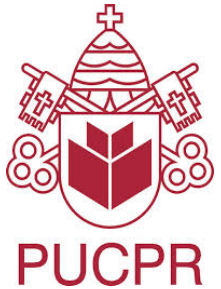



# Chamada de Procedimento Remoto

André Murbach Maidl

Escola Politécnica

Pontifícia Universidade Católica do Paraná

# Objetivos da aula

- Breve revisão sobre redes de computadores
- Discutir a comunicação entre sistemas distribuídos por meio de troca de mensagens
- Apresentar RPC de uma maneira geral
- Apresentar RPC em Go

# Protocolos em camadas

- Devido a ausência de memória compartilhada, toda a comunicação em SDs é baseada na troca de mensagens.
- Aplica o estilo arquitetural das camadas na implementação de pilhas de protocolos.

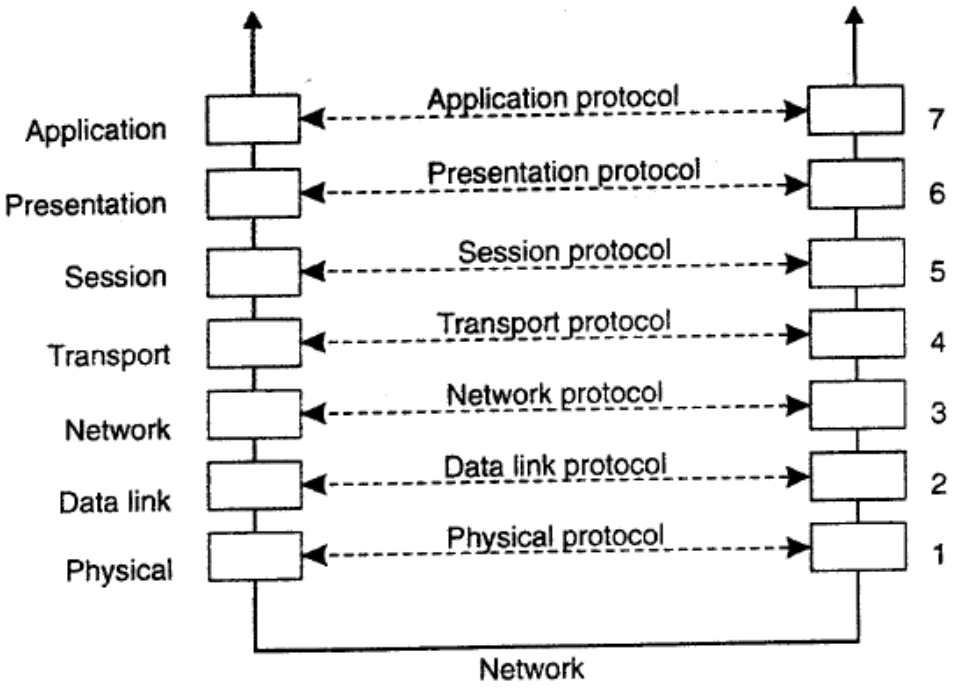



**Figure 4-1.** Layers, interfaces, and protocols in the OSI model.

# Protocolos de baixo nível

- Físico
  - Contém a especificação e a implementação dos bits e a transmissão dos bits entre quem envia e quem recebe.
- Link
  - Prescreve a transmissão de uma série de bits em um *frame* para permitir o controle de fluxo e tratamento de erros.
- Rede
  - Descreve como os pacotes são roteados em uma rede de computadores.

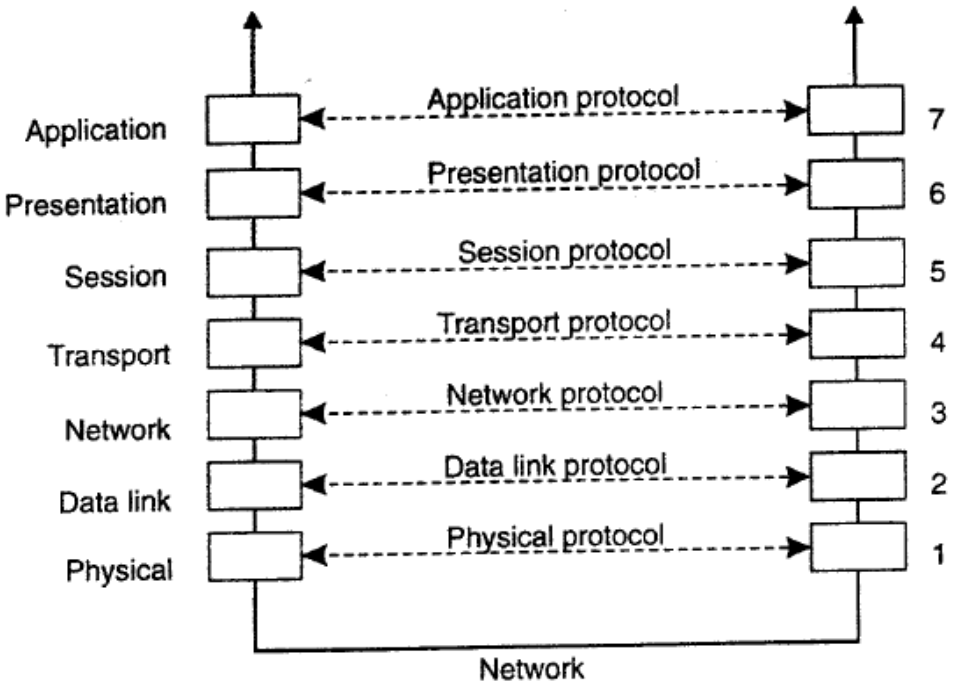



**Figure 4-1.** Layers, interfaces, and protocols in the OSI model.

# Protocolos de transporte

- Disponibilizam os meios de comunicação na maioria dos sistemas distribuídos.
- TCP
  - Orientado a conexão, confiável, comunicação orientada a *streams*.
- UDP
  - Comunicação orientada a datagrama não confiável.

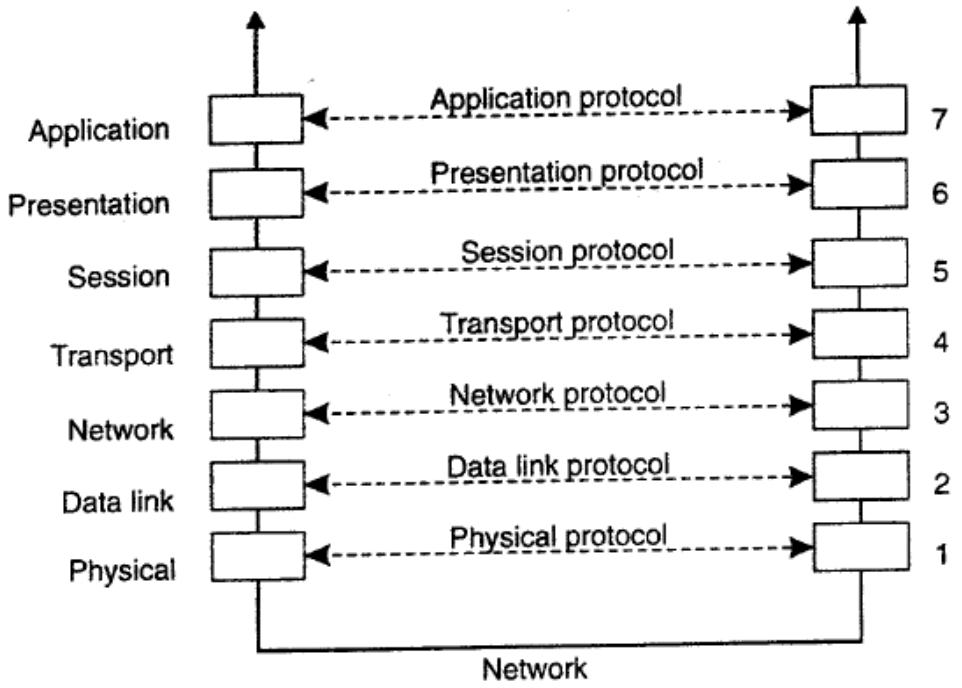



**Figure 4-1.** Layers, interfaces, and protocols in the OSI model.

# Protocolos de alto nível

- Sessão
  - Uma versão aprimorada do protocolo de transporte.
- Apresentação
  - Se preocupa com o significado dos bits em uma mensagem.
- Aplicação
  - O mais usado na prática, pois contém todas as aplicações e protocolos que não se encaixam nas camadas subjacentes.

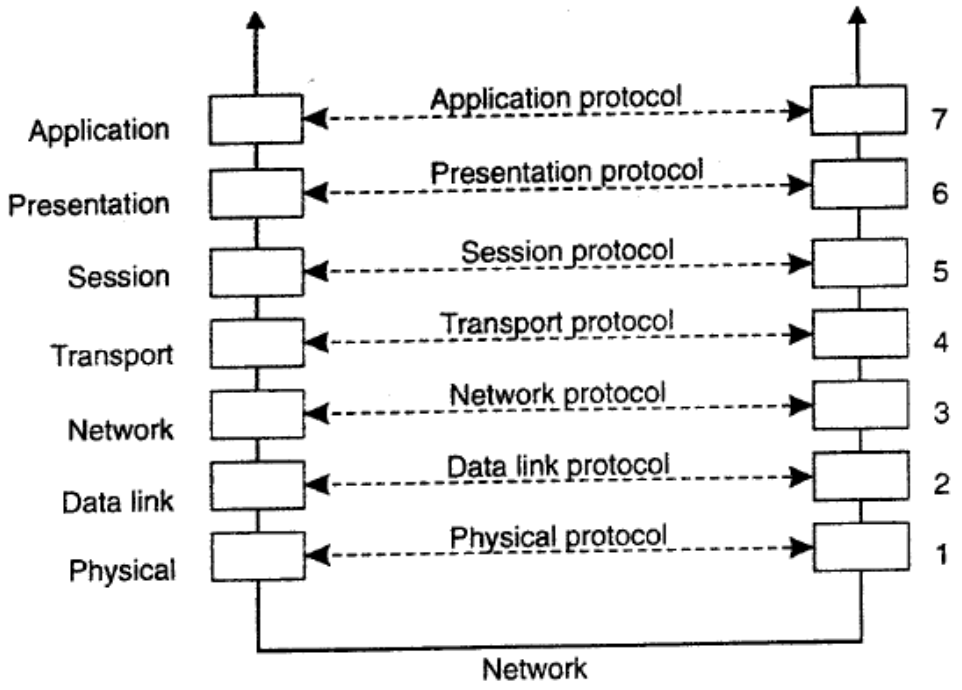



**Figure 4-1.** Layers, interfaces, and protocols in the OSI model.

# Protocolos de *middleware*

- Nem sempre as aplicações se comunicam diretamente por meio dos protocolos de transporte.
  - Por exemplo, Web Services usam SOAP (Simple Object Access Protocol).
  - O SOAP geralmente é encapsulado em HTTP, que por usa vez é encapsulado em TCP.

# Protocolos de *middleware*

- Os protocolos de alto nível (não específicos de uma aplicação) são chamados de **protocolos de *middleware***
- Eles podem suportar serviços como transações, autenticação, autorização e sincronização

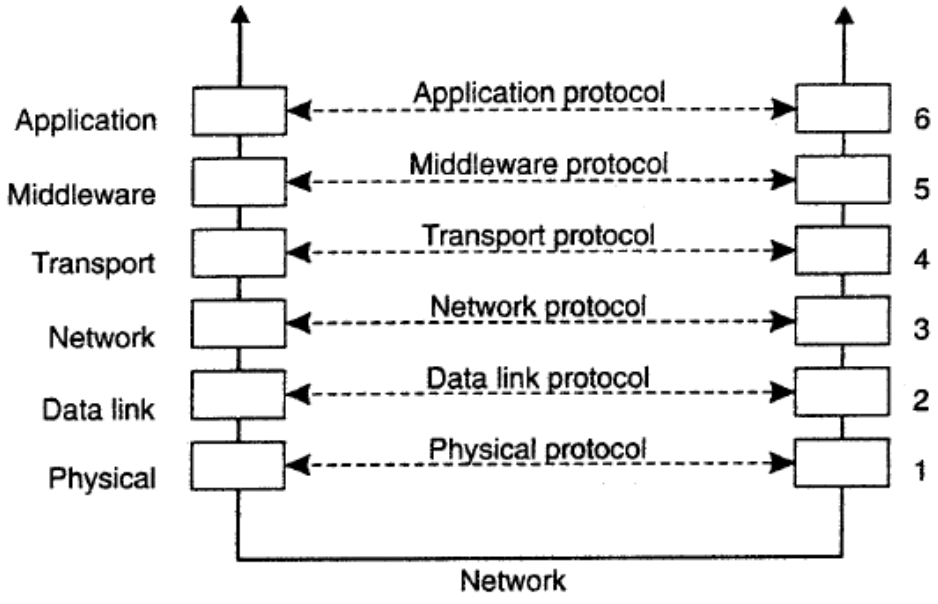
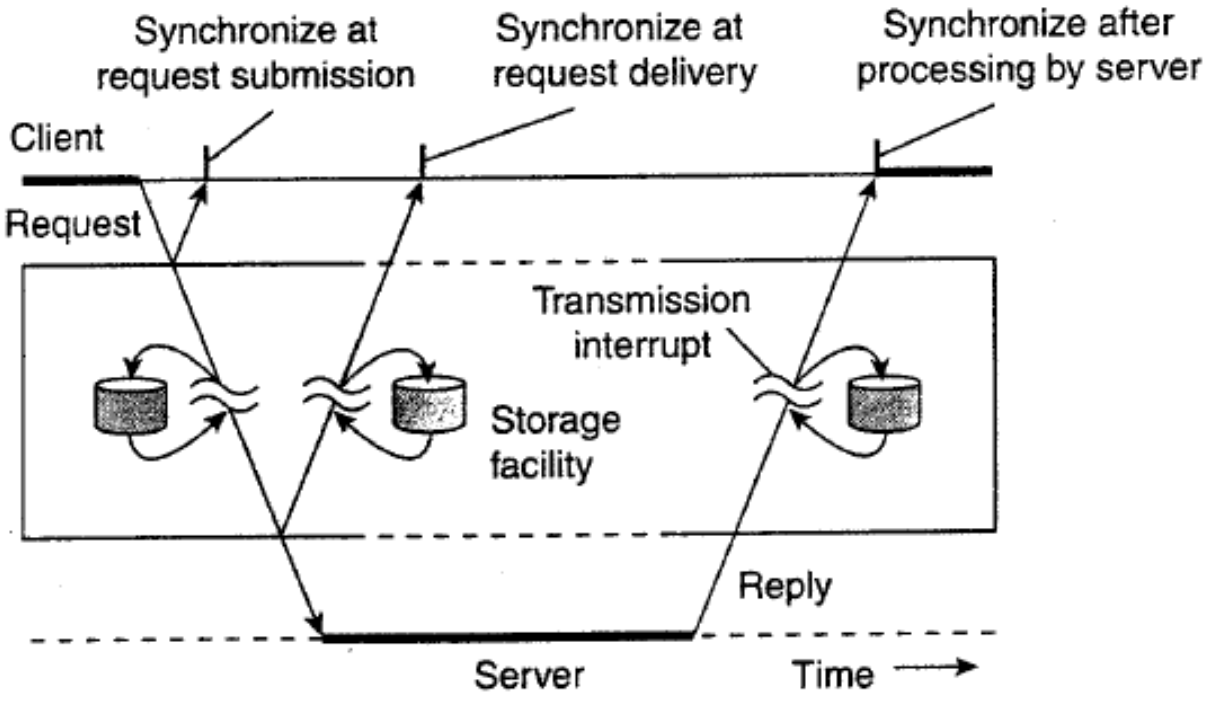



Figure 4-3. An adapted reference model for networked communication.

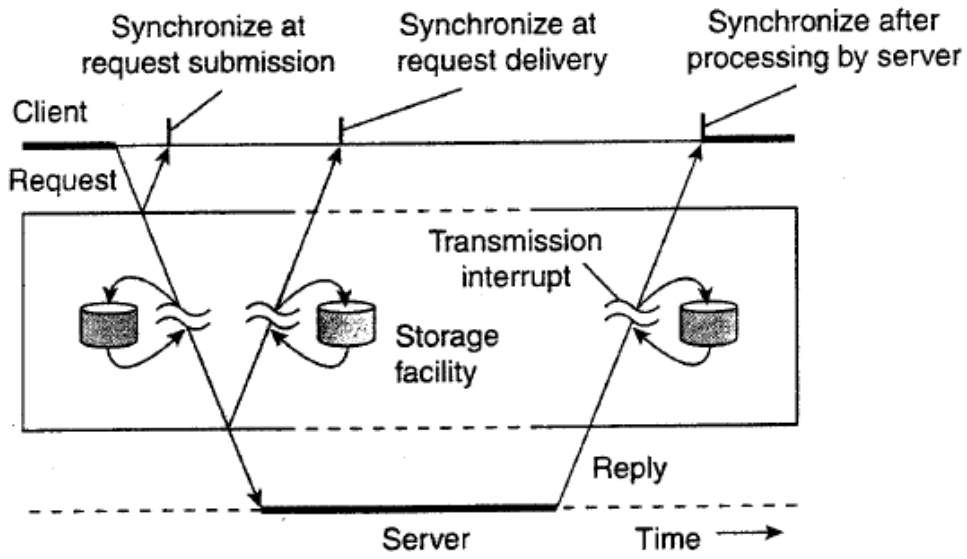
# Tipos de Comunicação




- Comunicação transitória versus persistente
- Comunicação síncrona versus assíncrona

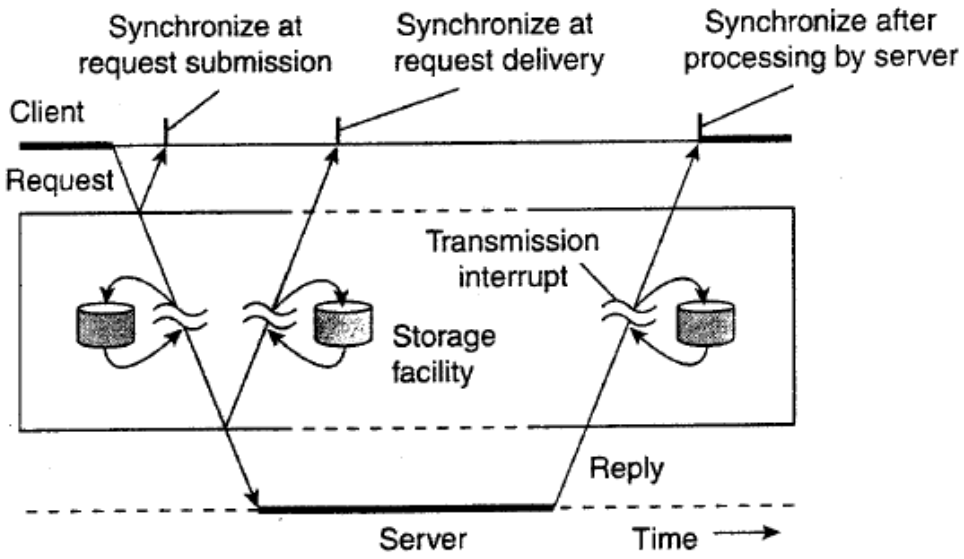
# Tipos de Comunicação

- Transitória
  - A mensagem é armazenada no servidor de comunicação apenas enquanto o emissor e o receptor estão ativos.
- Persistente
  - A mensagem é armazenada no servidor de comunicação até ser entregue ao receptor.



# Tipos de Comunicação

- Síncrona
  - O cliente bloqueia à espera da resposta do servidor.
- Assíncrona
  - O cliente não bloqueia à espera do servidor, e recebe uma notificação quando a resposta está disponível.



# O modelo cliente-servidor

- Geralmente segue o modelo de comunicação transitória e síncrona:
  - Tanto cliente quanto servidor devem estar ativos no momento da comunicação.
  - O cliente envia uma requisição e bloqueia até receber a resposta.
  - O servidor essencialmente apenas espera por requisições e em seguida processa essas requisições.

# Limitações da comunicação síncrona

- O cliente não consegue fazer outros trabalhos enquanto espera pela resposta.
- Erros devem ser resolvidos imediatamente, pois o cliente está esperando.
- Esse modelo pode não ser apropriado para alguns serviços.

# *Middleware* orientado a mensagens

- O seu objetivo é a comunicação persistente e assíncrona:
  - Processos trocam mensagem entre si, as quais são enfileiradas.
  - Quem envia não precisa esperar uma resposta imediata, e pode fazer outras coisas enquanto espera.
  - *Middleware* geralmente garante tolerância a faltas.

# Chamada de procedimentos remotos

- Remote Procedure Calls (RPC)
  - Operações básicas
  - Passagem de parâmetros
  - Variações

# Comunicação por mensagens

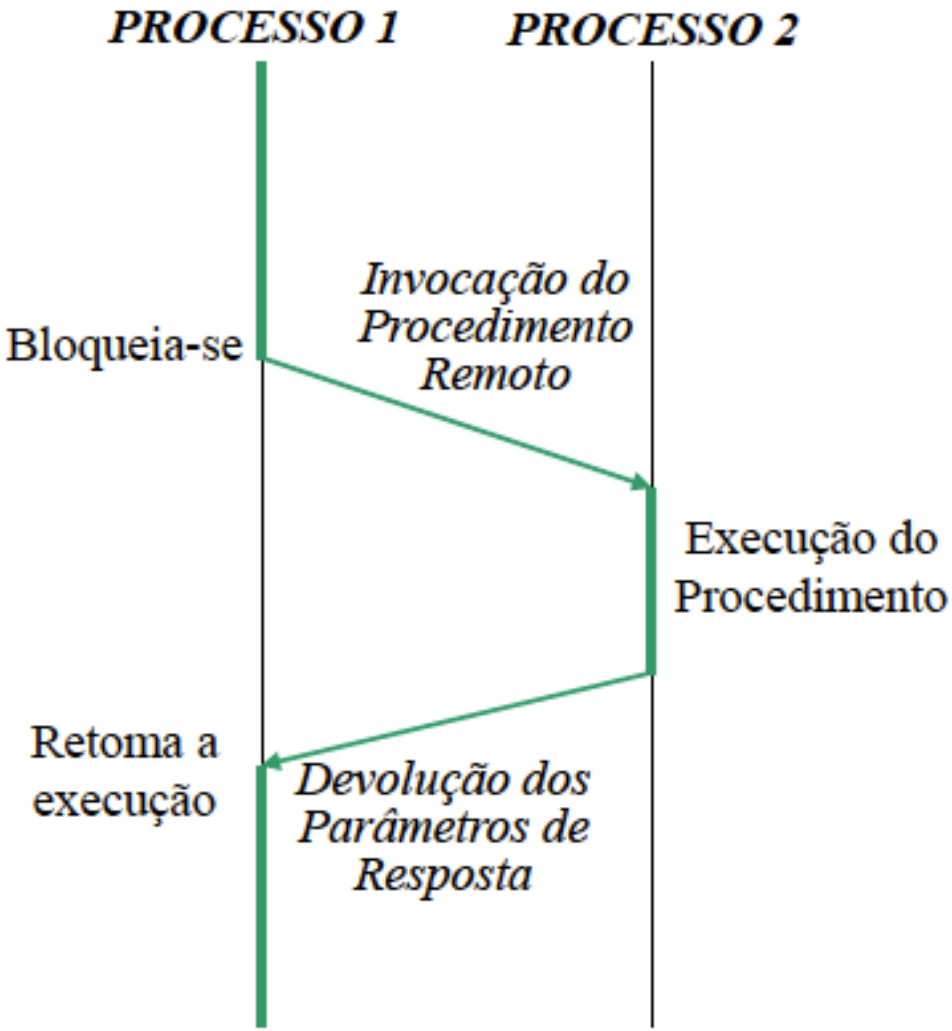
- A comunicação por mensagens tem duas limitações importantes:
  - É difícil e enfadonha de programar.
    - Por exemplo, temos que tratar diferentes formatos de rede e implementar funções tipo `struct2buffer` e `buffer2struct`.
  - Funções como `send` e `receive` não fornecem transparência de localização e de acesso, as quais são objetivos importantes em SDs.

# Comunicação por mensagens

- Além dessas duas limitações importantes, também precisamos inventar o protocolo de transmissão de dados.
  - Pode ser complexo, difícil de manter, pode não interagir bem com outros protocolos, etc.

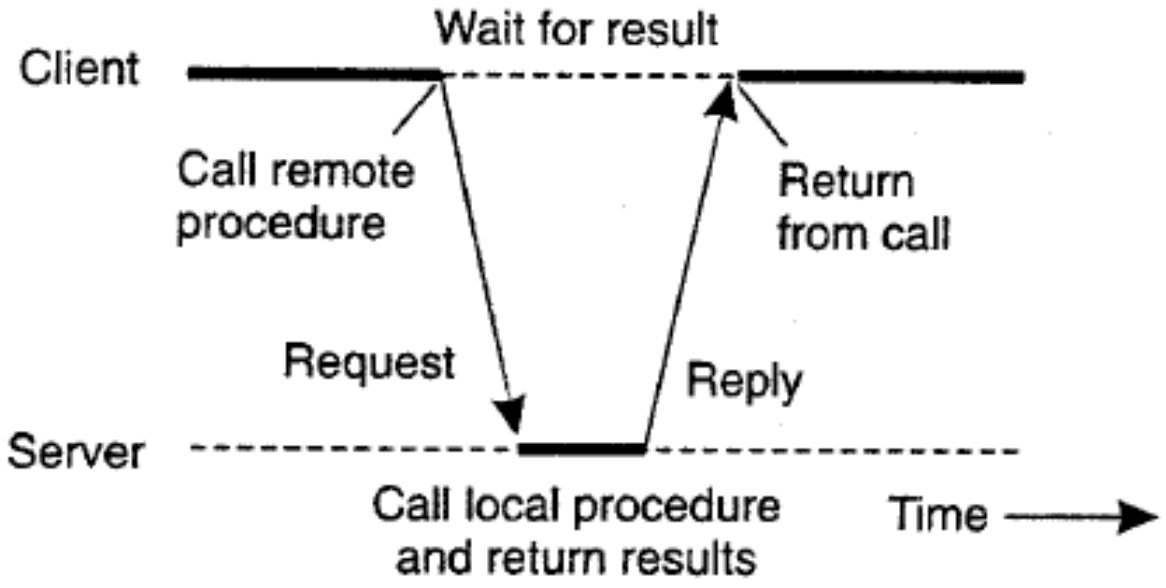
# Chamada de Procedimentos Remotos

- Nas linguagens procedurais existe um fluxo de atividade que chama sincronamente os procedimentos do programa.
- A chamada de procedimentos permite as transferência de controle e dados dentro do programa.



# Chamada de Procedimentos Remotos

- A chamada de procedimentos remotos (RPC) pode ser vista como uma extensão deste modelo para o caso dos sistemas distribuídos.



# Resumindo, RPC é

- um modelo de comunicação ao estilo cliente-servidor, o qual tenta fazer com que chamadas de procedimentos remotos se pareçam com chamadas de procedimentos locais.

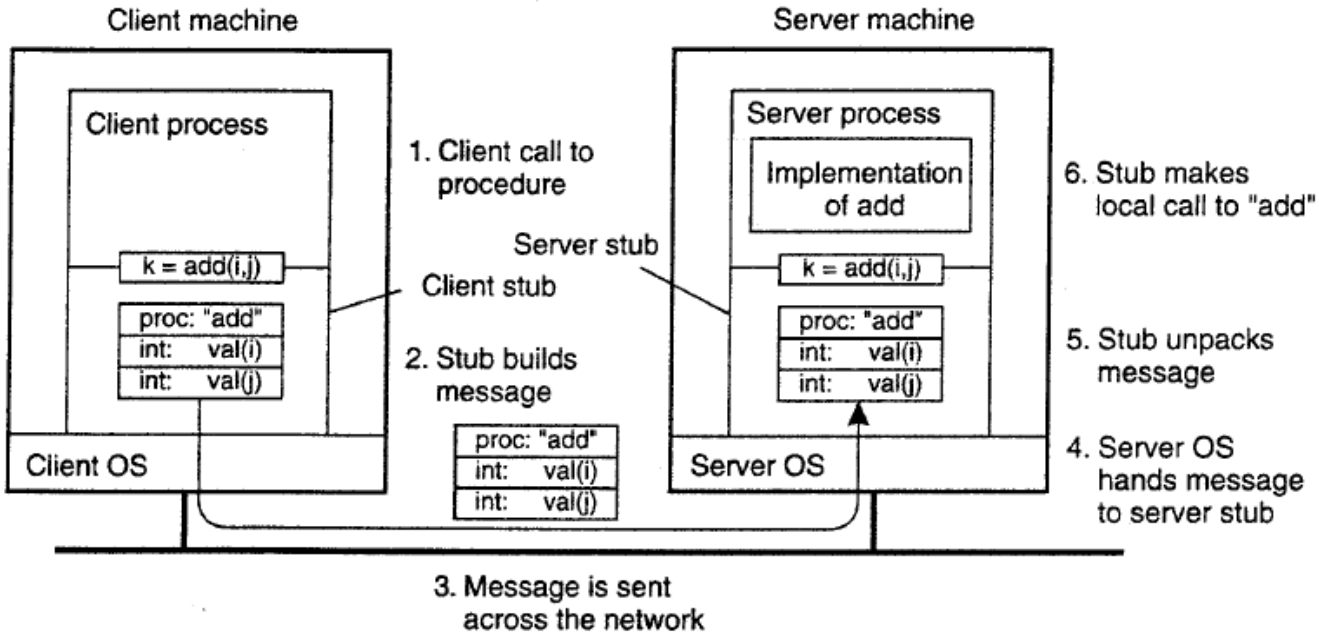
```
/* Cliente */
{
   ...
   resp = add(a, b)
   ...
}
```

```
/* Servidor */
int add (int a, int b)
{
   ...
   return resp
}
```

# Objetivos da RPC

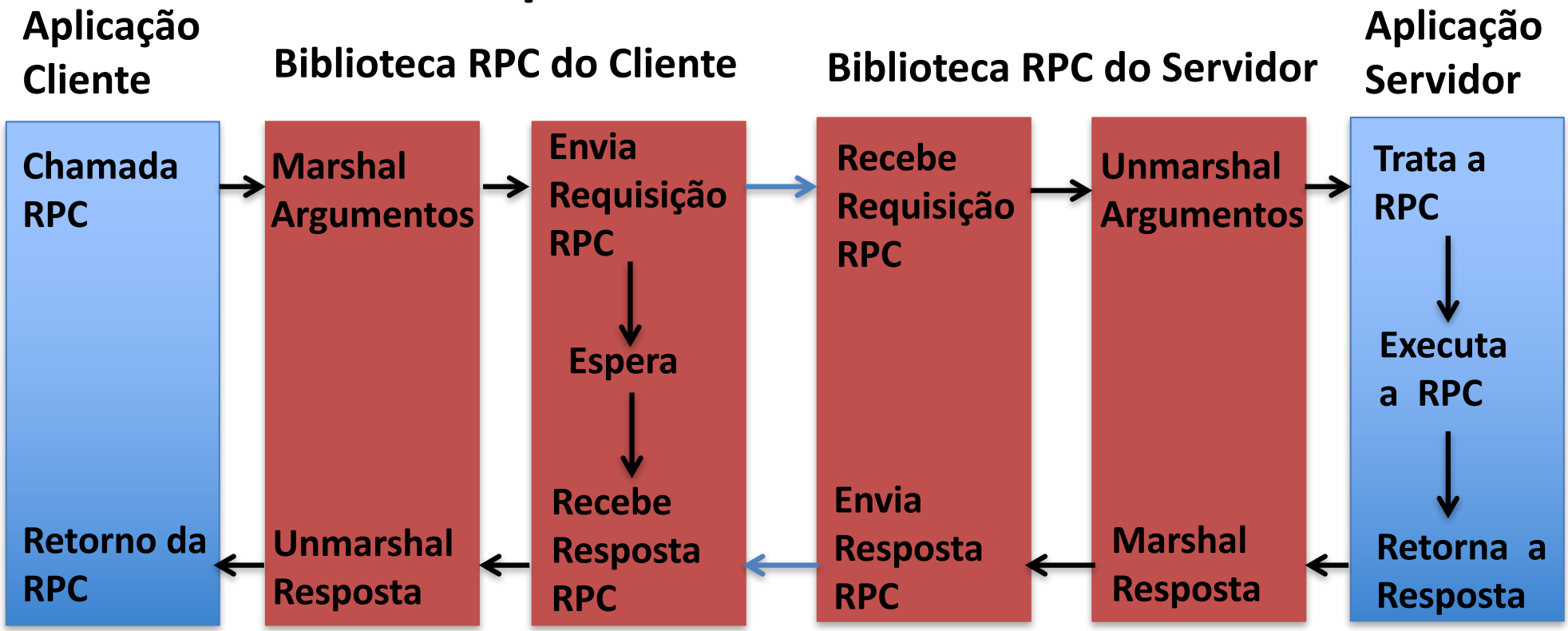
- Facilitar a programação de aplicações cliente-servidor.
  - O modelo é familiar aos programadores, pois basta fazer uma chamada de função.
- Esconder a complexidade quando possível.
- Automatizar diversas tarefas de implementação.
- Padronizar alguns protocolos de empacotamento de dados entre os componentes do sistema distribuído.

# Operações básicas




1. Chamada de rotina de adaptação do cliente (stub).
2. A rotina de adaptação constrói a mensagem e passa-a ao SO.
3. O SO cliente transmite a mensagem pela rede.
4. O SO remoto passa a mensagem à rotina de adaptação do servidor (stub).
5. A rotina desempacota os parâmetros e chama o procedimento local.
6. O servidor executa o pedido e devolve o resultado pela rotina de adaptação.
7. A rotina empacota o resultado em uma mensagem e passa-a ao SO.
8. O SO remoto transmite a mensagem pela rede.
9. O SO cliente passa a mensagem à rotina de adaptação do cliente.
10. A rotina retorna o resultado.

# Arquitetura da RPC




```
/* Cliente */
{
    ...
    resp = add(a, b)
    ...
}
```

```
/* Servidor */
int add (int a, int b)
{
    ...
    return resp
}
```

# Marshaling e Unmarshaling?

- Os procedimentos executam em máquinas diferentes, com diferentes espaços de memória.
  - Isto significa que ponteiros não têm significado, podemos ter diferentes ambientes, diferentes sistemas operacionais, diferentes arquiteturas, etc.
  - Por exemplo: o problema da ordenação dos *bytes*.
    - Se enviamos uma requisição para transferir R$ 1 de uma máquina *little endian*, o servidor pode tentar transferir R$ 16M se a máquina é *big endiam*.

# Marshaling e Unmarshaling?

- Precisamos converter os dados para a representação que será usada localmente.
  - Estruturas, strings, números ponto flutuante, estruturas aninhadas, etc.
  - E estruturas complexas?
    - Alguns sistemas RPC nos permitem enviar e receber listas e mapas, por exemplo.
- Também chamamos de serialização e deserialização.

# Rotinas de adaptação

- Rotinas de adaptação (ou RPC *stubs*) são trechos de código que são gerados automaticamente.
- Apesar das rotinas de adaptação parecerem implementar os procedimentos desejados, elas apenas chamam a biblioteca RPC para que serialize/deserialize os dados e faça a transmissão.
- Como funciona a geração de rotinas de adaptação?

# *Interface Definition Language* (IDL)

- Tipicamente escrevemos a descrição da assinatura de uma função por meio de uma IDL, e então o compilador da biblioteca RPC gera o código da rotina de adaptação.
- Algumas se parecem com C e outras com XML.

# Rotinas de adaptação (cliente)

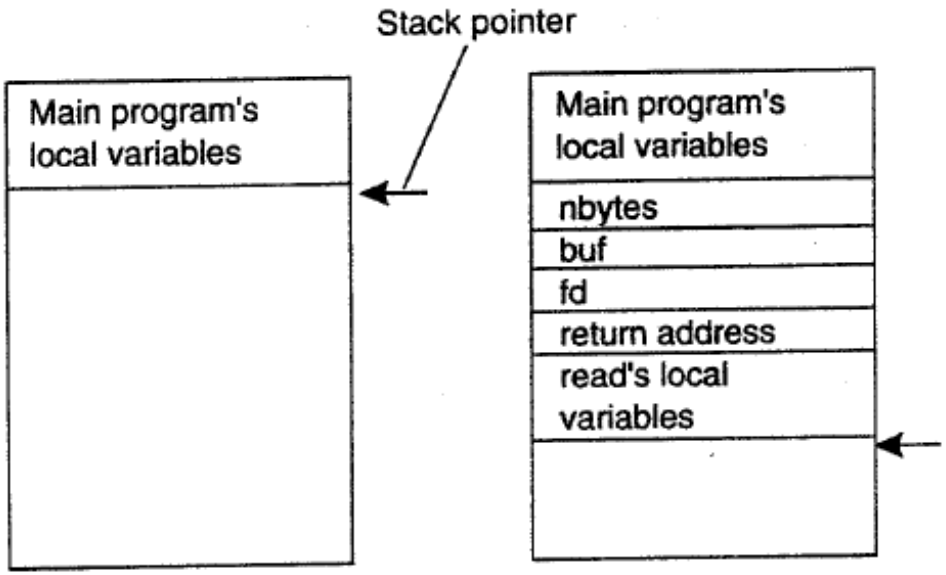
```
int add (int a, int b) {
  /* construir a mensagem pedido P,
     empacotando add, a e b */
  send(..., &P, ...);
  /* bloqueia à espera da mensagem
     de resposta R */
  receive(..., &R, ...);
  /* desempacota a resposta r da
     mensagem de resposta R */
  return r;
}
```

# Rotinas de adaptação (servidor)

```
int main (int argc, char **argv) {
  /* o servidor fica em loop enquanto estiver ativo */
  while(1) {
    /* bloqueia à espera de uma mensagem pedido P */
    receive(..., &P, ...);
    /* desempacota de P o nome do procedimento chamado */
    switch (procedimento) {
      case add: {
        /* desempacota a e b de P antes de
           executar o procedimento local */
        r = add(a, b);
        /* empacota r em na mensagem de resposta R */
        send(..., &R, ...);
        break;
      /* tratamento de outros procedimentos */
      }
    }
  }
  return 0;
}
```

# Revisão de chamada de procedimentos

- Considere a seguinte chamada em C:
  - `count = read(fd, buf, nbytes);`



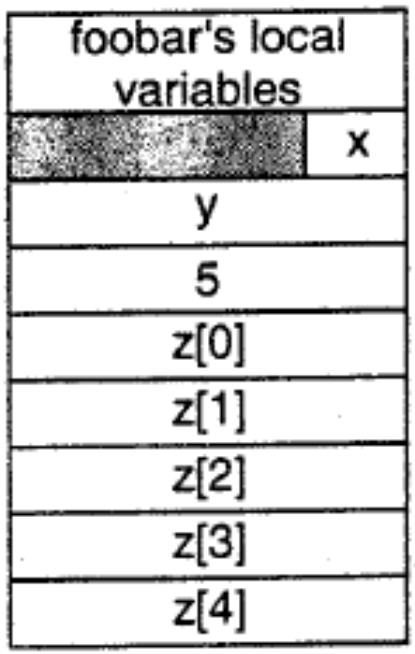

- Chamada ao procedimento
  - Os argumentos são colocados na pilha pela ordem inversa.
  - O procedimento é executado.
  - O endereço de retorno é usado para devolver o controle ao procedimento que fez a chamada.
- Tipos de argumentos
  - Chamada por valor
  - Chamada por referência

# Empacotamento dos parâmetros

- Para que a RPC funcione é fundamental que cliente e servidor concordem sobre o formato das mensagens de pedido e resposta.
- As mensagens são transmitidas *byte* a *byte*, em série.
- Devem concordar especialmente sobre a estrutura da mensagem e o formato dos dados (*boolean, int, float, string, etc*).

```
foobar( char x; float y; int z[5] )
{
   ....
}
```

(a)




(b)

Figure 4-9. (a) A procedure. (b) The corresponding message.

# Empacotamento dos parâmetros

- Para que a RPC funcione é fundamental que cliente e servidor concordem sobre o formato das mensagens de pedido e resposta.
- As mensagens são transmitidas *byte* a *byte*, em série.
- Devem concordar especialmente sobre a estrutura da mensagem e o formato dos dados (*boolean, int, float, string, etc*).

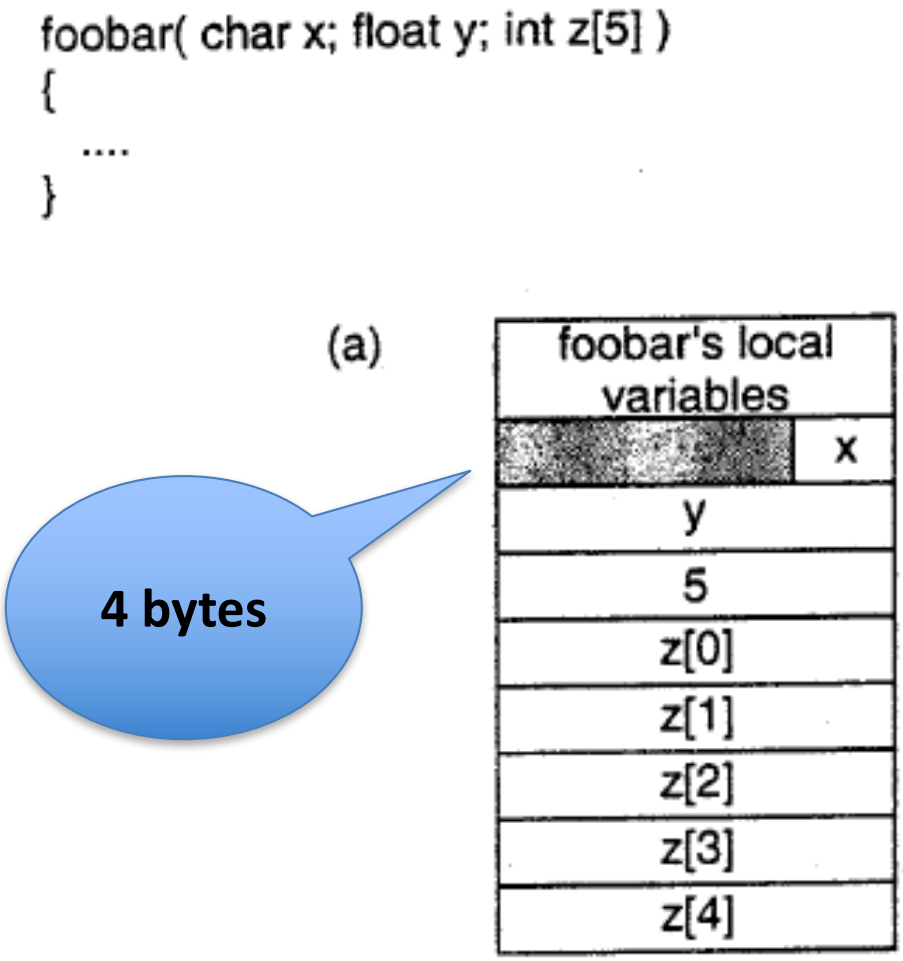


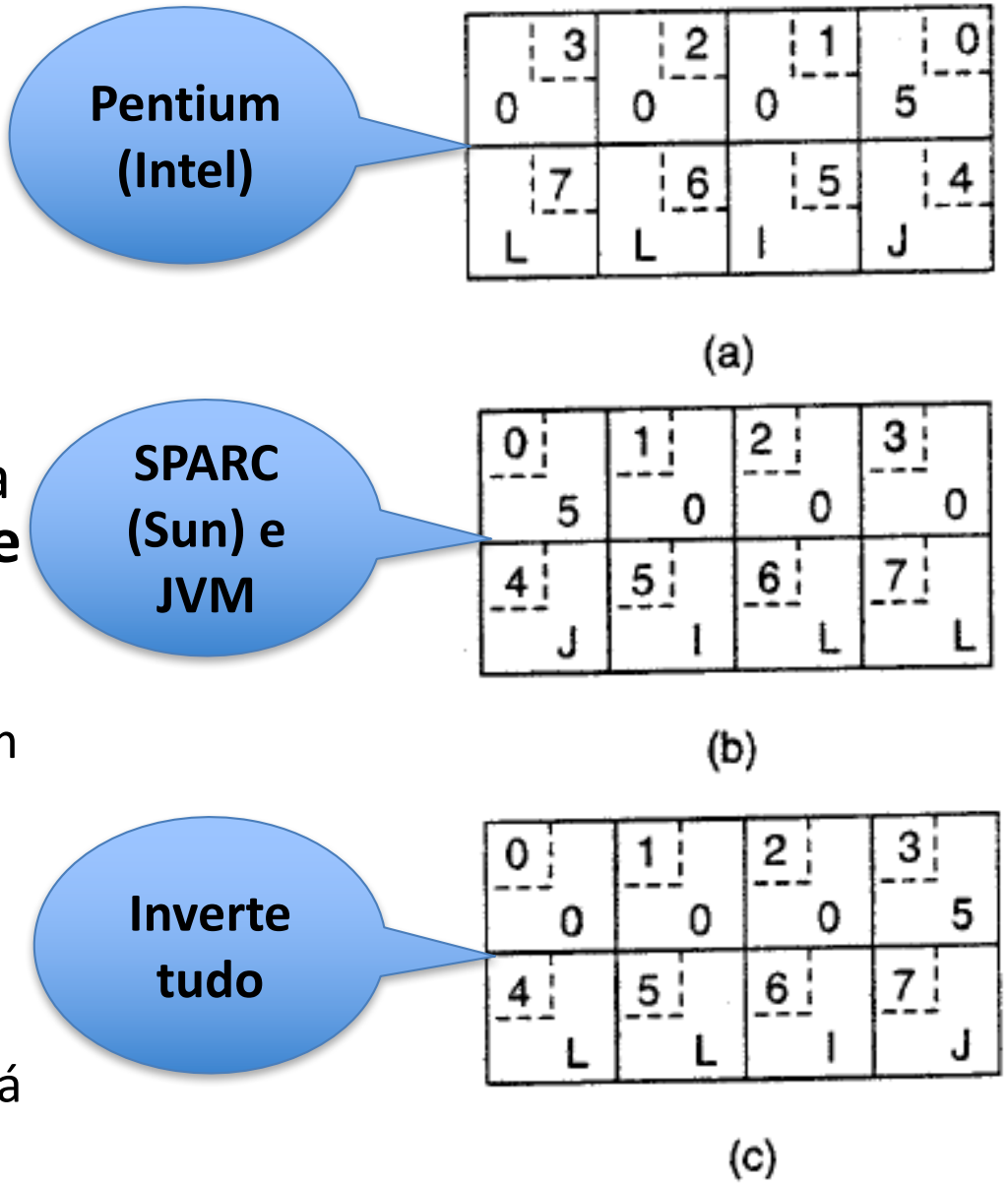

Figure 4-9. (a) A procedure. (b) The corresponding message.

# Passagem de parâmetro por valor

- Cópias dos parâmetros são enviadas para o servidor.
- Temos que tomar cuidado com a **representação dos dados**.
- Problemas
  - Interpretação do servidor dos *bytes* recebidos.
  - Conversão dos dados para o formato do servidor.
- Soluções
  - A existência de um acordo sobre o tamanho dos tipos básicos (`int` = 32 bits, `char` = 8 bits, etc)
  - E o formato dos dados?

# Passagem de parâmetro por valor

- Exemplos de formatos de dados:
  - `char` é ASCII (C) ou unicode (Java)?
  - `float` segue o padrão IEEE 754?
- **Big Endian** (*bytes* da direita para esquerda) versus **Little Endian** (*bytes* da esquerda para a direita)
  - Exemplo: enviar a mensagem <5,"JILL">.
  - As mensagens são enviadas bit a bit no nível físico.
  - Se não invertermos há problema (fig. b) e se invertermos tudo também há problema (fig. c).



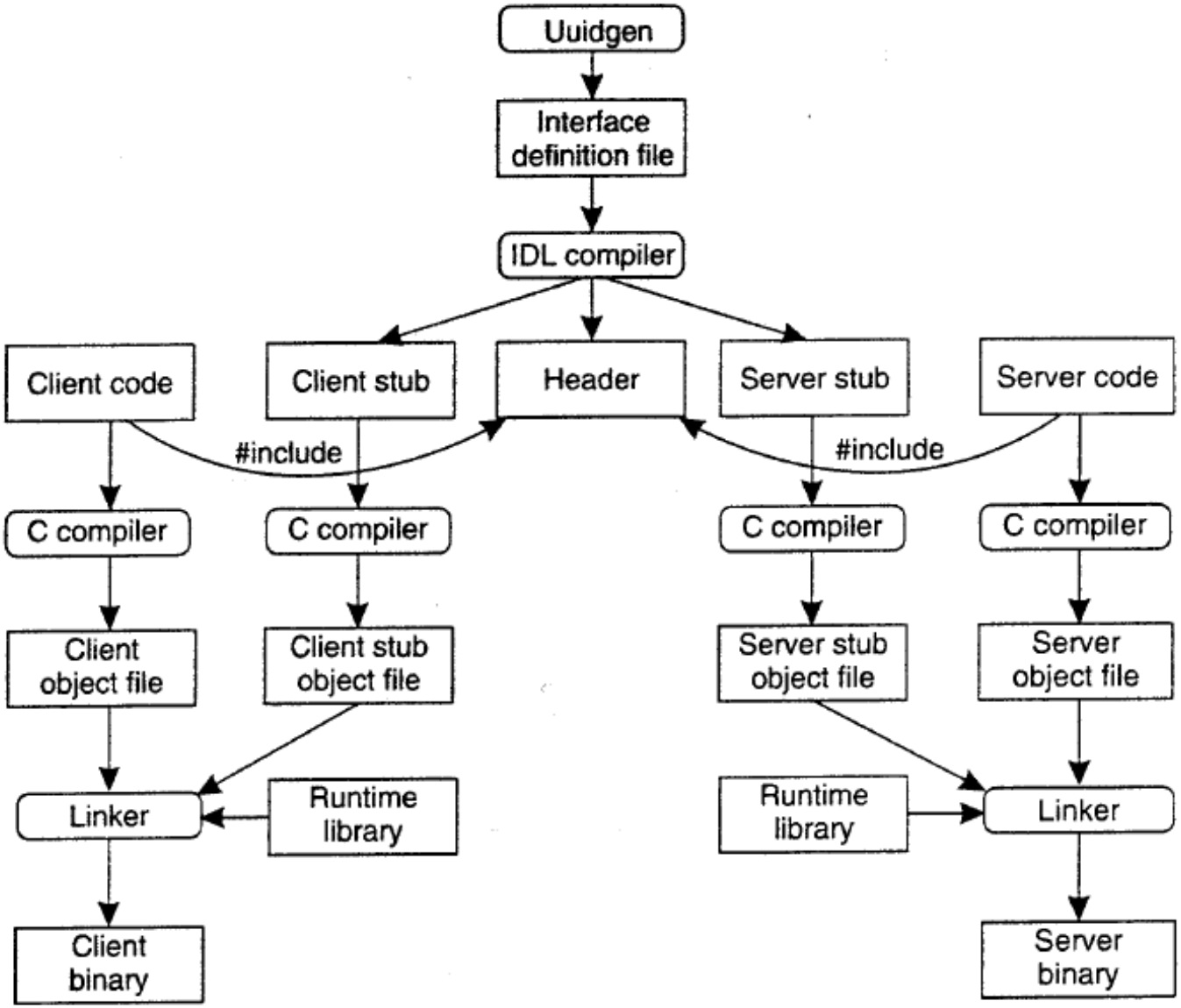
# Passagem de parâmetro por referência

- Ponteiros para regiões de memória.
- Problema
  - Os endereços das estruturas no cliente não têm significado válido no servidor.
- Solução genérica
  - Desambiguar a definição de certos tipos de dados, por exemplo, ponteiros genéricos (void *), cadeias de caracteres ou vetores de tamanho variável, por meio do enriquecimento da interface das rotinas de adaptação.

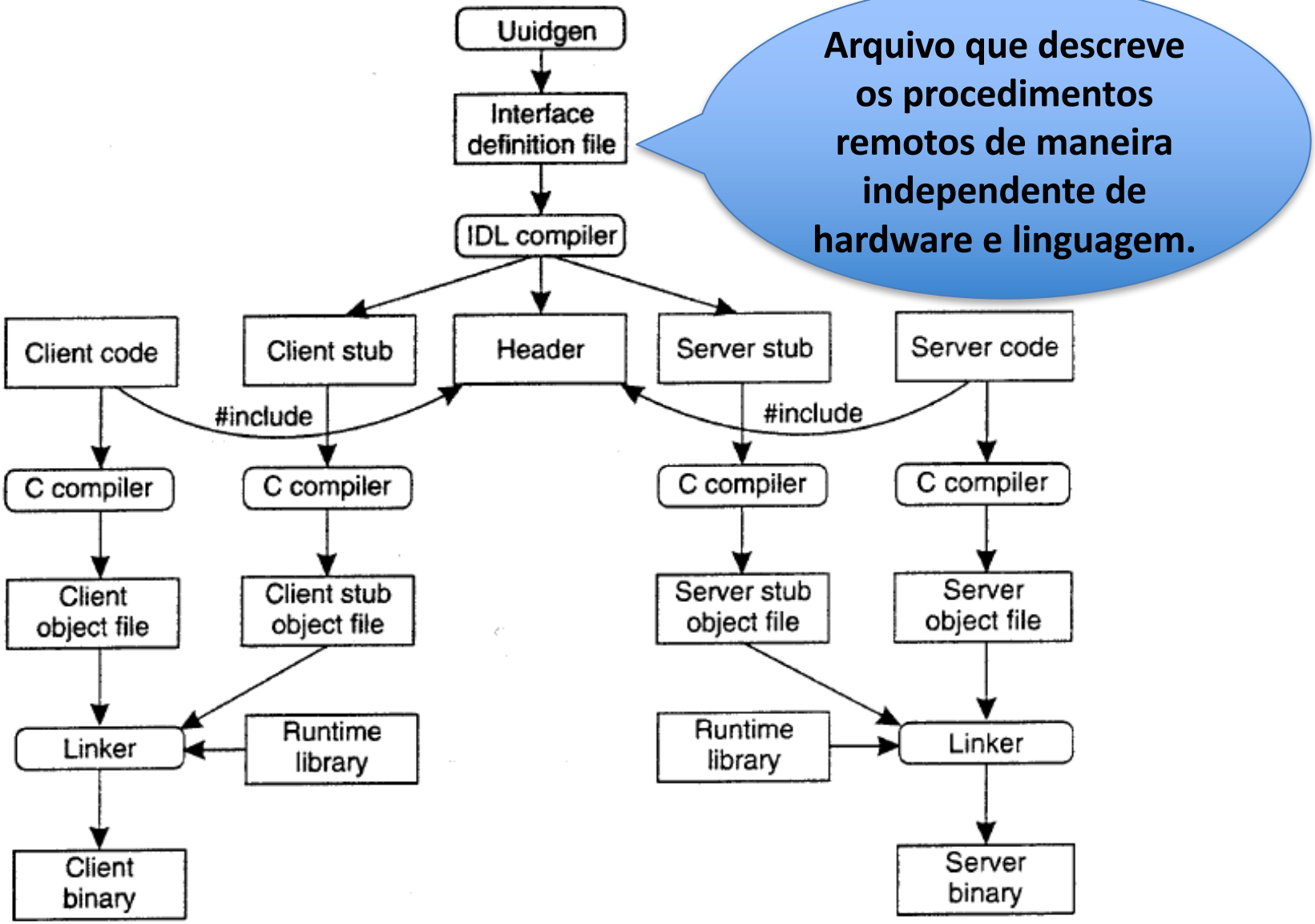
# Passagem de parâmetro por referência

- Solução para cada tipo de referência/ponteiro:
  - Tipos básicos
    - Passa-se o dado referenciado em vez do ponteiro
    - Por exemplo, enviamos um `int` se tivermos um `int*`
  - Vetor de tamanho fixo
    - Declara-se explicitamente o tamanho do vetor
    - Por exemplo, usamos `int[10]` declara um vetor de 10 posições
  - Estruturas
    - A definição da estrutura deve estar visível à interface
  - Listas e árvores
    - Não existe uma boa solução
    - Uma possibilidade consiste em chamar recursivamente a rotina de adaptação para linearizar a estrutura de dados, ou seja, transforma-la em um vetor de *bytes*)
- Os parâmetros passados por referência são alterados no servidor e depois retornados de volta ao cliente juntamente com a resposta.

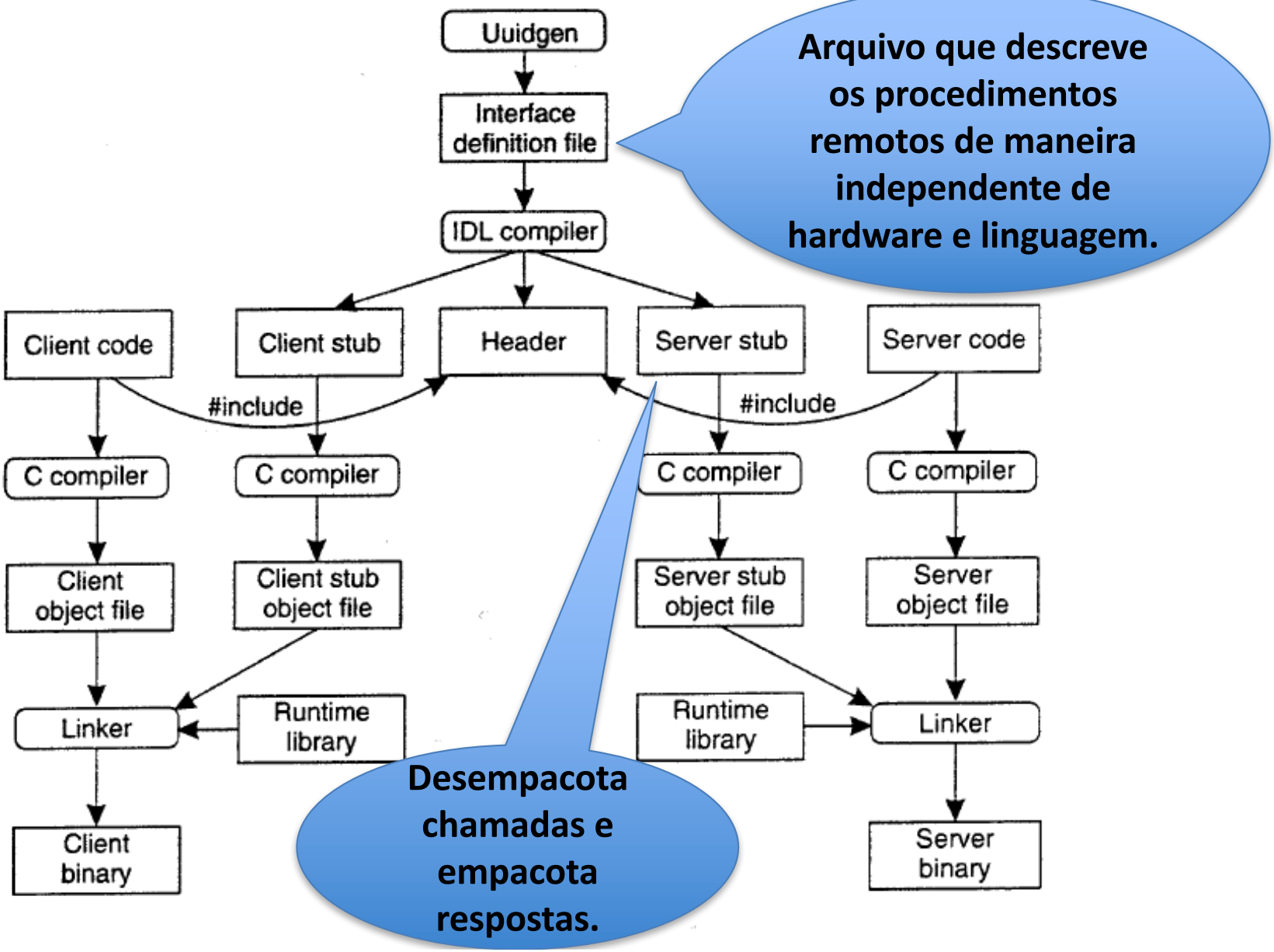
# Geração de rotinas de adaptação



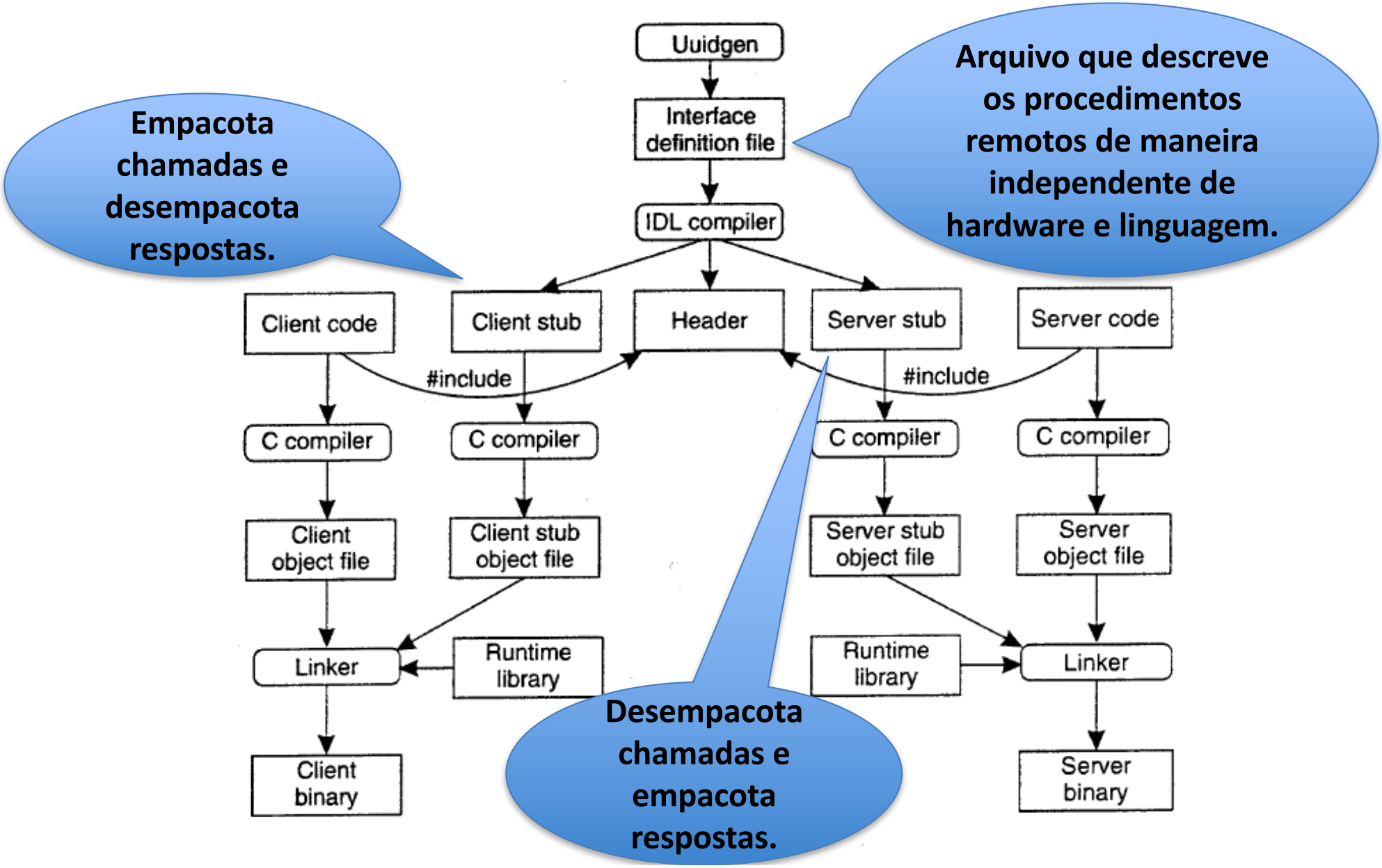
# Geração de rotinas de adaptação



# Geração de rotinas de adaptação



# Geração de rotinas de adaptação



# RPC assíncrona

- As RPCs assíncronas aumenta o desempenho das RPCs síncronas, pois evitam o bloqueio do cliente enquanto o pedido está sendo processado.

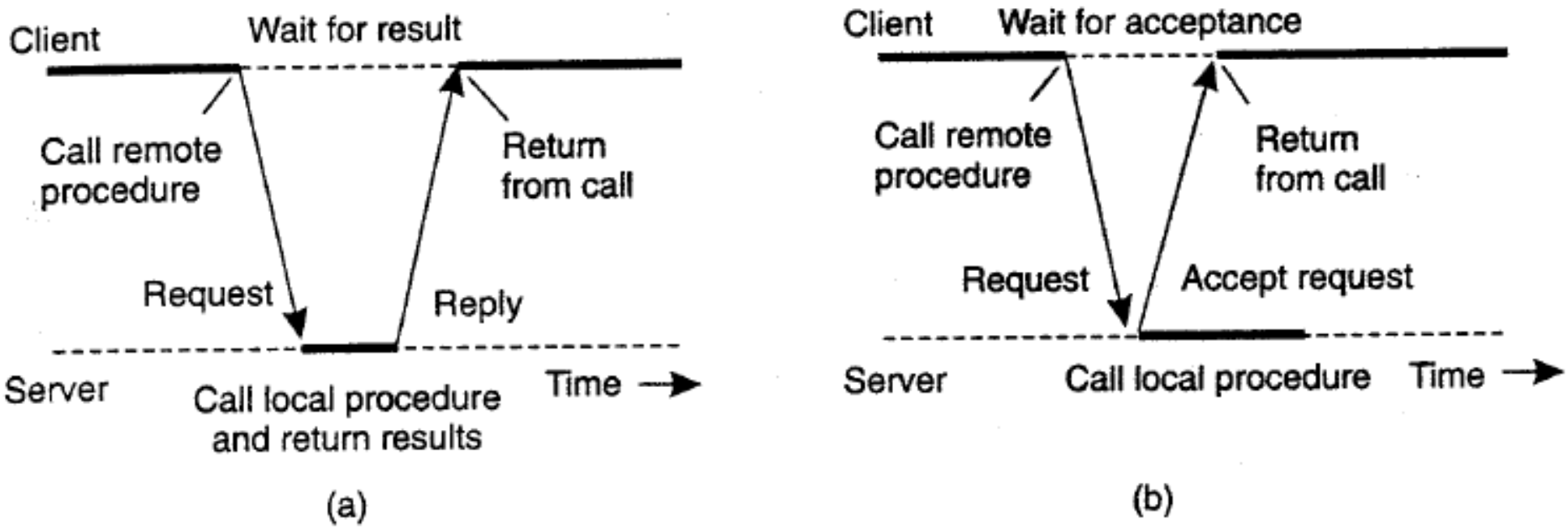



Figure 4-10. (a) The interaction between client and server in a traditional RPC. (b) The interaction using asynchronous RPC.

# RPC assíncrona

- Existem variantes assíncronas em que o cliente não espera pela confirmação de aceitação da chamada.

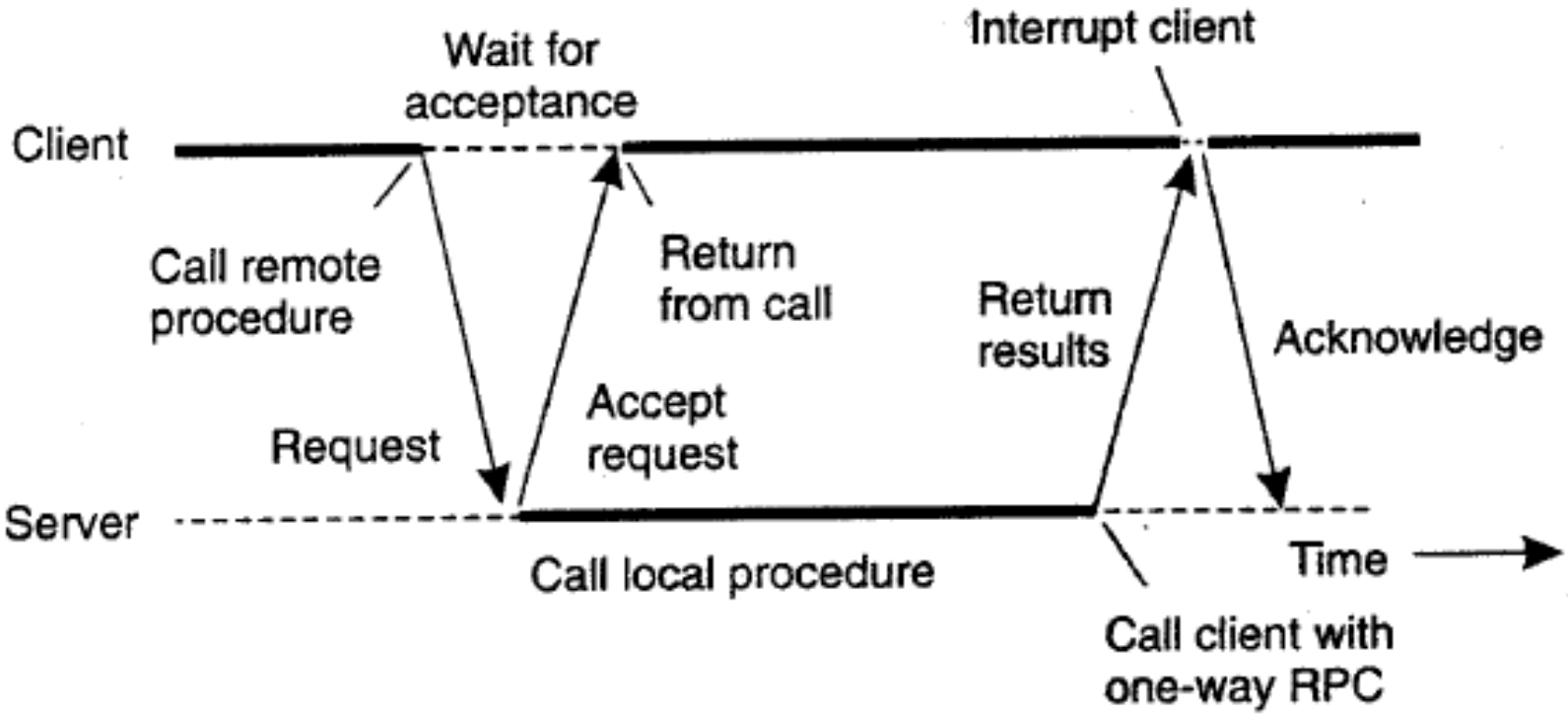



Figure 4-11. A client and server interacting through two asynchronous RPCs.

# RPC na prática

- Programação com *sockets* usa um paradigma orientado a mensagens.
- Um cliente envia uma mensagem para um servidor, o qual geralmente envia uma outra mensagem de volta.
- Ambos os lados devem concordar no formato da mensagem.

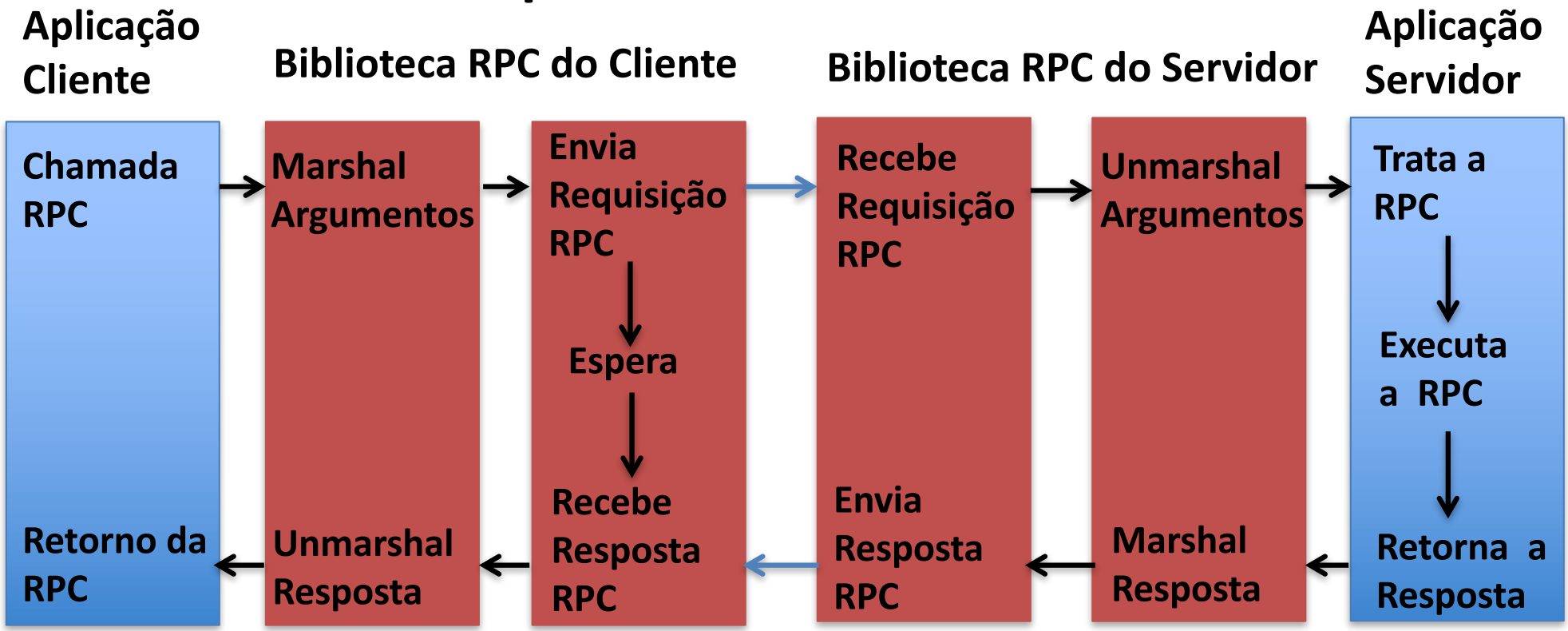
# RPC na prática

- No entanto, a maioria das aplicações locais não usa o mecanismo de troca de mensagens.
- Em geral, o mecanismo preferido é chamar funções/procedimentos.
- Ou seja, um programa chama uma função com uma lista de parâmetros para obter determinados resultados/valores de retorno.

# Chamada de procedimento remoto

- Uma chamada de procedimento remoto (RPC – *Remote Procedure Call*) é uma maneira de trazer esse estilo de programação para sistemas distribuídos.

# Arquitetura da RPC




```
/* Cliente */
{
    ...
    resp = add(a, b)
    ...
}
```

```
/* Servidor */
int add (int a, int b)
{
    ...
    return resp
}
```

# Estilos de implementação

- Existem dois estilos de implementar RPC:
  - Especificação do serviço via uma linguagem abstrata;
  - Especificação de uma API especial que deve ser usada no lado do cliente.

# Linguagem abstrata

- A descrição do serviço é dada em uma linguagem abstrata, por exemplo, CORBA IDL.
  - Exemplos: Sun RPC, CORBA e Java RMI.
- Esta especificação é complicada e gera código para as rotinas de adaptação do cliente e do servidor, conforme vimos anteriormente.
- Em Java RMI, a própria linguagem funciona como a linguagem abstrata.

# Tecnologias

- XML/RPC
  - Sobre HTTP e precisar tratar XML.
- SOAP
  - Padrão para Web Services via HTTP, e também deve tratar XML.
- CORBA
  - Extremamente complexo e pesado.
- COM
  - Geralmente usado em clientes Windows.
- Protocol Buffers
  - Desenvolvido por Google.
- Thrift
  - Desenvolvido por Facebook.

# API no cliente

- Usamos uma API especial no cliente.
  - Exemplos: alguns Web Services e Go.
- No lado do cliente fazemos as chamadas aos procedimentos com os respectivos parâmetros.
- No lado do servidor implementamos os procedimentos.
- Os dois lados devem concordar sobre as estruturas que são exportadas.

# RPC em Go

- As chamadas de procedimentos remotos em Go são bem diferentes de outras linguagens.
- Isso significa que um cliente Go só consegue chamar procedimentos de um servidor Go.
- Um dos motivos é que Go usa uma biblioteca de serialização específica para Go, a qual é chamada `gob`.
- Em uma próxima aula veremos como escapar dessa restrição.

# Restrições

- Sistemas de RPC geralmente fazem algumas restrições nas funções que podem ser chamadas pela internet, pois esses sistemas precisam determinar quais são os valores dos argumentos enviados, quais são as referências dos argumentos que devem receber as respostas e como sinalizar erros.

# Restrições em Go

- Em Go, apenas os métodos que seguem os seguintes critérios são disponibilizados para serem acessados remotamente:
  1. O tipo do método foi exportado;
  2. O método foi exportado;
  3. O método possui dois argumentos, e ambos os tipos foram exportados (mesmo que sejam tipos nativos);
  4. O segundo parâmetro do método é um ponteiro;
  5. O tipo de retorno do método é `error`.

# O padrão em si

- Resumindo, um método deve seguir o seguinte padrão:

```
func (this *T) MethodName (arg T1, reply *T2) error
```

# Exemplo

- Operações aritméticas usando RPC.

# Servidor

```
package main

import (
  "fmt"
  "net/rpc"
  "errors"
  "net"
  "os"
)

type Args struct {
  A, B int
}

type Quotient struct {
  Q, R int
}

type Arith int
```

# O procedimento `Multiply`

```
func (t *Arith) Multiply (args *Args,
                          reply *int) error {
  *reply = args.A * args.B
  return nil
}
```

# O procedimento `Divide`

```
func (t *Arith) Divide (args *Args,
                        reply *Quotient) error {
  if args.B == 0 {
    return errors.New("divide by zero")
  }
  reply.Q = args.A / args.B
  reply.R = args.A % args.B
  return nil
}
```

# A função `main`

```
func main () {
  arith := new(Arith)
  rpc.Register(arith)
  tcpAddr, err :=
    net.ResolveTCPAddr("tcp", "localhost:1234")
  checkError("ResolveTCPAddr: ", err)
  listener, err :=
            net.ListenTCP("tcp", tcpAddr)
  checkError("ListenTCP: ", err)
  for {
    conn, err := listener.Accept()
    checkError("Accept: ", err)
    go rpc.ServeConn(conn)
  }
}
```

# Cliente

```
package main

import (
   "net/rpc"
   "fmt"
   "os"
)

type Args struct {
   A, B int
}

type Quotient struct {
   Q, R int
}
```

# A função `main`

```
func main () {
  service := "localhost:1234"
  client, err := rpc.Dial("tcp", service)
  defer client.Close()
  checkError("Dial: ", err)
  fmt.Println("* - multiplicação")
  fmt.Println("/ - divisão")
  var op byte
  fmt.Scanf("%c\n", &op)
  switch op {
    /* código para implementar os
       procedimentos disponíveis */
  }
}
```

# O `switch` internamente

```
case '*':
  args := readArgs()
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  checkError("Multiply: ", err)
  fmt.Printf("%d * %d = %d\n",
              args.A, args.B, reply)
  os.Exit(0)
case '/':
  args := readArgs()
  var reply Quotient
  err = client.Call("Arith.Divide", args, &reply)
  checkError("Divide: ", err)
  fmt.Printf("%d / %d = (%d,%d)\n",
              args.A, args.B, reply.Q, reply.R)
  os.Exit(0)
default:
  fmt.Println("Opção inválida: ", op)
  os.Exit(1)
```

# A função `readArgs`

```
func readArgs () Args {
  var a, b int
  fmt.Println("A: ")
  fmt.Scanln(&a)
  fmt.Println("B: ")
  fmt.Scanln(&b)
  return Args{a, b}
}
```

# Exercícios

- Rode o servidor RPC na *espec* e o cliente RPC em outra máquina.
  - Lembre que você precisará configurar o túnel para que seja possível conectar-se ao servidor na *espec*.
- Adapte o código do servidor e do cliente para incluir as operações de soma e subtração via chamada de procedimento remoto.

# Cliente assíncrono

- Como podemos transformar o nosso cliente de síncrono para assíncrono?

# Cliente assíncrono

- Como podemos transformar o nosso cliente de síncrono para assíncrono?
  - Basta usarmos o método `Go` para efetuarmos a chamada do procedimento remoto, como se fosse uma *goroutine,* e canais para verificarmos quando ela retorna.

# O `switch` assíncrono internamente

```
case '*':
  args := readArgs()
  var reply int
  mulCall := client.Go("Arith.Multiply",
                       args, &reply, nil)
  replyMul := <- mulCall.Done
  checkError("Multiply: ", replyMul.Error)
  fmt.Printf("%d * %d = %d\n",
             args.A, args.B, reply)
  os.Exit(0)
case '/':
  args := readArgs()
  var reply Quotient
  divCall := client.Go("Arith.Divide",
                       args, &reply, nil)
  replyDiv := <- divCall.Done
  checkError("Divide: ", replyDiv.Error)
  fmt.Printf("%d / %d = (%d,%d)\n",
             args.A, args.B, reply.Q, reply.R)
  os.Exit(0)
```

# Exercícios

- Adapte o código do cliente síncrono para que seja assíncrono. Depois execute o cliente assíncrono para que efetue uma chamada de procedimento remoto ao servidor.
- Implemente o jogo *pedra, papel, tesoura* para que funcione ao estilo RPC.
  - Isto é, crie um cliente que permite jogar contra um servidor e a jogada é executada via uma chamada de procedimento remoto.