Slides do Capítulo 5
Invocação Remota

De: ***COULOURIS, G., DOLLIMORE, J. e KINDBERG, T., Sistemas Distribuídos: Conceitos e Projetos, Bookman Companhia Editora, 5a Edição***

Agenda

- Comunicação entre objetos distribuídos
- Chamada de procedimento remoto
- Eventos e notificações

Camadas do *Middleware*

Relembrar: Interfaces

- Regulam o acesso às características externamente visíveis de um objeto ou módulo
  - em geral: métodos e variáveis
- Papel fundamental no encapsulamento
- Em sistemas distribuídos:
  - apenas **métodos** são acessíveis através de interfaces
    - acesso de variáveis via métodos *getters* e *setters*
- Passagem de parâmetros
  - parâmetros de entrada e saída
    - Entrada – argumentos da mensagem de requisição
    - Saída – argumentos da mensagem de resposta
  - ponteiros não são permitidos
  - objetos como parâmetros: referência de objeto

Relembrar: Linguagens de Definição de Interfaces (IDL)

- Sintaxe (e semântica associada) para a
  - definição de:
    - operações: nome, parâmetros e valor de retorno
    - exceções
    - atributos
    - tipos primitivos e construídos (para os parâmetros e valores de retorno)
  - Exemplos:
    - CORBA IDL
    - DCOM IDL (Microsoft IDL)
    - Arquivos .proto

Relembrar: Linguagens de Definição de Interfaces (IDL)

- Interfaces de Serviço
  - Modelo Cliente-Servidor
  - Define o conjunto de procedimentos disponíveis e seus argumentos
- Interface Remota
  - Modelo de objeto distribuído
  - Define os Métodos de um objeto que estão disponíveis para invocação remota juntamente com seus argumentos de entrada e saída
- Diferença?
  - Métodos da interface remota podem passar e retornar objetos, bem como passar referências
- Ambas
  - Não podem fornecer acesso direto a variáveis

Relembrar: Exemplo de definição em CORBA IDL

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

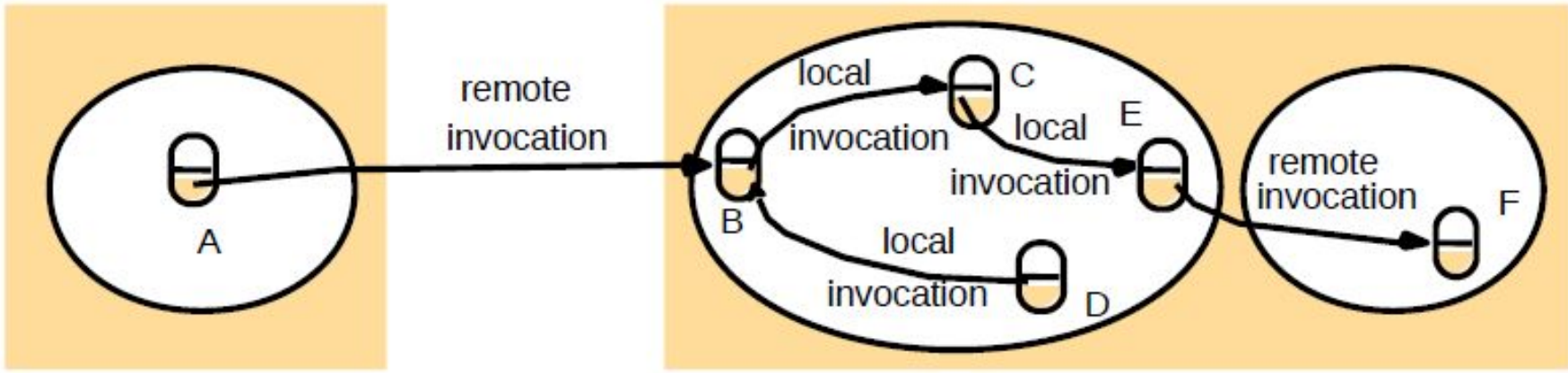
Modelo de objetos básico

- Referências de objetos
- Interfaces
- Ações
- Exceções
- Coleta de lixo

Modelo de objetos distribuídos

- Modelo de objetos básico
- Conceitos de objetos distribuídos
- Extensão do modelo de objetos convencional
- Questões de projeto
- Implementação
- Coleta de lixo distribuída

Chamadas locais e remotas

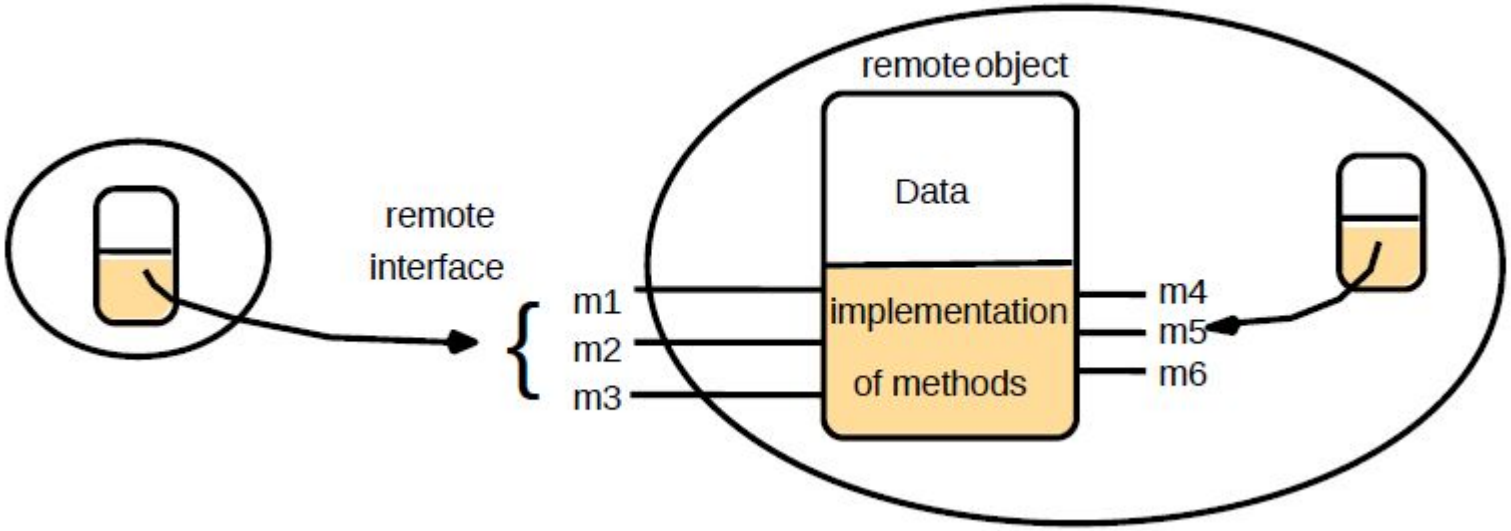


- Invocações Locais vs Invocações Remotas (Processos Diferentes)
- Referência de objeto remota
- Interface remota

O modelo de objetos distribuídos: uma extensão ao modelo de objetos básico

- Chamadas de métodos remotos (RMI)
- Referência de objeto remoto
  - funcionalmente semelhante a referências locais
  - estruturalmente diferente: identificador válido em todo o sistema distribuído
  - podem ser passadas como argumentos e retornadas como resultado
- Interface remota
  - define os métodos remotamente acessíveis
  - geralmente independente da linguagem de programação.

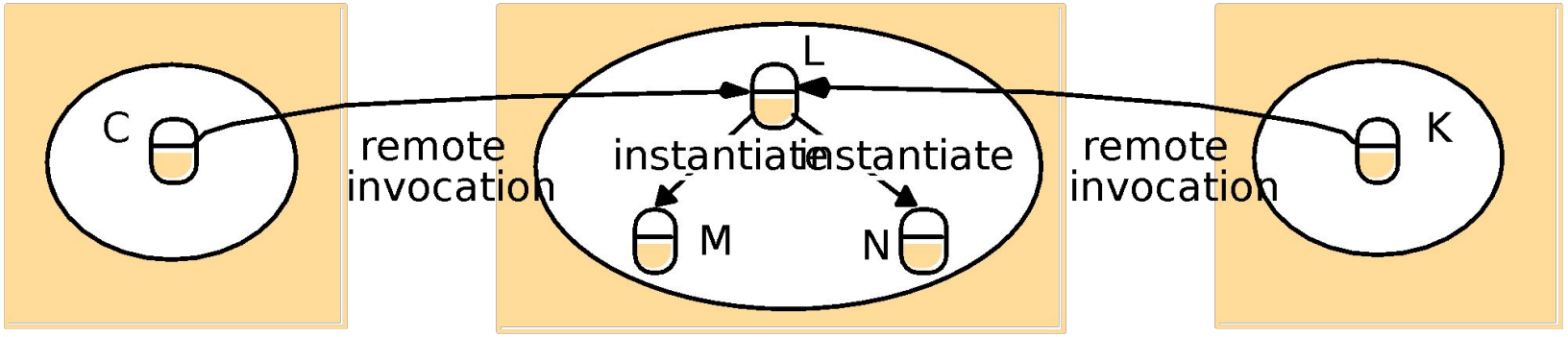
Um objeto com interfaces local e remota



O modelo de objetos distribuídos: uma extensão ao modelo de objetos básico

- Exemplos de ações:
  - requisição para executar alguma operação num objeto remoto
  - obtenção de referências de objetos remotos
  - instanciação de objetos remotos
- Coleta de lixo distribuída
  - requer contagem de referências explícita
  - rastreamento de todas as referências trocadas
  - pouco eficiente ou difícil de ser implementada

Instanciação de objetos remotos



O modelo de objetos distribuídos: uma extensão ao modelo de objetos básico

- Exceções
  - erros de aplicação: gerados pela lógica do servidor
  - erros de sistema: gerados pelo *middleware*
  - Erros e exceções são conduzidos de volta ao cliente sob a forma de mensagens

Questões de projeto para RMI

- Semântica de chamadas
  - talvez executada (*maybe, best-effort)*
  - executada pelo menos uma vez (*at least once)*
  - executada no máximo uma vez (*at most once)*
- Transparência
  - ideal, mas não 100% prática
    - falhas parciais
    - latência

Talvez

Semântica de chamadas em RMI

Modelo de tolerância a falhas			Semântica de Invocação
Reenvio de mensagem de requisição	*Filtragem de duplicadas*	*Reexecução de procedimento ou retransmissão de resposta*	
Não	Não aplicável	Não aplicável	*Talvez*
Sim	Não	Reexecuta o procedimento	*Pelo menos Uma vez*
Sim	Sim	Retransmite a reposta	*No máximo Uma vez*

Semântica **talvez**

- Nenhuma das medidas de tolerância a falhas é implementada.
- Pode ser executado uma vez ou não ser executado
- Falhas por omissão (mensagem de requisição ou resposta) ou colapso
- Útil para aplicações em que invocações mal-sucedidas ocasionais são aceitáveis
- CORBA - para métodos que não retornam valor (void)

Semântica **pelo menos uma vez**

- O invocador recebe um resultado quando o método foi executado pelo menos uma vez, ou recebe uma exceção, informando-o que nenhum resultado foi obtido
- Mascara falhas de omissão através de retransmissão de requisições
- Falhas de colapso.
- Falhas arbitrárias: executa o método mais de uma vez devido a mensagens duplicadas
  - Aceitável quando as operações forem idempotentes
- **(RPC da SUN): década de 80**

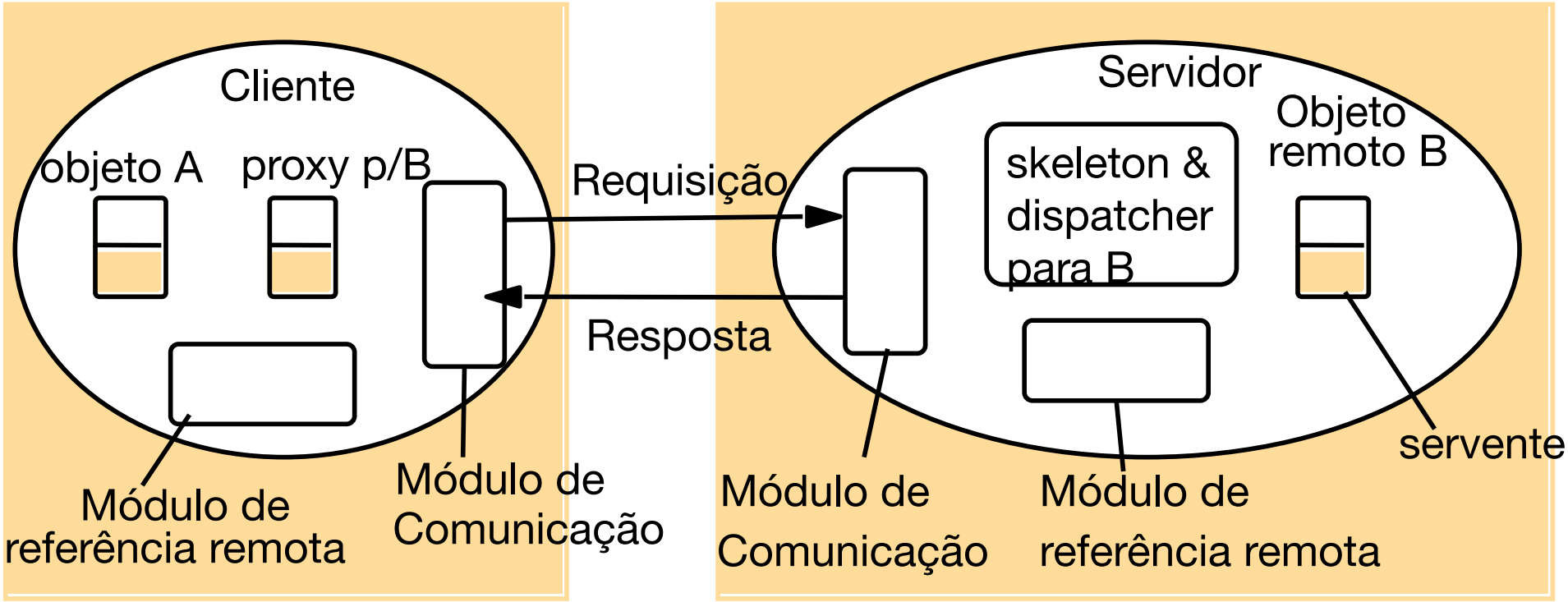
Semântica **no máximo uma vez**

- Ou o executor recebe um resultado quando o método for executado exatamente um vez, ou em caso contrário, uma exceção.
- **CORBA RMI e Java RMI: década de 90**

Transparência

- Invocações remotas devem ser transparentes
  - Sintaxe igual à invocação local
  - Empacotamento e troca de mensagens ocultas ao programador
- Diferenças entre objetos locais e remotos expressa em suas interfaces
  - Java RMI
    - Objetos remotos implementam a interface Remote
  - Invocações remotas são mais suscetíveis a falhas
    - Os invocadores devem saber diferenciar a invocação remota da local para tratarem as falhas de forma consistente

Implementação de RMI



Módulo de comunicação

- Cuida do protocolo de comunicação de mensagens entre cliente e servidor
- Implementa o protocolo requisição-e-resposta
- Define os tipos de mensagens utilizados
  - tipo de mensagem
  - requestID
  - referência remota do objeto
- Provê a identificação das requisições
- Semântica de chamadas (ex.: *at-most-once)*
  - Retransmissão e eliminação de duplicatas

Módulo de referências remotas

- Gerencia referências de objetos remotos
- No lado servidor:
  - tabela com as referências a objetos remotos que residem no processo local
  - ponteiro para o *skeleton* correspondente
- No lado cliente:
  - tabela com as referências a objetos remotos que residem em outros processos e que são utilizadas por clientes locais
  - ponteiro para o *proxy* local para o objeto remoto

Serventes

- Instância de uma implementação de objeto
- Hospedados em *processos servidores*
- Trata as requisições remotas repassada pelo *Skeleton*

Proxy (ou Stub)

- Objeto local que representa o objeto remoto para o cliente
- “Implementa” os métodos definidos na interface do objeto remoto
- Cada implementação de método no *proxy:*
  - *marshalling de requisições*
  - *unmarshalling de respostas*
- Torna a localização e o acesso ao objeto remoto transparentes para o cliente

Despachante

- Recebe requisições do módulo de comunicações e as repassa para o *skeleton*

Skeleton

- Implementa os métodos da interface remota
- Implementa os mecanismos de tratamento de requisições recebidas e repasse das mesmas (na forma de chamadas locais) ao objeto alvo:
  - *Unmarshalling de requisições*
  - *Marshalling de respostas*
- Específico para cada tipo de objeto remoto

Compilador de IDL

- Geração de *proxy e skeleton (e despachante) a* partir de uma definição de interface
- Segue o mapeamento da IDL para a linguagem de implementação do cliente e do servidor

Agenda

- Comunicação entre objetos distribuídos
- Chamada de procedimento remoto
- Eventos e notificações
- RMI Java

Java RMI

- *Remote Method Invocation: mecanismo de* chamada remota a métodos Java
  - Mantém a semântica de uma chamada local, para objetos remotos
  - Efetua automaticamente a **serialização** e **desserialização** dos parâmetros
  - Envia as **mensagens** de requisição e resposta
  - Faz o **agulhamento** para encontrar o objeto e o método pretendido

Java RMI

- Apesar do ambiente uniforme, um objeto tem conhecimento que invoca um método remoto porque tem de tratar ***RemoteException***
- A interface do objeto remoto por sua vez tem de ser uma extensão da interface ***Remote***

Java RMI

- No Java RMI, assume-se que os parâmetros de um método são parâmetros de **entradas** (*input*) e o resultado de um método é um único parâmetro de **saída** (*output*)
- Quando o parâmetro é um objeto remoto (herda de *java.rmi.Remote):* É sempre passado como uma referência para um objeto remoto
- Quando o parâmetro é um objeto local (caso contrário)
  - É **serializado** e passado por valor. Quando um objeto é passado por valor, uma nova instância é criada remotamente
  - Tem de implementar *java.io.Serializable*
  - Todos os tipos primitivos e objetos remotos são serializáveis. *(java.rmi.Remote descende de java.io.Serializable)*
- As classes de argumentos e valores de resultados são carregadas por ***download*** quando necessário

Exemplo de Serialização em Java

- Definição de um tipo *Pessoa em Java:*

```
public class Pessoa implements Serializable {
      private String nome; private String lugar; private int ano;
      public Pessoa(String nome, String lugar, int ano) {
      this.nome = nome; this.lugar = lugar; this.ano = ano;
      } // continua com a definição dos métodos...
}
```

- Exemplo (simplificado) de serialização de um objeto do tipo

Pessoa: Pessoa p = new Pessoa("Smith", "London", 1934);

Valores serializados				Comentário
Pessoa	No. de versão (8 bytes)		h0	nome da classe, número de versão
3	int ano	java.lang.String nome:	java.lang.String lugar:	número, tipo e nome das variáveis de instância
1934	5 Smith	6 London	h1	valores das variáveis de instância

- Download de classes
  - Classes são carregadas por *download* entre as JVMs
  - Se o destino ainda não possuir a classe de objeto passado **por valor,** seu código será carregado por *download* automaticamente
  - A mesma estratégia é utilizada para classes *Proxy*
  - Vantagens:
    - Não há necessidade de cada usuário manter o mesmo conjunto de classes em seu ambiente de trabalho
    - Os programas cliente e servidores podem fazer uso transparente de instâncias de novas classes, quando elas forem adicionadas

Funções do registry

- void rebind (String name, Remote obj)
  - Usado pelos servidores para **registar** a associação entre o objeto e o seu nome.
- void bind (String name, Remote obj)
  - Igual ao anterior, mas lança excepção se já existe a associação.
- void unbind (String name, remote obj)
  - Retira uma associação existente.
- Remote lookup (String name)
  - Usado pelos clientes para **localizar um objeto remoto** pelo seu nome. Retorna a referência para o objeto remoto.
- String [ ] list ( )
  - Retorna um vetor de Strings com os nomes presentes no **registry.**

Figure 5.12
Java Remote interface

```
package exemplo.ufc.br;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Calc extends Remote {
    String sayHello() throws RemoteException;
    double soma(double n1, double n2) throws RemoteException;
}
```

Os métodos sayHello e soma são expostos para serem executados remotamente

Figure 5.14
Java class *Server* with *main* method

```
package exemplo.ufc.br;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Calc {
    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public double soma(double n1, double n2) throws RemoteException {
        return n1 + n2;
    }
```

Implementa os métodos definidos na Interface Calc

Figure 5.14
Java class *Server* with *main* method (cont..)

```
public static void main(String args[]) {
        try {
            Server obj = new Server();
            Calc stub = (Calc) UnicastRemoteObject.exportObject(obj, 0);

            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Calc", stub);

            System.err.println("Servidor pronto");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Cria o servente e registra o objeto remote com o nome Calc

Figure 5.16
Java client

```
package exemplo.ufc.br;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry(“localhost”);
            Calc stub = (Calc) registry.lookup("Calc");
            String response = stub.sayHello();
            System.out.println("Resposta: " + response);
            double resultado = stub.soma(20, 35);
            System.out.println("Soma 20 + 35 = " + resultado);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
        }
    }
}
```

Cria o cliente, que busca pelo objeto remote cujo nome é Calc

Java RMI – executando o exemplo -

- Execute o RMIRegistry
  - rmiregistry
- Execute o Servidor
  - java -Djava.server.rmi.codebaseile:///rmi/ -Djava.security.policy=policy rmi.ShapeListServer
- Execute o Cliente
  - java -Djava.security.policy=policy rmi.ShapeListClient

Arquivo “policy”

```
grant{
    permission java.security.AllPermission;
};
```