UNIVERSIDADE FEDERAL DO CEARÁ - Campus Quixadá
Cursos: SI, ES, RC, CC e EC
Código: QXD0043
Disciplina: Sistemas Distribuídos

Capítulo 5 – Invocação Remota

Prof. Rafael Braga

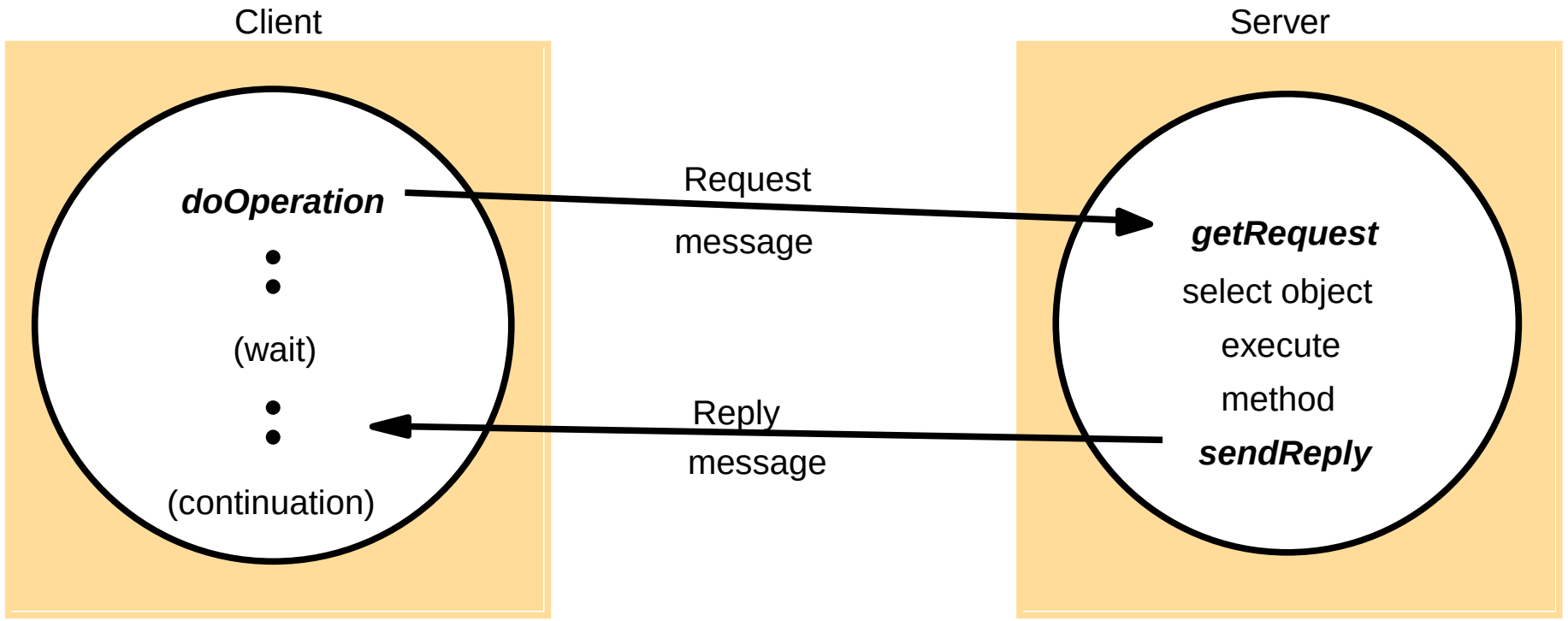
Agenda

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

Comunicação Cliente-Servidor

- ***Modelo geral: requisição-e-resposta***
- ***Variantes:***
  - síncrona: cliente bloqueia até receber a resposta
  - assíncrona: cliente recupera (explicitamente) a resposta em um instante posterior
    - não-bloqueante
- ***Implementação sobre protocolo baseado em datagramas é mais eficiente***
  - evita: reconhecimentos redundantes, mensagens de estabelecimento de conexão, controle de fluxo

Protocolo requisição-e-resposta



Operações utilizadas para implementar o protocolo de requisição-e-resposta

- ***public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)***

envia uma mensagem de requisição para o objeto remoto e retorna a resposta.

Os argumentos especificam o objeto remoto, o método a ser chamado e os argumentos para aquele método.

- ***public byte[] getRequest ();***

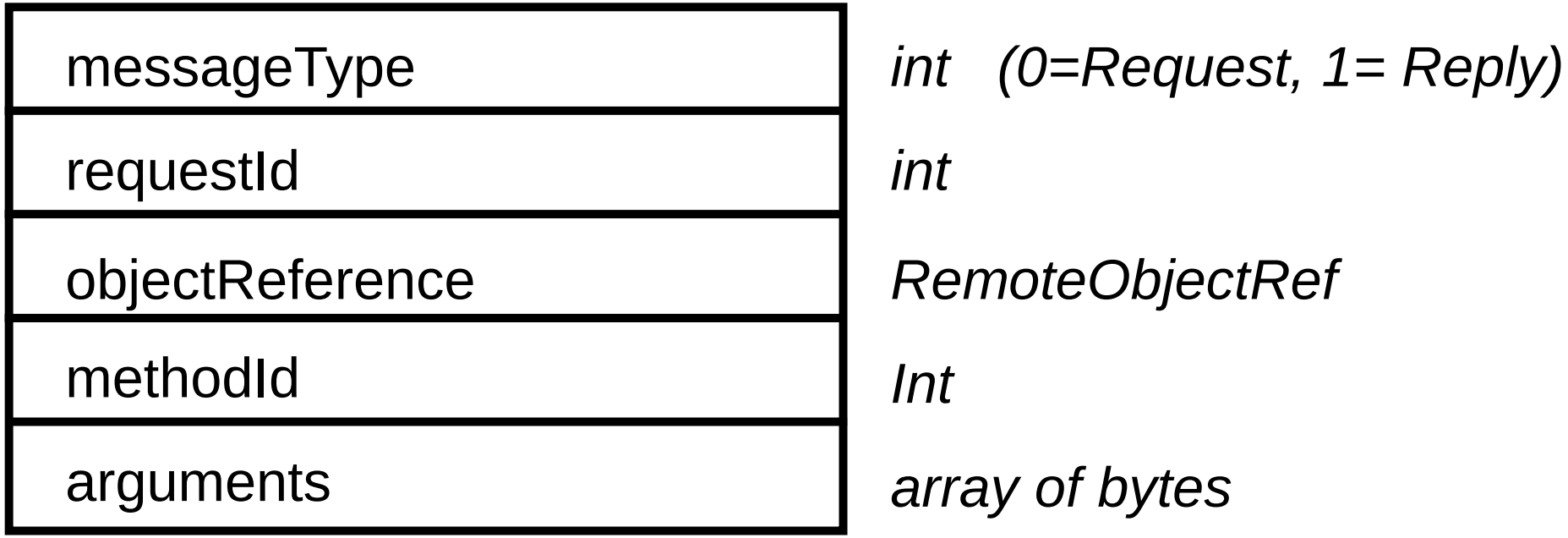
obtém uma requisição de um cliente através de uma porta servidora.

- ***public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);***

envia a mensagem de resposta para o cliente, endereçando-a a seu endereço IP e porta.

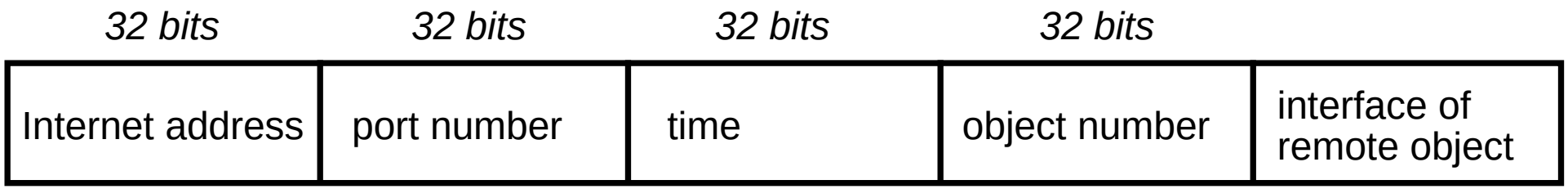
Estrutura de mensagens de requisição-e-resposta

- Identificador da mensagem:
  - request ID + ID do processo que fez a requisição



Representação de Referências de Objetos

- Necessária quando objetos remotos são passados como parâmetro
- A referência é serializada (não o objeto)
- Contém toda informação necessária para identificar (e endereçar) um objeto unicamente no sistema distribuído



Modelo de falhas

- ***Suposições***
  - processos: colapso (falham e permanecem parados)
  - canais: podem omitir mensagens, mensagens não chegam em ordem
- ***Timeouts: em doOperation***
  - detectar que a resposta (ou a requisição) foi perdida
- ***alternativas de tratamento:***
  - sinalizar a falha para o cliente (uma boa opção?)
  - repetir o envio da requisição (um certo número de vezes)
    - até ter certeza sobre a natureza da falha (quem falhou? O canal ou o processo?)

Modelo de falhas (2)

- ***Descarte de mensagens duplicadas***
  - o servidor deve distinguir requisições novas de requisições retransmitidas
    - apropriado re-executar a requisição?
- ***Em caso de perda da resposta***
  - servidor re-executa a requisição para gerar novamente a resposta?
  - servidor mantém um histórico das respostas geradas para o caso de precisar retransmitir?
    - o histórico precisa conter apenas a última resposta enviada para cada cliente. Por que? Suficiente?

Protocolos de troca de mensagens em RPC

- Tipo de garantia (confiabilidade) em cada caso?

Nome	***Mensagens enviadas pelo***		
	Cliente	***Servidor***	***Cliente***
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Protocolo de transporte utilizado: TCP

- ***não impõem limites no tamanho das mensagens***
  - na verdade, cuida da segmentação de forma transparente
  - permite argumentos de tamanho arbitrário (ex.: listas)
- ***transferência confiável***
  - lida com mensagens perdidas e duplicadas
    - Elimina necessidade de retransmissões e histórico
- ***controle de fluxo***
- ***overhead reduzido em sessões longas (muitas requisições e respostas sucessivas)***

Protocolo de transporte utilizado: UDP

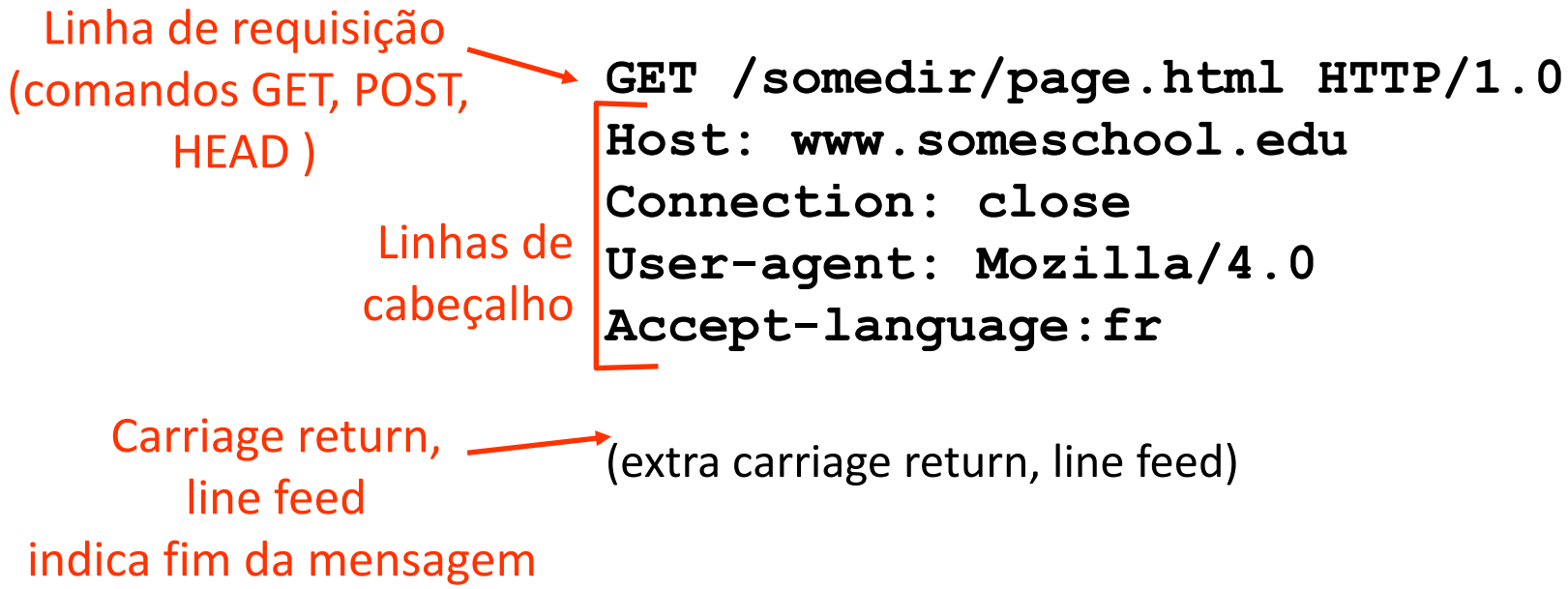
- ***se mensagens têm tamanho fixo***
- ***se sessões são curtas (apenas uma requisição e sua resposta)***
  - não compensa o overhead do handshake TCP
- ***se operações são idemponentes***
  - mensagens duplicadas (retransmissões) podem ser re-processadas sem problema
- ***otimização do protocolo de confiabilidade para casos específicos***

HTTP como protocolo de requisição-e-resposta

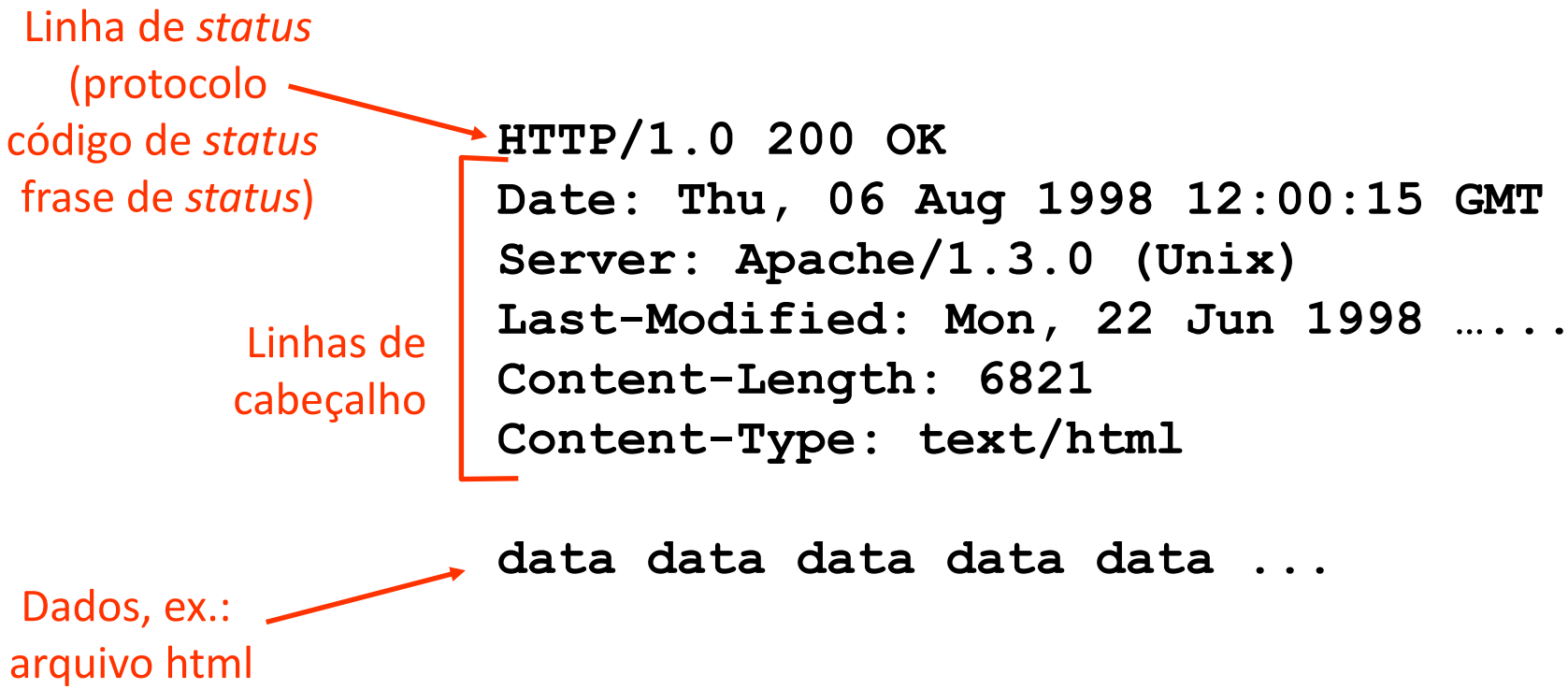
- ***Tipo RR***
- ***Regras para o formato de mensagens***
  - Marshalling (empacotamento)
- ***Conjunto fixo de métodos: PUT, GET, POST,...***
  - Aplicáveis a todos os seus recursos
- ***Permite negociar o tipo do conteúdo***
  - Multipurpose Internet Mail Extensions ***(MIME)***
- ***Conexões persistentes***
  - Para amortizar o custo de abrir e fechar conexões TCP
- ***Dois tipos de mensagens HTTP***
  - request, response

Mensagem HTTP request

- HTTP request message: ASCII (formato legível)



Mensagem HTTP response



Códigos de status das respostas

Na primeira linha da mensagem de resposta servidor cliente.

Alguns exemplos de códigos:

200 OK

- Requisição bem-sucedida, objeto requisitado a seguir nesta mensagem

301 Moved permanently

- Objeto requisitado foi movido, nova localização especificada a seguir nesta mensagem (Location:)

400 Bad request

- Mensagem de requisição não compreendida pelo servidor

404 Not Found

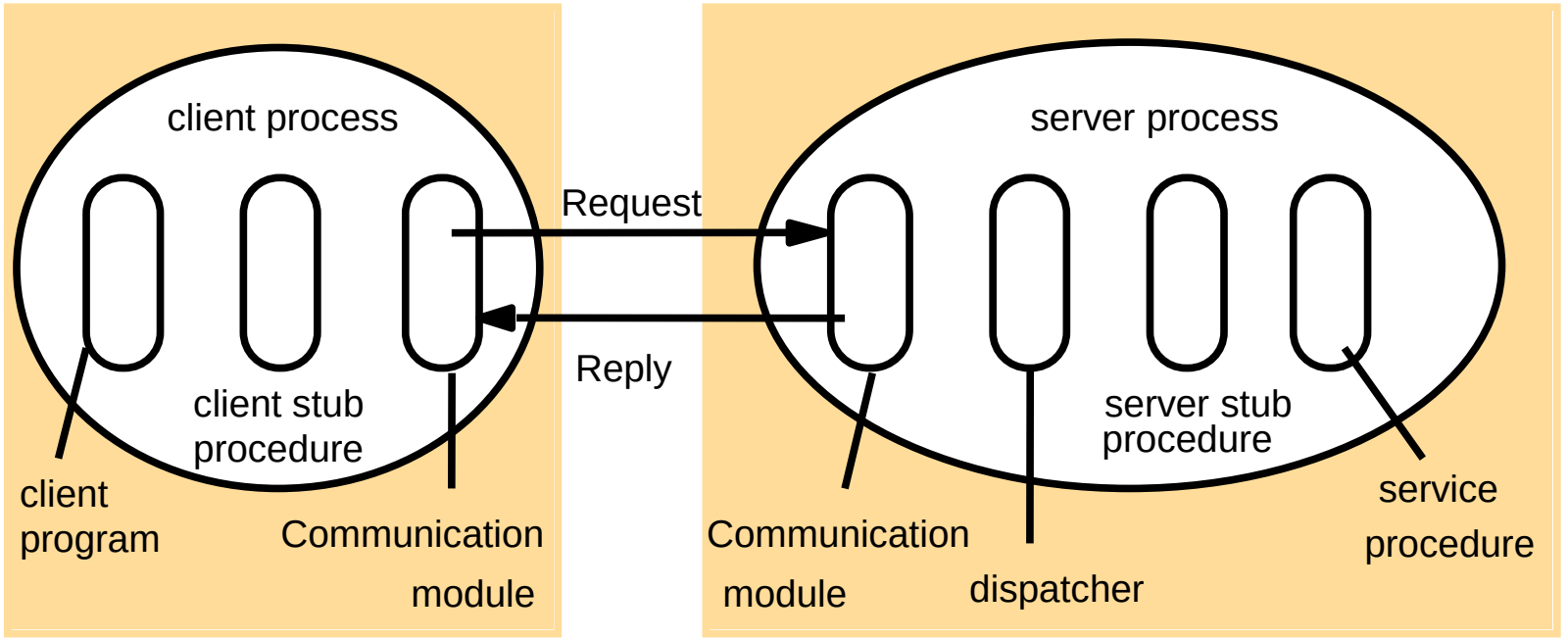
- Documento requisitado não encontrado neste servidor

505 HTTP version not supported

Agenda

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

Implementação de RPC



Implementação

- Procedimento *stub* de cliente
  - Um procedimento *stub* de cliente para cada procedimento da *interface de serviço*.
  - Semelhante ao *proxy*, se comporta como um procedimento local no cliente.
  - Empacota o identificador de procedimento e os argumentos em uma mensagem de requisição
  - Quando uma mensagem chega, ele desempacota o resultado e o retorna para o chamador

Implementação

- Despachante
  - Seleciona um dos procedimentos *stub* de servidor, de acordo com o identificador de procedimento presente na mensagem de requisição.
- Procedimento *stub* de servidor
  - Semelhante ao *Skeleton,* desempacota os argumentos presentes na mensagem de requisição, chama o procedimento de serviço correspondente e empacota os valores de retorno
- Procedimento de Serviço
  - Implementa os procedimentos da interface de serviço

Estudo de Caso: RPC da Sun

- Projetada para o NFS.
- Funciona sobre UDP(64KB, na pratica 8 ou 9KB) ou TCP
- Semântica pelo-menos-uma-vez
- Linguagem de definição de interface chamada XDR
- Compilador *rpcgen*
- Linguagem de programação C

Exemplo de definição de interface em Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
int length;
char buffer[MAX];
};
struct writeargs {
FileIdentifier f;
FilePointer position;
Data data;
};
```

```
struct readargs {
FileIdentifier f;
FilePointer position;
Length length;
};

program FILEREADWRITE {
   version VERSION {
void WRITE(writeargs)=1;          1
Data READ(readargs)=2;            2
   }=2;
} = 9999;
```

Linguagem XDR

- Originalmente para representação externa
- Não utiliza nomeação
  - número de programa – autoridade central
  - número de versão – muda quando a assisnatura do método muda
  - ambos são passados na mensagem de requisição/reposta para verificar se é a mesma versão
- Definição de Procedimento
  - Assinatura e um Número que o identifica
- Permite apenas um parâmetro de entrada
  - Vários parâmetros são passados por meio de estruturas
- Parâmetros de saída são retornados por meio de um único resultado

Compilador *rpcgen*

- A partir da definição de uma interface o compilador gera:
  - Procedimentos *stub* no cliente
  - Procedimento *main*, despachante e procediemntos *stub* no servidor
  - Procedimentos de empacotamento e desempacotamento de XDR
    - Usados pelo despachante e *stub*

Vinculação - RPC

- Mapeador de Porta
  - Usa um número de porta bem conhecido
  - Armazena para cada serviço em execução
    - Número de programa
    - Número de versão
    - Número da porta
  - Cliente envia número de programa e número de versão para porta bem conhecida
    - Mapeador retorna a porta de serviço

Autenticação – Sun RPC

- Tipos
  - Nenhuma
  - Estilo UNIX (gid + uid)
  - Chave compartilhada para assinar mensagens
  - Autenticação via Kerberos

C program for client in Sun RPC

http://www.cdk4.net/additional/rmi/Ed2/SunRPC.pdf

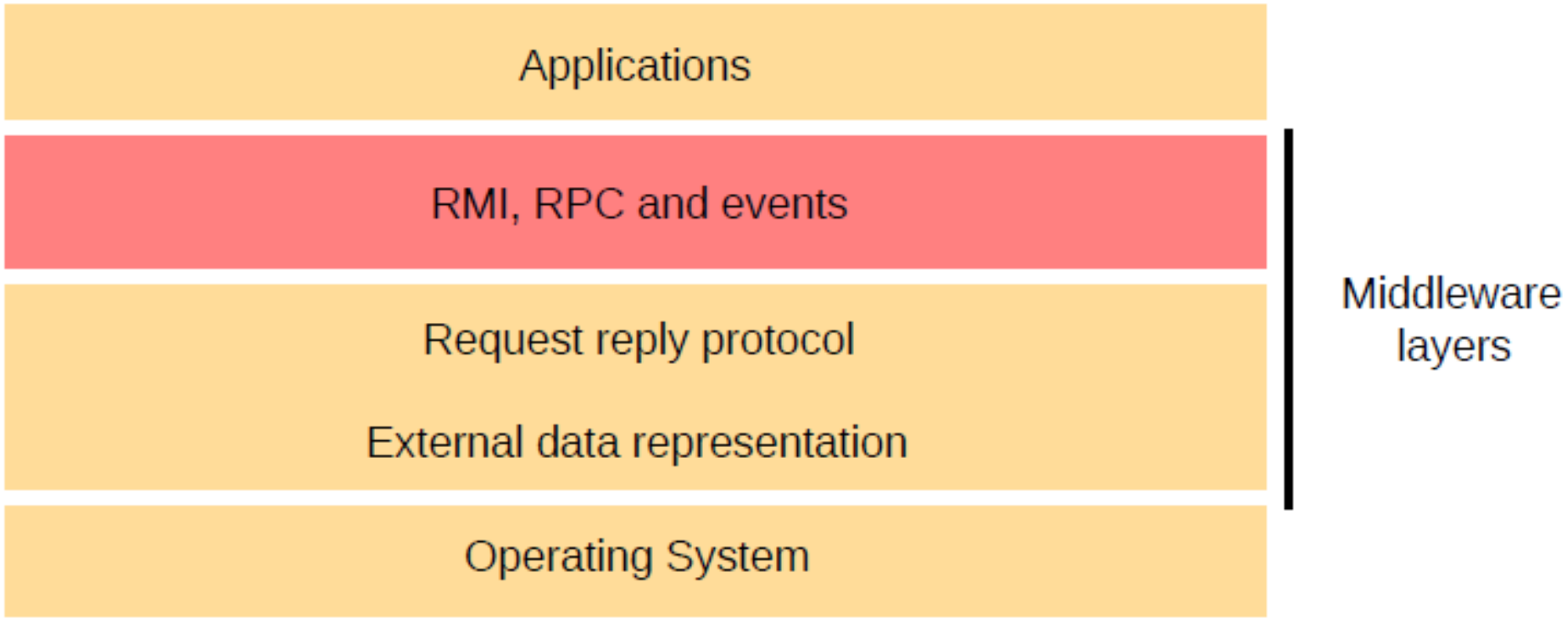
```
#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite .h"

main(int argc, char ** argv)
{
        CLIENT *clientHandle;
        char *serverName = "coffee";
        readargs a;
        Data *data;

        clientHandle= clnt_create(serverName, FILEREADWRITE,
              VERSION, "udp");      /* creates socket and a client handle*/
        if (clientHandle==NULL){
              clnt_pcreateerror(serverName); /* unable to contact server */
              exit(1);
        }
        a.f = 10;
        a.position = 100;
        a.length = 1000;
        data = read_2(&a, clientHandle);/* call to remote read procedure */
        •••
        clnt_destroy(clientHandle);           /* closes socket */
}
```

```
/* File S.c - server procedures for the FileReadWrite service */
#include <stdio.h>
#include <rpc/rpc.h>
#include"FileReadWrite.h"

void * write_2(writeargs *a)
{
 /* do the writing to the file */
}

Data * read_2(readargs * a)
{
        static Data result;       /* must be static */
        result.buffer  = ...      /* do the reading from the file */
        result.length  = ...      /* amount read from the file */
        return &result;
}
```

Agenda

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

Camadas do *Middleware*



Interfaces

- Provêem 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 reposta
  - ponteiros não são permitidos
  - objetos como parâmetros: referência de objeto

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

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

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 distribuídos

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

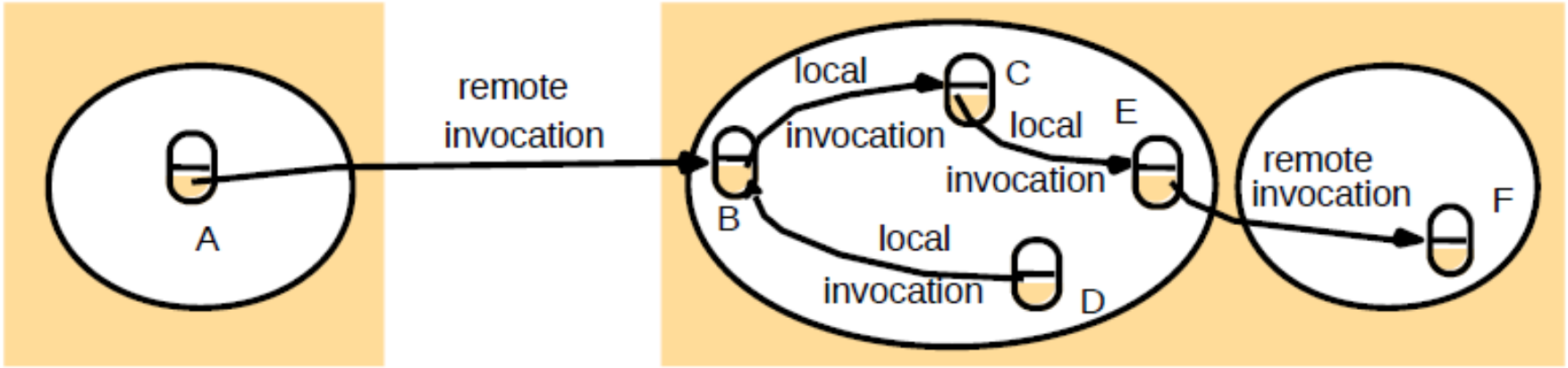
Modelo de objetos básico

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

Objetos distribuídos: características adicionais

- Arquitetura típica: cliente-servidor
  - Objetos gerenciados pelos servidores
  - Clientes realizam invocações remotas (RMI)
- Encapsulamento mais rigoroso
- Efeitos da concorrência
  - Usar primitivas de sincronização

Chamadas locais e remotas

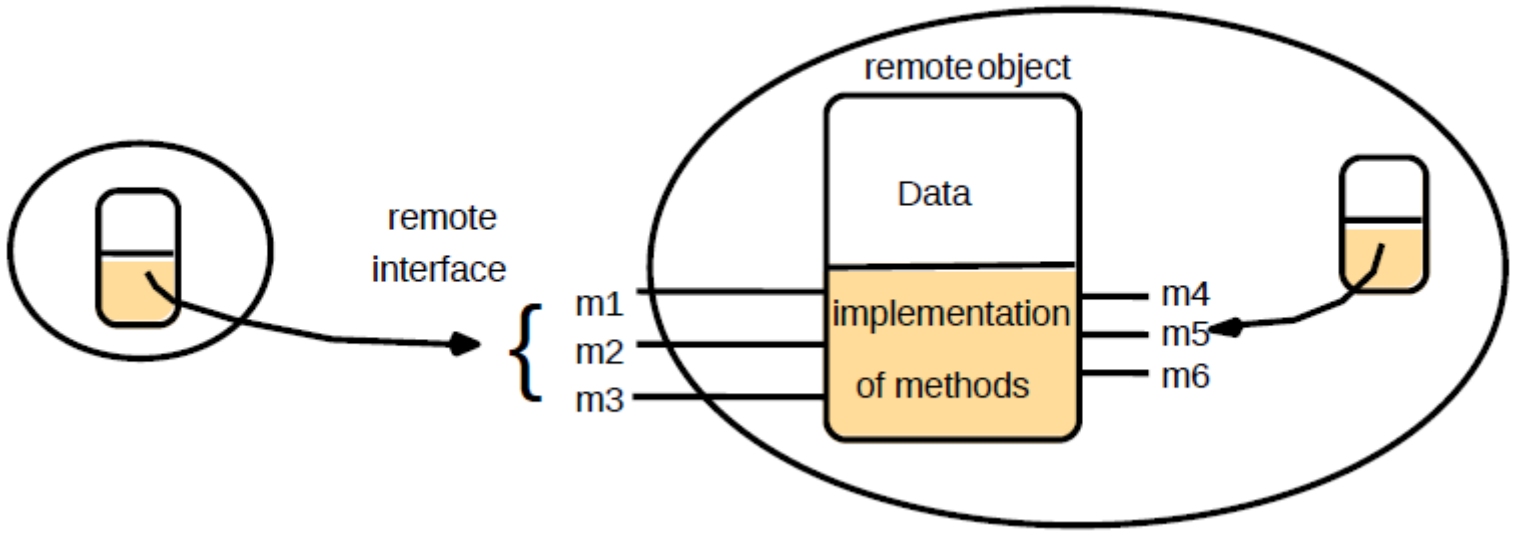


- Invocações Remotas(Processos Diferentes) vs Invocações Locais
- Referência de objeto remota
- Interface remota

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

- Objetos remotos vs. objetos locais
- 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.

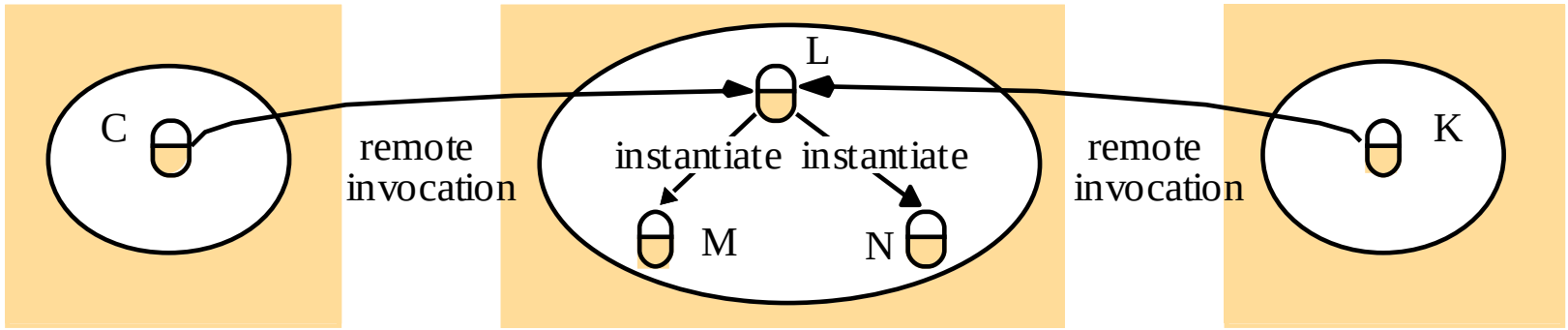
Um objeto com interfaces local e remota



O modelo de objetos distribuídos: uma extensão ao modelo de objetos básico (2)

- Exemplos de ações:
  - requisição para executar alguma operação em um 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 (3)

- Exceções
  - erros de aplicação: gerados pela lógica do servidor
  - erros de sistema: gerados pelo *middleware*
  - 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
  - sintaxe transparente, mas semântica explicitamente distinta (discutir com base no próximo slide)

Semântica de chamadas em RMI

Medidas de tolerância à falha			*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

Semântica **pelo menos uma vez**

- O invocador recebe um resultado quando o método foi executado pelo menos uma vez, ou recebe um 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)**

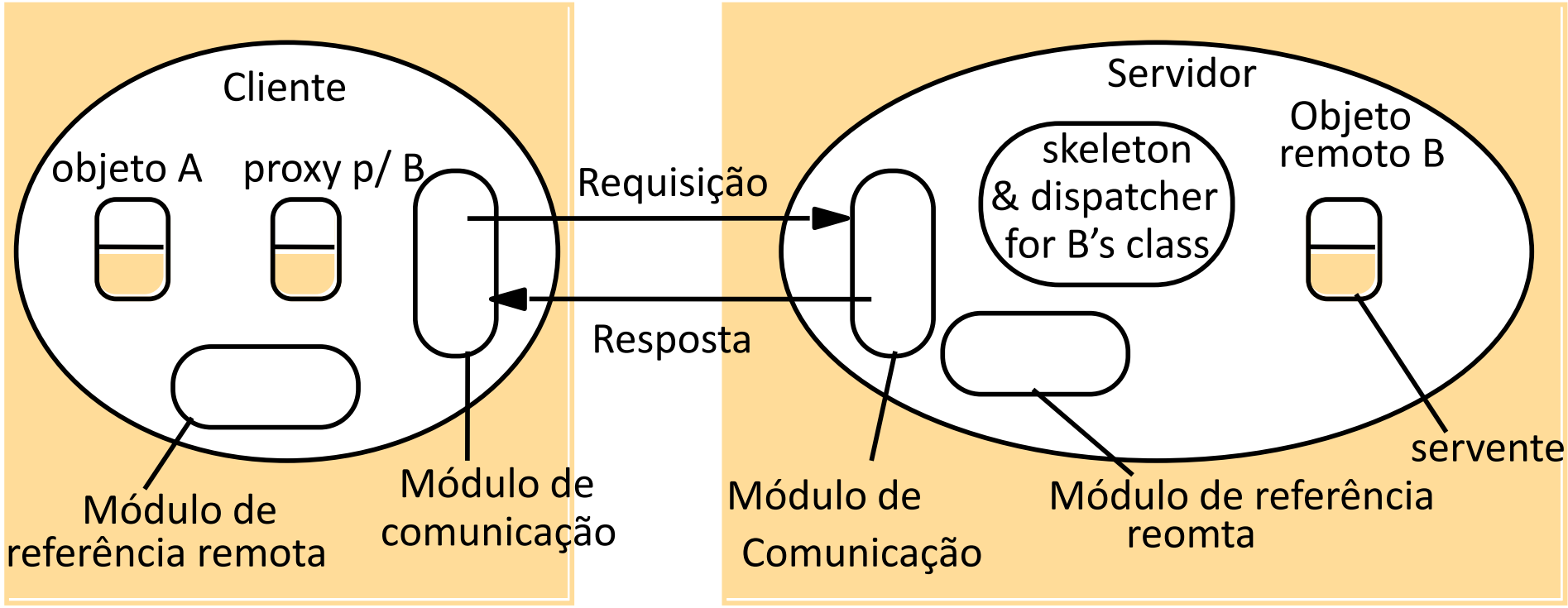
Semântica **no máximo uma vez**

- 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**

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
  - RMI java
    - Objetos remotos implementam a interface Remote
  - Invocações remotas são mais suscetíveis a falhas
    - Os invocadores devem saber diferencia 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
  - referencia 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*
- Duas variantes:
  - Despachantes específicos gerados para cada classe de objetos remotos
    - utiliza o ID do método para chamar o *skeleton*
  - Despachante genérico
    - utiliza o ID do objeto para chamar um método genérico do *skeleton associado ao objeto alvo da requisição*
    - o próprio *skeleton se encarrega de chamar o método-alvo*
    - abordagem do adaptador de objetos de CORBA

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

Chamadas dinâmicas

- *Proxy* dinâmico
  - genérico, independente de interface
  - utiliza uma definição de interface acessível dinamicamente para formatar as requisições
    - repositório de interfaces
  - útil quando não se conhece a interface em tempo de programação
- *Skeleton* dinâmico
  - permite receber chamadas destinadas a “qualquer” tipo de objeto/interface
  - útil na construção de servidores “genéricos”

Programas: Servidor e Cliente

- Servidor:
  - código para instanciação das implementações de objetos (instância da impl. de objeto = servente)
  - dá origem ao processo servidor que hospeda os objetos remotos
- Cliente:
  - código que faz uso de objetos remotos
  - não necessariamente composto de objetos

Serviços de suporte

- *Binder*
  - mapeia nomes para referências de objetos
  - Ex.: serviço de nomes de CORBA, *Registry* de RMI
- Serviço de localização
  - mapeia referências de objetos para as respectivas (prováveis) localizações físicas dos objetos
  - provê suporte para migração de objetos e redirecionamento
  - elimina a necessidade de guardar endereço IP e porta na referência de objeto!

Serviços de suporte (2)

- *Threads*
  - cada requisição é servida com o uso de uma *thread* separada no processo servidor
  - evita o bloqueio do servidor e permite que o mesmo processe várias requisições "simultaneamente"
- Ativação de objetos
  - objetos podem ser salvos em disco quando não utilizados (estado serializado + meta-dados)
    - economia de recursos (ex.: memória)
  - ao ser necessário, um objeto pode ser reativado
    - um novo servente é criado para "materializar" o objeto

Serviços de suporte (3)

- Armazenamento persistente de objetos
  - serialização do estado dos objetos
  - gerenciamento do armazenamento dos objetos
  - estratégia para definir
    - quando um objeto deve ser desativado (ir para o estado “passivo”)
    - quais partes do estado do objeto devem ser salvas
  - o uso de persistência deve ser transparente para o implementador do objeto e para os clientes

Coleta de lixo distribuída

- Idéia geral:
  - Um objeto existe enquanto houver alguma referência para ele no sistema distribuído
  - Quando a última referência for removida, o objeto pode ser destruído
- Mecanismo:
  - Interceptação de referências de objetos remotos passadas como parâmetros ou retorno de RMI
  - Contagem de referências (no processo servidor)
  - Contador incrementado/decrementado como resultado da criação/destruição de *proxies* nos clientes

Agenda

- Protocolos de requisição-resposta
- Chamada de procedimento remoto
- Comunicação entre objetos distribuídos
- 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 distantes
  - Efetua automaticamente o **empacotamento** e **desempacotamento** dos parâmetros
  - Envia as **mensagens** de pedido 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

Java RMI

- Download de classes
  - Classes são carregadas por *download* entre as JVM
  - 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.**

JNDI

- *Java Naming and Directory Interface*
  - Mecanismo de nomes do J2EE
  - Utilizado para **associar nomes a recursos e** objetos de forma portável
    - Identificação, localização, partilha
  - Mapeia nomes em referências para objetos
  - Uma instância do **registry** deve executar-se em todos servidores que têm objetos remotos.
  - Os clientes têm de dirigir as suas pesquisas para o **servidor pretendido**

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 objecto.
  - 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

Figure 5.12
Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;   1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;  2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

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

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new ShapeListServant();        1
            Naming.rebind("Shape List", aShapeList );             2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Figure 5.15 - Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
      private Vector theList;                    // contains the list of Shapes          1
      private int version;
     public ShapeListServant()throws RemoteException{...}
     public Shape newShape(GraphicalObject g) throws RemoteException {          2
          version++;
               Shape s = new ShapeServant( g, version);                          3
               theList.addElement(s);
               return s;
     }
     public  Vector allShapes()throws RemoteException{...}
     public int getVersion() throws RemoteException { ... }
}
```

Figure 5.16 - Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");     1
          Vector sList = aShapeList.allShapes();                            2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Java RMI – executando o exemplo -

- Códigos
  - http://www.cdk4.net/rmi/ + arquivo de política (próximo slide)
- Compile as classes
  - javac -d . *.java
- Crie o *Stub*(*Proxy*) e o *Skeleton*
  - rmic -d . rmi.ShapeListServant
- Rode o RMIRegistry
  - rmiregistry
- Rode o Servidor
  - java -Djava.server.rmi.codebaseile:///rmi/ -Djava.security.policy=policy rmi.ShapeListServer
- Rode o Cliente
  - java -Djava.security.policy=policy rmi.ShapeListClient

Arquivo “policy”

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

Exercícios

- Fazer todos os exercícios do capítulo 5 e entregar em modo manuscrito;