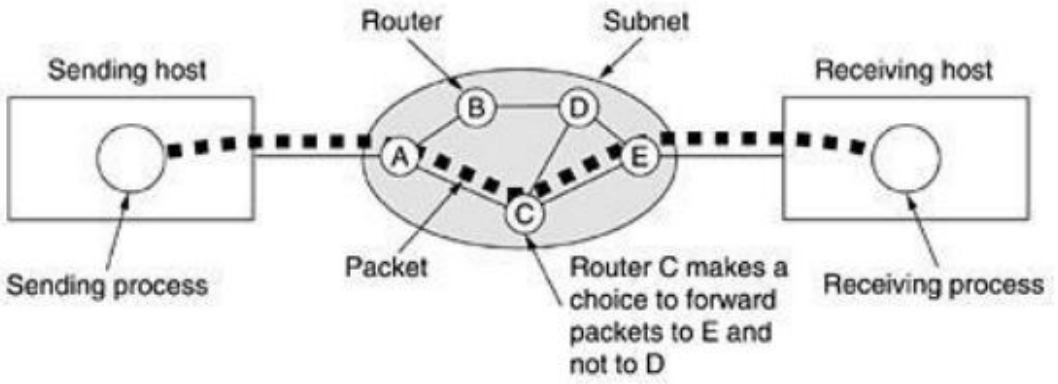
# Comunicação entre Processos em Sistemas Distribuídos

Slides Baseados no material dos professores FELIPE CUNHA, HUGO DE PAULA e GLEISSON

# Introdução aos serviços de rede

## Esquemas de comutação

- Broadcast
  - Tudo é enviado para todos os nodos.
- Comutação de circuitos
  - Estabelecimento de canais dedicados de comunicação
- Comutação de pacotes
  - Sistema de armazenamento e encaminhamento de pacotes com base nas informações de origem e destino.



# Introdução aos serviços de rede

Modelo cliente/servidor:

- Servidor:
  - programa que provê um serviço, torna algum recurso disponível a outros programas em qualquer lugar da rede.
  - Servidor é passivo. Normalmente é inicializado em tempo de boot e fica esperando uma requisição de uso dos recurso que ele controla.

# Introdução aos serviços de rede

Modelo cliente/servidor:

- Cliente:
  - Programa que usa o recurso disponibilizado pelo servidor, não importando a localização.
  - Procura ativamente na rede onde o recurso está.
- Servidores e clientes podem estar na mesma máquina ou em qualquer lugar da rede.

# Introdução aos serviços de rede

Modelo cliente/servidor: visa justamente permitir que um determinado recurso seja disponibilizado através da rede a qualquer aplicação que precise.

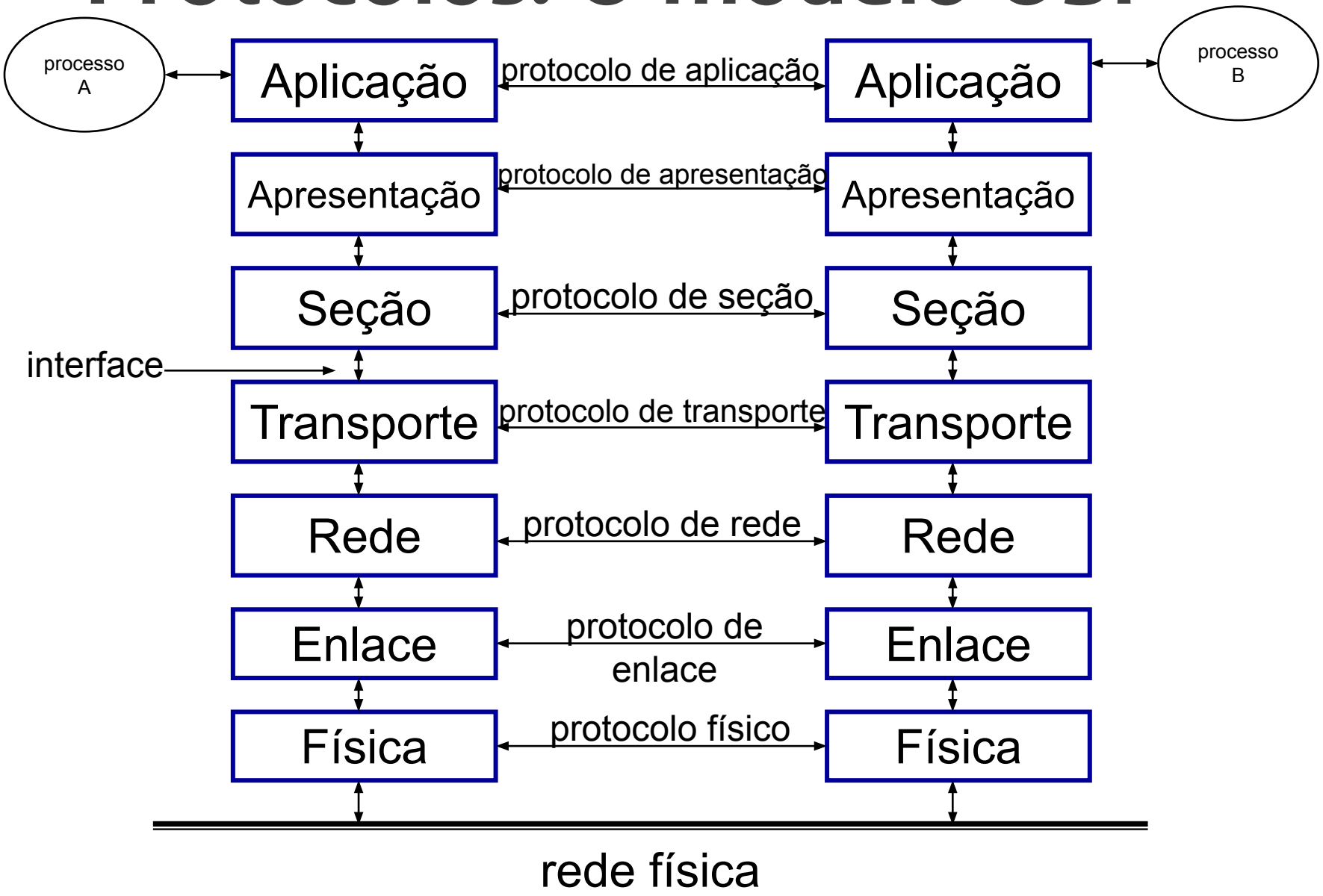
Daemon:

- Um servidor permanentemente ativo. Normalmente possui um endereço padrão conhecido.

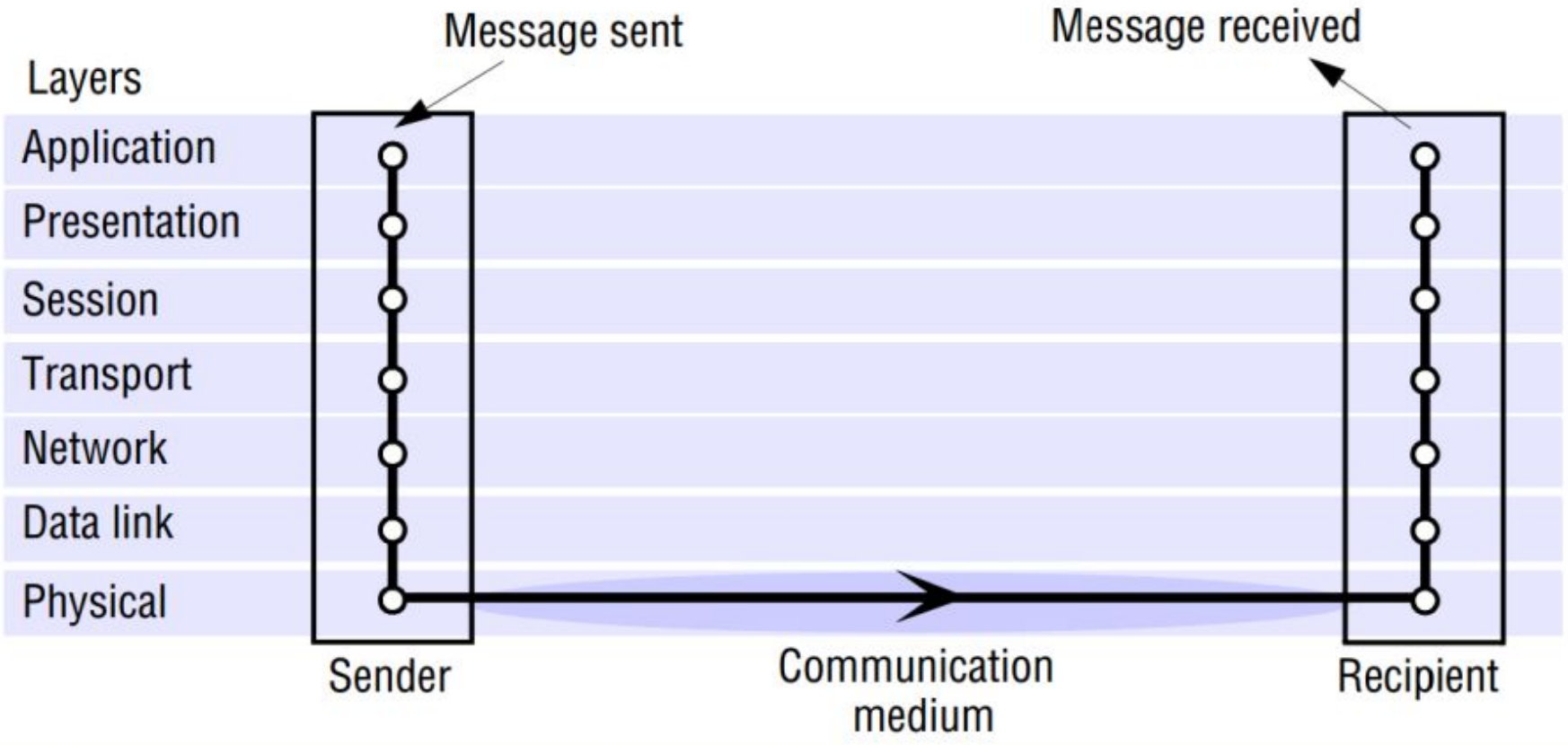
Protocolo:

- Conjunto de regras descrevendo como um cliente e um servidor interagem. Define de forma precisa os comandos aceitos pelo servidor e a forma das mensagens usadas na comunicação.

# Protocolos: O modelo OSI



# Protocolos: O modelo OSI



# Protocolos: O modelo OSI

| # | Camada | Descrição | Protocolos |
|---|---|---|---|
| 7 | Aplicação | Atende aos requisitos de comunicação de aplicativos específicos, definindo uma interface para um serviço | HTTP, SMTP, SNMP, FTP, Telnet, SSH, NFS, DNS |
| 6 | Apresentação | Transmitem dados em uma representação de rede independente das usadas em cada nó. Criptografia, se exigida é feita nesta camada | Segurança TLS, SMB, AFP |
| 5 | Sessão | Realiza operações relacionadas com a confiabilidade das conexões, detecção de falhas e recuperação automática | SIP, SSH, RPC, NetBIOS, ASP |
| 4 | Transporte | Nível mais baixo de manipulação das mensagens que são endereçadas para portas de comunicação | TCP, UDP, SPX |
| 3 | Rede | Transfere pacotes com base no endereçamento dos nodos, o que pode envolver o roteamento entre redes | IP, ICMP, IGMP, X.25, ARP, RARP, BGP, OSPF, RIP, IPX |
| 2 | Enlace de dados | Transmite pacotes entre nodos fisicamente conectados | Ethernet, Token Ring, PPP, HDLC, Frame Relay, ISDN, ATM, Wi-Fi |
| 1 | Física | Transmite sequências de dados binários envolvendo hardware e seus circuitos | Elétrico, radio, laser |

# Modelo TCP/IP ou UDP/IP

IP: Protocolo da camada de rede. Endereça duas máquinas

TCP: Protocolo equivalente ao nível da camada de transporte. Endereça portas. Cada porta em uma máquina corresponde a um serviço (processo). **Protocolo Orientado a conexão**

UDP: Semelhante ao TCP. Protocolo conectionless. **Não é orientado a conexão**. Não garante entrega das mensagens. Provê apenas endereçamento da aplicação via porta

# Endereçamento TCP/IP e UDP/IP

Endereço IP: Deve ser único para cada máquina. IPv4: 32 bits. IPv6: 128 bits

Porta TCP, ou UDP: em UNIX usualmente 16 bits. Portas de 0 a 1023 são reservadas para serviços conhecidos

Sistema operacional possui uma lista dos serviços ativos

Função *getservbyname*() recupera porta do servidor baseada em seu nome

# Comunicação entre Processos

Comunicação em Sistemas:

- Monolíticos: funções, variáveis globais etc
- Distribuídos: troca de mensagens

Troca de Mensagens:

- Conceito primitivo e de muito baixo nível
- Primitivas *send* e *receive*
- Modalidades de Comunicação:
  - Síncrona
  - Assíncrona

# Troca de Mensagens: Marshalling

Marshalling ("empacotamento"): preparação de um conjunto de dados para transmissão em mensagens

Exemplo: nome= "José", conta= 152, saldo= 25.2

```
msg:= new Msg;
msg.addField.asString (nome);
msg.addField.asInteger (conta);
msg.addField.asFloat (saldo);
msg.send (Q);
```

# Troca de Mensagens: Unmarshalling

Unmarshalling (“desempacotamento”): recuperação de um conjunto de dados enviados em mensagens

- msg:= new Msg;
- msg.receive (P);
- nome:= msg.getField.asString ();
- conta:= msg.getField.asInteger ();
- saldo:= msg.getField.asFloat ();

# Marshalling e Unmarshalling

Manualmente: tedioso

Automaticamente: a partir de uma especificação da estrutura de dados.

- Requer:
  - Linguagem para especificação
  - Compilador para esta linguagem

# Marshalling e Unmarshalling

## Implicit typing x Explicit typing

**Implicit typing ou tipagem implícita:** apenas os dados são transmitidos. Requer:

- Linguagem para especificação – ONC RPC XDR e DCE RPC NDR – e compilador para esta linguagem.

**Explicit typing ou tipagem explícita:** O tipo de cada campo é transmitido.

- Exemplo: JSON, XML e Google Protocol Buffers.

XML: texto

```
<ticket><preco moeda='real'>5</preco><vendedor>Shulambs</vendedor></ticket>
```

JSON: texto

```
{ 'moeda':'real' , 'vendedor':'Shulambs', 'preco':'5' }
```

# Marshalling e Unmarshalling

Problema: formatos diversos de representação interna de dados

Solução 1: Conversão para uma "representação neutra" (Exemplo: Sun XDR)

- Problema: conversões desnecessárias entre máquinas de mesma arquitetura

Solução 2: transmissão em formato nativo, porém com identificador da arquitetura

- Problema: transmissão da identificação

# Comunicação Síncrona

Síncrona: *send* e *receive* bloqueantes

- Processo que emitiu *send* permanece bloqueado até que outro processo execute *receive*.
- Processo que emitiu *receive* permanece bloqueado até a chegada de uma mensagem

# Comunicação Síncrona

Vantagem:

- Simplicidade de programação

Desvantagem:

- Desempenho (CPU fica bloqueada durante transmissão e recepção)
- Possível solução: múltiplas *threads*

# Comunicação Assíncrona

Assíncrona: send e receive não bloqueantes

- Processo que emitiu *send* prossegue execução assim que a mensagem é copiada para um *buffer* local
- Processo que emitiu *receive* prossegue execução mesmo que não exista mensagem.
  - *receive* fornece o endereço de um buffer que será preenchido em background.
  - Processo é notificado da chegada de uma mensagem via polling ou via interrupção

# Comunicação Assíncrona

Vantagem:

- Melhor desempenho (processamento prossegue durante transmissão e recepção)

Desvantagem:

- Tratamento do recebimento de mensagens é mais complexo (ocorre fora do fluxo normal de execução)

# Comunicação Síncrona X Assíncrona

| Síncrona | Assíncrona |
|---|---|
| 2 cópias da mensagem | 4 cópias da mensagem |
| Remetente espera envio da mensagem | Remetente envia mensagem pro buffer do kernel |
| Destinatário espera recebimento de mensagem | Destinatário é notificado de chegada de mensagem no buffer do kernel via interrupção ou polling |
| Simples de programar | Complexo: tratamento de várias linhas de execução |
| Sub-utilização dos recursos de máquina | Melhor desempenho |

# Comunicação entre Processos

Abstrações de nível mais alto:

- RPC
- CORBA
- Java RMI
- Microsoft DCOM
- Web Services

Motivação:

- Troca de mensagens é pouco natural
- Abstrações de mais alto nível para "esconder" troca de mensagens

# RPC

RPC: Chamada Remota de Procedimento

- Birrel & Nelson, 1984
- Protocolo de Apresentação (nível 6)

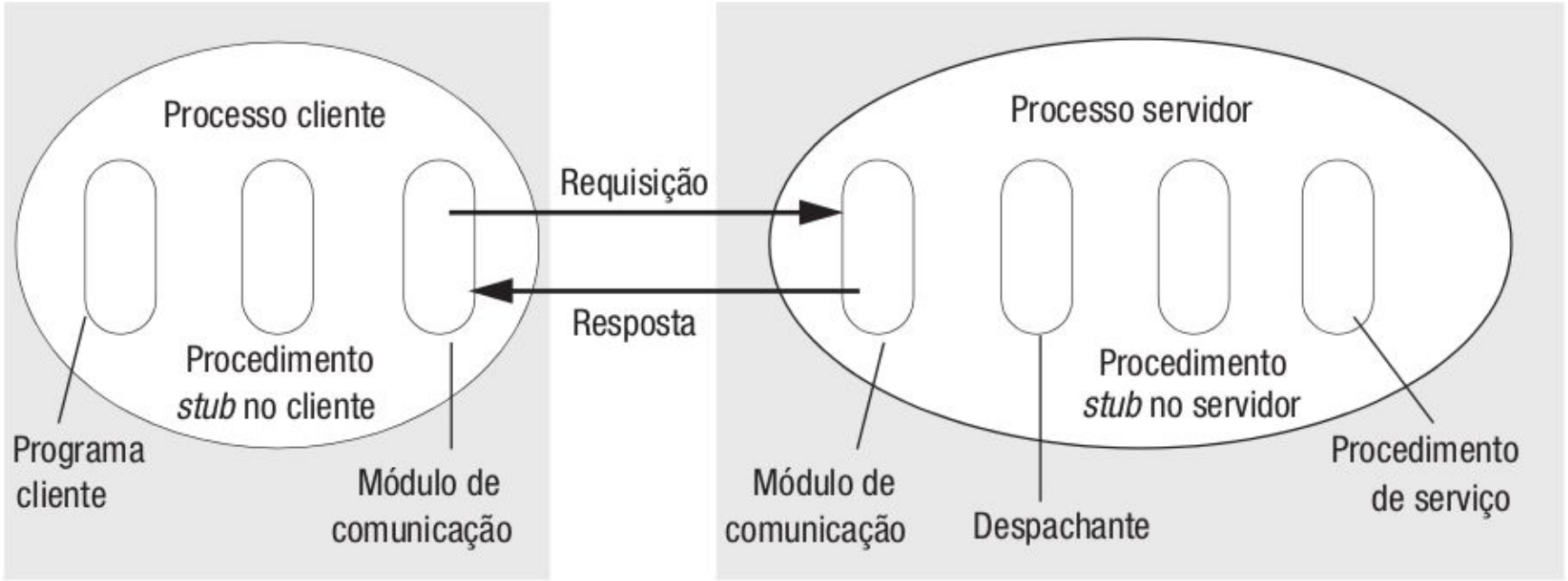
Arquitetura: Utilização de *stubs*

- Se comporta como um procedimento local para o cliente, mas em vez de executar a chamada, empacota o identificador de procedimento e os argumentos em uma mensagem de requisição
- Client Stub
- Server Stub

Automatização do processo de marshalling e unmarshalling

# RPC

RPC permite cliente executar procedimentos nos processos servidores baseados em uma interface de serviço definida.




[1]

[1] Coulouris, Figura 5.10

# RPC

**Módulo de comunicação:** implementa as escolhas de projeto desejadas em relação às retransmissões dos requests, tratamento de duplicatas e retransmissão de resultados.

**Stub do cliente:** se comporta como o procedimento local. Responsável pelo marshalling dos argumentos e unmarshalling do resultado.

**Dispatcher:** seleciona o stub do servidor baseado no identificador do procedimento e encaminha a requisição ao stub do servidor adequado.

**Stub do servidor:** realiza o unmarshalling dos argumentos e o marshalling do resultado, e encaminha a resposta ao cliente.

# Princípio da separação de privilégios

O código remoto será executado por um usuário diferente com privilégios diferentes do requisitante.

Pode aumentar a segurança de um sistema

- Cliente executa com o mínimo de privilégios.
- Exemplo: Banco de dados disponibilizado remotamente via RPC.

# RPC: Seqüência de Passos

Passo 1:

- Chamada local *soma (x, y)*

Passo 2:

- Stub do cliente "captura" chamada e realiza o *marshalling* de seus parâmetros

Passo 3:

- Envio de mensagem com os parâmetros

Passo 4:

- Recebimento da mensagem na máquina remota e chamada ao stub do servidor

# RPC: Seqüência de Passos

Passo 5

- Stub do servidor realiza o *unmarshalling* dos parâmetros e chama o procedimento remoto

Passo 6:

- Execução do procedimento remoto

Passo 7:

- Stub do servidor realiza o marshalling do resultado

# RPC: Seqüência de Passos

Passo 8:

- Envio da mensagem com o  resultado

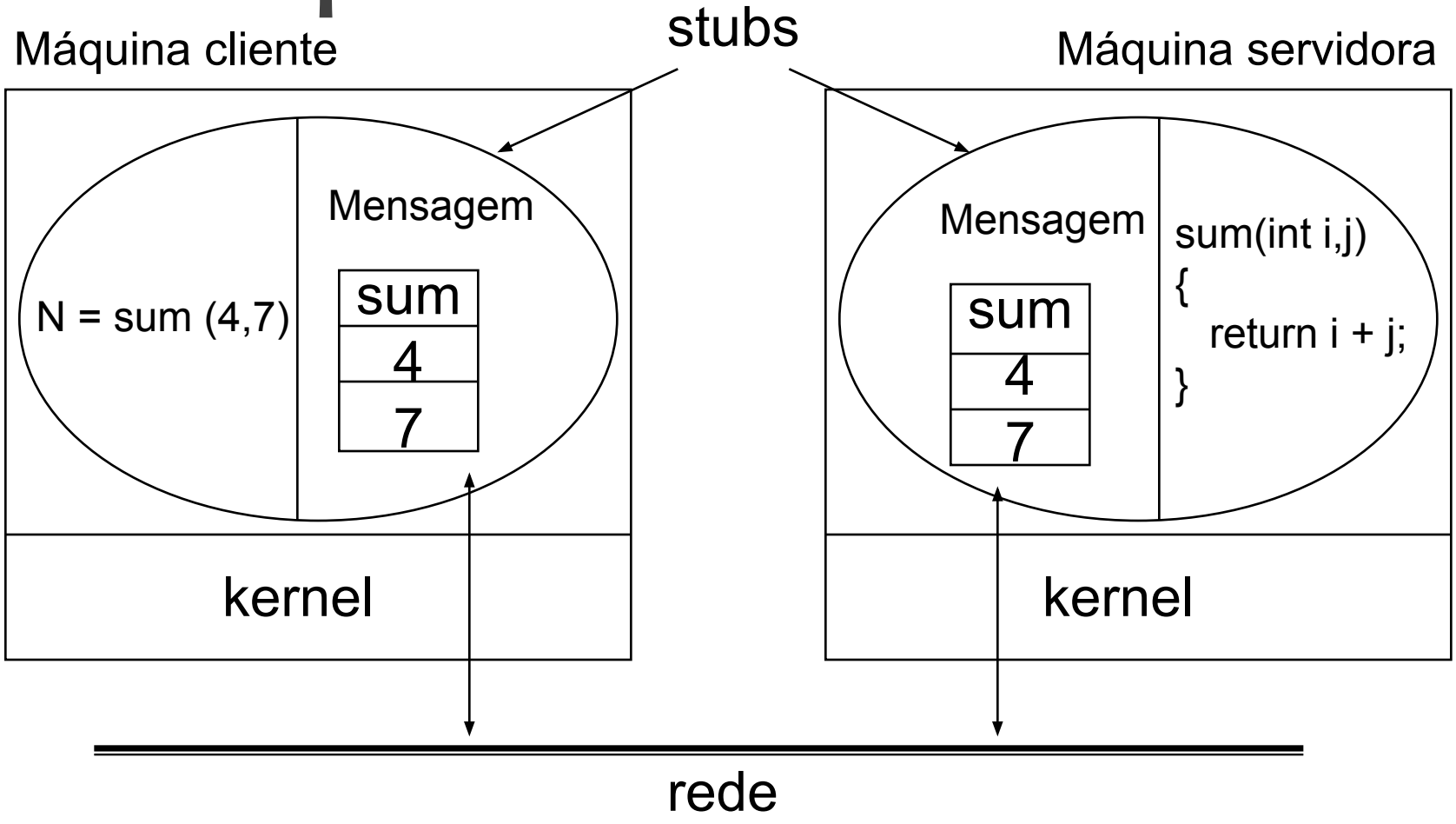
Passo 9:

- Recebimento do resultado e chamada ao stub do cliente

Passo 10:

- Stub do cliente realiza o unmarshalling do resultado e retorna o mesmo ao cliente

# RPC: Seqüência de Passos



# Exemplo: Sun RPC

Três componentes principais:

- Linguagem XDR (External Data Representation)
  - Linguagem para definição do cabeçalho dos procedimentos remotos
- Compilador rpc, chamado rpcgen
  - Dada uma especificação em XDR, gera stubs do cliente e do servidor
- Run-time (biblioteca)

# Exemplo de Aplicação

Suponha que se deseja implementar um servidor com dois procedimentos remotos:

- bin_date: retorna data atual em binário
- str_date: retorna data atual em forma de string

# 1º Passo: Especificação XDR

```
// arquivo date.x
program DATE_PROG {                  // nome do programa
  version DATE_VERS {
    long bin_date (void) = 1;        // procedimento num. 1
    string str_date (long) = 2;      // procedimento num. 2
  } = 1;                             // número da versão
} = 0x312434567;                     // número do programa
```

# 2º Passo: Geração dos stubs

Chamada: rpcgen date.x

Serão gerados os seguintes arquivos:

- date_svc.c: stub do servidor
- date_clnt.c: stub do cliente
- date.h: arquivo de header

# 2º Passo: Geração dos stubs

- Arquivo date.h:

```
#define DATE_PROG 0x312434567;
#define DATE_VERS 1
#define BIN_DATE 1
#define STR_DATE 2
long *bin_date_1  (void *,   CLIENT *);
char **str_date_1 (long *, CLIENT *);
```

# 3º Passo: Procedimentos Remotos

Procedimentos disponibilizados pelo servidor:

```
// arquivo dateproc.c
// (chamado pelo server stub)
long *bin_date_1 (long *date) {
   ..... time (); ......
}
char **str_date_1 (long *bindate) {
   ..... ctime (bindate); ........
}
```

# 4º Passo: Cliente

```
// arquivo rdate.c
#include <rpc.h>
#include <date.h> // gerado pelo rpcgen
void main (int argc, char **argv[]) {
    CLIENT *cl;
    char *server= argv[1];
    long * lresult;
    char ** sresult;
```

# 4º Passo: Cliente

```
cl= clnt_create(server, DATE_PROG, DATE_VERS, “udp”);

........

lresult= bin_date_1 (NULL, cl);        // chamada remota
printf (“Data no servidor %s = %d\n”, server, *lresult);

........
sresult= str_date_1 (lresult, cl);     // chamada remota
printf (“Data no servidor %s = %s\n”, server, *sresult);

........

clnt_destroy (cl);
}
```

# 5º Passo: Compilação

Compilação do Cliente:

- cc -o rdate.c date_clnt.c -l rpclib

Compilação do Servidor:

- cc -o date_proc.c date_svc.c -l rpclib

# 6º Passo: Execução

Servidor:

$ date_svc &

Cliente:

$ rdate localhost

Data no servidor localhost: 609264219

Data no servidor localhost: Thu 17 18:10:10 2000

# Ambiente de Execução

Como o cliente determina a porta em que o servidor foi instalado ?

Ativação do Servidor:

- Servidor requisita uma porta UDP
- Servidor registra as seguintes informações junto a um processo especial chamado portmapper:

( prog_number, version_number, protocol, port_number )

# Ambiente de Execução

Ativação do cliente (função clnt_create)

- Realiza consulta ao portmapper da máquina servidora:
  - Parâmetros: prog_number, version_number, protocol
  - Resultado: port_number
- Stub do cliente envia mensagem para a porta obtida acima.
- Mensagem é recebida pelo stub do servidor.

# Rotinas de alto nível

clnt_create()

- Cria um cliente genérico. O programa avisa à função clnt_create () onde o servidor está localizado e qual o tipo de transporte (TCP/UDP) a ser usado.

svc_create()

- Cria uma referência ao servidor para todos os transportes (TCP/UDP). Possui as variações svctcp_create e svcudp_create. O programa avisa à função qual despachante usar.

# RPC x chamadas locais

RPC são de muito mais lentas – 10x, 100x ou mais.

Marshalling e unmarshalling realizam transformação de dados.

- consome memória e processamento.

Stubs devem lidar com falhas na rede e problemas de versionamento.

Necessário implementar mecanismos de autenticação.

# Segurança

RPC possui três mecanismos de autenticação:

- Null authentication (nenhum): não possui checagem de segurança.
- Estilo UNIX: transmite o UNIX user ID e group ID do cliente.
- Baseado em chave compartilhada para assinar mensagens RPC (por exemplo: DES – Data Encryption Standard).

Possui ainda mecanismos de compatibilidade com o Kerberos – protocolo de autenticação de rede para comunicações individuais seguras e identificadas.

# Análise de RPC em Relação ao Critério de Transparência

Até que ponto uma chamada remota é equivalente a uma chamada local?

- Passagem de Parâmetros
- Binding
- Semântica das Chamadas

# Passagem de Parâmetros

Passagem por valor:

- Implementação direta com semântica de cópia

Passagem por Referência:

- Implementação bem mais difícil
- Problema: endereço da máquina cliente
- Possíveis soluções:
  - Utilizar semântica de valor/resultado (enviar a cópia do valor e atualizar no cliente com base no resultado)
  - Memória compartilhada
  - Não permitir passagem por referência

# Passagem de Parâmetros

Sun RPC

- Passagem apenas por valor
- Um único parâmetro
  - Mais de um parâmetro: devem ser empacotados em uma struct

# Binding

Como o cliente localiza o servidor ?

- Solução 1: indicar o nome do servidor no próprio código
  - Usado no Sun RPC
  - Problema: falta de flexibilidade
- Solução 2: Usar um serviço de diretório
  - Registrar neste serviço (prog_number, version_number, protocol, port_number)
  - Cliente consulta este servidor
  - Usado no DCE RPC

# Semântica das Chamadas

Normalmente, o stub do cliente retransmite uma mensagem após um certo time-out

Como então determinar quantas vezes o procedimento remoto é executado ?

- Uma vez: quando tudo funciona corretamente
- Zero vezes: quando a mensagem do cliente não chega ao servidor, apesar das várias retransmissões
- Mais de uma vez: quando a resposta do servidor se perde

# Semântica de Chamadas

Tipos de semânticas em chamadas RPC:

- Exactly Once (“exatamente uma vez”):
  - Desejada, mas difícil de conseguir
- At Least Once (“pelo menos uma vez”):
  - Basta que cliente tente até conseguir
  - RP pode ser executado mais de uma vez
- At Most Once (“no máximo uma vez”)
  - RP pode ser executado 0 ou uma vez

## Implementação de Chamadas "At Most Once"

- Cada chamada RPC possui um identificador
- Servidor armazena em um cache os identificadores das chamadas que já foram processadas e os resultados das mesmas
- Em caso de duplicação, RPC não é executado de novo e envia-se o resultado armazenado no cache
- Usado no Sun RPC

# Tipos de Operações

Operações Idempotentes:

- Podem ser executadas qualquer número de vezes, pois não tem efeito colateral
- Exemplo: consultas em geral (hora, saldo etc)
- Semântica “at least once”

Operações não Idempotentes:

- Possuem efeito colateral.
- Exemplo: atualizações (transferência bancária, gravação de um registro etc)
- Semântica “exactly once”