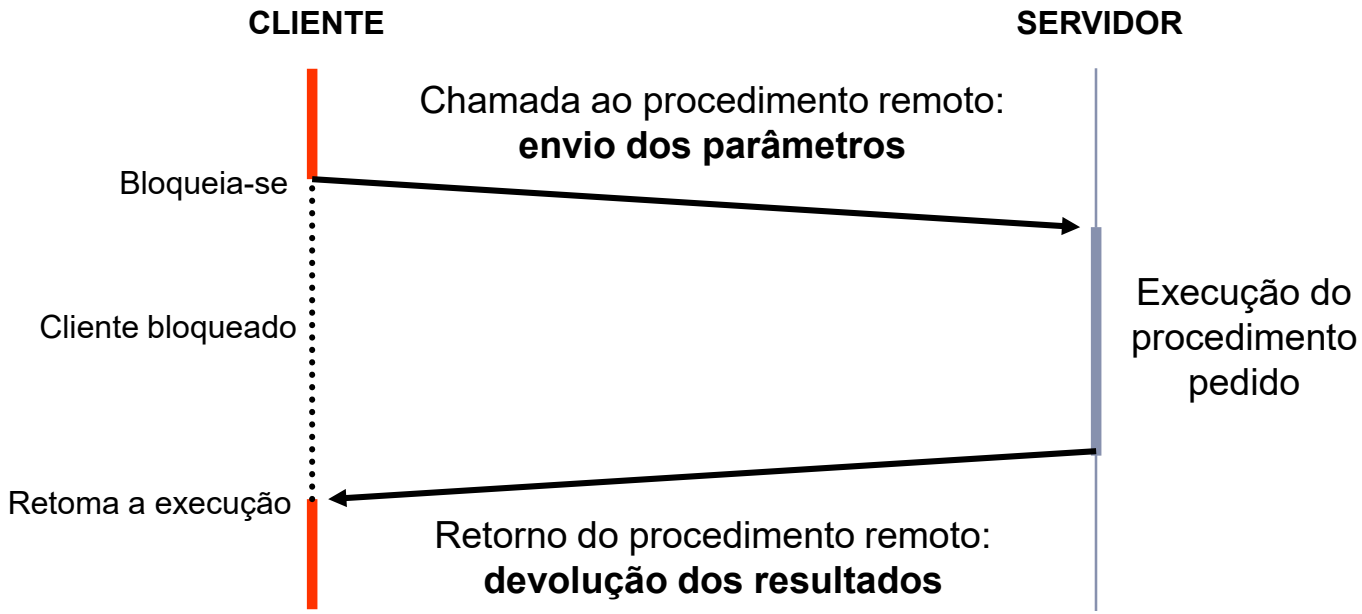
Capítulo 3:
Chamadas de Procedimentos Remotos

Chamada de Procedimentos Remotos
RPC - Remote Procedure Call

- Modelo de programação da comunicação num sistema cliente-servidor
- Objectivo:

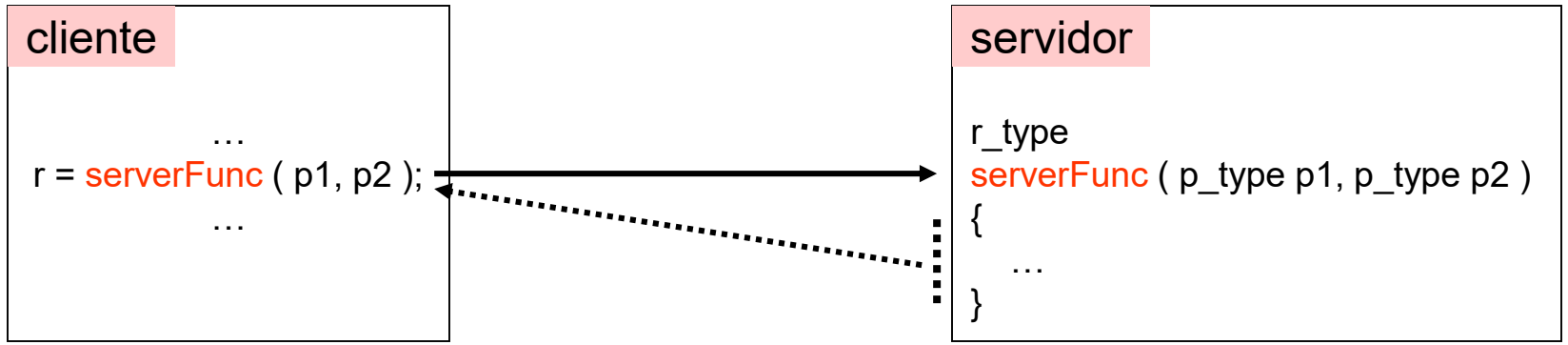
  Estruturar a programação distribuída com base na chamada pelos clientes de procedimentos que se executam remotamente no servidor

RPC: Fluxo de execução



Chamada de um Procedimento Remoto

- 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



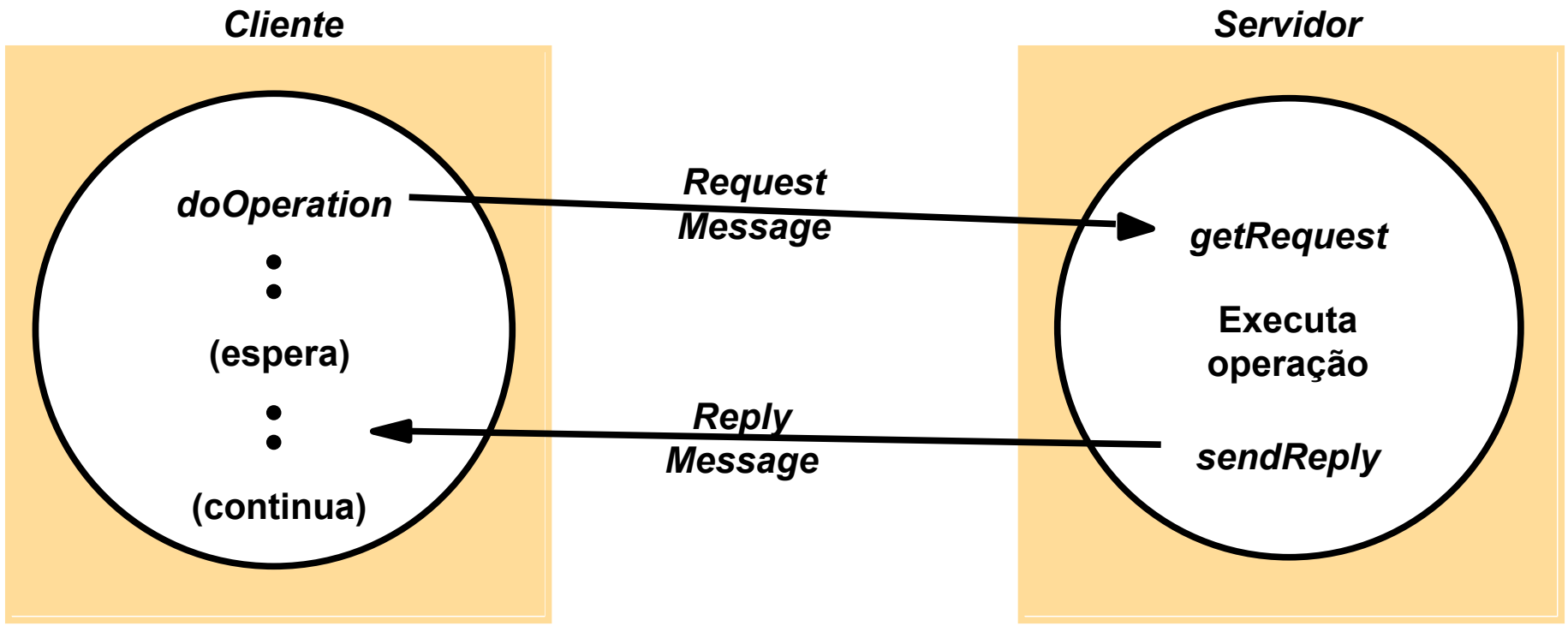
Projecto hipotético

- Implementar servidor de contagem que mantém contador e oferece estas operações aos clientes:
  - Colocar contador a zero
  - Avançar o contador em *x* unidades
  - Consultar valor atual do contador
- Requisitos adicionais:
  - Rede não é fiável
    - As mensagens podem perder-se, etc.
  - Sistema heterogéneo
    - O servidor e os clientes representam os números inteiros de forma diferente

Tentemos implementar isto com sockets...

Como executar cada operação?

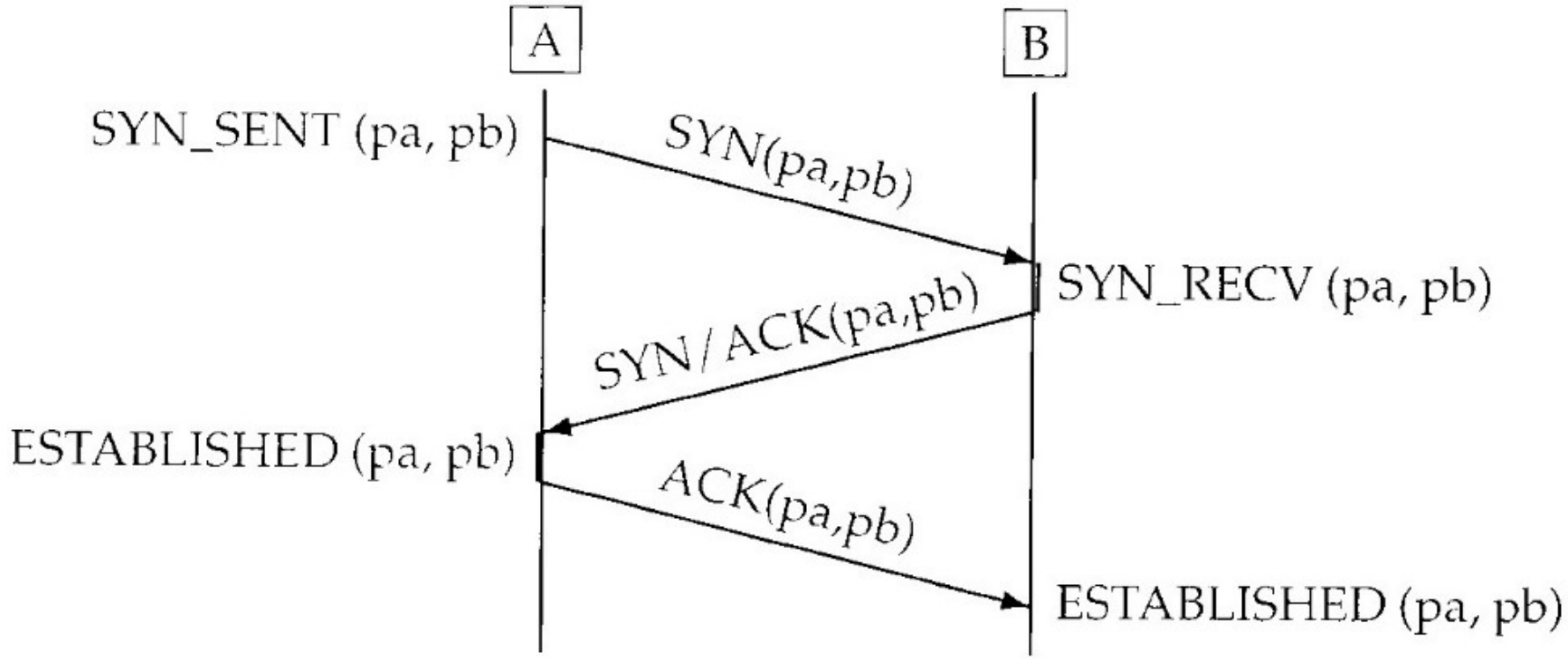
Como executar cada operação?
Protocolo RR – Request Reply



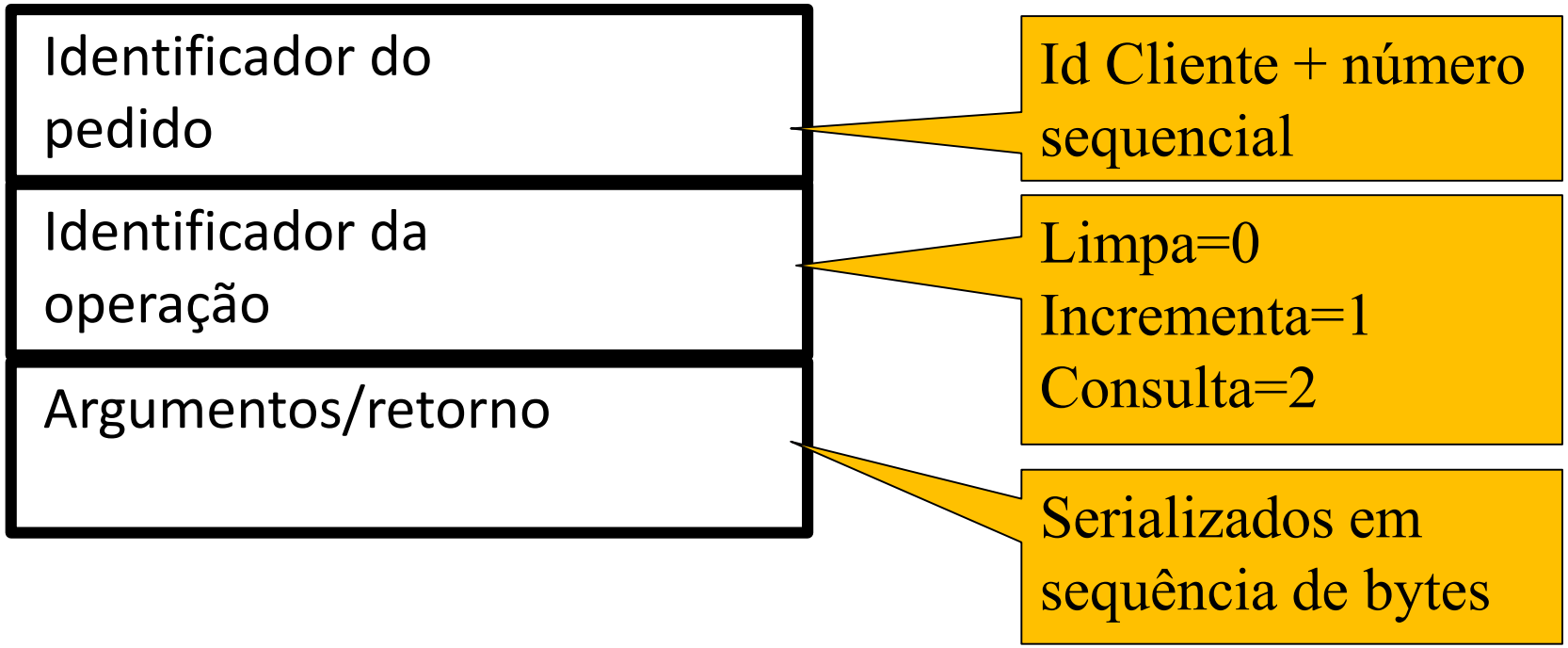
Vamos escolher TCP ou UDP?

- Vantagens do TCP
  - Oferece canal fiável sobre rede não fiável
- Mas talvez seja demasiado pesado. Porquê?
  - Para cada invocação remota passamos a precisar de mais 2 pares de mensagens
  - Gestão de fluxo é redundante para as invocações simples do nosso sistema
  - *Acknowledges* nos pedidos são desnecessários
- Logo optemos por UDP!

TCP connection



O que levam as mensagens de pedido/resposta?



Como serializar os argumentos/retorno?

- Necessário:
  - Converter estruturas de dados para sequência de bytes
  - Encontrar forma de emissor e receptor heterogéneos interpretarem dados correctamente
    - Por exemplo: serializar em “formato de rede”

h to n …

- The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to **"big-endian"** before sending them out. Since Intel is a "little-endian" machine, it is far more politically correct to call our preferred byte ordering "Network Byte Order". So these functions convert from your native byte order to network byte order and back again.
- Note that the types involved are 32-bit (4 byte, probably int) and 16-bit (2 byte, very likely short) numbers. 64-bit machines might have a htonll() for 64-bit ints, but I've not seen it. You'll just have to write your own.
- if you're converting from host (your machine's) byte order or from network byte order. If "host", the first letter of the function you're going to call is "h". Otherwise it's "n" for "network". The middle of the function name is always "to" because you're converting from one "to" another, and the penultimate letter shows what you're converting to. The last letter is the size of the data, "s" for short, or "l" for long.

  - `htons() - host to network short`
  - `htonl() - host to network long`
  - `ntohs() - network to host short`
  - `ntohl() - network to host long`

Este é um dos aspetos fundamentais da simplificação:
garantir que a heterogeneidade fica resolvida pela infra-estrutura

E se as mensagens forem maiores que um datagrama UDP?

- Podemos usar variante multi-pacote dos protocolos anteriores…
  - Implica implementar protocolo complicado
- Ou usar TCP!
  - Boa opção quando tamanho dos pedidos/respostas pode ser arbitrariamente grande
    - Exemplo: HTTP
  - Nesse caso, implementação é mais simples pois TCP já assegura fiabilidade da comunicação
  - Como evitar o custo de estabelecimento de ligação?
    - Exemplo: HTTP versão 1.1.

Modelo de faltas

- Usando UDP para enviar mensagens, estas podem:
  - Perder-se
  - Chegar repetidas
  - Chegar fora de ordem
- Processos podem falhar silenciosamente

- Como lidar com isto?

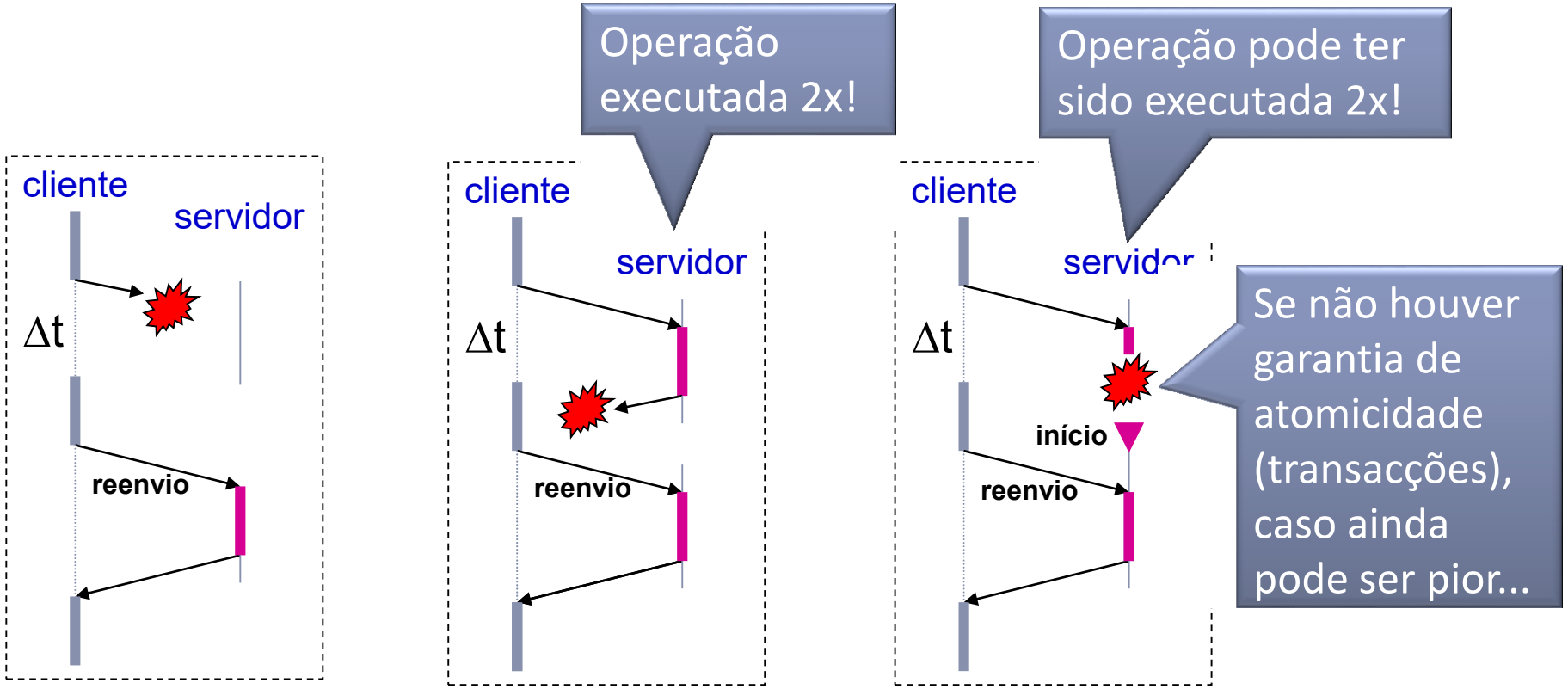
Temporizador no cliente

- Situação: cliente enviou pedido mas resposta não chega ao fim do *timeout*
- O que deve o cliente fazer?

- Hipótese 1: Cliente retorna erro.
  - Pouco interessante...
- Hipótese 2: Cliente re-envia pedido
  - Repete re-envio até receber resposta ou até número razoável de re-envios

Timeout no cliente com re-envio

- Quando a resposta chega após re-envio, o que pode ter acontecido?



Execuções repetidas do mesmo pedido: Qual o problema?

- Perde-se tempo desnecessário
- Efeitos inesperados se operação não for **idempotente**

> Operação que, se executada repetidamente,
> produz o mesmo resultado que se for executada só 1 vez

- Função *limpa* é idempotente?
- Função *incrementa* é idempotente?

Execuções repetidas do mesmo pedido: Como evitar?

- Servidor deve ser capaz de verificar se o identificador do pedido já foi recebido antes
- Se é a primeira vez, executa
- Se é pedido repetido?
  - Deve guardar história de respostas de pedidos executados
    - Tabela com (id. pedido, resposta)
  - E retornar a resposta correspondente

Quantos pedidos manter por cliente?

Como escalar para grande número de clientes?

RPC: 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

Esconde diversos detalhes do transporte

- Endereçamento do servidor
- Envio e recepção de mensagens
- Tratamento de erros

Simplifica a divulgação de serviços (servidores)

- A interface dos serviços é fácil de documentar e apresentar
- A interface é independente dos protocolos de transporte

RPC – Elementos constituintes

IDL Linguagem de Descrição de Interfaces

- Compilador e bibliotecas

Run-time para execução das funções de RPC genéricas

Stub routines para adaptarem cada procedimento

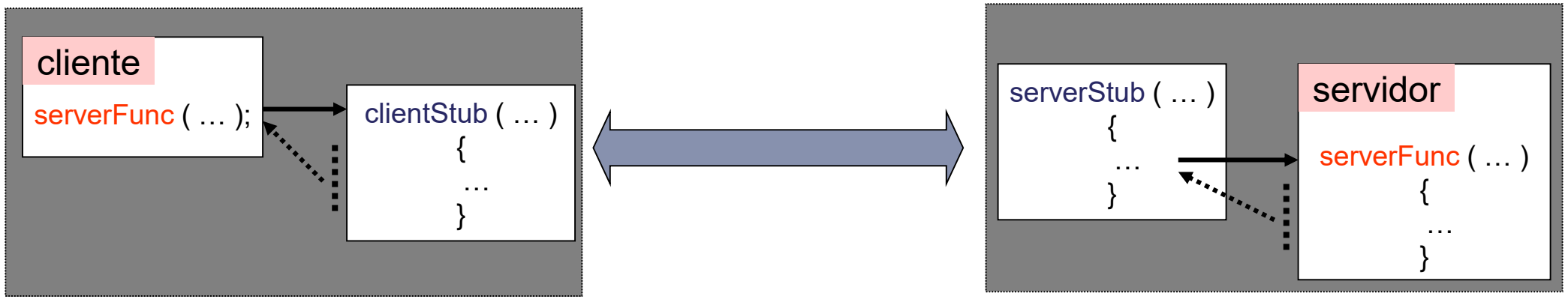
Gestor de Nomes

Estrutura do RPC

RPC: Rotinas de adaptação (*stubs*)
(também chamados *ties* do lado do servidor)

- 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)



Arquitetura

- Stubs - Rotinas de Adaptação
  - Efectuam as conversões de envio e recepção dos parâmetros da Chamada Remota
  - Cada rotina tem os seus parâmetros específicos pelo que cada uma tem um stub
  - Do lado do servidor existe um stub ou tie que executa as mesmas conversões pela ordem inversa
- Dispatch - 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
- Run-time system (RTS) - Sistema de Suporte
  - Executa as operações genéricas do RPC, por exemplo:
    Abrir a ligação com o servidor,
    efetuar o envio e receção das mensagens, fechar ligação
  - Como são operações genéricas existe apenas um RTS por cliente e um por servidor

A Visão da Programação do RPC

IDL - Linguagem de Descrição de Interfaces

RPC IDL: Características

- Linguagem própria
  - Linguagem declarativa (não tem a parte operacional das linguagens de programação)
- Permite definir
  - Tipos de dados
  - Protótipos de funções
    - Fluxo de valores (IN, OUT, INOUT)
  - Interfaces
    - Conjuntos de funções

IDL: Pode ser apenas um ".h"?

- Quais os parâmetros de entrada/saída da seguinte função?

```
int transfere(int origem, int destino, int valor,
  int *saldo, char *descr);
```

RPC IDL: 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 referencial dos dados enviados

Exemplo: Interface em C

```
resul = criar (long valor,char* nome,char* morada,long*
            numero)
resul = saldo (long nConta,long* valor);
resul = depositar (long nConta,long valor);
resul = levantar (long nConta,long valor);
resul = transferir (long nContaOrig,long nContaDest,long
            valor);
resul = pedirExtrato (long nConta,long mes,long ano,
            struct dadosOperacao* dados,int*
            nElementos);
```

RPC IDL: Soluções para alguns dos problemas

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

Exemplo: IDL
Sun RPC Ficheiro banco.x

```
program BANCOPROG {
  version BANCOVERS {
    criarRet          CRIAR(criarIn) = 1;
    saldoRet          SALDO(int) = 2;
    resultado         DEPOSITAR(contaEvalor) = 3;
    resultado         LEVANTAR(contaEvalor) = 4;
    resultado         TRANSFERIR(transferirIn) = 5;
    pedirExtratoRet   PEDIREXTRATO(pedirExtratoIn) = 6;
  } = 1;
} = 0x20000005;
```

Exemplo: Interface em IDL RPC Microsoft

```
[
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;
```

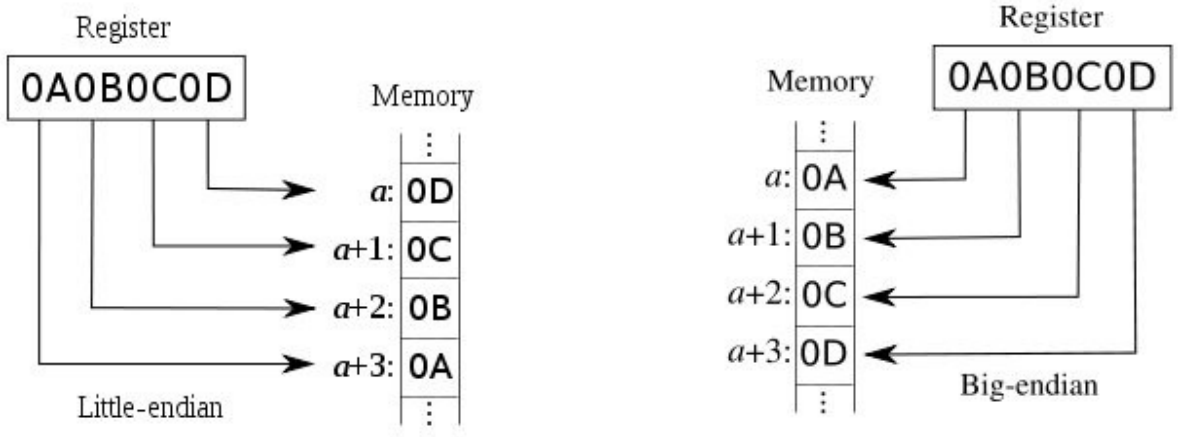
```
resultado criar([in] handle_t     h,
           [in] long         valor,
           [in, string] char nome[],
           [in, string] char morada[],
           [out] long        *numero);

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);
```

Protocolos e Serviços necessário para suportar o RPC

Heterogeneidade

Heterogeneidade

- Nos sistemas distribuídos a heterogeneidade é a regra
- Os formatos de dados são diferentes
  - Nos processadores (ex.: little endian, big endian, apontadores, vírgula flutuante)



  - Nas estruturas de dados geradas pelos compiladores
  - Nos sistemas de armazenamento (ex: strings ASCII vs Unicode)
  - Nos sistemas de codificação

Marshalling

- As redes transmitem bytes entre as máquinas
  - Necessário serializar objectos em memória para sequência de bytes
  - Des-serializar sequência de bytes para objectos na máquina destino
- Máquinas representam tipos de formas diferentes
  - É necessário traduzir entre representação de tipos do emissor e representação de tipos do receptor
- Marshalling: serializar + traduzir
  - Unmarshalling: operação inversa

Resolução da Heterogeneidade na Comunicação

- Modelo OSI → camada de Apresentação
  - Protocolo ASN.1
- Sistemas de RPC → aproveitam descrição formal da interface
  - heterogeneidade resolvida através de técnicas de compilação.
- A partir destes sistemas a heterogeneidade na comunicação ficou resolvida no ambiente de execução.

Protocolos de Apresentação no RPC - Decisões a efectuar

Estrutura das mensagens

- **Implícita** – as mensagens apenas contêm os dados a transmitir
- **Explícita** – Auto-descritiva (marcada, *tagged*)

Protocolos de Apresentação no RPC - Decisões a efectuar

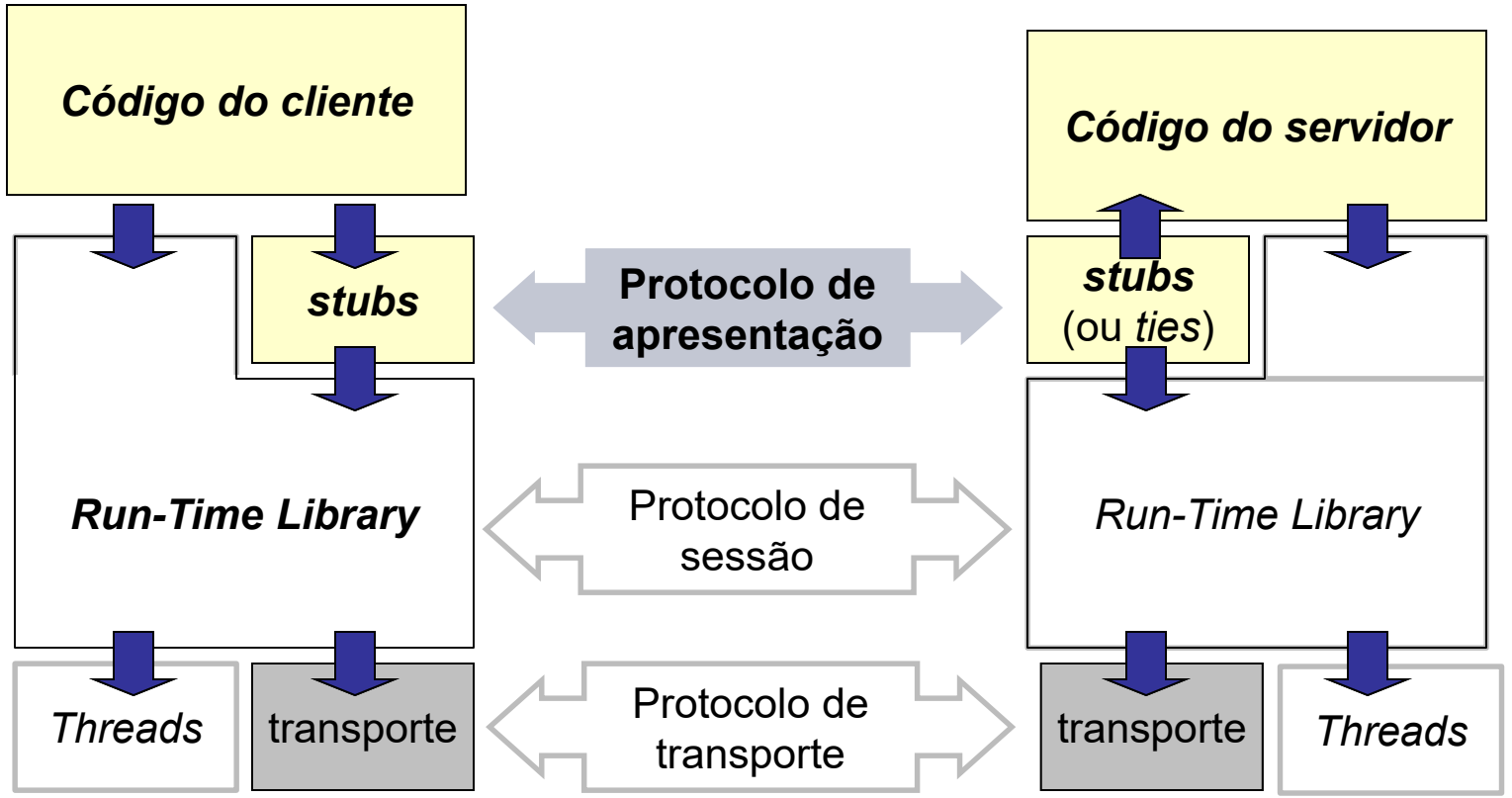
Políticas de conversão dos dados

- **Canónica** – Uma única representação para que todos convertem
  - N formatos ⇒ N funções
  - Não há comportamentos variáveis
  - É preciso converter mesmo quando é inútil
- **O-recetor-converte** (*Receiver-makes-it-right*)
  - Poupa conversões inúteis
  - N formatos ⇒ N x M funções (M arquitecturas)

Protocolos de Apresentação

	XDR (eXternal Data Representation)	NDR (Network Data Representation)	ASN.1 (Abstract Syntax Notation)	CDR (Common Data Representation)	Java Object Serialization	XML (Extensible Markup Language)
Tecnologia	Sun RPC	DCE RPC Microsoft RPC	OSI	CORBA	Java RMI	W3C
Conversão	Canónica	O-receptor-converte	Canónica	O-receptor-converte	Canónica	Canónica
Estrutura das msgs.	Implícita Binária Comprimentod e vectores variáveis Alinhamento a 32 bits (excepto vectores de caracteres)	Implícita Binária Marcas arquitecturais (architecture tags)	Explícita – Tagged Binária Encoding Rules: Basic Distinguished Canonical Packed	Implícita Binária	Explícita Binária	Explicita – Tagged Textual Tipos de Documentos DTD XML schema

Arquitetura do sistema de RPC:
Blocos funcionais das aplicações

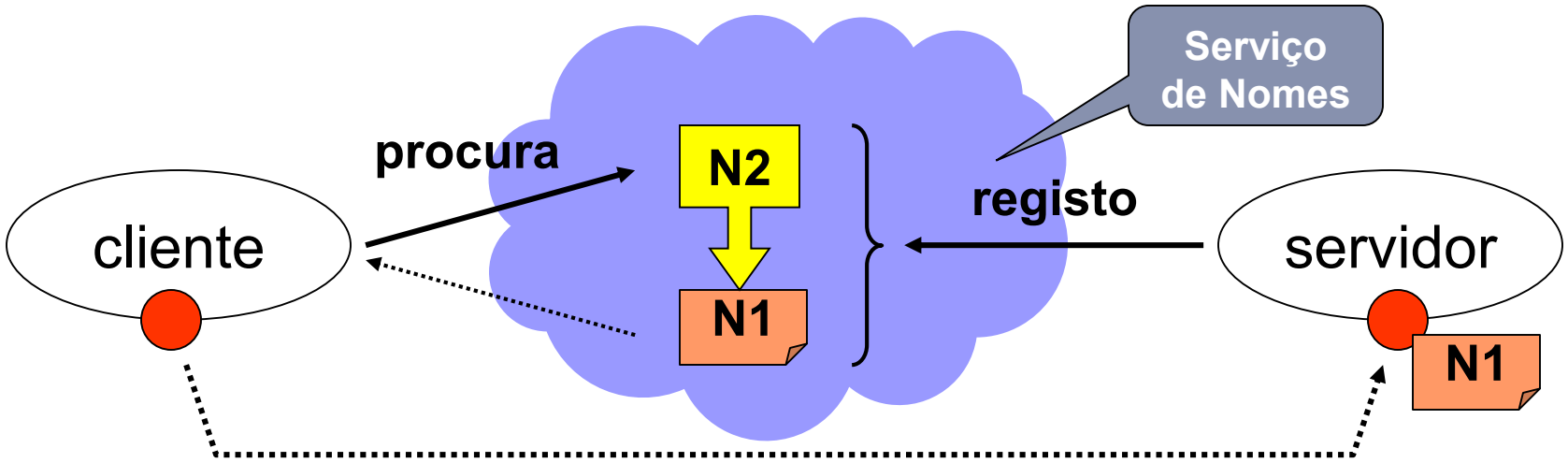


Vinculação entre o Cliente e o Servidor

Ligação ou Binding

RPC: Serviço de Nomes

- Permite que o servidor registe um nome de um serviço
  - Que tem de estar associado ao identificador de um porto de transporte
- Permite que um cliente consiga encontrar o servidor através do nome do serviço.
  - Obter o nome do seu porto de transporte



O registo tem de ser efectuado pelo servidor

- Registo
  - Escolha da identificação do utilizador
    - Nome do porto de transporte
    - Outros nomes alternativos
  - Registo dessa identificação
- {Esperar por pedidos de criação de sessões}*
  - Estabelecimento de um canal de transporte
  - Autenticação do cliente e/ou do servidor
- {Esperar por invocações de procedimentos}*
  - Enviados pelos clientes ligados
- Terminação da sessão
  - Eliminação do canal de transporte

O binding tem de ser efectuado pelo cliente

- Estabelecimento da sessão – vinculação ao servidor (binding)
  - Localização do servidor
  - Autenticação do cliente e/ou do servidor
  - Estabelecimento de um canal de transporte

{Chamada de procedimentos remotos}* - efetua os RPC necessários

- Terminação da sessão
  - Eliminação do canal de transporte

Referências de sessão – binding handles

- Cliente
  - Criação do *binding handle* no extremo cliente
    - Identifica um canal de comunicação ou um porto de comunicação para interatuar com o servidor
- Servidor
  - Possível criação de um *binding handle* no extremo servidor
    - Útil apenas se o servidor desejar manter estado entre diferentes RPCs do mesmo cliente
    - Um servidor sem estado não mantém *binding handles*

Exemplo Binding : Cliente – Sun RPC

```
void main (int argc, char *argv[]){
  CLIENT *cl;
  int a, *result;
  char* server;
  if (argc < 2) {
      fprintf(stderr, "Modo de Utili
         servidor\n", argv[0]);
      exit(1);
  }
  server = argv[1];
  cl = clnt_create(server, BANCOPROG, BANCOV
  if(cl == NULL) {
      clnt_pcreateerror(server);
      exit(1);
  }
  sresult = saldo_1(nconta, cl);
}
```

A função retorna um binding handle

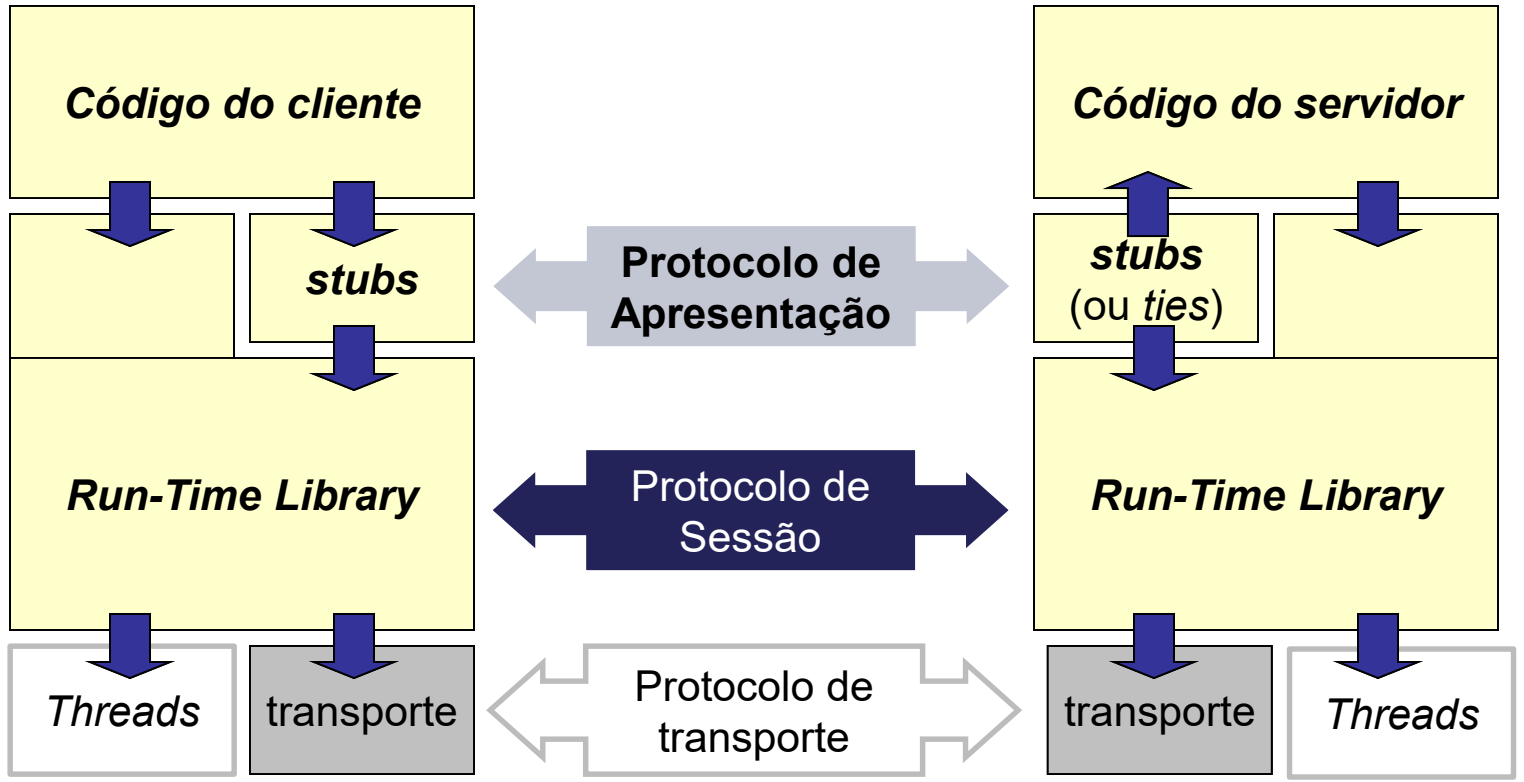
A chamada ao procedimento remoto explicita o binding handle

Ligação cliente-servidor

- Os *binding handles* podem ser usados pelos *stubs* de uma forma
  - Explícita
  - Implícita
  - Automática

	Explícito (Sun RPC, DCE RPC)	Implícito (DCE RPC)	Automático (DCE RPC)
Inicialização do Cliente	Obtém informação de ligação Usa-a explicitamente em cada RPC	Obtém informação de ligação Guarda-a numa variável global	(não necessária)
Stub cliente	A função de stub tem um parâmetro de entrada que especifica o handle a usar na chamada	Usa a variável global	Obtém informação de ligação Guarda-a localmente e usa-a

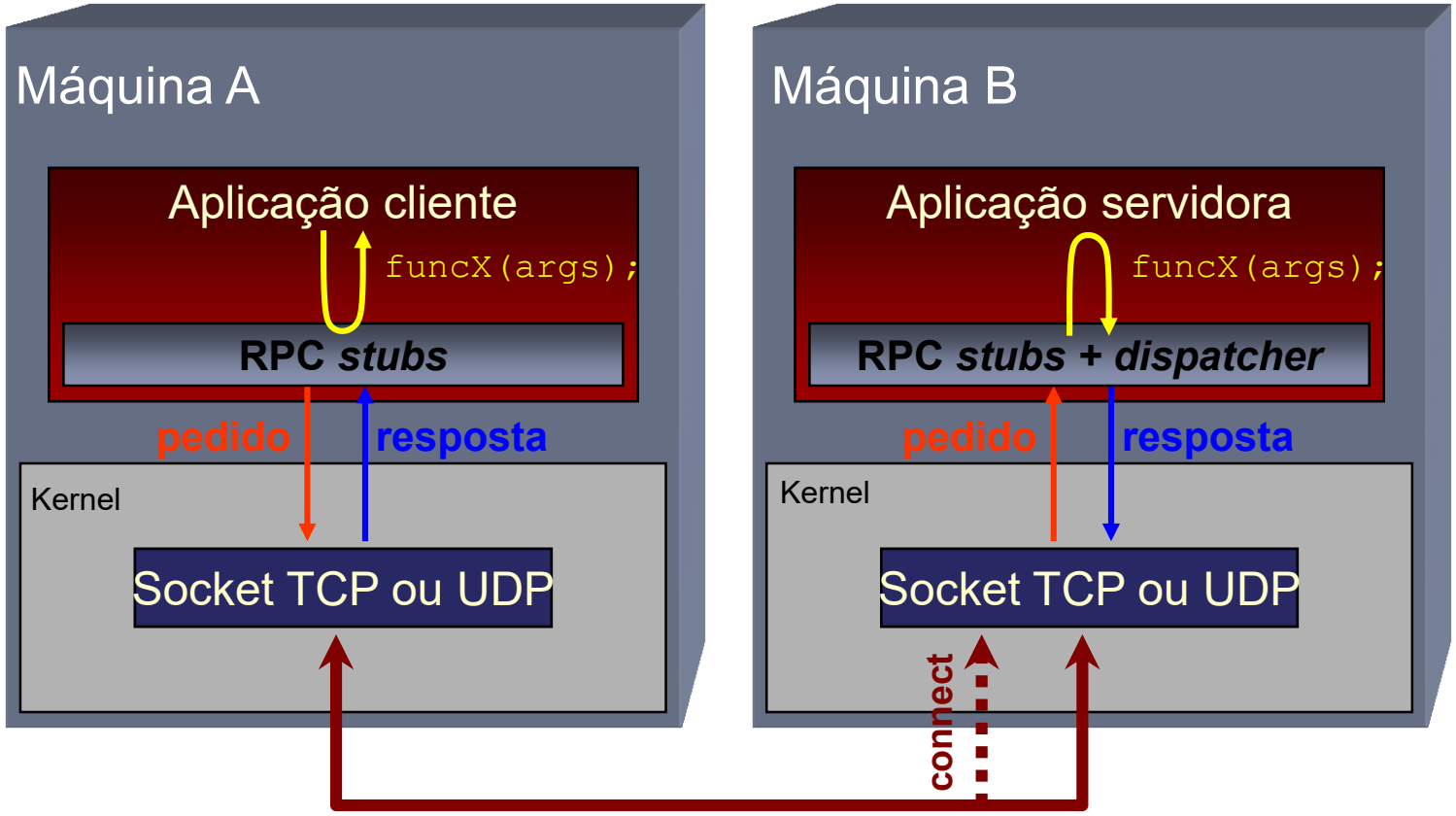
Arquitectura do sistema de RPC:
Blocos funcionais das aplicações



Exemplo de RPC: SUN RPC

- Sistema de RPC desenvolvido pela SUN cerca de 1985
- Destinado inicialmente a suportar o sistema de ficheiros distribuído NFS
- Especificação de domínio público
- Implementação simples e muito divulgada em grande número de plataformas (Unix, MS-DOS, …)

Objectivos


Máquina A
Aplicação cliente
funcX(args);
RPC stubs
pedido
resposta
Kernel
Socket TCP ou UDP
Máquina B
Aplicação servidora
funcX(args);
RPC stubs + dispatcher
pedido
resposta
Kernel
Socket TCP ou UDP
connect

SUN RPC

- Linguagem de IDL semelhante a C, suportada pelo compilador rpcgen
  - Apenas um parâmetro de entrada e um parâmetro de saída
  - Vários valores são transferidos em estruturas
  - Construção que permite transmitir condicionalmente informação
  - Todos os parâmetros são passados por referência para os stubs
- rpcgen gera automaticamente o programa principal do servidor
- Biblioteca de RPC inicialmente usava sockets, atualmente usa TLI

Sun RPC: Exemplo de IDL

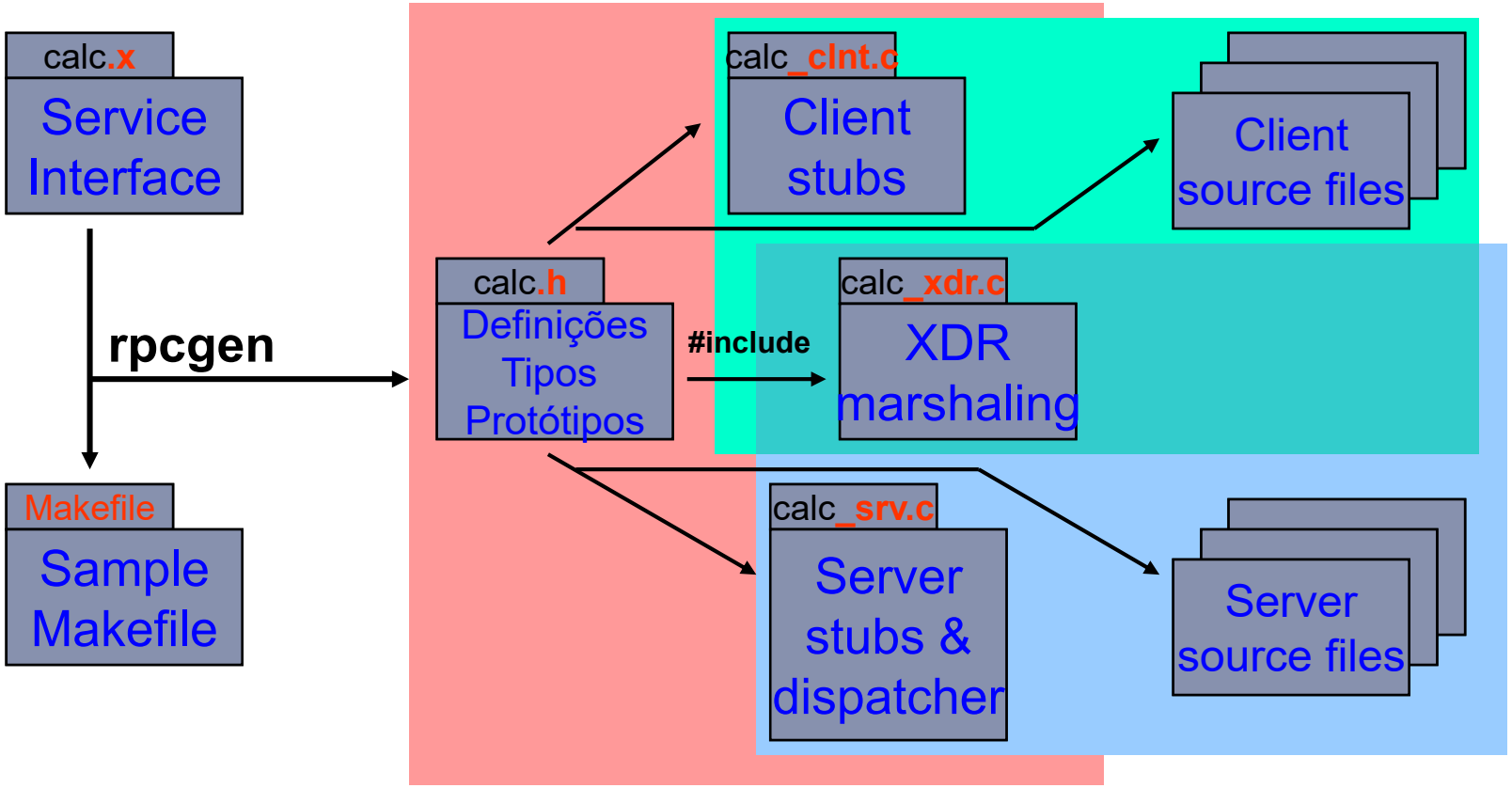
```
program BINOP {
    version BINOP_VERS {
        long BINOP_ADD ( struct input_args ) = 5;
    } = 1;
} = 300030;

struct input_args {
    long a;
    long b;
};
```

- Identificação (nomes)
  - Interface → (número programa, número versão)
    (300030, 1)
  - Função → (Interface, número função)
    (300030, 1, 5)

Sun XDR (eXternal Data Representation)

- Estrutura implícita
  - É suposto cliente e servidor conhecerem *a priori* os tipos dos parâmetros em cada invocação (do IDL)
- Representação binária, conversão canónica
  - Parâmetros serializados em sequências de blocos de 4 bytes
  - Inteiros, booleanos, etc: 1 bloco de 4 bytes
    - Convenção define qual byte de cada bloco é o mais significativo
  - Arrays, estruturas e cadeias de caracteres ocupam múltiplos blocos
    - Seguidos por bloco que indica o número de blocos
    - Existe uma convenção para definir qual das extremidades da cadeia de blocos corresponde ao primeiro
  - Caracteres em ASCII

Diagrama de ficheiros



rpcgen: definição e compilação da interface

calc.x

```
enum calc_error {
  CALC_OK = 0,              /* No error */
  CALC_ZERO_DIV = 1         /* Division by zero */
};

struct calc_args {
   double v1;
   double v2;
};

struct calc_result {
   calc_error error;
   double value;
};

program CALC_PROG {
   version CALC_VERS {
      calc_result SUM(calc_args) = 1;
      calc_result SUB(calc_args) = 2;
      calc_result MUL(calc_args) = 3;
      calc_result DIV(calc_args) = 4;
   } = 1;
} = 100400;
```

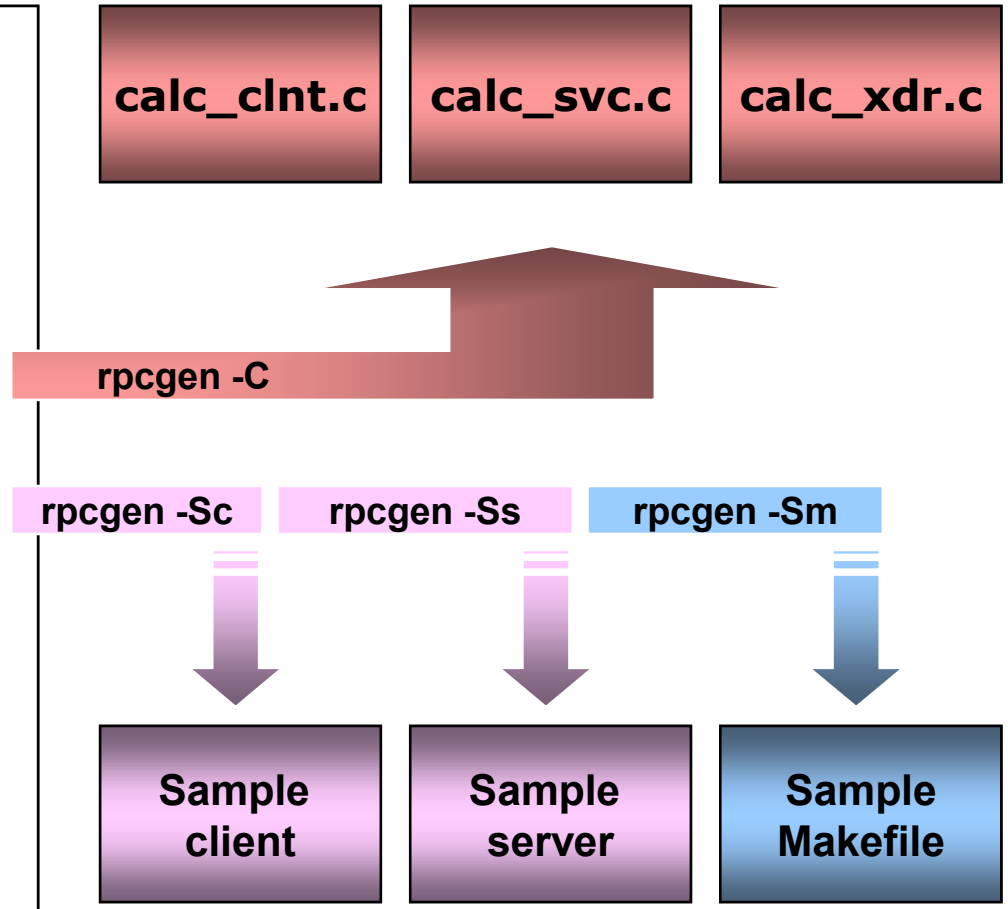
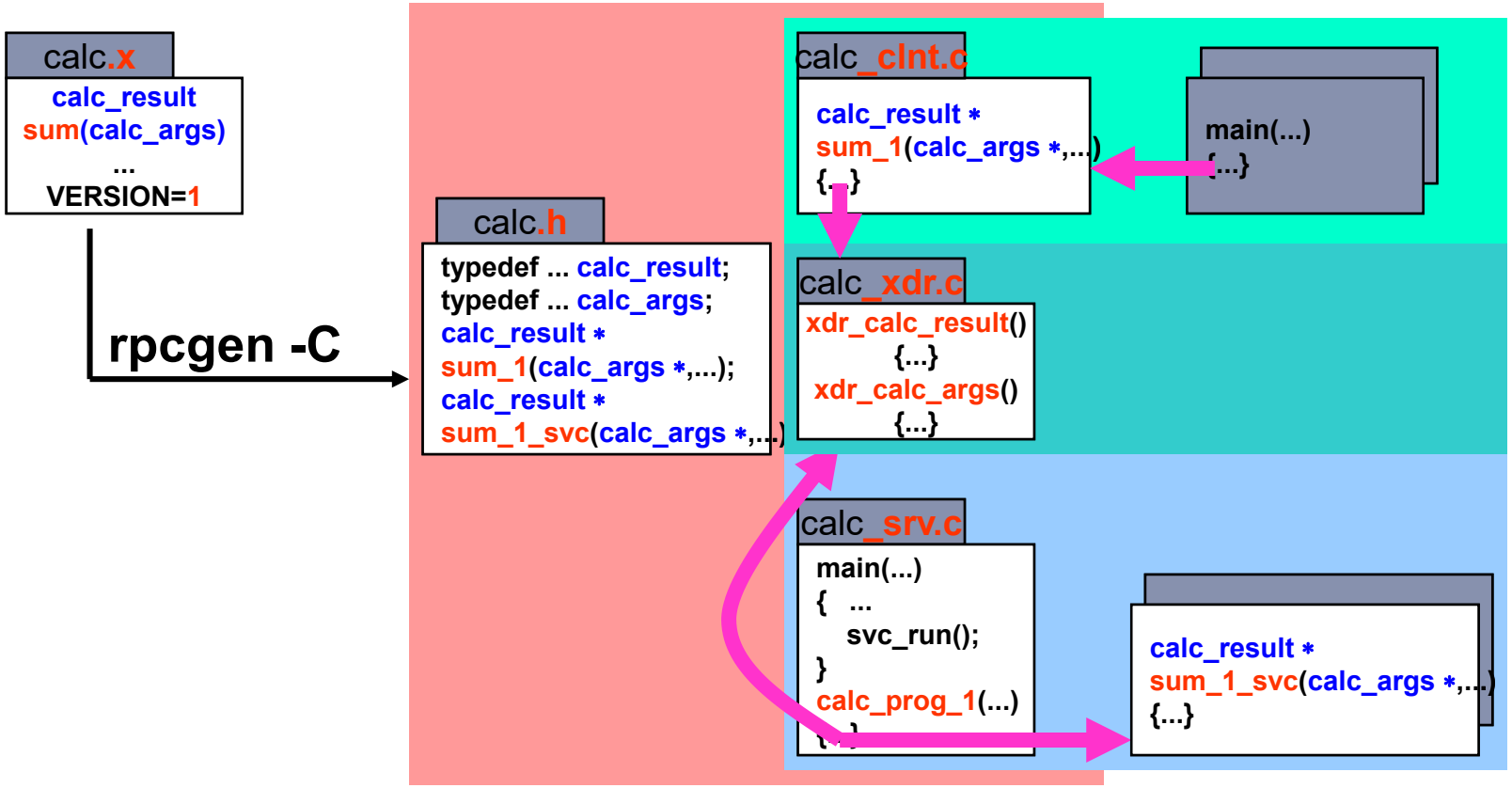


Diagrama de ficheiros (cont.)



Funções de conversão via XDR

calc.x

```
enum calc_error {
    CALC_OK = 0,          /* No error */
    CALC_ZERO_DIV = 1     /* Div. by zero */
};

struct calc_args {
    double v1;
    double v2;
};

struct calc_result {
    calc_error error;
    double value;
};

program CALC_PROG {
    version CALC_VERS {
        calc_result SUM(calc_args) = 1;
        calc_result SUB(calc_args) = 2;
        calc_result MUL(calc_args) = 3;
        calc_result DIV(calc_args) = 4;
    } = 1;
} = 100400;
```

rpcgen -C

calc_xdr.c

```
#include "calc.h"
bool_t
xdr_calc_error(XDR *xdrs, calc_error *objp)
{
   if (!xdr_enum(xdrs, (enum_t *)objp))
            return (FALSE);
     return (TRUE);
}
bool_t
xdr_calc_args(XDR *xdrs, calc_args *objp)
{...}
bool_t
xdr_calc_result(XDR *xdrs, calc_result *objp)
{
     if (!xdr_calc_error(xdrs, &objp->error))
            return (FALSE);
     if (!xdr_double(xdrs, &objp->value))
            return (FALSE);
     return (TRUE);
}
```

Funções de conversão para cada tipo definido no IDL

Chamadas a funções de conversão de tipos base (oferecidas na biblioteca run-time do SUN RPC)

Funções do cliente (*stubs*)

Função genérica de chamada de procedimento remoto (da biblioteca de run-time)

calc.x

```
enum calc_error {
    CALC_OK = 0,            /* No error */
    CALC_ZERO_DIV = 1       /* Division by zero */
};

struct calc_args {
    double v1;
    double v2;
};

struct calc_result {
    calc_error error;
    double value;
};

program CALC_PROG {
    version CALC_VERS {
        calc_result SUM(calc_args) = 1;
        calc_result SUB(calc_args) = 2;
        calc_result MUL(calc_args) = 3;
        calc_result DIV(calc_args) = 4;
    } = 1;
} = 100400;
```

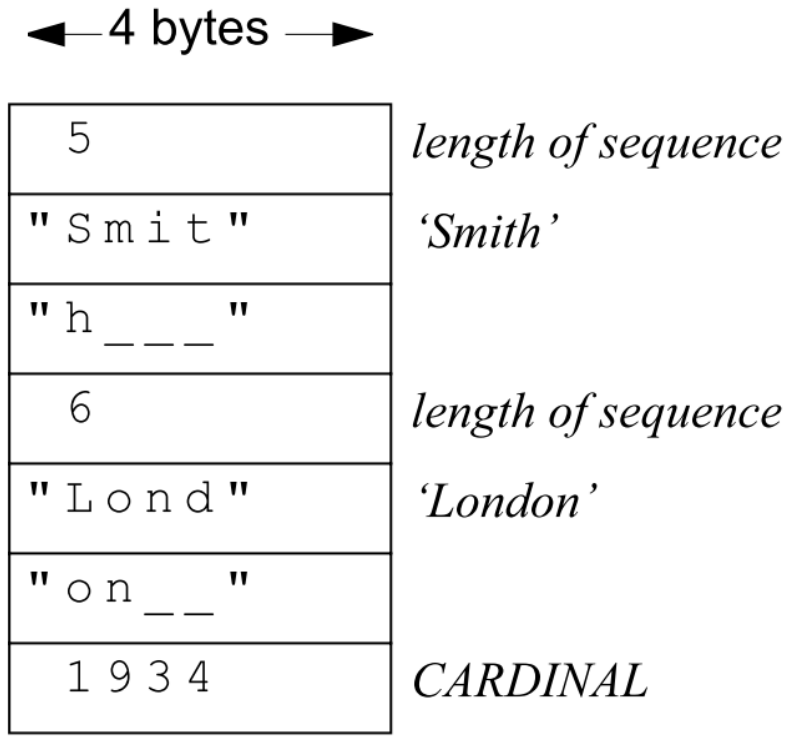
rpcgen -C

calc_clnt.c

```
#include "calc

static struct timeval TIMEOUT =     , 0 };

calc_result *
sum_1(calc_args *argp,  LIENT *clnt)
{
    static calc_result clnt_res;
    if (clnt_call(clnt, SUM, xdr_calc_args, argp,
        xdr_calc_result, &clnt_res,TIMEOUT)  != RPC_SUCCESS)
            {
                return (NULL);}
     return (&clnt_res);
}

calc_result *
sub_1(calc_args *argp, CLIENT *clnt) {...}
calc_result *
mul_1(calc_args *argp, CLIENT *clnt) {...}
calc_result *
div_1(calc_args *argp, CLIENT *clnt) {...}
```

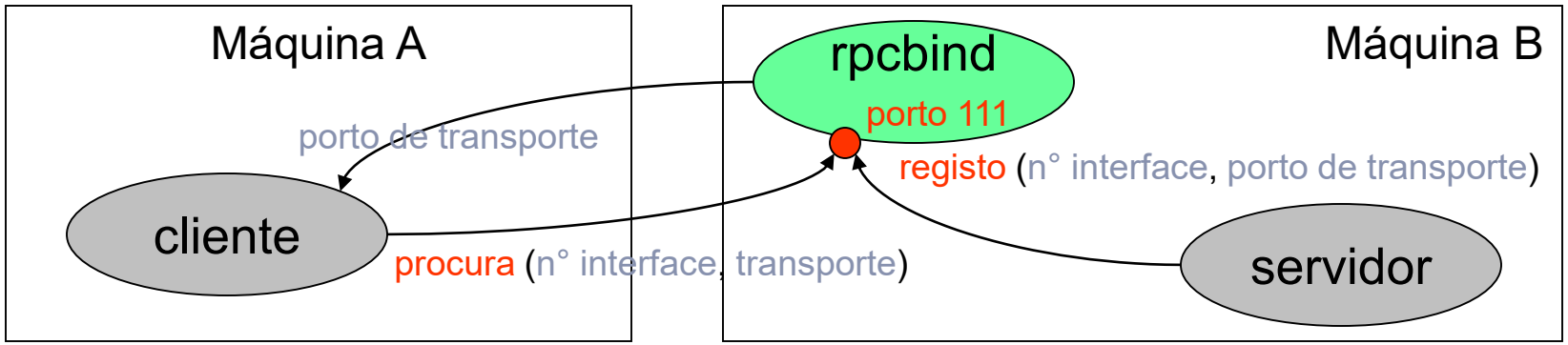
Sun XDR (eXternal Data Representation)

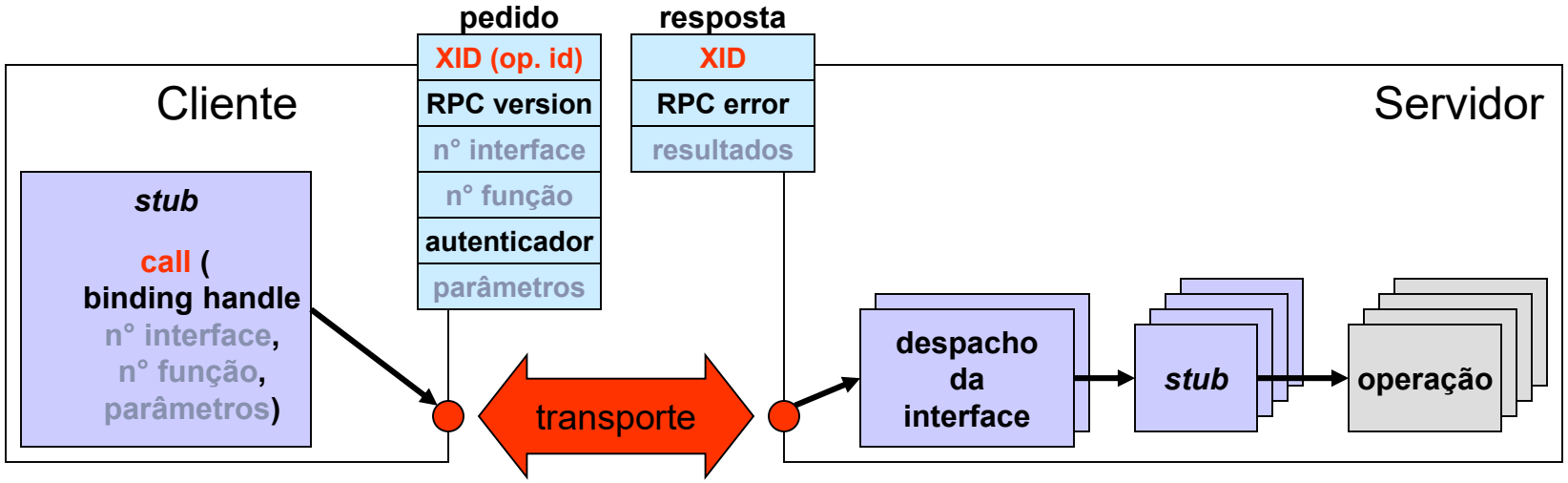
- Exemplo de marshalling de parâmetros em mensagem Sun XDR:
  - 'Smith', 'London', 1934



Sun RPC:

Serviço de Nomes e encaminhamento de RPCs





Exemplo: IDL
Sun RPC Ficheiro banco.x

```
program BANCOPROG {
  version BANCOVERS {
    criarRet           CRIAR(criarIn) = 1;
    saldoRet           SALDO(int) = 2;
    resultado          DEPOSITAR(contaEvalor) = 3;
    resultado          LEVANTAR(contaEvalor) = 4;
    resultado          TRANSFERIR(transferirIn) = 5;
    pedirExtratoRet    PEDIREXTRATO(pedirExtratoIn) = 6;
  } = 1;
} = 0x20000005;
```

Exemplo: Ficheiro banco_svc.c Gerado pelo rpcgen

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "banco.h"
static void bancoprog_1();

main()
{
  register SVCXPRT *transp;

  (void) pmap_unset(BANCOPROG, BANCOVERS);
  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {
    fprintf(stderr, "cannot create udp service.");
    exit(1);
  }
  if (!svc_register(transp, BANCOPROG, BANCOVERS, bancoprog_1, IPPROTO_UDP)) {
    fprintf(stderr, "unable to register (BANCOPROG, BANCOVERS, udp).");
    exit(1);
  }
  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
    fprintf(stderr, "cannot create tcp service.");
    exit(1);
  }
  if (!svc_register(transp, BANCOPROG, BANCOVERS, bancoprog_1, IPPROTO_TCP)) {
    fprintf(stderr, "unable to register (BANCOPROG, BANCOVERS, tcp).");
    exit(1);
  }
  svc_run();
  fprintf(stderr, "svc_run returned");
  exit(1);
  /* NOTREACHED */
}
```

Cria porto UDP

Associa nome da interface e função de despacho ao porto

Função de despacho

Faz o mesmo para porto TCP

Lança ciclo de espera de pedidos
Quando chegar algum pedido, **svc_run()** chama a função de despacho

Exemplo: Ficheiro banco_svc.c Gerado pelo rpcgen

Função de despacho

```
static void
bancoprog_1(rqstp, transp)
struct svc_req *rqstp;
register SVCXPRT *transp;
{
  union {
    criarIn criar_1_arg;
    int saldo_1_arg;
    contaEvalor depositar_1_arg;
    contaEvalor levantar_1_arg;
    transferirIn transferir_1_arg;
    pedirExtratoIn pedirextrato_1_arg;
  } argument;
  char *result;
  bool_t (*xdr_argument)(), (*xdr_result)();
  char *(*local)();

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

  case CRIAR:
    xdr_argument = xdr_criarIn;
    xdr_result = xdr_criarRet;
    local = (char *(*)()) criar_1;
    break;

  case SALDO:
    xdr_argument = xdr_int;
    xdr_result = xdr_saldoRet;
    local = (char *(*)()) saldo_1;
    break;

  case PEDIREXTRATO:
    xdr_argument = xdr_pedirExt      n;
    xdr_result = xdr_pedirExt      et;
    local = (char *(*)()) p      xtrato_1;
    break;

  default:
    svcerr_noproc(tra      p);
    return;
  }
  bzero((char *)&argument, sizeof(argument));
  if (!svc_getargs(transp, xdr_argument, &argument)) {
    svcerr_decode(transp);
    return;
  }
  result = (*local)(&argument, rqstp);
  if (result != NULL && !svc_sendreply(transp, xdr_result,
                                       result)) {
    svcerr_systemerr(transp);
  }
  if (!svc_freeargs(transp, xdr_      gument, &argument)) {
    fprintf(stderr, "unable to f      arguments");
    exit(1);
  }
  return;
}
```

Função genérica para obter argumentos (da biblioteca de run-time)

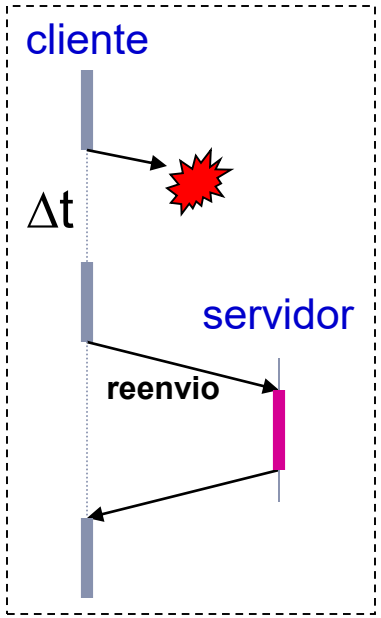
Função genérica para enviar resposta (da biblioteca de run-time)

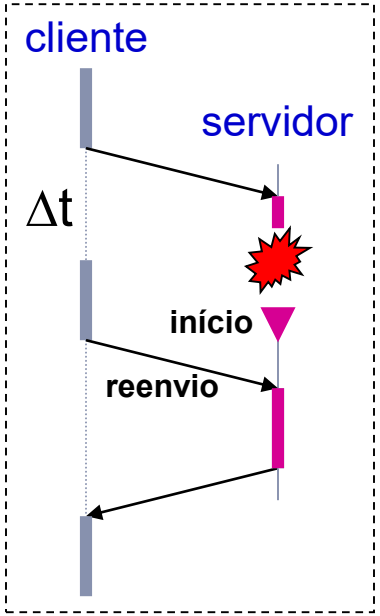
Modelo de faltas

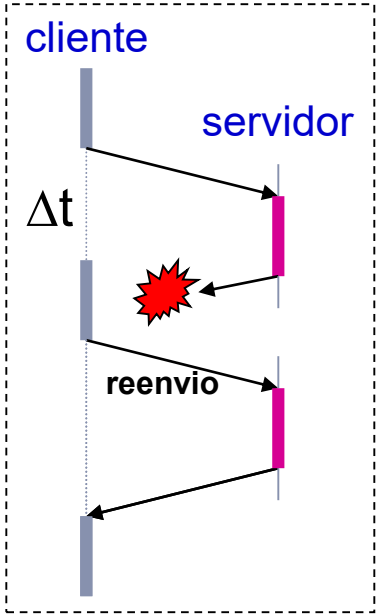
Semânticas de execução

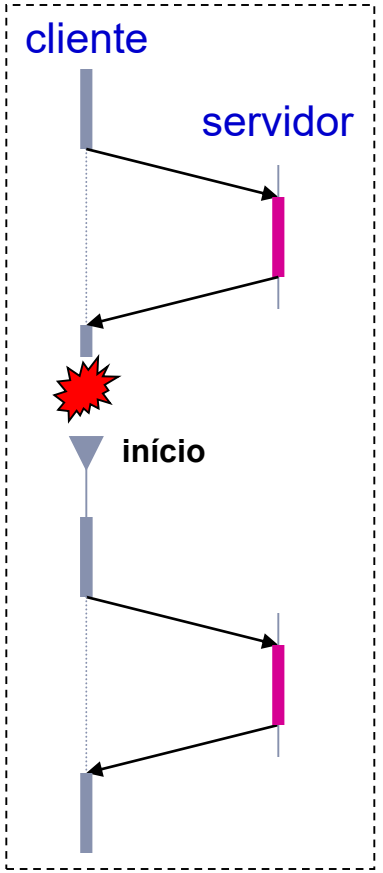
- A semântica de execução determina o modelo de recuperação de faltas
  - Semântica ideal ≡ procedimento local
- Modelo de faltas
  - Perda, duplicação ou reordenação de mensagens
  - Faltas no servidor e no cliente
  - Possibilidade de servidor e cliente reiniciarem após a faltas

Algumas faltas possíveis









Semânticas de execução

- Talvez (*maybe*)
- Pelo-menos-uma-vez (*at-least-once*)
- No-máximo-uma-vez (*at-most-once*)
- Exatamente-uma-vez (*exactly-once*)

Semântica de execução

- A semântica de execução do RPC é **sempre** considerada na **ótica do cliente**

- Se a chamada retornar no cliente, o que é que se pode inferir da execução, considerando que existe uma determinado modelo de faltas

- O modelo de faltas especifica quais as **faltas** que se consideram como **possíveis de ocorrer**

Semânticas de execução

Semântica talvez

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e a chamada retorna com erro
- O cliente não sabe se o pedido foi executado ou não

Protocolo

- Protocolo não pretende tolerar nenhuma falta
  pelo que nada faz para recuperar de uma situação de erro

Semânticas de execução

Semântica pelo-menos-uma-vez

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e
  o *stub* cliente repete o pedido até obter uma resposta
- Se receber uma resposta o cliente tem a garantia que
  o pedido foi executado pelo menos uma vez
- Para evitar que o cliente fique permanentemente bloqueado em caso de falha do servidor existe um segundo *timeout* mais amplo

Para serviços com funções idempotentes

Semânticas de execução

Semântica no-máximo-uma-vez

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e o *stub* cliente repete o pedido
- O servidor não executa pedidos repetidos
- Se receber uma resposta,
  o cliente tem a garantia que o pedido foi executado no máximo uma vez

O protocolo de controlo tem que:

- Identificar os pedidos para detetar repetições no servidor
- Manter estado no servidor acerca dos pedidos em curso ou
  que já foram atendidos

Resumo de técnicas para cada semântica

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

E no caso da semântica exatamente-uma-vez?

Semânticas de execução

Semântica exatamente-uma-vez

- O *stub* cliente não recebe uma resposta num prazo limite
- O timeout expira e o *stub* cliente repete o pedido
- O servidor não executa pedidos repetidos
- Se o servidor falhar existe a garantia de fazer *rollback* ao estado do servidor

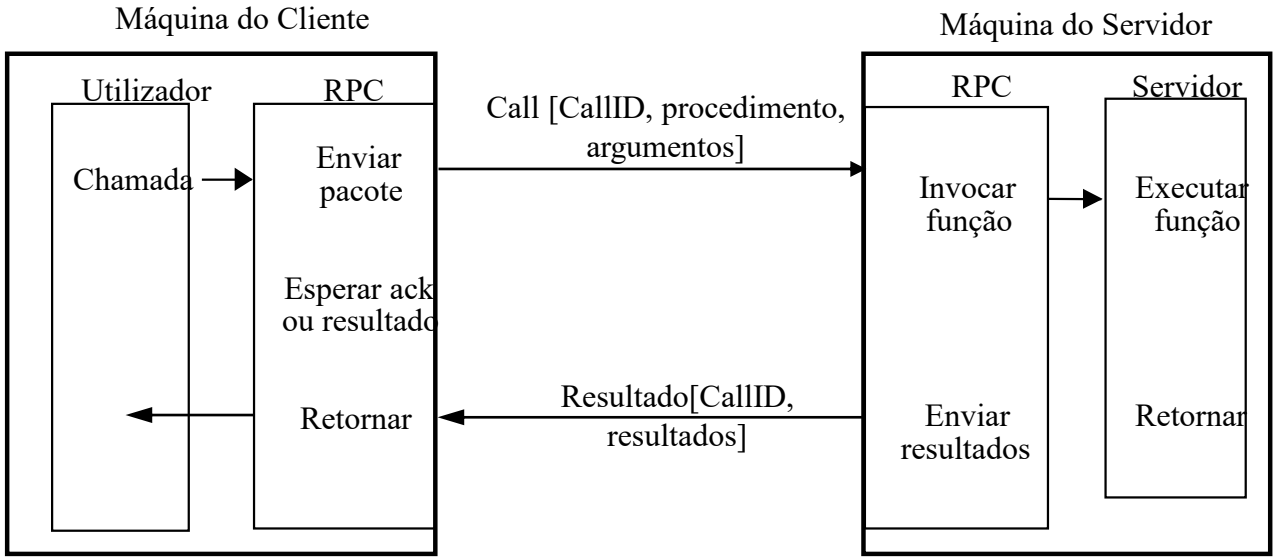
Protocolo

- Servidor e cliente com funcionamento transacional

Protocolo de controlo do RPC

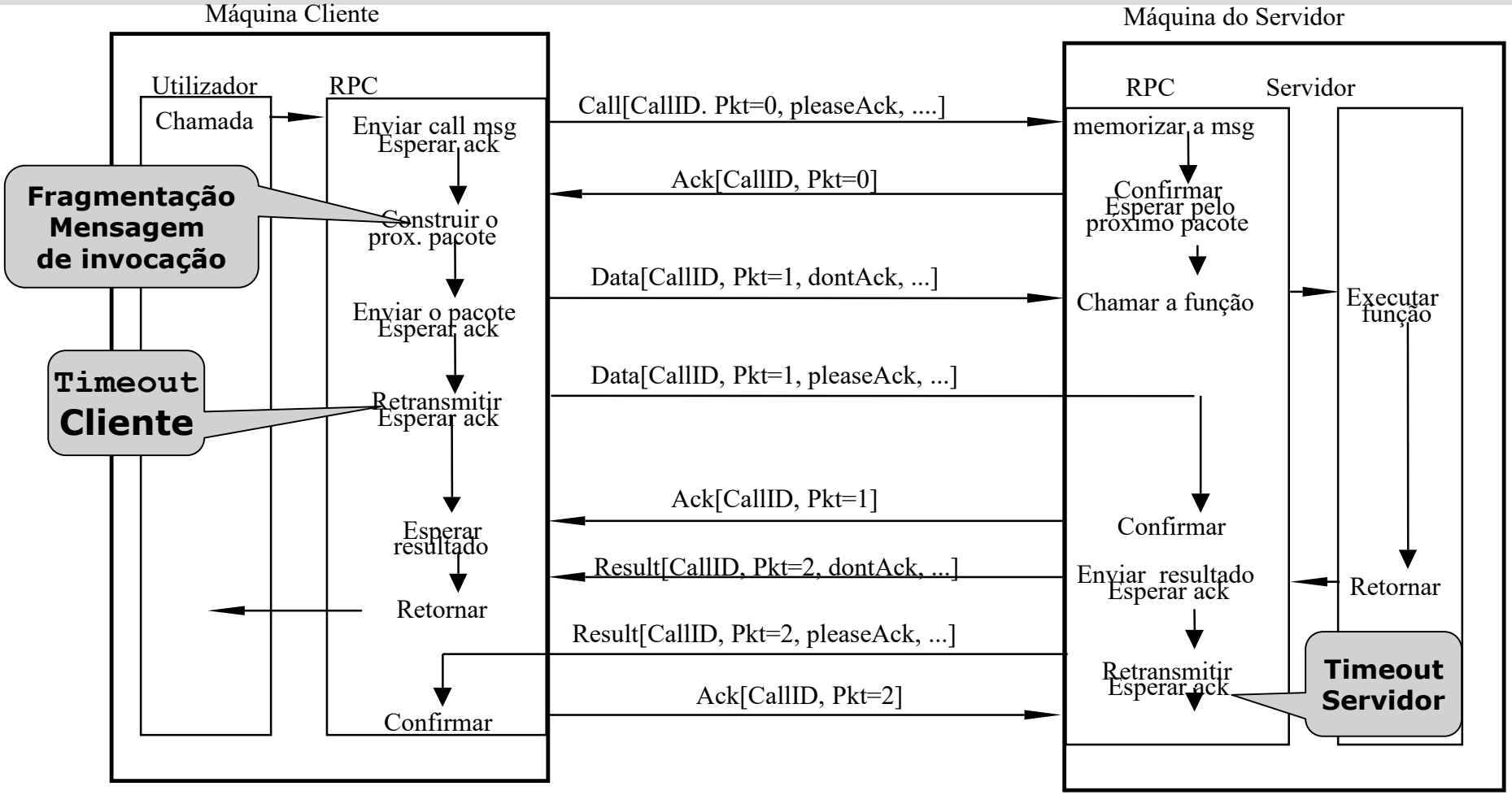
- Suporte de vários protocolos de transporte
  - Com ligação
    - O controlo é mais simples
    - RPCs potencialmente mais lentos
  - Sem ligação
    - Controlo mais complexo (mais ainda se gerir fragmentação)
    - RPCs potencialmente mais rápidos
- Emparelhamento de chamadas/respostas
  - Identificador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

Protocolo de RPC
Situação Ideal



Apenas são necessárias duas mensagens com um transporte do tipo UDP

Protocolo de RPC: Situação Complexa

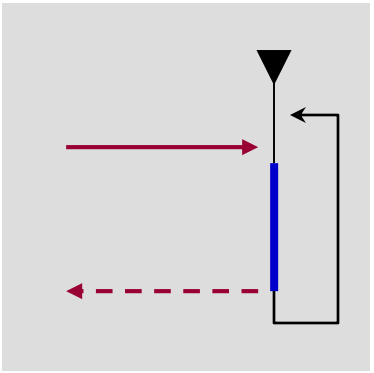


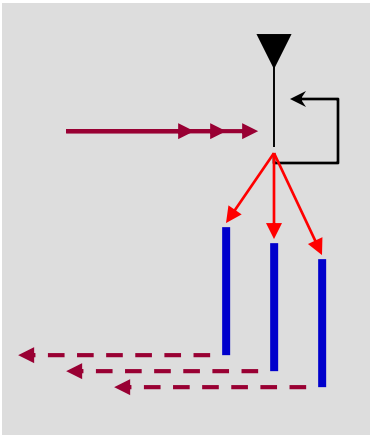
RPC sobre UDP ou TCP?

- Vantagens do uso de TCP
  - Possibilidade de envio de parâmetros de tamanho arbitrário
    - Datagramas UDP limitados a 8 KBytes, tipicamente
    - Envio de parâmetros maiores que 8 KB implica protocolo RPC multi-pacote que é mais complexo de implementar
  - TCP já assegura entrega fiável de pedido/resposta, tornando RPC mais simples
    - Qual a semântica oferecida por um RPC sobre TCP?
  - Mecanismos de controlo de fluxo do TCP adequados para envio de parâmetros de grande dimensão
- Vantagens do uso de UDP
  - Evita tempo de estabelecimento de ligação TCP
  - Quando os aspetos acima não são relevantes, envia mensagens mais eficientemente

Execução de RPCs: (i) Fluxos de execução simples

- Servidores
  - 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



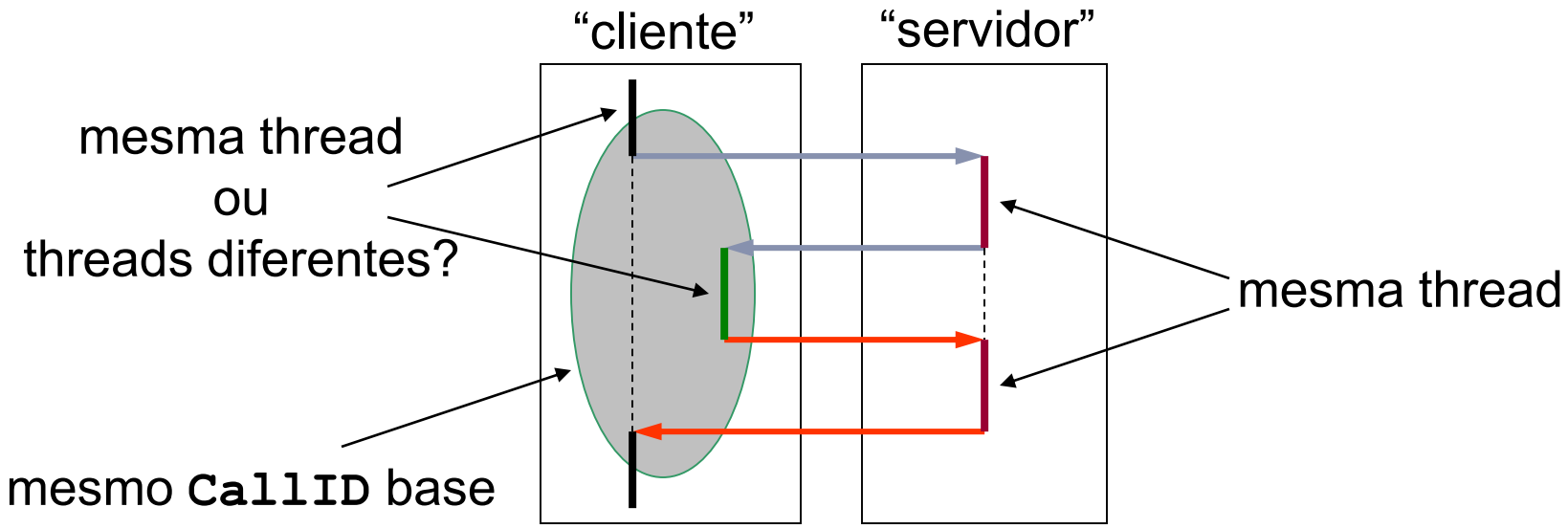


Execução de RPCs: (ii) Fluxos de execução simples

- Clientes
  - Um pedido de cada vez
  - 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:
(iii) Fluxos de execução complexos

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



Resumo

Sumário do RPC: Infraestrutura de suporte

- No desenvolvimento
  - Uma linguagem de especificação de interfaces
    - Interface Description Language, IDL
  - Compilador de IDL
    - Gerador de stubs
- Na execução
  - Serviço de Nomes
  - Biblioteca de suporte à execução do RPC (RPC Run-Time Support)
    - Registo de servidores
    - Binding – protocolo de ligação do cliente ao servidor
    - Protocolo de controlo da execução de RPCs
    - Controlo global da interação cliente-servidor

RPC: Entraves à transparência

- Passagem de parâmetros
  - Semânticas não suportadas pelo RPC
- Execução do procedimento remoto
  - Tolerância a faltas e notificação de faltas
- Desempenho
  - Depende em grande medida da infra-estrutura de comunicação entre cliente e servidor

Sumário do RPC: comparação com Mensagens

Positivo	Negativo
▪ Programação usando uma IDL que é uma linguagem idêntica às linguagens de programação habituais. ▪A interface do serviço encontra-se claramente especificada e não é apenas um conjunto de mensagens ▪ O modelo de invocação de uma função e respectiva sincronização simplificam a programação ▪ Os dados são automaticamente codificados e descodificados resolvendo o problema da heterogeneidade ▪Mecanismo de estabelecimento da ligação entre o cliente e o servidor é automatizado através do serviço de nomes e rotinas do run-time de suporte ao RPC ▪ As funções do cliente e do servidor são consistentes, o sistema garante que ambas são modificadas coerentemente ▪As exceções adaptam-se bem ao tratamento de erros nas invocações remotas	• Só são bem suportadas as interacções 1-para-1 (ou seja não suporta difusão) • Funcionamento síncrono •A semântica exatamente-uma-vez não é possível sem um suporte transaccional •A transparência de execução entre uma chamada local e remota •Existem mais níveis de software que implicam maior *overhead* na execução