Computação Distribuída

Cap IV – Invocação Remota

Licenciatura em Engenharia Informática

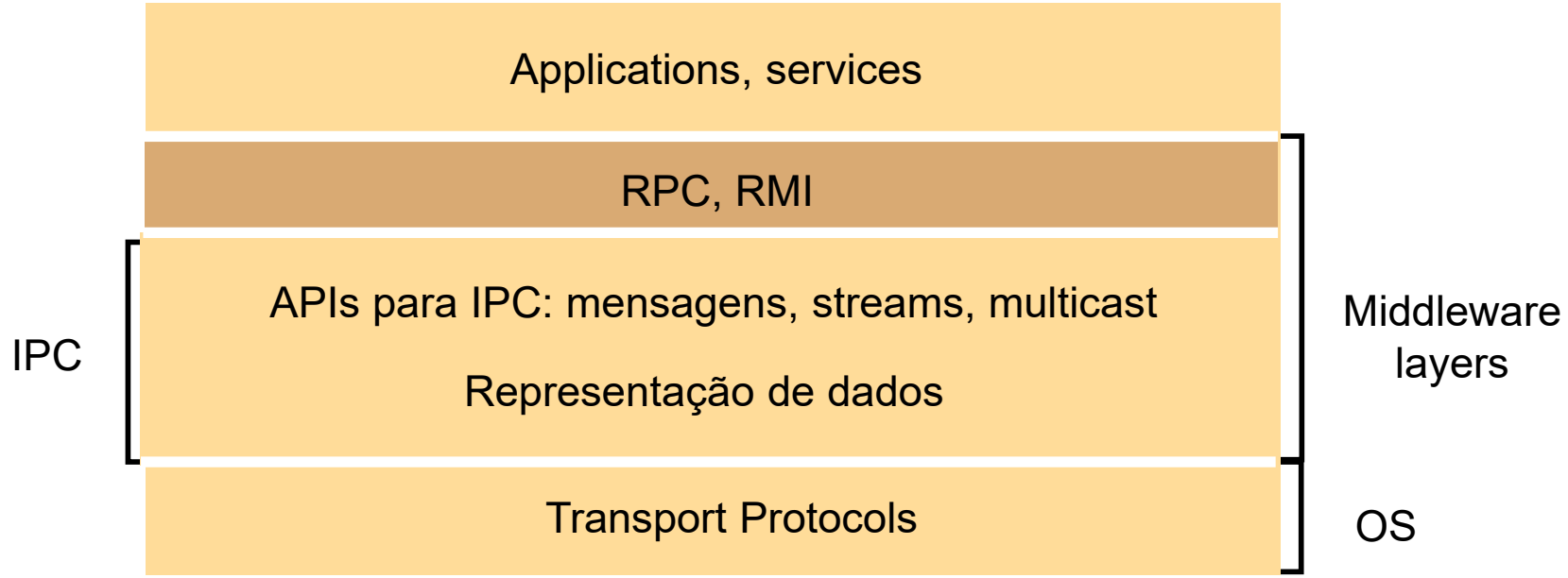
Universidade Lusófona

Prof. Paulo Guedes (paulo.guedes@ulusofona.pt)

Sumário

- Remote Procedure Call (RPC)
  - Conceito, estrutura do programa cliente, estrutura do programa servidor
  - Infraestrutura de RPC: Interface Description Language (IDL), rotinas de adaptação (stubs), biblioteca de RPC
  - Semânticas de RPC vs protocolos de comunicação, protocolos de representação de dados
- Sun RPC
  - Descrição, exemplos
- Invocação remota de objetos (RMI)
  - Conceito de objetos distribuídos, interfaces remotas, instanciação remota
  - Java RMI: descrição, exemplos

Modelos de Programação

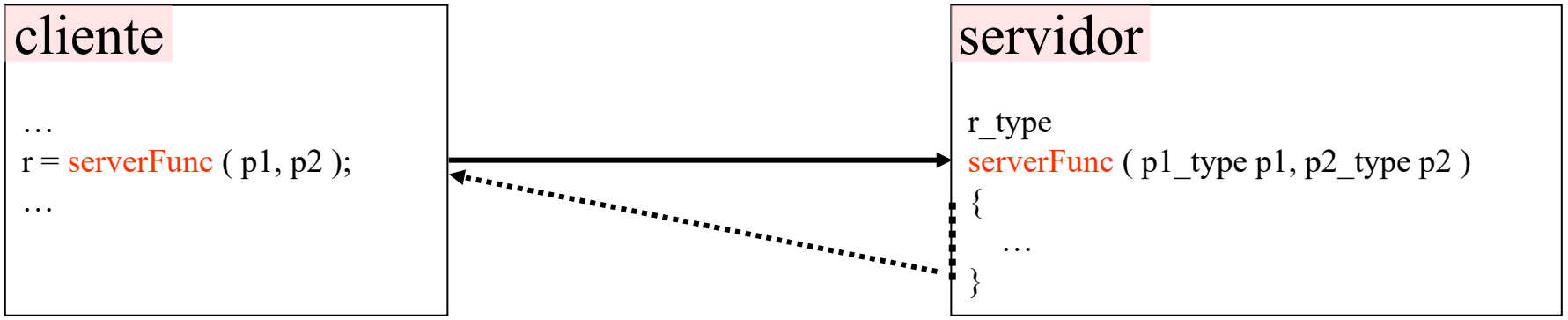


- O nível de invocação fornece um modelo de programação para as aplicações distribuídas
  - Extensão do modelo clássico de invocação de rotinas ou métodos
  - Implementação real da transparência de acesso e localização
- Modelos de programação
  - *Remote Procedure Call* - RPC
  - *Remote Method Invocation* - RMI

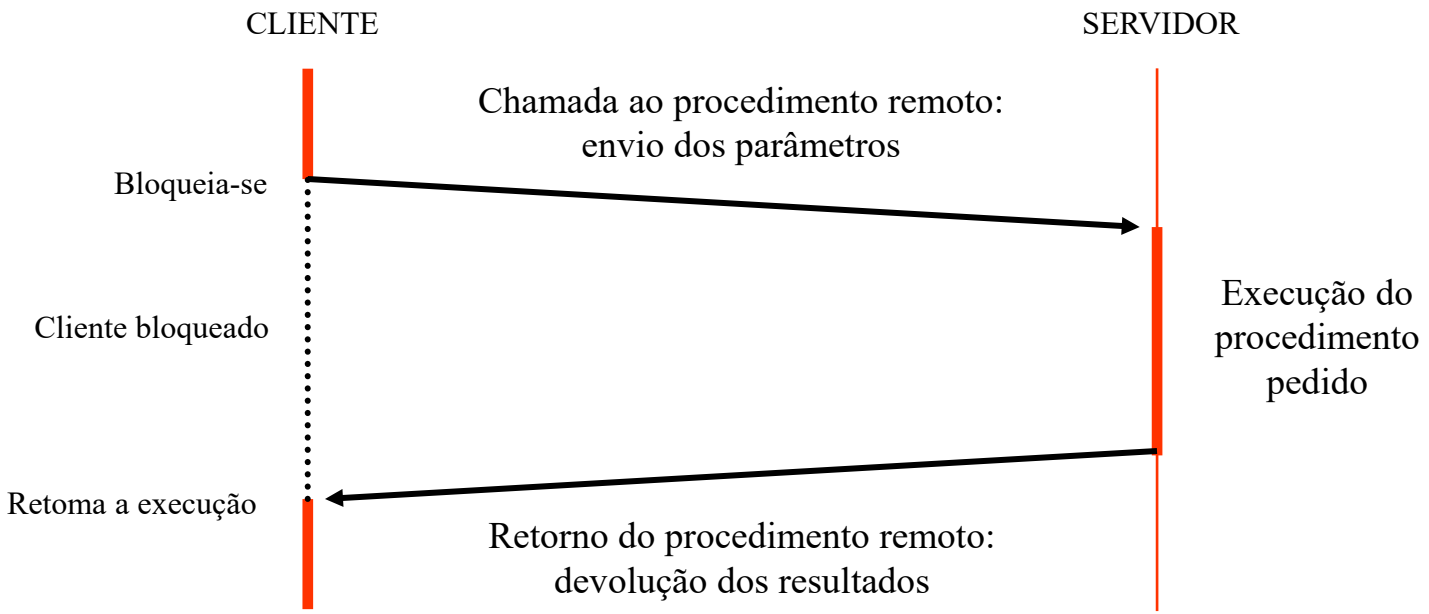
Chamada de Procedimento Remoto (*Remote Procedure Call*, RPC)

Objectivo: Estruturar a programação distribuída com base na chamada pelos clientes de procedimentos que se executam remotamente no servidor

- Execução de um procedimento noutro processo
  - O chamador (cliente) envia uma mensagem com um pedido
  - O chamado (servidor) devolve uma mensagem com a resposta
- O programador chama um procedimento local normal
  - O envio e recepção de mensagens são escondidos



RPC: (i) Fluxo de execução



RPC: (ii) Benefícios

- Adequa-se ao fluxo de execução das aplicações
  - Chamada síncrona de funções
- Simplifica tarefas fastidiosas e delicadas
  - Construção e análise de mensagens
  - Heterogeneidade de representações de dados
- Simplifica a divulgação de serviços (servidores)
  - A interface dos serviços é fácil de documentar e apresentar
  - A interface é independente dos protocolos de transporte
- Esconde diversos aspectos específicos do transporte
  - Endereçamento do servidor
  - Envio e recepção de mensagens
  - Tratamento de erros

RPC: (iii) Utilização no cliente

1. Estabelecimento da ligação (*binding*)
   1) Identificação do servidor
   2) Localização do servidor
   3) Estabelecimento de um canal de transporte
   4) Autenticação do cliente e/ou do servidor
2. Chamada de procedimentos remotos

3. Terminação da ligação
   - Eliminação do canal de transporte

Estrutura do Programa Cliente

- Inicialização
  - Cria os seus canais de comunicação
  - Efectua o binding ao servidor
- Código da aplicação com
  - Chamadas a procedimentos locais
  - Chamadas a procedimentos remotos

    (Poderá ser difícil de distinguir uns dos outros...)
- Terminação da ligação ao servidor (opcional)

RPC: (iv) Utilização no servidor

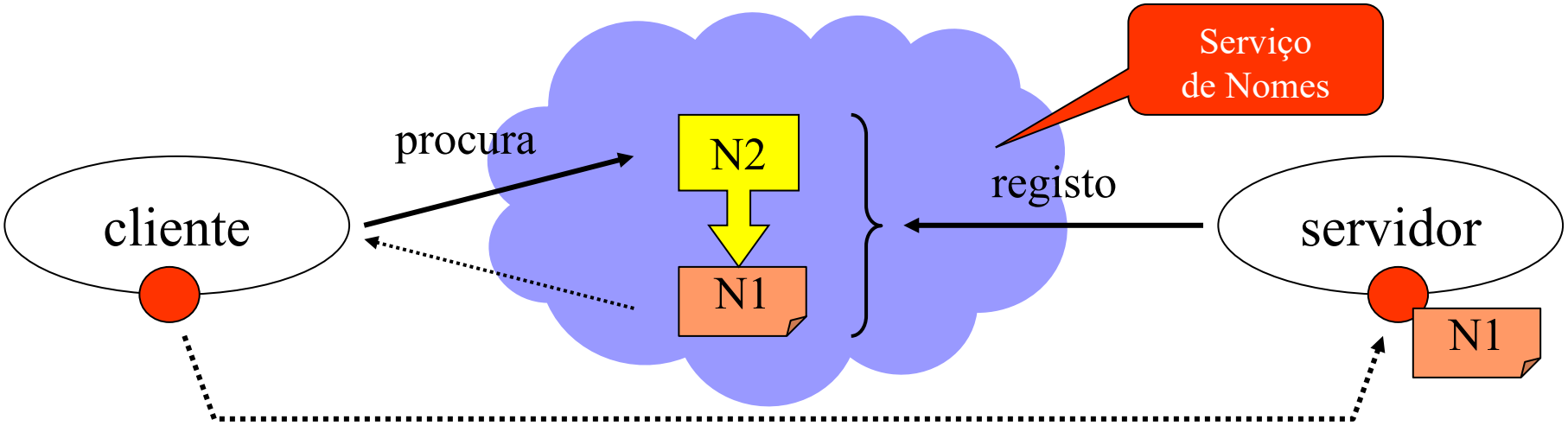
1. Registo
   1) Escolha da identificação do utilizador
      - Nome do porto de transporte
      - Outros nomes alternativos
   2) Registo dessa identificação
2. Esperar pela criação de ligações
   1) Estabelecimento de um canal de transporte
   2) Autenticação do cliente e/ou do servidor
3. Esperar por pedido de execução de procedimentos
   - Enviados pelos clientes ligados
4. Terminação da ligação
   - Eliminação do canal de transporte

Estrutura do Programa Servidor

- Inicialização
  - Cria os seus canais de comunicação
  - Regista a sua interface no Serviço de Nomes
  - Chama a biblioteca de RPC
- Código da aplicação
  - Procedimentos isolados que serão chamados remotamente pelos clientes
  - Não existe “programa principal” (“main()”). É a biblioteca de RPC que espera pelos pedidos e chama as rotinas do programa
  - Aplicação funciona como uma máquina de estados:
    - Um procedimento é chamado pelo cliente e executa-se. O estado fica mantido em variáveis globais, ficheiro, base de dados...
    - O procedimento retorna para a biblioteca de RPC que envia a resposta ao cliente.
    - O servidor tem que estar preparado para qualquer sequência de chamada dos seus procedimentos. Ele não pode controlar o seu fluxo de execução, pois este é determinado pelas chamadas efectuadas pelos clientes.

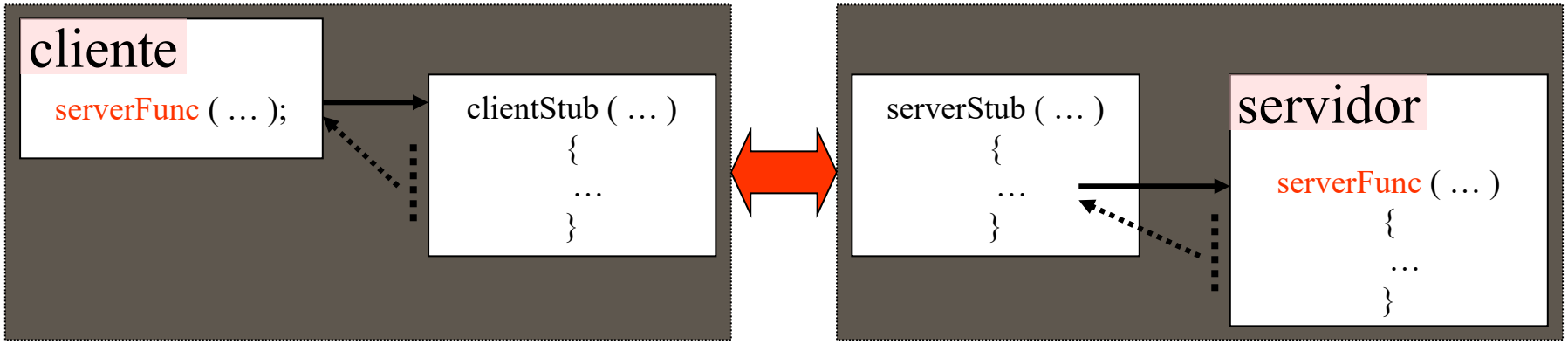
RPC: (v) Serviço de Nomes

- Permite que o servidor registe um nome
  - De alguma forma associado ao nome de um porto de transporte que possui
- Permite que um cliente consiga encontrar um servidor
  - Obter o nome do seu porto de trasporte

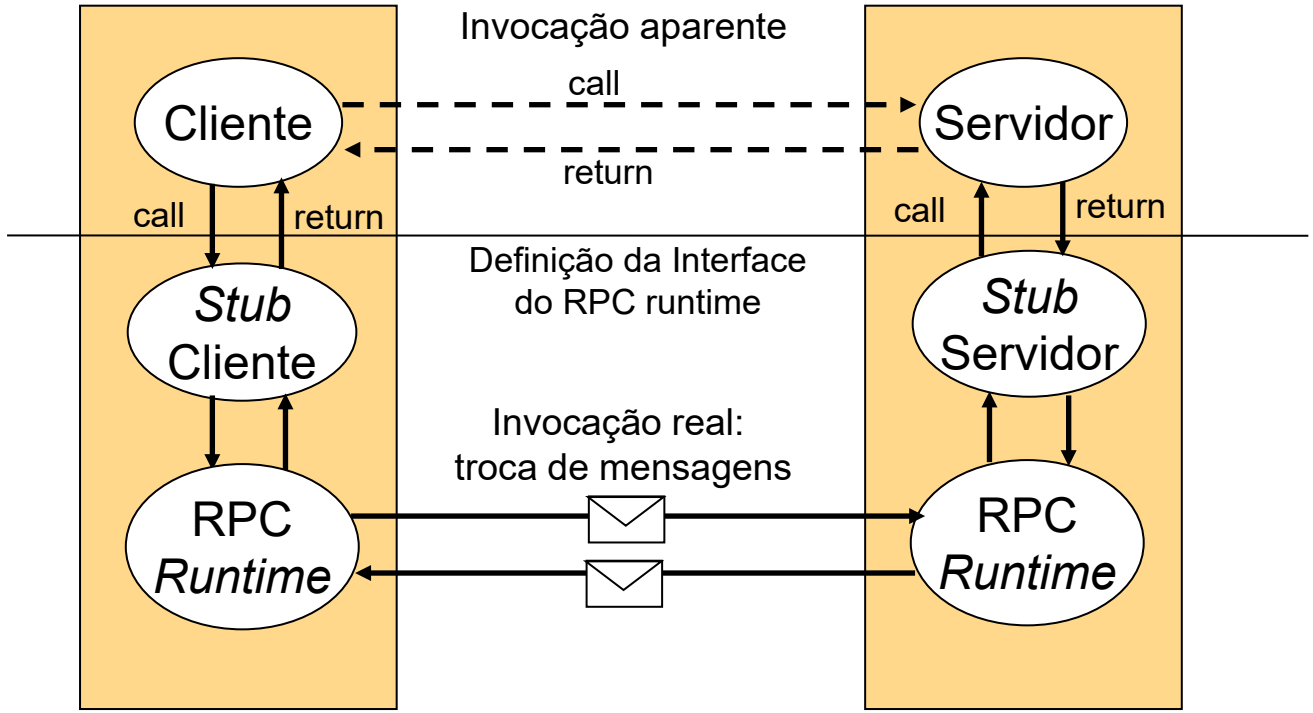


RPC: (vi) Rotinas de adaptação (*stubs*)

- Cliente
  - Conversão de parâmetros
  - Criação e envio de mensagens (pedidos)
  - Recepção e análise de mensagens (respostas)
  - Conversão de resultados
- Servidor
  - Recepção e análise de mensagens (pedidos)
  - Conversão de parâmetros
  - Conversão de resultados
  - Criação e envio de mensagens (respostas)



Sequência de Invocação de um RPC



- A sequência de invocação de um método remoto é implementada por vários módulos de software
  - *Client Stub*: funciona como o agente de representação (proxy) do método remoto
  - *Server Stub*: realiza a chamada no contexto do servidor ao método invocado
  - Os *runtimes* do RPC asseguram o encaminhamento das mensagens e o *dispatching* destas para o *stubs*

Perguntas a que devo ser capaz de responder

- Qual é a relação entre o RPC e a arquitetura cliente-servidor ?
- Quais são as componentes de um sistema de RPC ?
- Qual é a sequência de execução dos componentes de RPC durante a execução de uma chamada remota ?

RPC: (vii) Infra-estrutura de apoio

- No desenvolvimento
  - Um linguagem de especificação de interfaces
    - *Interface Description Language*, IDL
  - Compilador de IDL
    - Gerador de *stubs*
- Em tempo de execução
  - Serviço de Nomes
  - Biblioteca de suporte (*RPC Run-Time Support*)
    - Registo de servidores
    - *Binding*
    - Protocolo de execução de RPCs
    - Controlo global da interacção cliente-servidor

RPC: (viii) Entraves à transparência

- IDL
  - Normalmente diferente da linguagem usada pelos programas
- Passagem de parâmetros
  - Semânticas não suportadas pelo RPC
- Execução do procedimento remoto
  - Fluxos de execução complexos
  - Tolerância a faltas e notificação de faltas
- Desempenho
  - Depende em grande medida da infra-estrutura de comunicação entre cliente e servidor

RPC IDL: (i) Características

- Linguagem própria
  - Depende dos protocolos de sessão e apresentação do RPC
- Apenas de declarativa, permite definir
  - Tipos de dados
  - Protótipos de funções
    - Fluxo de valores (IN, OUT, INOUT)
  - Interfaces
    - Conjuntos de funções
  - Identificadores
    - Serviços, interfaces, etc.

RPC IDL: (ii) Código gerado pelo compilador

- *Stubs*
  - No cliente
    - Traduzem e empacotam parâmetros numa mensagem
    - Enviam a mensagem para o servidor, esperam uma resposta
    - Desempacotam a mensagem e traduzem a resposta
  - No servidor
    - Desempacotam a mensagem e traduzem os parâmetros
    - Invocam a função desejada e esperam pelo seu retorno
    - Traduzem e empacotam a resposta numa mensagem
- Função de despacho do servidor
  - Espera por mensagens de clientes num porto de transporte
  - Envia mensagens recebidas para o *stub* respectivo
  - Recebe mensagens dos *stubs* e envia-os para os clientes

RPC IDL: (iii) Limitações usuais

- Ambiguidades acerca dos dados a transmitir:
  - Endereçamento puro de memória (void *)
  - Flexibilidade no uso de ponteiros para manipular vectores
    - Passagem de vectores (normalmente por ponteiro)
    - *Strings* manipuladas com char *
  - Passagem de variáveis por referência (&var)
- Semânticas ambíguas
  - Valores booleanos do C (0 → False, != 0 → True)
- Problemas complexos (durante a execução)
  - Transmissão de estruturas dinâmicas com ciclos
  - Integridade referêncial dos dados enviados

RPC IDL: (iv) Soluções para alguns dos problemas

- Novos tipos de dados próprios do IDL
  - Sun RPC define 3 novos tipos de dados
    - string: para definir cadeias de caracteres
    - bool: valor booleano, apenas dois valores
    - opaque: bit-arrays, sem tradução
- Agregados próprios do IDL
  - Uniões (*unions*) com discriminantes
  - Vectores conformes (DCE/Microsoft)
  - Vectores variáveis (Sun, DCE/Microsoft)

Exemplo: Interface em IDL

```
[
uuid(00918A0C-4D50-1C17-9BB3-92C1040B0000),
version(1.0)
]
interface banco
{
typedef enum {
    SUCESSO,
    ERRO,
    ERRO_NA_CRIACAO,
    CONTA_INEXISTENTE,
    FUNDOS_INSUFICIENTES
  } resultado;

typedef enum {
    CRIACAO,
    SALDO,
    DEPOSITO,
    LEVANTAMENTO,
    TRANSFERENCIA,
    EXTRATO
  } tipoOperacao;

typedef struct {
    long dia;
    long mes;
    long ano;
  } tipoData;

typedef struct {
    tipoData data;
    tipoOperacao operacao;
    long movimento;
    long saldo;
  } dadosOperacao;
```

```
resultado criar([in] handle_t     h,
          [in] long         valor,
          [in, string] char nome[],
          [in, string] char morada[],
          [out] long        *numero);

resultado saldo([in] handle_t h,
          [in] long     nConta,
          [out] long    *valor);

resultado depositar([in] handle_t h,
          [in] long     nConta,
          [in] long     valor);

resultado levantar([in] handle_t h,
          [in] long     nConta,
          [in] long     valor);

resultado transferir([in] handle_t h,
          [in] long     nContaOrigem,
          [in] long     nContaDest,
          [in] long     valor);

resultado pedirExtrato([in] handle_t h,
          [in] long     nConta,
          [in] long     mes,
          [in] long     ano,
          [in, out, ptr] dadosOperacao dados[50],
          [out] long *nElemento);
}
```

Perguntas a que devo ser capaz de responder

- O que é um IDL ?
- O que é que um compilador de IDL produz ?
- Quais são as diferenças entre um IDL e uma linguagem de programação ?

Semânticas de Invocação

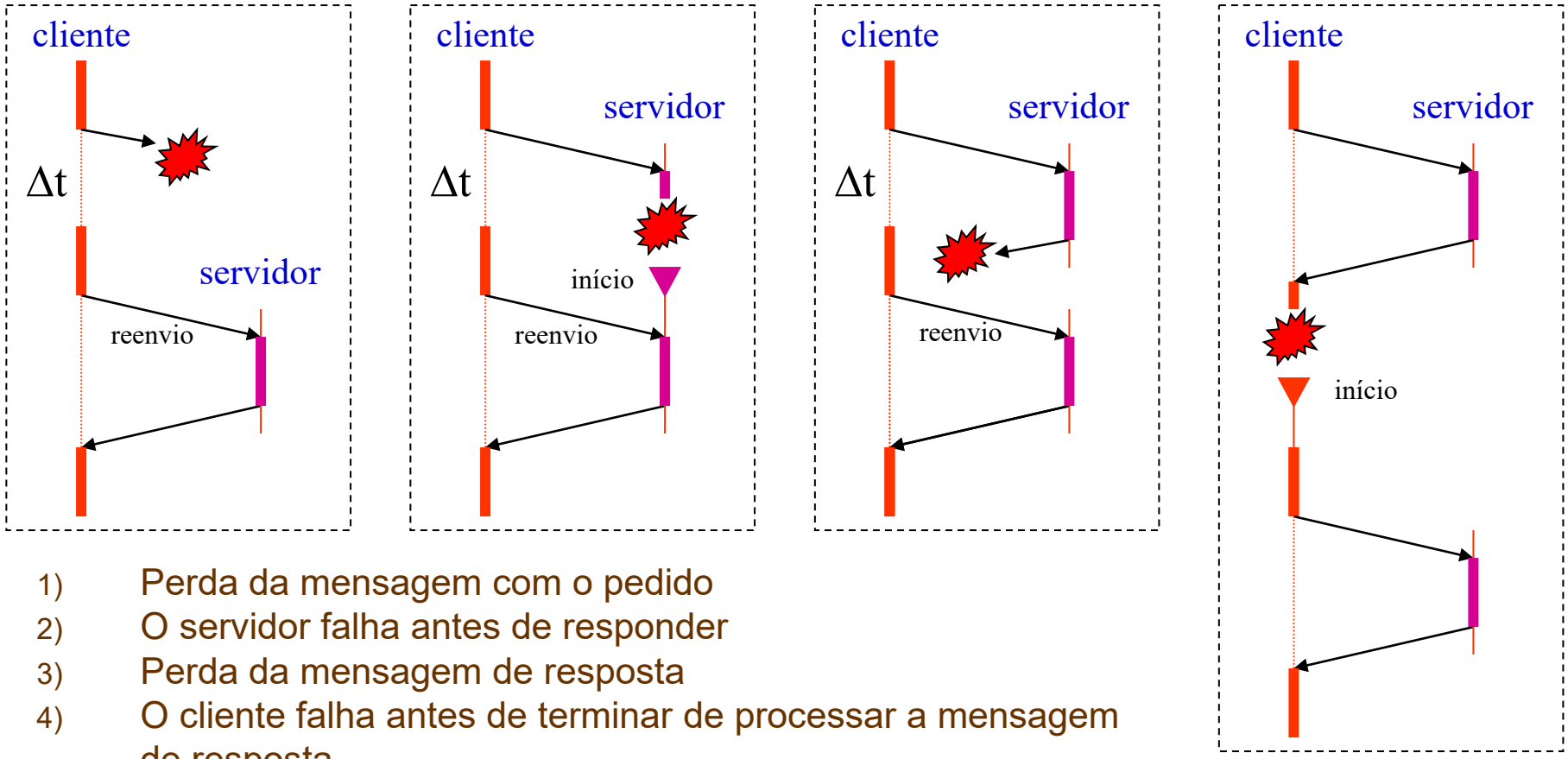
- A semântica de uma invocação indica a forma como são executados os vários passos que a compõem e quais as possíveis acções e consequências em caso de falha

  Semântica ideal ≡ procedimento local

Modelo de falhas

- Falhas por omissão e ordenação
  - Se o protocolo de transporte não garante entrega ordenada das mensagens o middleware de invocação deve gerir retransmissões e números de sequência
- Falhas Temporais
  - A ocorrência de *timeouts* pode significar perca de mensagem ou *crash* do serviço
- Em caso de falha, uma invocação pode ser realizada múltiplas vezes
  - O efeito da invocação repetida tem de ser gerido correctamente de forma a evitar incoerências no resultado
- O tipo da operação condiciona a semântica da invocação
  - Operações **idempotentes:** o efeito da sua execução mais que uma vez é equivalente à sua execução uma só vez. Estas funções podem ser realizadas várias vezes
    - Ex. leitura de valores; consulta de saldo
  - Operações **não idempotentes** só podem ser realizadas uma vez
    - Ex: escrita e acumulação de valores; depósito numa conta

RPC: Diagrama das falhas possíveis



1) Perda da mensagem com o pedido
2) O servidor falha antes de responder
3) Perda da mensagem de resposta
4) O cliente falha antes de terminar de processar a mensagem de resposta

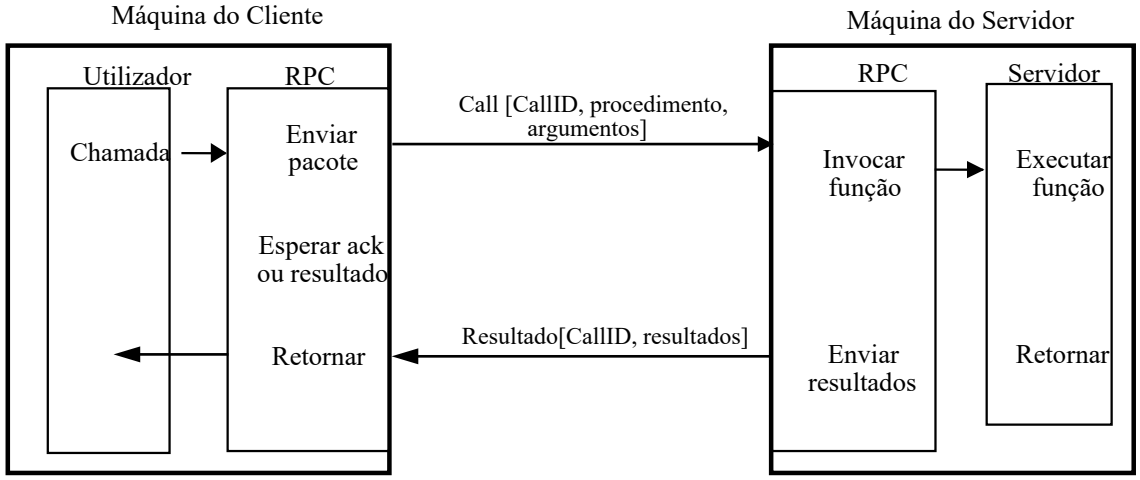
RPC: Semânticas de execução

- Recuperação de falhas
  - Normalmente não contempla falhas de máquinas ou de servidores
  - Assume que as máquinas são *fail-silent*
    - Em caso de falha não interagem erradamente
- Semânticas
  - Talvez (*maybe*)
  - Pelo-menos-uma-vez (*at-least-once*)
  - No-máximo-uma-vez (*at-most-once*)
  - Exactamente-uma-vez (*exactly-once*)

RPC: Semânticas de execução

- Semântica talvez
  - O *stub* cliente retorna um erro se não receber uma resposta num prazo limite
  - Sem uma resposta o cliente não sabe se o pedido foi executado ou não
- Semântica pelo-menos-uma-vez
  - O *stub* cliente repete o pedido até obter uma resposta
  - Caso haja uma resposta o cliente tem a garantia que o pedido foi executado pelo menos uma vez
  - Útil para serviços com funções idempotentes
- Semântica no-máximo-uma-vez
  - O protocolo de controlo tem que:
    - Identificar os pedidos para detectar repetições no servidor
    - Manter estado no servidor acerca de que pedidos estão em curso ou já foram atendidos
- Semântica exactamente-uma-vez
  - Na presença de falhas a actividade do servidor tem que ser controlada por monitores transaccionais

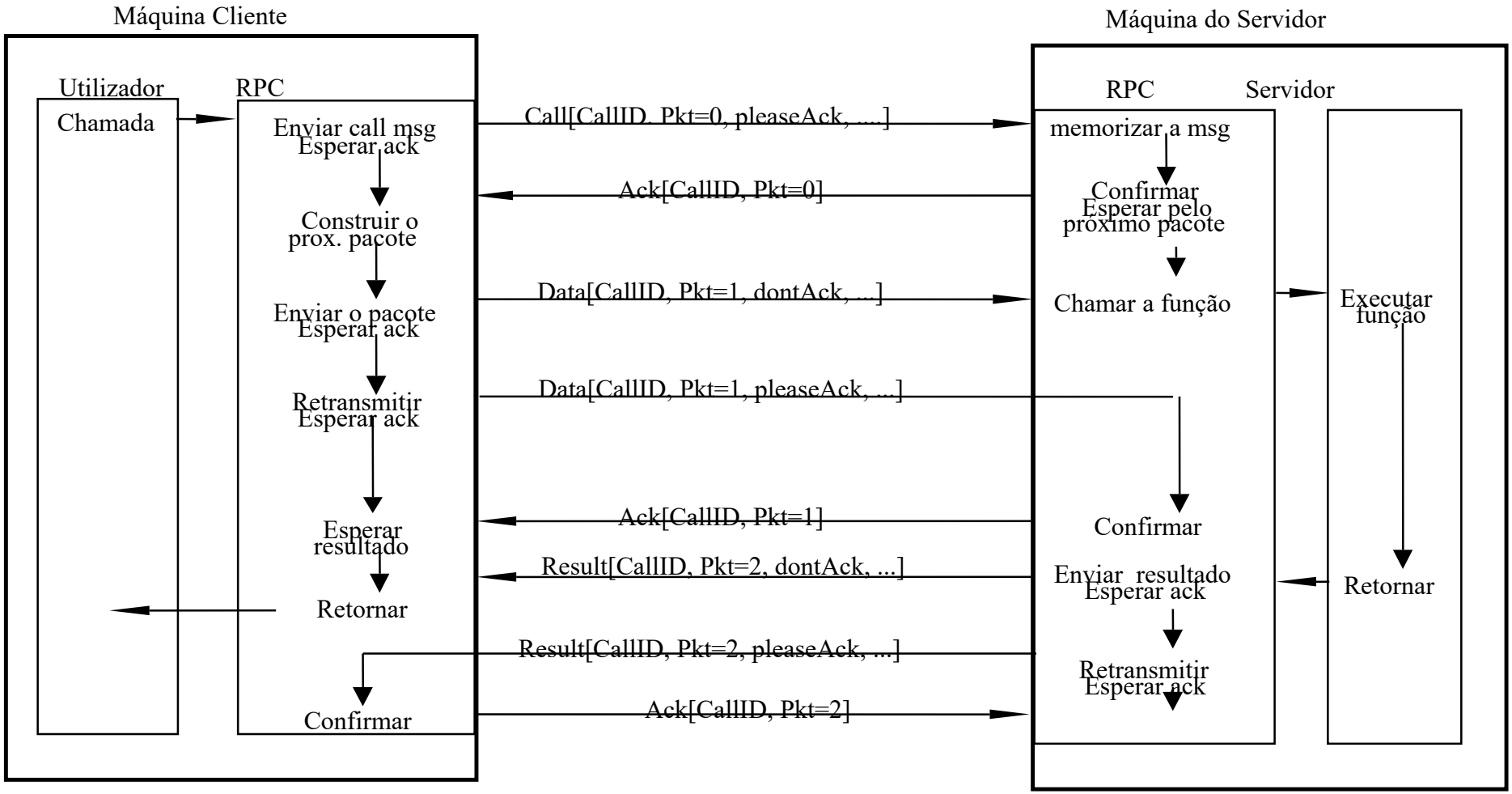
Protocolo de RPC: Situação Ideal



RPC: Protocolo de controlo

- Suporte de vários protocolos de transporte
  - Com ligação
    - O controlo é mais simples
    - RPCs potencialmente mais lentos
  - Sem ligação
    - Controlo mais complexo (mais ainda se gerir fragmentação)
    - RPCs potencialmente mais rápidos
- Emparelhamento de chamadas/respostas
  - Identificador de chamada (CallID)
- Confirmações
  - Temporizações
  - Evolução monotónica dos CallIDs evita envio de confirmação ao servidor
- Estado do servidor para garantir semânticas
  - Tabela com os CallIDs das chamadas em curso
  - Tabela com pares (CallID, resposta enviada)

Protocolo de RPC: Situação Complexa



RPC: Semânticas de execução e protocolos

- A semântica pode ser impostas pelo mecanismo de transporte
  - Sun RPC sobre UDP/IP:
    - Se houver resposta: pelo-menos-uma-vez
    - Sem resposta: talvez
  - Sun RPC sobre TCP/IP:
    - Se houver resposta: exactamente-uma-vez
    - Se não houver resposta: no-máximo-uma-vez

Perguntas a que devo ser capaz de responder

- Quais são as semânticas possíveis na execução de um RPC ?
- Que semântica de execução de RPC se obtém com o protocolo TCP e com o protocolo UDP quando não há falhas ? E quando há falhas ?

RPC: Representação dos dados

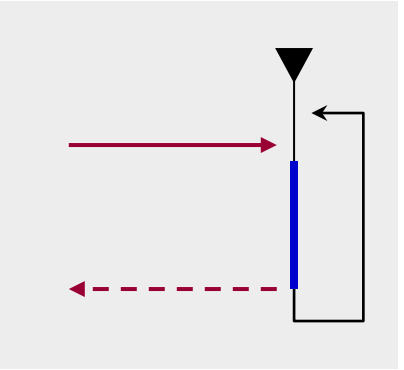
- Estrutura dos dados
  - Implícita
  - Autodescritiva (marcada, *tagged*)
- Políticas de conversão dos dados
  - Canónica
    - N formatos $\Rightarrow$ N funções
    - Não há comportamentos variáveis
    - É preciso converter mesmo quando é inútil
  - O-receptor-converte (*Receiver-makes-it-right*)
    - Poupa conversões inúteis
    - N formatos $\Rightarrow$ N x N funções

RPC: Protocolos de representação dos dados

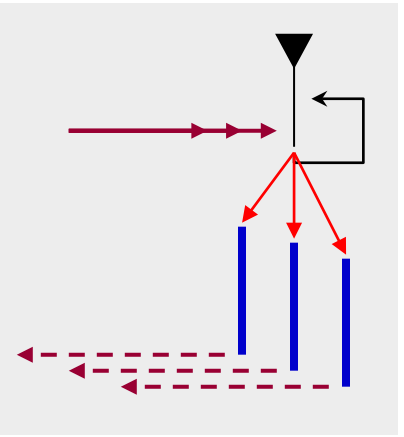
	XDR **(eXternal Data Representation)**	NDR **(Network Data Representation)**	XML
Standard para	**Sun RPC**	**DCE RPC** **Microsoft RPC**	**Web Services, outros**
Conversão	**Canónica**	**O-receptor-converte**	**Canónica**
Estrutura	**Implícita** **Comprimentos de vectores variáveis** **Alinhamento a 32 *bits* (excepto vectores de caracteres)**	**Implícita** **Marcas arquitecturais (*architecture tags*)**	**Explícita** ***Encoding Rules*: Schemas extensíveis**

Execução de RPCs:
Múltiplos fluxos de execução no servidor

- Um pedido de cada vez
  - Serialização de pedidos
  - Uma única *thread* para todos os pedidos

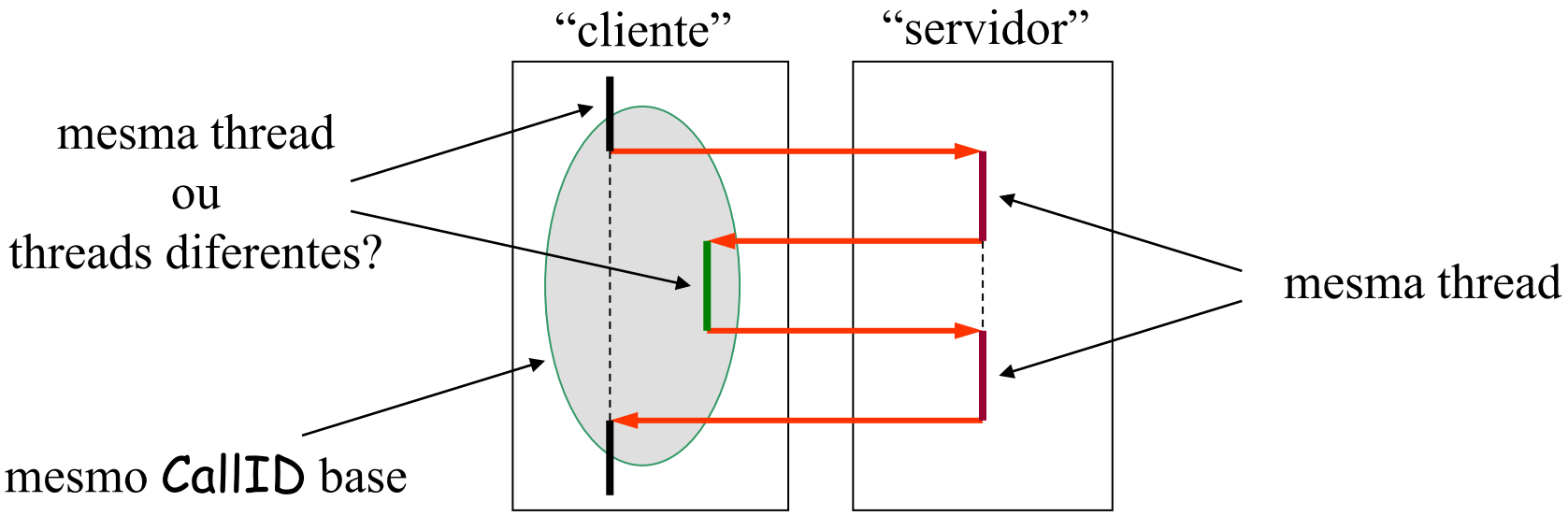


- Vários pedidos em paralelo
  - Uma *thread* por pedido
  - A biblioteca de RPC tem que suportar paralelismo:
    - Sincronização no acesso a *binding handles*
    - Sincronização no acesso a canais de comunicação

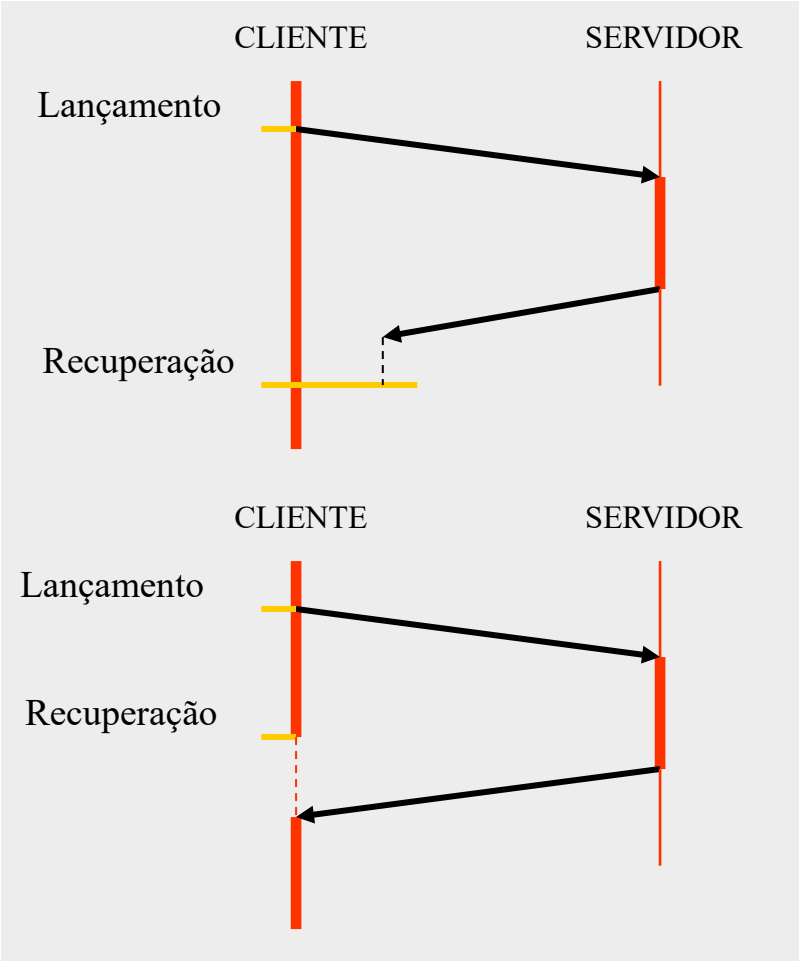


Execução de RPCs:
Fluxos de execução complexos

- Chamadas em ricochete (*callbacks*)
  - Um "cliente" é contactado como sendo um "servidor" no fluxo da sua chamada



Execução de RPCs:
Fluxos de execução alternativos



- Chamadas assíncronas (*follow-up RPC*)
  - Duas operações não consecutivas:
    - Lançamento
    - Recuperação
  - Permitem optimizar os clientes
    - Menos tempo bloqueado
    - Transparente para os servidores
  - Podem simplificar a realização de RPCs concorrentes

Execução de RPCs:
Fluxos de execução alternativos

- Chamadas sem retorno (*one-way operation*)
  - Equivalente a uma assíncrona sem recuperação
  - São definidas na interface dos serviços
    - Afecta todos os clientes (ex. atributo maybe no DCE IDL)
  - Não permitem retornar resultados
    - Porque não há qualquer mensagem de resposta
  - Semânticas limitadas
    - Talvez ou no-máximo-uma-vez (ex.talvez no DCE IDL)
  - Algumas falhas podem ser detectadas pelos clientes
    - Erros de transmissão locais

Execução de RPCs:
Fluxos de execução alternativos

- RPCs locais (numa única máquina)
  - Podem-se simplificar ou optimizar várias acções protocolares
  - Simplificações:
    - Eliminação dos protocolos de apresentação
  - Optimizações:
    - Partilha de tampões para troca de mensagens
    - Diminuição de cópias de dados
    - Troca de contextos de execução (*hand-off*)
  - A maioria dos RPCs de uso genérico não optimizam significativamente
    - Principalmente porque operam fora do núcleo do SO

Execução de RPCs:
Fluxos de execução alternativos

- RPC em difusão (*broadcast*)
  - Questões técnicas e semânticas
    - Qual a abrangência da difusão?
    - Como se processa o *binding*?
    - Qual o suporte de transporte à difusão?
    - Qual a política de envio e recolha de respostas?
  - Exemplo do Sun RPC:

    Transporte → UDP/IP em difusão, mediação através de do rpcbind de cada máquina

    Binding → Implícito para um porto UDP/IP fixo em todas as máquinas (111, porto do rpcbind)

    Respostas → Notificação de recepção, controlo da espera de respostas, os servidores normalmente não respondem em caso de erro

Perguntas a que devo ser capaz de responder

- Qual componente do sistema RPC implementa o protocolo de representação de dados ?
- Como é possível ter um servidor RPC concorrente ?

O Sun RPC

- O Sun RPC foi um dos primeiros exemplos de *middleware* para programação de aplicações distribuídas
- Desenvolvido para a implementação do sistema de ficheiros distribuído NFS - *Network File System*
  - Meados dos anos 80
  - Código posto no domínio público permitiu uma enorme expansão
  - Durante anos foi um standard *de facto* para aplicações distribuídas
- Plataforma simples e rudimentar
  - Inicialmente só suportava UDP, depois também TCP
  - Semântica de invocação *at-least-once*
- Fornece alguns serviços complementares
  - Autenticação
    - Unix style, Kerberos
  - Broadcast
    - Feita através do *Port Mapper*

(Sun Microsystems)

Definição de Interfaces (i)

- A interface especifica o formato e o tipo de interacção fornecido por uma função ou objecto
  - Define as funcionalidades de um serviço
- A interface define os parâmetros que são fornecidos à rotina invocada
  - Passagem por valor: definição de parâmetros de entrada e de saída
  - Passagem por referência: impossível se não está no mesmo processo que o programa principal
    - Pode ser simulada através do contexto de chamada dos *stubs*
    - Ponteiros não têm significado
- A especificação de interfaces no contexto de RPC é feita utilizando uma linguagem de definição
  - IDL: *Interface Definition Language*
  - Integrada com a linguagem de invocação
    - RPC SUN: C, C++, C# e Java

Definição de Interfaces (ii)

- O mecanismo de RPC localiza o serviço através do identificador das interfaces
  - Identificador único
  - Mecanismo bastante rudimentar
- No Sun RPC o identificador é composto por 3 campos:
  - *Program number* -> identifica um grupo de procedimentos fornecidos por um serviço
  - *Version Number* -> identifica a versão do serviço
  - *Procedure Number* -> identifica o método ou procedimento
- O Identificador é atribuído pelo programador do serviço
  - Gama de identificadores reservados
  - 20000000 - 3FFFFFFF atribuídos ao utilizador
  - Ver detalhes em:
    - http://docs.sun.com/app/docs/doc/816-1435/6m7rrfn76?a=view

Sun RPC - Exemplo de Definição de Interface

```
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;
        Data READ(readargs) = 2;
   } = 2;
} = 0x20000100;
```

- Serviço de Acesso a Ficheiros Remotos:
  - Program FILEREADWRITE id = 0x20000100
  - Version 2
- Fornece dois métodos:
  - Write id = 1
  - Read id = 2
- Argumentos
  - Definidos por estruturas
  - Sintaxe linguagem C
- Ver mais exemplos em
  - http://netlab.ulusofona.pt/cd/praticas/rpc

← Identificador do Método

← Identificador do Serviço

Exemplo XDR

- Definição da estrutura *person*
- Utilização do utilitário *rpcgen*
- Geração automática do código de *marshalling*

```
typedef string nametype<MAXNAMELEN>;

struct person {
     nametype name;
     nametype place;
     int year;
};
```

```
#include "person.h"

bool_t
xdr_nametype (XDR *xdrs, nametype *objp)
{
    register int32_t *buf;

     if (!xdr_string (xdrs, objp, MAXNAMELEN))
         return FALSE;
    return TRUE;
}

bool_t
xdr_person (XDR *xdrs, person *objp)
{
    register int32_t *buf;

     if (!xdr_nametype (xdrs, &objp->name))
         return FALSE;
     if (!xdr_nametype (xdrs, &objp->place))
         return FALSE;
     if (!xdr_int (xdrs, &objp->year))
         return FALSE;
    return TRUE;
}
```

Exemplo Sun RPC – versão RPC – Interface e Servidor

```
/*
 * calc.x: Remote caculator interface
     definition
 * Implementes sum, substraction,
 * multiplication, division and modulo
     operations
 */

struct calcargs {
     float a;
     float b;
};

program REMOTECALC {
        version CALCVERS {
                float ADD(calcargs) = 1;
                float SUB(calcargs) = 2;
                float MUL(calcargs) = 3;
                float DIV(calcargs) = 4;
                float MOD(calcargs) = 5;
        } = 1;
} = 0x20000100;
```

```
/* calc.h Please do not edit this file.
 * It was generated using rpcgen.  */
#include <rpc/rpc.h>
struct calcargs {
     float a;
     float b;
};
typedef struct calcargs calcargs;
#define REMOTECALC 0x20000100
#define CALCVERS 1
-------------------------------------------

/* calc_server.c
 * This is sample code generated by rpcgen.v*/
#include "calc.h"

float * add_1_svc(calcargs *argp, struct
     svc_req *rqstp)
{
     static float  result;
     /*
      * insert server code here
      */
     result = argp->a + argp->b;
     return &result;
}
```

Exemplo Sun RPC – versão RPC – Cliente

```
/*
 * This is sample code generated by rpcgen.
*/
#include "calc.h"

void remotecalc_1(char *host)
{
     CLIENT *clnt;
     float  *result_1;
     calcargs  add_1_arg;

     clnt = clnt_create (host, REMOTECALC,
     CALCVERS, "udp");
     if (clnt == NULL) {
          clnt_pcreateerror (host);
          exit (1);
     }
     add_1_arg.a=1.0; add_1_arg.b=2.0;
     result_1 = add_1(&add_1_arg, clnt);
     if (result_1 == (float *) NULL) {
          clnt_perror (clnt, "call
     failed");
     }
     printf ("Result=%f\n", *result_1)
     clnt_destroy (clnt);
}
```

```
int
main (int argc, char *argv[])
{
     char *host;

     if (argc < 2) {
          printf ("usage: %s server_host\n",
     argv[0]);
          exit (1);
     }
     host = argv[1];
     remotecalc_1 (host);
     exit (0);
}
```

Exemplo Sun RPC – versão RPC – Stub cliente

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include <memory.h> /* for memset */
#include "calc.h"

/* Default timeout can be changed using
     clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

float *
add_1(calcargs *argp, CLIENT *clnt)
{
     static float clnt_res;

     memset((char *)&clnt_res, 0,
     sizeof(clnt_res));
     if (clnt_call (clnt, ADD,
          (xdrproc_t) xdr_calcargs,
     (caddr_t) argp,
          (xdrproc_t) xdr_float, (caddr_t)
     &clnt_res,
          TIMEOUT) != RPC_SUCCESS) {
          return (NULL);
     }
     return (&clnt_res);
}
```

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "calc.h"

bool_t
xdr_calcargs (XDR *xdrs, calcargs *objp)
{
     register int32_t *buf;

      if (!xdr_float (xdrs, &objp->a))
           return FALSE;
      if (!xdr_float (xdrs, &objp->b))
           return FALSE;
     return TRUE;
}
```

Exemplo Sun RPC – versão RPC – programa servidor

```
int
main (int argc, char **argv)
{
     register SVCXPRT *transp;

     pmap_unset (REMOTECALC, CALCVERS);

     transp = svcudp_create(RPC_ANYSOCK);
     if (transp == NULL) {
          fprintf (stderr, "%s", "cannot
     create udp service.");
          exit(1);
     }
     if (!svc_register(transp, REMOTECALC,
     CALCVERS, remotecalc_1, IPPROTO_UDP))
     {
          fprintf (stderr, "%s", "unable to
     register (REMOTECALC, CALCVERS,
     udp).");
          exit(1);
     }
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
if (transp == NULL) {
          fprintf (stderr, "%s", "cannot create tcp
service.");
          exit(1);
}
if (!svc_register(transp, REMOTECALC, CALCVERS,
remotecalc_1, IPPROTO_TCP)) {
          fprintf (stderr, "%s", "unable to register
(REMOTECALC, CALCVERS, tcp).");
          exit(1);
}

svc_run ();
fprintf (stderr, "%s", "svc_run returned");
exit (1);
/* NOTREACHED */
}
```

Exemplo Sun RPC – versão RPC – Stub servidor

```
/* Please do not edit this file.
 * It was generated using rpcgen.  */
#include "calc.h"
static void
remotecalc_1(struct svc_req *rqstp,
     register SVCXPRT *transp)
{
     union {
          calcargs add_1_arg;
          calcargs sub_1_arg;
          calcargs mul_1_arg;
          calcargs div_1_arg;
          calcargs mod_1_arg;
     } argument;
     char *result;
     xdrproc_t _xdr_argument, _xdr_result;
     char *(*local)(char *, struct svc_req
     *);

     switch (rqstp->rq_proc) {
     case NULLPROC:
     (void) svc_sendreply (transp,
     (xdrproc_t) xdr_void, (char *)NULL);
     return;
```

```
case ADD:
     _xdr_argument = (xdrproc_t) xdr_calcargs;
     _xdr_result = (xdrproc_t) xdr_float;
     local = (char *(*)(char *, struct svc_req *)) add_1_svc;
     break;
case SUB:
     break;
case MUL: /* code for MUL */
     break;
case DIV: /* code for DIV */
     break;
case MOD:
default:
     svcerr_noproc (transp);
     return;
}
memset ((char *)&argument, 0, sizeof (argument));
if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t)
&argument)) {
          svcerr_decode (transp);
          return;
}
result = (*local)((char *)&argument, rqstp);
if (result != NULL && !svc_sendreply(transp, (xdrproc_t)
_xdr_result, result)) {
          svcerr_systemerr (transp);
}
if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument,
(caddr_t) &argument)) {
          fprintf (stderr, "%s", "unable to free arguments");
          exit (1);
}
return;
```

Exemplo Sun RPC – parâmetros complexos

```
/*
 * dir.x: Remote directory listing protocol
 */
const MAXNAMELEN = 255;
typedef string nametype<MAXNAMELEN>; /* directory
     entry */
typedef struct namenode *namelist; /* link in the
     listing */
struct namenode { /* A node in the directory
     listing */
     nametype name; /* name of directory entry */
     namelist next; /* next entry */
};
/* The result of a READDIR operation
 * The union is used
 * here to discriminate between successful
 * and unsuccessful remote calls.
 */
union readdir_res switch (int errno) {
     case 0:
          namelist list; /* no error: return
     directory listing */
     default:
          void; /* error occurred: nothing else
     to return */
};
```

```
/* The directory program definition */

program DIRPROG {
     version DIRVERS {
          readdir_res READDIR(nametype) = 1;
     } = 1;
} = 0x20000076;
```

Perguntas a que devo ser capaz de responder

- Quais são as componentes do SUN RPC ?
- Quais são as principais diferenças entre o IDL Sun RPC e a linguagem C ?
- Que código produz o compilador de IDL Sun RPC ?
- O cliente ou o servidor de Sun RC pode estar escrito noutra linguagem de programação que não C ?

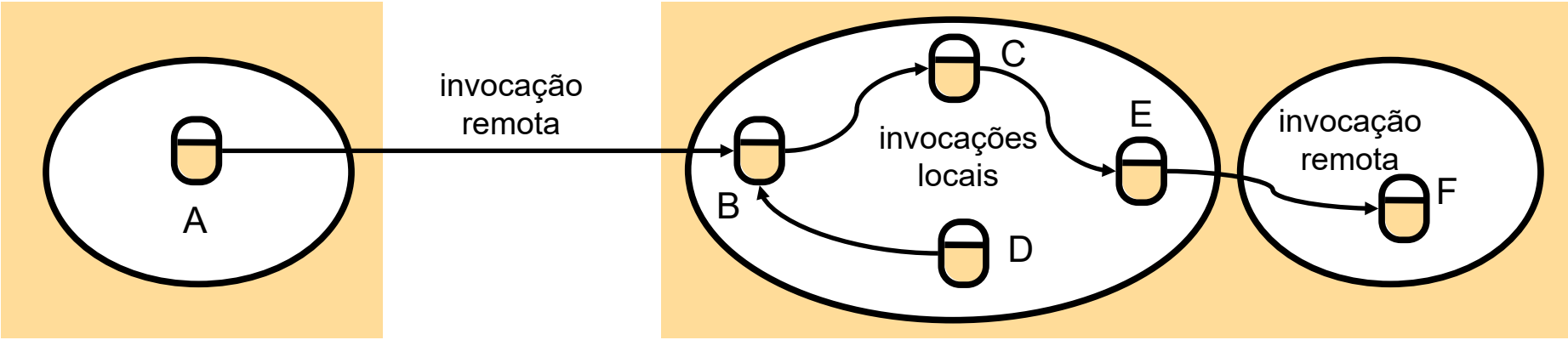
RMI - Invocação Remota de Objectos

- Extensão do modelo a objectos não residentes no mesmo processo
  - Utilização de um Mecanismo de Invocação Remota de Métodos
    - A acção de invocação do método remoto é inserida numa mensagem
    - RMI: ***Remote Method Invocation*** - Modelo Cliente / Servidor
  - A invocação pode também utilizar outros modelos
    - Objectos replicados ou migrantes
- A encapsulação em objectos aumenta a modularidade das funcionalidades dos serviços
  - Funcionalidades acedidas só através dos métodos predefinidos
  - Favorece a concorrência e isolamento de falhas

Modelo de Programação por Objectos

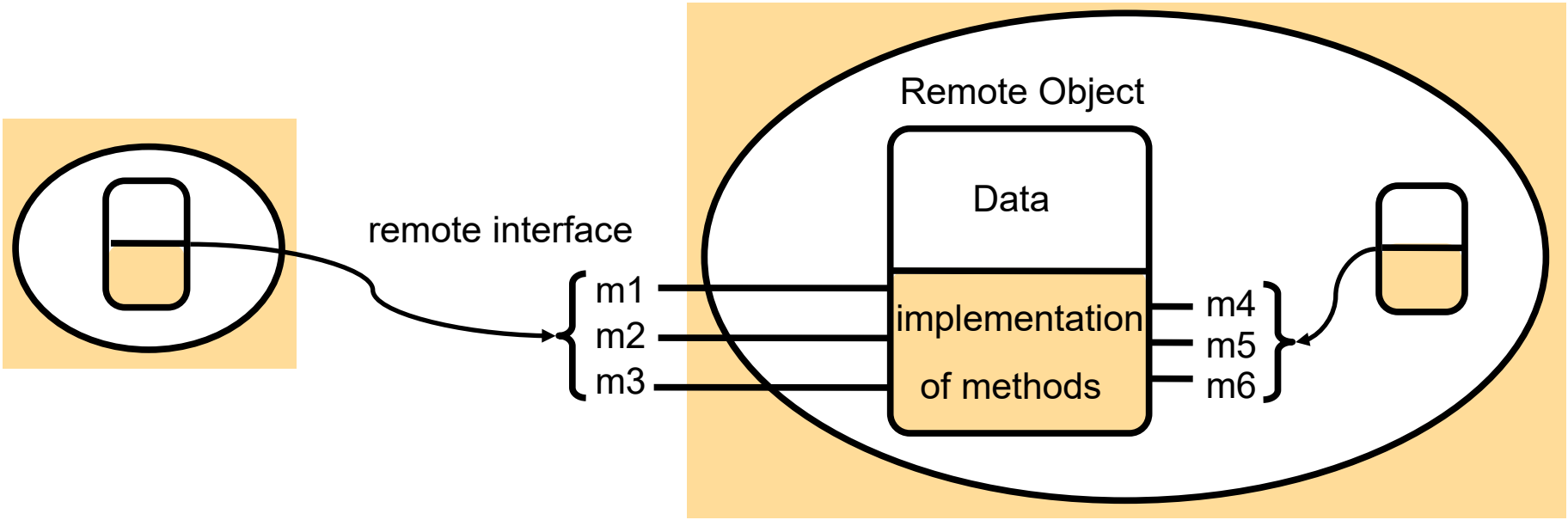
- Um **objecto** é uma entidade que encapsula dados e operações sobre os mesmos – ***métodos***
  - Corresponde à instanciação de uma ***classe***
- Os objectos são identificados e acedidos através de ***referências***
  - As referências podem ser afectadas a variáveis, passadas como parâmetros ou devolvidas pela execução de métodos
- Uma ***interface*** define a assinatura dos métodos de um objecto
  - Argumentos, tipos, valores de retorno e excepções
  - Uma classe pode implementar diversas interfaces
- A ***invocação*** de um método é uma ***acção*** que provoca uma mudança de estado no objecto invocado e o retorno de um valor
- A execução de um método pode provocar erros que geram ***excepções***
  - Forma elegante de gerir falhas sem ter de inserir testes em cada invocação
- A reciclagem de espaço memória libertada por objectos pode ser feita explícita ou implicitamente por ***Garbage Collection***

Modelo de Objectos Distribuídos



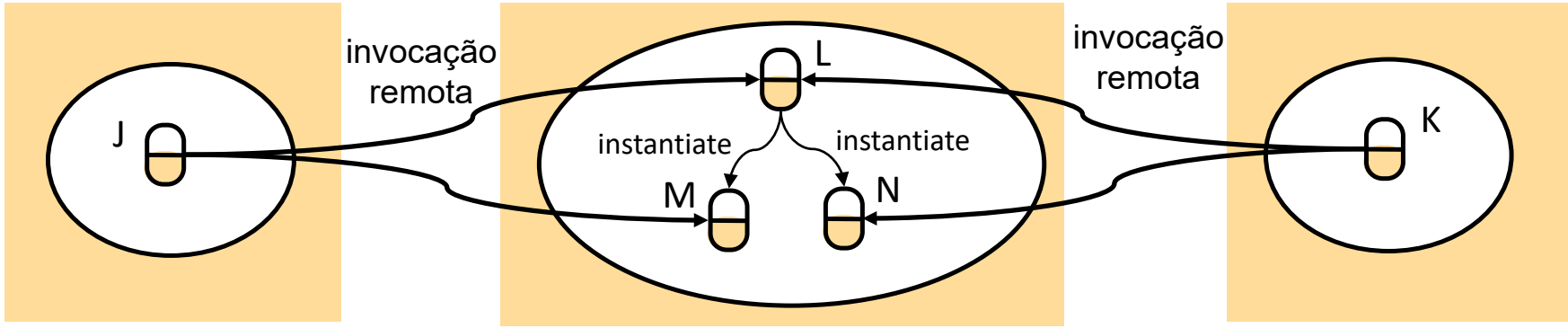
- O objecto A invoca um método do objecto remoto B
- Objectos remotos podem receber invocações locais ou remotas
  - Os processos podem residir ou não no mesmo servidor
- Para haver invocação remota é necessário
  - Possuir uma *referência* para o objecto remoto
    - Extensão da referência local de um objecto válida no contexto de um sistema distribuído
  - Conhecer a *definição* da interface do objecto remoto
    - Só podem ser invocados remotamente os métodos definidos na interface

Interfaces Remotas



- A classe que define um objecto remoto implementa os métodos da interface remota
- As interfaces são definidas através de uma linguagem de definição
  - ex.: Corba IDL
- Ou utilizando a própria linguagem de objectos
  - ex.: Java, por extensão da interface *remote*

Instanciação Remota



- Para que uma invocação seja efectuada, o objecto remoto tem de existir
  - Tem de ser instanciado no servidor e a sua referência exportada
- Uma invocação remota pode dar origem a uma instanciação de objecto
  - A classe pode ter um método construtor invocável remotamente
  - O objecto instanciado reside no servidor e pode ser invocável remotamente por outros objectos se a sua referência for passada ou exportada
- Os objectos remotos não referenciados devem ser libertados por *Garbage Collection* distribuído (se a linguagem o suportar)
- O mecanismo de RMI tem de garantir a propagação de excepções para o objecto que invoca remotamente para manter a mesma semântica da invocação local

Transparência

- O objectivo da invocação remota de métodos é esconder a natureza distribuída do sistema
  - Tornar transparente para o programador a utilização do protocolo de transporte e a linearização dos parâmetros de invocação
- Contudo não é desejável esconder totalmente a natureza remota da invocação
  - A distribuição introduz mais factores de falhas que a aplicação deve saber gerir
    - Excepções remotas
  - Existem grandes diferenças a nível temporal que podem condicionar o desempenho das aplicações
    - Latência do canal
    - Timeouts
- É portanto desejável que o facto de estar a invocar um método remoto não seja completamente escondido
  - Mesma sintaxe de invocação
  - Definição, encadeamento e mecanismo de invocação explícitos
  - Ex.: Java RMI
    - *Throws RemoteException*
    - *Implements Remote*

Java RMI

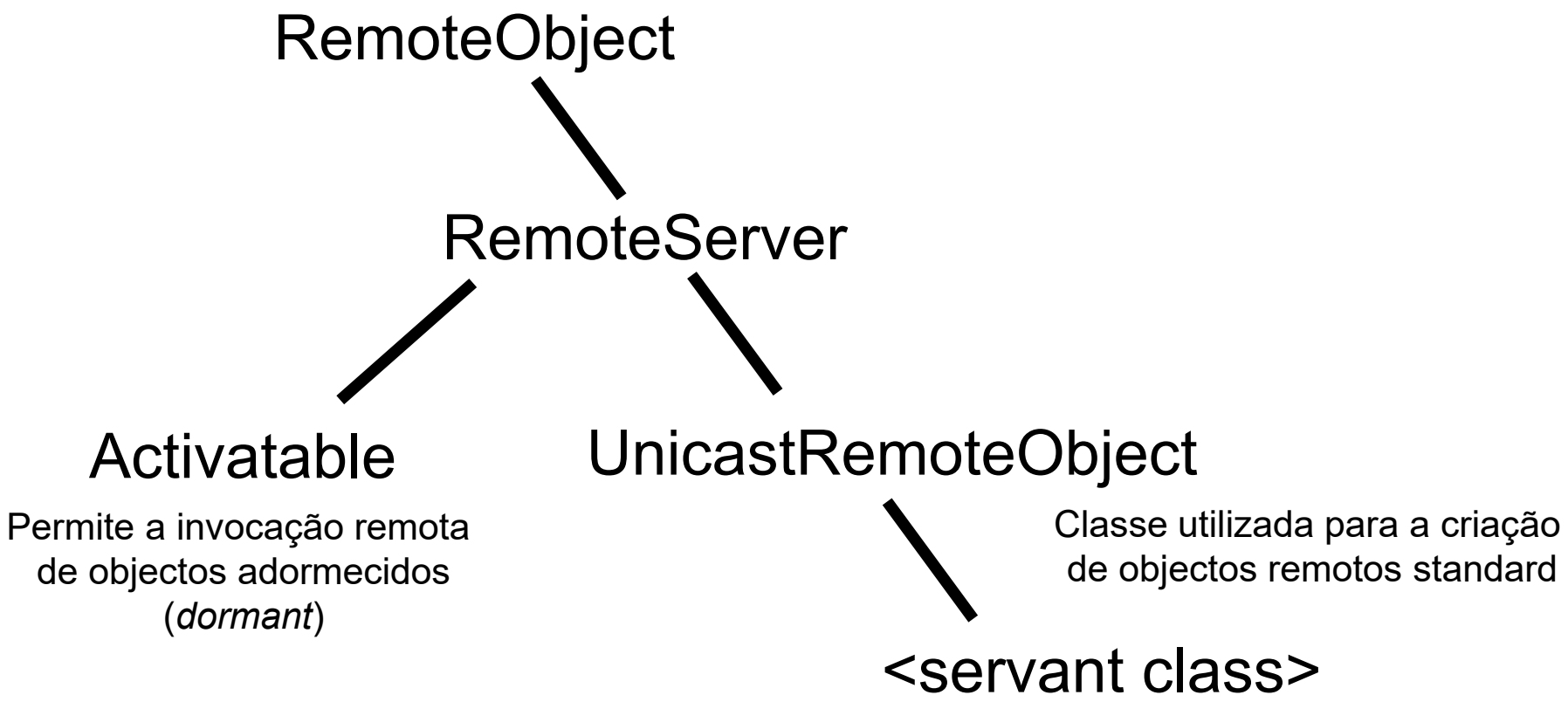
- Referência: Tutorial da Sun (Oracle) sobre RMI
  - http://docs.oracle.com/javase/tutorial/rmi/index.html
- Objectivo
  - Disponibilizar um modelo de objectos distribuídos naturalmente integrado com a linguagem de programação
- Princípios Básicos
  - Comportamento: definido por interfaces
  - Implementação: definida por classes

Interface e Implementação

- A interface deve ser definida como uma extensão da classe *Remote*
  - `public interface <Name> extends Remote`
  - Deve conter a assinatura de todos os métodos implementados
- Deve ser fornecida uma implementação para cada método
  - A classe de implementação do servidor implementa a interface

    `public class <NameClass> implements <Name>`
  - O stub do servidor é uma extensão de uma classe que permite instanciar um objecto remoto de tipo `UnicastRemoteObject`
  - O stub do cliente é um objecto que obedece à interface <Name>. O primeiro é obtido por consulta ao servidor de nomes, os outros podem ser obtidos como resultado da invocação de métodos remotos
- A implementação deve prever excepções de tipo remoto
  - `throws java.rmi.RemoteException`

Hierarquia de Classes Remotas



Nomeação de Objectos Remotos

- O registo e descoberta de serviços RMI é feita através de um serviço de directório chamado *RMI Registry*
  - Pode utilizar outros serviços de directório (JNDI -> LDAP)
- O *RMI Registry* (semelhante ao *Port Mapper*) regista e fornece informação sobre os serviços de um servidor
  - Acessível por TCP no porto 1099 (por defeito)
- *RMI Registry* acessível através da classe *Naming*
  - Formato do URL:

    ```
    rmi://<host_name> [:<name_service_port>] /<service_name>
    ```

  - Cliente: usa o método *lookup()* para procurar o serviço através do seu nome
  - Servidor: usa o método *rebind()* para registar o objecto remoto que implementa os métodos da interface

Exemplo: hello – interface e servidor

```
/*
 * Interface
 */
package example.hello;

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

public interface Hello extends Remote {
    String sayHello()
          throws RemoteException;
}
```

```
package example.hello;

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

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

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

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

      // Bind the remote object's stub in the registry
      Registry registry = LocateRegistry.getRegistry();
      registry.bind("Hello", stub);

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

Exemplo: hello - cliente

```
package example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

          String host = (args.length < 1) ? null : args[0];
          try {
              Registry registry = LocateRegistry.getRegistry(host);
              Hello stub = (Hello) registry.lookup("Hello");

              String response = stub.sayHello();

              System.out.println("response: " + response);
          } catch (Exception e) {
              System.err.println("Client exception: " + e.toString());
              e.printStackTrace();
          }
    }
}
```

Implementação do Cliente

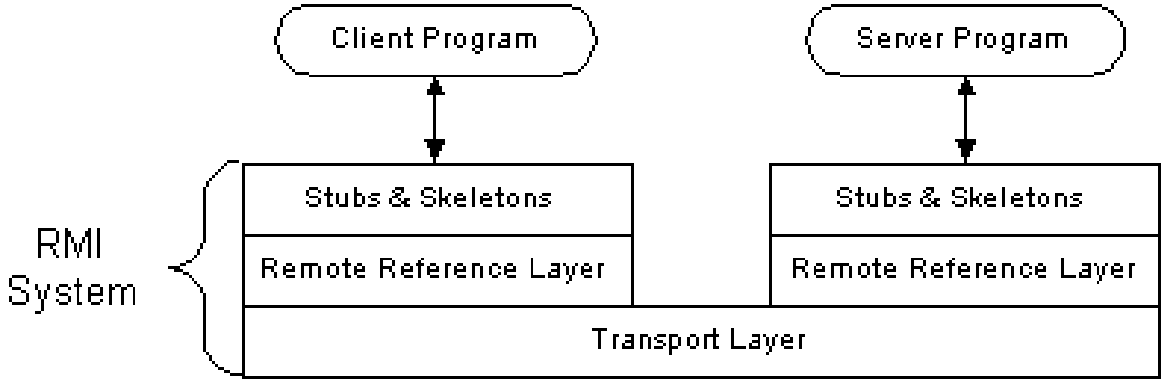
- O cliente tem de incluir as classes de resolução de nomes
  - `import java.rmi.Naming`
- Deve inquirir o serviço para obter uma referência para o objecto remoto
  - `Ref = Naming.lookup(rmi://<host_name>[:<name_service_port>]/<service_name>`
- A partir da referência obtida, o cliente pode invocar os métodos do objecto remoto definidos na interface como se fossem locais
  - `Ex: stub.sayHello();`

Implementação do Servidor e Execução da Aplicação

- O servidor instancia a classe de implementação e regista o objecto criado no *RMI Register*
  - `Naming.rebind("rmi://<host>/<Service>", impl_obj)`
- Todas as classes devem ser compiladas
  - `javac File.java`
- O RMI Registry deve ser lançado
  - É necessário indicar onde estão armazenadas as classes compiladas
    - Variável de ambiente **`CLASSPATH`**
  - `rmiregistry`
- O servidor pode então ser lançado
  - Regista-se junto do RMIRegistry
  - O cliente pode invocar os seus serviços

RMI: Arquitectura

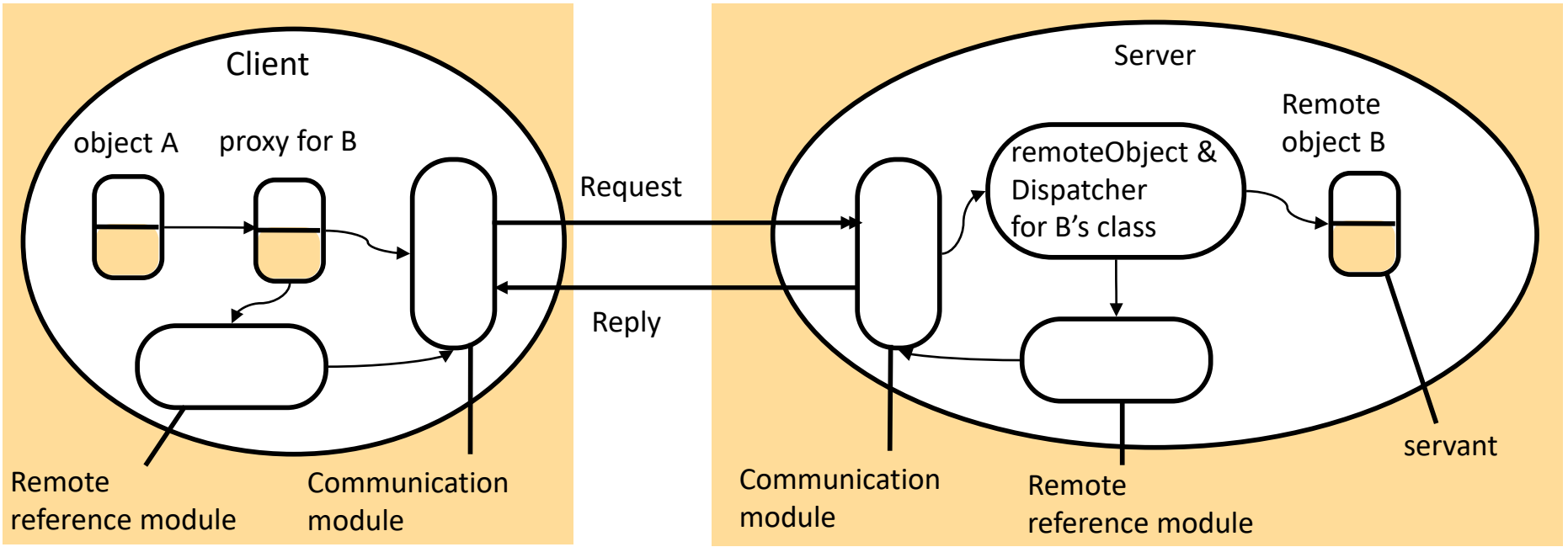
- A arquitectura do RMI é baseada em 3 níveis de abstracção
  - Nível *Stubs & remoteObject*: realiza a interface com os módulos servidor e cliente da aplicação
  - Nível *Remote Reference*: realiza a correspondência e ligação entre referências de objectos remotos e locais
    - Inicialmente uma ligação ponto a ponto
    - Na versão Java 2 suporta activações de objectos adormecidos (*dormant*)
  - Nível Transporte
    - Baseado em ligações TCP persistentes



Stubs & remoteObjects

- O Proxy representa o objecto invocado do lado do cliente
  - Fornece uma interface idêntica à que é implementada no servidor
  - Realiza a serialização dos argumentos e dos resultados
- O *remoteObject* é uma classe auxiliar do lado do servidor
  - Recebe as invocações, recupera os argumentos e invoca o método da implementação através do *dispatcher*
  - O dispatcher é genérico para todas as classes remotas

Implementação do RMI



- *Proxy*
  - Representação local do objecto remoto (≡ *client stub rpc*)
- *remoteObject (eram Skeletons)*
  - Classe de invocação do método remoto (≡ *server stub rpc*)
- *Dispatcher*
  - Cada classe tem um *dispatcher* que selecciona o método a invocar a partir do identificador
- *Remote Reference Module*
  - Realiza o mapeamento entre referências de objectos locais e remotos, contendo tabelas ligando os *proxies* aos *remoteObject*
- Os *proxy, remoteObject* e *class dispatcher* são gerados pelo compilador de interfaces

Funcionalidades Adicionais

- Invocação Dinâmica de Interfaces (DII)
  - Permite invocar objectos cujas interfaces são desconhecidas *a priori*
  - Utilização directa da operação de invocação de operação remota (callRemote) e da construção dinâmica da mensagem a partir da definição da interface
    - A definição da interface pode ser obtida dinamicamente por *reflection* em Java RMI ou através do *Interface Repository* em Corba.
- Activação de Objectos Remotos
  - Possibilidade de manter objectos "adormecidos" e de os reactivar quando são invocados, no estado em que foram guardados (*snapshot*)
  - Os objectos adormecidos são designados por persistentes e armazenados serializados em repositórios de objectos
- *Garbage Collection* Distribuída
  - Garante que se não houver referências remotas para um objecto os seus recursos são reciclados
  - Envolve a contabilização das referências de objectos mantidas nos proxies de clientes remotos
  - Utiliza a noção de sessão, finda a qual a referência é reciclada

Perguntas a que devo ser capaz de responder

- Porque é que o modelo de invocação remota de objetos se adequa muito bem à arquitetura cliente-servidor ?
- Em que é que a invocação remota de objetos pode diferir significativamente do RPC ?
- Qual é o IDL do Java para RMI ? Como se definem as interfaces remotas ?
- Que elementos do Java RMI têm as funções equivalentes aos stubs do RPC ?
- Porque é que a invocação remota de objetos pode conduzir à necessidade de garbage collection distribuída ?