Sistemas Distribuídos

Aula 3 – Processos e Threads, Comunicação entre processos

DCC/IM/UFRRJ
Marcel William Rocha da Silva

Objetivos da aula

- **Aula anterior**
  - Arquiteturas de sistemas
  - Arquiteturas vs. *middleware*

- **Aula de hoje**
  - Processos e Threads
    - Clientes e servidores multithread
    - Projeto de processos servidor
  - Comunicação entre processos
    - *Middleware* de comunicação
      - Chamada de procedimento remoto (RPC)

Conteúdo Programático

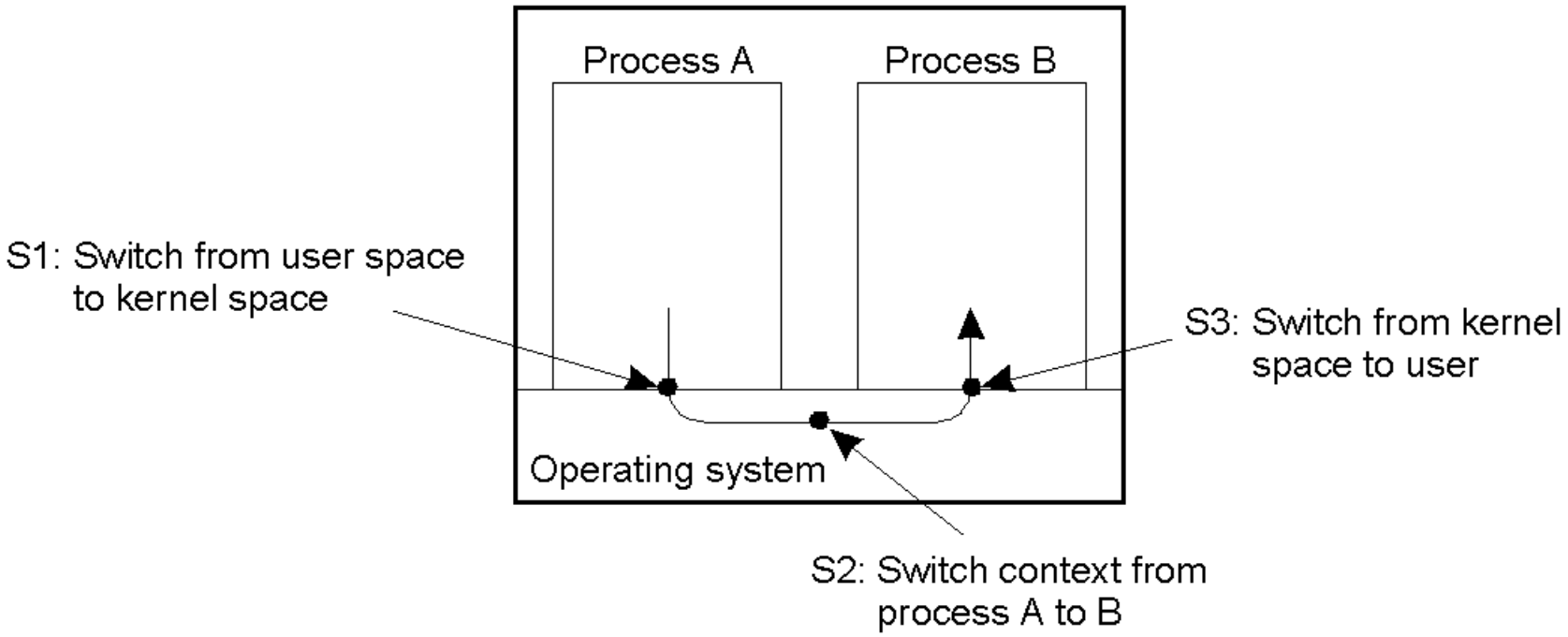
- Introdução e visão geral
- Princípios de sistemas distribuídos
  - Arquiteturas
  - Processos
  - Comunicação
  - Nomeação
  - Sincronização
  - Consistência e replicação
  - Tolerância à falha
  - Segurança

Processos

- Definição
  - Programa em execução
- SO garante o “isolamento” entre processos
  - Contexto de processo e chaveamento de contexto
    - Contexto de *hardware* → registradores (PC, SP, etc)
    - Contexto de *software* → informações sobre processo (UID, PID, prioridade, etc)
    - Espaço de endereçamento → páginas da memória
- Custo do chaveamento de contexto entre processos pode ser alto

Processos

- Chaveamento de contexto

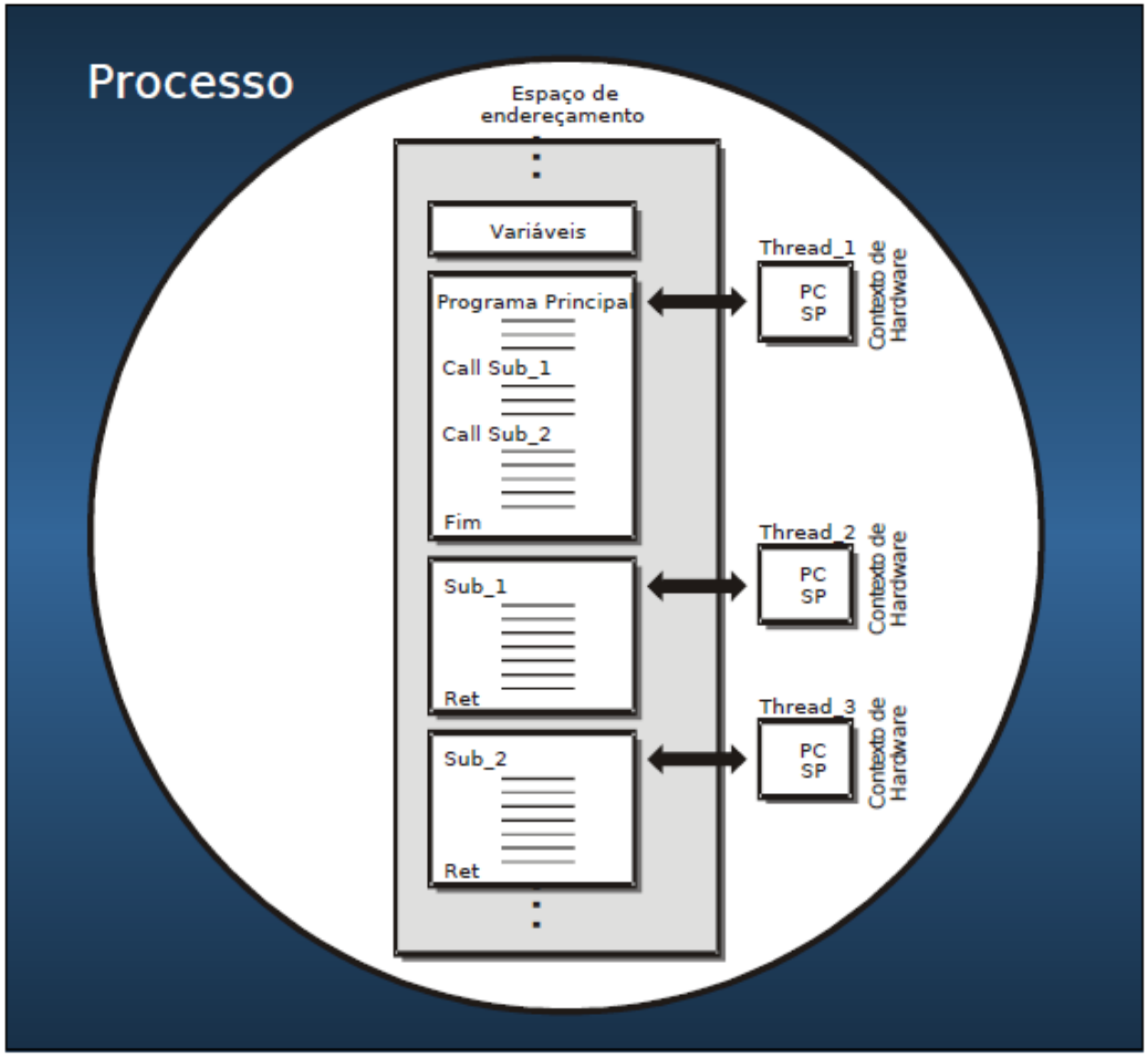


Threads

- Semelhante a noção de processo
  - Cada thread com um trecho de código próprio
  - Executadas de forma independente e concorrente

- Entretanto, com menos informação de controle
  - Contexto de *hardware* próprio
    - PC, SP, registradores...
  - Algumas informações adicionais para permitir o gerenciamento da execução concorrente
    - Contexto de *software*
  - Espaço de endereçamento compartilhado entre threads de um mesmo processo → Baixo custa na mudança de contexto entre threads!

Threads



Cuidados com o isolamento entre as threads fica por conta do desenvolvedor da aplicação

Uso de threads

- Em **sistemas não distribuídos**
  - Evita espera do processo em chamadas ao sistema bloqueadoras
    - Ex.: Leitura/escrita em disco, comunicação via rede
  - Permite explorar o paralelismo em máquinas multiprocessadas
    - Uma thread executada em cada CPU
  - Simplifica a comunicação entre processos (IPC)
    - IPC convencional implica em chaveamento de contexto intenso → grande *overhead*
    - Uso de threads permite a comunicação através de memória compartilhada → mais veloz

Uso de threads

- Em **sistemas distribuídos** o uso de threads facilita a comunicação
  - Comunicação é uma tarefa bloqueadora
    - Envio de requisição, e espera pela resposta
    - Problema em cenários de grandes atrasos
  - Manutenção de múltiplas conexões lógicas simultâneas
    - Especialmente útil em cenários multiprocessados
- Exemplo: Aplicação peer-to-peer
  - Tarefas cliente e servidor simultâneas

Clientes multithreads

- Permite ocultar a latência de comunicações de longa distância
  - Realizar trabalho útil durante o bloqueio
  - Explorar paralelismo na comunicação

- Exemplo
  - Navegador Web

Clientes multithreads

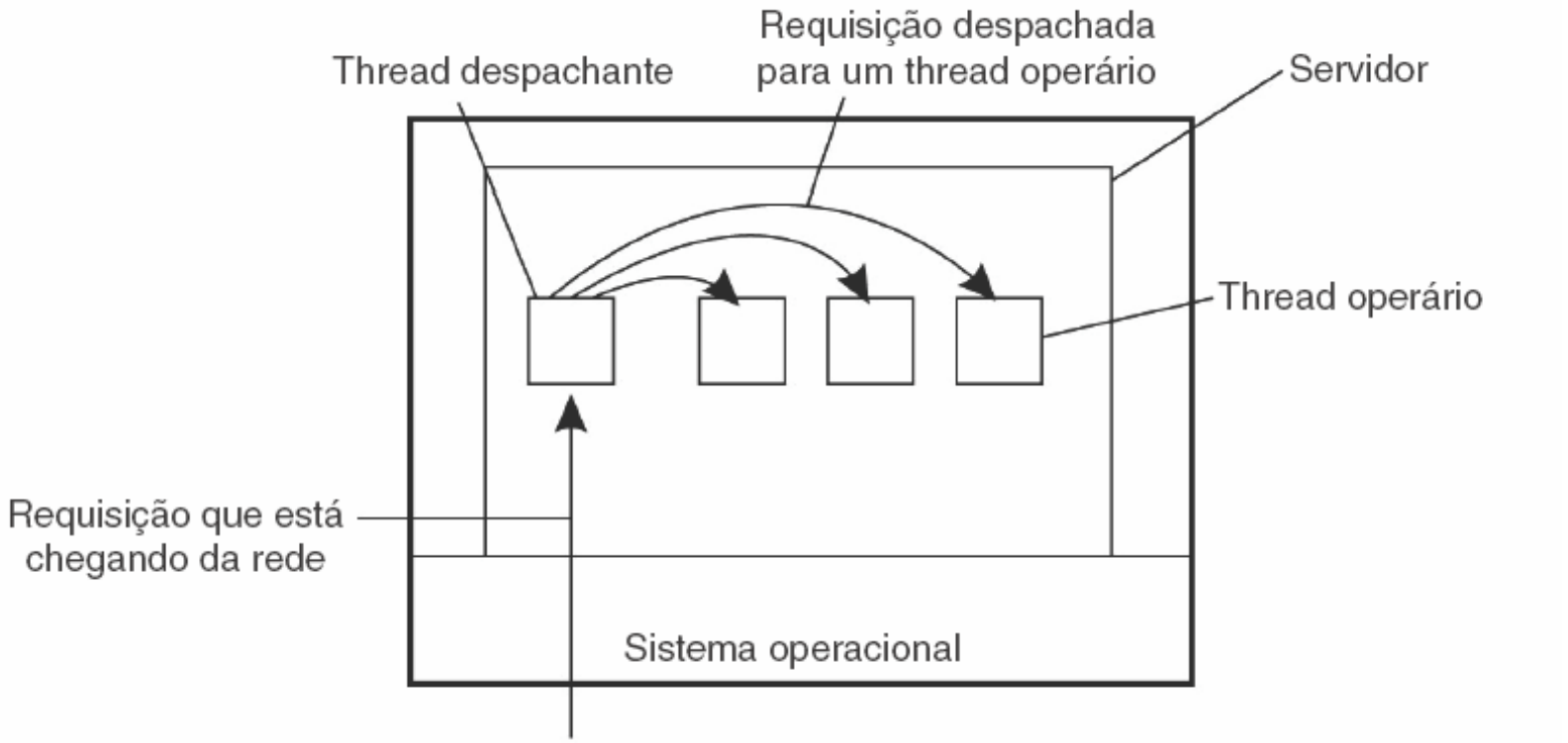
- Navegador Web
  - Documentos Web possuem diversos objetos
    - Texto, imagens, áudio, vídeo, etc
  - Cada objeto pode ser buscado com uma conexão TCP
    - Conexão e leitura dos dados é uma tarefa bloqueadora
  - Com objetos no mesmo servidor...
    - Conexão persistente com paralelismo resolve → mas servidor deve dar suporte!

Clientes multithreads

- Navegador Web
  - Com objetos em servidores diferentes, ou sem HTTP persistente e paralelo...
    - Uso de threads permite obter objetos em paralelo
    - Thread para a visualização e threads para obter os objetos
      - Melhor experiência para o usuário
      - Realizar trabalho útil durante o bloqueio

Servidores multithread

- Threads permitem melhor desempenho
  - Atendimento a múltiplos clientes simultâneos
- Modelo de servidor multithread despachante/operário



Projeto de servidores

- Questões para o projeto:
  - Iterativo ou concorrente?
  - Como receber as requisições?
  - Como encerrar um serviço?
  - Manter ou não estado dos clientes?

Projeto de servidores

- Iterativo ou Concorrente?
  - **Iterativo**: O servidor é um processo único
    - Recebe a requisição do cliente e trata imediatamente
  - **Concorrente**: O "processo servidor" não manipula diretamente a requisição
    - Requisição repassada para uma thread separada ou um outro processo
    - **`fork()`** também é muito utilizado em servidores Web

Projeto de servidores

- Como receber as requisições?
  - Requisições são enviadas ao processo servidor (*daemon*) através de uma porta
    - Alguns serviços possui portas padrão
      - Ex.: HTTP 80, FTP 21, SSH 22
  - Em alguns casos, pode-se implementar um processo **supervisor**, que "escuta" nas portas conhecidas e redireciona a requisição para a porta do serviço
    - No Unix → *inetd*
    - Evitar ter vários serviços carregados em espera → será executado apenas quando chegam requisições

Projeto de servidores

- Como encerrar um serviço?
  - Considere uma transferência de arquivo → usuário decide cancelar a transferência
  - Possibilidades?
    - Cliente encerra aplicação (mais comum na Internet!)
      - Servidor encerra, após um tempo, a conexão
    - Transmissão de dados de controle “fora da banda”
      - Servidor possui uma conexão de controle específica
      - Servidor trata, em uma mesma conexão, dados urgentes

Projeto de servidores

- Manter ou não estado dos clientes?
  - **Servidor sem estado** → não mantém informações sobre os estados de seus clientes
    - Pode mudar o seu estado sem ter que informar a nenhum cliente
    - Ex.: Servidor Web, servidor de arquivos
    - Em alguns casos pode guardar estado temporário
      - Mas não é crítico para o seu funcionamento
      - Estado flexível (***soft state***)
      - Ex.: Login em página Web que expira

Projeto de servidores

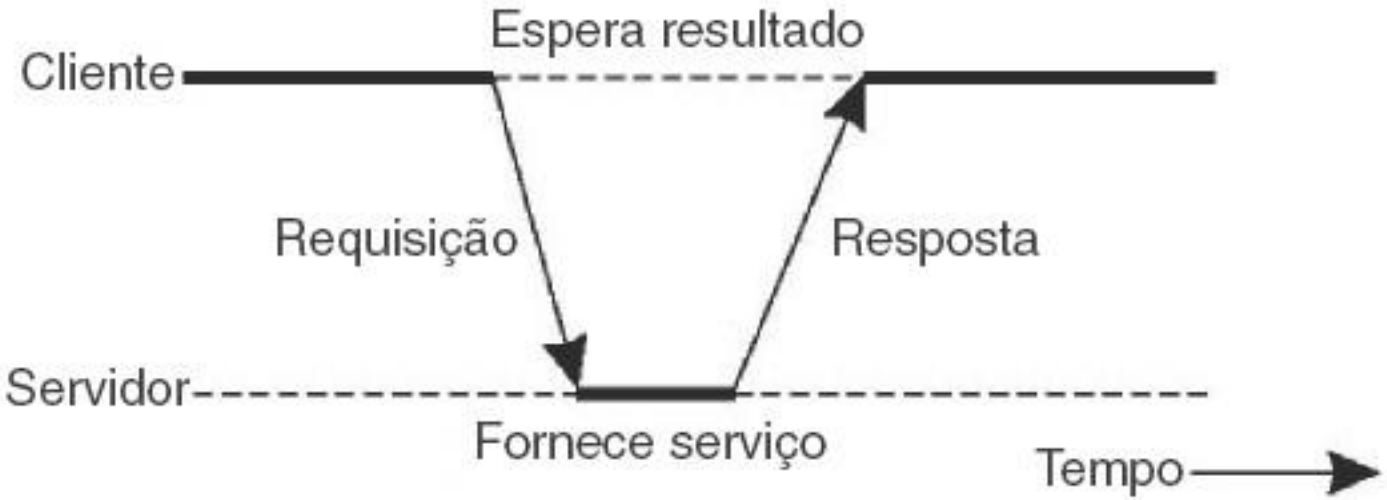
- Manter ou não estado dos clientes?
  - **Servidor com estado** → mantém informações persistentes sobre seus clientes
    - Ex.: Pasta compartilhada no Dropbox
    - Necessário que o servidor saiba os usuários que podem acessar e modificar os dados compartilhados
    - Qualquer mudança feita por um usuário deve ser replicada para os demais

Comunicação entre processos

- Coração de um sistema distribuído!
- Como promover a troca de informações entre processos em diferentes máquinas?
  - Não é simples!
  - Diferentes maneiras!
- Desejável que o modelo de comunicação seja transparente para o desenvolvedor

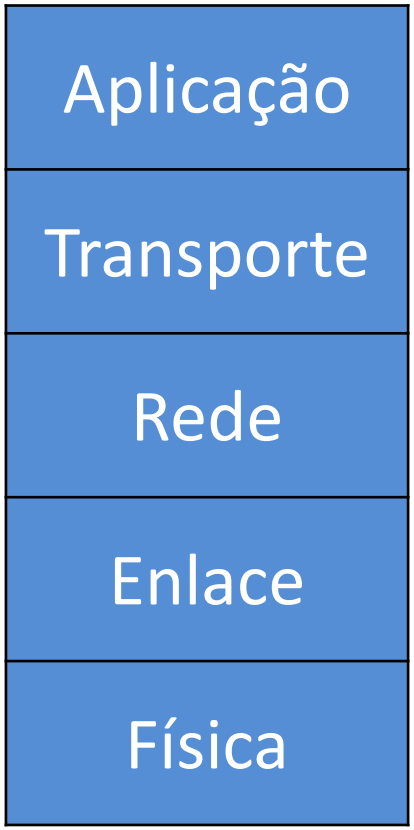
Modelo cliente-servidor

- Partes envolvidas:
  - **Servidor**: implementa um serviço
  - **Cliente**: solicita ao servidor um serviço e espera por uma resposta
- Comportamento requisição-resposta:



Protocolos em camadas

- **Aplicação**: programas dos usuários
  - Web, email, torrent, ...
- **Transporte**: transferência de dados fim-a-fim
  - TCP e UDP
- **Rede**: roteamento de pacotes da origem ao destino
  - IP, protocolos de roteamento
- **Enlace**: transferência de dados entre elementos vizinhos
  - PPP, Ethernet, 802.11, ...
- **Física**: transferir bits pelo meio físico



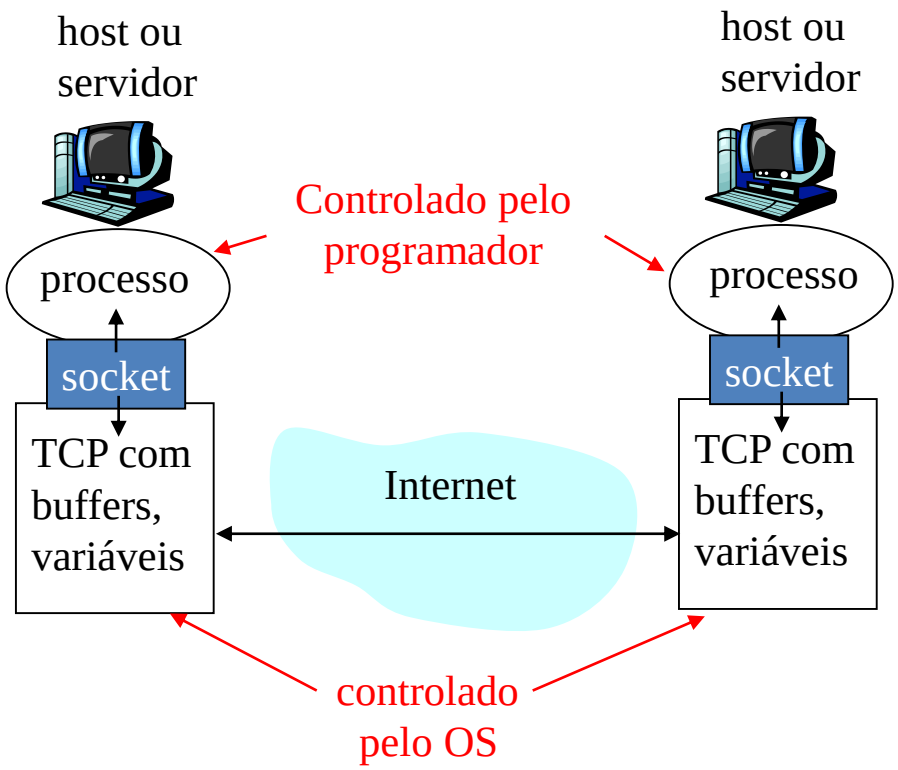
Pilha de protocolos TCP/IP

Protocolos de transporte

- **TCP**
  - Orientado a conexão
  - Confiável, porém pode ser “lento”
- **UDP**
  - Sem conexão
  - Rápido, porém não-confiável
- Escolha depende do tipo de aplicação!

Sockets

- Interface usada pelos processos para o envio de mensagens
  - Interface entre as camadas de aplicação e transporte
  - Analogia da porta da casa
    - Processo “passa” (envia) mensagens pela porta (socket)
    - Mensagens entram (são recebidas) pela porta
- Sem transparência de distribuição
  - Comunicação explicita através de métodos **send** e **receive**
- Solução:
  - **Middlewares de comunicação!**



Camada de middleware

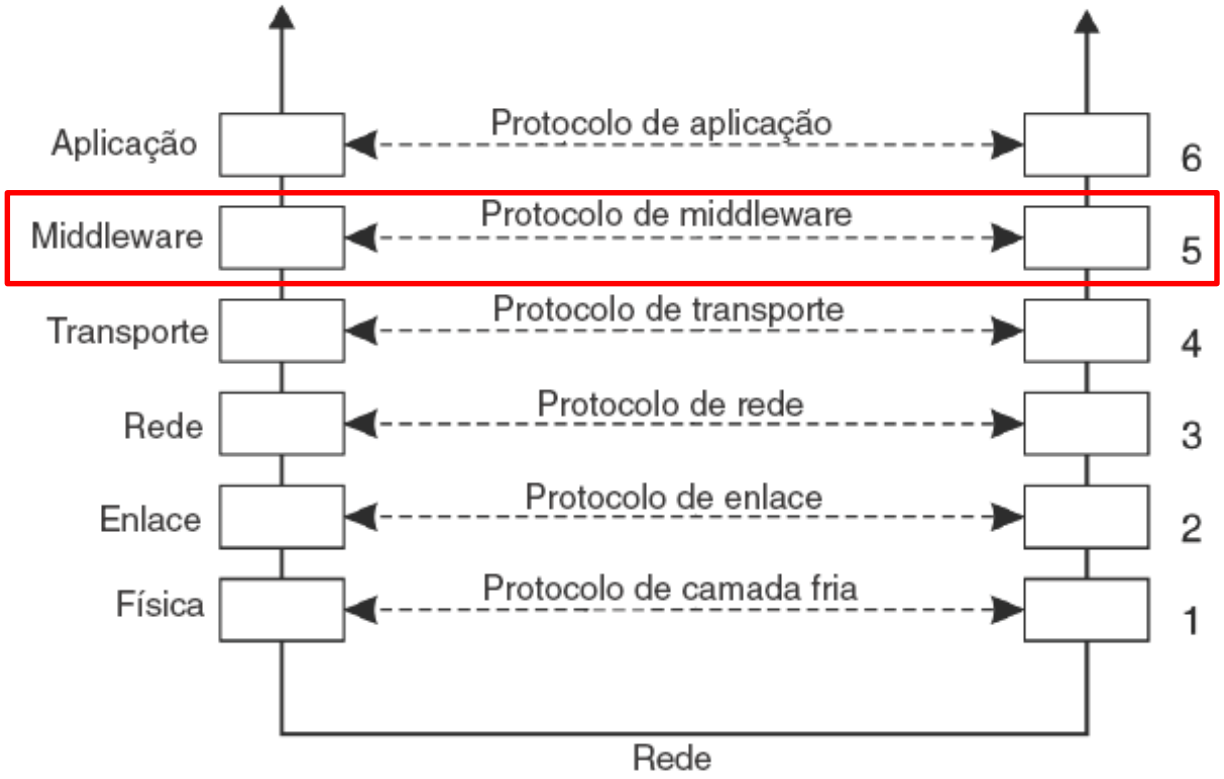


Figura 4.3 Modelo de referência adaptado para comunicação em rede.

Camada de middleware

- Camada de software situada entre a camada de nível mais alto (aplicações e usuários) e as camadas de nível mais baixo (SO, detalhes de comunicação e hardware)

- **Serviços da camada de middleware**
  - Autenticação
  - Comprometimento
  - Transparência
  - Comunicação (serviços de alto nível)
  - Outros...

Classificação da comunicação

- Quanto a **persistência**:
  - Persistente ou transiente
- Quanto a **sincronização**:
  - Síncrona ou assíncrona
- Quanto a **granularidade**:
  - Discreta ou fluxo

Tipos de middleware de comunicação

- **Chamada de procedimento remoto**
  - *Remote Procedure Call* (RPC)
  - *Remote Method Invocation* (RMI)
- **Comunicação orientada a mensagens**
  - *Publish/subscribe*
- **Comunicação orientada a fluxo**
  - Videoconferência

Chamada de procedimento remoto

- RPC – *Remote Procedure Call*
- Permite a processos chamar procedimentos localizados em outras máquinas
- Oculta detalhes da comunicação via rede do desenvolvedor
  - Nada de sockets!

Chamada de procedimento remoto

- Mas não é tão simples... problemas
  - Arquiteturas de máquinas diferentes
  - Espaços de endereçamento distintos
  - Passagem de parâmetros

- Principal objetivo é a transparência
  - **Chamada de procedimento remoto** deve parecer uma **chamada de procedimento local**

Operação básica de RPC

- Similar ao modelo de chamadas **locais**
  - Rotina que invoca os procedimentos coloca os argumentos em uma área da memória conhecida e transfere o controle para o procedimento, que lê os argumento e processa
  - Resultado armazenado na área conhecida da memória
  - Controle passado de volta para a rotina, que obtém o resultado da memória conhecida

Operação básica de RPC

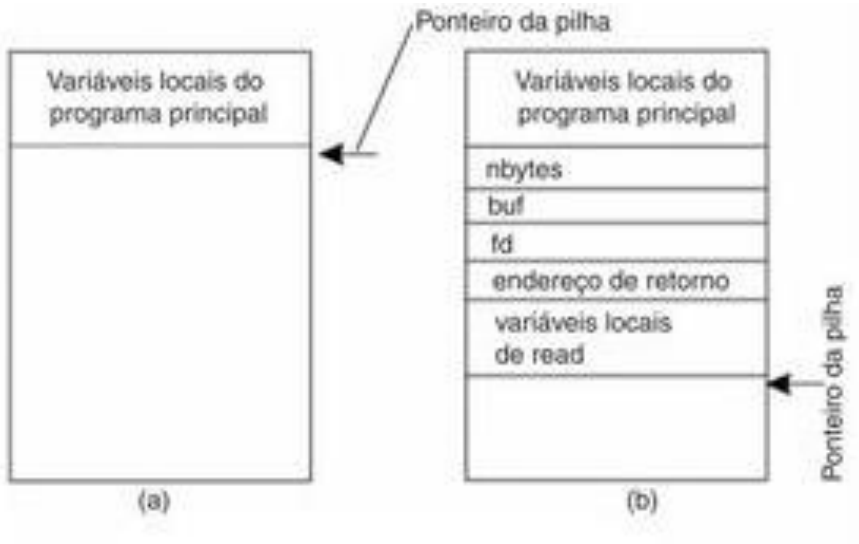
- Em RPC
  - Processo invocador primeiro manda uma mensagem para o processo servidor e aguarda (bloqueia) uma mensagem de resposta
  - A mensagem de invocação contém os parâmetros do procedimento e a mensagem de resposta contém o resultado da execução do procedimento
  - Processo servidor permanece em espera até a chegada de uma mensagem de invocação. Quando uma mensagem de invocação é recebida, o servidor extrai os parâmetros, processa-os e produz os resultados, que são enviados na mensagem de resposta

Operação básica de RPC

- Exemplo:

```
count = read (fd, buf, nbytes)
```

- **`fd`** → descritor de um arquivo
- **`buf`** → vetor para o qual os caracteres serão lidos
- **`nbytes`** → quantos bytes serão lidos



Operação básica de RPC

- Chamada de procedimento convencional
  - Em C, parâmetros podem ser **chamados por valor** ou **chamados por referência**

- Em RPCs
  - A diferença entre chamadas por valor e por referência é importante
  - Outro mecanismo de passagem de parâmetros é o **copiar/restaurar**:
    - "chamador" copia a variável para a pilha e ao copiá-la de volta, sobrescreve o valor original → semelhante a chamada por referência

Operação básica de RPC

- Consideremos a chamada da função read
  - Em um sistema tradicional, a função read é um tipo de interface entre o código de usuário e o sistema operacional local

- Usando RPC é possível conseguir transparência na execução da função read
  - Por ser um procedimento remoto, uma versão diferente da função read, denominada apêndice de cliente é colocada na biblioteca
  - Ela não realiza a tarefa no sistema local, mas empacota os parâmetros em uma mensagem e envia ao servidor

Operação básica de RPC

- Usando RPC, é possível conseguir transparência na execução da função read
  - Apêndice de cliente bloqueia até que a resposta volte
  - Quando a mensagem chega ao servidor, o SO do servidor a passa para um apêndice do servidor
  - O apêndice de servidor desempacota os parâmetros da mensagem e chama o procedimento do servidor da maneira usual

Operação básica de RPC

- Usando RPC, é possível conseguir transparência na execução da função read
  - Do ponto de vista do servidor, é como se ele fosse chamado diretamente pelo cliente
  - No caso de read o servidor colocará os dados no buffer, interno ao apêndice de servidor
  - Ao final, empacota o buffer em uma mensagem e retorna o resultado ao cliente
  - Ao chegar no cliente, o SO reconhece que a msg está direcionada ao processo cliente
  - Msg é copiada para o buffer e o processo cliente é desbloqueado

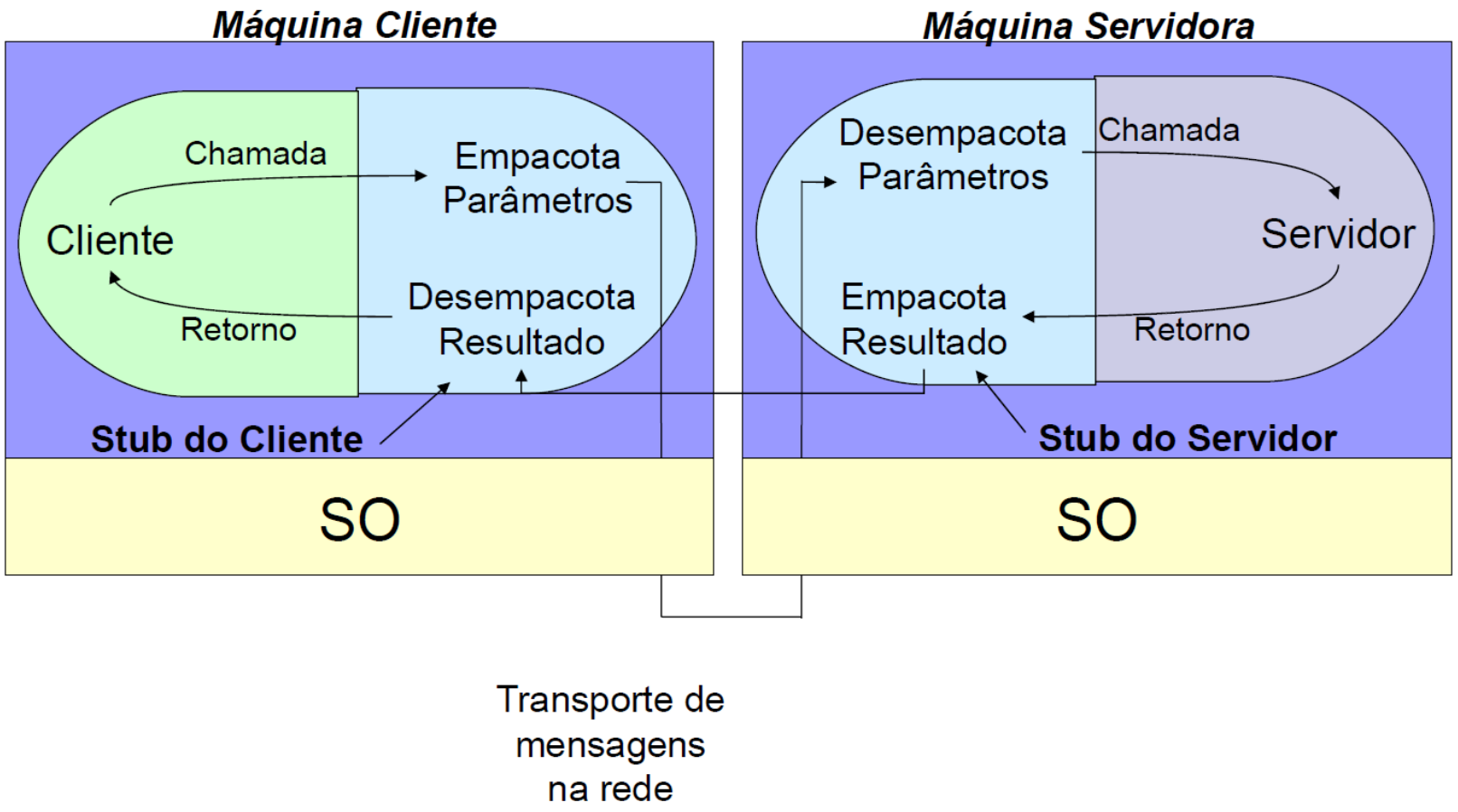
Operação básica de RPC

Figura 4.6 *Princípio de RPC entre um programa cliente e um programa servidor.*

Operação básica de RPC

- Uso de **stubs** (apêndices)
  - **Stub do cliente** → responsável por empacotar os parâmetros em uma msg e enviar a msg para a máquina do servidor. Quando resposta chega, resultado é copiado para cliente, e controle volta a ele
  - **Stub do servidor** → responsável por desempacotar parâmetros, chamar o procedimento real do servidor e retornar resposta para máquina do cliente

Operação básica de RPC



Passagem de parâmetros

- Operação parece simples, mas tem alguns complicadores...

- Passagem de parâmetros é um problema
  - Arquiteturas diferentes?
  - Como passar um ponteiro?

Passagem de parâmetros por valor

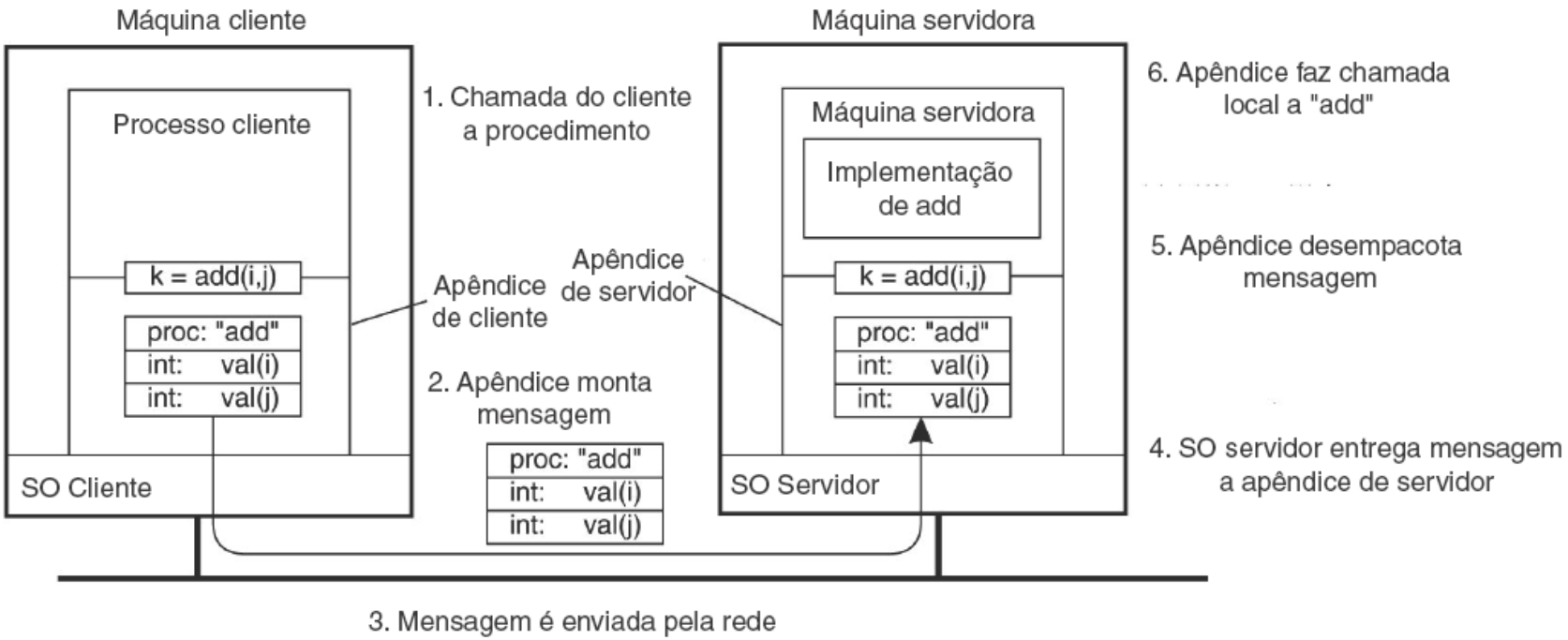


Figura 4.7 Etapas envolvidas para fazer um cálculo remoto por meio de RPC.

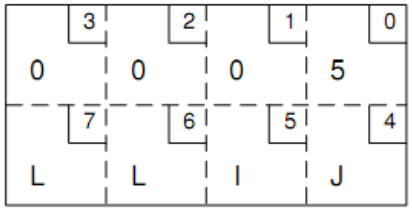
Passagem de parâmetros por valor

- Exemplo anterior funciona se sistemas cliente e servidor forem idênticos

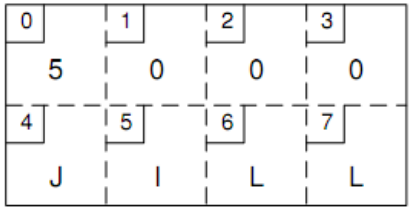
- Problemas:
  - Diferentes codificações para caracteres
    - Codificação: ASCII, UTF, ISSO-8859, etc
  - Ordenação dos bytes
    - *Little endian* ou *big endian*

Passagem de parâmetros por valor

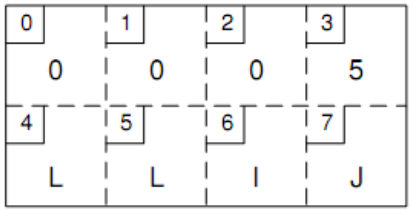
Mensagem de 32 bits
2 parâmetros:
um inteiro (5)
uma string (JILL)



Mensagem original na arquitetura Intel
(numera os bits de um inteiro
da direita para a esquerda –
Little endian)



Mensagem recebida na SPARC
(na ordem inversa – Big endian)



Mensagem invertida

Solução: Cliente diz seu tipo. Conversão feita pelo servidor se tipos forem diferentes

Passagem de parâmetros por referência

- Como são passados ponteiros ou, em geral, referências?
  - Só fazem sentido dentro do espaço de endereçamento nativo
- Consideremos a função read, stub do cliente "sabe" que o segundo parâmetro aponta para um conjunto de caracteres
  - Suponha que o cliente saiba o tamanho do vetor...

Passagem de parâmetros por referência

- Solução → **Copiar/restaurar**:
  - Copiar o vetor para a mensagem e enviar ao servidor
  - Stub do servidor, chama o servidor com um ponteiro para este vetor
  - Modificação feita pelo servidor é armazenada diretamente no vetor que está no stub
  - Ao enviar o vetor de volta ao stub do cliente, o vetor é copiado de volta ao cliente