Sistemas Distribuídos

Comunicação

Edeyson Andrade Gomes
www.edeyson.com.br

Roteiro da Aula

Roteiro da Aula

- Comunicação entre Processos
- Protocolos
- *Modelo OSI*
- Modelo Cliente Servidor

Comunicação entre Processos

Comunicação entre Processos

- Sistema Monoprocessado
  - Memória compartilhada
    - Exclusão mútua, regiões críticas, etc.
    - Ex.: Produtor e Consumidor
    - Semáforos
      - Variáveis compartilhadas.

Comunicação entre Processos

- Sistemas Distribuídos
  - Multi-Computadores
    - Não há memória compartilhada
    - Troca de Mensagens
    - Protocolos de comunicação
      - Regras para a correta troca de mensagens
      - Interpretação das mensagens

Protocolos

Protocolos

- Regras para comunicação
- Camadas
  - Responsabilidades específicas na troca de mensagens

- **Modelo OSI**
  - Comunicação entre sistemas abertos

Protocolos

- **Modelo OSI**
  - Um sistema aberto é preparado para se comunicar com qualquer outro sistema aberto através de regras que ditam o formato, conteúdo e significado das mensagens enviadas e recebidas.
    - Protocolo

Protocolos

- Modelos gerais de Protocolos
  - Orientado a Conexão
  - Sem Conexão

Protocolos

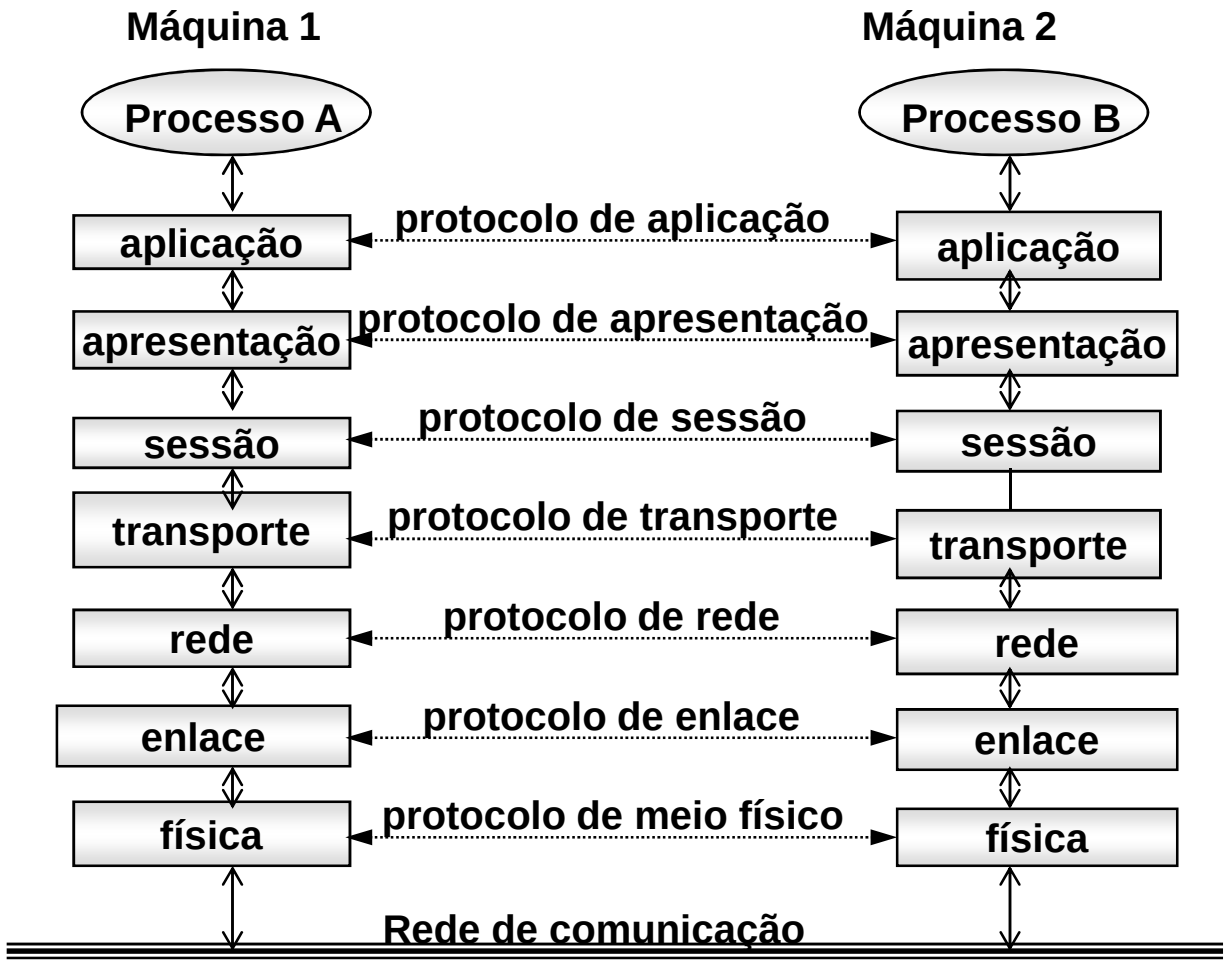
- Protocolo Orientado a Conexão
  - Emissor e Receptor devem estabelecer uma conexão (podendo negociar o protocolo usado) antes da troca de dados
    - Ex.: Comunicação telefônica
- Protocolo Sem Conexão
  - Não há conexão entre Emissor e Receptor

Modelo OSI

Modelo OSI

- Divisão do Modelo OSI
  - 7 Camadas
  - Cada camada trata de um aspecto da comunicação
  - Cada camada provê uma ***interface*** às adjacentes
    - Operações que definem os serviços da camada
  - Cabeçalhos e finalizadores podem ser acrescentados e retirados às mensagens

Modelo OSI


Máquina 1
Máquina 2
Processo A
Processo B
aplicação
protocolo de aplicação
aplicação
apresentação
protocolo de apresentação
apresentação
sessão
protocolo de sessão
sessão
transporte
protocolo de transporte
transporte
rede
protocolo de rede
rede
enlace
protocolo de enlace
enlace
física
protocolo de meio físico
física
Rede de comunicação

Modelo OSI

- A Camada Física
  - Responsável pela transmissão dos bits de informação (0's e 1's) no meio de comunicação
    - Padronização elétrica
    - Sincronização
    - Padronização de conectores
  - Ex.: RS-232-C
    - Comunicação serial

Modelo OSI

- A Camada de Enlace de Dados
  - Realiza a detecção e correção de erros
  - Padroniza o formato de um quadro
  - Adiciona bits de paridade, *start*, *stop* e *checksum*
  - Ordena os quadros e pede retransmissão de quadros perdidos

Modelo OSI

- A Camada de Rede
  - Determina as ***Rotas*** que os pacotes seguem do remetente ao destinatário
    - Roteamento
    - Custo

Modelo OSI

- A Camada de Rede
  - Procura a melhor rota
    - Distância
    - Velocidade (tráfego)
      - Atrasos
      - Variante com o tempo

Modelo OSI

- A Camada de Rede
  - Protocolos Orientados a Conexão
    - Rota é criada do emissor ao receptor
      - Caminho de transmissões futuras
    - Ex.: X.25
      - *Call Request*
        - Início de comunicação
        - Identificador de Conexão

Modelo OSI

- A Camada de Rede
  - Protocolos Sem Conexão
    - Ex.: IP
    - Pacote IP é enviado sem necessidade de início de comunicação
    - Pacotes seguem caminhos independentes

Modelo OSI

- A Camada de Transporte
  - Conexão confiável
    - Detecta e corrige perda de pacotes
    - Provê confiabilidade ponto a ponto
  - Pode ser construído sobre X.25 ou IP
    - X.25
      - Seqüência de Pacotes
    - IP
      - Ordem dos pacotes depende dos caminhos
    - TCP/IP e UDP/IP

Modelo OSI

A Camada de Sessão

- Provê sincronização
- Uma evolução da camada de transporte
- Permite inserir pontos de verificação em mensagens longas
- Raramente é implementada

Modelo OSI

- A Camada de Apresentação
  - Relacionada ao significado dos bits
    - Mensagens
  - Trata da representação da informação
  - Facilita a comunicação entre máquinas com diferentes formatos de dados

Modelo OSI

- A Camada de Aplicação
  - Miscelânea de protocolos para atividades comuns
    - E-mail
    - FTP
    - HTTP

Modelo Cliente Servidor

Modelo Cliente Servidor

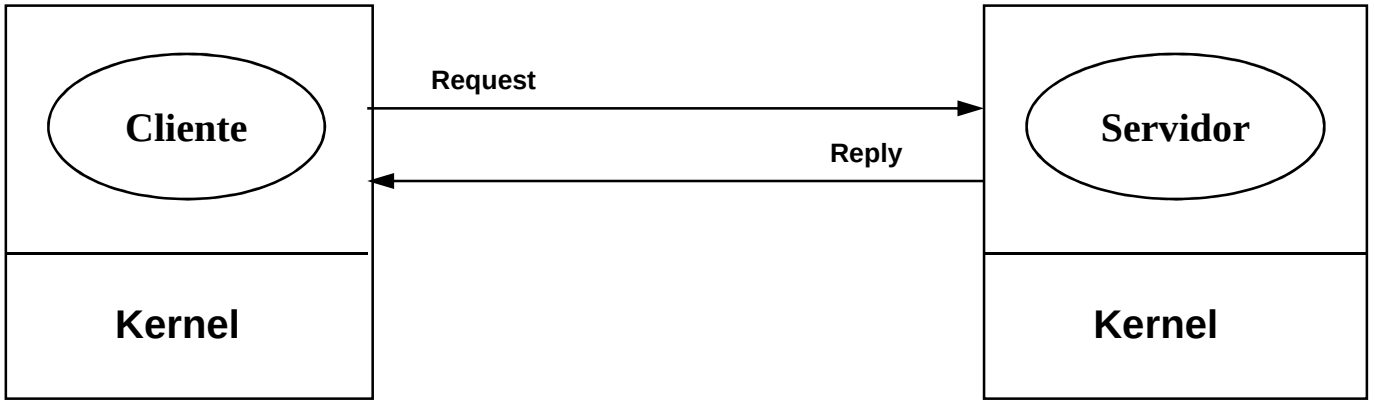
Modelo Cliente Servidor

- Estruturação de Sistemas Distribuídos
  - Sistema estruturado como uma coleção de processos cooperantes
    - Servidores
  - Serviços para Usuários
    - Clientes
  - Clientes e Servidores
    - Processos no modo Usuário

Modelo Cliente Servidor

- Baseado em protocolo simples
  - Evitar Overhead do OSI ou TCP/IP
  - Baseado em *Request* - *Reply*
    - Não orientado a conexão
- Somente 3 camadas são necessárias
  - Física
  - Enlace de dados
  - Camada de Sessão
    - Protocolo *Request* - *Reply*

Modelo Cliente Servidor



Modelo Cliente Servidor

Cliente

```
strcpy(Buffer, “Requisição para o Servidor...”);
send(enderecoServidor, Buffer, sizeof (Buffer), 0 );
recv(enderecoServidor, Buffer, sizeof(Buffer),0);
printf(“Reply do Servidor: %s", Buffer);
```

Servidor

```
recv(enderecoCliente, Buffer, sizeof (Buffer),0 );
printf(“MSG de request do cliente: %s, Buffer);
strcpy(Buffer, “Servidor Responde...”);
send(enderecoCliente,Buffer, sizeof(Buffer),0);
```

Endereçamento no MCS

- Envio de mensagens
  - Endereço do destinatário

- Endereçamento
  - Endereço de Máquina
    - Monitorado pelo *Kernel* local
    - Não identifica qual processo deve receber a mensagem
      - Múltiplos processos na máquina destino

Endereçamento no MCS

- Endereço de Processo:
  - Composto
    - Máquina + Processo (ID Local)
    - Kernel identifica o processo correspondente
      - Não há ambigüidade
  - Não é um esquema transparente
    - O usuário conhece a localização do servidor

Endereçamento no MCS

- Endereço de processo único e independente de máquina
  - Provedor de endereços centralizado
  - Problema com escalabilidade e confiabilidade

Endereçamento no MCS

- Localização do Servidor
  - Mensagem via Broadcast
    - Cliente envia Pacote de Localização
      - Endereço do processo destino
    - Cada *Kernel* avalia mensagem
      - O Servidor destinatário responde com seu endereço
      - O Cliente guarda o endereço
    - Esquema transparente
    - Carga na rede

Endereçamento no MCS

- Localização do Servidor
  - Servidor de Nomes
    - Processos servidores são referenciados por nomes (*strings* ASCII)
    - Cada nome é mapeado para o endereço de máquina + ID local do processo
    - Solução centralizada
      - Escalabilidade
      - Falha
    - Replicação
      - Vários servidores de nomes

Primitivas de Comunicação

- Envio e Recebimento de Mensagens
  - *Send* (destino, msg)
  - *Receive*(origem, msg)

- Características das Primitivas de Comunicação
  - Sincronização (Bloqueio)
  - *Bufferização (Caixa Postal)*
  - Confiabilidade (Controle de Fluxo)
    - Escolha dos projetistas de sistemas

Primitivas de Comunicação

- Sincronismo (Bloqueio)

```
1. strcpy(msg, "teste de envio com send");
2. send(porta, msg, sizeof(msg), 0)
3. strcpy(msg, "lixo");
```

  - *SEND* síncrono (com bloqueio)
    - Bloqueia o remetente até que a mensagem seja entregue ao destinatário
    - Após liberação a área de memória de **msg** está livre para ser manipulado
      - A linha 3 só será executada após o término do send da linha 2.
      - O receptor receberá *"teste de envio com send"*

Primitivas de Comunicação

- Sincronismo (Bloqueio)

1. **strcpy(msg, "lixo");**
2. **recv(porta, msg, sizeof(msg), 0)**
3. **printf(msg);**

  - *RECEIVE* síncrono
    - Bloqueia o processo até a mensagem ser colocada na área de memória de **msg**
    - A linha 3 só executa após o *recv* ter colocado a mensagem na variável *msg*. Se nada foi enviado, fica bloqueado aguardando.

  - Primitivas Síncronas
    - Simplicidade de implementação

Sincronismo

- *SEND* assíncrono (sem bloqueio)

```
1. strcpy(vetor[0].buf, "teste de envio com send assíncrono");
2. WSASend(conn_socket, vetor, 1, &bytes, 0, &overlap , NULL);
3. strcpy(vetor[0].buf, "lixo.....");
```

  - Retorna o controle ao emissor imediatamente, antes da mensagem ser enviada
  - Paralelismo entre o processo emissor e a transmissão da mensagem

Sincronismo

- *SEND* assíncrono (sem bloqueio)

```
1. strcpy(vetor[0].buf, "teste de envio com send assíncrono");
2. WSASend(conn_socket, vetor, 1, &bytes, 0, &overlap , NULL);
3. strcpy(vetor[0].buf, "lixo.....");
```

  - A área de memória com a mensagem (vetor[0].buf) não deve ser modificado até que a mensagem chegue ao destino
    - O *Kernel* deve fazer uma cópia da mensagem ou interromper o remetente para avisar que a área de memória com a mensagem está livre novamente

Sincronismo

- *RECEIVE* assíncrono

```
1. strcpy(vetor[0].buf, "lixo");
2. WSARecv(conn_socket, &vetor, 1, &bytes_transfclient, &flagclient,
                        &overlap, NULL);
3. printf(vetor[0].buf);
```

  - Informa ao *Kernel* onde a área de memória onde será colocada a mensagem recebida (vetor[0].buf) está localizado
  - Se ao executar a linha 2 não houver mensagem na caixa postal, a linha 3 imprime "lixo". Caso contrário, imprime a mensagem da caixa.

Sincronismo

- *RECEIVE* assíncrono

```
1. strcpy(vetor[0].buf, "lixo");
2. WSARecv(conn_socket, &vetor, 1, &bytes_transfclient, &flagclient,
                        &overlap, NULL);
3. printf(vetor[0].buf);
```

  - Permite que o receptor continue sua execução
  - O receptor pode utilizar uma primitiva para interrogar o núcleo sobre o estado da operação
  - *Receive* condicional com *timeout*

Sincronismo

- Primitivas e Escalabilidade

```
for (i = 0; i < 100; i++)
    send(sock[i], msg, 1024, 0);

for (i = 0; i < 100; i++)
    WSASend(sock[i], vetor[0].buf, 1, &bytes, ...);
```

➔ Maior Escalabilidade

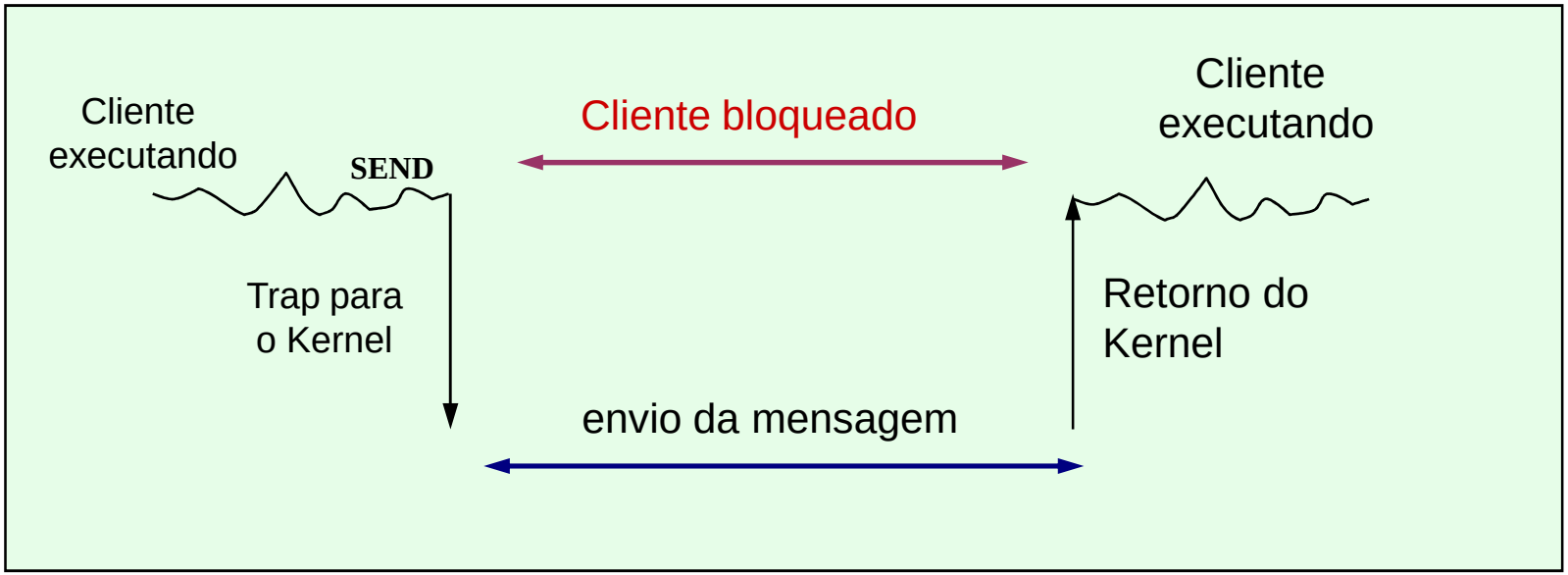
Se o modelo síncrono utilizasse THREADS a escalabilidade seria a mesma, mas o consumo de recursos do SO seria maior.

Sincronismo

- Primitivas Assíncronas
  - Maior complexidade e escalabilidade
  - Uso de *WAIT* e *TEST*
    - WSAGetOverlappedResult
      - Testa estado da transmissão via Kernel

Sincronismo

- Primitiva SEND com Bloqueio

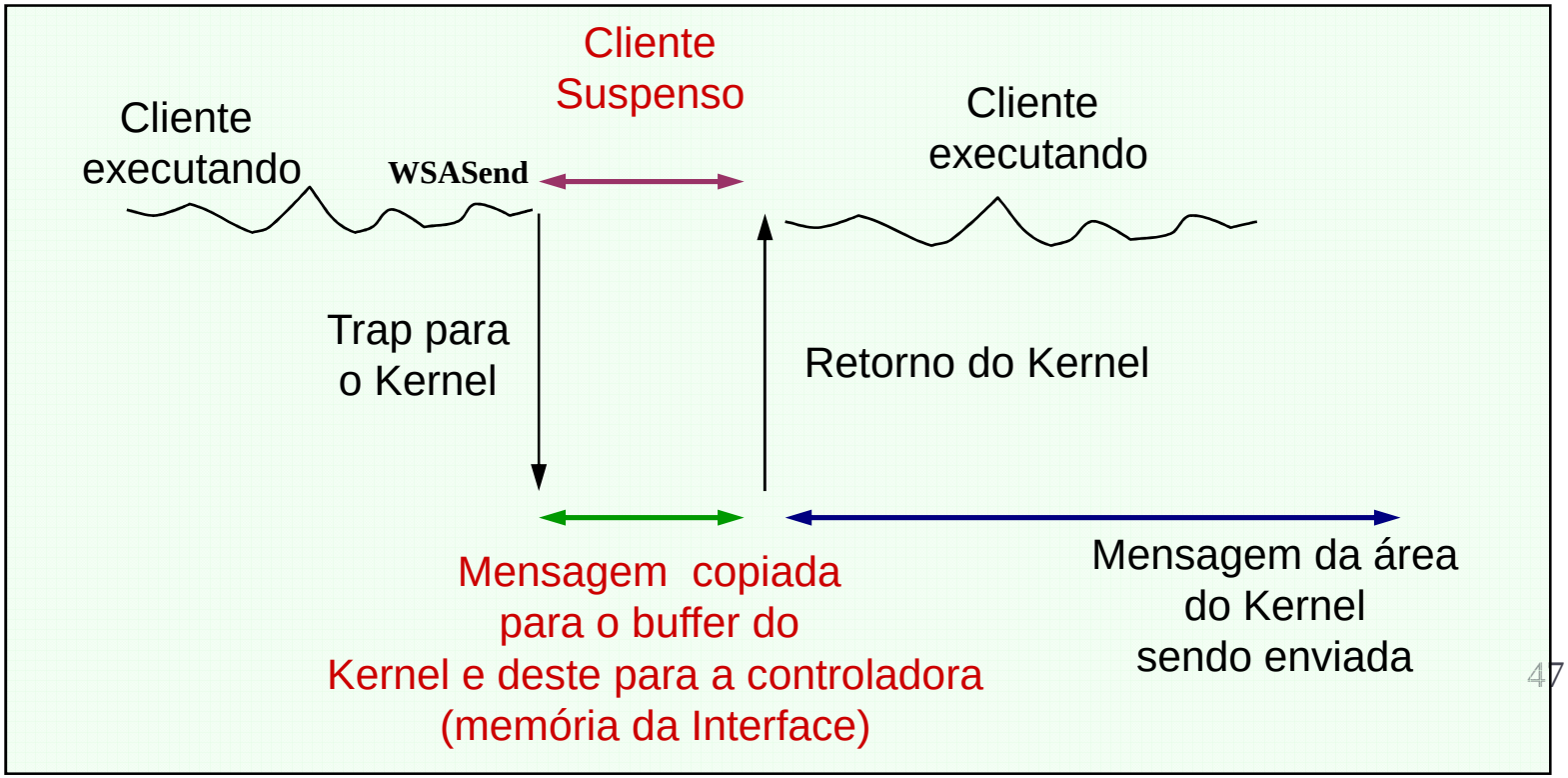


Sincronismo

- Primitiva SEND com Bloqueio
  - send(sock, msg, sizeof(msg),0);
  - Ponto de vista do emissor
    - Retorno do SEND significa que a área de memória da mensagem (msg) está Livre
    - Emissor sabe que a mensagem foi entregue
  - Simples

Sincronismo

- Primitivas SEND sem Bloqueio (Assíncrono)



Sincronismo

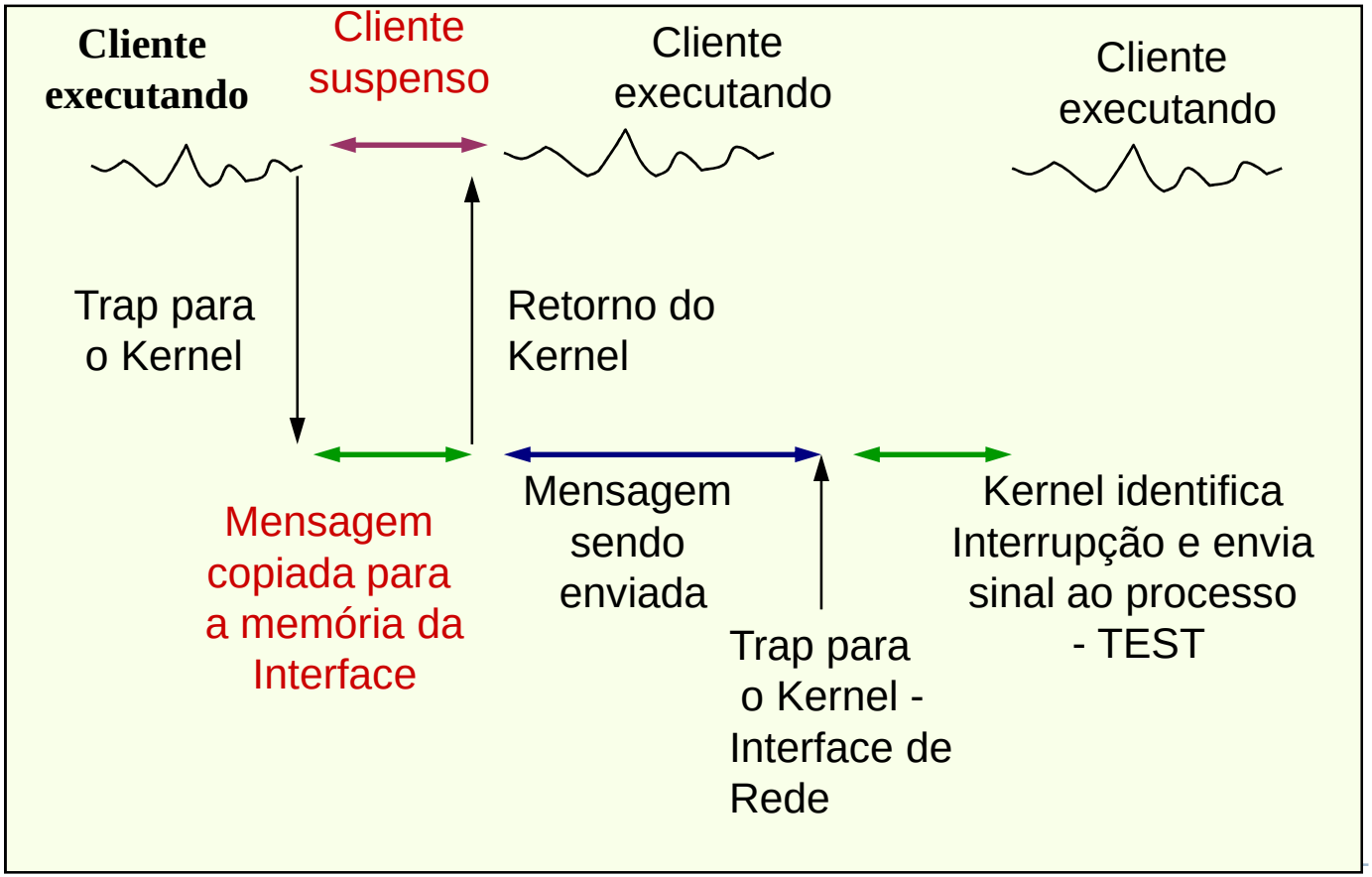
- Primitivas SEND sem Bloqueio
  - Ponto de vista do emissor
    - Semelhante ao com Bloqueio
  - Retorno do SEND significa que a área de memória da mensagem está Livre

Sincronismo

- Primitivas SEND sem Bloqueio
  - Emissor não sabe que a mensagem ainda não foi entregue
    - A mensagem é enviada em paralelo com a execução do cliente
      - Obrigação do cliente em testar a conclusão do envio com a primitiva TEST
  - Desvantagem
    - Cópia para área do *Kernel* (desnecessária)
    - Cópia para memória da Interface de rede

Sincronismo

Primitivas SEND sem Bloqueio



Sincronismo

- Primitivas SEND sem Bloqueio
  - Não requer cópia para o *Kernel*
    - Maior escalabilidade
    - Maior complexidade de código
    - Retorno NÃO significa que a área de memória da mensagem está Livre

Sincronismo

- Primitivas Síncronas
  - Definição I
    - Emissor pode reusar a área de mensagem imediatamente após o retorno do comando
      - Quando a mensagem efetivamente chega ao destino é irrelevante

Sincronismo

- Primitivas Síncronas
  - Definição 2
    - O emissor é bloqueado até o receptor aceitar a mensagem e o reconhecimento retorna
  - Timeout
    - Previne bloqueio eterno

Sincronismo

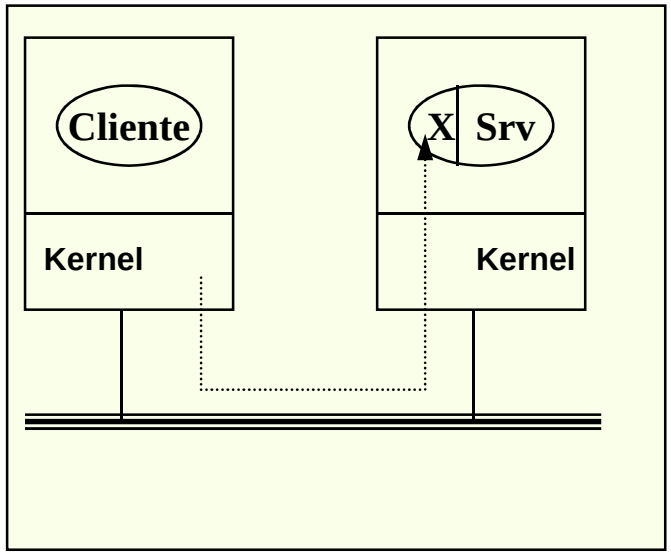
- *Desacordo*
  - Modo intermediário
    - Mensagem copiada ou copiada e enviada, mas sem reconhecimento do receptor
  - Visão do projetista de S.O
    - Definição 1
  - Programador
    - Definição 2

Bufferização

recv(sock, msg, sizeof (msg),0)

O RECEIVE determina:

1. O endereço onde o processo aguarda a mensagem – *sock*
2. O endereço de memória onde o conteúdo da mensagem deve ser armazenada - *msg*



Primitivas sem Bufferização

Bufferização

- Esse esquema funciona bem quando o Servidor executa o ***RECEIVE*** antes do Cliente efetuar o ***SEND***.
  - Comunicação simétrica
  - O que acontece quando o *SEND* é executado antes do *RECEIVE?*
    - O Kernel do Servidor não identifica qual o processo destinarário
      - Descarte de mensagem

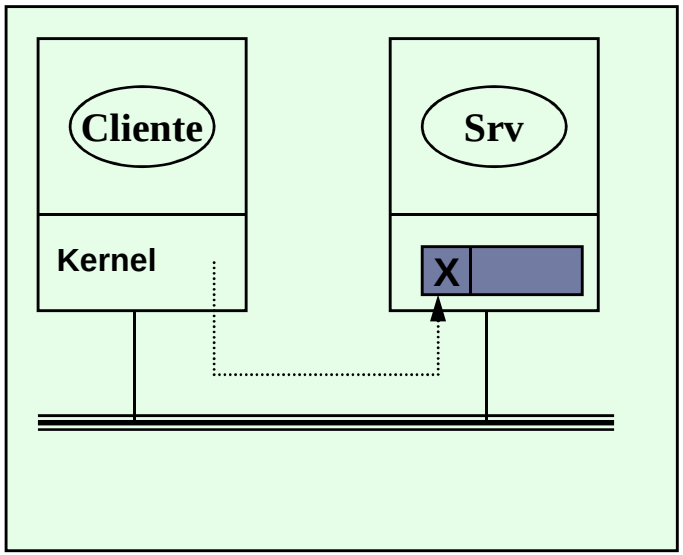
Bufferização

recv(sock, msg, sizeof (msg),0)

O RECEIVE determina:

1. O endereço onde o processo aguarda a mensagem – *sock*
2. O endereço de memória onde o conteúdo da mensagem deve ser armazenada - *msg*
3. ***O Kernel reserva uma área de memória para copiar a mensagem antes de entregar ao processo***
    1. ***Buffer do Kernel***
    2. ***Mailbox***

Primitivas com Bufferização



Bufferização

- *Definição*
  - Esquema de caixa de mensagem
    - Controle do *Kernel*
  - O que acontece quando o *SEND* é executado antes do *RECEIVE?*
    - O *Kernel* do Servidor guarda a mensagem até o processo destinatário requisitá-la (*RECEIVE*) ou ocorrer tempo de descarte (*timeout*)

Bufferização

- Primitivas não *bufferizadas*
  - Descarte de mensagem
    - O *Kernel* assume que o emissor vai retransmitir quando o seu *timeout* expirar
    - O cliente pode precisar retransmitir várias vezes até obter sucesso
    - O cliente pode desistir, interpretando uma falha no servidor ou no endereçamento da mensagem
    - Quando a carga é alta, a retransmissão contribui para o congestionamento da rede

Bufferização

- Primitivas não *bufferizadas*
  - *Timeout* antes do descarte
    - O núcleo espera um tempo predefinido antes de descartar a mensagem

Bufferização

- Caixas postais
  - *Buffers* definidos pelo processo destinatário
    - Uma caixa postal por socket
    - Mensagens são enviadas para caixas postais
  - Finitas
    - Podem encher, recaindo no problema do descarte de mensagens

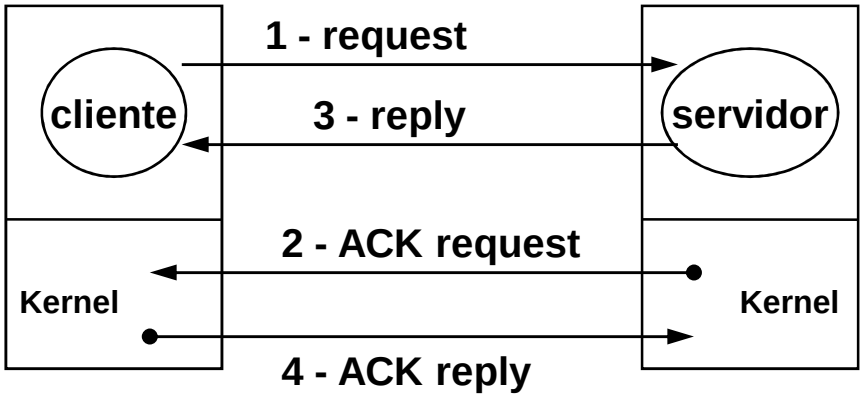
Bufferização

- Técnicas de Controle de Fluxo
  - Bloqueio do cliente até confirmação de recebimento
    - Se a caixa postal do destino estiver cheia, retorna IP do emissor e bloqueia
      - Liberação causa novo SEND

Confiabilidade

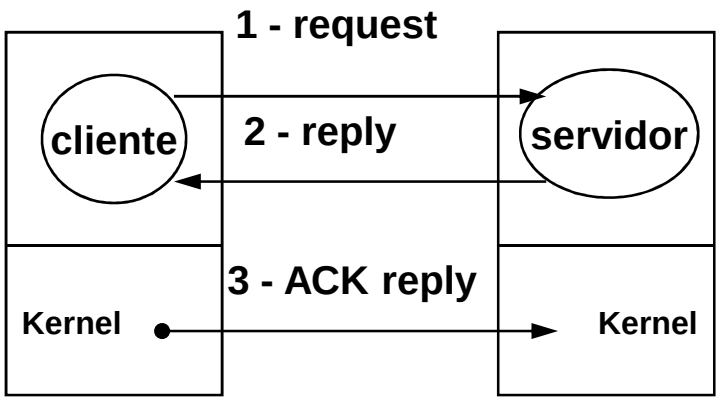
- Quando o Cliente envia uma mensagem, o Servidor *DEVE* recebê-la
  - Perda de mensagem?
- Primitiva *SEND* não confiável
  - Não há garantia de entrega
- *SEND* Confiável
  - Reconhecimento pelo Kernel receptor
    - Kernel no emissor libera Processo Cliente

Confiabilidade



Confiabilidade

- *SEND* Confiável
  - Resposta do Servidor (REPLY) serve de reconhecimento
  - ACK Reply é necessário?
    - Depende do Request.
      - Idempotência
      - Computação no servidor
  - Perda de mensagem
    - Timeout



Modelo Cliente Servidor

Código	Tipo	De	Para	Descrição
REQ	request	cliente	servidor	o cliente solicita serviço
REP	reply	servidor	cliente	resposta do servidor
ACK	acknowledge	ambos	ambos	ack do último pacote ou mensagem
AYA	are you alive?	cliente	servidor	o servidor está no ar?
IAA	I am alive	servidor	cliente	o servidor está no ar!
TA	try again	servidor	cliente	o buffer do servidor está cheio
AU	Adress unknown	servidor	cliente	nenhum processo usa esse endereço

Modelo Cliente Servidor

- Problemas do Modelo
  - O paradigma de comunicação é centrada em Entrada e Saída (I/O)
    - *SEND* e *RECEIVE*
  - Como E/S não é um conceito fundamental em sistemas centralizados
    - Falta de Transparência

Modelo Cliente Servidor

- Problemas do Modelo
  - O principal objetivo de um sistema distribuído é se parecer com um sistema centralizado
    - Como fazer que a programação distribuída utilize os mesmos conceitos da programação centralizada?
    - E/S não é a solução.