SSC150 – Sistemas Computacionais Distribuídos

Comunicação em Sistemas Distribuídos

2ª aula
11/03/10

Profa. Sarita Mazzini Bruschi
sarita@icmc.usp.br

Slides baseados no material de:
Prof. Rodrigo Mello (USP / ICMC)
Prof. Edmilson Marmo Moreira (UNIFEI / IESTI)

Introdução

- Principal diferença entre sistema distribuídos e sistemas uniprocessados é a forma de comunicação entre os processos
- Uniprocesso:
  - Memória compartilhada
- Sistemas Distribuídos:
  - Troca de mensagens

Exemplo

- Quando um processo A quer se comunicar com um processo B
  - Primeiramente deve-se construir uma mensagem em seu espaço de endereçamento
  - Depois executa uma chamada de sistema que faz com que o sistema operacional envie a mensagem para B através da rede de comunicação

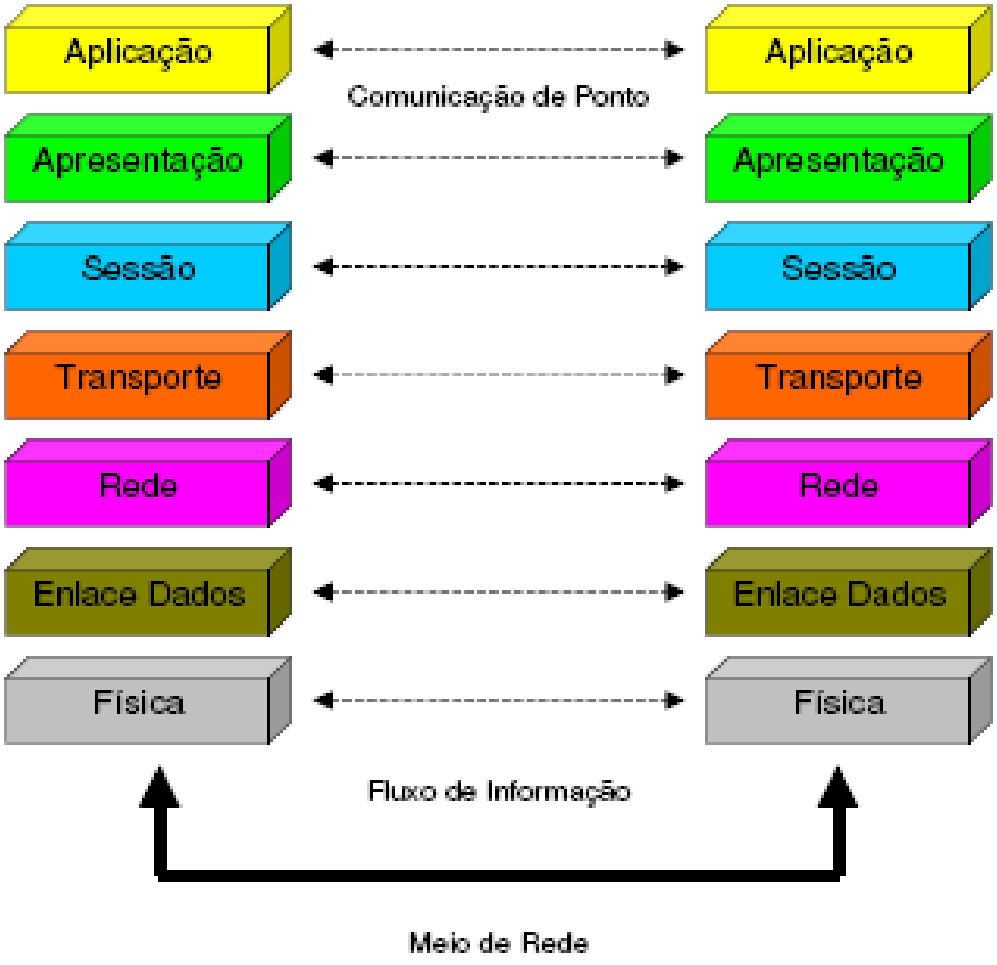
Exemplo

- Porém, A e B devem concordar em como os bits serão enviados:
  - Quantos volts devem ser usados para o sinal do bit 0 e quantos volts devem ser usados para o bit 1?
  - Como o processo receptor reconhece o último bit da mensagem?
  - Como se pode detectar se uma mensagem foi perdida ou está corrompida? E o que deve ser feito para encontrá-la ou corrigi-la?
  - Qual o tamanho em bits dos números, strings, e ouros tipos de dados, e como eles são representados?

Protocolos em camadas

- Modelo em camadas OSI, especificado pela ISO (*International Standards Organization*)
  - Modelo de Referência para Interconexão de Sistemas Abertos
- Padroniza camadas de protocolos
- Protocolo
  - Acordo sobre como a comunicação deve ser feita

Protocolos em camadas



Protocolos em Camadas

- Cada camada fornece serviço para a camada imediatamente superior
- Um processo pede para sua camada montar uma mensagem
  - Mensagem é composta de cabeçalho + dados
- Cada camada requisita serviços das camadas inferiores que adicionam cabeçalhos e tratamentos à essa mensagem

Nível Físico

- Trata da transmissão de 0s e 1s
- Principais pontos:
  - Trata dos volts necessários para representar 0s e 1s lógicos
  - Quantos bits serão transmitidos por segundo
  - Transmissão pode ser feita simultaneamente ou nos dois sentidos?
  - Tamanho e formato dos receptores
  - Interfaces de redes (placas de redes)

Nível de Enlace de Dados

- O nível físico faz a troca de bits mas não trata erros de comunicação (perda de informação)
- Principais tarefas:
  - Implementar detecção e correção de erros
  - Os bits são agrupados em quadros
  - Checa um determinado padrão de bits no início e no fim do quadro e faz checksum, o qual passa a fazer parte integrante do quadro

Nível de Enlace de Dados

- ...
  - Quando um quadro é recebido, o checksum é calculado novamente
    - Se igual, quadro está correto,
    - Errado: receptor requisita novamente o quadro, através dos IDs
  - Ineficiente

Nível de Rede

- Em uma rede local, o transmissor não precisa localizar o receptor
  - A mensagem é colocada na rede
  - O receptor testa cabeçalho da mensagem para saber se é para ele ou não
- Em redes de longa distância existe um grande número de máquinas
  - Para uma mensagem sair do transmissor e chegar ao receptor são necessárias escalas

Nível de Rede

- Redes de longa distância
  - A mensagem deve escolher um entre os caminhos disponíveis
  - Roteamento
    - escolha do menor caminho
    - nem sempre o menor caminho físico é o melhor caminho
      - Pode haver congestionamento
      - Algoritmos de roteamento consideram o atraso nas rotas para escolha dos melhores caminhos

Nível de Rede

- Redes de longa distância
  - Exemplo de protocolo: IP (*Internet Protocol*)
    - Parte da pilha de protocolos do DoD
    - A mensagem nessa camada é denominada pacote
    - Um pacote IP não requer conexão entre transmissor e receptor
    - Cada pacote IP pode ser roteado de maneira distinta de todos os outros pacotes

Nível de Transporte

- Os pacotes podem ser perdidos no caminho entre o transmissor e o receptor
- Apesar de algumas aplicações tratarem internamente essa perda, outras preferem utilizar um protocolo que trate isso automaticamente
- Essa é a função do nível de transporte

Nível de Transporte

- Exemplos de protocolos:
  - TCP (*Transmission Control Protocol*)
    - Orientado à conexão
    - Confiável
    - Não existe rota pré-definida
      - Pacotes podem chegar fora de ordem
    - Software reordena esses pacotes
      - Dá a impressão de um único caminho de transmissão
  - UDP (*User Datagram Protocol*)
    - Não é confiável
    - Não é orientado à conexões
    - Não ordena pacotes
    - Mais veloz, não usa ACKs

Nível de Sessão

- Não está presente na pilha TCP/IP
- Controla pontos de sincronização
- Exemplo mais simples :
  - Quando uma conexão é feita a um site que tem suporte à sessões um ID (cookie ou url rewritten) sabe que usuário está tentando novas conexões
  - Consegue identificar usuários, reiniciando suas operações e podendo tratá-los de maneira personalizada

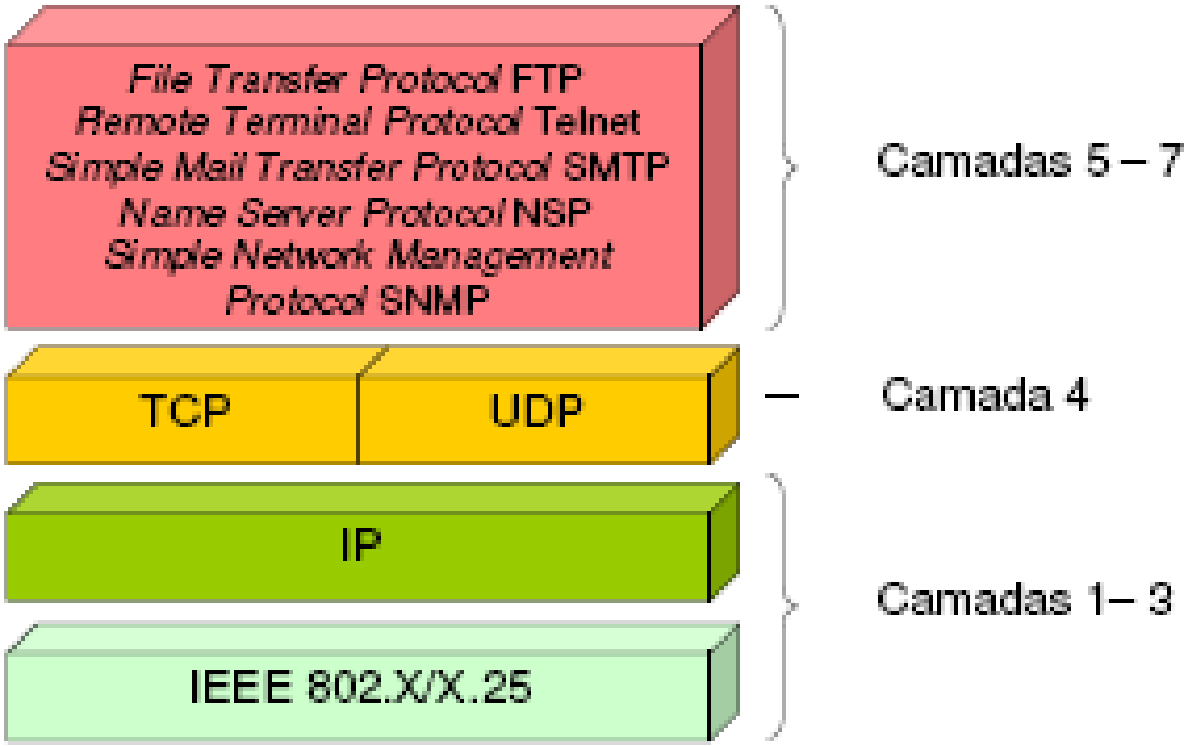
Nível de Apresentação

- Utilizado para reagrupar e estruturar informações e interpretá-las de maneira mais fácil pelas aplicações
- Interpretam grupos de bits como strings, inteiros, etc (conversão para formatos big endian, little endian, etc)

Nível de Aplicação

- Composto por um conjunto de protocolos de mais alto nível
  - Comandos
  - Interpretação de comandos
- Exemplos:
  - Apache
  - Servidores de e-mail

Protocolos TCP/IP



Protocolos em Camadas

- Cada camada adicional agrega cabeçalhos, os quais causam overhead no empacotamento e desempacotamento das mensagens
  - Em redes de longa distância, isso é pouco notado
  - Porém, nos SDs, que normalmente são implementados em LANs, a utilização de protocolos em camadas frequentemente provoca sobrecarga no sistema.
  - Desse modo, não são utilizados protocolos em camadas como no modelo OSI, ou no máximo trabalham com algumas camadas
    - Solução: modelo cliente/servidor

Modelo Cliente-Servidor

- Idéia: estruturar o SO como um grupo de processos que cooperam entre si, denominados *servidores* e *clientes*
- Os servidores oferecem serviços e os clientes utilizam esses serviços
- Uma máquina pode executar
  - Um único processo cliente ou servidor,
  - Múltiplos clientes
  - Múltiplos servidores
  - Combinação das anteriores

Modelo Cliente-Servidor

- Para evitar overhead das conexões orientadas a protocolos (OSI ou TCP/IP), o modelo cliente-servidor é baseado em um simples protocolos denominado *request/reply*

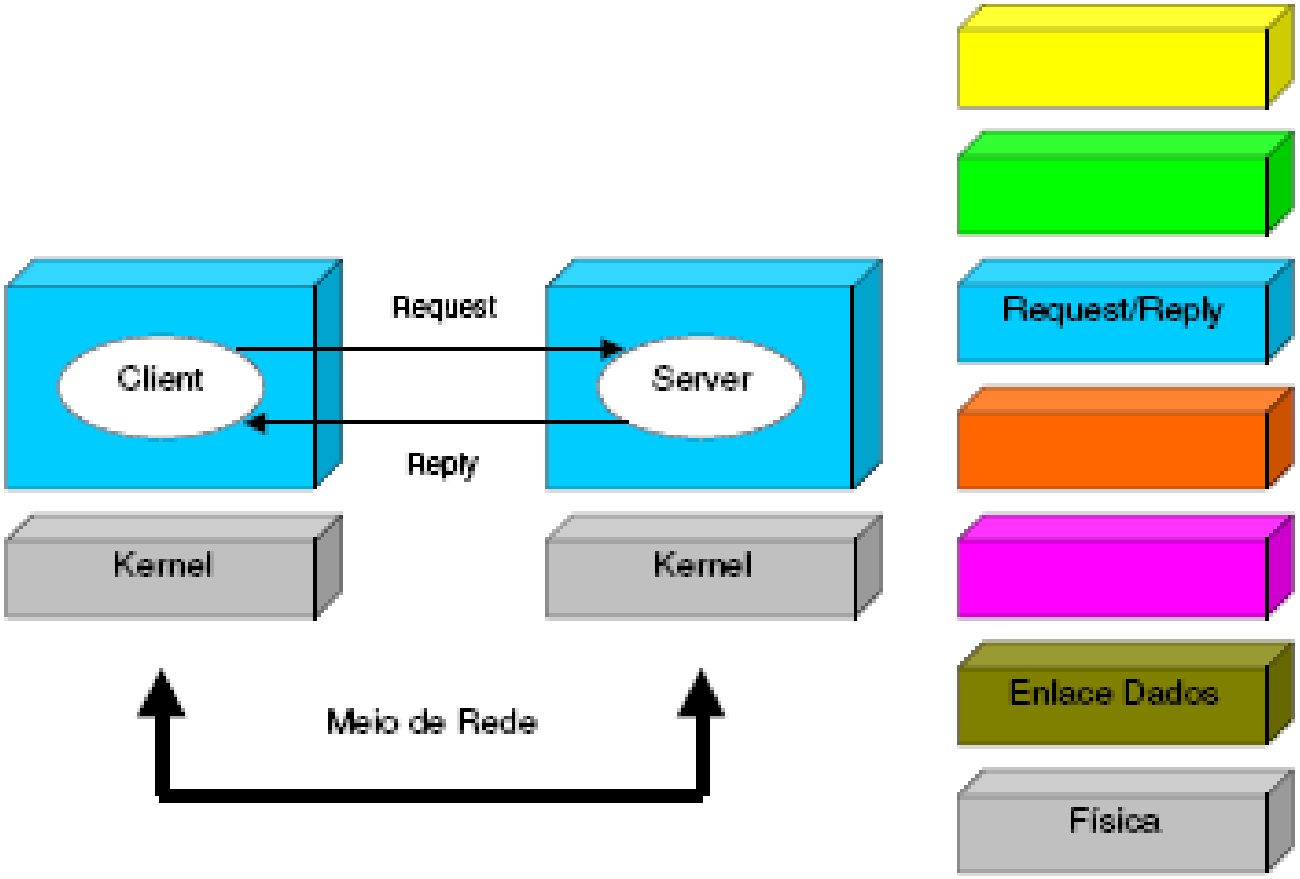
Modelo Cliente-Servidor

- Vantagens
  - Simplicidade;
  - Considerando rede local sem perda de informações, não tem necessidade de estabelecer uma conexão (e finalizar a conexão)
  - A mensagem de resposta serve de *acknowledgement*;
  - Eficiência;

Modelo Cliente-Servidor

- Se as máquinas forem idênticas, são necessários somente três níveis de protocolo
  - Físico e Enlace de dados
    - Responsáveis pela transmissão dos dados
  - Request / Reply
    - Define o conjunto de requisições válidas e o conjunto de respostas para essas requisições
- Não há a necessidade de conexão, portanto as camadas 3 e 4 não são necessárias
- Não existe gerenciamento de sessões, pois não há sessões. Também não são necessárias as camadas de apresentação e aplicação

Modelo Cliente-Servidor



Modelo Cliente-Servidor

- 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;
  - *receive (endereço, &ptm sg)*
    - bloqueia o processo que executou o *receive* até que uma mensagem chegue. Quando ela chega é copiada para o buffer apontad o por *ptmsg*. O parâmetro *endereço* específica o endereço no qual receptor está esperando uma mensagem.

Modelo Cliente-Servidor
Exemplo – arquivo *header.h*

```
/ * Definições dos Clientes e Serv idores */
#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

/ * Definições do formato de mens agem */
struct mes sage {
long source;
long des t ;
long opcode;
long count ;
long offset ;
long extra1;
long extra2;
long result ;
char name [MAX_PATH];
char data[BUF_SIZE];
};
```

Modelo Cliente-Servidor
Servidor

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

Modelo Cliente-Servidor

Exemplo de um cliente usando um servidor para cópia de arquivo

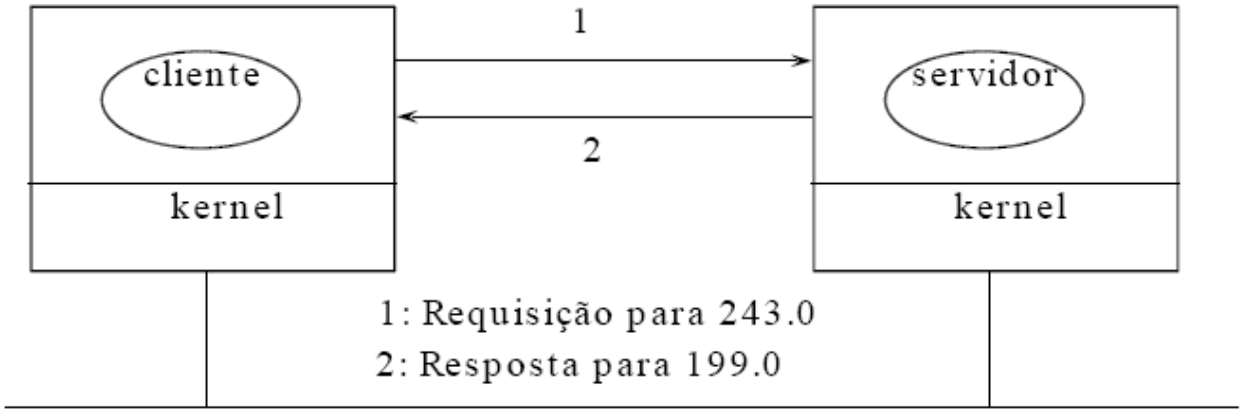
```
#include <header.h>
int cop y(char *src, char *dst )
{
        struct message m1;
        long position, client = 110;
        initialize();
        position = 0;
        do {
                / * Pega um bloco de dado do arquivo fonte */
                m1.opcode = READ;
                m1.offset = position;
                m1.count = BUF_SIZE;
                strcpy(&m1.name, src);
                send (FILE_SERVER, &m1);
                receive (client , &m1);
                / * Escreve o bloco de dado recebido no arquivo destino */
                m1.opcod e = 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

- O endereço do exemplo anterior foi definido como uma constante. Mas o que significa esse número? É referente ao processo ou à máquina?
- Se for máquina, pode-se extrair esse número e utilizá-lo para o envio da mensagem de resposta. Isso funciona se existe somente um processo sendo executado na máquina destino. Mas e se existe mais do que um processo sendo executado, para qual a mensagem deve ser encaminhada?
- O endereço pode representar o processo, não mais as máquinas.
- Problema: como os processos são identificados?

Endereçamento

- Endereço no formato: máquina.processo
- Exemplo: 243.0 ou 0@243
- Problema: não é transparente
  - Se um servidor não estiver disponível, tem que recompilar com o endereço de um novo servidor



Endereçamento

- Cada processo possui um endereço único
  - Processo centralizado mantendo um contador e cria o número dos processos
    - Desvantagem: componente centralizador
  - Póssível solução: cada processo associa seu próprio endereço a partir de uma espaço grande de possibilidades, como por exemplo, um número inteiro binário de 64 bits (endereçamento aleatório)

Endereçamento

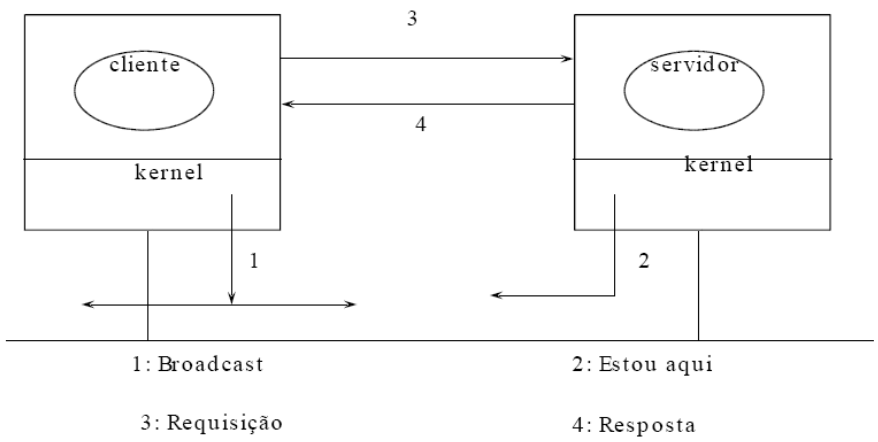
- Endereçamento aleatório
  - Probabilidade de 2 possuirem o mesmo número é pequena
  - Problema: como descobrir para que máquina enviar a requisição? (só lembrando que o processo que vai enviar a requisição tem que saber o número do processo servidor)
    - Solução: *Broadcasting*

para descobrir em

que máquina

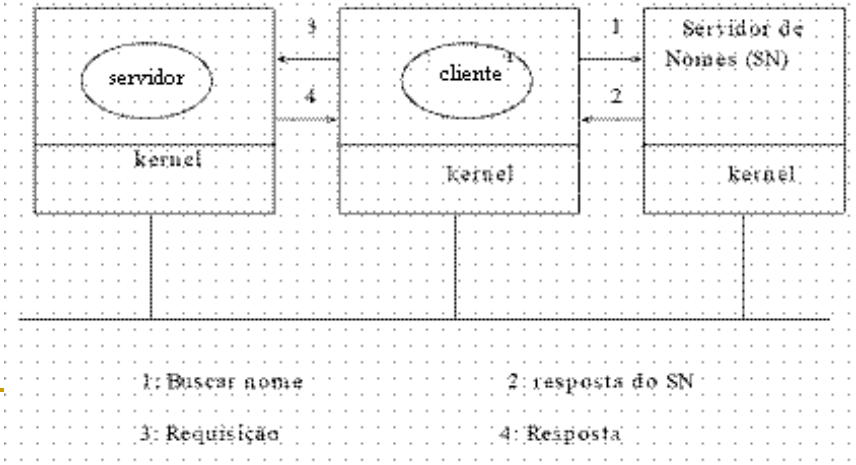
está o processo

servidor



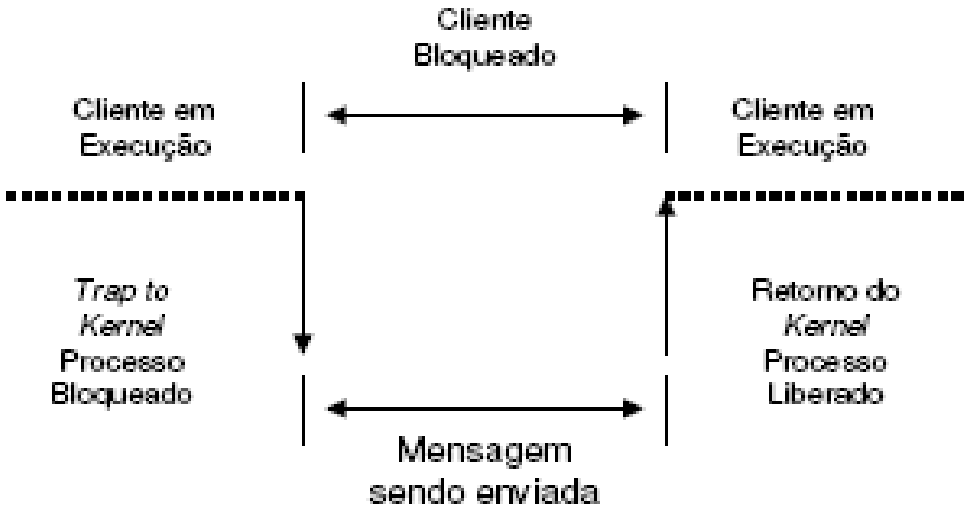
Endereçamento

- Servidor de Nomes
  - Problema da solução anterior: sobrecarga no sistema com o *broadcasting*
  - Solução: centralizar numa única máquina os endereços dos processos servidores, denominada Servidor de Nomes



Primitivas bloqueantes x Primitivas não bloqueantes

- As primitivas apresentadas até o momento são bloqueantes (*send* e *receive*)
- Enquanto a mensagem está sendo enviada ou recebida, o processo permanece bloqueado (suspenso)
- Mensagem é copiada para um local denominado *message buffer* e posteriormente esse *buffer* é enviado

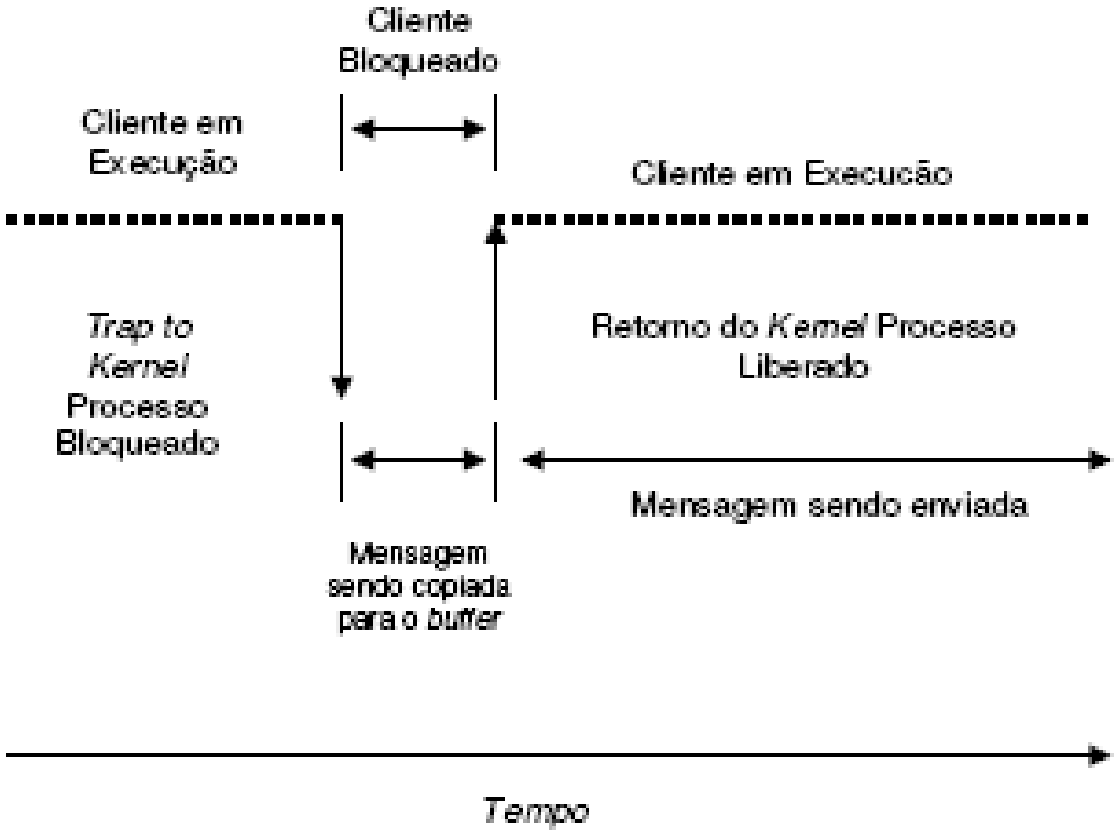


Primitivas bloqueantes x Primitivas não bloqueantes

- Primitivas não bloqueantes (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;
  - O processo que executa o send não pode modificar o buffer de mensagem até a mensagem ter sido enviada;
  - Problema: processo não tem idéia de quando o buffer estará livre;

Primitivas bloqueantes x Primitivas não bloqueantes

- Primitivas não bloqueantes (assíncronas)



Primitivas bloqueantes x Primitivas não bloqueantes

- Há dois modos para o processo ficar sabendo que a mensagem já foi enviada:
  - *Send* com cópia
    - É feita uma cópia dos dados da área do usuário para o espaço do kernel. Nesse período o processo fica bloqueado. Logo após o processo volta e pode utilizar seu buffer (no espaço do usuário)
    - Desperdício do tempo de CPU com a cópia extra;
  - *Send* com interrupção
    - Programação muito mais difícil
    - Enviar uma interrupção para o processo quando a mensagem tiver sido enviada (como sinal)

Primitivas bloqueantes x Primitivas não bloqueantes

- A escolha do tipo de primitiva a ser usada (bloqueadante ou não bloqueadante) é feita pelo projetista do sistema
  - normalmente é usado um dos dois tipos, em poucos casos tem-se os dois tipos disponíveis;

Primitivas bloqueantes x Primitivas não bloqueantes

- **Primitiva *Receive* não Bloqueante**
  - 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
      - Primitiva usada quando o receptor ficar totalmente bloqueado esperando o término da operação quando necessário
    - *Receive* condicional
      - Implementado com chamadas que testam periodicamente se buffer está cheio.
      - Caso não esteja pode executar outras operações e voltar mais tarde
    - Interrupção
      - Quando buffer estiver cheio, o processo recebe uma interrupção para “pegar” os dados

Primitivas bloqueantes x Primitivas não bloqueantes

- Timeout
  - Nas primitivas bloqueantes 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

Primitivas bloqueantes x Primitivas não bloqueantes

- Alguns sistemas diferem primitivas bloqueantes de síncronas
- Nesse caso, a primitiva bloqueante garante que a mensagem foi enviada pela rede, mas não é necessário que uma função *receive* esteja esperando por essa mensagem
- Para ser uma primitiva síncrona, o processo é desbloqueado somente quando se garante que a mensagem foi enviada e que foi recebida corretamente pela função *receive* no outro processo

Primitiva Bufferizada x Primitiva Não-Bufferizada

- As primitivas descritas até agora são não-bufferizadas. Como o kernel saberá onde colocar uma mensagem recebida se o processo ainda não realizou uma chamada a função *receive*? Não sabe!
- Solução: Descartar a mensagem e esperar o timeout expirar para o cliente enviar novamente a mensagem. Contudo após alguns reenvios cliente pode desistir !!!!

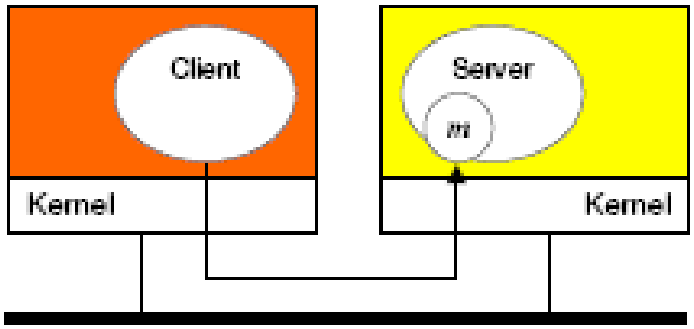
Primitiva Bufferizada x Primitiva Não-Bufferizada

- **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

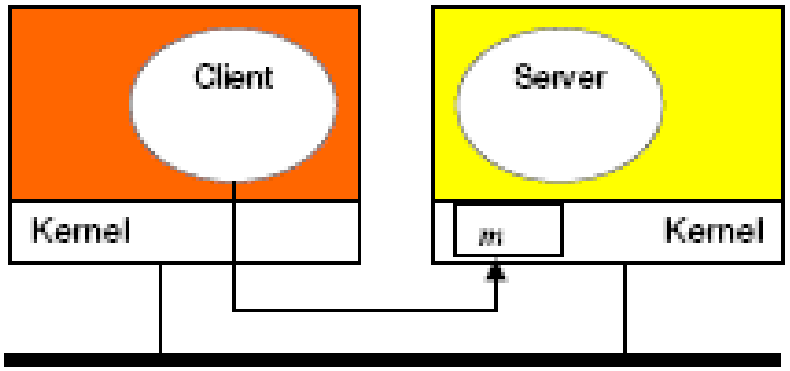
Primitiva Bufferizada x Primitiva Não-Bufferizada

- 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.
  - Problema: os *mailbox* são finitos. Desse modo, quando o *mailbox* estiver cheio, o kernel deve novamente decidir o que fazer com a mensagem

Primitiva Bufferizada x Primitiva Não-Bufferizada



Unbuffered



Buffered

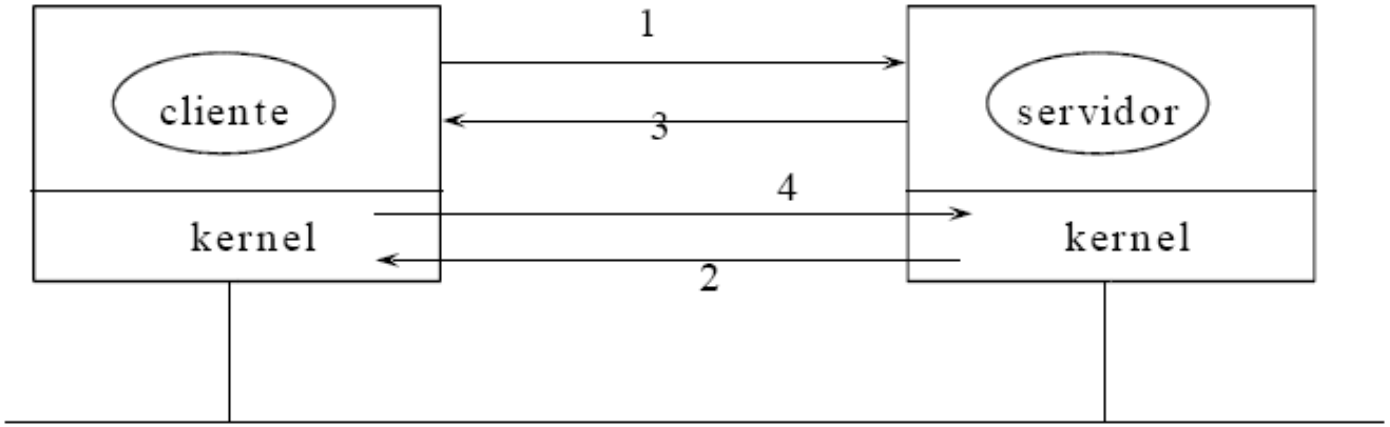
Primitivas Confiáveis x Primitivas não Confiáveis

- 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 bloqueante 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.

Primitivas Confiáveis x Primitivas não Confiáveis

- Três abordagens para o problema:
  1. Redefinir a semântica do send para este ser não confiável
     - O sistema não dá garantias sobre mensagens sendo enviadas, isto é, tornar a comunicação confiável é tarefa do usuário;
  2. 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

Primitivas Confiáveis x Primitivas não Confiáveis

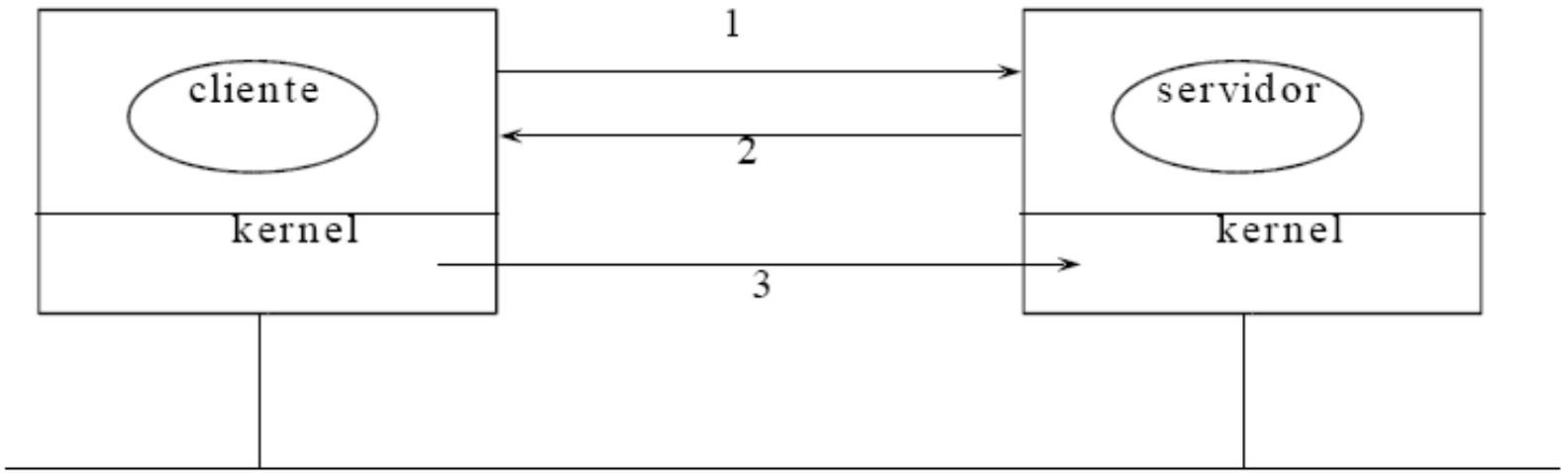


1: Requisição (Cliente para Servidor)
2: ACK (Kernel para Kernel)
3: Resposta (Servidor para Cliente)
4: ACK (Kernel para Kernel)

Primitivas Confiáveis x Primitivas não Confiáveis

- Três abordagens para o problema:
  1. 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 do cliente pode reenviar a requisição, se precavendo contra a perda de mensagem

Primitivas Confiáveis x Primitivas não Confiáveis



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

Implementação do modelo Cliente - Servidor

- Detalhes de como a passagem de mensagem é implementada dependem 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 seqüência

Implementação do modelo Cliente - Servidor

- 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

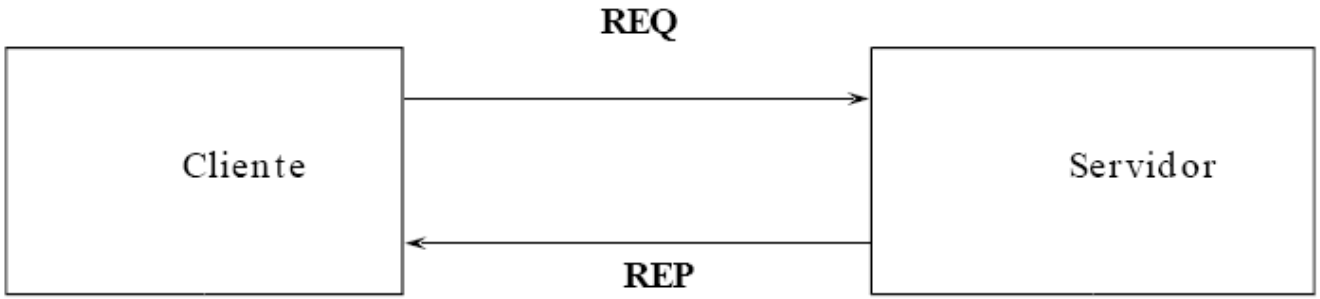
Implementação do modelo Cliente - Servidor

- 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

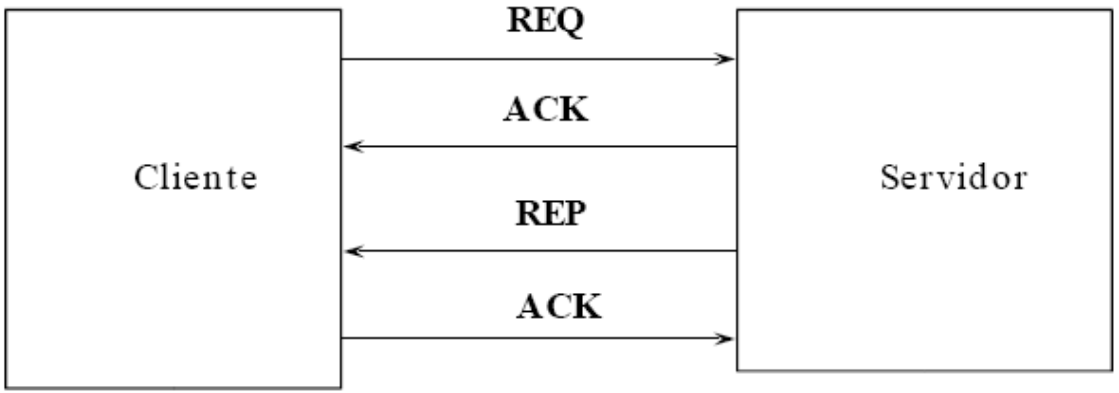
Implementação do modelo Cliente - Servidor

- Protocolo de Comunicação Cliente – Servidor:
  - Requisição, Resposta



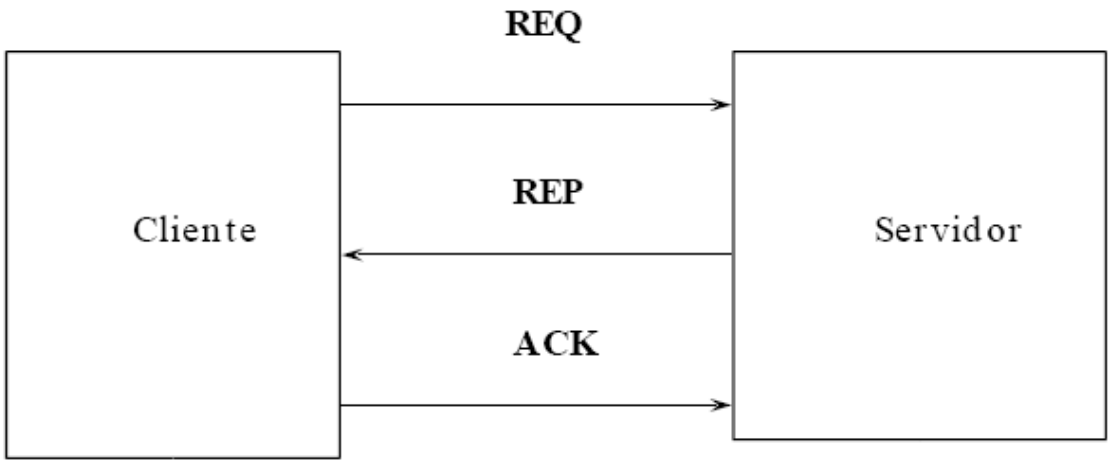
Implementação do modelo Cliente - Servidor

- Protocolo de Comunicação Cliente – Servidor:
  - Requisição, ACK, Resposta, ACK



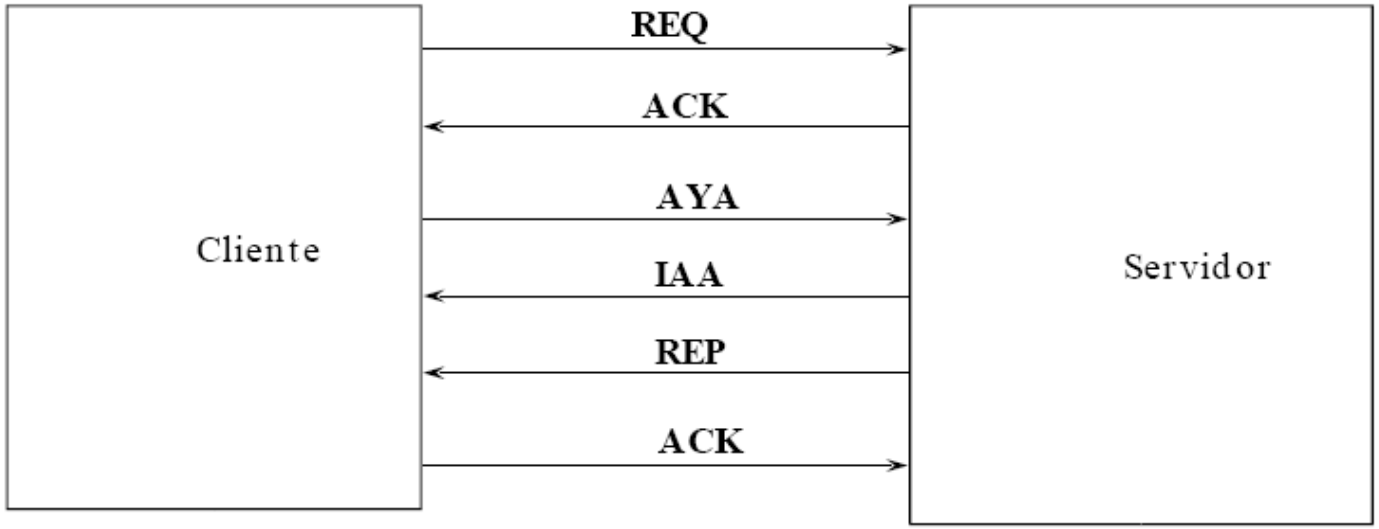
Implementação do modelo Cliente - Servidor

- Protocolo de Comunicação Cliente – Servidor:
  - Requisição, Resposta, ACK



Implementação do modelo Cliente - Servidor

- Protocolo de Comunicação Cliente – Servidor:
  - Verificando se o servidor ainda está “vivo”



SSC150 – Sistemas Computacionais Distribuídos

Comunicação em Sistemas Distribuídos
Sockets

Profa. Sarita Mazzini Bruschi

sarita@icmc.usp.br

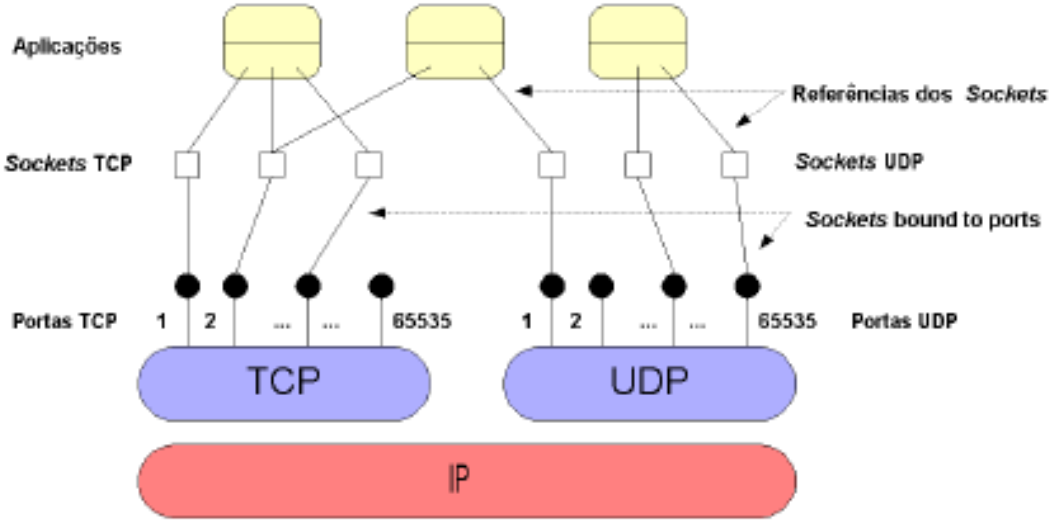
Slides baseados no material de:
Prof. Edmilson Marmo Moreira (UNIFEI / IESTI)

Sockets
Introdução

- Uma rede de computadores é composta por máquinas interconectadas e canais de comunicação que permitem a transmissão de bytes de uma máquina para outra
- No TCP/IP, para um programa comunicar com outro, ele utiliza duas informações:
  - Endereço Internet: usado pelo protocolo IP
  - Número de porta: utilizado pelo protocolo de transporte (UDP ou TCP)

Sockets
Definição

- Um **socket** é uma abstração através da qual uma aplicação pode enviar e receber dados, de uma maneira semelhante à utilizada para ler e escrever em arquivos



- Uma abstração socket pode ser referenciada por diversos programas usuários

Sockets usando Java

Sockets
Conceitos Básicos

- Um cliente deve utilizar o endereço IP do servidor para iniciar uma comunicação
- Em Java, o endereço pode ser especificado usando uma *string* que contém o endereço IP ou o nome correspondente
- O endereço IP é encapsulado na classe **`InetAddress`** que provê 3 métodos estáticos:
  - `getByName`
  - `getAllByName`
  - `getHostAddress`

Sockets
Exemplo

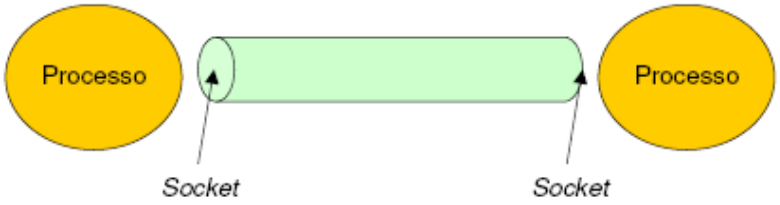
```
import java.net.*; // for InetAddress

class InetAddressExample {
    public static void main (String[] args) {
    // Get name and IP address of the local host
    try {
          InetAddress address = InetAddress.getLocalHost();
          System.out.println("Local Host:");
          System.out.println("\t" + address.getHostName());
          System.out.println("\t" + address.getHostAddress());
          } catch (UnknownHostException e) {System.out.println("Unable to determine this
    host's address");}

    for (int i=0; i < args.length; i++) {
          // Get name(s)/address(es) of hosts given on command line
          try {
                    InetAddress[] addressList =   InetAddress.getAllByName(args[i]);
                    System.out.println(args[i] + ":");
                    System.out.println("\t" + addressList[0].getHostName());
                    for (int j=0; j < addressList.length; j++)
                              System.out.println ("\t" +
                              addressList[j].getHostAddress());
                    } catch (UnknownHostException e) {System.out.println("Unable to find
    address for" + args[i]);}
          }
    }
}
```

Sockets TCP

- Java fornece duas classes para trabalhar com o TCP:
  - `Socket`
  - `ServerSocket`
- Uma instância de `Socket` representa uma conexão de TCP fim-a-fim
  - Um canal cujas pontas são indentificadas por um endereço IP e um número de porta



Sockets TCP

- Período de Setup
  - Período que se inicia com o cliente enviando uma requisição de conexão para o servidor
- Uma instância de `ServerSocket` aguarda conexões TCP e cria uma nova instância de `Socket` para manipular os pedidos de conexão

Sockets TCP

- Cliente TCP
  - Os clientes iniciam a comunicação com um servidor que está passivamente aguardando por uma conexão
  - Um cliente típico segue os seguintes passos:
    - Constrói uma instância de `Socket`: o construtor estabelece uma conexão TCP para um host e uma porta especificada;
    - Comunica-se utilizando o *socket's I/O stream*: uma instância de `Socket` contém objetos `InputStream` e `OutputStream` que pode ser utilizado como qualquer outro *Java I/O stream*
    - Encerra a conexão utilizando o método `close()` da classe `Socket`

Sockets
TCP – Exemplo - Cliente

```
import java.net.*; // for Socket
import java.io.*; // for IOException and Input/OutputStream
public class TCPEchoClient {
    public static void main (String[] args) throws IOException {
           if ((args.length < 2) || (args.length > 3))
                      throw new IllegalArgumentException ("Parameter(s): <Server> <Word> [<Port>]");
           String server = args[0];
           // Convert input String to bytes using the default character
           // encoding
           byte[] byteBuffer = args[1].getBytes();
           int servPort=(args.length == 3) ? Integer.parseInt(args[2]) :7;
           // Create socket that is connected to server on specified port
           Socket socket = new Socket(server, servPort);
           System.out.println("Connected to server... sending echo string");
           InputStream in = socket.getInputStream();
           OutputStream out = socket.getOutputStream();
           out.write(byteBuffer); // Send the encoded string to the server
           // Receive the same string back from the server
           int totalBytesRcvd = 0;
           int bytesRcvd;
           while (totalBytesRcvd < byteBuffer.length) {
              if ((bytesRcvd = in.read(byteBuffer, totalBytesRcvd, byteBuffer.length - totalBytesRcvd)) == -1)
                      throw new SocketException("Connection closed prematurely");
              totalBytesRcvd += bytesRcvd;
           }
           System.out.println("Received: " + new String(byteBuffer));
           socket.close();
    }
}
```

Sockets TCP

Servidor TCP

- A tarefa do servidor é preparar o canal de chegada dos pedidos e aguardar por conexões dos clientes
- Um servidor típico segue os seguintes passos:
  - Cria uma instância local da classe `ServerSocket`, especificando uma porta local. Esse objeto fica ouvindo a porta, aguardando por pedidos de conexão
  - Repetidamente
    - Chama o método `accetp()` do `ServerSocket` para obter a próxima conexão dos clientes. Quanto um novo cliente se conecta, uma nova instância da classe `Socket` é criada e retornada pelo método `accept()`
    - Comunica-se com o cliente usando os objetos `InputStream` e `OutputStream` do objeto `Socket`
    - Encerra a conexão com o cliente usando o método `close()` da classe `Socket`

Sockets
TCP – Exemplo - Servidor

```
import java.net.*; // for Socket, ServerSocket
import java.io.*; // for IOException and Input/OutputStream
public class TCPEchoServer {
    private static final int BUFSIZE = 32; // Size of receive buffer
    public static void main (String[] args) throws IOException {
    if (args.length != 1)
           throw new IllegalArgumentException("Parameter(s): <Port>");
    int servPort = Integer.parseInt(args[0]);
    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(servPort);
    int recvMsgSize; // Size of receive message
    byte[] byteBuffer = new byte[BUFSIZE]; // Receive buffer
    for (;;) { // Run forever, accpeting and servicing connections
           Socket clntSock = servSock.accept(); // Get client connection
           System.out.println("Handling client at " + clntSock.getInetAddress().getHostAddress()
    + " on port " + clntSock.getPort());
           InputStream in = clntSock.getInputStream();
           OutputStream out = clntSock.getOutputStream();
           // Receive until client closes connection,
           // indicated by -1 return
           while ((recvMsgSize = in.read(byteBuffer)) != -1)
                      out.write(byteBuffer, 0, recvMsgSize);
           clntSock.close(); // Close de socket.
           // We are done with this client
    }
    /* NOT REACHED */
    }
}
```

Sockets
UDP

- O socket UDP provê um serviço diferente de comunicação em relação ao protocolo TCP
- O UDP executa somente duas funções:
  - Adiciona outra camada de endereçamento (portas) para o IP;
  - Detecta dados corrompidos que podem ocorrer na transmissão das mensagens

Sockets
UDP

- Características
  - Não precisam ser conectados antes de serem utilizados
  - Cada mensagem carrega as próprias informações de endereçamento, sendo independente das outras mensagens
  - Durante o recebimento, funciona como uma caixa de correio onde cartas e pacotes de diferentes lugares podem ser colocados
  - Assim que é criado, um socket UDP pode ser usado para enviar e receber mensagens para qualquer endereço
  - Não há garantias de quem uma mensagem enviada por um socket UPD chegará ao seu destino
  - As mensagens podem ser entregues fora de ordem

Sockets
UDP

- Java fornece duas classes para trabalhar com UDP:
  - `DatagramPacket`
  - `DatagramSocket`
- Clientes e servidores utilizam DatagramSockets para enviar e receber DatagramPackets

Sockets
UDP

- `DatagramPacket`
  - Ao invés de enviar streams, como o TCP, o UDP envia mensagens autocontidas, denominadas datagramas, que são representadas em Java como instâncias `DatagramPacket`
  - Para enviar uma mensagem, um programa Java constrói uma instância de `DatagramPacket` e passa como argumento para o método `send()` da classe `DatagramSocket`

Sockets
UDP

- `DatagramPacket`
  - Para receber uma mensagem, um programa Java cria uma instância de `DatagramPacket` com espaço pré-alocado de memória (objeto `byte[]`), onde o conteúdo da mensagem recebida pode ser copiado e passa a respectiva instância para o método `receive()` da classe `DatagramSocket`

Sockets UDP

- `DatagramPacket`
  - Cada instância de `DatagramPacket` também contém informações de endereços e portas
  - Quando um `DatagramPacket` é enviado, o endereço e a porta identificam o destinatário
  - Quanto um `DatagramPacket` é recebido, o endereço identifica o remetente
  - Quando um `DatagramPacket` é recebido no servidor, este pode modificar seu conteúdo e enviar o mesmo `DatagramPacket`

Sockets
UDP

- Cliente UDP
  - Os clientes iniciam a comunicação enviando um datagrama para um servidor que está passivamente
  - Um cliente UDP típico segue os seguintes passos:
    - Constrói uma instância de `DatagramPacket`, opcionalmente indicando o endereço e a porta
    - Comunica-se através do envio e do recebimento de instâncias de `DatagramPacket` usando os métodos `send()` e `receive()` da classe `DatagramSocket`
    - Quanto finaliza, desaloca o socket usando o método `close()` da classe `DatagramSocket`

Sockets
UDP – Exemplo – Cliente

```
import java.net.*; // for DatagramSocket,DatagramPacket,InetAddress
import java.io.*; // for IOException
public class UDPEchoClientTimeout {
	private static final int TIMEOUT = 3000; // Resend timeout
	private static final int MAXTRIES = 5; // Maximum retransmissions
	public static void main (String[] args) throws IOException {
		if ((args.length < 2) || (args.length > 3))
				throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
		InetAddress serverAddress = InetAddress.getByName(args[0]);
		// Convert the argument String to bytes using the
		// default encoding
		byte[] bytesToSend = args[1].getBytes();
		int servPort = (args.length == 3) ? Integer.parseInt(args[2]):7;
		DatagramSocket socket = new DatagramSocket();
		socket.setSoTimeout(TIMEOUT); // Maximum receive blocking time
		DatagramPacket sendPacket = new DatagramPacket(bytesToSend, bytesToSend.length, serverAddress,
	servPort);
		DatagramPacket receivePacket = new DatagramPacket (new byte[bytesToSend.length],
	bytesToSend.length);
		int tries = 0;
		boolean receivedResponse = false;
```

Sockets
UDP – Exemplo – Cliente (continução)

```
                do {
                        socket.send(sendPacket); // Send the echo string
                        try {
                                socket.receive(receivePacket); // Attempt echo reply
                                // reception
                                // Check source
                                if (!receivePacket.getAddress().equals(serverAddress))
                                        throw new IOException("Received packet from an unknown source");
                                receivedResponse = true;
                        } catch (InterruptedIOException e) { // We didn't get anything
                                tries += 1;
                                System.out.println("Timed out, " + (MAXTRIES - tries) + " more tries ...");
                        }
                } while ((!receivedResponse) && (tries < MAXTRIES));
                if (receivedResponse)
                {
                        System.out.println("Received: " + new String(receivePacket.getData()));
                        System.out.println("Porta: " + (int)socket.getPort());
                }
                else
                        System.out.println("No response -- giving up.");
                socket.close();
        }
}
```

Sockets
UDP

- Servidor UDP
  - Como um servidor TCP, um servidor UDP é inicializado e aguarda passivamente por um cliente
  - Como não há a necessidade de se estabelecer conexão, a comunicação UDP é iniciada assim que chega um datagrama

Sockets UDP

- Servidor UDP
  - Um servidor UDP típico segue os seguintes passos:
    - Cria uma instância da classe `DatagramPacket`, especificando uma porta local e, opcionalmente, um endereço local. O servidor está pronto para receber datagramas de qualquer cliente
    - Recebe uma instância de `DatagramPacket` usando o método `receive()` da classe `DatagramPacket`. Quando `receive()` retorna, o datagrama contém o endereço do cliente. Desta forma, o servidor reconhece o destinatário da resposta
    - Comunica-se com os clientes pelo envio e recebimento de DatagramPackets usando os métodos `send()` e `receive()` da classe `DatagramSocket`
    - Quando finalizado, deloca-se o socket usando o método `close()` da classe `DatagramSocket`

Sockets
UDP – Exemplo – Servidor

```
import java.net.*; // for DatagramSocket and DatagramPacket
import java.io.*; // for IOException
public class UDPEchoServer {
	private static final int ECHOMAX = 255; // Maximum size of echo
	// datagram
	public static void main (String[] args) throws IOException {
		if (args.length != 1)
			throw new IllegalArgumentException("Parameter(s): <Port>");
		int servPort = Integer.parseInt(args[0]);
		DatagramSocket socket = new DatagramSocket(servPort);
		DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX],ECHOMAX);
		for (;;) { // Run forever, accpeting and echoing datagrams
			socket.receive(packet); // Receive packet from client
			System.out.println ("Handling client at " + packet.getAddress().getHostAddress() + "
	on port " + packet.getPort());
			System.out.println ("Message received " + new String(packet.getData()));
			socket.send(packet); // Send de same packet back to client
			packet.setLength(ECHOMAX);		// Reset length to avoid

		//		shrinking buffer
		}
		/* NOT REACHED */
	}
}
```

Sockets
Envio e recebimento de mensagens com *Sockets* UDP

- Uma diferença entre TCP e UDP é que o UDP preserva as últimas mensagens
  - A cada chamada do método `receive()` são retornados dados de pelo menos uma chamada `send()`
  - Além disso, diferentes chamadas a `receive()` nunca irão retornar dados de uma mesma chamada `send()`
  - Quando uma chamada ao método `write()` em um *stream* de saída TCP retorna, todos os chamadores sabem que os dados foram copiados para um *buffer* para transmissão, independente dos dados terem sido transmitidos
  - Entretando, o UDP não provê recuperação de erros na rede e, dessa forma, não existem *buffers* de dados para possíveis retransmissões

Sockets
Envio e recebimento de mensagens com *Sockets* UDP

- Entre o tempo de uma mensagem chegar pela rede e o tempo de seus dados serem retornados pelos métodos `read()` e `receive()`, os dados são armazenados em uma fila FIFO
- Com uma conexão TCP, todos os bytes recebidos, mas ainda não entregues, são tratados como uma seqüência contínua de bytes

Sockets
Envio e recebimento de mensagens com *Sockets* UDP

- No protocolo UDP, os dados podem ter sido originados de diferentes emissores. Um dado recebido de um socket UDP é mantido em uma fila de mensagens, cada uma com a informação associada que identifica sua origem
- Uma chamada a `receive()` nunca irá retornar mais do que uma mensagem
- Entretando, se `receive()` é chamada com um `DatagramPacket` contendo um buffer de tamanho n e o tamanho da primeira mensagem da fila de mensagens recebidas for maior do que n, somente os primeiros n bytes da mensagem são retornados. Os bytes restantes são descartados sem nenhuma indicação ao programa receptor de que a informação foi perdida

Sockets
Envio e recebimento de mensagens com *Sockets* UDP

- Por essa razão, um receptor deve sempre prover um `DatagramPacket` com um *buffer* grande o suficiente para manter a maior mensagem permitida pelo protocolo da aplicação durante uma chamada ao método `receive()`
- Essa técnica irá garantir que nenhum dado se perderá, entretanto, o tamanho máximo de um datagrama UDP é 65.507 bytes

Sockets
Envio e recebimento de mensagens com *Sockets* UDP

- É importante lembrar que cada instância da classe `DatagramPacket` não possui, internamente, noção do tamanho das mensagens, o qual pode ser alterado toda vez que uma mensagem é recebida.
- Aplicações que chamam o método `receive()` mais de uma vez, com a mesma instância de `DatagramPacket`,devem explicitamente reiniciar o tamanho interno do *buffer* atual antes de cada chamada ao método.