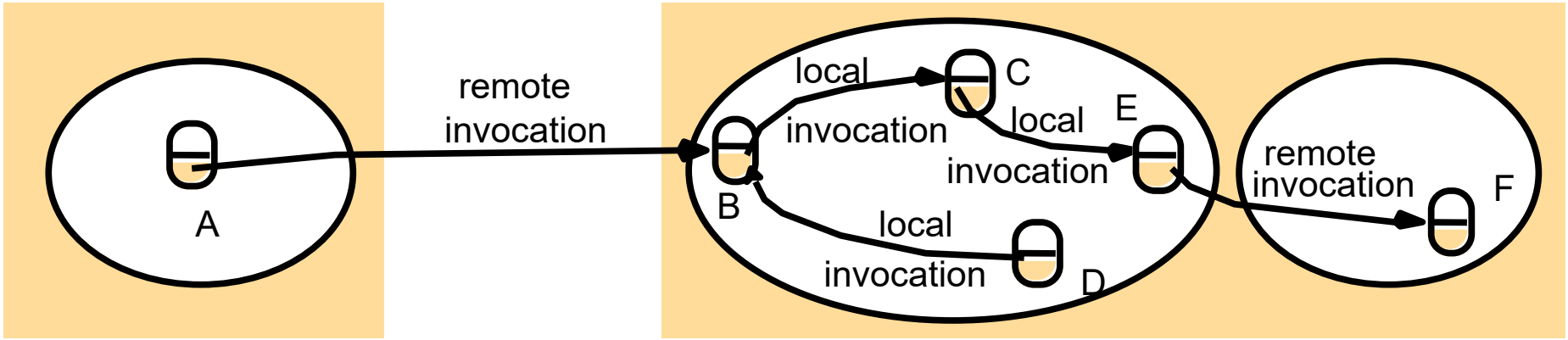
Invocação de Métodos em Objetos Remotos

Invocações de métodos remotas e locais



Invocação de Métodos em Objectos Remotos

- Um sistema de objetos distribuídos é uma extensão ao conceito de RPC
- Um objeto invoca um método noutro objeto localizado remotamente
- Num sistema de objetos, o RPC designa-se por **Remote Method Invocation** ou **RMI**
- A IDL é uma linguagem orientada aos objetos tipicamente baseada em C++, Java, ou C#

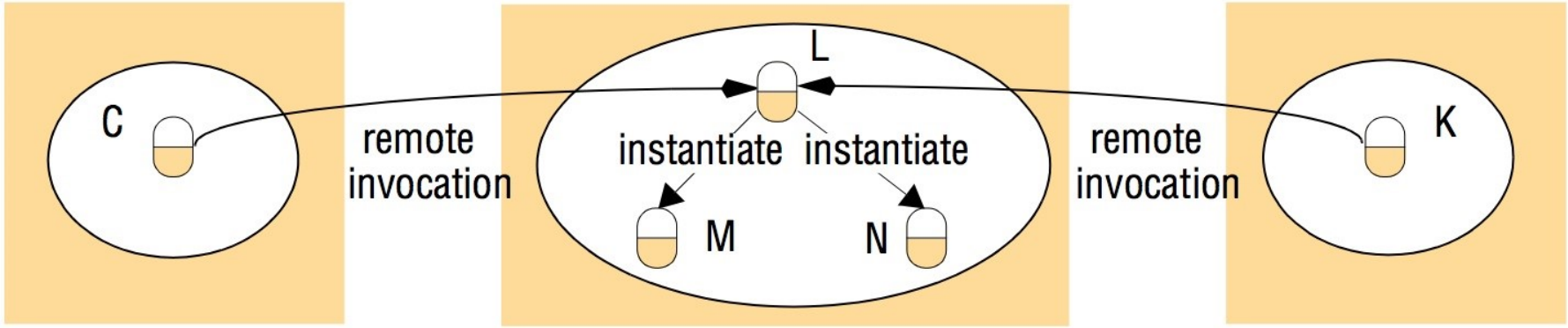
RMI vs RPC

- Semelhanças do RMI com RPC:
  - Usamos interfaces para definir os métodos invocáveis remotamente
  - Existe um protocolo de invocação remota que oferece as semânticas habituais de RPC
    - Pelo-menos-uma-vez, no-máximo-uma-vez, etc.
  - Temos um nível de transparência semelhante
    - Invocações locais e remotas com sintaxe próxima, mas programador é exposto a alguns aspetos da distribuição

RMI vs RPC

- Diferenças do RMI face ao RPC:
  - Programador tem ao seu dispor o poder completo do paradigma OO
    - Objetos, classes, herança, polimorfismo, etc.
  - Passagem por referência é agora permitida
    - Cada objeto remoto pode ser referenciado remotamente
    - Referências remotas podem ser passadas por argumento ou retorno quando método é invocado remotamente
  - Num servidor existem vários objetos remotos que oferecem as respetivas interfaces remotas
    - Em RPC, 1 servidor oferecia 1 interface remota
  - E outros aspetos que analisaremos em breve
    - (Seção Características Avançadas)

Invocação de métodos



- Invocação num sistema de objetos distribuído
  - Iniciada pela invocação de um método sobre uma referência para um objeto
  - Invocação pode resultar noutras invocações de métodos noutros objetos, quer locais quer remotos
  - Podem levar à criação de novas instâncias de objetos
    - Normalmente a instância reside na mesma máquina do objeto criador

Modelo de Objetos Distribuído

Interface Remota

- Conjunto de métodos de uma classe que
  podem ser invocados remotamente nos objetos dessa classe
  - Uma classe pode definir métodos
    que podem ser acedidos apenas localmente

- A especificação da interface é semelhante a
  uma classe abstrata ou interface na linguagem Java

Exemplo em Java RMI: Interfaces remotas

```
import java.rmi*;

public interface Account extends Remote {
      float debit(float amount) throws RemoteException, InsufficientFundsException;
      float credit(float amount) throws RemoteException;
}
public interface AccountList extends Remote {
      Account getAccount(int id) throws RemoteException;
}
```

O super-tipo **Remote** indica uma interface que tem métodos invocáveis remotamente

Os métodos remotos lançam uma exceção especifica: **RemoteException**

Exemplo em Java RMI:
Classe que implementa interface remota

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class AccountListServant
        extends UnicastRemoteObject
        implements AccountList {
        private Vector<Account> theList; // contains the list of accounts
        public AccountListServant()throws RemoteException { ... }
        public Account getAccount(int id) throws RemoteException {
                return theList[id];
        }
}
Public class AccountImpl implements Account { ... }
```

Referências remotas

- Os objetos são invocados através de referências que os identificam
- Num sistemas distribuído os objetos remotos têm referências remotas
- Referência para um objeto remoto
  - Referência para um objeto conhecido e sobre o qual pode ser efetuada a invocação de um método (evolução do *binding handle*)
  - Podem ser passados como argumentos ou valores de retorno dos métodos remotos

Continuação do exemplo (I)
classe BankServer com o método main

```
import java.rmi.*;
public class BankServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            AccountList accountList = new AccountListServant();
            Naming.rebind("AccountList", accountList );
            System.out.println("accountLis  server ready" );
        } catch(Exception e) {
            System.out.println("accountList se  in " +
                e.getMessage());}
        }
}
```

Criar um gestor de segurança para evitar problemas com classes que são carregadas a partir de outros sites.

Associa um nome a uma instância a partir da qual se pode aceder a outros objetos do servidor

Continuação do exemplo (II)
Cliente

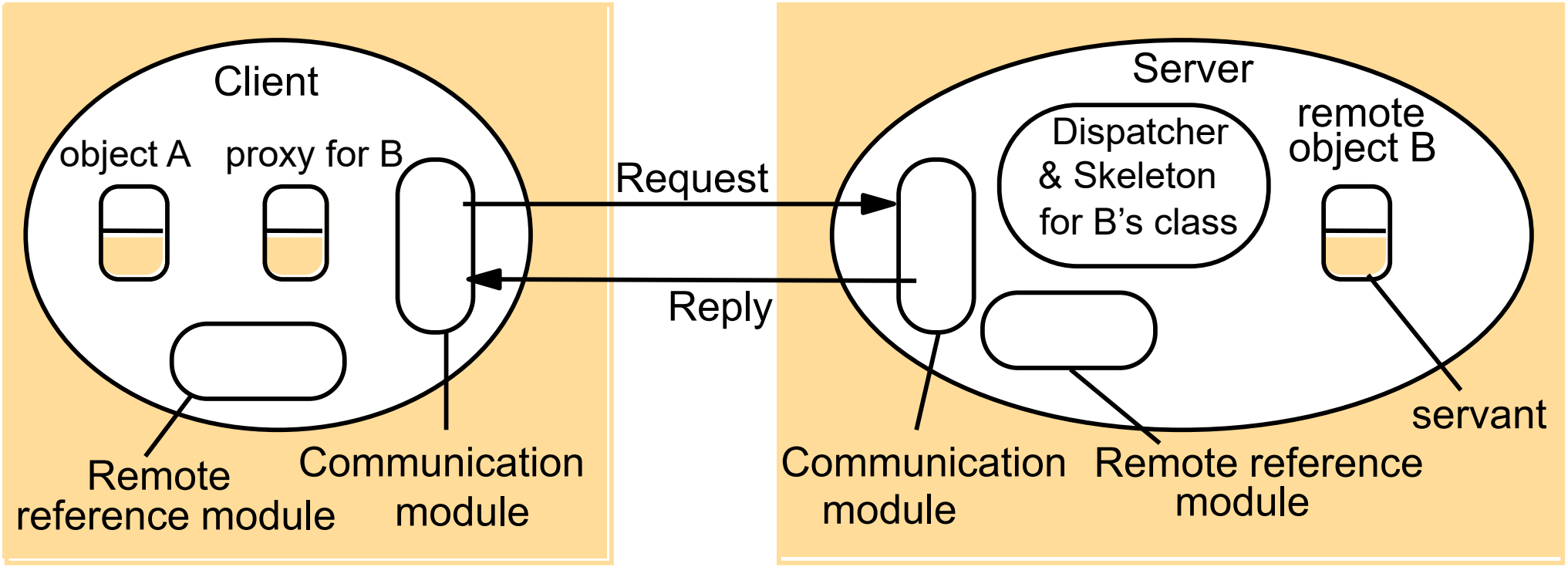
```
public static void main(String args[]){
  acList = (AccountList) Naming.lookup("//host/AccountList");
  try {
        Account a = acList.getAccount(2);
        a.debit(1000);
        Account b = new Account(1000);
        acList.addAccount(b);
  } catch(RemoteException e){ System.out.println(e.getMessage()); }
}
```

Outros aspectos fundamentais do Modelo de Objectos Distribuído

- Recuperação de memória (*garbage collection*)
  - Deve ser suportada remotamente pelo RMI
- Exceções
  - As invocações remotas podem falhar devido a exceções.
  - A maioria das linguagens Object-Oriented disponibilizam exceções que podem ser estendidas
  - Deve ser suportado na IDL

Arquitectura do RMI

Elementos principais da arquitetura do RMI



Objetos remotos / *Servants*

- Instância da classe no servidor
- Clientes que chamam métodos desse objecto podem ser remotos ou locais
- Alguns sistemas de RMI permitem que invocações remotas concorrentes corram em *threads* diferentes
  - Programador deve ter o cuidado de incluir sincronização adequada (trincos / monitores)!

Dispatcher

- Função de despacho do servidor
- Responsável por converter o *methodId* da mensagem de invocação no método do objeto local que está a ser invocado remotamente
- Existe um por cada classe invocável remotamente

Skeleton

- *Stub* do lado Servidor
- Implementa os métodos da interface remota
- Converte os parâmetros do pedido (*unmarshalling*)
- Invoca o método no objeto de destino
- Quando o método termina, converte os parâmetros de retorno e codifica eventuais exceções
- Envia a resposta

Proxy (*Stub*)

- Torna o RMI transparente para o cliente
  atuando como um objeto local
  - Esconde os detalhes de:
    - Tradução das referências para objetos
    - Conversão e serialização dos parâmetros (*marshalling*)
- A interface do *proxy* é idêntica à interface do objeto remoto
- Existe uma classe *proxy* para cada interface remota
  usada pelo cliente
- Existe um *proxy* por cada objeto remoto (noutro processo)
  que o cliente referencia

Como são gerados os *dispatchers*, *skeletons* e *proxies*?

- Gerados automaticamente por compilador de interfaces
  - Compilador extrai a informação declarada em cada interface remota
  - Muitas vezes este passo de compilação é totalmente escondido do programador

Módulo de referências remotas

- Pedidos de outros clientes trazem identificador de objeto remoto mas não podem ter a referência local
  - Como é que o módulo de comunicação do servidor sabe qual a referência local? E vice-versa?

- O módulo de referências remotas traduz entre referências remotas e locais

Módulo de referências remotas

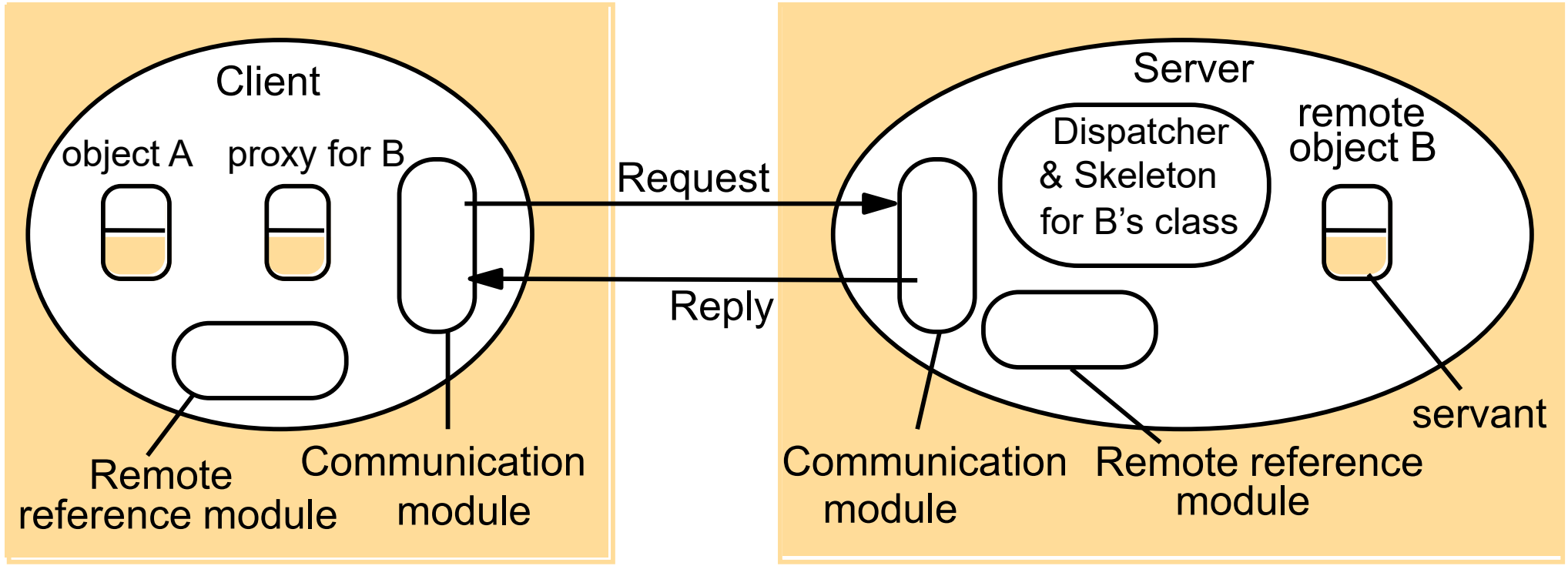
- Mantém Tabela de Objetos Remotos com entradas para:
  - Objetos remotos que residem no processo
  - *Proxies* para objetos remotos (em outros processos)
- Gere as referências remotas que saem e entram no processo:
  - Quando a referência para o objeto remoto é enviada a outro processo como parâmetro de entrada ou de saída pela primeira vez, cria uma referência remota (e adiciona-a à tabela)
  - Quando recebe uma referência remota que não existe na tabela, cria um *proxy* e acrescenta a entrada correspondente na tabela

Módulo de comunicações

- Um no cliente, outro no servidor
- Implementa protocolo de invocação remota
  - Idênticos à biblioteca de *Run-Time* do RPC
  - Asseguram semântica de invocação pretendida
    - talvez, pelo-menos-uma-vez, no-máximo-uma-vez, exatamente-uma-vez
- Módulo de comunicações do servidor seleciona o *dispatcher* da classe do objeto invocado
  - Passa-lhe a referência para o objeto local sobre o qual é feito o pedido do cliente
  - Passa também o pedido

Modelo de Objectos Distribuído

Elementos principais da arquitetura



Características avançadas

- Divulgação e descoberta de objetos
- Recuperação de memória (*garbage collection*)
- Invocação dinâmica
- Ativação dos objectos

Características avançadas

Divulgação e descoberta de objetos

Divulgação e descoberta de objectos

- O cliente precisa de obter referência remota para um primeiro objeto no servidor
  - A partir da qual poderá invocar métodos e receber outras referência remotas para outros objetos remotos
- O *binder* é um serviço distribuído que permite:
  - Servidores registarem objetos remotos neles instanciados
    - Cada objeto registado tem um nome
  - Clientes podem consultar o *binder* por nome de objeto
    - Se registado, o *binder* devolve referência remota

Características avançadas

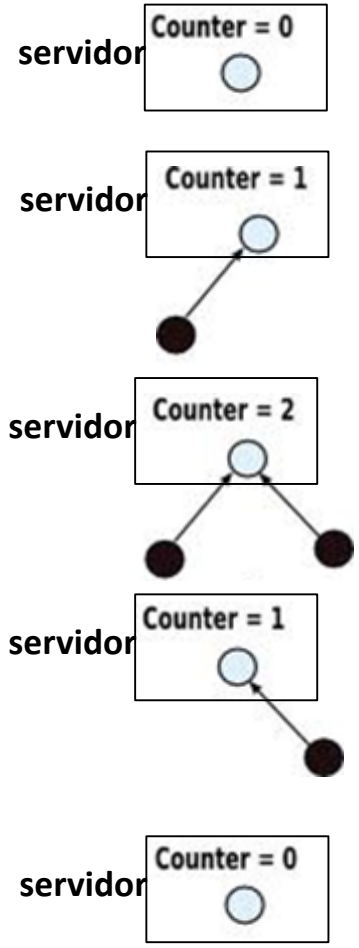
Recuperação de memória (*garbage collection*)

Recuperação de memória (*garbage collection*)

- Recuperar a memória usada por objetos remotos quando mais nenhum cliente se lhes refere
- Duas principais abordagens:
  - Contagem de referências
  - *Leases*

Recuperação de memória
Contagem de referências

- Servidor mantém estado sobre quais clientes detêm *proxy* para objeto remoto no servidor
- Cada vez que um cliente recebe uma referência a um objeto remoto faz uma chamada **addRef()** ao servidor e instancia o respetivo *proxy*
  - Servidor adiciona esse cliente à lista associada ao objeto remoto
- Quando o *garbage collector* local deteta que o proxy não pode ser mais usado, cliente faz uma chamada a **removeRef()**
  - Servidor retira esse cliente da lista associada ao objeto remoto
- Quando a contagem de referência no servidor se torna zero o servidor recupera a memória.



Recuperação de memória
Leases

- Uma *lease* é uma licença para usar o objeto durante um certo período de tempo
  - O servidor permite o acesso ao objeto remoto durante um certo período
  - Se a *lease* não for renovada quando o tempo expira a referência é automaticamente removida

- Comparando com a contagem de referências?
  - Vantagens?
  - Desvantagens?
  - Faz sentido combinar ambos?

Características avançadas

Invocação dinâmica

Invocação estática vs. dinâmica

- Normalmente, o *proxy* é estático
  - Gerado a partir de uma interface remota conhecida
  - Compilado junto com o código do cliente
- Consequências?
  - Implementação dos objetos servidores pode mudar?
    - Sim
  - Interface dos objetos remotos pode mudar?
    - Não

- E se o cliente quiser invocar métodos sobre interface remota que não era conhecida quando foi compilado?
  - Invocação dinâmica

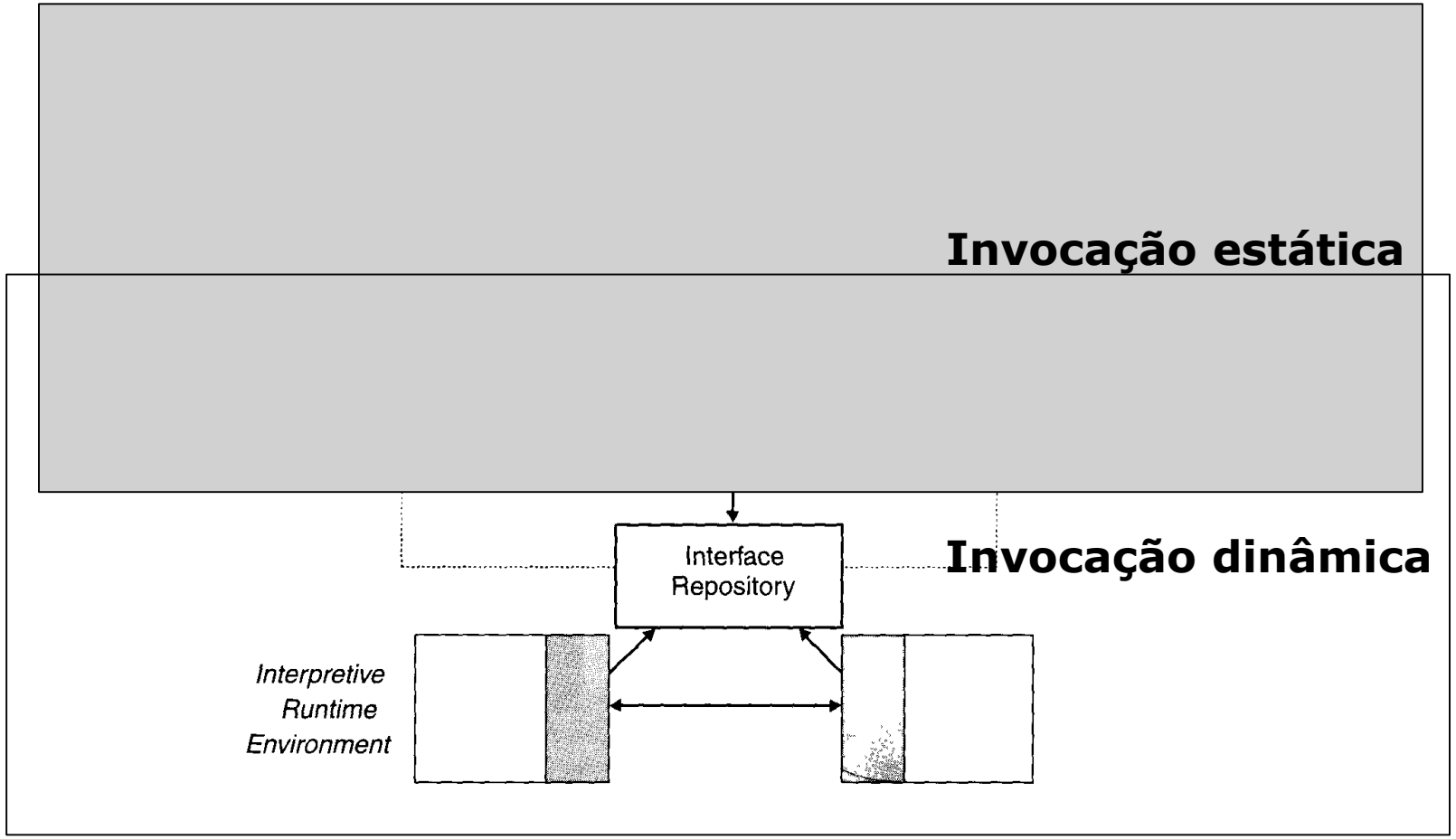
Invocação dinâmica

- Exemplo:
  - Cliente que permite jogar diferentes jogos disponíveis no servidor (jogo do galo, etc.)
  - Incluindo novos jogos que venham a surgir
    - Cuja interface é desconhecida quando o cliente é programado/compilado

Invocação dinâmica

- Paradigma poderoso mas mais difícil para o programador
  - Programa do cliente deve inspecionar a nova interface remota e descobrir:
    - Métodos disponíveis na interface
    - Parâmetros de cada método
  - Se encontrar método que quer invocar, chamar método genérico *doOperation* que recebe
    - Nome do método remoto a invocar
    - Lista de argumentos

Invocação dinâmica



Reflexão *(Reflection)*

- O paradigma de programação reflexiva baseia-se em meta-informação
  - Informação sobre a informação (sobre a estrutura do programa)
- Meta-informação:
  - Nome da classe e das suas super-classes e interfaces
  - Nome dos métodos, parâmetros e resultados
- Com esta informação, um objeto pode ser analisado para descobrir que operações é que disponibiliza

```
Method[] methods = object.getClass().getMethods();
for(Method method : methods){
    System.out.println("method = " + method.getName());
}
```

- Pode ser construída uma invocação em tempo de execução

Características avançadas

Ativação dos objectos

Ativação dos Objetos

- Os objetos no servidor podem ter um período de vida longo
- Recursos estão a ser gastos se os objetos não estiverem a ser invocados
- Leva a distinguir entre objetos:
  - Ativos: prontos para ser invocados
  - Passivos : não estão ativos mas podem ser ativados

Ativação dos Objetos

- Tornar o objecto passivo implica salvar o estado do objecto para poder ser em seguida reactivado.
  - O estado tem de ser transformado num formato coerente (serializado) e guardado persistentemente
- Tem de existir uma componente no servidor que saiba activar o objecto quando for necessário

RMI vs RPC (recapitulando)

- Reutilização de conceitos OO (Object-Oriented)
  - Melhor ligação às metodologias e linguagens OO
  - Utilização de conceitos OO na programação das interfaces
    - Separação da interface e da implementação
    - Desenvolvimento incremental
      - Herança – simplifica desenvolvimento
      - Polimorfismo – permite redefinir interfaces
- Granularidade
  - Mais fina do que em servidores que tipicamente disponibilizam interfaces para programas com alguma complexidade

RMI vs RPC (recapitulando)

- Objetos passados por valor ou referência
  - Os objetos podem migrar ou são invocados localmente
- Invocação dinâmica
  - Sistemas de objetos → desenvolvimento incremental / *loosely coupled*
    - Permitem que um objeto possa invocar dinamicamente outros dos quais obtém a interface em tempo de execução
- Gestão do Ciclo de Vida dos objetos

Principais Sistemas de Objetos Remotos

- Corba
- RMI (Java / J2EE)
- Remoting (C# / .NET)

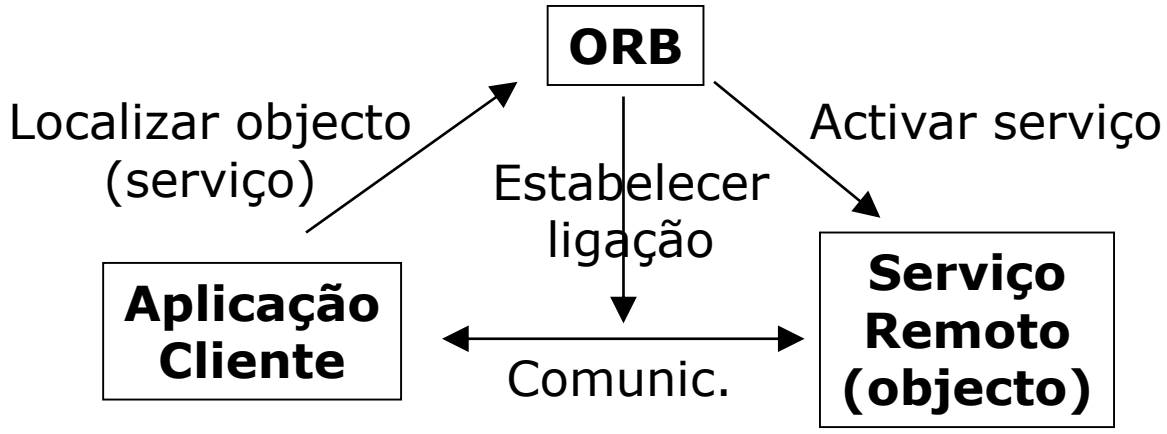
CORBA

CORBA

- Tem origem no Object Management Group (OMG) criado em 1989
- Modelo Conceptual é uma síntese entre o modelo cliente–servidor e as arquiteturas de objectos
- A proposta original do Corba - *Object Management Architecture* foi publicada em 1990 e continha:
  - Serviços de suporte ao ciclo de vida dos objetos
  - *Object request broker*

Object Request Broker

- Novo paradigma proposto consistia num serviço que auxilia a invocação de objetos remotos – (evolução do *Run-Time* do RPC)
- O papel do ORB é localizar o objeto, ativá-lo se necessário, enviar o pedido do cliente ao objeto



CORBA

- CORBA 2.0 publicado em 1996
- Os principais elementos da plataforma são:
  - IDL – linguagem *Object-Oriented* com suporte para herança
  - Arquitetura – define o ambiente de suporte aos objetos, à invocação, ao ciclo de vida
  - GIOP – *General Inter Orb Protocol* – protocolo de invocação remota entre Orb de fabricantes diferentes
  - IIOP – *Internet Inter Orb Protocol* – implementação do GIOP sobre protocolos TCP/IP
- Atualmente
  - Atividade de normalização reduzida
  - Vários produtos disponíveis
    - Visigenic/Visibroker
    - IONA
    - Menos utilizado do que as plataformas Java Enterprise Edition / .Net

Modelo de Objectos

- Um objeto CORBA implementa uma interface descrita na IDL CORBA
- Um objeto CORBA pode ser invocado remotamente através de uma referência remota
- Os objetos CORBA residem no servidor
- Os clientes podem ser objetos ou programas que enviam as mensagens corretas para os objetos
- Os objetos CORBA não têm de ser implementados numa linguagem *Object-Oriented*, podem ser em Cobol, C, etc.

Passagem de Parâmetros

- Todos os parâmetros cujo tipo é especificado pelo nome de uma interface é uma referência a um objeto CORBA e são passados como referências a objetos remotos

- Os argumentos do tipo primitivo ou estruturas são passados por valor
  - Na chegada, um novo objeto é criado no processo recetor (pode ser no cliente ou no servidor)
  - Este mecanismo é idêntico ao do RPC

CORBA IDL

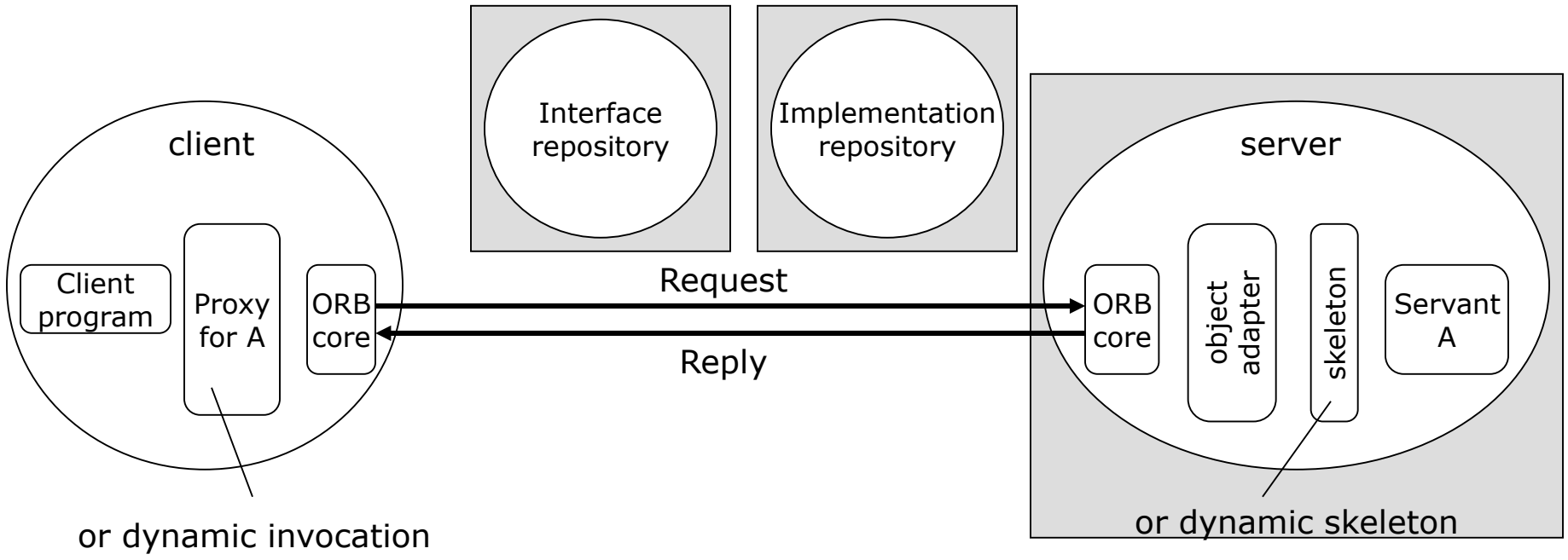
- A interface é escrita em OMG IDL
  - A interface é *Object-Oriented* com sintaxe muito semelhante ao C++
    - A herança é suportada
  - Especifica um nome de uma interface e um conjunto de métodos que os clientes podem invocar
  - Descreve os parâmetros e o respetivo sentido in, out, inout
  - Os métodos podem ter exceções

Exemplo da IDL CORBA

```
module Accounts
{
        interface Account
        {
            readonly attribute string number;
            readonly attribute float balance;

            exception InsufficientFunds (string detail);

             float debit (in float amount) raises (insufficientFunds);
             float credit (in float amount);
        }

        interface InterestAccount : Account
        {
            readonly attribute float rate;
        }
 }
```

Herança

Arquitetura



Elementos da Arquitetura

- ORB – núcleo – *Run-Time* da invocação remota, conjunto de funções residentes quer no cliente quer no servidor
  - Implementa a infraestrutura de comunicação
  - O ORB tem funções para ser inicializado e parado
- Servidor
  - *Object adapters* - rotina de despacho que recebe as mensagens e invoca os *stubs* apropriado
    - O nome do *object adapter* faz parte da referência remota e permite a sua invocação
    - Despacha cada invocação via um *skeleton* para o método apropriado
  - *Skeletons*
    - Funções de adaptação que fazem a conversão dos parâmetros de entrada e saída e o tratamento das exceções
- Client proxies
  - Para as linguagens *Object-Oriented*
  - Efetua a conversão dos parâmetros de entrada e de saída

Referências Remotas

- Para invocar uma operação remota é necessário que o objeto invocador tenha uma referência remota para o objeto
- Uma referência remota pode ser obtida como
  - Resultados de invocações remotas anteriores – a servidores ou ao gestor de nomes
  - Ter sido obtida como parâmetro de um invocação a um objecto local
- É diferente de *binding handle* estático dos RPC

IOR

IDL interface type name	Protocol and address details			Object key	
interface repository identifier	IIOP	host domain name	port number	adapter name	object name

Invocação

- A invocação do método tem, por omissão, uma semântica no-máximo-uma-vez
- A heterogeneidade é resolvida com a conversão para CDR – *Common Data Representation*
  - Inclui 15 tipos básicos
  - *Receiver makes it right*

CORBA CDR

index in sequence of bytes	← 4 bytes →	notes on representation
0-3	5	*length of string*
4-7	"Smit"	*"Smith"*
8-11	"h___"	
12-15	6	*length of string*
16-19	"Lond"	*London*
20-23	"on___"	
24-27	1934	*unsigned long*

```
Struct Pessoa {
        string Nome;
        string Lugar;
        unsigned long Ano;
};
```

Representa a struct *Person* com os valores: {'Smith', 'London', 1934}

Elementos da Arquitetura: Servidor de Interfaces

- *Interface repository*
  - Dá informação sobre as interfaces registadas
  - Para uma interface pode dar a informação dos métodos e dos respetivos parâmetros
  - O compilador de IDL atribui um número único a cada tipo IDL que compila. Esta facilidade permite a invocação dinâmica em CORBA.
  - Se um cliente recebe uma referência remota para um novo objeto CORBA de que não tem um *proxy*, pode ir buscar esta informação ao *Interface repository*

Elementos da Arquitetura: servidor do código das implementações

- *Implementation repository*
  - Contém localização das implementações das classes
  - Permite ao *object adapter* carregar o código das classes para instanciar os respectivos objectos

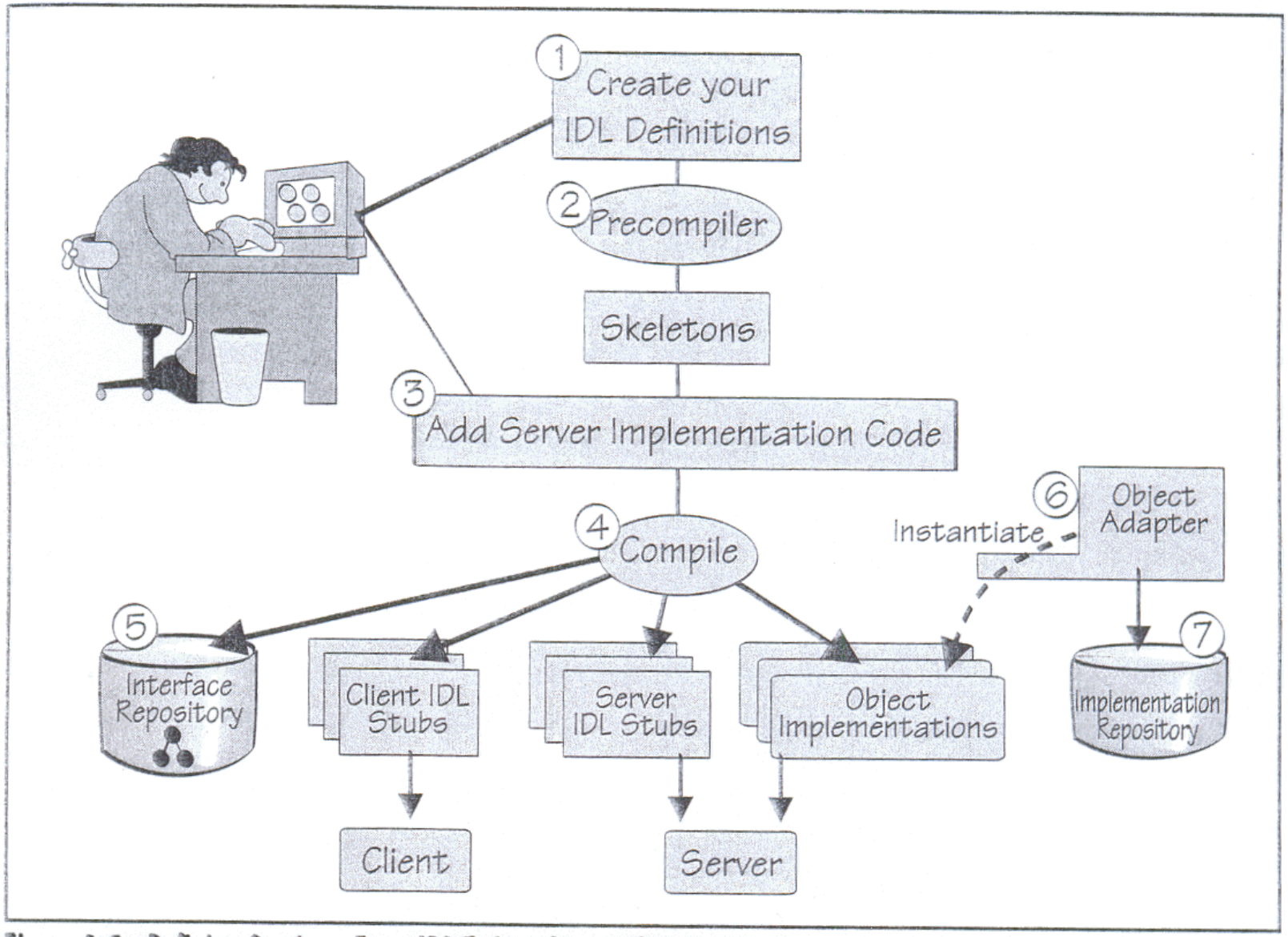


Figure 4-5. Defining Services: From IDL To Interface Stubs.

Java RMI

Java RMI

- Sistema de RMI para a linguagem Java
- Pressupõe que todos os processos no sistema correm em máquinas virtuais Java
  - Obvia muitos problemas de heterogeneidade
  - Mas não todos, pois *hardware* pode ser heterogéneo!

Interfaces remotas em Java RMI

- Interfaces remotas têm de ser uma extensão da interface **Remote**
- Todos os métodos da interface remota lançam exceção do tipo **RemoteException**

RMI Registry

- Corresponde ao *binder* em Java RMI
  - Utilizado para associar nomes a recursos e objetos de forma portável
  - Permitindo identificação, localização, partilha
- Normalmente corre um processo RMI registry em cada máquina
  - Gere os objetos remotos registados dessa máquina
- Nome é do tipo *//hostname:port/nomeObjecto*
  - Em que hostname:port se refere à máquina e ao porto onde o nome está registado

RMI Registry

- Implementa a interface JNDI (*Java Naming and Directory Interface*)

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name.
void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.
void unbind (String name, Remote obj)
This method removes a binding.
Remote lookup(String name)
This method is used by clients to look up a remote object by name. A remote object reference is returned.
String [] list()
This method returns an array of Strings containing the names bound in the registry.

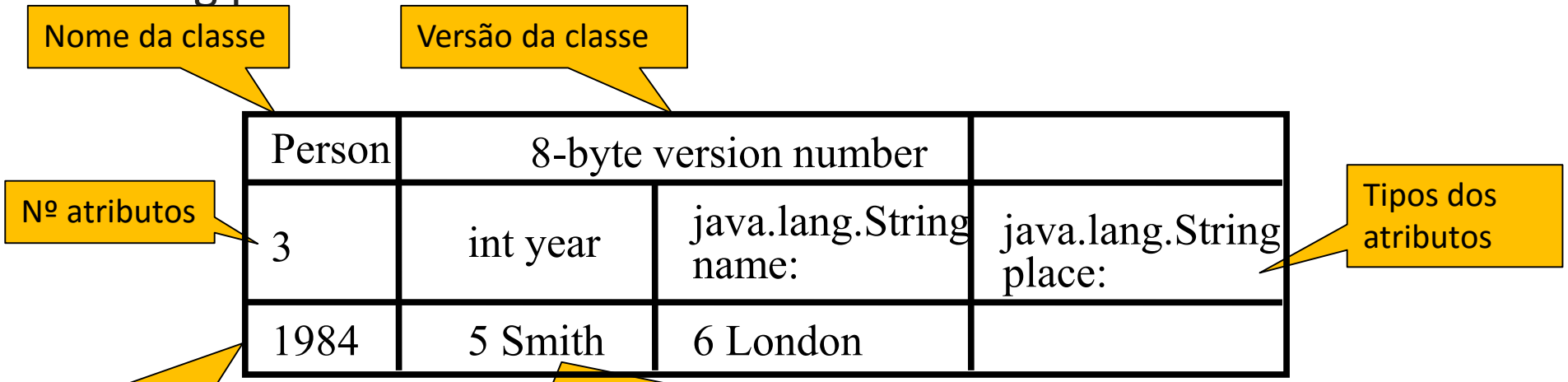
Passagem de parâmetros “por referência” versus “por valor”

Os parâmetros (argumentos ou retorno) de um método remoto são:

- **Passados por Referência** quando o parâmetro herda de java.rmi.**Remote**
  - Nesse caso diz-se que é um objeto remoto
- **Passado por Valor** caso contrário
  - Nesse caso diz-se que é um objeto local
    - Tem de implementar java.io.**Serializable**
    - Todos os tipos primitivos são serializáveis
  - Estado do objecto é serializado na mensagem de pedido (se for argumento) ou resposta (se for retorno)
    - Uma nova instância é criada remotamente
    - As cópias são independentes e as atualizações podem torná-las inconsistentes

Passagem de parâmetros “por valor”: Serialização de objeto

- Exemplo (simplificado) de serialização de instância da classe Person, cujos atributos são:
  - int year = 1984
  - String name = “Smith”
  - String place = “London”



Passagem de parâmetros “por valor”: Carregamento dinâmico de classes

- Quando há passagem por valor,
  o recetor pode não ter a classe respetiva
  - Exemplo?
- Problema: Como “des-serializar” o objeto e como criar nova instância no recetor se este não conhece a classe?
- Solução: quando um objeto é serializado, é enviado:
  - Informação sobre qual a sua classe
  - URL de servidor onde a classe pode ser descarregada dinamicamente

Serialização e variáveis estáticas (static)

- O que acontece às variáveis de classe?
  - Podem conter informação importante…
- As variáveis static não são guardadas na serialização
  - A não ser que se acrescente código à medida para o efeito
  - No contexto das invocações remotas,
    em que objetos são enviados pela rede entre clientes e servidores,
    as variáveis globais são, em geral, uma má ideia

Protocolos de invocação remota

- RMI pode ser suportado usando o *Java Remote Method Protocol* (JRMP) ou o *Internet Inter-ORB Protocol* (IIOP).
  - O JRMP é um protocolo específico criado para o RMI
  - O IIOP é um protocolo normalizado para comunicação entre objetos CORBA

- O RMI usando o IIOP permite aos objetos remotos em Java comunicarem com objetos em CORBA que podem estar programados com linguagens diferentes do Java

Exemplo em Java RMI: Interfaces remotas

```
import java.rmi*;

public interface Account extends Remote {
      float debit(float amount) throws RemoteException, InsufficientFundsException;
      float credit(float amount) throws RemoteException;
}
public interface AccountList extends Remote {
      Account getAccount(int id) throws RemoteException;
}
```

A palavra chave **Remote** indica uma classe que tem métodos invocáveis remotamente

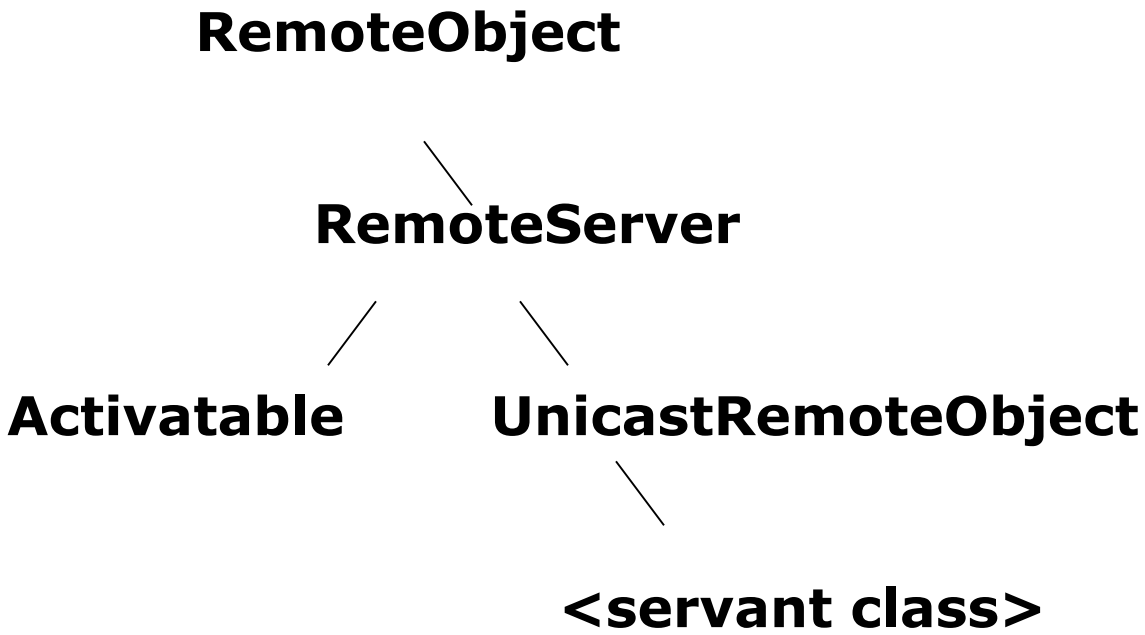
Os métodos remotos tratam uma exceção específica **RemoteException**

Classe AccountListServant implementa a interface AccountList

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
Import java.util.Vector;
public class AccountListServant extends UnicastRemoteObject implements
AccountList {
        private Vector<Account> theList; //contains the list of accounts
private
      public AccountListServant()throws RemoteException{...}
        public Account getAccount(int id) throws RemoteException {
            return theList[id];
         }
}
```

Necessário também implementar interface Account

Classes que suportam o Java RMI



Classe UnicastRemoteObject o objecto criado fica em memória
Classe Activatable pode activado quando necessário

Cliente da interface AccountList

```
import java.rmi.*;
import java.rmi.server.*;
public class AccountListClient{
      public static void main(String args[])
         System.setSecurityManager(ne      SecurityManager());
          AccountList acList =     ull;
          try{
              acList = (AccountList)Naming.lookup("//host/AccountList");
              Account a = acList.getAccount(2);
              a.debit(1000);
          }catch(RemoteException e)     ystem.out.println(e.getMessage());
        }catch(Exception e){System.out.p          ("Client:"+e.getMessage());}
      }
}
```

Obteve referência remota pelo nome da instância remota

Obteve referência remota por retorno de método

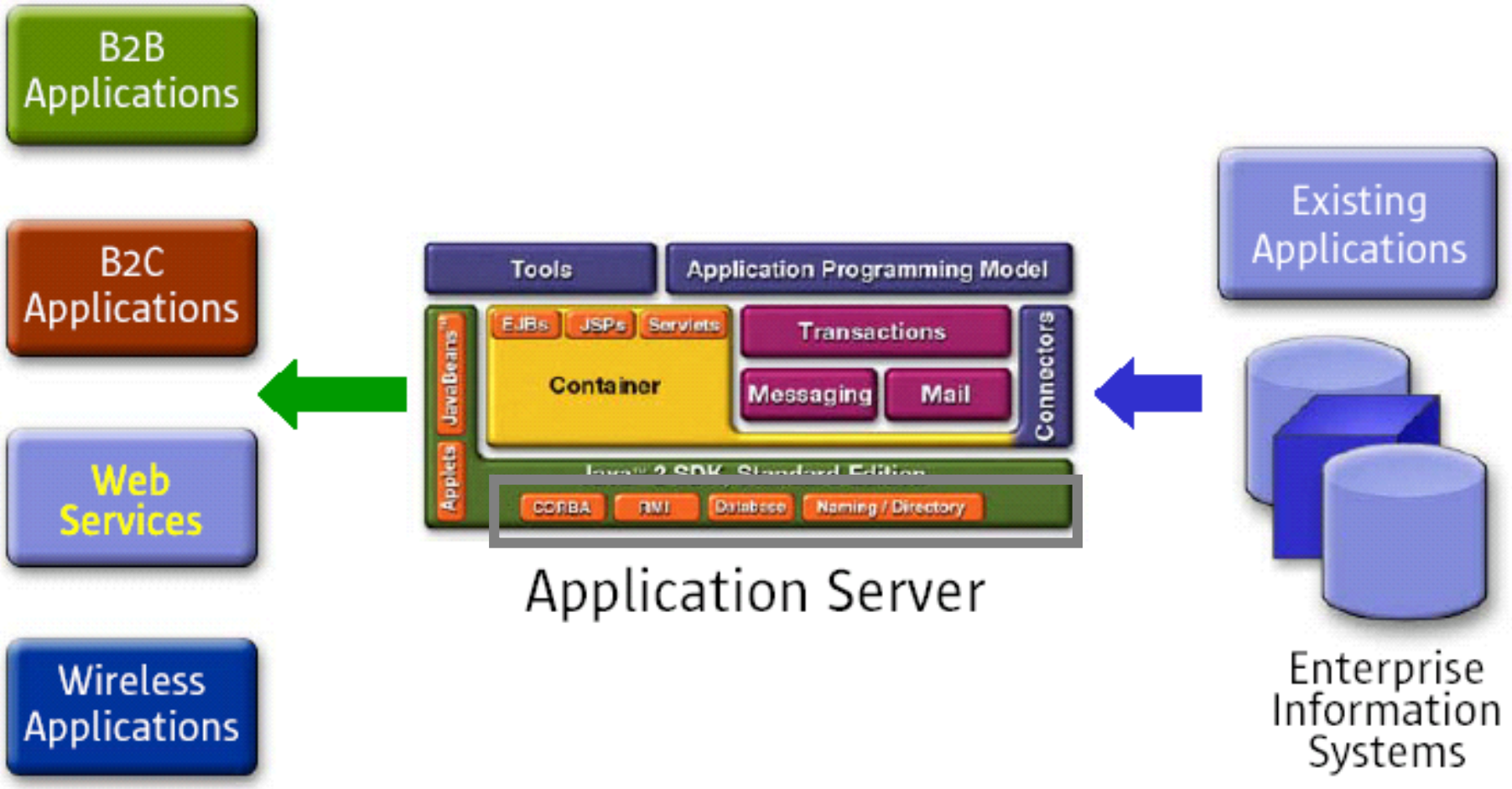
Java RMI – mecanismo de reflexão

- A reflexão é usada para passar informação nas mensagens de invocação sobre o método que se pretende executar
- API de reflexão permite, em *Run-Time*:
  - Determinar a classe de um objeto.
  - Obter informação sobre os métodos / campos de uma classe / interface
  - Criar uma instância de uma classe cujo nome é desconhecido antes da execução
  - Invocar métodos que são desconhecidos antes da execução

Reflexão no RMI – como usar

- Visão geral:
  - Obter instâncias das classes genéricas “Class” e “Method”
  - Cada instância de Method representa as características de um método (classe, tipo dos argumentos, valor de retorno, e exceções)
  - Invocar métodos remotos com o método “invoke” da classe “Method”.
    Requer dois parâmetros:
    - O primeiro é o objeto a invocar
    - O segundo é um vetor de Object contendo os argumentos
- Mais detalhes em http://java.sun.com/docs/books/tutorial/reflect/

Arquitetura J2EE – Java Enterprise Edition



Equivalente ao RMI para C# na plataforma .NET.

Vamo-nos concentrar nas novas opções fornecidas

.NET Remoting

Objetos Remotos

- Qualquer objeto pode ser usado remotamente derivando-o de **`MarshalByRefObject`** ou de **`MarshalByValObject`**
- MarshallByValue
  - Objetos serializáveis, são copiados para o cliente
  - Pode ser muito ineficiente
- MarshallByRef
  - Quando o cliente recebe uma referência de um objeto remoto é uma referência a um objeto *proxy* local criado automaticamente pelo .NET Remoting.

Utilização de Objectos Remotos

- Requer a respetiva ativação, que tem dois modos:
- Singleton
  - Apenas uma instância em cada instante
  - Criada aquando da primeira invocação
  - Requer sincronização no acesso a estado partilhado
- SingleCall
  - Uma nova instância é criada para cada pedido.
  - Após a execução de uma chamada, a próxima chamada será servida por outra instância

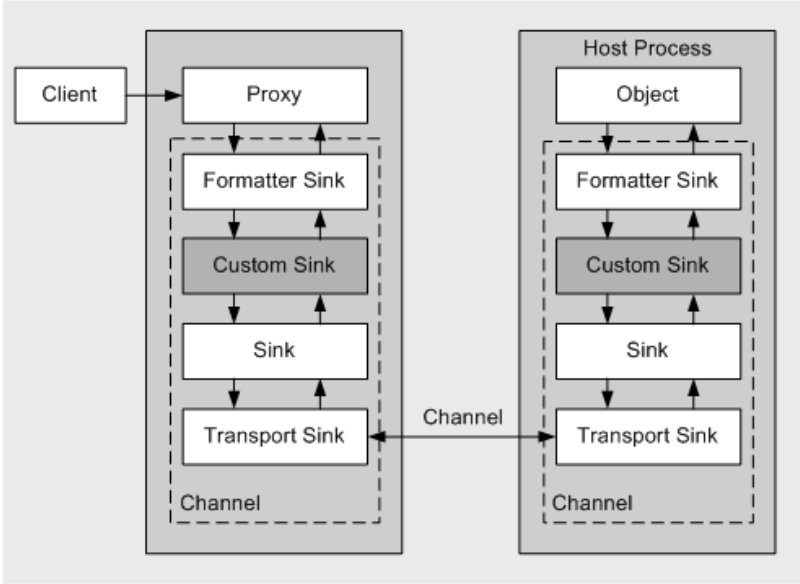
Tempo de Vida dos Objectos

- Tempo de vida dos objectos Singleton determinado por sistema de *leases*
  - Ao expirar um *lease*, este deve ser renovado, caso contrário a memória ocupada pelo objecto é recuperada pelo *garbage collector*

Java RMI escolheu solução mais complexa: contagem de referência distribuída.
Problema: falhas na rede e nos servidores?
Java RMI usa adicionalmente *leases* para tolerar falhas

Canais

- A comunicação entre dois processos distintos é realizada através de canais, cuja função é:
  - Empacotar a informação de acordo com um tipo de protocolo
  - Enviar esse pacote a outro computador.
- Dois tipos pré-definidos:
  - **TcpChannel**
    (que envia os dados por TCP em formato binário)
  - **HttpChannel**
    (que utiliza o protocolo HTTP em formato XML)
- Maior flexibilidade do que no RMI na escolha dos mecanismos de serialização e
  protocolos de transmissão



Resumo das Opções

- MarshallByRef vs. MarshallByValue
- Canais – TcpChannel vs. HttpChannel vs. Custom
- Activação – Singleton vs. Single Call
- Tempo de vida – Single Call vs. Leases

Exemplo: Interface

```
public interface Account {
    float debit(float amount);
    float credit(float amount);
}

public interface AccountList {
      Account getAccount(int id);
}
```

Exemplo: Servidor

```
class bankServer : MarshalByRefObject, accountList {
   ArrayList acList;
   public bankServer () {...}

   public Account getAccount(int id) {
         return acList[id];
   }

   static void Main() {
         TcpChannel chan1 = new TcpChannel(8086);
         ChannelServices.RegisterChannel(chan1);
         RemotingConfiguration.RegisterWellKnownServiceType(typeof(bankServer),
                              "accountList", WellKnownObjectMode.Singleton);
         System.Console.WriteLine("<enter> para sair...");
         System.Console.ReadLine();
   }
}
```

Exemplo: Cliente

```
class cl {
  static void Main() {
    TcpChannel chan = new TcpChannel();
    ChannelServices.RegisterChannel(chan);
    accountList acList =
        (accountList) Activator.GetObject(typeof(accountList),
                "tcp://localhost:8086/accountList");
    if (acList == null)
      System.Console.WriteLine("Could not locate server");
    else {
        Account a = acList.getAccount(2);
        Console.WriteLine(a.debit(1000));
  }
}
```