# Sistemas Distribuídos

Prof. Marcio Seiji Oyamada

# Objetivos

- Estudar as principais características de Sistemas Distribuídos
- Estudar os componentes necessários para construção de sistemas distribuídos na Internet.
- Estudar a intercomunicação entre processos através de sockets e chamadas remotas.

# Programa

- Introdução
- Hardware para sistemas distribuídos
- Software para sistemas distribuídos
- Comunicação via sockets
- Arquitetura Cliente-Servidor
- Protocolos da Internet
  - POP
  - HTTP
- Objetos distribuídos

# Bibliografia


- COULOURIS, G. et all. Distributed Systems: Concepts and Design. Ed. Addison-Wesley, 2001.
- TANENBAUM, A. Operating Distributed Systems. Ed. Addison-Wesley, 1995.
- RUSTY, H. E. Java: Network Programming,Ed. O'Erilly, 2a. Edição, 2000.
- ORFALI, D. et all. Client/Server Programming with Java and CORBA. Ed.John Wiley and Sons, 1998.
- JAVA SOCKETS TUTORIAL. Documentação Java.

# Sistema Distribuído

- Tanenbaum: “Um conjunto de máquinas independentes que fornecem uma visão de uma única máquina para os usuários”
- Coulouris: “Sistemas onde componentes de hardware e software localizados em rede, comunicam-se somente através de troca de mensagens”

# Sistemas Distribuídos

- Por que construir um Sistema Distribuído
  - Compartilhamento de recursos
  - Ambientes personalizados
  - Independência de localização
  - Pessoas e informação são distribuídas
  - Custo/desempenho
  - Modularidade
  - Expansão
  - Disponibilidade
  - Escalabilidade
  - Confiabilidade

# Sistemas Distribuídos

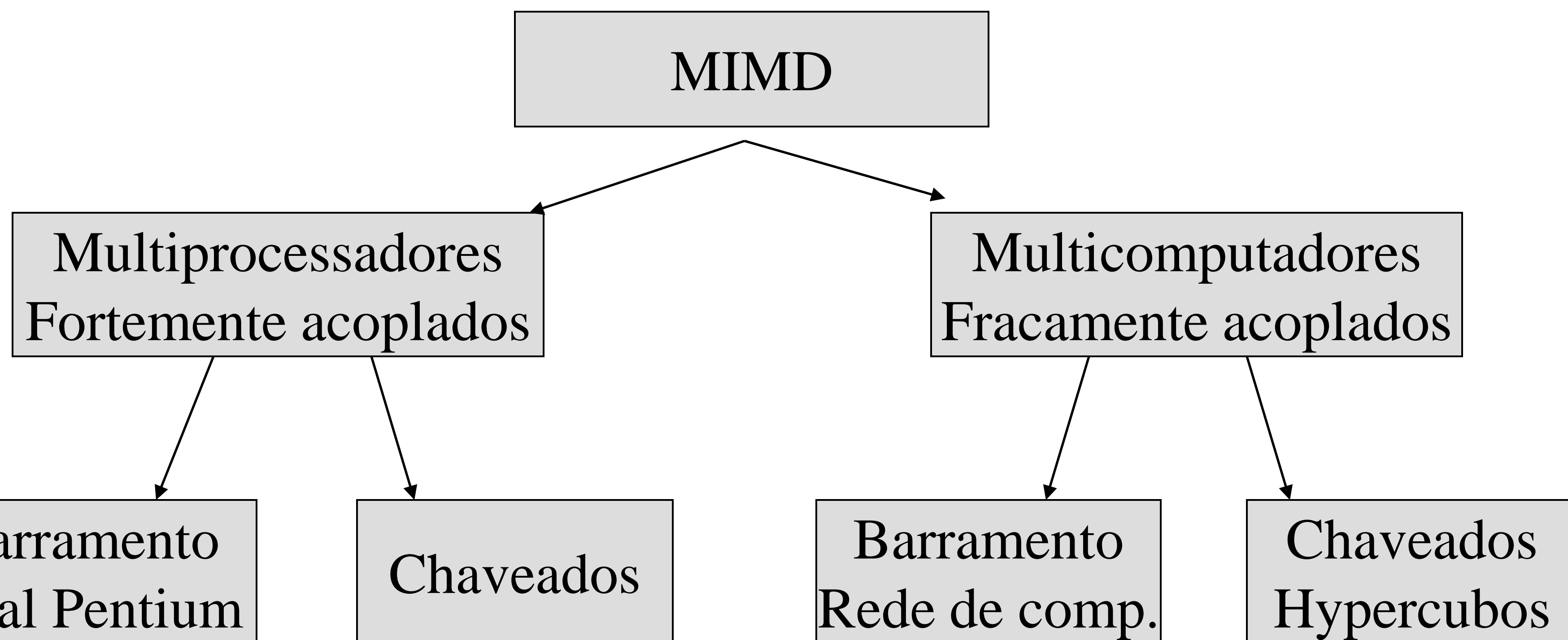
- Hardware
- Software

# Taxonomia do HW em SD

- Taxonomia de Flynn: classificação relacionada ao fluxo de dados e instruções.
  - SISD(Single Instruction Single Data): Computadores pessoais
  - SIMD(Single Instruction Multiple Data): computadores vetoriais
  - MISD (Multiple Instruction Single Data)
  - MIMD (Multiple Instruction Multiple Data): máquinas paralelas, sistemas distribuídos

# Taxonomia de HW

- Taxonomia de Flynn insuficiente para classificar todos os tipos de HW
- Classificação de Tanenbaum

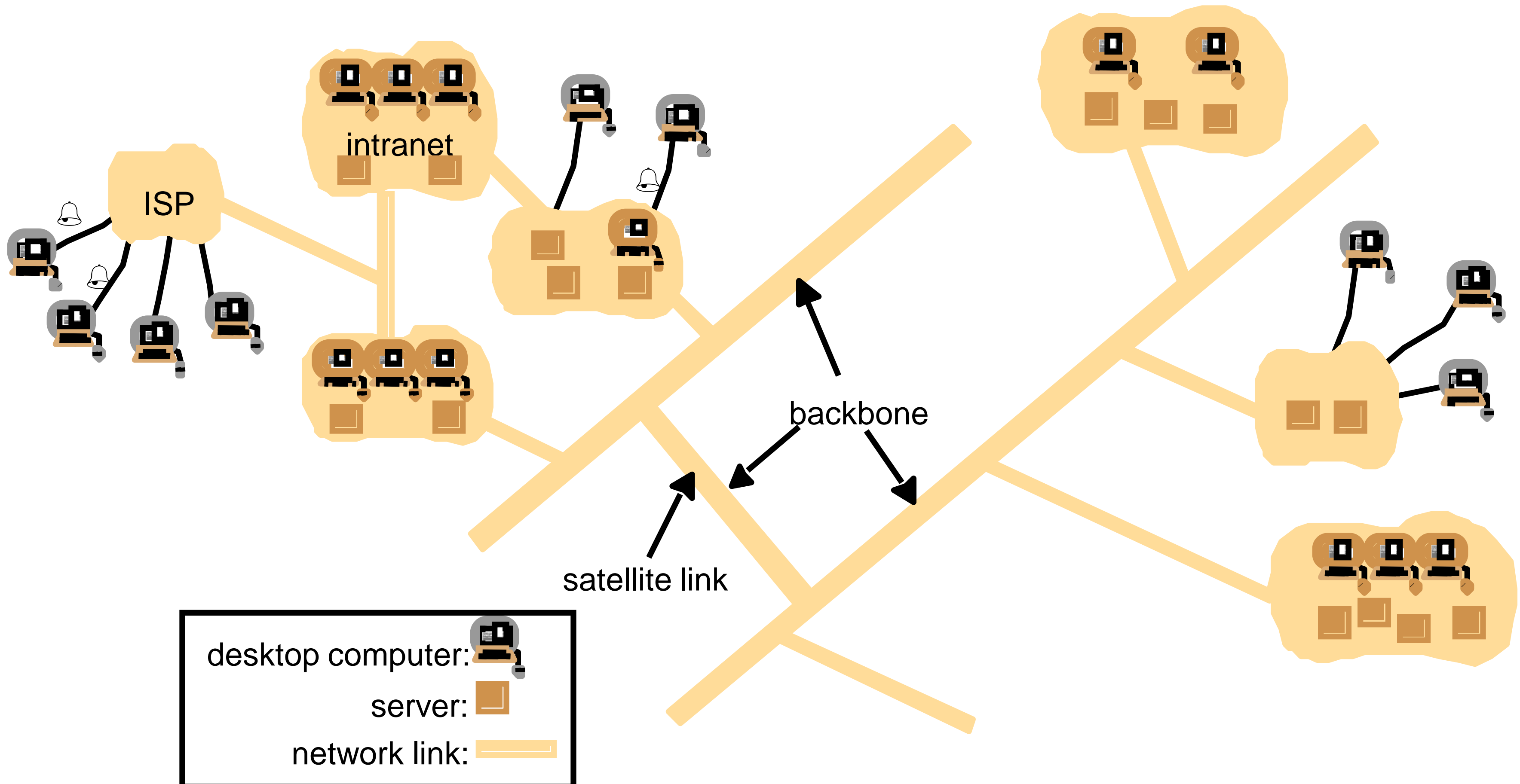


# Taxonomia de HW

- Barramentos
  - ethernet
  - grid
- Interconexão
  - Omega network
  - Hipercubos

# Sistemas Distribuídos

- Internet



# Sistemas Distribuídos

- Computadores x Servidores WEB

| *Date* | *Computers* | *Web servers* | *Percentage* |
|---|---|---|---|
| 1993, July | 1,776,000 | 130 | 0.008 |
| 1995, July | 6,642,000 | 23,500 | 0.4 |
| 1997, July | 19,540,000 | 1,203,096 | 6 |
| 1999, July | 56,218,000 | 6,598,697 | 12 |

Fonte: Coulouris, 2001

# Sistemas Distribuídos

- Desafios
  - Heterogeneidade
  - Flexibilidade
  - Segurança
  - Escalabilidade
  - Manipulação de falhas e confiabilidade
  - Concorrência
  - Transparência

# Heterogeneidade

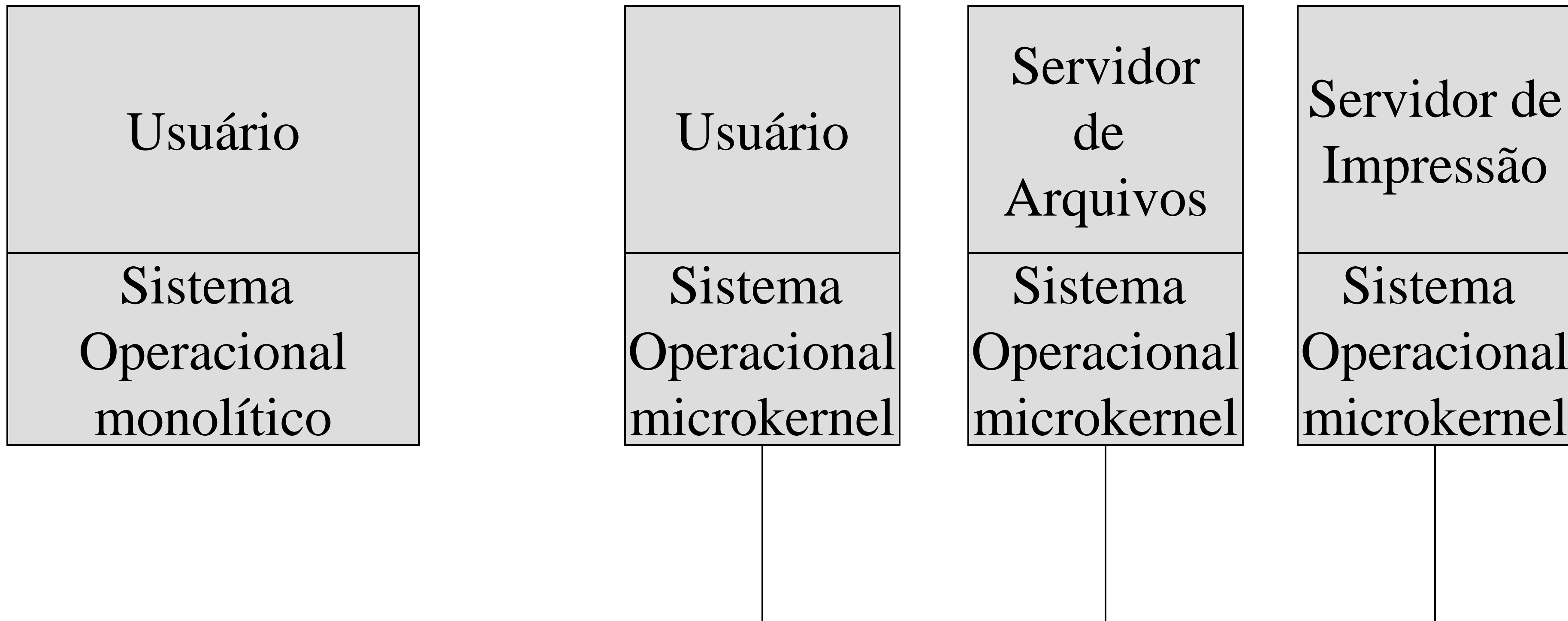
- Coleção de componentes diferentes acessados da mesma forma
- Hardware, Sistemas Operacionais, linguagens de programação
- Internet: todos os computadores compartilham protocolos comuns. (http, ftp entre outros)
- Muitos Sistemas Operacionais (SO) utilizam interfaces diferentes. Representação de dados diferentes em máquinas (big endian x litlle endian)

# Flexibilidade

- Capacidade do sistema ser estendido e reimplementado
- Utilização de interfaces padronizadas, para que novos serviços possam ser disponibilizados
- Um sistema desenvolvido por várias equipes de trabalho pode ser acoplado facilmente

# Flexibilidade

- Ex: Sistemas operacionais monolíticos x Micro kernel



# Segurança

- Informações críticas são enviadas pela rede
- Informação pode ser capturada por agentes indesejáveis
- Canais seguros: sockets (SSL), https, ssh (login remoto seguro)
- VPN (Virtual Private Network)

# Escalabilidade

- Uma solução deve suportar o crescimento do sistema
- Crescimento: número de máquinas e usuários
- Evitar soluções centralizadas
  - Serviços (ex: servidor de e-mail)
  - Tabelas
  - Algoritmos centralizados

# Confiabilidade

- Medida de qualidade de um serviço: as respostas são confiáveis. Ex: serviço de e-mail
- Disponibilidade: medida do tempo em que um serviço está disponível
- Ideal: altamente disponível e altamente confiável
- Replicação de serviços
  - Consistência: várias cópias de um mesmo dado armazenados em máquinas diferentes
  - Complexidade de gerenciamento x grau de confiabilidade

# Concorrência

- Recursos de um sistema distribuído sendo utilizado por vários usuários simultaneamente
- Problema: consistência dos dados??? (Impressora, arquivos)
- Soluções:
  - Bloqueio de dados ou recursos (Banco de dados, impressora)
  - Escalonamento de recurso (CPU)

# Transparência

- *Transparência de acesso:* recursos locais e remotos são acessados utilizando-se as mesmas operações
- *Transparência de localização:*recursos podem ser acessados sem conhecimento de sua localização
- *Transparência de concorrência:* vários processos podem operar concorrentemente usando recursos compartilhados sem interferência.
- *Transparência de replicação:* múltiplas instâncias de recursos podem ser utilizadas para aumentar o desempenho e confiabilidade, sem a necessidade de conhecimento por parte dos usuários e programadores

# Transparência

- *Transparência de falha:* sistemas tolerantes a falhas, onde os usuários podem terminar suas tarefas mesmo na presença de falhas de componentes de hardware e software.
- *Transparência de mobilidade:* permite que os recursos e clientes possam movimentar-se pelo sistema sem afetar sua operação
- *Transparência de escalabilidade:* permite a expansão do sistema e aplicações sem a mudança da estrutura do sistema e algoritmos da aplicação

# Sistemas Distribuídos

- Dificuldades
  - Concorrência
  - Sem relógio global
  - Estados inconsistentes
  - Falhas independentes
  - Escalabilidade

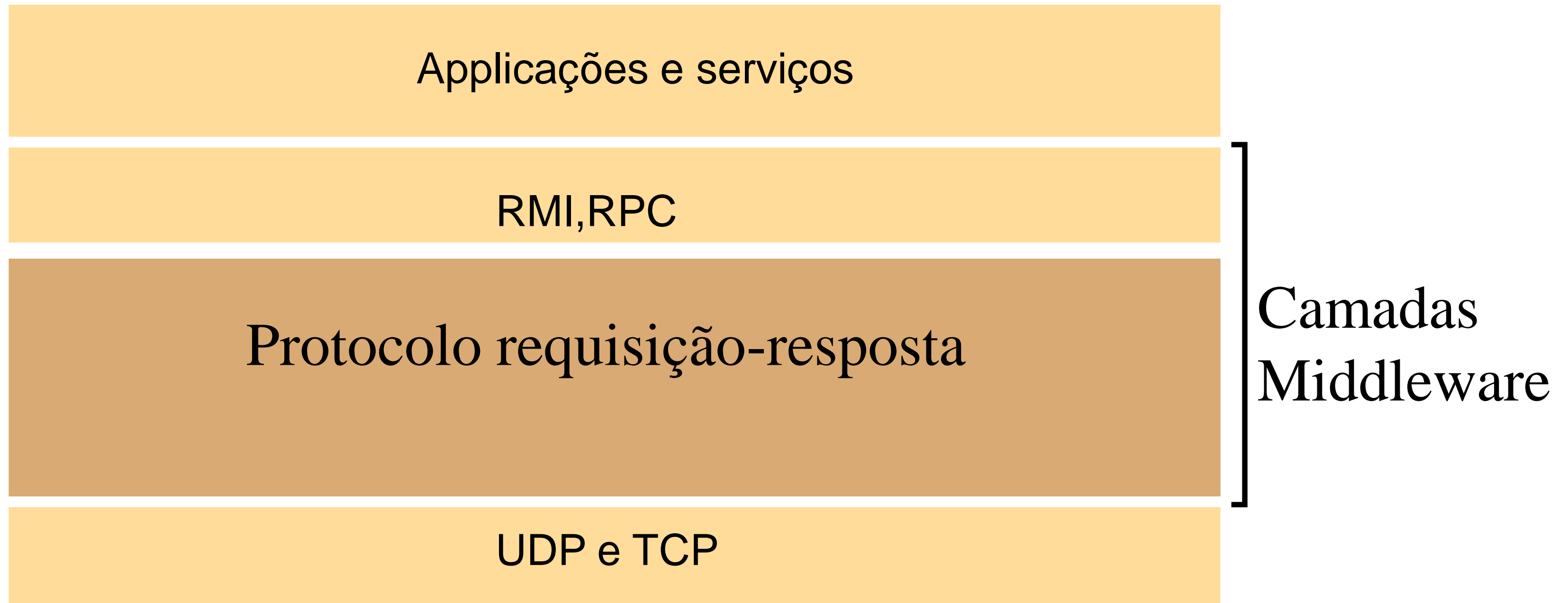
# Middleware

- Plataforma: arquitetura + Sistema Operacional
  - X86/Linux
  - Sparc/Linux
  - HP-PA/HP-UX
- Variedade de plataformas existentes no mercado
- Como desenvolver software para serem utilizados em todas as plataformas???

# Middleware

- API- Interface de programação de aplicações
  - Fornece um conjunto de operações para comunicação entre processos
  - Protocolo: regras para comunicação
  - Abstração de dados: forma canônica para troca de informações (arquiteturas diferentes)

# Middleware

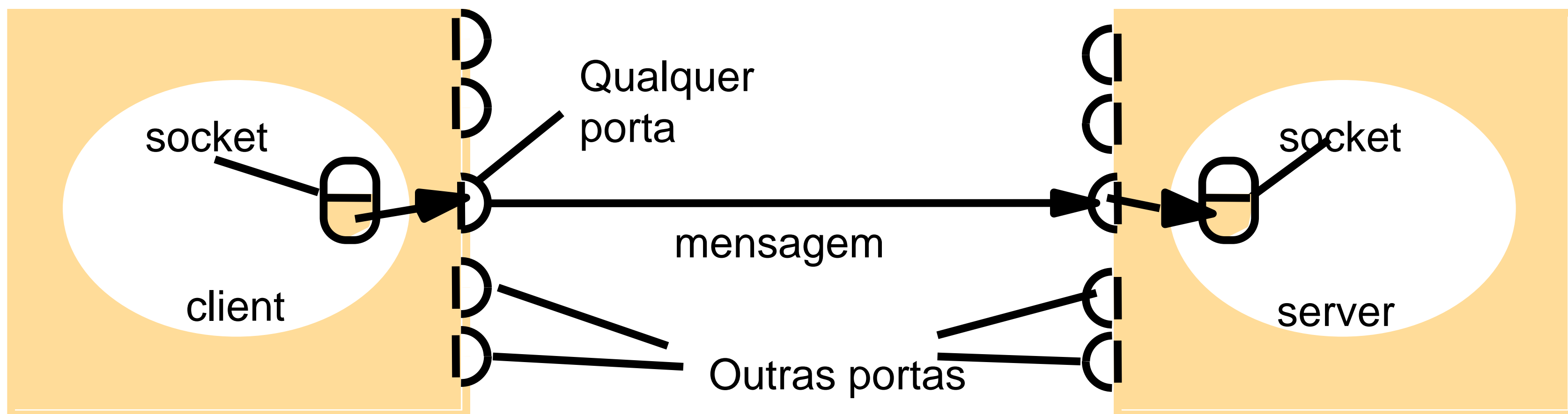


# Sockets

- Sockets fornecem um ponto de comunicação entre processos
- Origem: BSD Unix
  - Atualmente presente  na maioria dos sistemas operacionais
- Transmitir mensagens através de sockets abertos por processos em diferentes máquinas

# Sockets

- Números de portas disponíveis= $2^{16}$
- 2 processos não podem estar utilizando uma porta ao mesmo tempo
- Portas padrão: http = 80, ftp, telnet, ssh



# UDP

- UDP
- Características
  - Tamanho da mensagem: conjunto de bytes, sendo necessário indicar o tamanho. A maioria dos ambientes permite pacotes de tamanho máximo 8k bytes
  - Bloqueio: envio sem bloqueio/ recepção com bloqueio
  - Tempo de espera: a recepção de uma mensagem pode bloquear indefinidamente o processo
  - Recepção: um processo pode receber um datagrama de qualquer máquina, ou especificar uma única máquina no qual o socket receberá pacotes

# UDP

- Falhas
  - Mensagens podem ser perdidas pela origem ou destino (erro de *checksum*, espaço no *buffer*)
  - Ordenação: mensagens podem ser entregues fora da ordem de envio
- Utilização do UDP
  - Aplicações onde a perda eventual de uma mensagem é tolerada
  - DNS (Domain Name Service)
  - Baixo *overhead*

# UDP

- Java API
  - DatagramPacket: classe responsável pelo encapsulamento de mensagens UDP
    - Construtor: Mensagem (*array de bytes*), tamanho da mensagem, endereço da Internet, número da porta
    - Instâncias desta classe são enviadas pela rede

# UDP

- DatagramSocket: classe suporta *sockets*, sendo responsável pelo envio e recebimento de datagramas
  - Construtores: DatagramSocket(), DatagramSocket(2000)
  - send(DatagramPacket): envia um datagrama
  - receive(DatagramPacket): recebe um datagrama
  - setSoTimeout(tempo): define um *timeout* de espera no bloqueio do método *receive*.

# UDP: envio da mensagem

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
    // args give message contents and server hostname
    DatagramSocket aSocket = null;
      try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 7000;
            DatagramPacket request = new DatagramPacket(m,
    args[0].length(), aHost, serverPort);
            aSocket.send(request);
      }catch (SocketException e){System.out.println("Socket: " +
    e.getMessage());
     }catch (IOException e){System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
   }
}
```

# UDP: recepção de mensagens

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
    DatagramSocket aSocket = null;
       try{
          aSocket = new DatagramSocket(7000);
          byte[] buffer = new byte[1000];
          while(true){
             DatagramPacket request = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(request);
          }
       }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
       }finally {if(aSocket != null) aSocket.close();}
    }
}
```

# UDP: Exercícios

- Desenvolva um programa que abra 2 *sockets* em uma mesma porta. Qual o resultado?
- Desenvolva um programa que envie um datagrama para um destino onde não existe um processo "escutando" na porta. Qual o resultado?
- Desenvolva um programa que faça a leitura de dois números do teclado e envie para um servidor que receberá os dados e retornará a soma destes números.

# TCP

- Abstração de uma seqüência de *bytes* que podem ser escritos e lidos
  - Tamanho da mensagem: a aplicação escolhe o número de bytes que serão lidos. A camada TCP decide quando os bytes serão enviados. Normalmente os *sockets* fornecem uma forma para que os dados sejam enviados imediatamente
  - Perda de mensagens: um esquema utilizando mensagens de reconhecimento é utilizado
  - Controle de fluxo: caso o produtor/consumidor tenham velocidades diferentes, estes são sincronizados através do bloqueio das operações de leitura e escrita
  - Duplicação e ordenação de mensagens: utilização de identificadores para detectar e corrigir a recepção de mensagens
  - Destino das mensagens: um canal é estabelecido entre 2 processos que realizam leituras e escritas neste canal

# TCP

- Dados devem concordar com um protocolo. Ex: cliente escreve um número inteiro e um número real, o servidor deverá ler um número inteiro e um número real
- Bloqueio: o recebimento de mensagens fica bloqueado até que a quantidade de dados requisitada esteja disponível. O envio é bloqueado caso o *buffer* destino esteja cheio.
- *Threads*: é possível utilizar *threads* para comunicar-se com os clientes, para que as operações de leitura e escrita não bloqueiem outros clientes

# TCP

- A retransmissão e reordenação de mensagens é garantida pelo protocolo
- Não é possível distinguir entre falhas na rede ou no processo
- Não é possível garantir se uma mensagem foi entregue ou não durante uma falha
- Utilização: vários serviços da Internet como http, ftp, telnet utilizam TCP

# TCP

- ServerSocket: responsável por criar um *socket* e verificar por conexões dos clientes.
  - *accept:* verifica a fila de conexões, caso esteja vazia espera por um pedido de conexão. Quando uma conexão é estabelecida, um objeto *socket* é retornado representando a nova conexão entre o cliente e servidor

# TCP

```
import java.net.*;
import java.io.*;
public class TCPClient {
 public static void main (String args[]) {
  Socket s = null;
  try{
    int serverPort = 8000;
    s = new Socket(args[1], serverPort);
    DataInputStream in= new DataInputStream(s.getInputStream());
    DataOutputStream out= new DataOutputStream(s.getOutputStream());
    out.writeUTF(args[0]);
    String data = in.readUTF();
    System.out.println("Received: "+ data) ;
  }catch (UnknownHostException e){
        System.out.println("Sock:"+e.getMessage());
  }catch (EOFException e){
       System.out.println("EOF:"+e.getMessage());
  }catch (IOException e){
       System.out.println("IO:"+e.getMessage());}
  }finally {
       if(s!=null)
       try {
         s.close(); }
       catch (IOException e){
          System.out.println("close:"+e.getMessage());
       }
  }// try
 } // main
} // class
```

# TCP

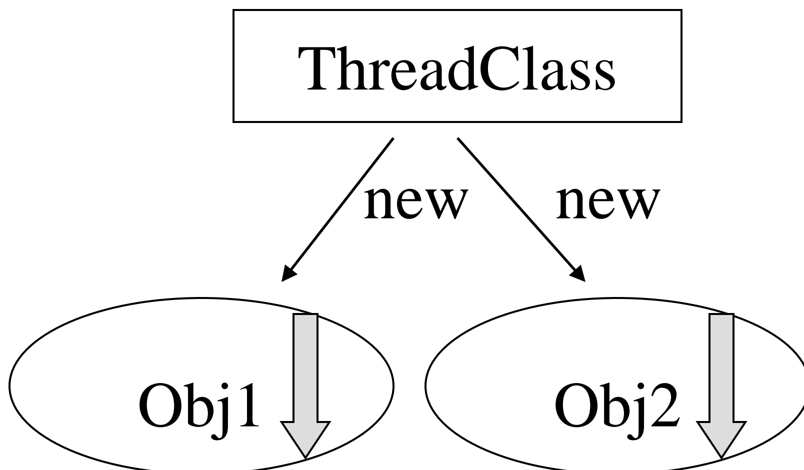
```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
   try{
     int serverPort = 8000;
     ServerSocket listenSocket = new ServerSocket(serverPort);
     Socket clientSocket = listenSocket.accept();
     in = new DataInputStream( clientSocket.getInputStream());
     out =new DataOutputStream( clientSocket.getOutputStream());
     String data = in.readUTF();
      System.out.println(data);
     out.writeUTF(data);
   } catch(EOFException e) {
     System.out.println("EOF:"+e.getMessage());
   } catch(IOException e) {
     System.out.println("IO:"+e.getMessage());
    } finally{
      try {
        clientSocket.close();
      }catch (IOException e){/*close failed*/}
    } // try
  } // main
} // class
```

# TCP: Exercícios

- Implemente um programa cliente que envie 2 inteiros para o servidor e este retorne a soma.

# TCP

- Servidores *multithread*: atender vários clientes ao mesmo tempo sem bloqueio
- Linguagem deve suportar *multihread*
- *Threads* em Java: criar uma classe que herda o comportamento da classe Thread



# Threads

```
class TestaThread{
   public static void main() {
       ClThread t1, t2;
       t1= new ClThread("Thread 1");
       t2= new ClThread("Thread 2");
       t1.join();
       t2.join();
   }
}

class ClThread extends Thread{
   String texto;
   public CLThread(String t){
      texto= t;
      start();
   }
   public void run(){
      while(1) {
         System.out.println(texto);
      }
   }
}
```

# TCP

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
     try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                        Socket clientSocket = listenSocket.accept();
                        Connection c = new Connection(clientSocket);
            }
     } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}
```

# TCP

```
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            start();
        } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {                                             // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
    }
}
```

# TCP: Exercícios

- Reimplemente o programa cliente/servidor para soma de 2 números inteiros, agora com suporte para múltiplo clientes

# Cliente/Servidor

- Requisição/resposta
  - Interface: Conjunto de operações no qual o servidor pode tratar
  - Servidor: recebe a requisição e executa, retornando um resultado
  - Cliente: envia a requisição, e espera pela resposta

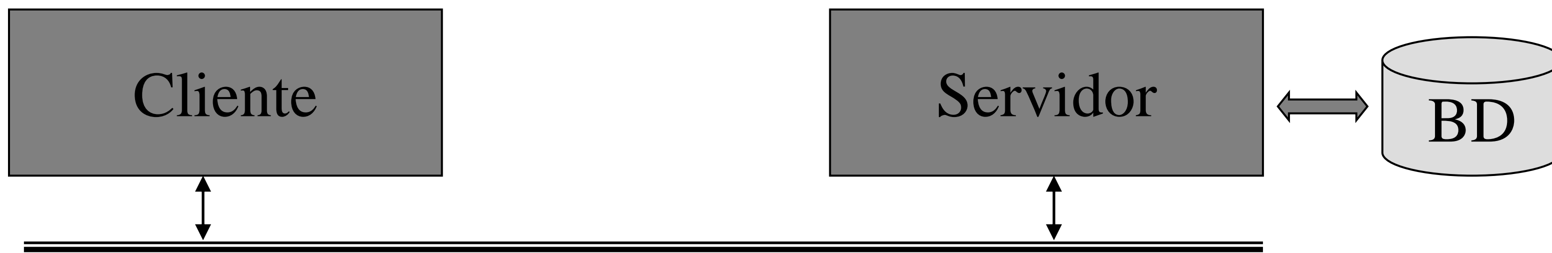
# Cliente/Servidor

- Agenda:
  - Inserir telefone
  - Consultar telefone
  - Remover telefone
- Utilizando Banco de Dados
  - Nome = string
  - Endereço= string
  - Data Nascimento = Date
  - Telefone= string

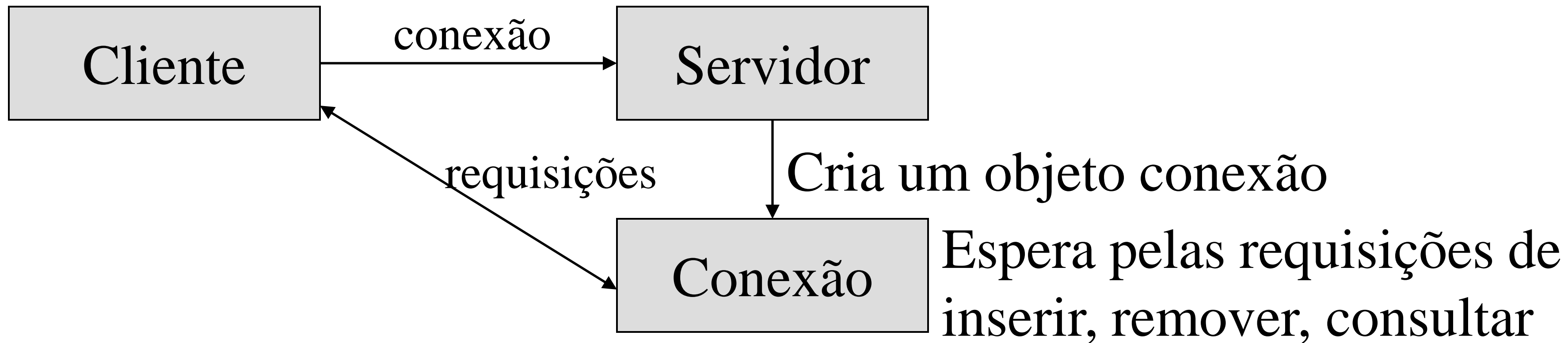
# Cliente/Servidor

- Interface
  - Inserir:
    - Entrada: Nome, Endereco, Data Nasc., Telefone
    - Saída: resultado da operação (0= operação não concluída, 1= operação concluída
  - Consultar:
    - Entrada: Nome
    - Saída: Nome, Endereço, Data Nasc., Telefone => registro encontrado

      Nome= “” =>registro não encontrado
  - Remover
    - Entrada: Nome
    - Saída:resultado da operação (0= operação não concluída, 1= operação concluída



# Cliente/Servidor



# Serviços da Internet

- Serviços são baseados em um protocolo definido
- Descrições dos protocolos normalmente encontrados na RFC (Request for comments)
- www.rfc.net
  - ftp
  - telnet
  - Pop3
  - http (www.w3.org)

# Serviços da Internet

- Desenvolvendo um cliente POP3
  - Baseado na comunicação TCP entre cliente e servidor
    - Servidor: Microsoft Exchange, Sendmail, Lotus
    - Cliente: Outlook, Netscape Mail,..
  - Protocolo POP3: RFC 1939

# Serviços da Internet

- Protocolo

Conexão: host, porta= 110

**+OK Microsoft Exchange POP3 server version 5.5.2653.23 ready**

**USER EU**

**+OK**

**PASS XXXX**

**+OK**

**Comandos do POP3**

**Ex:**

**STAT => +OK 3 300 = número de mensagens, tam. da caixa de mensagens**

**RETR n => busca a mensagem n na caixa de mensagens**

**+OK 120 octets**

**DELE n => apaga a mensagem n da caixa de mensagens**

**QUIT => fechar sessão**

# Serviços da Internet

```
String mens;
Socket so= new Socket("cac-bdc01.unioeste.br", 110);
BufferedReader in= new BufferedReader(new
                   InputStreamReader(so.getInputStream()));
BufferedWriter out= new BufferedWriter(new
                   OutputStreamWriter(so.getOutputStream()));
System.out.println(in.readLine());
mens= "user moyamada\r\n";
out.write(mens);
out.flush();
System.out.println(in.readLine());

mens= "pass XXX\r\n";
out.write(mens);
out.flush();
```

# Serviços da Internet

- Classe URL: abstração de protocolos da Internet.
  - http://www.inf.unioeste.br
  - ftp://ftp.inf.unioeste.br
  - Evitar que os detalhes de protocolo necessitem ser implementados
  - A comunicação é vista como um canal onde os dados podem ser lidos e escritos
  - Pertencente ao pacote java.net.URL

# Serviços da Internet

- public URL(String url) throws MalformedURLException

```
try {
   URL u= new URL(“http://www.inf.unioeste.br”);
}catch (MalformedURLException e){
   System.out.println(“Endereço inválido”+e);
}
```

- Os protocolos suportados pela classe URL são dependentes da implementação da máquina virtual Java
  - http
  - ftp
  - telnet

# Serviços da Internet

- public InputStream openStream() throws IOException
  - Abre um canal para que informações possam ser transferidas
- public URLConnection openConnection() throws IOException
  - Abre uma conexão (entrada e saída)
- public Object getContent() throws IOException
  - Obtém o dado em forma de um objeto. Ex: imagens(gif, jpeg) são retornadas como objetos do tipo java.awt.image, possibilitando o tratamento e a visualização

# Serviços da Internet

- Utilizando a classe URL para verificar o conteúdo de páginas Web

```
class BuscaWeb(){
  public static void main(String args[]){
    if (args.length < 1) {
       System.out.println(“java BuscaWeb  <URL>”);
    }else{
       URL u= new URL(args[0]);
       BufferedReader in= new BufferedReader(new
              InputStreamReader(u.openStream()));
       String s;
       while ((s=in.readLine()!= null) {
          System.out.println(s+ “\n”); //mostra a pagina na tela
       }
    }
  }
}
```

# Serviços na Internet

- Exercício:
  - Faça um analisador de páginas Web. Considere que usuário informará a página e qual palavra o mesmo está querendo analisar. O seu programa deve retornar quantas vezes esta palavra aparece na página
  - Dicas:
    - **regionMatches(boolean ignoreCase,**
      **int toffset,**
      **String other,**
      **int ooffset,**
      **int len)**

**ignoreCase= true, ignora maiúsculas e minúsculas**

**toffset= deslocamento na String s**

**other= string a ser comparada**

**Ooffset= deslocamento na String a ser comparada**

**len= tamanho da String a ser comparada**

# Serviços na Internet

- S=“Pagina WEB teste 123”  , sub= “WEB”
- S.regionMatches(true, 0, sub, 0, 3); falso
- S.regionMatches(true, 7, sub, 0, 3); true

# Objetos Distribuídos

- Middleware
- RPC (Remote Procedure Call): Chamada remota de procedimentos
  - Procedimentos são executados em máquinas remotas
  - A chamada é realizada da mesma forma que procedimentos locais
  - Utilização de um middleware para converter as chamadas em pacotes de rede para o envio da requisição e recebimento da resposta

# Objetos Distribuídos

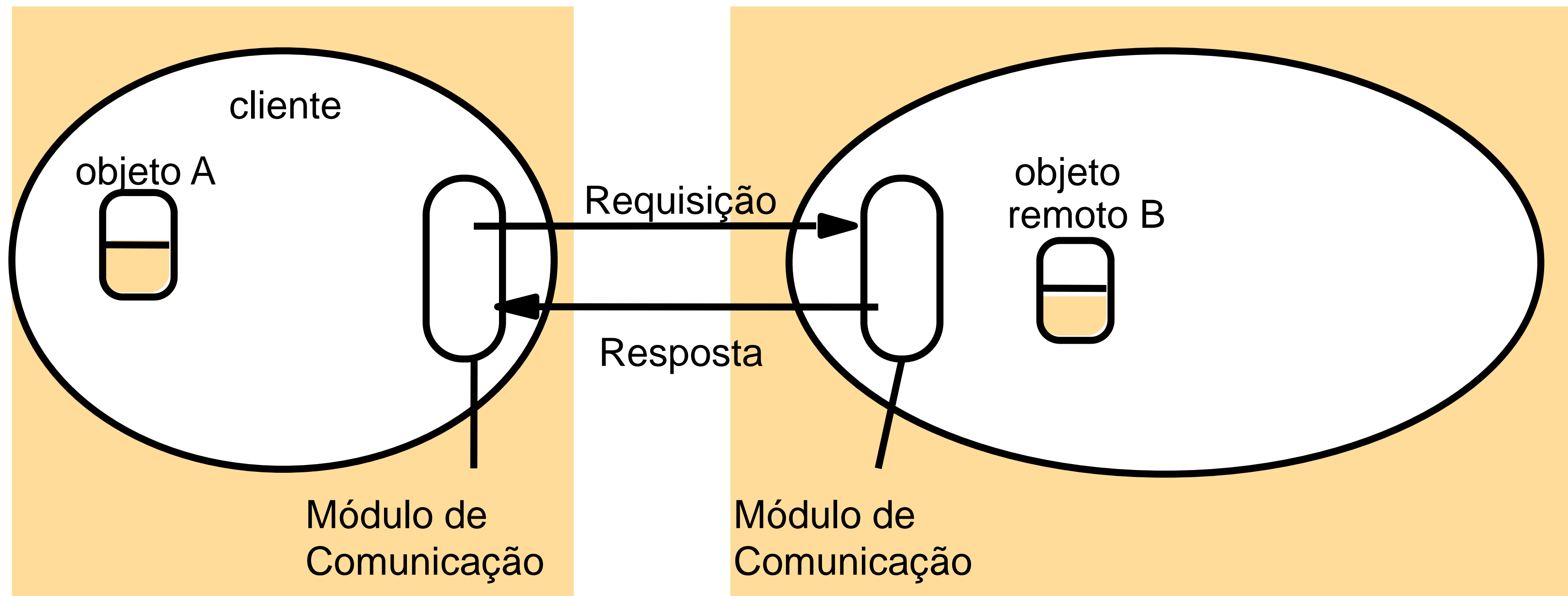
- RMI (Remote Method Invocation): Invocação remota de métodos
  - Objetos podem invocar métodos de outros objetos, que residem em outras máquinas
  - A chamada é realizada da mesma forma que objetos locais
  - Ex: cc.Credita(1000); // objeto residente na máquina local ou pode estar sendo implementado em um servidor remoto

# Objetos Distribuídos

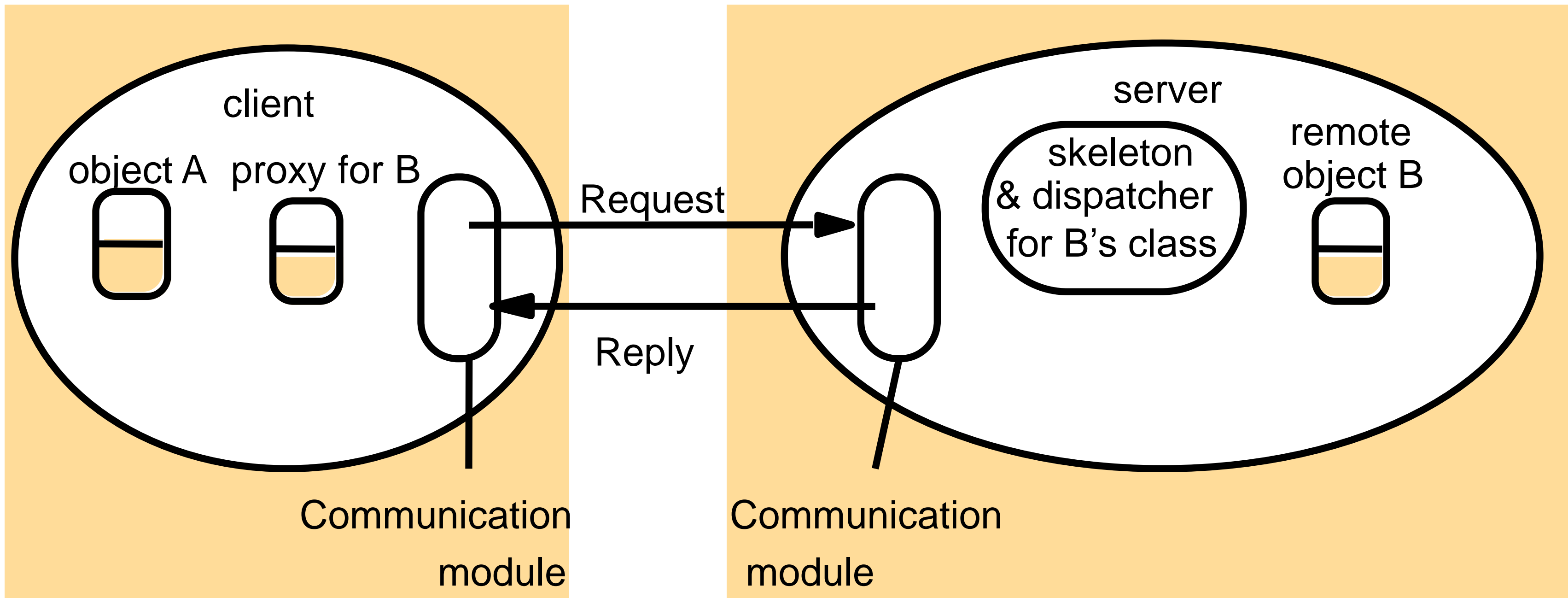
- Vantagens:
  - Transparência: a invocação é realizada da mesma forma tanto para métodos locais quanto remotos (apenas a instanciação é realizada de forma diferente).
  - Protocolo de transferência é implementado pelo middleware
  - Transparência de plataforma(hardware+sistema operacional)
  - Servidor de aplicações: facilidade de manutenção, correção de erros

# Objetos Distribuídos

- Interfaces
  - Todos os métodos podem ser acessados ?
  - Quais métodos e parâmetros do objeto remoto podem ser acessados?



# Objetos distribuídos



# Objetos Distribuídos

- Middleware para objetos distribuídos
  - CORBA (Common Object Request Broker Architecture):
    - Definido pela OMG(Object Management Group): um organização com vários membros que definem a especificação de serviços CORBA
    - A partir da especificação existem vários implementadores da "arquitetura CORBA"
    - Nenhum produto implementa completamente a especificação
    - Serviços: Nomes, Persistência, Replicação, Tempo Real, Tolerância a Falhas
    - Interoperabilidade entre objetos implementados utilizando diferentes linguagens
    - Na prática não existe interoperabilidade entre ORBs

# Objetos distribuídos

- Middleware para objetos distribuídos
  - RMI (Remote Method Invocation): existe a técnica RMI e a tecnologia RMI fornecida pela SUN.
    - Invocação remota de métodos para objetos Java
    - Os objetos são implementados em uma mesma linguagem
    - A interface é definida em linguagem Java
    - Objetos complexos podem ser passados como parâmetros (Implementar a interface Serializable)

# Java RMI

- Interface: define um conjunto de métodos que podem ser acessados remotamente
  - Ex:

```
import java.rmi.*;
interface InterfaceCC extends Remote{
  void setNome(String nome) throws RemoteException;
  void setNumero(String num) throws RemoteException;
  void setSaldo(double saldo) throws RemoteException;
  void Credita(double Valor) throws RemoteException;
  int Debita(double Valor) throws RemoteException;
  boolean loadBD() throws RemoteException;
  boolean saveBD() throws RemoteException;

}
```

# Java RMI

- Alguns objetos precisam ser alterados para que possam ser acessados remotamente
  - A interface não contém construtores
  - Ex: acessos a banco de dados devem ser realizados no servidor
  - Objeto executa na máquina servidor. Recursos de hardware e software

# Representação de dados

- Dados devem ser convertidos em bytes antes do envio
- Representação de dados diferentes em arquiteturas
  - Little endian x big endian
  - Unicode x ASCII
  - Ponto flutuante

# Representação de dados

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <errno.h>

struct teste{
  int i;
  int j;
};

int main(){
   int s;
   struct sockaddr_in sa;
   struct teste t;
   char host[256];
   s= socket(PF_INET, SOCK_STREAM, 0);

   scanf("%s", host);
   sa.sin_family= PF_INET;
   sa.sin_addr.s_addr= inet_addr(host);
   sa.sin_port= htons(8000);
   if (connect(s, (struct sockaddr*)&sa, sizeof sa) < 0){
          perror("connect");
          exit(1);
   }

   t.i= 10;
   t.j= 20;
   write(s, &t, sizeof(t));
   close(s);
   exit(0);
}
```

# Representação de dados- Soluções

- A) valores convertidos em um formato externo antes do envio, e no destino convertido novamente para o formato local
- B) tipos transmitidos no formato do enviador e convertidos no destino caso necessário

- Marshalling: transforma um conjunto de dados em uma seqüência de bytes
- Unmarshalling
- Corba: CDR (common data representation)
- Java: Interface Serializable

# Variantes do modelo cliente servidor

- Processos pares (peer process)
- Código móvel: enviado para execução no cliente (applets)
  - Agentes móveis (código + dados): movimentam-se na rede com algum objetivo específico

# Implementando o objeto servidor

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.sql.*;
class IntCCServant extends UnicastRemoteObject implements InterfaceCC
   {
  private String Numero;  private String Nome;  private double Saldo;
  private Connection conn;
  public IntCCServant(Connection conexaoBD){
     this.conn= conexaoBD;
  }
  public void Credita(double val) throws RemoteException{
    // implementação do método
  }
  public int Debita(double val) throws RemoteException{
    //implementação do método
  }
  public boolean loadBD() throws RemoteException{
    //implementação do método
  }
  public boolean saveBD() throws RemoteException{
     //implementação do método
  }
  public void setNumero(String num) throws RemoteException{ ...}
}
```

# Implementação do inicializador do objeto

```
import java.rmi.*;
Import java.sql.*;

public class AtivaIntCC {
 public static void main(String args[]){
   System.setSecurityManager(new RMISecurityManager());
   try{
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     Connection conn= System.DriverManager("jdbc:odbc:BB",
  "","");
     IntCCServant intcc= new IntCCServant(conn);
     Naming.rebind("IntCC", intcc);
     System.out.println("Objeto Conta Corrente ativo");
   catch (Exception e){
     System.out.println(e);
   }
 }
}
```

# Implementação do cliente

```
import java.rmi.*;
import java.rmi.server.*;
class ClienteCC{
   public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
      IntCCServant cc;
      try{
         cc= IntCCServant)Naming.lookup("rmi://localhost/intCC");
         cc.setNumero("123");
         if (cc.loadBD()){
            System.out.println("Conta não encontrada");
         }
      }catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

# Compilando o projeto

- Gerando as interfaces necessárias para comunicação remota
- Javac InterfaceCC.java
- Javac IntCCServant.java
  - rmic IntCCServant
    - Stub: cliente
    - Skeleton: servidor
- Javac ClienteCC.java
- Javac AtivaIntCC.java

Tempo e estado global em sistemas
distribuídos

# Estados globais

- Estados globais em SD
  - Deadlocks
  - Acesso exclusivo a um banco de dados distribuídos
  - Garbage collection em objetos distribuídos
- Problemas
  - Passagem de mensagens assíncrona
  - Relógios físicos locais: impossível determinar a seqüencia dos eventos e construir um estado global

# Sistema distribuído

- Coleção de processos $p_1,.., p_n$
- $p_i$=single thread
- Processos interagem através de trocas de mensagens
- Cada processo funciona como um transformador de estado

- Onde $e_k$
  - SEND
  - RECEIVE
  - AÇÃO INTERNA

# Sistema distribuído (2)

- Seqüência de eventos: e= $e_i..e_{i+1}$ define a ordenação total de eventos de um processo $p_i$. Relação local aconteceu-antes (happen-before) $\rightarrow_i$
- Histórico do processo $p_i$: seqüência de eventos com relação aconteceu-antes
  - $h_i = <e_0, e_1, ..>$ para todo $e_k$ pertencente ao processo $p_i$

# Relógios físicos

- Contadores em hardware. Registrador H(t)
- Utilizados para gerar interrupções
- Relógio em software
  - $Ci(t) = \alpha Hi(t) + \beta$
  - Resolução do relógio é dependente do contador H(t)
- Skew (distorção): divergência na leitura de 2 relógios
- Drift (desvio): diferença na taxa de contagem de 2 relógios
  - Devido a diferenças no cristal, mas também temperatura, voltagem, etc...
- Clock drift rate: diferença na precisão entre uma referência perfeita e o relógio físico
  - Tipicamente $10^{-6}$/sec

# UTC

- Universal coordinated time
- Relógios atômicos
  - Desvio: $10^{-13}$
  - Disponível através do GPS

# Sincronização

- Externa: sincronizar um relógio de um processo com uma referência externa S(t), limitando a distorção em D
  - $| s(t) – C(t)| < D$ para todo t

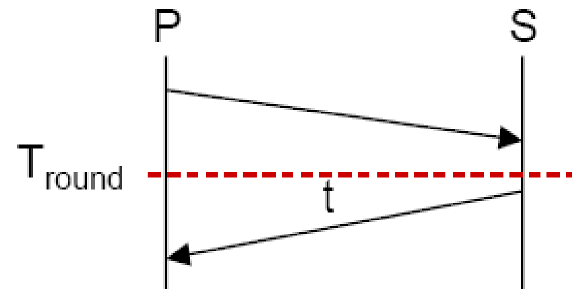
- Interna: sincronizar os relógios locais em um sistema distribuído
  - $| Ci(t) – Cj(t) | < D$ para todo i, j, t
- Para um sistema sincronizado externamente em D, a distorção interna será 2D

# Relógio

- Correctness
  - Se a taxa de desvio é limitada a $r > 0$, então para qualquer $t$ e $t'$ onde $t' > t$, os seguintes limites de erros são obtidos
  - $(1-r)(t'-t) \leq H(t') - H(t) \leq (1+r)(t'-t)$
  - Não é permitido saltos no relógio além do limite
  - Monotocidade
    - $t' < t$ então $H(t') < H(t)$

# Algoritmo de Cristian

- Sistemas assíncronos
- Os tempos de ida e volta são relativamente pequenos, ainda que teoricamente ilimitados
- Praticável, se o tempo de ida e volta for pequeno em relação a precisão requerida
- Princípio
  - Utilizar um servidor S de tempo
  - Um processso P envia uma requisição a S e mede Tround
  - Estimativa: atualiza o relógio para t + Tround/2



# Algoritmo de Cristian (precisão)

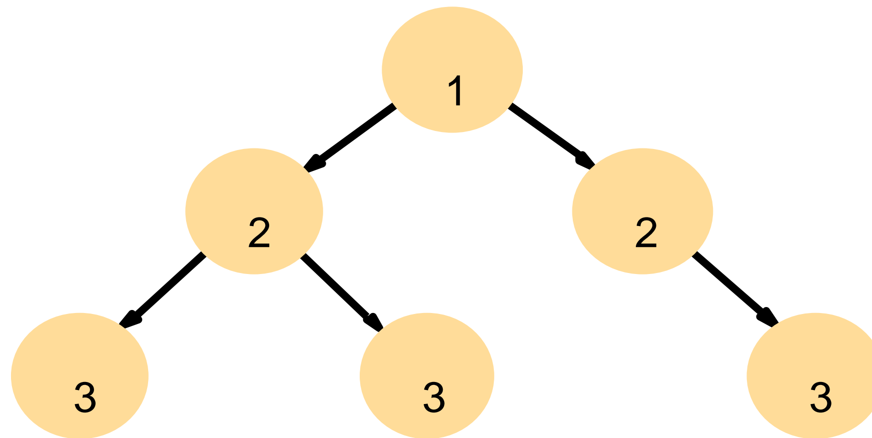
- Precisão da estimativa?
- Considerações:
  - Requisição e resposta trafegam na mesma rede
  - O tempo mínimo de transmissão (min) é conhecido ou pode ser calculado de maneira conservativa
- Calculo
  - O mais rápido que S pode responder : t + min
  - O mais tarde que S pode responder: t + Tround - min
  - Faixa de resposta: Tround - 2 min
  - precisão +/- (Tround/2 - min)
- Aplicavél somente para redes locais e intranet
- Falha em S: redundância
- Impostor: autenticação

# Algoritmo de Berkeley- Interna

- Mestre consulta os computadores escravos sobre seus relógios locais
- Estimativa do relógio local: algoritmo de Cristian
- Média dos valores obtidos: não privilegiar os relógios rápidos e lentos
- Envia aos escravos a quantidade de tempo que eles devem sincronizar
- Ex:
  - 15 computadores
  - Relógios sincronizados a cada 20-25 msec
  - Desvio $< 2x10^{-5}$
  - $Tround_{max} = 10$ msec

# NTP – Network time protocol

- Possibilitar a sincronização de clientes externos na UTC
- Prover confiabilidade em redes de baixa conectividade
- Subredes de sincronização



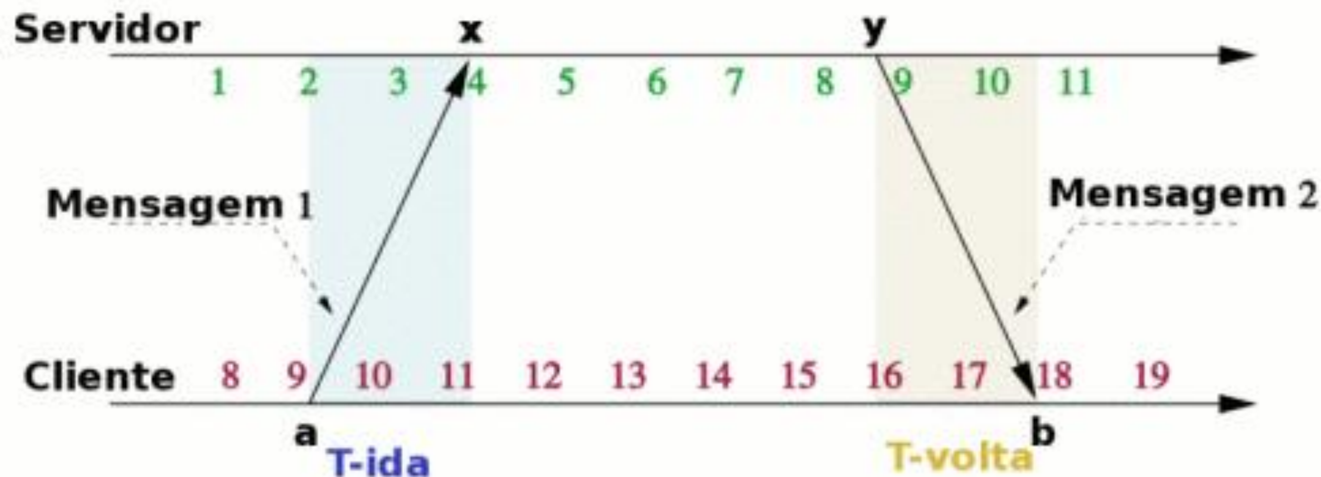

Note: Arrows denote synchronization control, numbers denote strata.

# NTP

- Arquitetura em camadas, cliente-servidor utilizando UDP
- Os clientes das camadas maiores tem menor precisão
- Tolerância a falhas: se o servidor da camada 1 falha, um servidor da camada 2 é eleito
- Modos de sincronização
  - Multicast
  - RPC (chamada de procedimento remoto)
  - simétrico

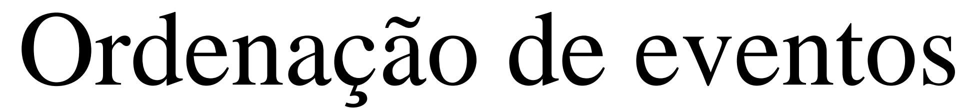
# NTP (2)

- Estimativa da diferença do relógio
  - $o_i = (x - a + y - b)/2$
  - $di = (b\text{-}a) - (y\text{-}x)$
  - Precisão: $o_i$ é uma estimativa do deslocamento real o
    - $o_i - d_i/2 \leq o \leq o_i + d_i/2$



# NTP(3)

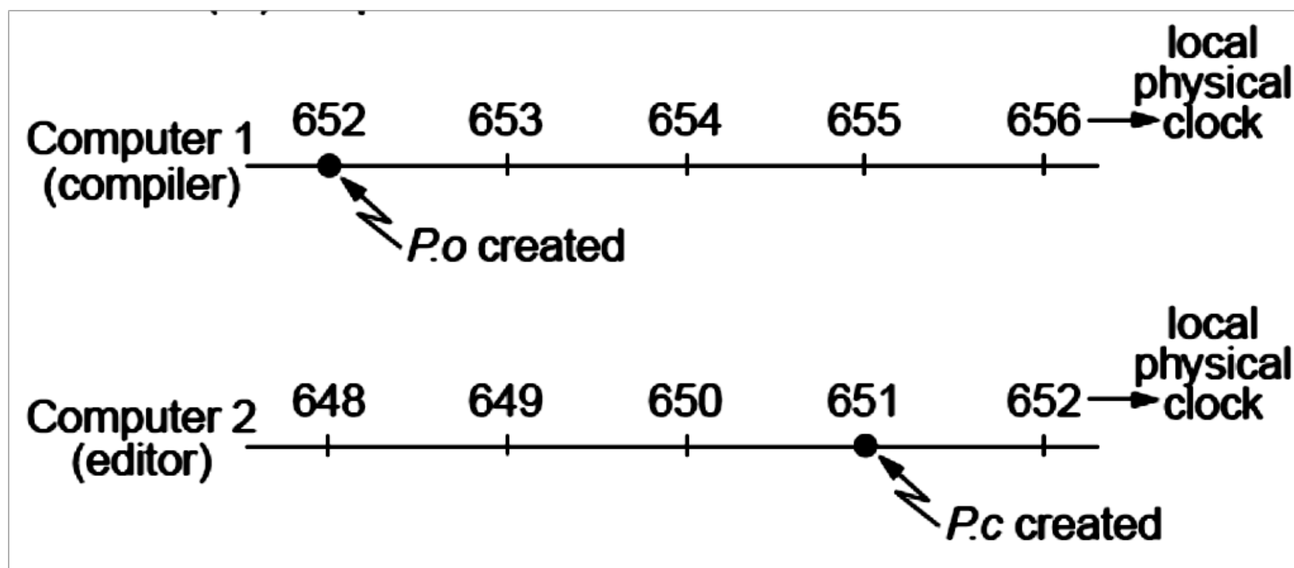
- Estatística: analisa os 8 pares recentes de <oi, di> para verificar a qualidade da estimativa
- Escolhe como fator de correção o menor di
- Experimentos reportam precisões de 10 ms para internet e 1 ms para LAN



# Ordenação de eventos

# Ordenação de eventos

- Utilizando relógios em hardware
  - Supondo que uma sincronização interna de 10-3s pode ser obtida
  - Quantas instruções são executadas neste intervalo?
  - Precisão insuficiente para marcar o instante de uma operação utilizando tais relógios

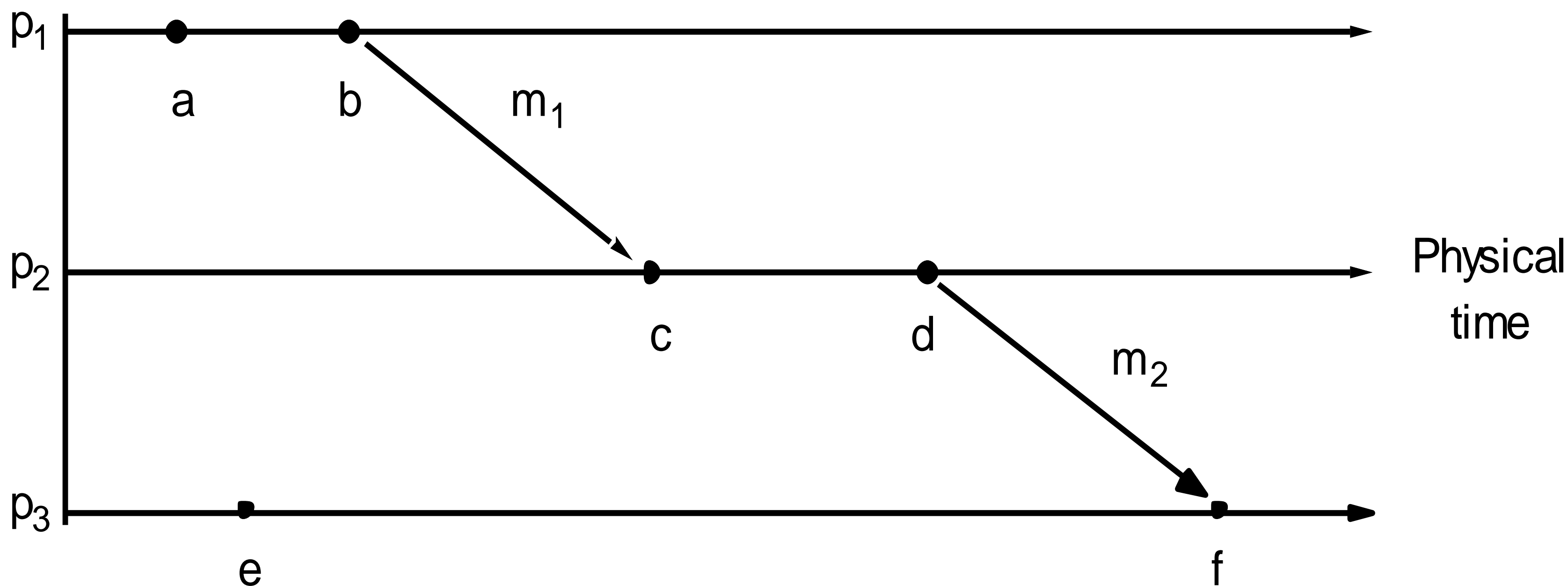


# Ordenação lógica de eventos

- Lamport: relação aconteceu-antes (happened-before)
  - Se dois eventos ocorrem em um **mesmo processo** pi, então eles ocorreram na ordem observada por pi (isto é, relação local aconteceu-antes)
  - Para qualquer **troca de mensagem**, o evento de *enviar* acontece antes do evento *receber*
  - Relação aconteceu-antes (->):
    - AA1: para qualquer par de eventos e  e', se existe um processo pi onde e ->i e', então e -> e'
    - AA2: para qualquer par de eventos e  e', onde e = send(m) e' = receive(m), então e -> e'
    - AA3: se e, e', e'' são eventos, se  e -> e', e' -> e'', então e -> e''(fechamento transisti)
  - AA define uma ordem parcial

# Ordenação de eventos

- Concorrência
  - Se e not -> e’, e’ not ->, e || e’ (eventos concorrentes)

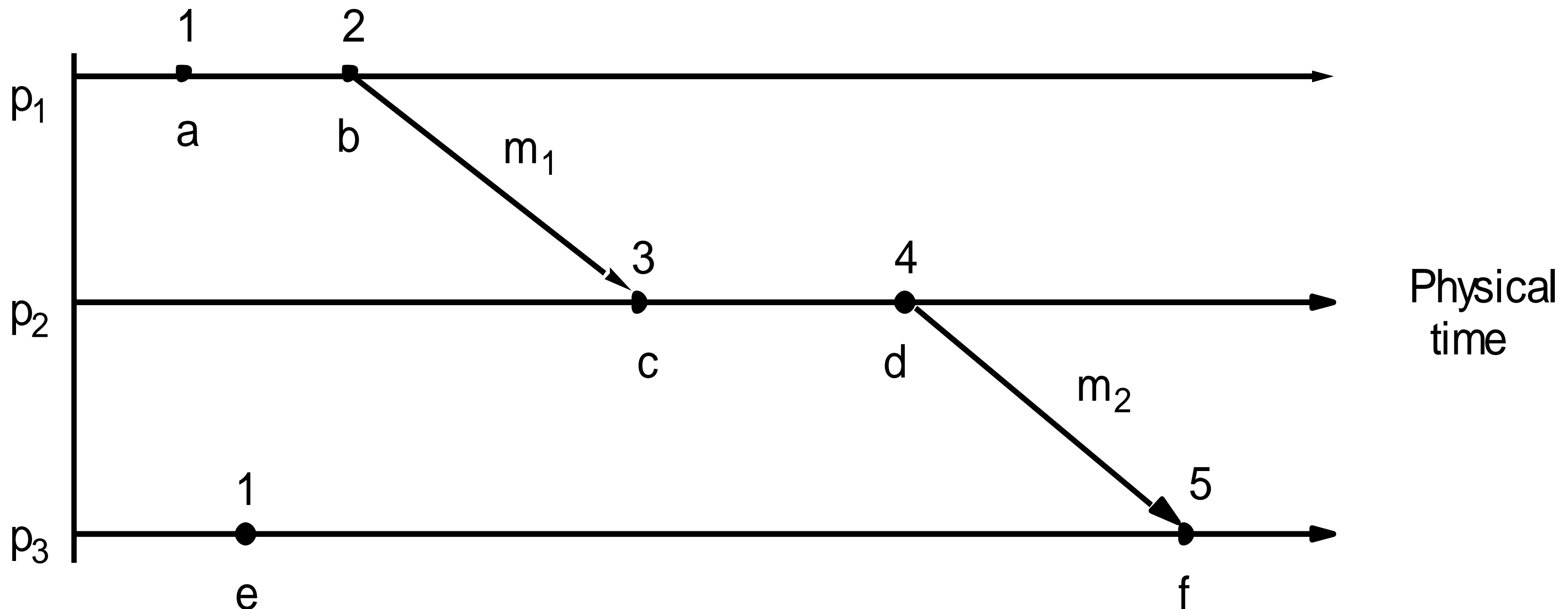


# Relógio lógico de Lamport

- Relógio lógico: permite inferir sobre a ordem de ocorrência dos eventos
- Aumenta de forma monotônica os contadores que impõem uma ordem total dos eventos
- Todo processo p1, mantém um relógio local
- Li(e): marca de tempo (timestamp) do evento e no processo i
- Mensagens transportam a marca de tempo do evento send
- Regras para atualizar o relógio lógico:
  - LC1: Li é incrementado antes de cada evento no processo pi
  - LC2: um processo i transmite Li em todas as mensagens enviadas
  - LC3: recebendo uma mensagem (m, t), um processo pj computa Lj := max(Lj, t), incrementa Lj, e então marca a mensagem receive(m, t)

# Relógio lógico de Lamport(2)

- Se e -> e’, então LC(e) < LC(e’), porém se LC(e) < LC(e’) não podemos afirmar que e -> e’
- Determinar o acesso de processos concorrentes esperando para entrar na seção crítica

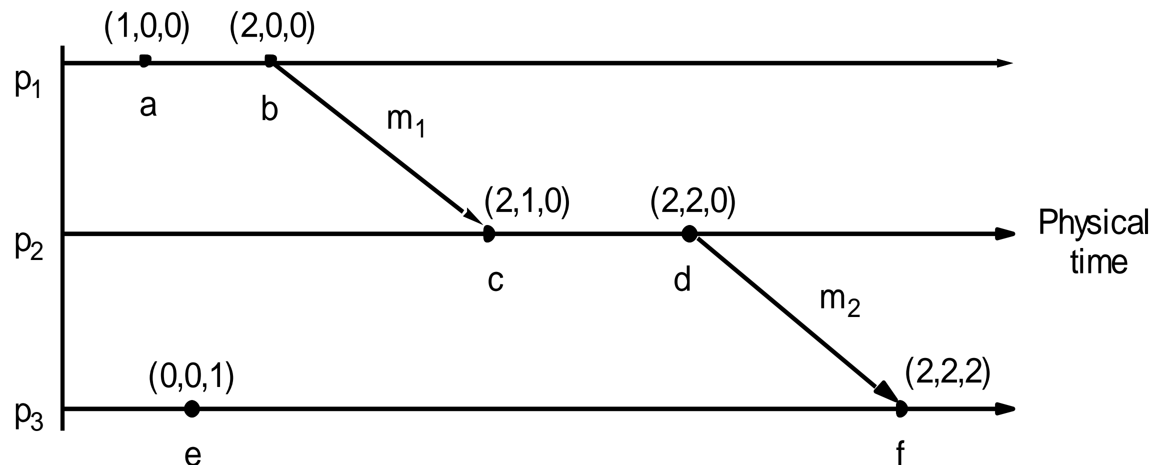


# Relógios vetoriais

- Vetor de N inteiros, onde n= número de processos
- Cada processo i tem seu vetor Vi[1,..,N]
- A marca dos eventos segue a mesma regra de RL de Lamport
- Regras de atualização dos relógios
  - VC1: inicialmente todos os relógios começam em 0
  - VC2: Vi[i]= Vi[i]+1 antes de cada evento
  - VC3: i inclui t= Vi em todas as mensagens enviadas
  - VC4: i recebe a marca t, então Vi[j]= max(Vi[j], t[j]) para qualquer j=1..N

# Relógios vetoriais

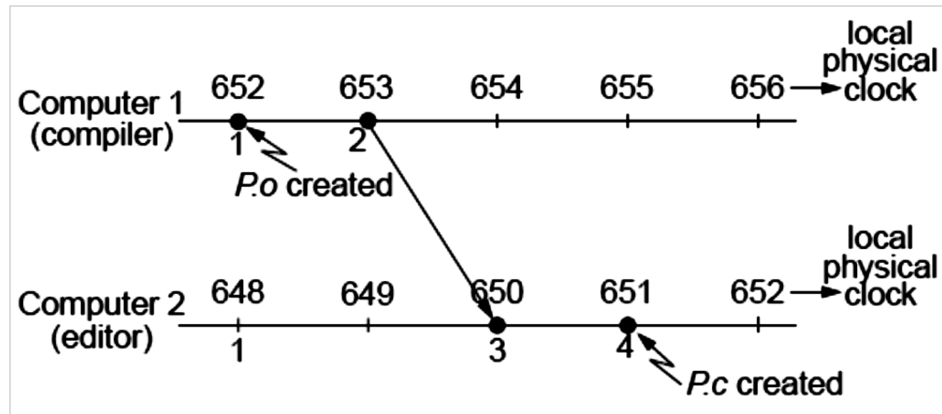
- Vi[j]= é o número de eventos em j que i foi potencialmente afetado
  - V = V', se somente se V[j]= V'[j], para qq j=1..N
  - $V \leq V'$ se somente se $V[j] \leq V'[j]$, para qq j= 1..N
  - V < V'se somente se $V \leq V'$ e $V \neq V'$




V(b) < V(d)

V(e) e V(d) e || d

# Exemplo: Make



Sincronização em sistemas distribuídos

# Sincronização

- Sistemas distribuídos assíncronos: nenhum processo tem uma visão consistente do estado global
- Objetivos comuns
  - Detecção de falhas
  - Exclusão mútua:
  - Eleição
  - Multicast
  - Consenso

# Exclusão mútua

- Em sistemas multitarefas
  - Acesso a recursos compartilhados: memória, impressora, etc
  - Instruções test&set
  - Semáforos
  - Monitores
- Sistemas distribuídos
  - Sem memória compartilhada
  - Sem um elemento central (kernel do sistema operacional que recebe as chamadas para adquirir e liberar o semáforo)

# Requisitos da exclusão mútua

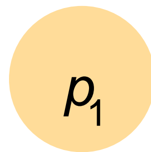
- Regra 1: Exclusão mútua
- Regra 2: Progressão
  - Nenhum processo fora da seção crítica pode bloquear um outro processo
- Regra 3: Espera limitada
  - Nenhum processo pode esperar infinitamente para entrar na seção crítica
- Regra 4
  - Nenhuma consideração sobre o número de processadores ou velocidades relativas

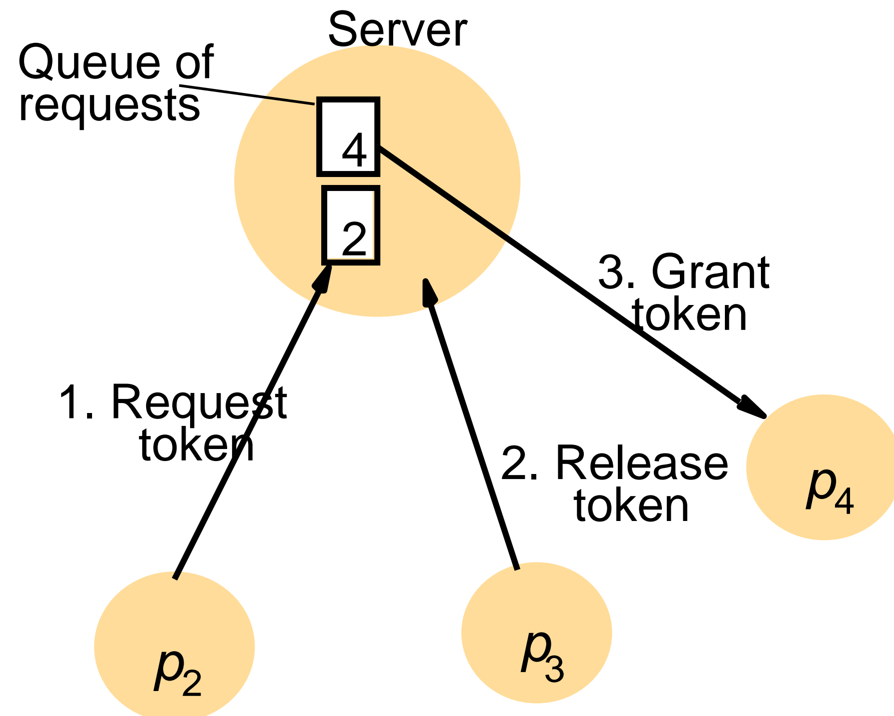
# Exclusão mútua

- Critérios de avaliação
  - Bandwidth: número de mensagens consumidas para sincronizar a entrada e saída da seção crítica
  - client delay: espera para cada entrada e saída da seção crítica
  - throughput: número de entradas na seção crítica em um espaço de tempo – atraso de sincronização entre a saída de um cliente e a entrada em de outro cliente

# Cliente-servidor: exclusão mútua

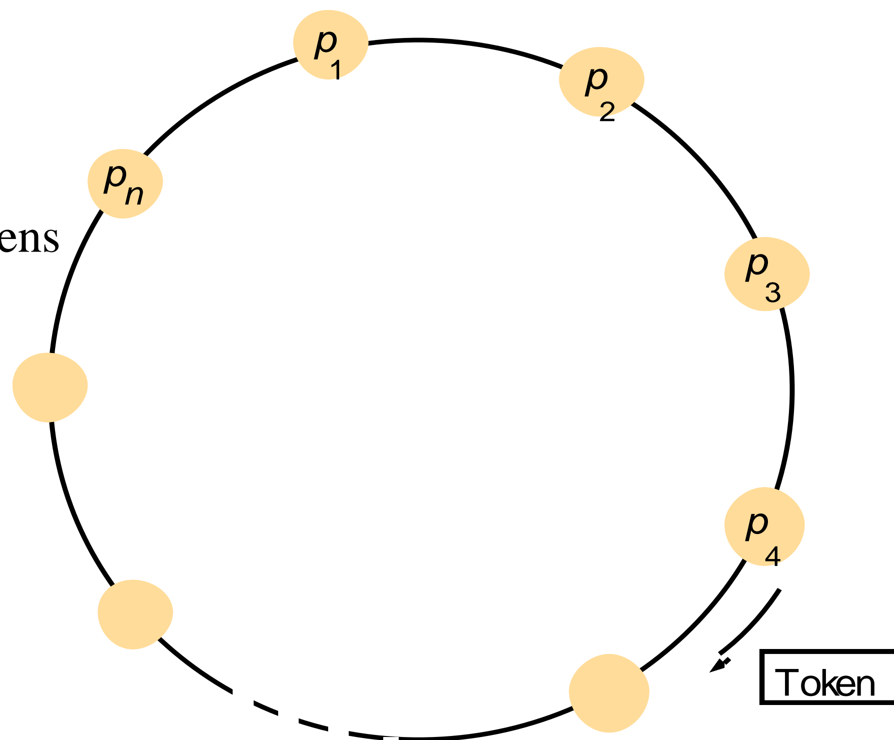
- Um servidor central recebe as requisições
  - Se não existe processo esperando, o processo entra na seção crítica
  - Se existe um processo na seção crítica, a requisição é enfileirada
- Quando um processo deixa a seção crítica
  - Libera o acesso para o próximo processo na fila ou espera
- Duas mensagens por requisição, uma para entrar e outra para sair da seção crítica
- Desempenho e disponibilidade do servidor são os gargalos



# Token-ring

- Anel lógico ou físico: todo o processo pi tem uma conexão com o processo pi+1
- O token circulo em uma direção através do anel
- Na chegada do token
  - Se o processo que recebe o token deseja entrar na seção crítica, prende o token
  - Senão, repassa o token para o vizinho
  - Não obedece a ordem de solicitação
  - Atraso de entrada: 0 a N mensagens
  - Atraso de Sincronização : 0 a N mensagens



# Algoritmo Ricart e Agrawala

- Baseado em multicast
  - Um processo requisitando entrada na seção crítica, envia a solicitação via multicast
  - Um processo somente entra na seção crítica se todos os processos responderem com ACK positivo
- Considerações
  - Todos os processos têm canais de comunicação para outros processos
  - Todos os processos tem um ID único e mantêm relógios lógicos

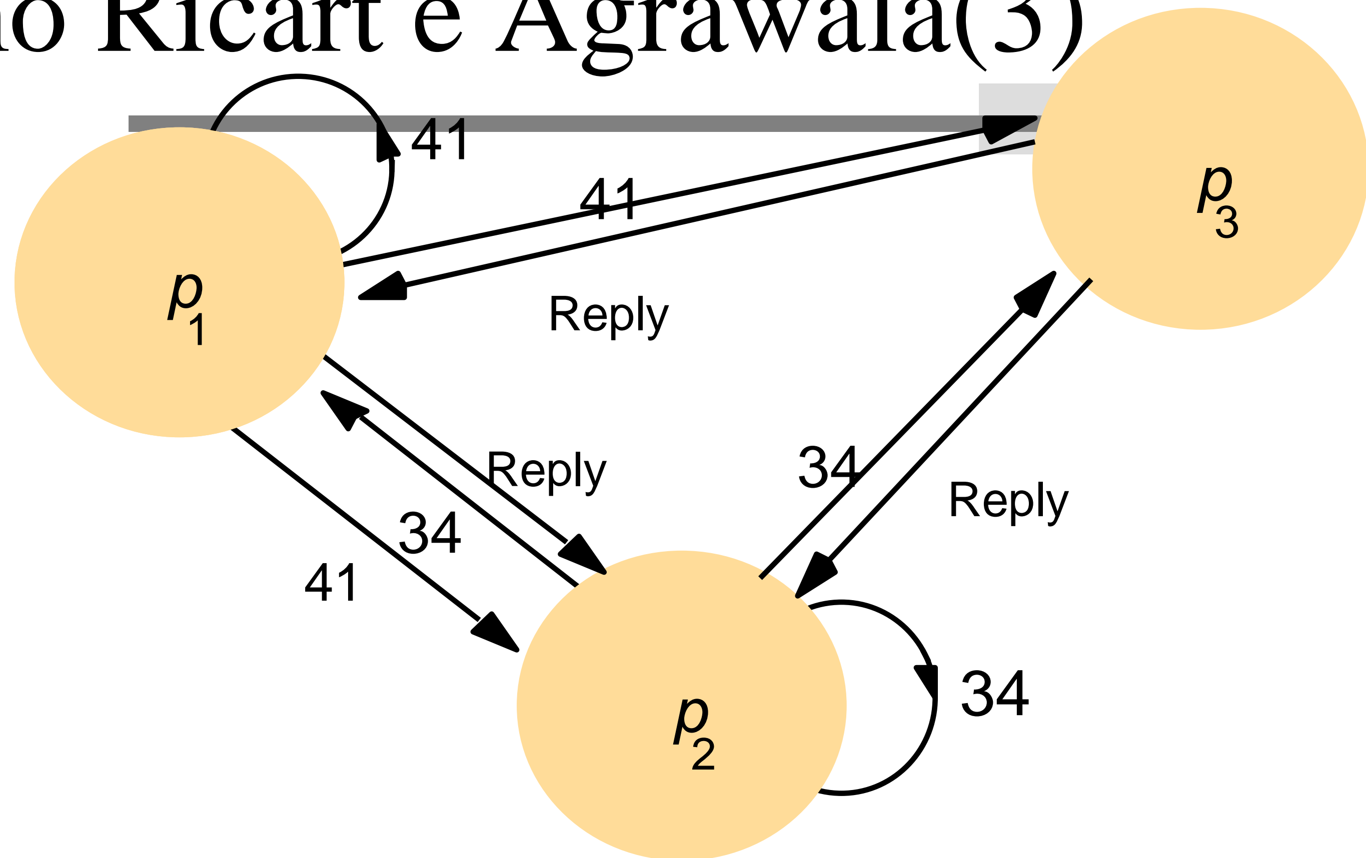
# Algoritmo Ricart e Agrawala(2)

- Desempenho
  - Acesso a seção crítica necessita de 2(N-1) mensagens
  - Atraso de sincronização: ida e volta (roundtrip)

*On initialization*
  *state* := RELEASED;
*To enter the section*
  *state* := WANTED;
  Multicast *request* to all processes;  request processing deferred here
  *T* := request's timestamp;
  *Wait until* (number of replies received = ($N$ – 1));
  *state* := HELD;

*On receipt of a request* $<T_i, p_i>$ *at* $p_j$ $(i \neq j)$
  *if* (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
  *then*
    queue *request* from $p_i$ without replying;
  *else*
    reply immediately to $p_i$;
  end if
*To exit the critical section*
  *state* := RELEASED;
  reply to any queued requests;

# Algoritmo Ricart e Agrawala(3)




- P1 e P2 requisitam a entrada na seção crítica
  - P3 responde imediatamente
  - P2 recebe a requisição de P1, timestamp(P2) < timestamp(p1), logo P2 não responde
  - P1 nota que o seu timestamp é maior que o da requisição de P2, então ele responde imediatamente para P2
  - P2 irá responder a requisição de P1 quando o mesmo sair da seção crítica

# Algoritmo de Maekawa

- Baseado em votação
  - Para o obter acesso à seção crítica, nem todos os processos devem concordar
  - É necessário apenas dividir o conjunto de processos em subconjuntos (conjuntos de votação), que se sobrepõem
  - É necessário apenas que haja consenso em todo subconjunto
- Modelo do sistema
  - Processos p1,.., pN
  - Conjunto de votação V1,.., VN escolhidos de tal forma que para qq i, k e para algum inteiro M:
    - $Pi \in Vi$
    - $Vi \cap Vk \neq \emptyset$
    - $|Vi| = K$ (todos os conjuntos de votos têm o mesmo tamanho)
    - Cada processo Pk, está contido em M conjunto de votos

# Algoritmo de Maekawa(2)

- Protocolo
  - Para obter a seção crítica, Pi enviar uma mensagem de requisição para todos os processos K-1 do conjunto de votos Vi
  - Não pode entrar até receber K-1 respostas
  - Na saída da seção crítica, envia uma mensagem liberar  para todos os membros de Vi
  - Quando recebe uma requisição
    - Se estado = OBTEVE ou já replicou (votou) desde a última requisição
      - então, enfileira a requisição
    - Senão, responde imediatamente
  - Quando recebe um liberar
    - Remove a requisição da cabeça da fila e envia uma resposta

# Algoritmo de Maekawa (3)

*On initialization*
  *state* := RELEASED;
  *voted* := FALSE;
*For $p_i$ to enter the critical section*
  *state* := WANTED;
  Multicast *request* to all processes in $V_i$;
  *Wait until* (number of replies received = $K$);
  *state* := HELD;
*On receipt of a request from $p_i$ at $p_j$*
  *if* (*state* = HELD *or voted* = TRUE)
  *then*
    queue *request* from $p_i$ without replying;
  *else*
    send *reply* to $p_i$;
    *voted* := TRUE;
  *end if*

*For $p_i$ to exit the critical section*
  *state* := RELEASED;
  Multicast *release* to all processes in $V_i$;
*On receipt of a release from $p_i$ at $p_j$*
  *if* (queue of requests is non-empty)
  *then*
    remove head of queue – from $p_k$, say;
    send *reply* to $p_k$;
    *voted* := TRUE;
  *else*
    *voted* := FALSE;
  *end if*

$S_1 = \{1, 2, 4\}$
$S_2 = \{2, 6, 7\}$
$S_3 = \{3, 4, 6\}$
$S_4 = \{4, 5, 7\}$
$S_5 = \{5, 2, 3\}$
$S_6 = \{6, 5, 1\}$
$S_7 = \{7, 3, 1\}$

# Algoritmo de Maekawa (4)

- Otimização: minimizar K, e ao mesmo tempo obter a exclusão mútua
  - Pode ser obtida quando K~ √N e M= K
- Conjunto de votos ótimo: não é trivial calcular
- Aproximação de Vi, para que | Vi | ~ 2 √N
  - Colocar os processos em uma matriz √N por √N
  - Pegar Vi como sendo a união das linhas e colunas contendo Pi
- Exclusão mútua
  - Como sempre haverá uma intersecção, o processo da intersecção vai decidir por um processo ou outro
- Requisição
  - Deadlocks são possíveis
  - Considere 3 processos
    - V1= { p1, p2}, V2={ p2, p3}, V3= {p3, p1}
    - É possível construir um grafo cíclico
  - p1 responde para p2, mas enfileira uma requisição de p3
  - p2 responde para p3, mas enfileira uma requisição de p1
  - p3 responde para p1, mas enfileira uma requisição de p2

# Algoritmo de Maekawa (5)

- O algoritmo pode ser adaptado para ser tornar livre de deadlocks
  - Uso de relógios lógicos
  - Processos enfileiram requisições na ordem aconteceu-antes
- Desempenho
  - Largura de banda
    - 2 $\sqrt{N}$ por entrada, $\sqrt{N}$ por saída
    - 3 $\sqrt{N}$ é melhor que o algoritmo de Ricart e Agrawala par $N > 4$
  - Atraso no cliente
    - Igual ao de Ricart e Agrawala
  - Atraso na sincronização
    - Tempo de ida e volta, ao invés de uma simples mensagem como o Ricart e Agrawala

RPC

# SUN/RPC

- Chamada remota de procedimentos (RPC): capacidade de executar  procedimentos implementados em outras máquinas
- Serviços que utilizam RPC
  - NIS
  - NFS

# RPC: Modelo cliente e servidor

- **Caller**: programa que chama o procedimento
- **Callee**: implementador do procedimento
- **Client**: requisita as conexões através da rede
- **Server**: programa que aceita conexões e disponibiliza serviços para o cliente

# Execução

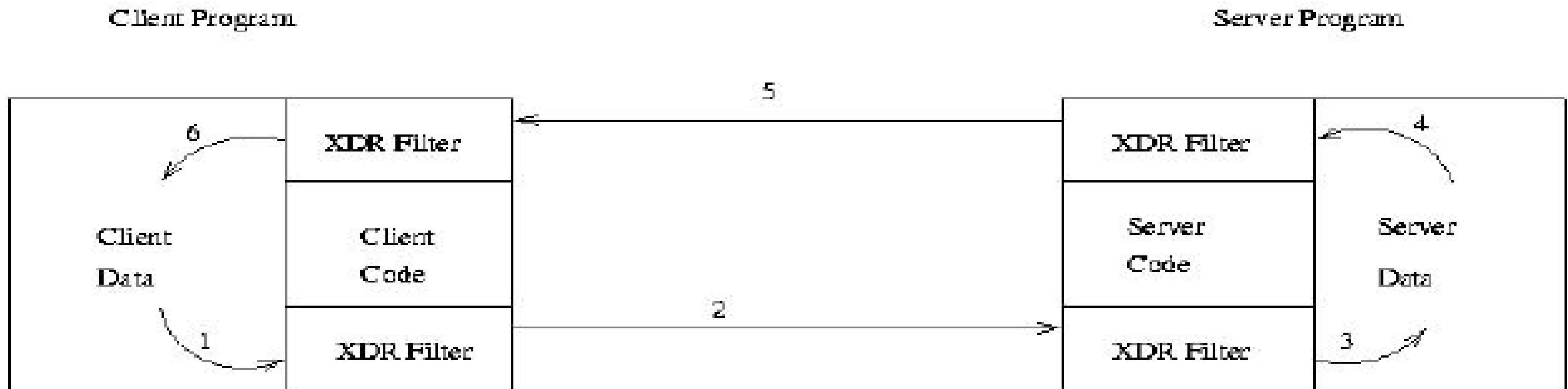
1. O *caller* deve preparar qualquer parametros de entrada para serem passados para o RPC. O *caller* e *callee* podem executar em máquinas completamente diferentes
2. No RPC o endereço é normalmente um endereço de rede onde o servidor está executando
3. O RPC recebe os dados de entrada executa o programa, e retorna qualquer resultado necessário para o *caller*.
4. O *caller* recebe o resultado e continua a execução

# XDR- Representação externa

- XDR: coleção de funções e macros em C que realizam a conversão de dados e representações para um formato padrão. No recebimento, ele deve converter novamente os dados para o padrão da máquina local
  - int, float and string
  - Define a transmissão de dados complexos como registros, arrays, etc

# RPC

## RPC Dataflow




1. Client encodes data through XDR filter.
2. Client passes XDR encoded data accross network to remote host
3. Server decodes data through XDR filter.
4. Server encodes function call result through XDR filter
5. Server pass XDR encoded data accross network back to client
6. Client decodes **RPC** result through XDR filter and continues processing

Figure 1.

# Portmapper

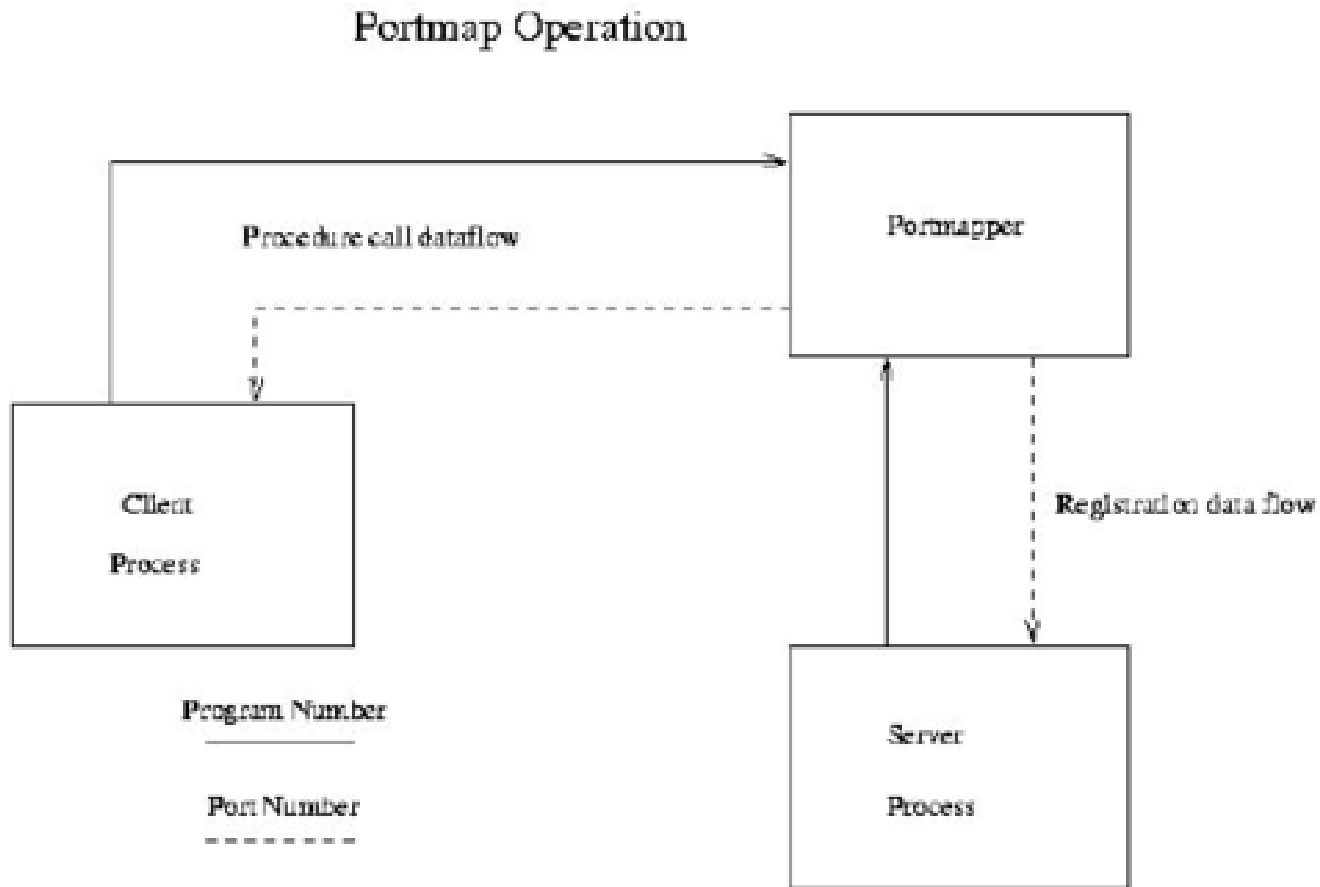


Figure 2

# Servidor: Soma de dois números

```
/* Arquivo soma.x */
struct numeros {
    int a;
    int b;
};
program SOMAPROG {
    version SOMAVERS {
        int soma(numeros) = 1;
    } = 1;
} = 3501;
```

# IDL

RPCdemo.x

Define program

Define interface

```
program RPCDEMO {
        version RPCDEMOVERS {

                int ADD(int, int) = 1;

        } = 1;

} = 0x20004088;
```

Define function

Ensure uniqueness (e.g., last four student ID digits)

# rpcgen

- Gerando os stubs

  rpcgen –N –a soma.x

Alterar o arquivo soma_client.c

result_1 = soma_1(soma_1_arg1, soma_1_arg2, clnt);

substituir por:

result_1 = soma_1(10, 20, clnt);

printf(“Soma = %d”, result_1);

Alterar o arquivo soma_server.c

result= arg1+ arg2;

# Gerando os arquivos

make –f Makefile.soma

Executando o servidor

soma_server

Executando o cliente

soma_client localhost

# Arrays e estruturas

```
struct b{
  unsigned char buf[512];
};
typedef struct b buf;
program RPCDEMO{
   version RPCv1 {
      buf zera() =1 ;
      buf set(char )  =2;
      void grava(buf)=3;
  } = 1;
}= 0x20000000;
```

# Definindo as interfaces

```
/* Arquivo quadrado.x */
program QUADRADOPROG {
    version QUADRADOVERS {
        int quadrado(int num) = 1;
    } = 1;
} = 3500;
```

- rpcgen quadrado.x

# RPC Server

```
#include <rpc/rpc.h>
#include "quadrado.h"
#include <stdio.h>
int a;

int * quadrado_1(int *input,
   CLIENT *client) {
   a= *input;
  a= a*a;
  return(&a);
}

int * quadrado_1_svc(int *input,
   struct svc_req *svc) {
  CLIENT *client;
  return(quadrado_1(input,client));
}
```

# RPC Client

```
#include "quadrado.h"
#include <stdlib.h>
void quadradoprog_1( char* host, int argc, char *argv[]){
   CLIENT *clnt;
   int value;
   int *result_1;
   value= 3;
   clnt = clnt_create(host, QUADRADOPROG,   QUADRADOVERS, "udp");
   if (clnt == NULL) {
      clnt_pcreateerror(host);
      exit(1);
   }
   result_1 = quadrado_1(&value, clnt);
   if (result_1 == NULL) {
      clnt_perror(clnt, "call failed:");
   }
   clnt_destroy( clnt );
      printf("quadrado = %d\n",*result_1);
}

main( int argc, char* argv[] ){
   char *host;

   host = argv[1];
   quadradoprog_1( host, argc, argv);
}
```

# Program number

- 0x00000000 - 0x1fffffff defined by SUN
- 0x20000000 - 0x3fffffff defined by user
- 0x40000000 - 0x5fffffff transient
- 0x60000000 - 0xffffffff reserved

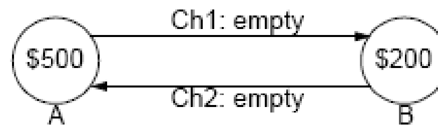
Estados Globais

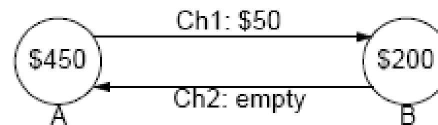
# Estado global

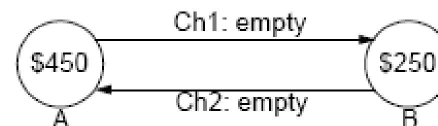
- Como coletar e armazenar um estado global consistente de um sistema distribuído?
- Problema
  - Não existe relógio global e não existe memória compartilhada

# Estado global

- Duas contas em um banco armazenadas em locais diferentes




| A | Ch1 | B | C : consistent<br>NC: not consistent |
|---|---|---|---|
| 500 | empty | 200 | C |
| 500 | 50 | 200 | NC |
| 450 | 50 | 200 | C |
| 450 | empty | 200 | NC |
| 500 | 50 | 250 | NC |
| 450 | 50 | 250 | NC |
| 450 | empty | 250 | C |
| 500 | empty | 250 | NC |

# Estado global

- Conjunto de estados locais e os estados dos canais de comunicação
- O estado do canal de comunicação em um estado global consistente deverá ser a seqüência de mensagens enviadas  através do canal antes que o estado do processo de enviador seja gravado, excluindo a seqüência de mensagens recebido pelo canal antes do estado do receptor seja gravado
- Como os estados dos canais não são armazenados, normalmente os estados globais não são armazenados com os estados dos canais

# Estado global consistente

- Construir um estado global a partir das informações do processo, de forma que esse corte  consistente