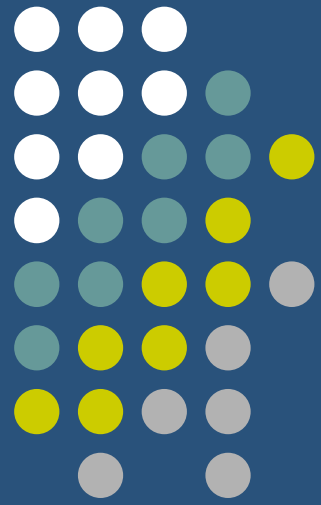
Sistemas Distribuídos

Módulo 7 – Remote Procedure Call (RPC)

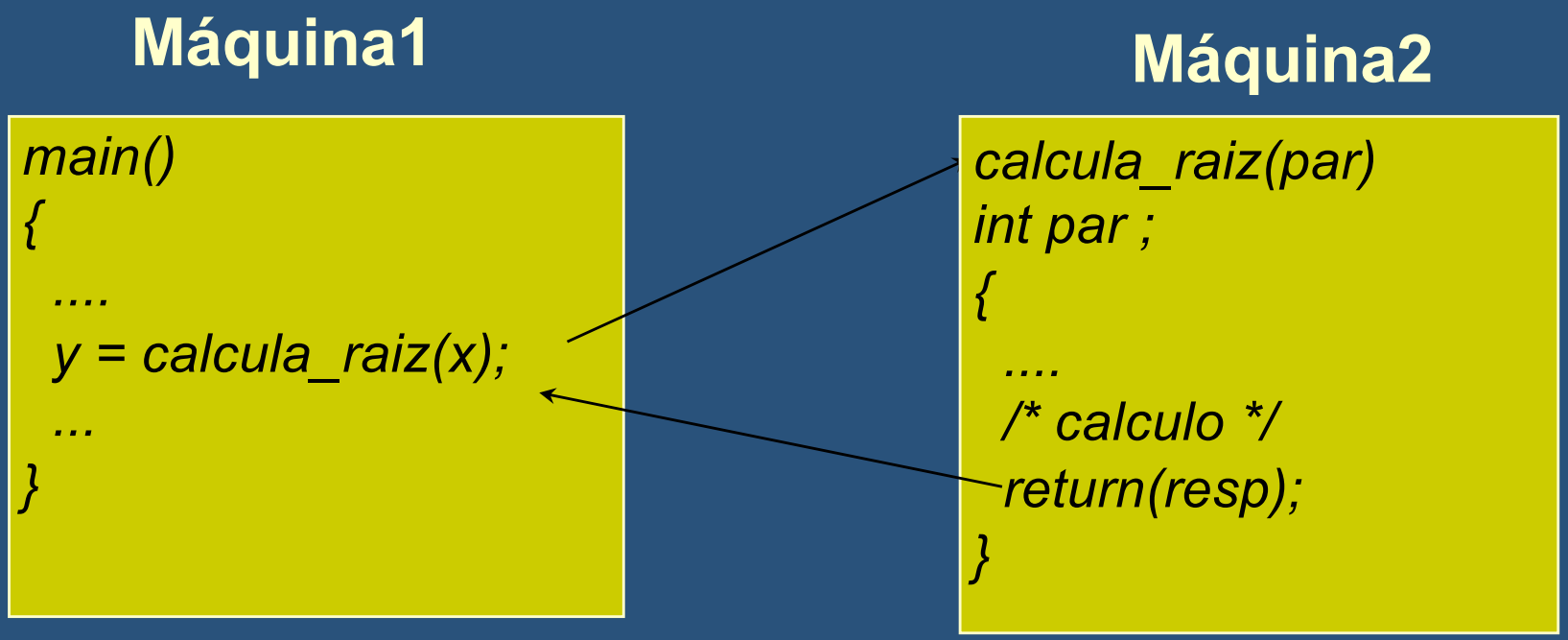


Introdução

- A troca de mensagens é o paradigma de programação que mais se aproxima da arquitetura genérica de sistemas distribuídos.
- Em sistemas com máquina única, no entanto, raramente nós utilizamos este paradigma.
- Muitos programadores acham que a programação por troca de mensagens é difícil.
- No intuito de tornar a programação de sistemas distribuídos mais próxima da programação dos sistemas monoprocessadores, foi proposta a chamada a procedimento remoto (Remote Procedure Call) - RPC.

Introdução

- A RPC permite que processos executem funções pertencentes a processos que rodam em outra máquina.



Introdução

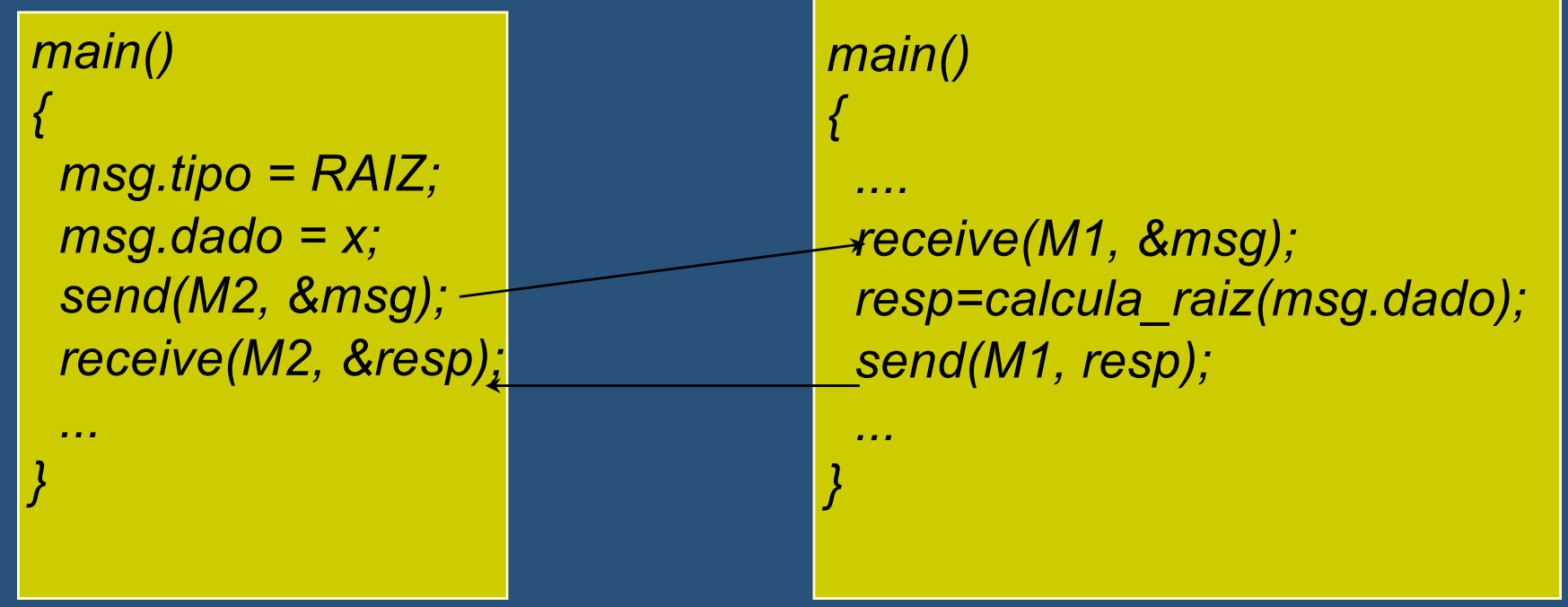
- Mesmo procedimento programado com troca de mensagens:

Máquina1

```
main()
{
  msg.tipo = RAIZ;
  msg.dado = x;
  send(M2, &msg);
  receive(M2, &resp);
  ...
}
```

Máquina2

```
main()
{
  ....
  receive(M1, &msg);
  resp=calcula_raiz(msg.dado);
  send(M1, resp);
  ...
}
```



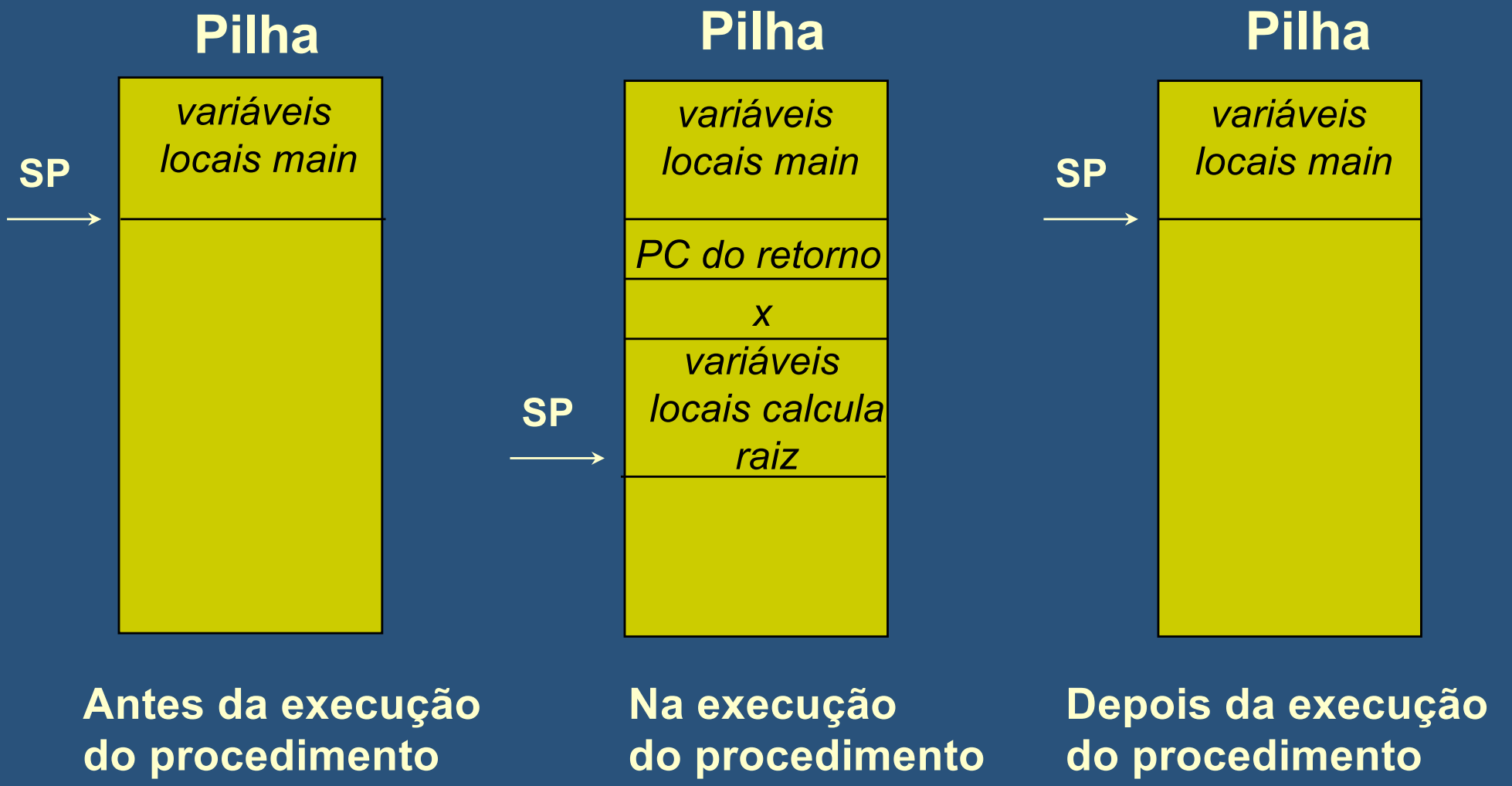
Introdução – Chamada a Procedimento Local

- Chamada local a procedimento:

```
main()
{
   ...
   resp = calcula_raiz(x);
}
```

- Em C, os parâmetros podem ser passados por duas maneiras:
  - por valor: Exemplo: calcula_raiz(x);
  - por referência: Exemplo: calcula_raiz(&x);

Introdução - Chamada a Procedimento Local



Funcionamento da RPC

- A RPC deve ser, o máximo possível, transparente ao programador.
- O programador não deve notar que está chamando uma rotina que se encontra em outra máquina.
- A chamada ao procedimento remoto é idêntica a uma chamada local.

LPC vs RPC

Local Procedure Call

Memória

```
main()
{
…
r=calcula(x);
…
}

calcula(int i)
{
…
 return(resp);
}
```

Remote Procedure Call

Memória Maq1

```
main()
{
…
r=calcula(x);
…
}
```

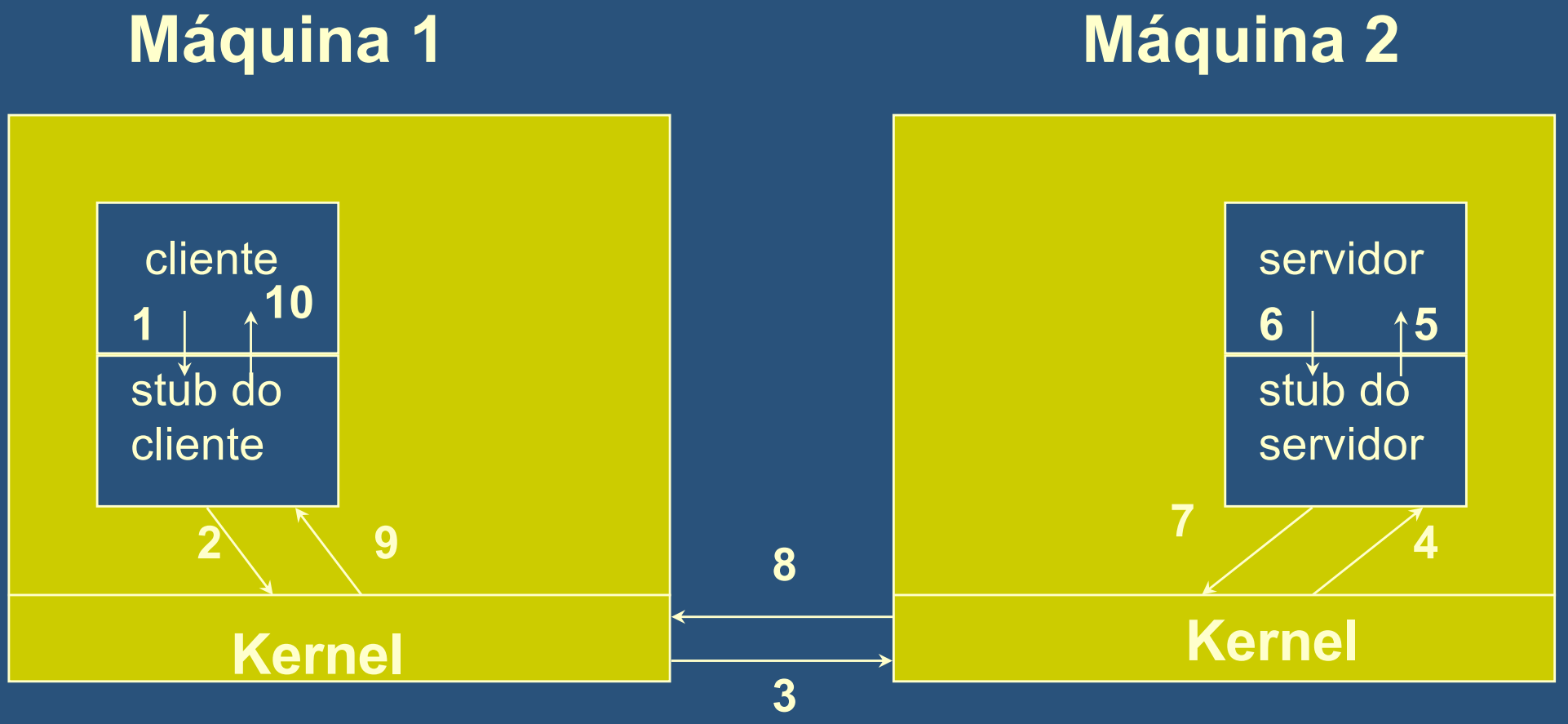
Memória Maq2

```
calcula(int i)
{
…
 return(resp);
}
```

Funcionamento da RPC

- Como fazer chamadas a processos que estão em máquinas remotas?
- Devemos introduzir procedimentos adicionais (stub) para serem chamados pelos processos cliente e servidor.
- Os procedimentos stub transformam a chamada a prodedure em troca de mensagens, de maneira transparente.

Funcionamento da RPC



Funcionamento da RPC

1. O cliente faz chamada a um procedimento remoto
2. É executado um procedimento (stub de cliente) que empacota os parâmetros e chama o kernel para enviar uma mensagem ao servidor.
3. O kernel origem envia a mensagem ao kernel destino.
4. O kernel destino chama o stub do servidor, passando a mensagem.
5. O stub do servidor recebe a mensagem, desempacota-a, e chama o procedimento remoto no servidor.

Funcionamento da RPC

6. O servidor executa a solicitação e devolve o resultado ao stub do servidor.
7. O stub do servidor empacota a mensagem e a envia ao kernel.
8. O kernel do servidor envia a mensagem ao kernel do cliente.
9. O kernel do cliente passa a mensagem ao stub do cliente.
10. O stub do cliente desempacota os resultados e os passa ao cliente.

Funcionamento da RPC

- Stub cliente:
  - Recebe os parâmetros, empacota-os e envia a mensagem ao stub servidor
  - Ordenação de parâmetros: ordem de empacotamento de parâmetros
  - Na mensagem, são colocados os parâmetros e o nome do procedimento remoto a ser executado
  - O stub cliente faz parte do código do cliente!!!

Funcionamento da RPC

- Stub servidor:
  - Recebe a mensagem do kernel e faz a chamada ao procedimento a ser executado no servidor.
  - Recebe o resultado do servidor e o empacota para que o kernel o envie ao cliente.
  - O stub servidor faz parte do código do servidor!!!

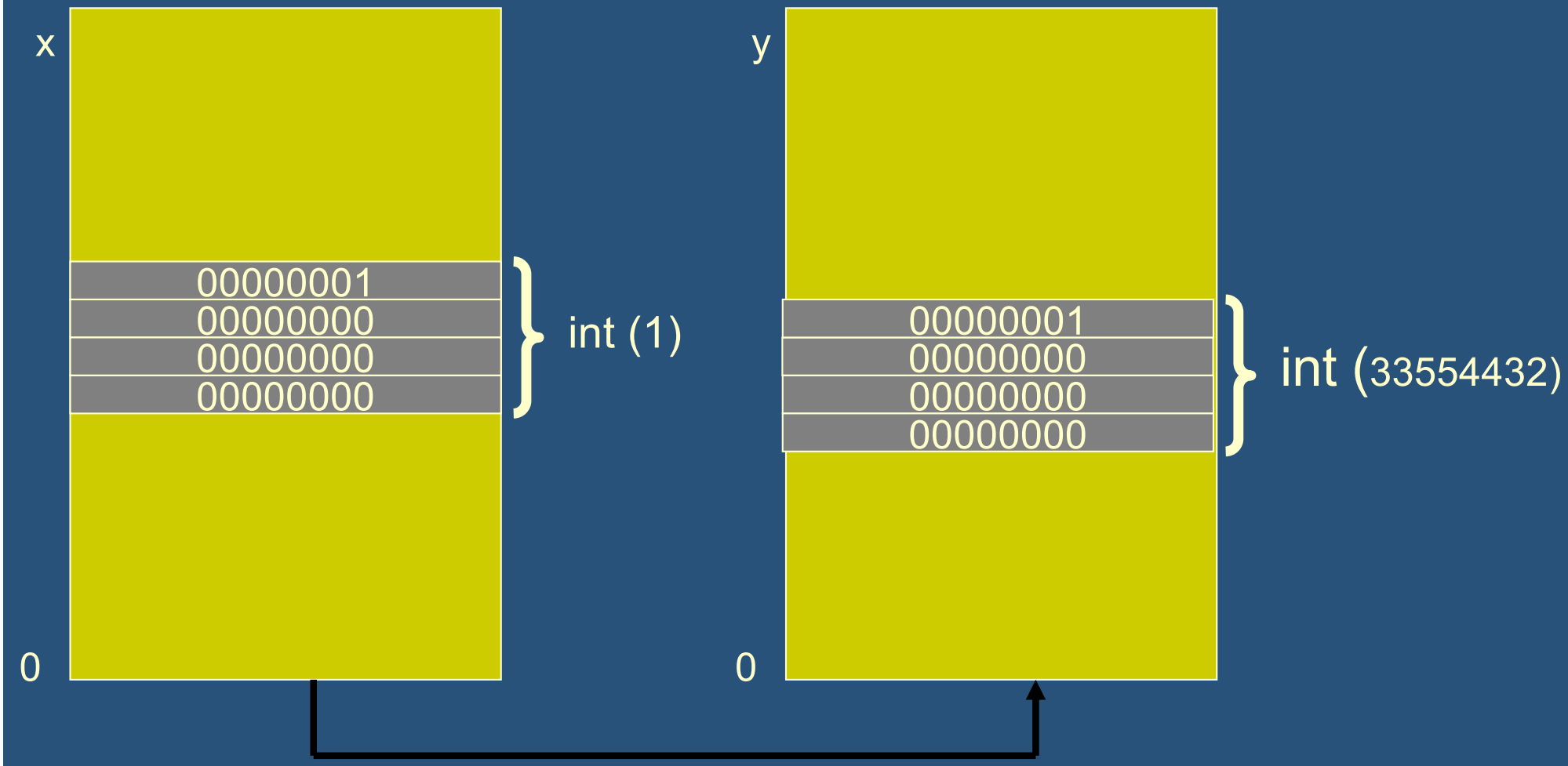
RPC – Interpretação de Parâmetros

- Problema: Como garantir que o dado enviado para uma máquina será interpretado da mesma maneira na máquina destino?
  - Em redes homogêneas, não há problema. Todos os tipos básicos (int, float, char) são armazenados da mesma forma.
  - Como proceder no caso de redes heterogêneas?

RPC – Interpretação de Parâmetros

- Representação interna dos tipos básicos:
  - Mainframe IBM: EBCDIC x IBM-PC: ASCII
  - Representação de inteiros: complemento a 1 x complemento a 2
  - Representação de ponto flutuante (mantissa e expoente): formato IEEE e formatos proprietários.
  - Ordenação de bytes: esquerda para a direita (big endian): SPARC x direita para a esquerda (little endian): Intel 386.

RPC – Interpretação de Parâmetros



RPC – Interpretação de Parâmetros

- Solução 1:
  - receber os parâmetros, empacotá-los de acordo com uma representação padrão, enviá-los. Na recepção, converter da representação padrão para o formato da máquina local
  - Ineficiente se máquinas forem do mesmo tipo.
- Solução 2:
  - enviar a msg no formato nativo, mas com um campo informando qual o formato utilizado.
    - Perda da flexibilidade.

RPC – Interpretação de Parâmetros

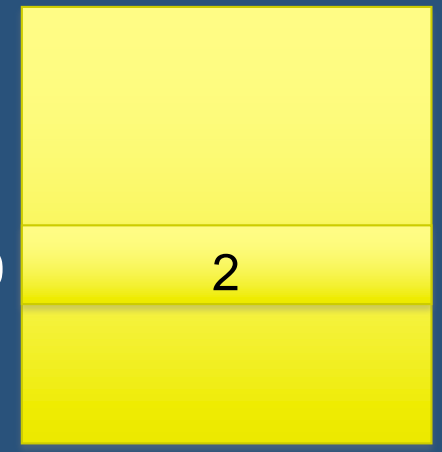
- A solução 1 é de longe a mais adotada.
- Marshalling: transformar uma coleção de dados do formato nativo para o formato padrão
- Unmarshalling: transformar uma coleção de dados do formato padrão para o formato nativo
- External Data Representation (XDR): padrão aceito entre as partes para a representação de estruturas de dados e valores primitivos

LPC vs RPC

Local Procedure Call

```
main()
{
int x;
x=2;
r=calcula(&x);
…
}
```

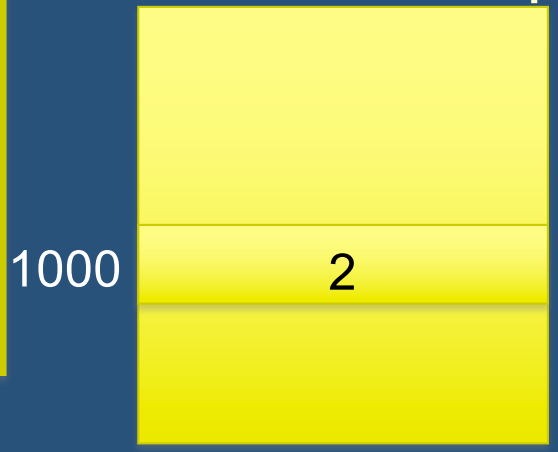
Memória



Remote Procedure Call

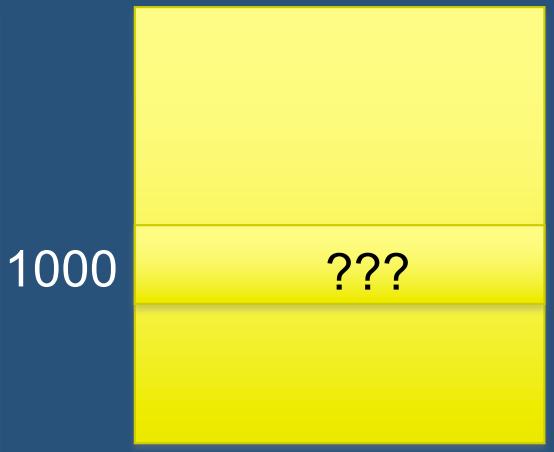
```
main()
{
int x;
x=2;
r=calcula(&x);
…
}
```

Memória Maq1




```
calcula(int *i)
{
…
}
```

Memória Maq2

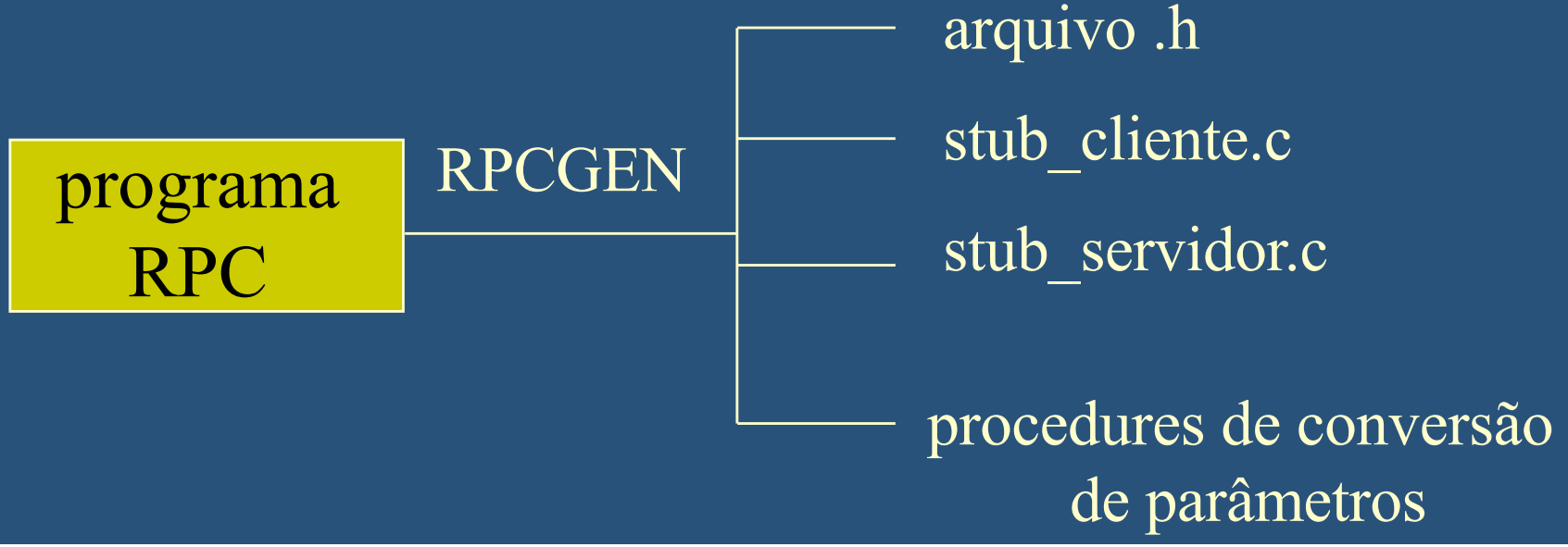


RPC - Passagem de parâmetros

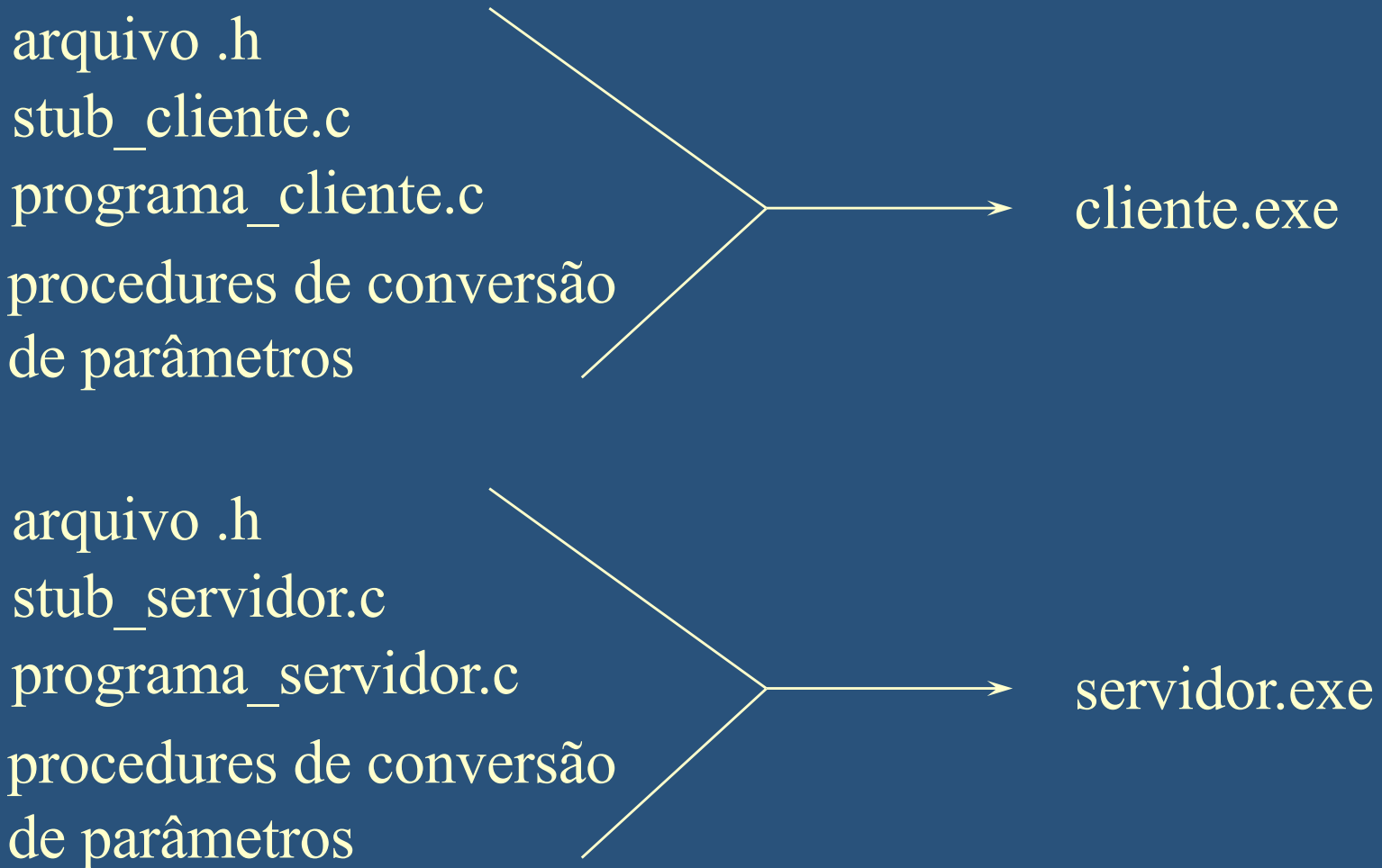
- Como passar parâmetros por referência (ponteiros)? Exemplo: rotina calculo(&a,&b)
  - Proibir a passagem de ponteiros
  - O stub cliente acessa o endereço &a e copia o valor de a para a mensagem. O stub servidor recebe a mensagem e chama a rotina do servidor, utilizando agora um endereço interno ao stub (onde o parâmetro foi colocado, dentro da mensagem recebida do cliente). Isso funciona no caso de inteiros e no caso de strings dos quais sabemos o tamanho. Não funciona para ponteiros genéricos.

RPC - Geração de Stubs

- Os programadores compõem um arquivo com o procedimento remoto a ser executado no servidor e um gerador de código automático (compilador) gera programas C que contem os stubs. No NFS, o utilitário se chama RPCGEN.



RPC – Geração do Programa Cliente e do Programa Servidor



RPCGEN - Exemplo

- Exemplo de especificação de rotina em linguagem RPC, que será utilizada pelo RPCGEN para gerar os stubs cliente e servidor:

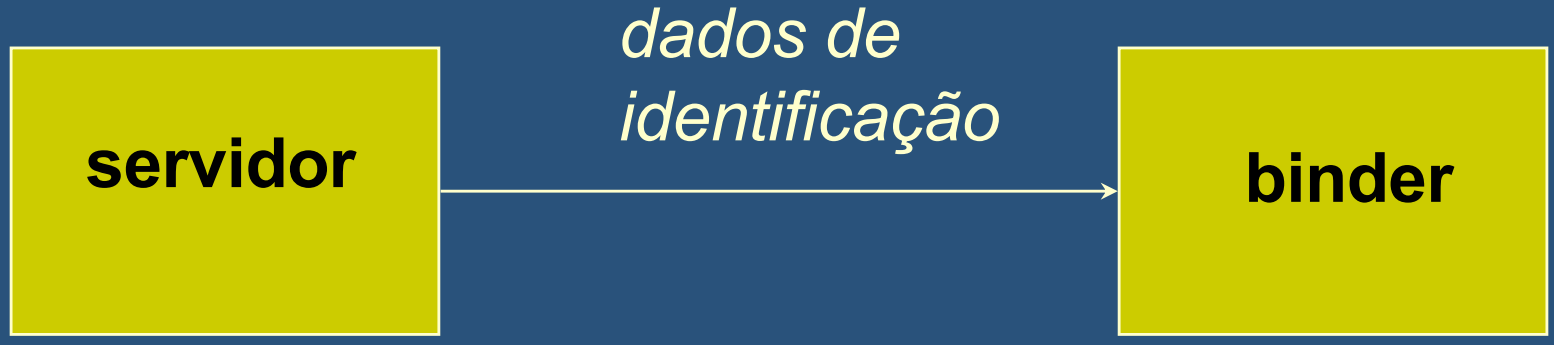
```
program BINOP {
    version BINOP_VERS {
    long BINOP_ADD (struct input_args) = 1;
    } = 1;
} = 300030;

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

RPC - Ligação dinâmica

- Como um cliente localiza o servidor? A maneira mais simples é associar estaticamente o endereço do servidor ao seu nome, gravando no hardware do cliente o endereço de rede do servidor. Esta abordagem é bastante inflexível.
- Com a ligação dinâmica a correspondência entre o nome do servidor e seu endereço é feita dinamicamente.

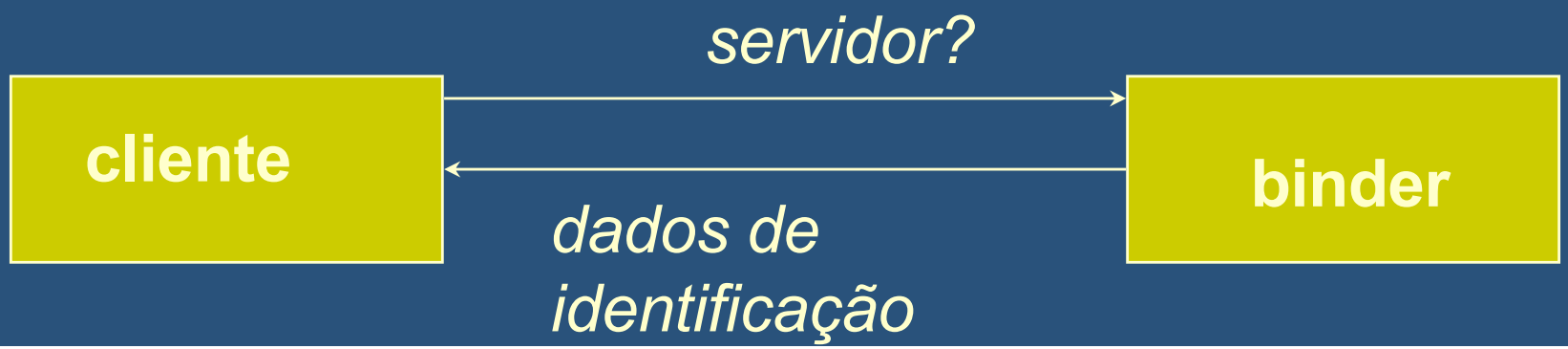
RPC - Ligação dinâmica

- Rodar o servidor. Antes de entrar no loop principal, o servidor chama a rotina initialize.
  - rotina initialize: **exporta** a interface do servidor
    - O servidor envia uma mensagem contendo o seu nome, a versão, um identificador único e o seu endereço para o binder. Este processo é chamado de registro do servidor.
  - Exportação de interface:



RPC - Ligação dinâmica

- Rodar o cliente. Ao executar o primeiro procedimento remoto, o stub do cliente verifica que não há nenhum servidor ligado a ele.
- Envia, então, uma mensagem ao binder, pedindo a **importação** da interface do servidor.
- O binder verifica se há algum servidor "exportado". Se sim, ele fornece ao stub do cliente, o identificador e o endereço do servidor.
- Importação de interface:



RPC - Ligação dinâmica

- Vantagens:
  - flexibilidade
  - tolerância a falhas
- Desvantagens:
  - overhead
  - o ligador, se centralizado, torna-se facilmente um gargalo. Ligadores distribuídos tem o problema de coerência.

RPC - Produtos comerciais

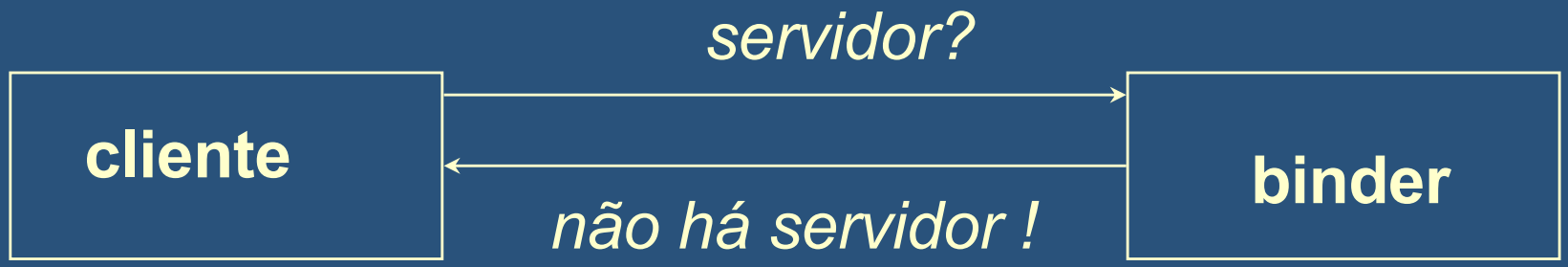
- ONC RPC - RPC da Sun (padrão)
  - Primeiro sucesso comercial
  - O NFS foi implementado com a ONC RPC
  - TI RPC (transport-independent RPC): RPC independente do protocolo de transporte. O TI RPC pode usar tanto o TCP/IP (confiável) como o UDP/IP (não-confiável)
- DCE RPC - Evolução da ONC RPC

RPC - Tratamento de falhas

- O principal problema do modelo RPC é a manutenção da transparência na presença de falhas.
- Tipos comuns de falhas:
  - O cliente não localiza o servidor
  - Perda de mensagens solicitando serviços
  - Perda de mensagens com resposta
  - Queda do servidor
  - Queda do cliente

RPC - Tratamento de falhas
O cliente não acha o servidor

- Essa situação pode ocorrer se o servidor não está no ar ou a versão do cliente é diferente da versão do servidor. O procedimento remoto deve retornar um erro.



RPC - Tratamento de falhas
O cliente não acha o servidor

- Que erro retornar?
  - Como no Unix, pode retornar -1 e setar a variável global errno com o erro correspondente à "servidor inválido".
    - Problema: -1 pode ser uma resposta válida para alguma função.
  - Gerar uma interrupção, inventando um novo sinal SIGNOSERVER.
    - Problema: várias linguagens não suportam sinais

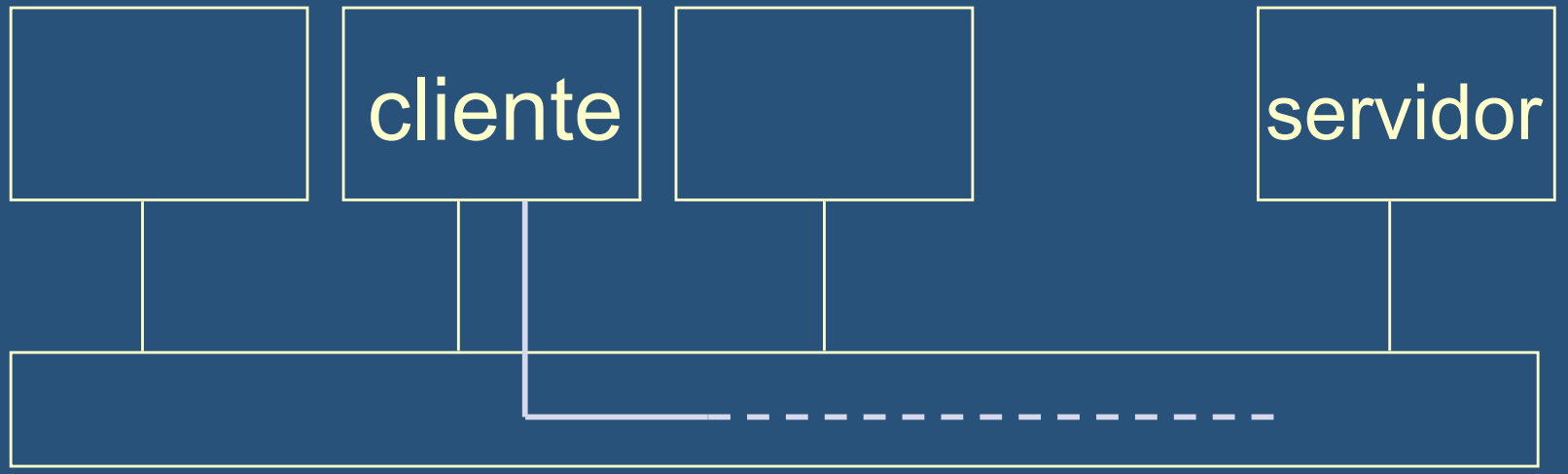
```
ret = rotina_rpc(…);
```

RPC - Tratamento de falhas
O cliente não acha o servidor

- Nas duas soluções anteriores há uma perda da transparência. O modelo de RPC diz que a chamada remota deve ser exatamente igual à chamada local. Em uma chamada local, o erro NOSERVER nunca seria retornado!!!

RPC - Tratamento de falhas
Perda da msg de solicitação

- O cliente manda executar um procedimento remoto, mas esta solicitação se perdeu na rede. Neste caso, o cliente pode ficar esperando indefinidamente.

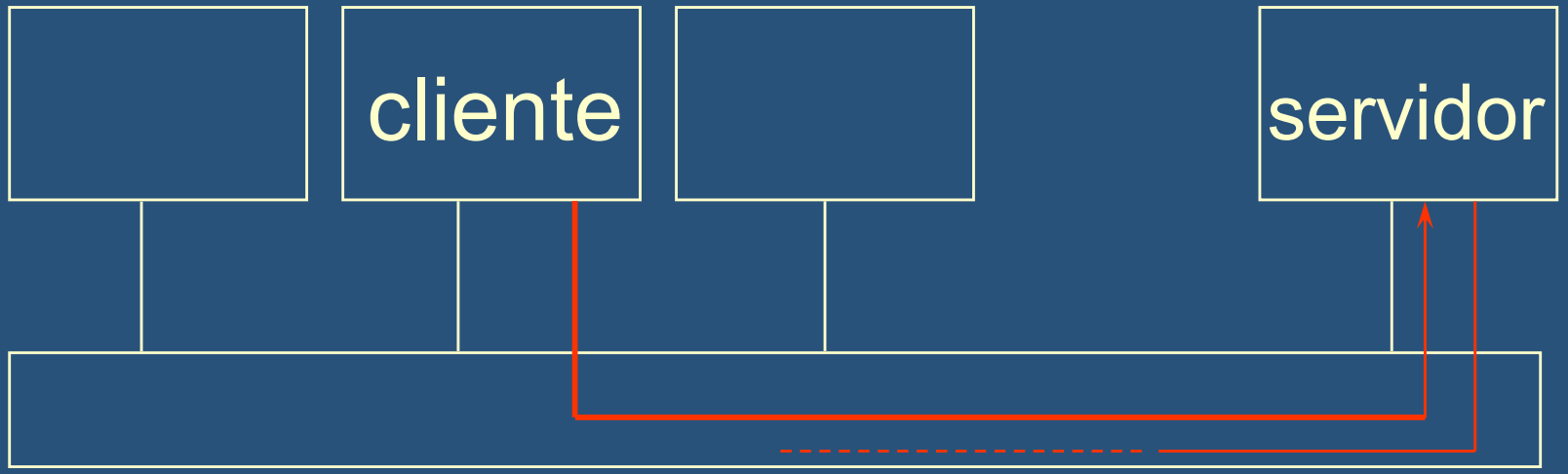


RPC - Tratamento de falhas
Perda da msg de solicitação

- Solução:
  - Uso de temporizador no cliente. O temporizador é setado a cada chamada. Se o tempo se esgotar, é feita a retransmissão. Se várias retransmissões são feitas em um curto espaço de tempo, o cliente conclui que o servidor caiu.
  - Problemas:
    - overhead de processamento e de mensagens, caso o timeout seja mal estipulado

RPC - Tratamento de falhas
Perda da resposta

- O servidor executou a função remota e enviou a resposta. Só que a resposta não chegou ao cliente.

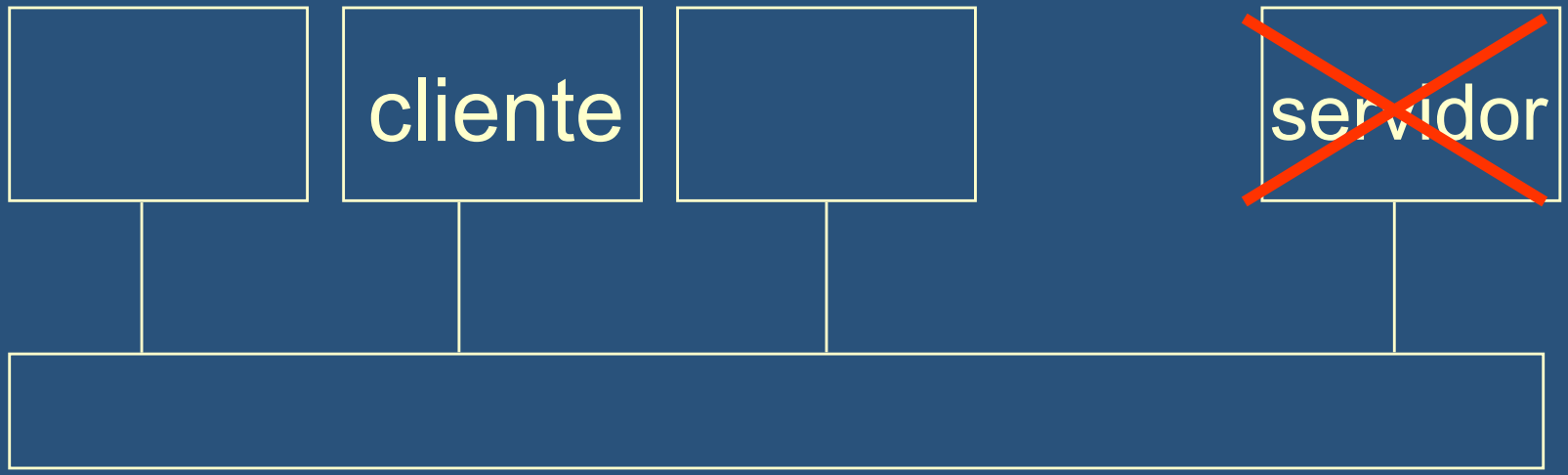


RPC - Tratamento de falhas
Perda da resposta

- Retransmitir a solicitação
  - Problema: a função será executada duas ou mais vezes, quando na realidade, só deveria ser executada uma vez.
- Retransmitir a solicitação com numeração lógica.
  - Permite a distinção entre uma nova solicitação e uma retransmissão.

RPC - Tratamento de falhas
Queda do servidor

- O servidor saiu do ar.
  - Quando ele saiu do ar?
    - Esperando mensagem ou
    - Executando o procedimento remoto ou
    - Enviando a resposta

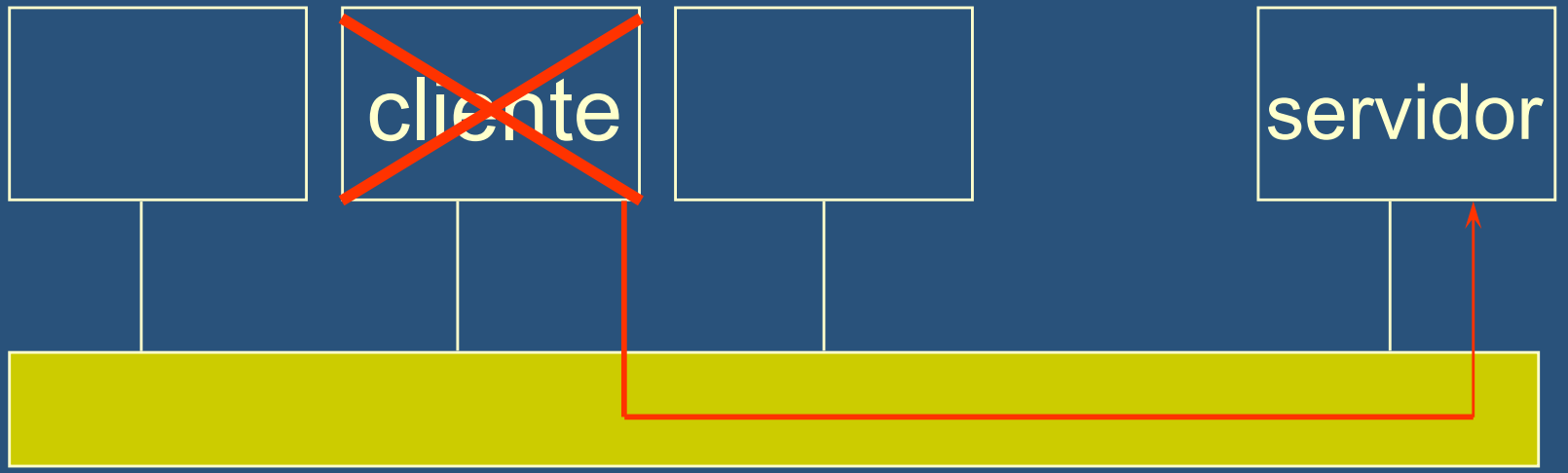


RPC - Tratamento de falhas
Queda do servidor

- Para decidir o que fazer, deve-se determinar o número de vezes que uma solicitação pode ser executada.
  - Executar no mínimo uma vez. O kernel do cliente espera até que o servidor dê um novo boot ou que uma ligação com outro servidor equivalente seja estabelecida. Neste ponto, pede a operação novamente.
  - Executar no máximo uma vez. Desistir imediatamente quando der o timeout.
  - Executar de 0 a n vezes. Não dar nenhuma garantia para o cliente.

RPC - Tratamento de falhas
Queda do cliente

- O cliente enviou uma solicitação e saiu do ar antes que o servidor respondesse.
  - Neste caso, nós temos um procedimento órfão. Um procedimento órfão é executado sem ter ninguem esperando por seu resultado.



RPC - Tratamento de falhas
Queda do cliente

- Como tratar os procedimentos órfãos?
  - extermínio
  - reencarnação
  - expiração

RPC - Tratamento de falhas
Queda do cliente

- Extermínio: antes de enviar uma mensagem de RPC, o stub cliente grava em um arquivo de log a informação: “enviando RPC”. Quando o cliente entrar no ar de novo, todos os órfãos daquele cliente são assasinados (kill).
- Problemas:
  - overhead de armazenamento de logs
  - os órfãos criados por órfãos (RPC que faz RPC) não serão exterminados

RPC - Tratamento de falhas
Queda do cliente

- Reencarnação: Uso de timestamps.
  - Quando o cliente se recupera, ele envia um timestamp em broadcast aos servidores (timestamp de início).
  - Todas as RPCs em andamento para este cliente são verificadas e as que tiverem sido lançadas em tempo anterior são eliminadas.
  - Os resultados contém também um timestamp. Assim, mesmo que alguns órfãos sobrem, suas respostas serão desconsideradas pelo cliente porque elas dizem respeito a uma “outra vida”.

RPC - Tratamento de falhas
Queda do cliente

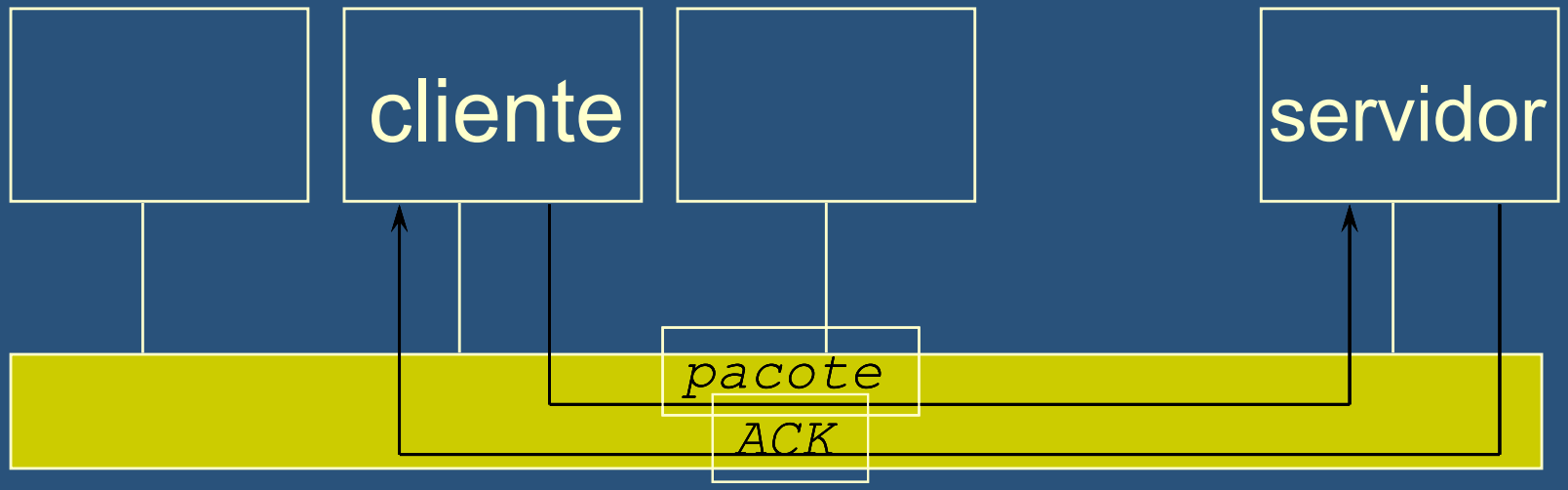
- Expiração: a cada chamada remota é associado um tempo máximo (T) no qual o trabalho deve ser realizado. Se o tempo expirar, os procedimentos são abortados e é preciso solicitá-los novamente. Se o cliente esperar T até entrar no ar, terá certeza de que os órfãos foram mortos.
- Como estipular T?
  - Em que pontos da execução é seguro eliminar um processo?

RPC - Implementação
Tratamento dos Acks

- O problema a ser tratado aqui é a definição do que confirmar: pacotes ou mensagens?
  - **Protocolo pare e espere**
  - **Protocolo em rajadas**
  - **Protocolo híbrido**

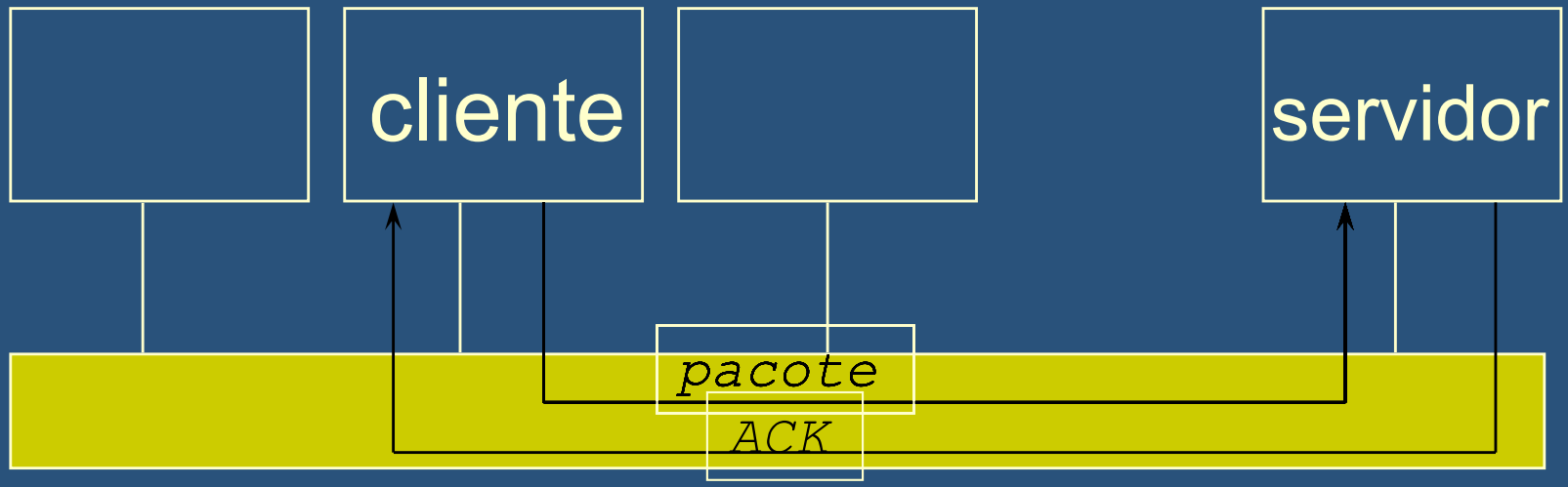
RPC - Tratamento dos Acks Protocolo Pare e Espere

- Um ack é enviado do servidor ao cliente a cada pacote que compõe a mensagem. Só depois do recebimento do ACK é que o próximo pacote é enviado. Se um pacote se perdeu, somente ele é retransmitido. O overhead na rede é grande.



RPC - Tratamento dos Acks Protocolo em rajadas

- Um ack é enviado do servidor ao cliente somente quando todos os pacotes que compõem a mensagem são recebidos. No caso de perda de pacotes, o servidor pode enviar um NAK e neste caso a mensagem toda é retransmitida



RPC - Tratamento dos Acks Protocolo híbrido

- Se não há perda de pacotes, o protocolo híbrido comporta-se exatemente como o protocolo em rajadas.
- No caso de perda de pacotes, o servidor envia um ACK contendo os números dos pacotes que foram recebidos.
- Neste caso, somente os pacotes perdidos são retransmitidos.

RPC - Implementação
Saturação do fluxo de controle

- Os chips de interface de rede tem capacidade de armazenamento finita.
- Por isso, não devemos ter um fluxo muito grande de envio de pacotes.
- O erro “excesso de pacotes” é produzido quando um pacote chega e o receptor não está pronto para aceitá-lo. Geralmente, ele está tratando o pacote anterior.
- A RPC deve ser projetada para reduzir a probabilidade de saturação do fluxo de controle

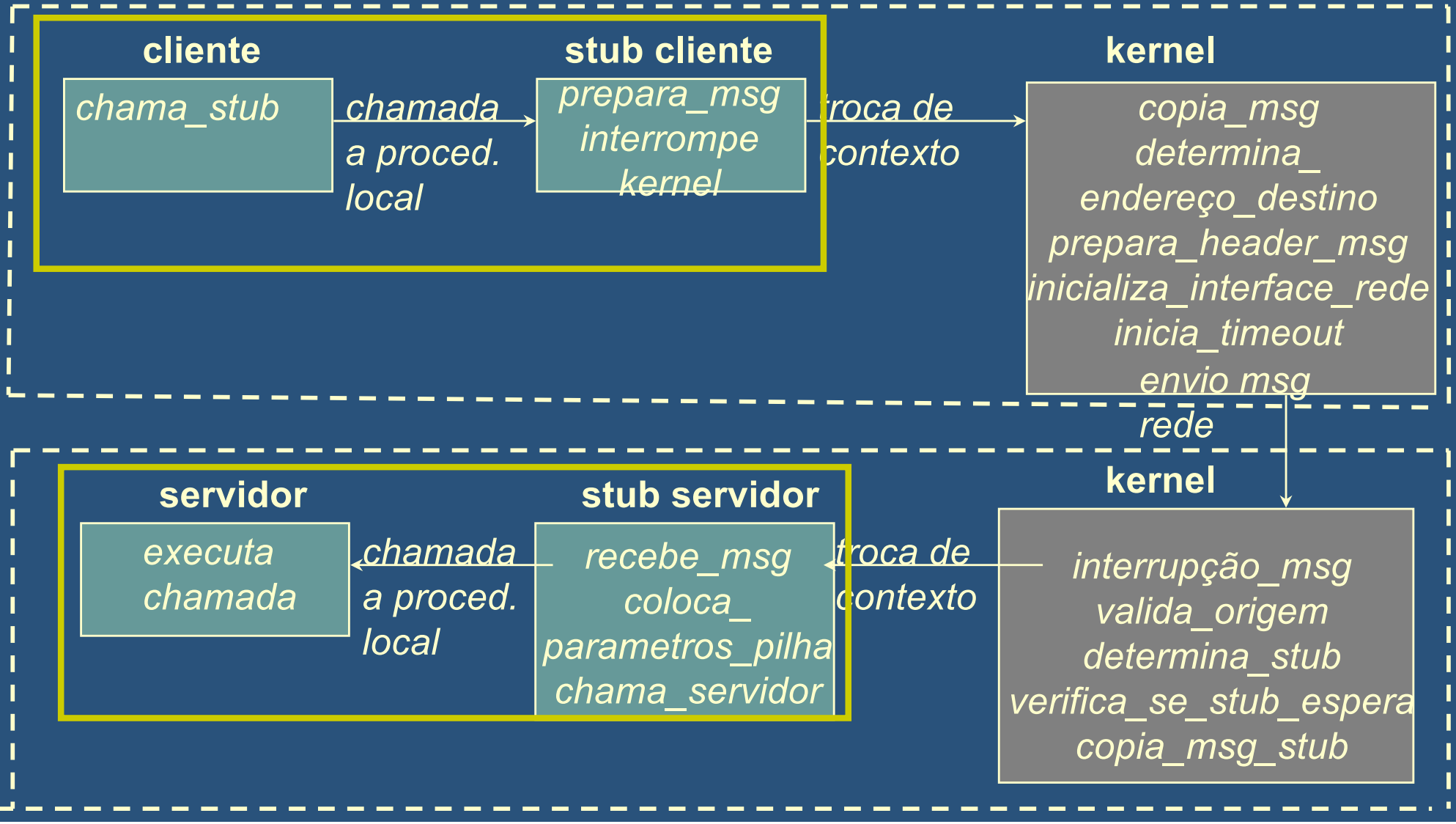
RPC - Implementação
Saturação do fluxo de controle

- Protocolo pare e espere: o excesso de pacotes raramente ocorre, pois há um certo sincronismo no envio de mensagens.
- Protocolo em rajadas: pode ocorrer com muita frequência, já que o envio é assíncrono.

RPC - Implementação
Saturação do fluxo de controle

- Soluções:
  - O transmissor introduz um retardo entre o envio de pacotes (busy waiting ou bloqueio)
  - Se o chip tem capacidade de armazenamento de *n* pacotes, enviar *n* pacotes de cada vez

RPC - Implementação
Caminho crítico



RPC - Implementação
Caminho crítico

- Para melhorar a performance de um sistema, devemos saber onde mais tempo é consumido. Este lugar pode depender das máquinas nas quais a RPC roda.
- Experimentos na DEC Firefly (5 processadores VAX). Protocolo UDP. Chamadas otimizadas. Busy-waiting.
  - *Chamada nula*: custos maiores para a troca de contexto para o stub servidor e movimento de pacotes para a interface de rede.
  - *Chamada com 1440 bytes*: custos maiores para transmissão pela rede e movimento de pacotes para a interface de rede.

RPC - Implementação Cópias

- Como as operações de cópia (tanto locais como remotas) são operações que envolvem bastante tempo, devemos reduzir o número de cópias no projeto de mecanismos do sistema operacional.
- RPC: pior caso:
  - stub cliente -> kernel
  - kernel -> placa da interface de rede
  - interface de rede origem -> interface de rede destino
  - interface de rede -> kernel
  - kernel -> stub servidor

RPC - Implementação
Otimização das cópias

- *Scatter-gather:* implementado no hardware da interface de rede: uma mensagem pode ser montada pela concatenação de buffers. O cabecalho da mensagem pode estar em um espaço de endereçamento (kernel) e o resto da mensagem em outro espaço de endereçamento (cliente)

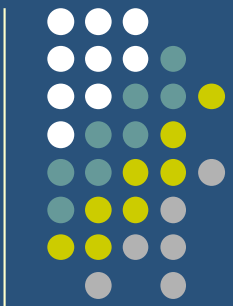
RPC - Implementação
Otimização das cópias

- Mapeamento do buffer da mensagem na memória do kernel e do cliente. O corpo da mensagem é mantido no espaço de endereçamento do cliente.
- Alinhamento do buffer em relação à página
  - Proteção

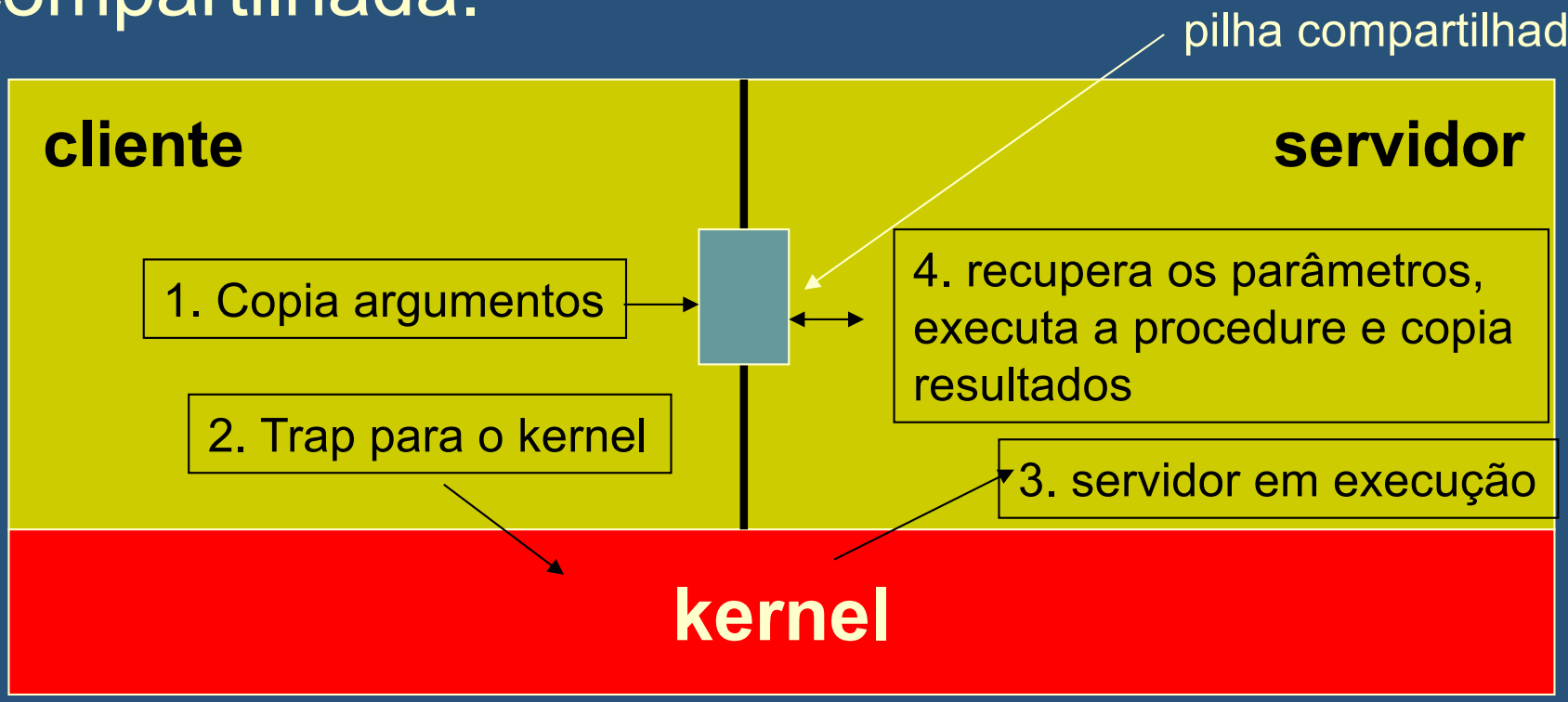
RPC – Implementação Lightweight RPC

- Caso o cliente e o servidor estejam na mesma máquina, a RPC é implementada exatamente como se fosse remota e somente a troca de mensagens é local.
- Caso fosse testado se cliente e servidor residem na mesma máquina, otimizações poderiam ser aplicadas.
- Uma destas otimizações é a Lightweight RPC.

RPC – Implementação Lightweight RPC



- Na LRPC, se estão na mesma máquina, cliente e servidor comunicam-se por memória compartilhada.



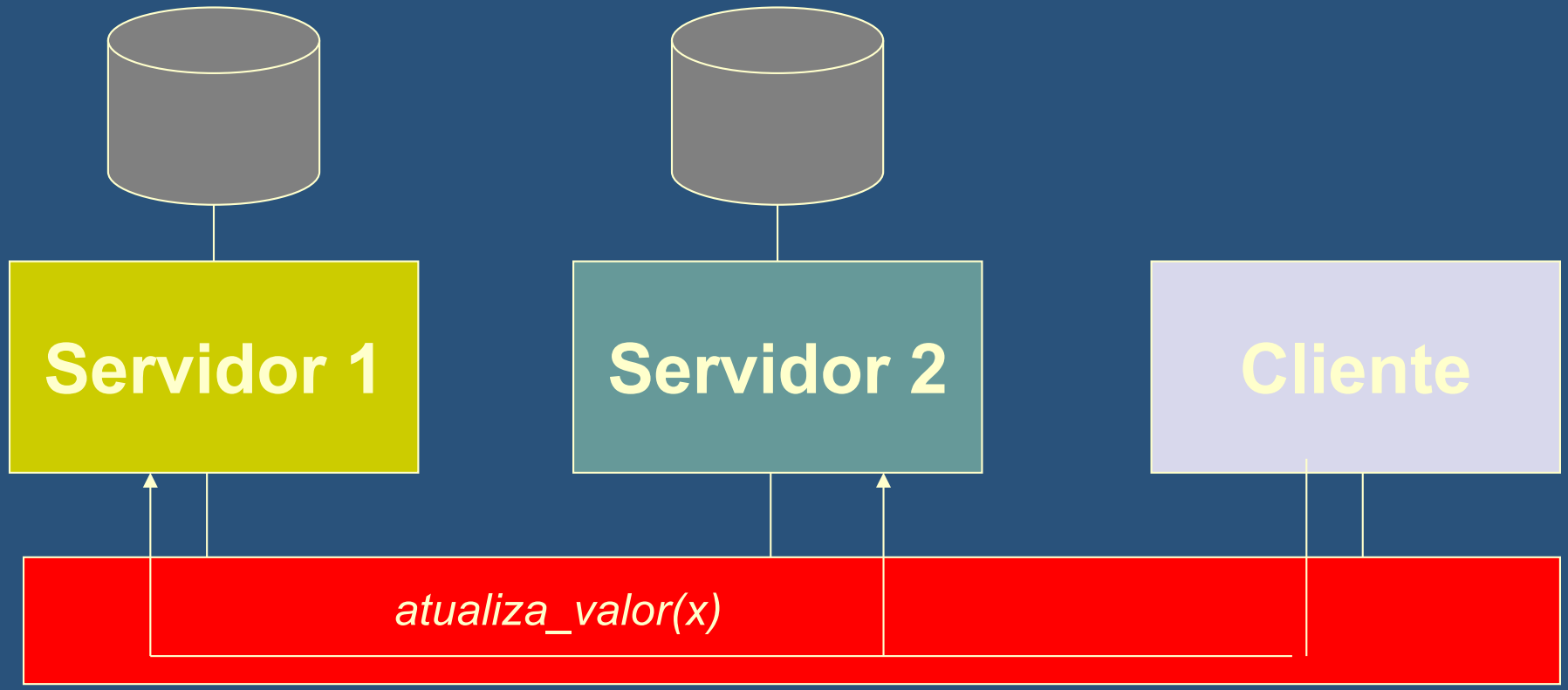
RPC - Níveis de Utilização

- Apesar da filosofia da RPC requerer que a chamada remota se comporte exatamente como uma chamada local, na prática nós podemos utilizar a RPC em 3 níveis:
  - Alto nível: chamada transparente
  - Nível médio: a rotina é chamada através de callrpc e o servidor se registra
  - Baixo nível: controle sobre o protocolo, retransmissões, número de sockets alocados, erros, etc

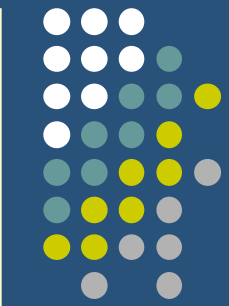
RPC - Características Adicionais

- Broadcast RPC: O cliente envia um broadcast na rede e vários servidores podem responder.
  - Protocolo UDP
  - Os servidores só respondem quando tiverem processado a solicitação. Em caso de erro, ficam silenciosos.
- Até quantas respostas esperar?
  - Nenhuma, uma, a maioria, todas, etc.

Broadcast RPC



RPC - Características Adicionais



- No-Response RPC ou batching: As RPCs deste tipo assumem que o servidor não deve enviar resposta. Assim, o cliente continua a execução logo depois da chamada à RPC, sem esperar que o seu tratamento seja terminado ou mesmo iniciado.
- Requer o protocolo TCP.
  - Aumenta o paralelismo da aplicação distribuída

No Response RPC

Máquina1

```
main()
{
  ....
  calcula_raiz(x);
  ...
}
```

Máquina2

```
calcula_raiz(par)
int par ;
{
  ....
  /* calculo */

}
```