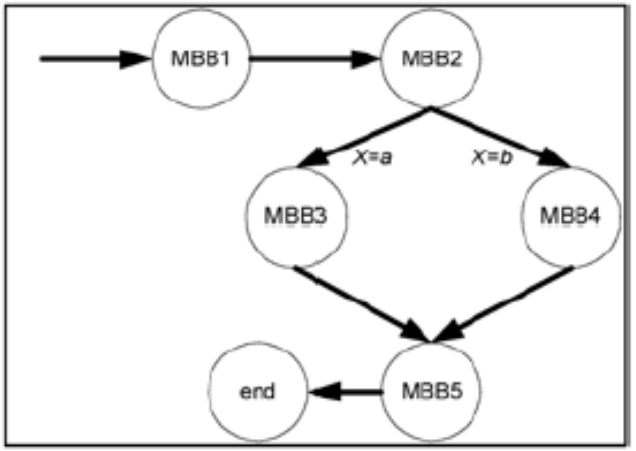
MBB

- Entrada, estado e saída: tuplas na forma (nome, valor)
- Estado é armazenado em uma área gerenciada pelo sistema
  - Facilita troca do MBB, já que não é necessário transferir estado
- MBBs não possuem “referências” para outros MBBs



Ação

- Especifica a ordem de execução dos MBBs (lógica do sistema)
- Pode ser interpretada ou compilada
- Ação interpretada: grafo
  - Nodos são MBBs e arestas definem o fluxo de execução
  - Arestas podem possuir uma condição usada para determinar o próximo MBB a ser executado
- Artigo não mostra como grafo é definido (função de transição etc)
- Programadores podem modificar o grafo em tempo de execução



Ação Compilada

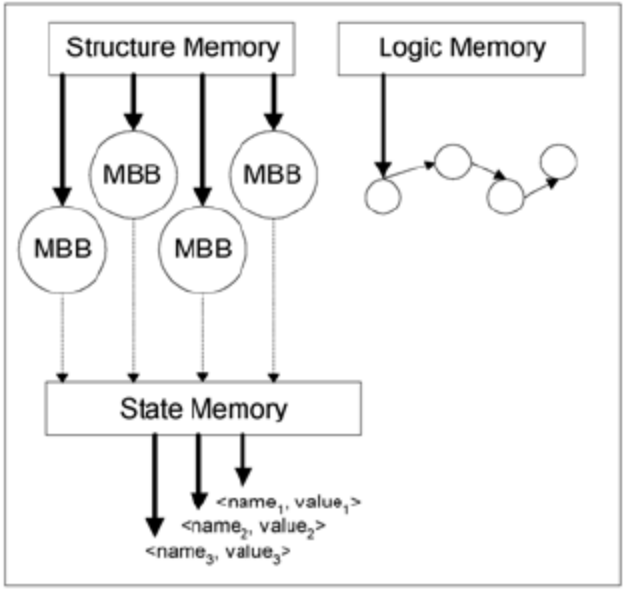
- Uma ação compilada é um fragmento de código que especifica a ordem de invocação de MBBs

```
action Test
{
    outputParams = InvokeMBB(MBB1, inputParams);
    outputParams = InvokeMBB(MBB2, outputParams);
    char X = outputParams.get("X");
    if (X=='a')
      outputParams = InvokeMBB(MBB3, outputParams);
    if (X=='b')
      outputParams = InvokeMBB(MBB4,outputParams);
    outputParams = InvokeMBB(MBB5, outputParams);
}
```

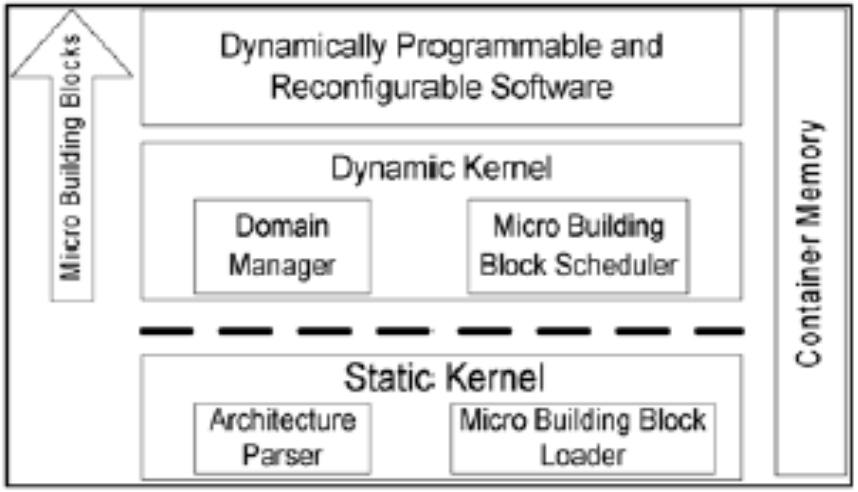
- Artigo não define linguagem usada para descrever o código da ação
- Código pode chamar métodos da biblioteca do sistema
- Ação compilada: execução mais rápida que a ação interpretada
- Porém, ao contrário da ação interpretada, não permite inspeção e modificação em tempo de execução (menos flexível)
- Modificações na ordem de execução requerem modificações no código da ação e recompilação

Domínio

- Coleção de MBBs + memória de estado + coleção de ações
- Especificado via um descritor de arquitetura
- Domínios podem ser hierarquicamente organizados



Run-time



- MBB Scheduler: controla execução do sistema (percorre o grafo de execução, avalia condições e chama MBBs)
- MBB Scheduler pode ser substituído/configurado pelo usuário (real-time, tolerância a falhas, concorrência etc)
- MBB Scheduler externaliza as seguintes informações:
  - Ação que está sendo executada
  - MBB que está sendo executado (bem como seus parâmetros)

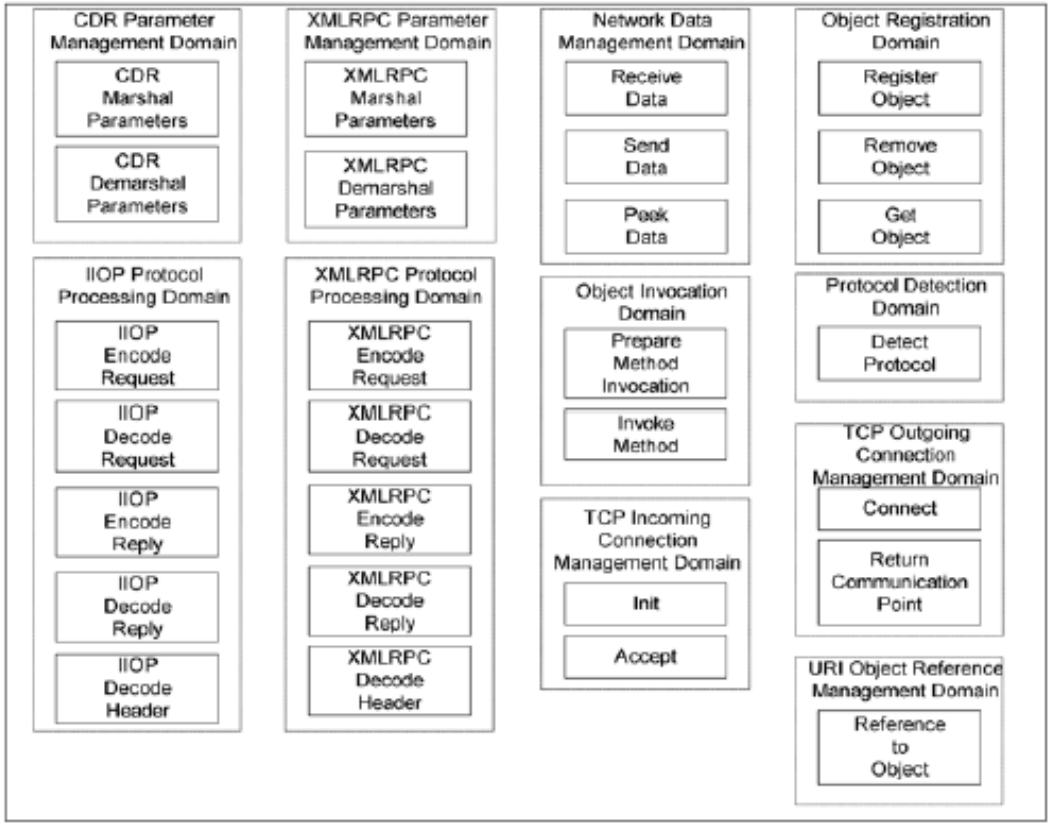
Run-time

- MBB Manager: módulo responsável pela inserção, remoção e substituição de MBBs
- Pontos de reconfiguração: pontos onde é possível reconfigurar o sistema de forma segura e consistente
- Principal contribuição do modelo de execução de DPRS: pontos de reconfiguração podem ser determinados automaticamente
- Em DPRS, pontos de reconfiguração ocorrem entre invocações de MBBs

Exemplo de Uso e Avaliação

ExORB

- Middleware construído seguindo a arquitetura DPRS:
  - Funcionalidade cliente ou cliente/servidor
  - Protocolos de middleware: IIOP ou XML-RPC
- 28 MBBs, pertencentes a 11 domínios, ocupando ao todo 70 KB



ExORB: Domínios

1. Gerenciamento de parâmetros no formato CDR: marshal e unmarshall em CDR (padrão CORBA de representação de dados)
2. Gerenciamento de parâmetros XML-RPC: idem, mas XML-RPC
3. Codificação e decodificação de mensagens IIOP
4. Codificação e decodificação de mensagens XML-RPC
5. Envio e recebimento de dados pela rede
6. Invocação de objetos: preparação e invocação de métodos remotos
7. Recebimento de Conexões TCP
8. Estabelecimento de Conexões TCP
9. Registro de objetos remotos
10. Identificação do protocolo de middleware (IIOP ou XMLRPC)
11. Gerenciamento de URIs: dada a URI de um objeto, retorna identificação do mesmo

ExORB: Avaliação Quantitativa

- Servidor: método que calcula o cubo de um dado inteiro
- Cliente: chama servidor 10 mil vezes
- Chamadas/segundo foram medidas para:
  1. Configuração estática : implementação não baseada em MBBs
  2. Versão não otimizada de ExORB
  3. Versão de ExORB com ações compiladas
  4. Versão de ExORB com ações interpretadas e com a memória consistindo de tuplas na forma (inteiro, valor), isto é, variáveis são índices de um vetor e não mais strings usadas como chaves de uma tabela hash
  5. Versão de ExORB com ações compiladas e índices

ExORB: Avaliação Quantitativa

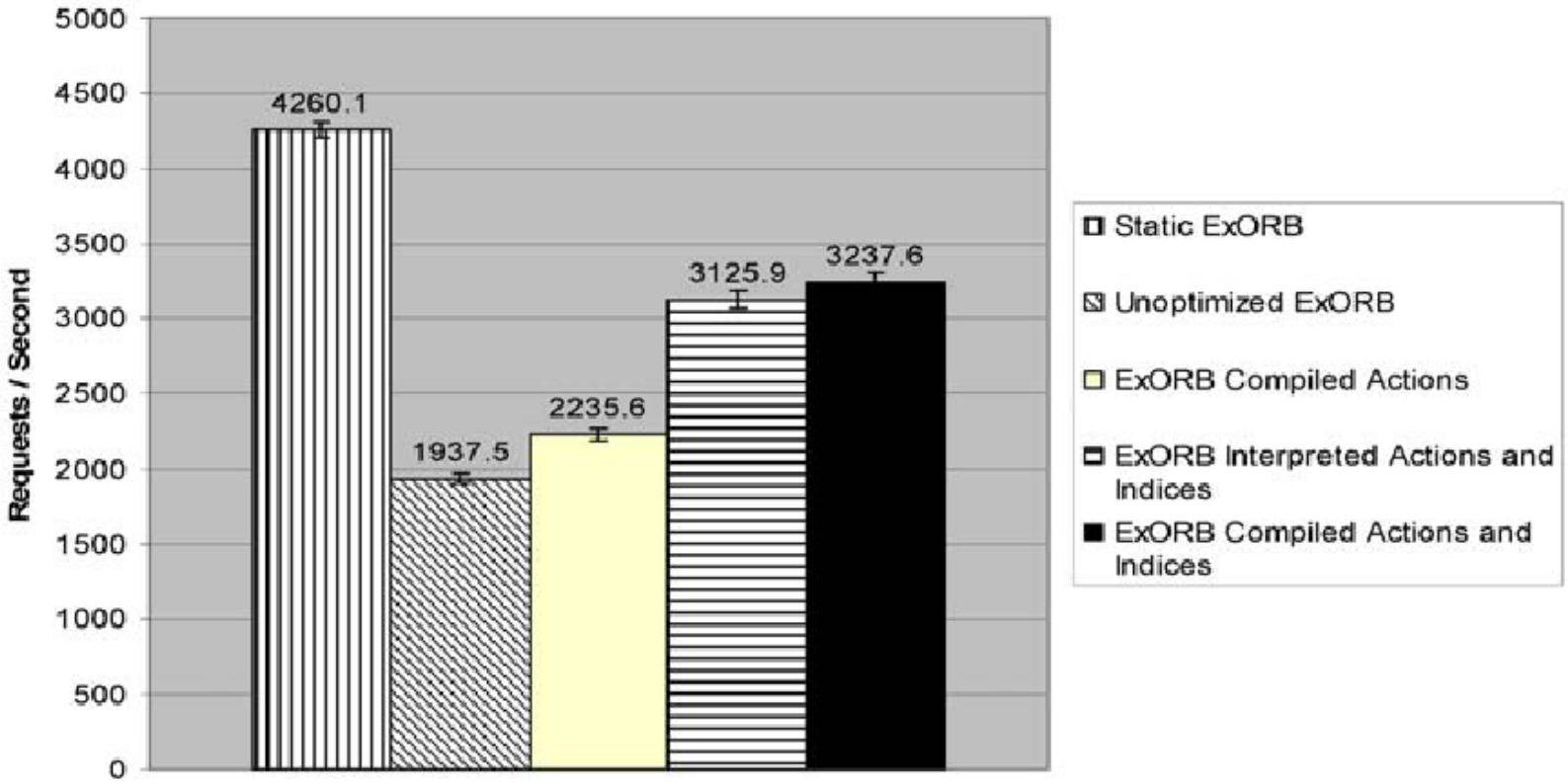
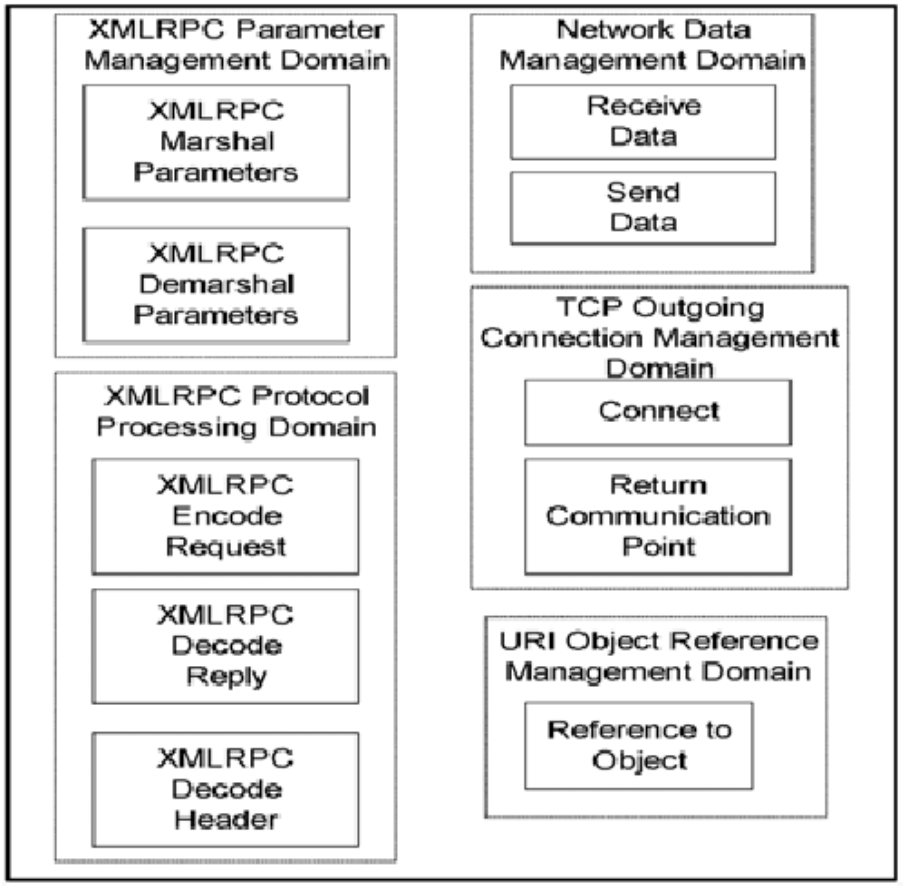


Fig. 12. DPRS Performance Evaluation.

- Diferença entre ações compiladas vs interpretadas não é grande
- “Otimizações” introduzem algum ganho de desempenho

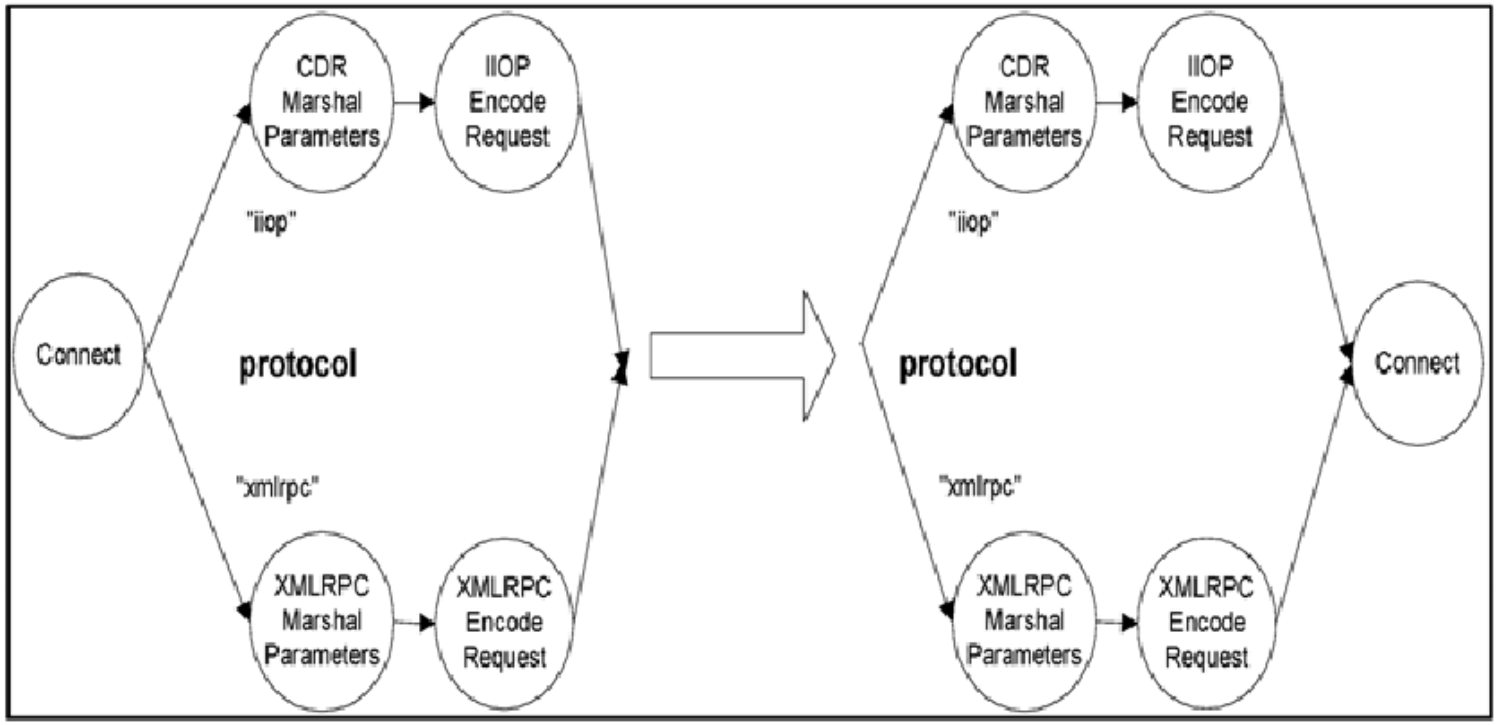
ExORB: Avaliação Qualitativa

- Configurability:
  - Versão com apenas funcionalidade cliente e XML-RPC (43 KB)
  - Versão foi gerada modificando o descritor de arquitetura



ExORB: Avaliação Qualitativa

- Updateability:
  - MBBs e ações podem ser modificados pelo MBB Manager
  - Experimento realizado: alteração da ação para primeiro realizar o marshalling dos parâmetros e depois conectar com objeto remoto



ExORB: Avaliação Qualitativa

- Upgradeability:
  - Criptografia de mensagens (novos MBBs para criptografar e descriptografar mensagens; nova versão da ação para chamar estes MBBs)