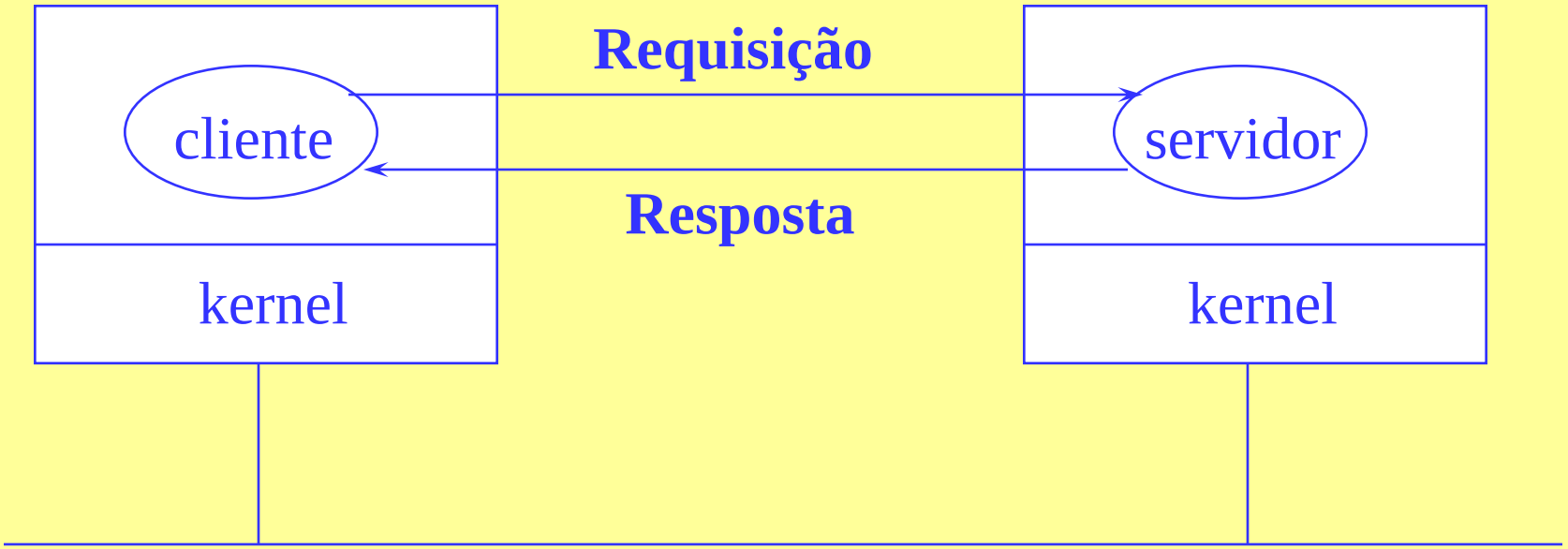
Comunicação em Sistemas Distribuídos

A diferença mais importante entre os Sistemas Distribuídos e os Sistemas Uniprocessadores é a comunicação inter-processo. Nos uniprocessadores esta comunicação é feita por meio de memória partilhada. Nos SDs não tem memória partilhada e toda a comunicação inter-processo tem que ser reformulada.

Modelo Cliente - Servidor

- A idéia é estruturar o S.O. como um grupo de processos cooperativos chamados *clientes e servidores;*
- *Servidores* oferecem serviços e *Clientes* usam os serviços oferecidos;
- Uma máquina pode executar um único processo *cliente* ou *servidor,* pode executar múltiplos *clientes*, múltiplos *servidores*, ou ainda uma combinação dos dois.

Modelo Cliente - Servidor



Principais Vantagens do Modelo

- Simplicidade;
- Não tem necessidade de estabelecer uma conexão (e finalizar a conexão);
- A mensagem de resposta serve de acknowledgement;
- Eficiência;
- Se as máquinas forem idênticas são necessários somente três níveis de protocolo (físico, data link e Request/Reply)

Serviços de Comunicação do kernel

São necessárias duas chamadas do Sistema:

- *send (destino, &ptmsg)*

  envia uma mensagem apontada por *ptmsg* para o processo identificado por *destino*, bloqueando o processo que executou o *send* até que a mensagem tenha sido enviada;

Serviços de Comunicação do kernel

- *receive (endereço, &ptmsg)*

  bloqueia o processo que executou o *receive* até que uma mensagem chegue. Quando ela chega é copiada para o buffer apontado por *ptmsg*. O parâmetro *endereço* específica o endereço do qual o receptor está esperando uma mensagem.

Exemplo de Cliente e Servidor

header.h

```
/* Definições dos Clientes e Servidores  */
#define MAX_PATH           255
#define BUF_SIZE           1024
#define FILE_SERVER        243
/* Definições das operações permitidas */
#define CREATE        1
#define READ          2
#define WRITE         3
#define DELETE        4
/*  Códigos de Erros   */
#define OK                    0
#define E_BAD_OPCODE     -1
#define E_BAD_PARAM      -2
#define E_IO                  -3
```

Continuação (header.h)

```
/* Definições do formato de mensagem */
struct message {
        long source;
        long dest;
        long opcode;
        long count;
        long offset;
        long extra1;
        long extra2;
        long result;
        char name [MAX_PATH];
        char data[BUF_SIZE];
};
```

Exemplo Simples de um Cliente

```
#include <header.h>
void main (void)
{  struct message m1, m2;
     int r;
     while (1) {
          receive (FILE_SERVER, &m1);
          switch (m1.opcode) {
               case CREATE:     r = do_create (&m1, &m2);   break;
               case READ:       r = do_read (&m1, &m2);     break;
               case WRITE:      r = do_write (&m1, &m2);    break;
               case DELETE:     r = do_delete(&m1, &m2);    break;
               default:         r = E_BAD_OPCODE;  }
          m2.result = r;
          send (m1.source, &m2);  }
}
```

Exemplo Cliente usando Servidor
(Copia de um arquivo)

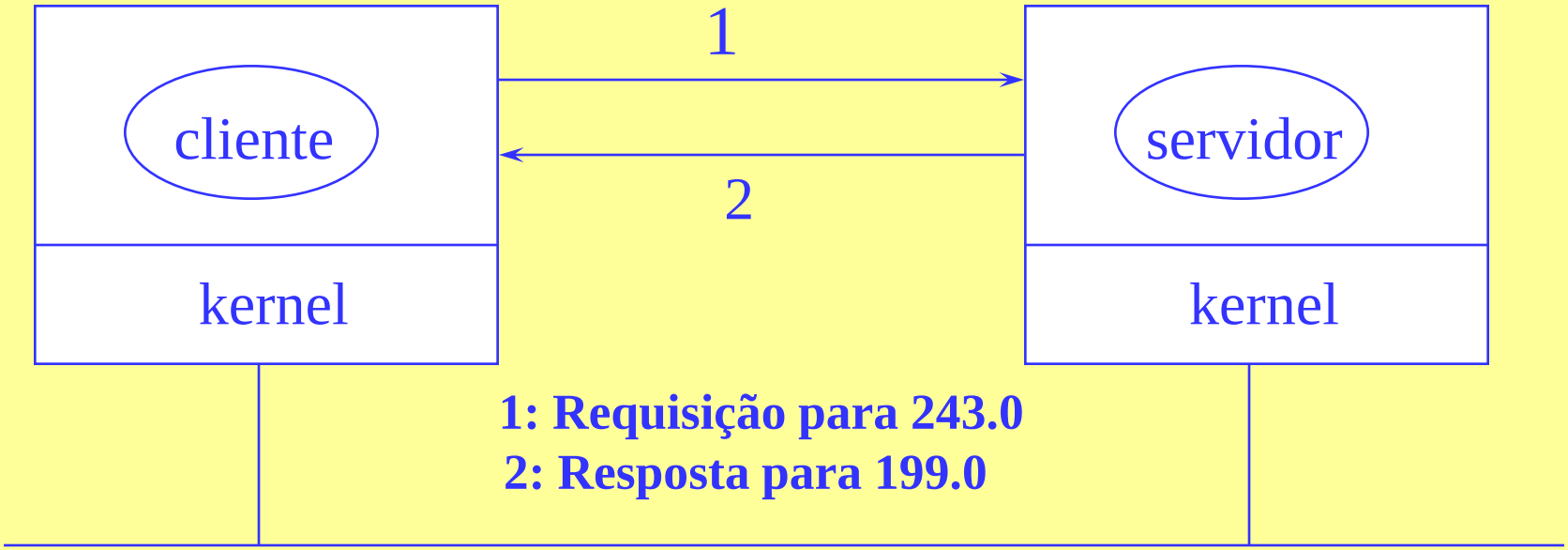
```
#include <header.h>
int copy(char *src, char *dst)
{   struct message m1;
     long position, client = 110;
      initialize();
      position = 0;
      do {
            /* Pega um bloco de dado do arquivo fonte */
            m1.opcode = READ;
            m1.offset = position;
            m1.count = BUF_SIZE;
            strcpy(&m1.name, src);
             send (FILE_SERVER, &m1);
            receive (client, &m1);
```

Continuação (Copia de um arquivo)

```
        /* Escreve o bloco de dado recebido no arquivo destino */
        m1.opcode = WRITE;
        m1.offset = position;
        m1.count = m1.result;
        strcpy(&m1.name, dst);
         send (FILE_SERVER, &m1);
        receive (client, &m1);
        position += m1.result;
    } while (m1.result > 0);
    return (m1.result >= 0 ? ok : m1.result);
}
```

Endereçamento

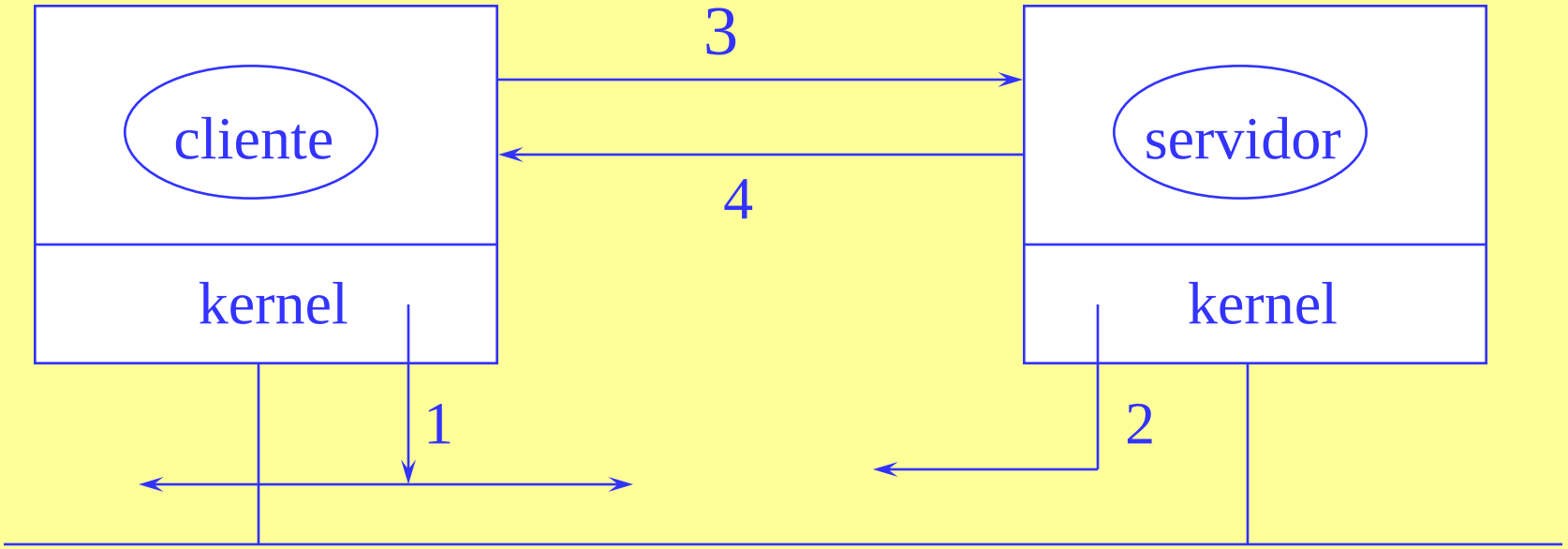
1- Endereçamento máquina.processo



- Não é transparente (se um servidor não estiver disponível teremos recompilação para poder realizar o serviço em outro servidor
- Não precisa de coordenação global

Continuação (Endereçamento)

1- Endereçamento randômico (gera carregamento extra no sistema)



1: Broadcast

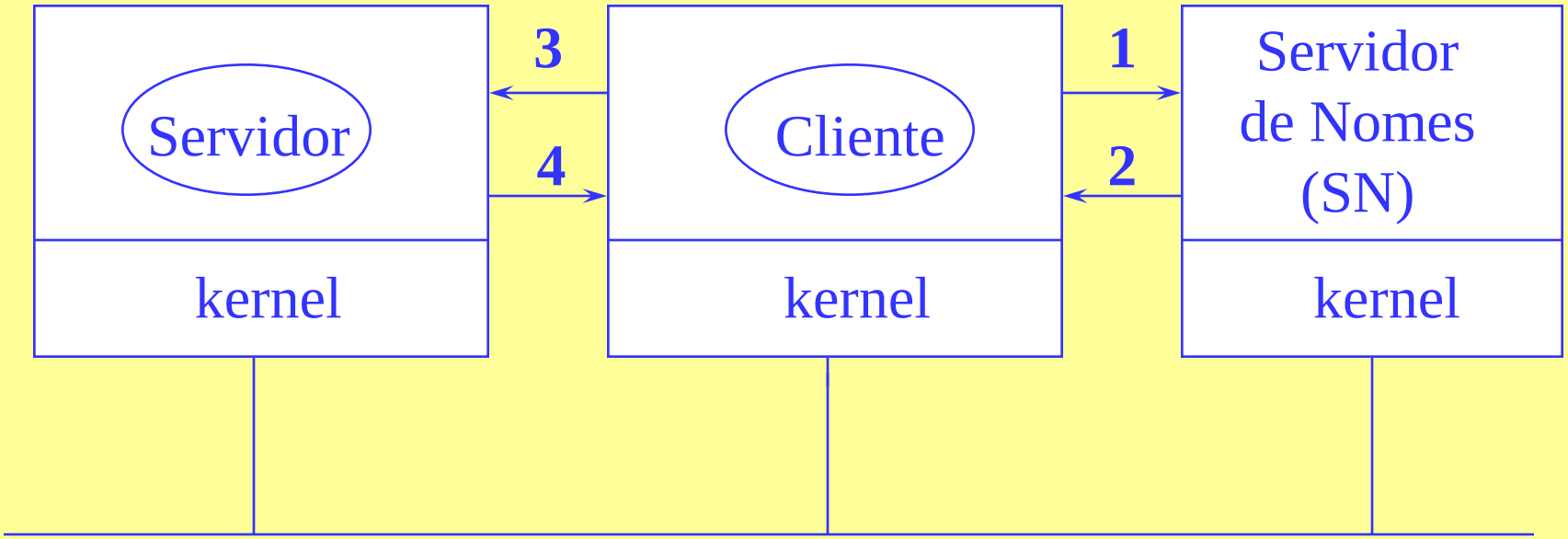
2: Estou aqui

3: Requisição

4: Resposta

Continuação (Endereçamento)

3- Endereçamento usando um Servidor de Nomes



1: Buscar nome

2: resposta do SN

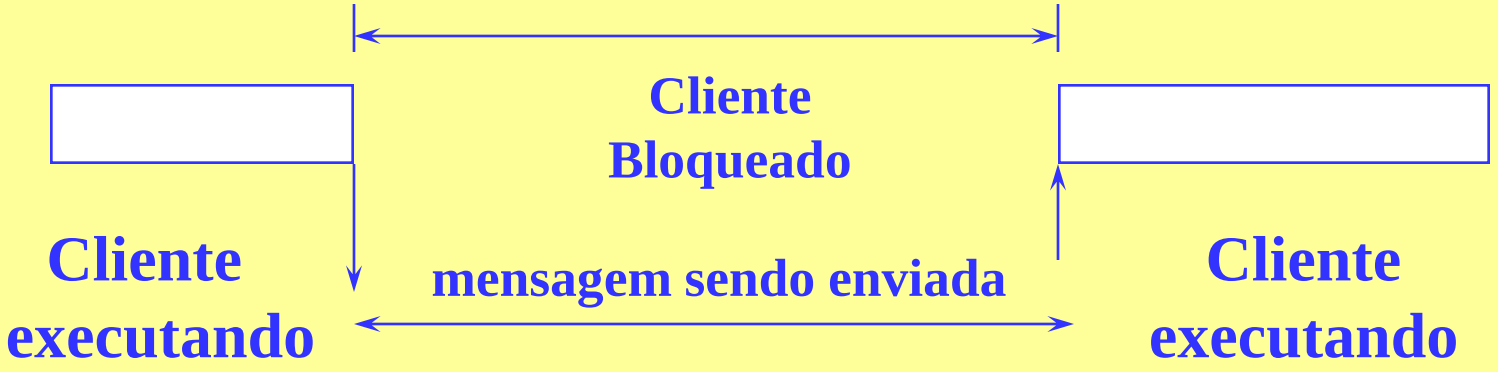
3: Requisição

4: Resposta

Primitivas Bloqueadas e Não Bloqueadas

Primitivas vistas até agora (send e receive) são chamadas primitivas bloqueadas (síncronas).

Enquanto a mensagem está sendo enviada ou recebida, o processo permanece bloqueado (suspenso).



Primitivas Não Bloqueadas (assíncronas)

- Quando um *send* é executado o controle retorna ao processo antes da mensagem ser enviada;
- O processo que executa o *send* pode continuar a executar enquanto a mensagem é enviada;
- A escolha do tipo de primitiva a ser usado (bloqueada ou não bloqueada) é feita pelo projetista do sistema (normalmente é usado um dos dois tipos, em poucos casos temos os dois tipos disponíveis);

- O processo que executa o send não pode modificar o buffer de mensagem até a mensagem ter sido enviada; tem dois modos para o processo ficar sabendo que a mensagem já foi enviada:

  1- *Send* com cópia - desperdício do tempo de CPU com a cópia extra;

  2- *Send* com interrupção - programação muito mais difícil

Primitiva *Receive* Não Bloqueada

Simplesmente informa o kernel onde é o buffer para receber a mensagem e retorna o controle ao processo que o executou.

Tem três modos do *receive* saber que a recepção já acabou:

- Primitiva wait;
- *Receive* condicional;
- interrupção.

Timeout

Nas primitivas bloqueadas geralmente é usado um “timeout” para que a primitiva não fique bloqueada indefinidamente. É especificado um tempo de resposta depois do qual, se nenhuma mensagem chegar, o send termina retornando um código de erro.

Primitiva Bufferizada e Não-Bufferizada

As primitivas descritas até agora são não-bufferizadas. O que acontece com um processo que chama *receive* (endereço, &m) quando o *receive* é executado depois do *send*? Como o kernel sabe onde colocar a mensagem que chegar? Não sabe!

Solução: Descartar a mensagem e esperar o timeout expirar. A mensagem é então retransmitida.

Primitiva Bufferizada

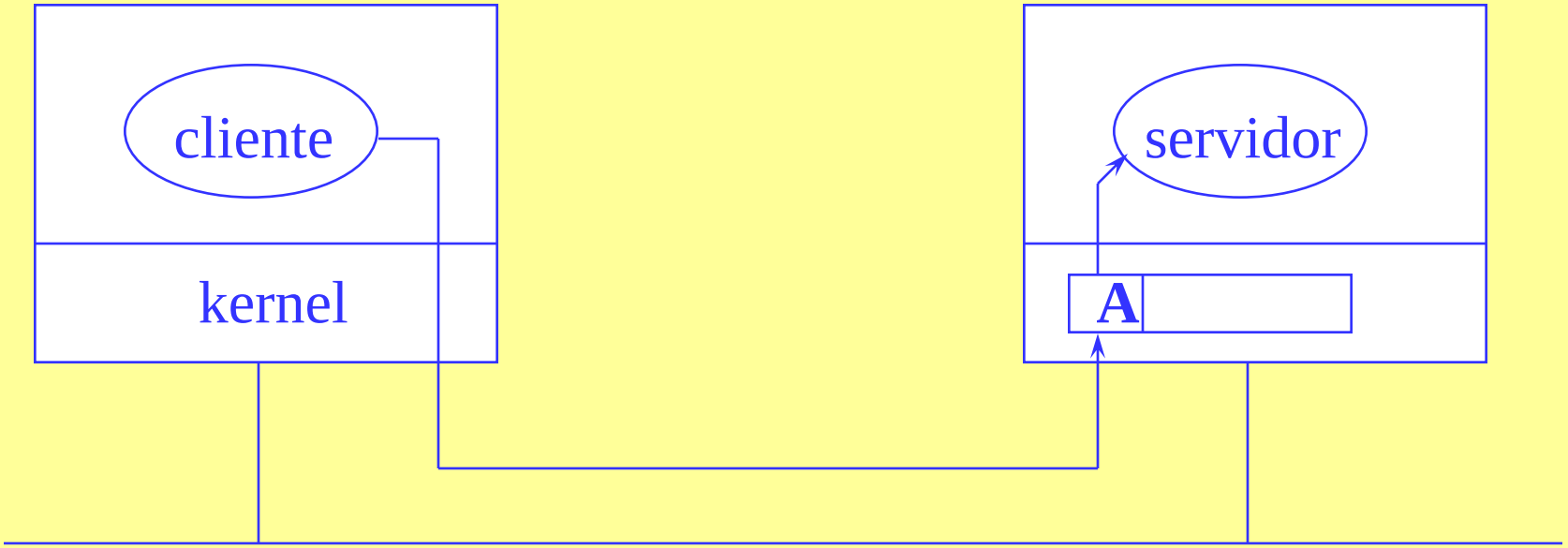
O kernel receptor mantém as mensagens armazenadas por um período de tempo, esperando que primitivas *receive* sejam executadas.

- Reduz a chance da mensagem ser descartada;
- Problema de armazenamento e gerenciamento de mensagens;
- Buffers precisam ser alocados.

Implementação

Definição de uma estrutura de dados chamada *mailbox*. Um processo interessado em receber mensagens pede ao kernel para criar um *mailbox* para ele. O *mailbox* recebe um endereço para poder ser manipulado. Toda mensagem que chega com aquele endereço é colocada no *mailbox*. *Receive* agora simplesmente remove uma mensagem do *mailbox*, ou fica bloqueado se não tem mensagem.

Mailbox



Primitiva Confiável e Não-Confiável

Até agora temos assumido que quando um cliente envia uma mensagem o servidor a receberá. O modelo real é um pouco mais complicado que o modelo abstrato; mensagens podem ser perdidas, afetando desta forma a semântica do modelo de comunicação.

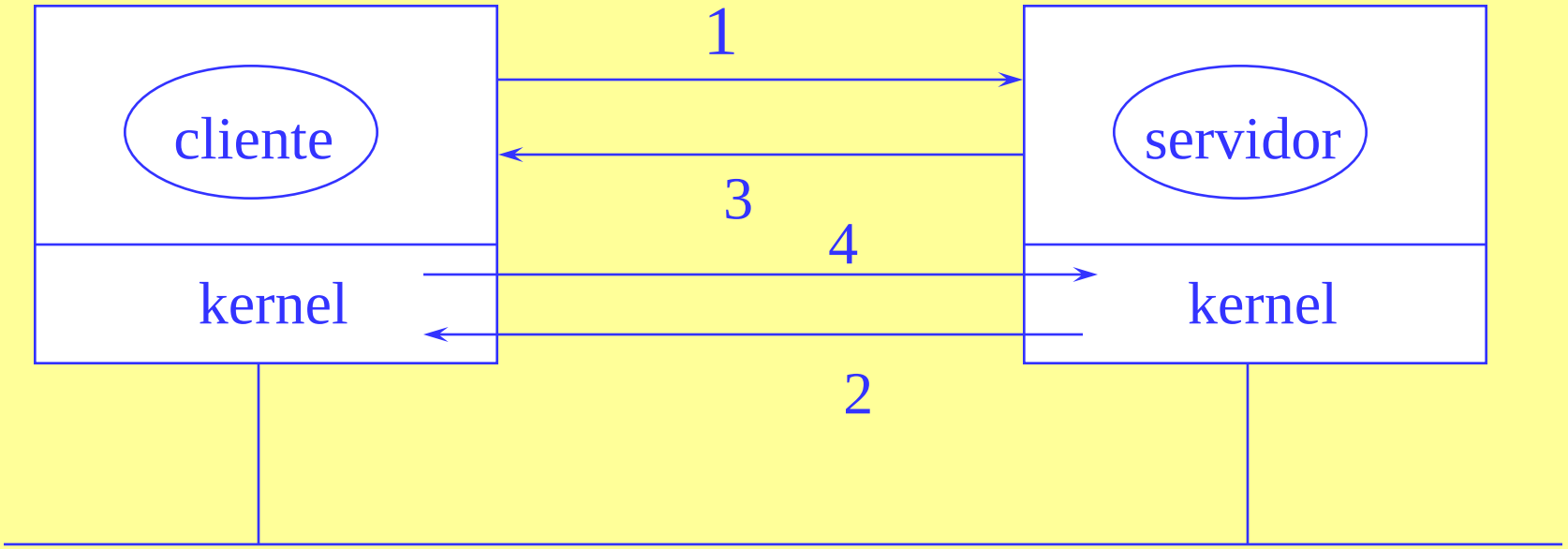
No caso de uma primitiva bloqueada estar sendo usada, quando um cliente envia uma mensagem ele é suspenso até que a mensagem ter sido enviada. Entretanto, não existe garantia de que quando ele for reativado a mensagem terá sido entregue.

Primitiva Confiável e Não-Confiável (...Cont.)

Tem três abordagens para o problema:

- Redefinir a semântica do *send* para ser não-confiável: O sistema não dá garantias sobre mensagens sendo enviadas; isto é, tornar a comunicação confiável é tarefa do usuário
- Requerer que o kernel da máquina receptora envie um “acknowledgment” para o kernel da máquina transmissora: o kernel só libera o cliente quando o “acknowledgment” for recebido. O “acknowledgment” é uma operação realizada pelos dois kernels, sem o conhecimento do cliente e servidor.

Mensagens de ACK individual

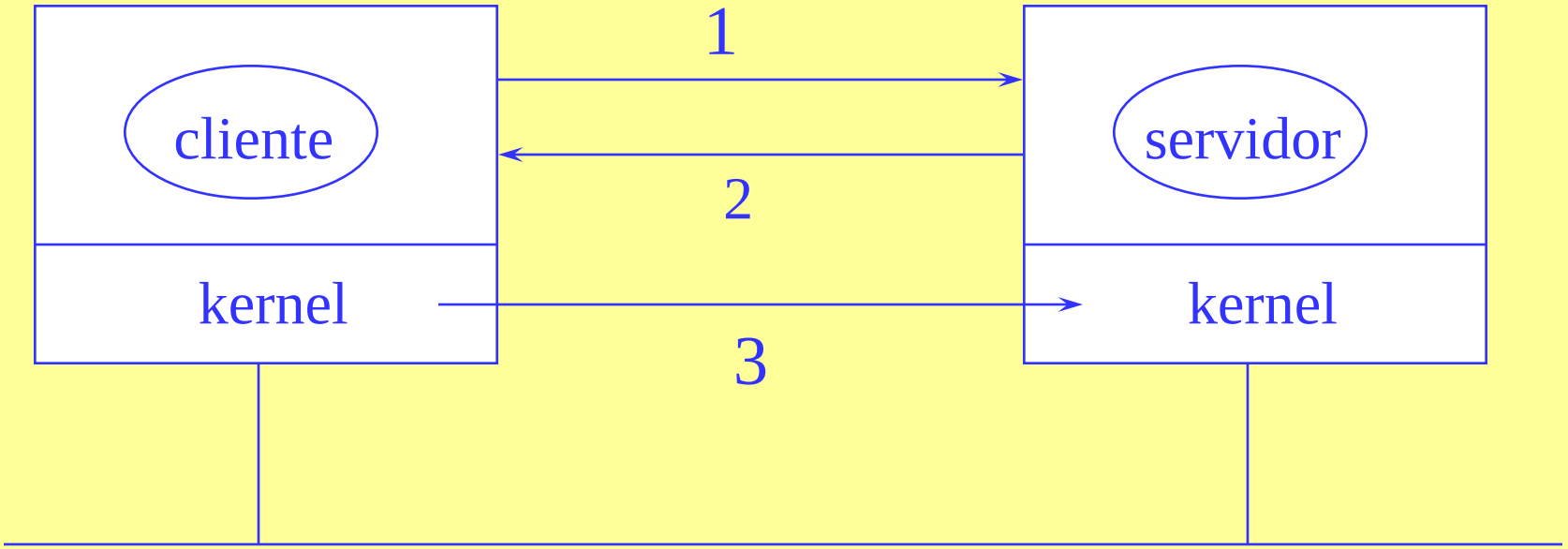


1: Requisição (Cliente para Servidor)
2: ACK (Kernel para Kernel)
3: Resposta (Servidor para Cliente)
4: ACK (Kernel para Kernel)

Primitiva Confiável e Não-Confiável (...Cont.)

- Aproveitar o fato que a comunicação cliente-servidor é estruturada como uma requisição do cliente para o servidor seguido de uma resposta do servidor para o cliente: O cliente é bloqueado depois de enviar a mensagem; o kernel do servidor não envia um “acknowledgment”, em vez disto a resposta serve de “acknowledgment”. Desta forma o processo que envia a mensagem permanece bloqueado até a resposta chegar. Se isto demorar muito o kernel pode reenviar a requisição, se precavendo contra a perda de mensagem.

Resposta sendo usada como ACK



1: Requisição (Cliente para Servidor)
2: Resposta (Servidor para Cliente)
3: ACK (Kernel para Kernel)

Implementação do Modelo Cliente-Servidor

Item	*Opção 1*	*Opção 2*	*Opção 3*
Endereçamento	Número de Máquina	Endereço aleatório	Nomes ASCII com Servidor
Bloqueamento	Primitivas Bloqueadas	Não-Bloqueada com cópia para o Kernel	Não-Bloqueada com interrupção
Buferização	Não-Buferizado, descartando mensagens não esperadas	Não-Buferizado, mantendo temporariamente mensagens não esperadas	mailboxes
Confiabilidade	Não-Confiável	Requisição-ACK-Resposta-ACK	Requisição-Resposta-ACK

Protocolo Cliente-Servidor

Código	Tipo	De	Para	Descrição
REQ	Requisição	Cliente	Servidor	O cliente quer serviço
REP	Resposta	Servidor	Cliente	Resposta do servidor para o cliente
ACK	ACK	Cliente Servidor	Cliente Servidor	O pacote anterior chegou
AYA	Are you alive?	Cliente	Servidor	Testar se o servidor não continua ativo
IAA	I am alive	Servidor	Cliente	O servidor continua ativo
TA	Try again	Servidor	Cliente	O servidor não pode atender
AU	Address unknown	Servidor	Cliente	Nenhum processo está usando este endereço

Implementando o modelo Cliente-Servidor(Cont)

Detalhes de como a passagem de mensagem é implementada depende das escolhas feitas durante o projeto. Algumas considerações:

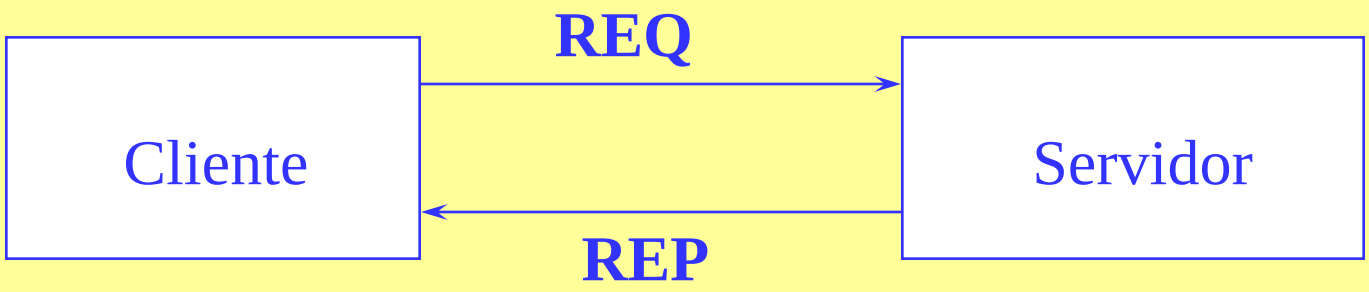
- Há um tamanho máximo do pacote transmitido pela rede de comunicação;
- Mensagens maiores precisam ser divididas em múltiplos pacotes que são enviados separadamente
- Alguns dos pacotes podem ser perdidos ou chegar na ordem errada. Solução: atribuir a cada mensagem o número da mensagem e um número de sequência

Implementando o modelo Cliente-Servidor(Cont)

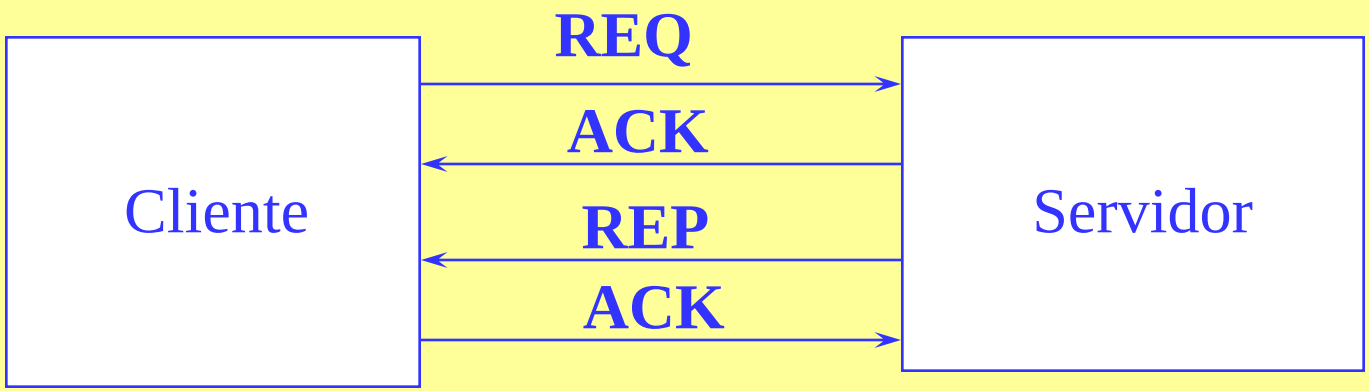
- O “acknowledgment” pode ser para cada pacote individual ou para a mensagem como um todo. No primeiro caso na perda de mensagem, somente um pacote precisa ser retransmitido, mas na situação normal requer mais pacotes na rede de comunicação; no segundo caso há a vantagem de menos pacotes na rede mas a desvantagem da recuperação no caso de perda de mensagem é mais complicada.

Conclusão: a escolha de um dos dois métodos depende da taxa de perdas na rede.

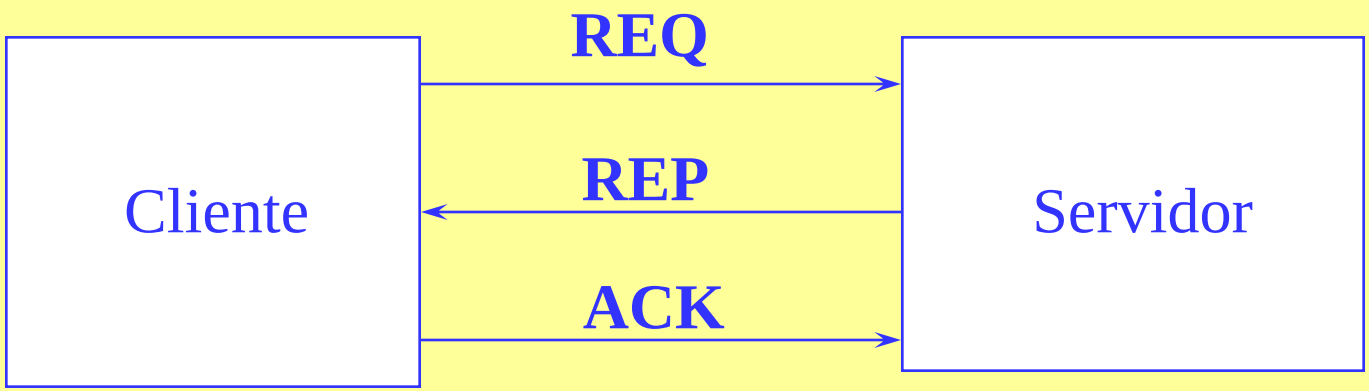
Protocolo Usado na Comunicação Cliente-Servidor (a)



Protocolo Usado na Comunicação Cliente-Servidor (b)



Protocolo Usado na Comunicação Cliente-Servidor (c)



Protocolo Usado na Comunicação Cliente-Servidor (d)