

SISTEMAS DISTRIBUÍDOS



O que são sistemas distribuídos?

“A collection of independent computers that appears to its users as a single coherent system”

Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.

(Tanenbaum e van Steen, 2007)

Pontos: Sistema único e transparência.

O que são sistemas distribuídos?

“A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.”

Sistema no qual os componentes, localizados em computadores interligados em rede, se comunicam e coordenam suas ações apenas enviando mensagens entre si. (Coulouris et al., 2012)

Pontos: Ausência de relógio global; independência de falhas; e execução concorrente dos componentes de hardware e software .

O que são sistemas distribuídos?

“Um sistema distribuído é aquele em que a falha de um computador que você nem sabia que existia, torna o seu computador indisponível” (Lamport, 1987)

```
Received: by jumbo.dec.com (5.54.3/4.7.34)
      id AA09105; Thu, 28 May 87 12:23:29 PDT
Date: Thu, 28 May 87 12:23:29 PDT
From: lamport (Leslie Lamport)
Message-Id: <8705281923.AA09105@jumbo.dec.com>
To: src-t
Subject: distribution
```

There has been considerable debate over the years about what constitutes a distributed system. It would appear that the following definition has been adopted at SRC:

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

The current electrical problem in the machine room is not the culprit--it just highlights a situation that has been getting progressively worse. It seems that each new version of the nub makes my FF more dependent upon programs that run elsewhere.

Having to wait a few seconds for a program to be swapped in is a lot less annoying than having to wait an hour or two for someone to reboot the servers. I therefore propose a development project to make our system more robust. I am not proposing any particular approach (enabling stand-alone operation is just one possibility).

I will begin the effort by volunteering to gather some data on the problem. If you know of any instance of user's FF becoming inoperative through no fault of its own, please send me a message indicating the user, the time, and the cause (if known).

Leslie

Pontos: **Transparência e necessidade de tolerância a falhas.**

O que são sistemas distribuídos?

“Originally, this term was used to refer to algorithms that were designed to run on many processors "distributed" over a large geographical area. But over the years, the usage of this term has been broadened, so that it now includes algorithms that run on local area networks and even algorithms for shared memory multiprocessors.”

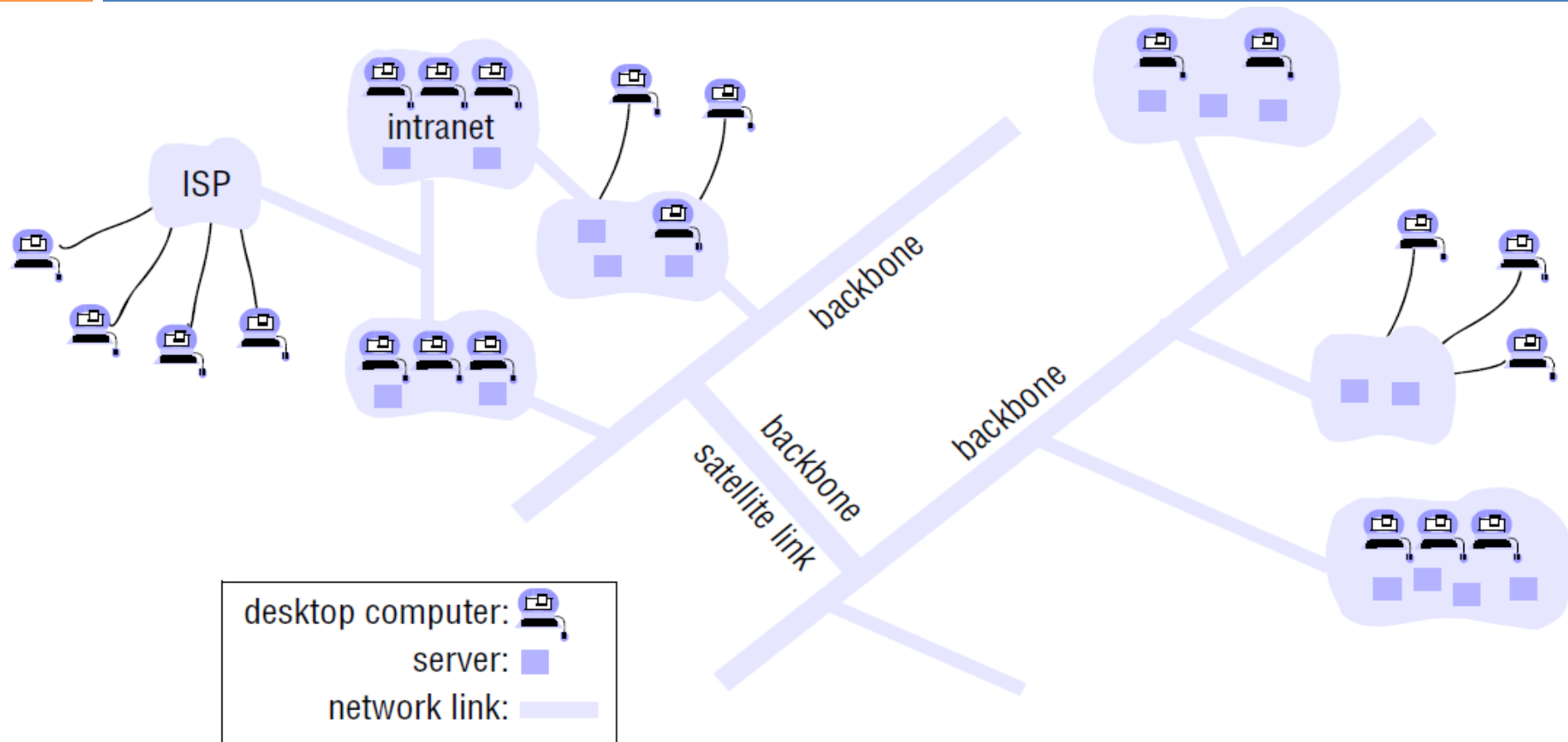
Originalmente, o termo é usado para se referir a algoritmos projetados para execução em processadores “distribuídos” sobre uma grande área geográfica. Mas ao longo dos anos, o uso do termo ampliou-se, incluindo algoritmos que executam em redes locais e mesmo algoritmos para multiprocessadores de memória compartilhada.
(Lynch, 1996)

Pontos: **Sistemas distribuídos baseados em troca de mensagens e em memória compartilhada** (os últimos não são abordados neste curso).

Principais implicações das definições

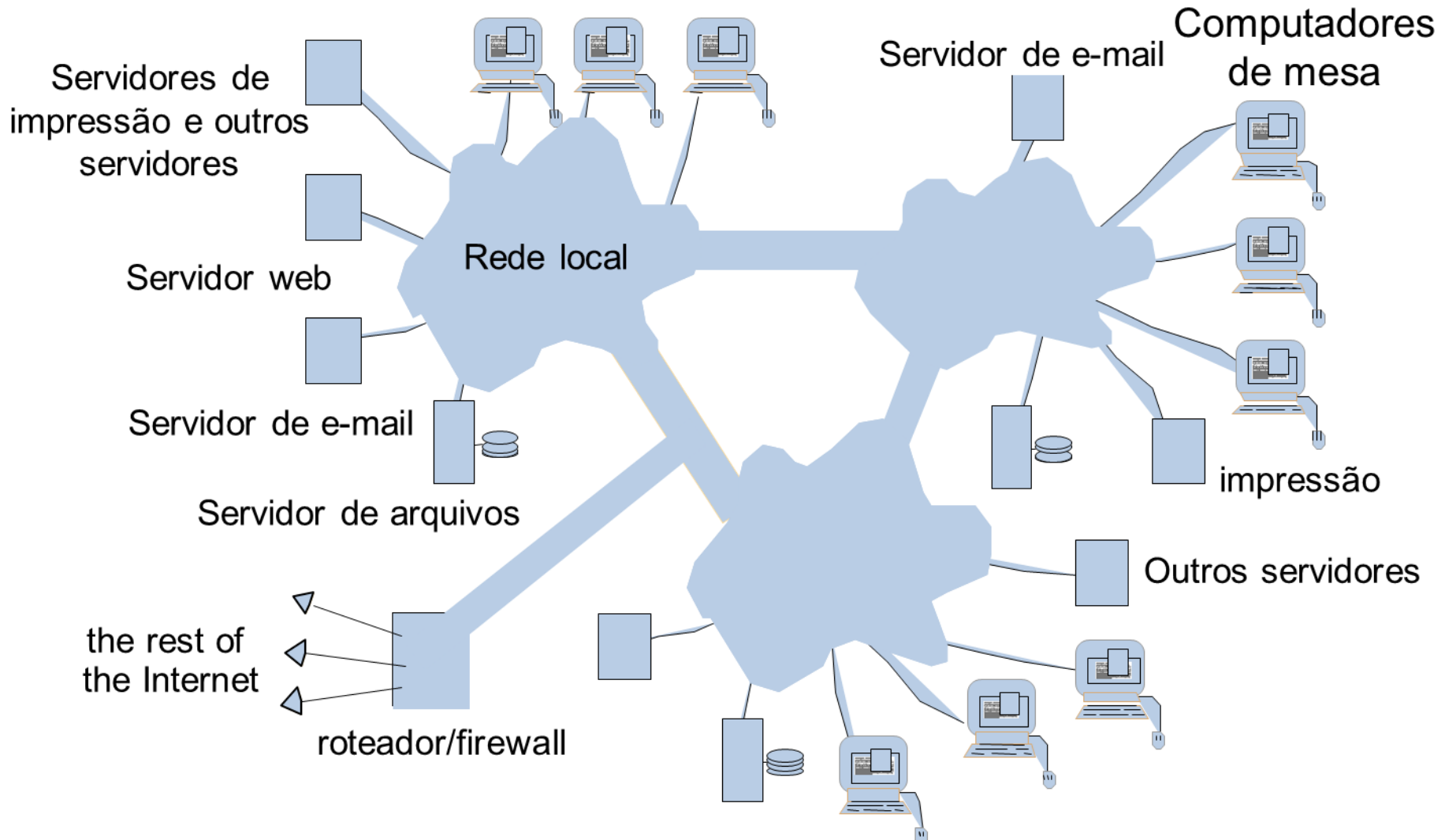
- ❖ Concorrência: Execução de programas (**processos**) de forma concorrente com possível acesso a recursos compartilhados.
- ❖ Ausência de relógio global: Os processos cooperam por meio de troca de mensagens e assim necessitam de uma noção compartilhada de tempo, embora não seja possível utilizar um relógio único. A depender do ambiente, relógios podem ser sincronizados entre si, mas há limites de precisão (**distorção entre relógios**) obtida pela sincronização.
- ❖ Falhas independentes: Qualquer componente do sistema pode falhar. As falhas podem ocorrer de forma independente (**sem correlação entre si**), se a infraestrutura subjacente (e.g. nós processadores, sistema de alimentação, rede etc.) for independente. Falhas não são imediatamente percebidas pelos demais componentes.

Exemplos: Internet



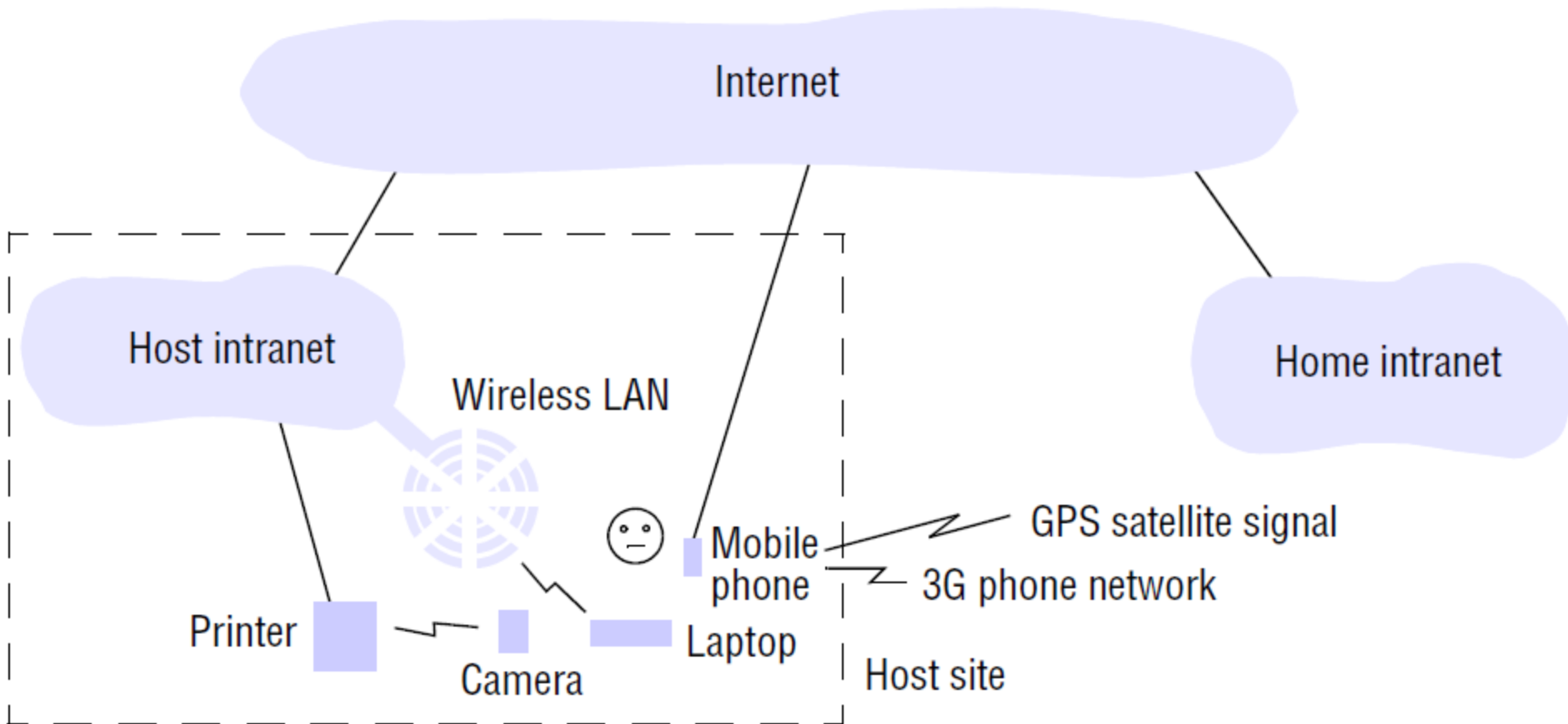
Internet: O conjunto de redes interconectadas que compõe a Internet forma um grande sistema distribuído, onde diversos serviços (como o acesso a Web, mantido por diferentes servidores Web com acesso pelo sistema de nomes DNS) é mantido de forma cooperativa por vários servidores que cooperam entre si.

Exemplos: Intranet



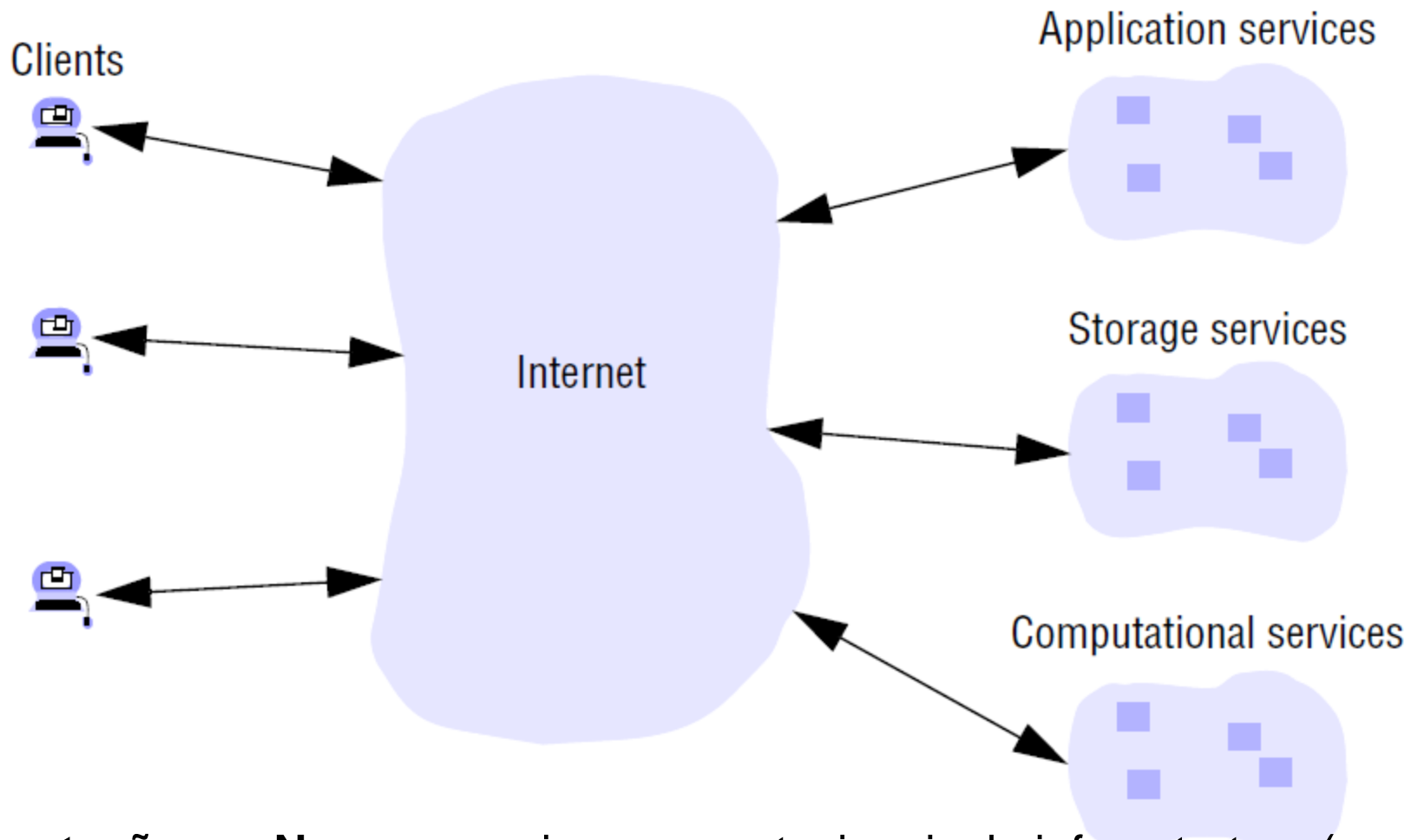
Intranet: Rede local de organização, mantida similar a Internet com servidores locais de serviços e com conexão mediada por um firewall

Exemplos: Computação Ubíqua



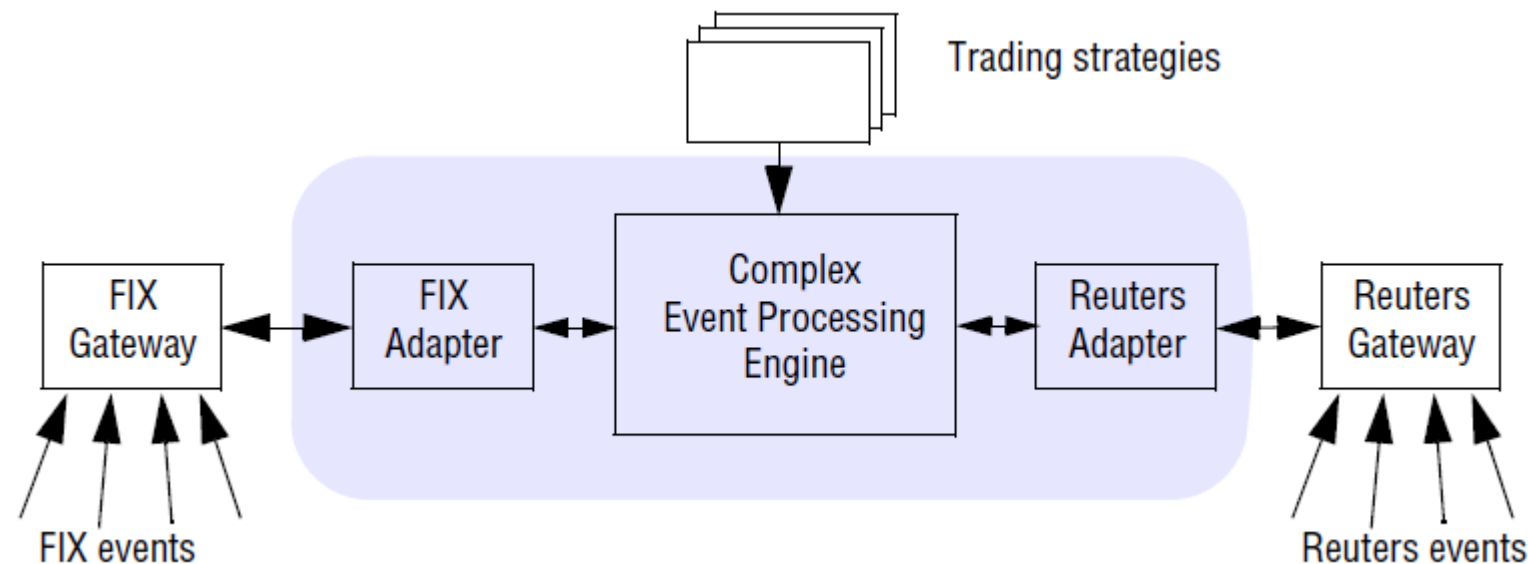
Computação ubíqua e pervasiva: utilização de dispositivos computacionais pequenos e baratos, presentes nos ambientes físicos dos usuários ou incorporados em aparelhos do dia a dia, com interação entre si e uso de informações do contexto.

Exemplos: Computação em Nuvem



Computação em Nuvem: serviços computacionais de infraestrutura (como armazenamento ou computação), plataforma ou aplicação são mantidos em estruturas denominadas nuvens acessíveis através da Internet e com alocação de recursos sob demanda.

Caso: Sistema Financeiro



Event-Based System: eventos podem determinar a execução de ações de forma pré-determinada (sistemas conversam com sistemas, com gateways para adaptação de formatos de eventos e operações).

O mercado de ações funciona nesta lógica e mesmo um pequeno investidor pode instruir as ações ao seu home-broker.

WHEN

MSFT price moves outside 2% of MSFT Moving Average

FOLLOWED-BY (

MyBasket moves up by 0.5%

AND

HPQ's price moves up by 5%

OR

MSFT's price moves down by 2%

)

)

ALL WITHIN

any 2 minute time period

THEN

BUY MSFT

SELL HPQ

Desafios

□ Heterogeneidade:

- O acesso a serviços e a execução de aplicativos é feito através de um conjunto heterogêneo (variado e de diferentes tipos) de componentes:
 - Redes, hardware de computador, sistemas operacionais, linguagens de programação, implementações de diferentes desenvolvedores.
 - O ambiente de execução de um nó é dito plataforma e inclui a arquitetura de computador, e o sistema operacional e conjunto de bibliotecas utilizada, por exemplo, linux-amd64-glibc2 ou windows-x86-glibc.
- Em muitas situações, há mecanismos de interoperação comum:
 - Todos os nós de um sistema distribuído baseado na Internet utiliza o conjunto de protocolos da Internet para a comunicação entre si.

Desafios

- A heterogeneidade implica em diferentes mecanismos e formatos de dados:
 - Tipos de dados podem ser armazenados de formas diferentes, o que requer negociar formatos padronizados para a comunicação.
 - Linguagens de programação diferentes também utilizam diferentes formatos para armazenar estruturas de dados.
 - Chamadas para comunicação podem ser diferentes em sistemas operacionais distintos, como Windows e Linux, o que pode ser um desafio em ambientes multiplataforma.

Desafios

- Abordagens para heterogeneidade:
 - **Middleware**: camada de software que provêm uma abstração determinando um **modo de programação** que pode ser utilizado para mascarar a heterogeneidade de plataformas
 - Diferentes middlewares serão abordados no curso.
 - Em ambientes com migração de código (**código móvel**, que pode ser enviado de um computador para outro, não necessariamente de mesma plataforma - *applets*), uma máquina virtual pode ser necessária para prover o ambiente de execução do código móvel, provendo inclusive isolamento do ambiente local.

Desafios

- Sistemas abertos
 - Pode ser estendido e reimplementado de várias maneiras;
 - Definido na especificação e documentação – principais interfaces são publicadas;
 - Entender a complexidade dos sistemas distribuídos, formados por muitos componentes;
 - Padronização de protocolos
 - Exemplo para a Internet – RFCs;
 - Sistemas distribuídos abertos – projetados a partir de padrões públicos; podem ser construídos a partir de software e hardware heterogêneos.

Desafios

- Aspectos de segurança (*security*) devem ser considerados:
 - Confidencialidade – proteção contra exposição para pessoas não autorizadas;
 - Integridade – proteção contra alteração ou dano;
 - Disponibilidade – proteção contra interferência com os meios de acesso aos recursos;
 - Autenticidade – garantia de identidade de quem acessa.

Desafios

- Segurança (cont.) -
 - Firewall – restringe o acesso de ou para a Internet, mas não garante o correto uso dos recursos da Intranet nem da Internet;
 - Autenticação pode ser feita usando criptografia;
 - Ataque de negação de serviço (*Denial of Service*) – um usuário interrompe um serviço:
 - *O serviço é bombardeado com um grande número de pedidos sem sentido, tornando o serviço indisponível;*
 - Segurança de código móvel – como garantir que o código recebido seja seguro? Uso de *sandbox ou maquina virtual*.

Desafios

- Segurança no funcionamento (*safety*): o algoritmo executa como especificado.
- Terminação (*liveness*): o algoritmo executa a termo (termina em algum momento).
- Os requisitos de *safety* e *liveness* são aspectos importantes na construção de um algoritmo para sistemas distribuídos.

Desafios

□ Escalabilidade –

- O sistema distribuído permanece eficiente quando há um aumento significativo no número de recursos e no número de usuários.

<i>Date</i>	<i>Computers</i>	<i>Web servers</i>
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866
2003, Jan.	171,638,297	35,424,956

Computadores (com endereços IP registrados) na Internet.

Desafios

- Escalabilidade (cont.) –
 - Controlar o custo dos recursos físicos – ampliar a disponibilidade de recursos no sistema, a um custo razoável, à medida em que a demanda aumenta (ex: *cloud computing*);
 - Controlar a perda de desempenho com a maior complexidade na gerência de mais recursos e usuários;
 - Impedir que os recursos de software se esgotem – pouca disponibilidade de endereços IP de 32 bits – IPv6 128 bits;
 - Evitar gargalos de desempenho – descentralizar os algoritmos – DNS!

Desafios

- Tratamento de falhas –
 - Componentes (software e hardware) de um sistema distribuído podem falhar, podendo gerar erros nos resultados do sistema.
 - Técnicas:
 - Detecção de falhas: algumas falhas podem ser detectadas (ex. paridade); gerenciar falhas que não podem ser detectadas mas que podem ser suspeitas (sistemas assíncronos);
 - Mascaramento de falhas: ocultar falhas (ex. retransmitir mensagens perdidas/ replicar dados); nem sempre é possível;

Desafios

- Tratamento de falhas (cont.) –
 - Técnicas (cont.):
 - Tolerância a falhas: executar os serviços apesar das falhas; informar os usuários das falhas nos serviços;
 - Recuperação de falhas: recuperar um estado consistente após uma falha;
 - Redundância: componentes redundantes (ex. rotas diferentes entre dois roteadores na Internet; no DNS toda tabela de nomes é replicada em pelo menos dois servidores; replicar bancos de dados em diversos servidores);
 - Sistemas distribuídos podem fornecer alto grau de disponibilidade de acordo com o projeto do sistema.

Desafios

- Concorrência –
 - Solicitações concorrentes a recursos compartilhados devem ser atendidas de forma concorrente, sem gerar conflitos ou resultados inconsistentes;
 - Utilizar diferentes fluxos de execução (*threads*) nos servidores para atender a cada solicitação;
 - Sincronização da execução concorrente: uso de técnicas como semáforos ou monitores.

Desafios

- **Transparências:**
 - *Transparência de Acesso:* permite que recursos remotos e locais sejam acessados utilizando operações idênticas (ex. sistema de arquivos)/ocultar diferenças em representação de dados e o modo como os recursos podem ser acessados por usuários – diferenças entre arquiteturas de máquinas (máquinas + sistema operacional);
 - *Transparência de localização:* permite que recursos sejam acessados sem o conhecimento de sua localização física ou da rede (por exemplo, qual prédio ou endereço IP) (ex. nomes de recursos);

Desafios

- Transparências (cont.):
 - *Transparência de concorrência*: permite que vários processos executem concorrentemente, usando recursos compartilhados sem interferência entre eles;
 - *Transparência de replicação*: permite que várias instâncias dos recursos sejam usadas para aumentar o desempenho e a confiabilidade, sem conhecimento das réplicas por parte dos usuários ou dos programadores de aplicativos;

Desafios

- Transparências (cont.):
 - *Transparência de falhas*: permite a ocultação de falhas, possibilitando que usuários e programas aplicativos concluam suas tarefas, apesar da falha de componentes de hardware ou software (ex. correio eletrônico – mensagens são entregues mesmo que falhas ocorram);
 - *Transparência de mobilidade*: permite a movimentação de recursos e clientes em um sistema, sem afetar a operação de programas ou usuários (telefones móveis);

Desafios

- Transparências (cont.):
 - *Transparência de desempenho*: permite que o sistema seja reconfigurado para melhorar o desempenho à medida que as cargas variam;
 - *Transparência de escala*: permite que os sistemas e aplicativos se expandam em escala, sem alterar a estrutura do sistema ou os algoritmos de aplicativo.

SISTEMAS OPERACIONAIS



Introdução

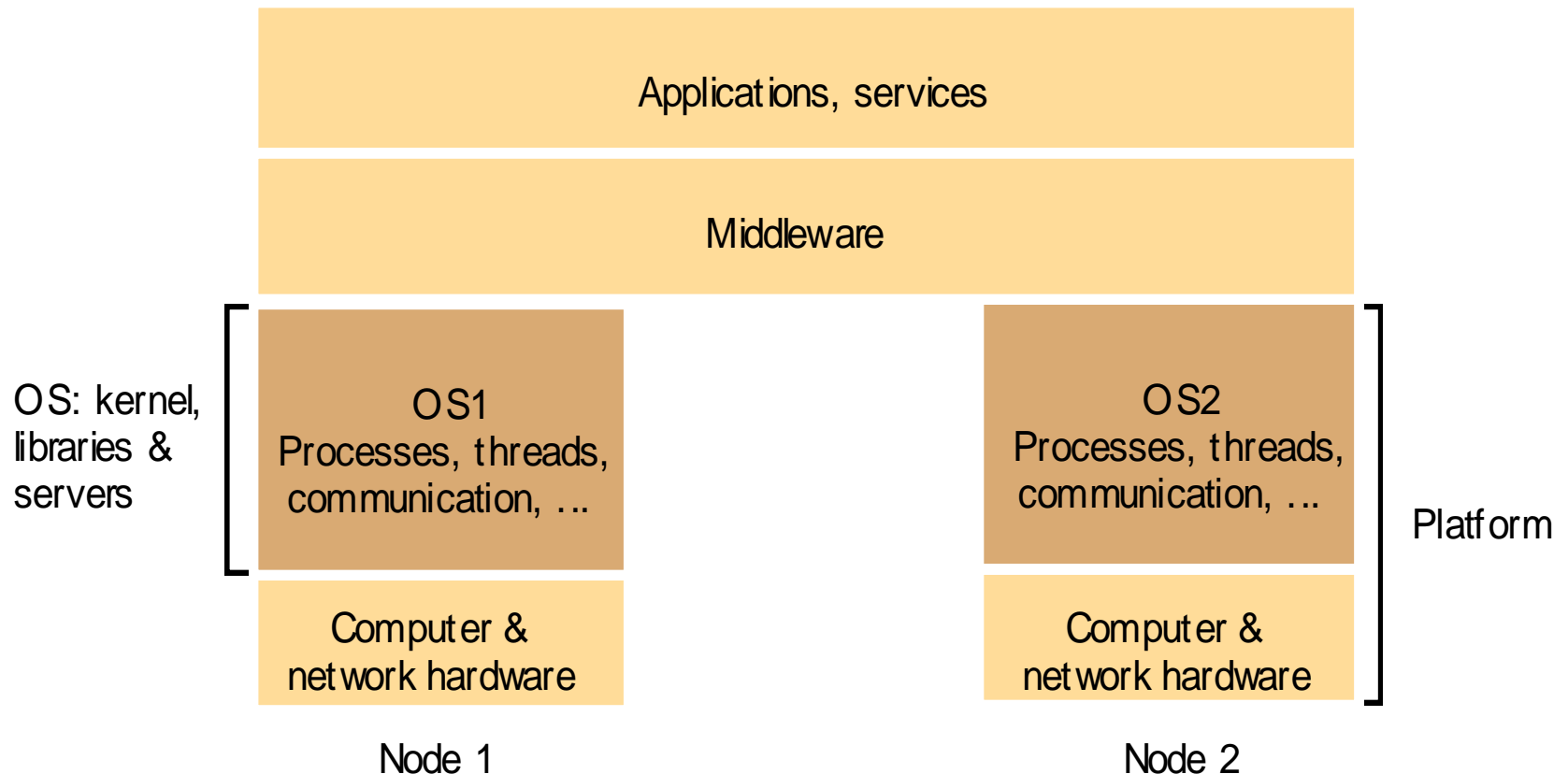
- O Sistema Operacional provê uma máquina estendida, que abstrai os detalhes da plataforma para as aplicações usuárias, e gerencia os recursos a serem utilizados por esta, e.g. processamento, armazenamento e comunicação.

Introdução

Diferentes abordagens de sistema operacional são possíveis em um cenário de sistemas distribuídos:

- Sistema operacional de rede – incorporam no núcleo recursos de interligação em rede.
 - ▣ Cada nó gerencia os seus recursos.
- Sistema Operacional Distribuído –
 - ▣ SO gerencia os recursos de todos os nós do sistema – usuário executa aplicações sem conhecer a localização dos recursos sendo utilizados.
- Middleware – uma abstração determinando um **modo de programação** que abstrai os detalhes de chamadas do sistema operacional para o uso de recursos da plataforma. Por exemplo, fornecendo invocações remotas entre objetos (RMI) nos nós de um sistema distribuído, utilizando sockets TCP/IP providos pelo S.O.

Visão da Aplicação Baseada em Middleware



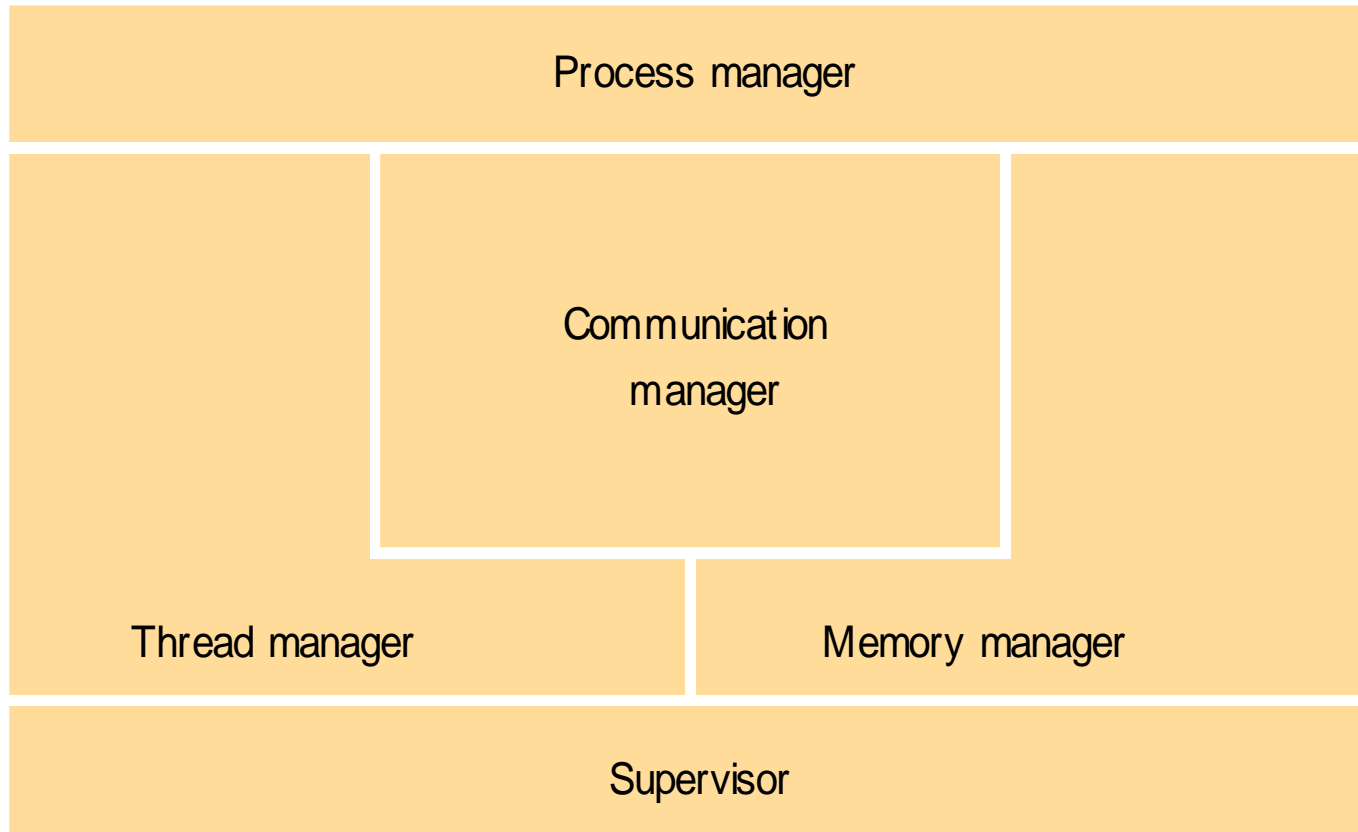
Camadas de Sistema Operacional

- Os serviços do S.O. são providos pelo seu núcleo e por processos servidores – normalmente implementados por meio de bibliotecas, observando-se:
 - ▣ **Encapsulamento** (interface de serviço útil para acesso aos recursos)
 - ▣ **Proteção** (impedir acessos ilegítimos)
 - ▣ **Acesso Concorrente** (gerenciar de forma transparente o acesso concorrente a recursos por diferentes processos ou linhas de execução)

Núcleo do Sistema Operacional

- ❑ Gerenciador de processos – criação de processos;
- ❑ Gerenciador de threads – criação, sincronização e escalonamento de threads;
- ❑ Gerenciador de comunicação – comunicação entre threads no mesmo nó;
- ❑ Gerenciador de memória – memória física e virtual;
- ❑ Supervisor – interrupções, chamadas de sistema e exceções.

Núcleo do Sistema Operacional



Camadas de Sistema Operacional

- Um *middleware* abstrai as interfaces relacionadas aos serviços, provendo novas interfaces e modos de programação.
- Por exemplo, um *middleware* de invocação remota abstrai:
 - ▣ Serviços de Comunicação – Uso de sockets para passagem de parâmetros e resultados de operações;
 - ▣ Serviços de Escalonamento – Agendamento da execução de uma operação invocada.

Proteção

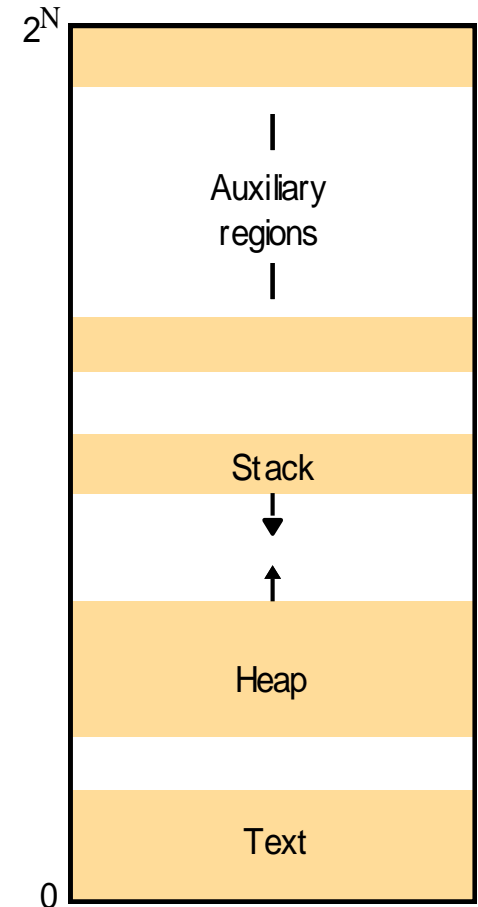
- Acesso ilegítimo (erro ou malícia) –
 - Invocação ilegítima – usar linguagem fortemente tipada, como Java ou Modula-3.
- Núcleo –
 - Modo de execução – supervisor (privilegiado) e usuário (não privilegiado);
 - Espaços de endereçamento para cada processo – conjunto de intervalos de posições de memória virtual nos quais se aplicam combinações de direitos de acesso.
 - Interrupções e chamadas ao sistema – chaveamento para modo supervisor.

Processos e Threads

- Processo – programa em execução
 - ▣ Ambiente de execução
 - Espaço de endereçamento / recursos de sincronização e comunicação entre threads (semáforos, soquetes, etc.) / recursos como arquivos e janelas (abstrações);
 - Proteção às threads do processo;
 - ▣ Threads – abstração para a execução de uma atividade – fluxo ou linha de execução
 - Threads compartilham recursos do ambiente de execução.

Processos e Threads

- Espaços de endereçamento
 - ▣ Extensão – menor endereço virtual e tamanho;
 - ▣ Permissões (leitura/gravação/execução) para as threads do processo;
 - ▣ Capacidade de crescimento.
- Paginação / segmentação /
paginação com segmentação.



Processos e Threads

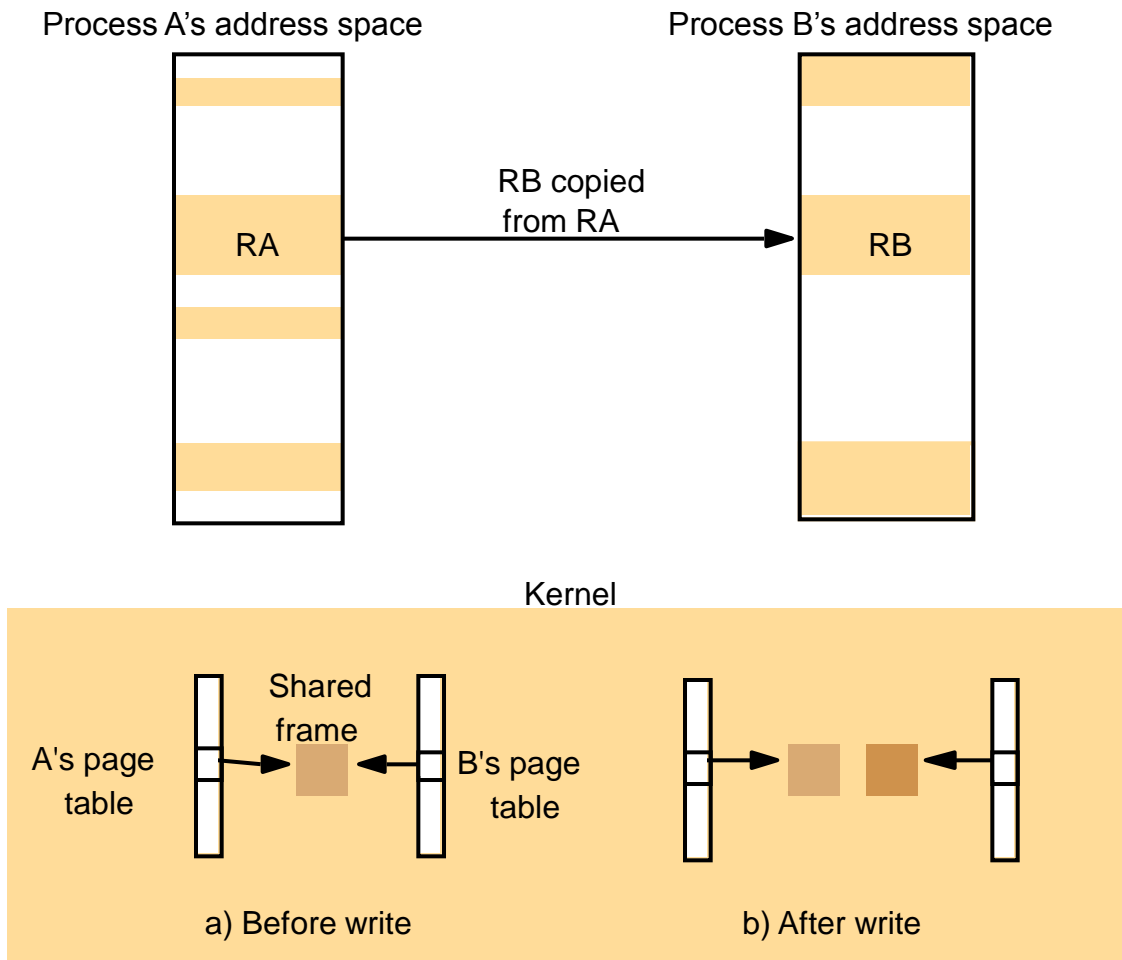
- Espaços de endereçamento (cont.)
 - Dividido em regiões – ex. texto, dados e pilha;
 - Memória compartilhada entre processos – Regiões compartilhadas
 - Bibliotecas – código compartilhado;
 - Dados (comunicação) –
 - Núcleo – evita trocar a tabela de páginas quando ocorre uma exceção.

Processos (*fork*) e Threads

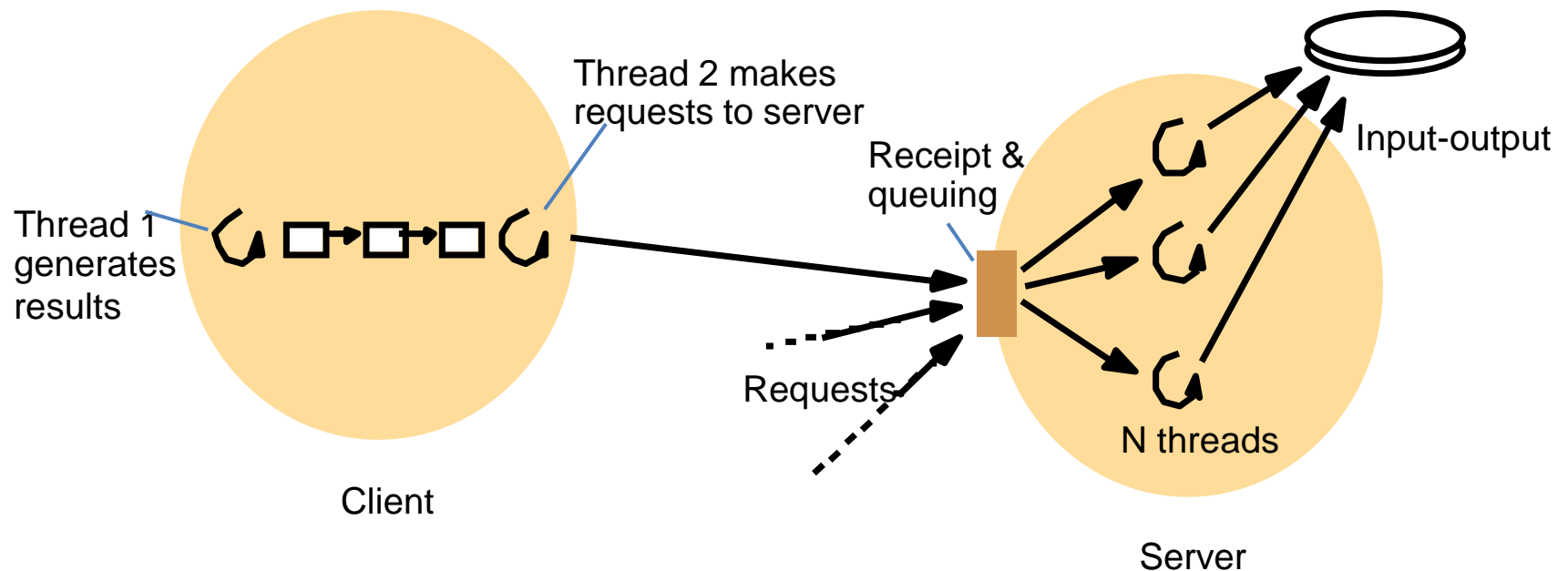
- Criação de um novo processo em um sistema distribuído –
 - Escolher o host destino – emprego de políticas – balanceamento de carga, recursos disponíveis e aspectos da arquitetura;
 - Criação de um ambiente de execução e de um processo novo a partir de um existente –
 - Unix – ambiente de execução é copiado (no Mach e Chorus - frames são compartilhados e usa abordagem *copy on write*) para o processo filho criado (*fork*);
 - Ao se utilizar Threads, se compartilha o ambiente de execução!

Processos (fork)

- Ao se realizar um *fork* de processo (ex: processo pai A faz fork para filho B) pode-se manter a referência a região até a escrita.
- Cópia na escrita (*Copy on Write*)
- Quadros são substituídos quando são modificados.

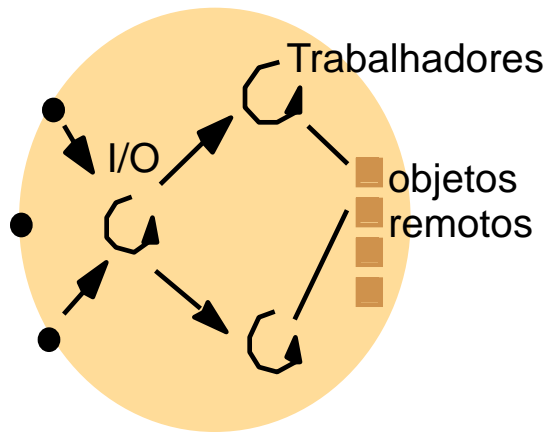


Threads

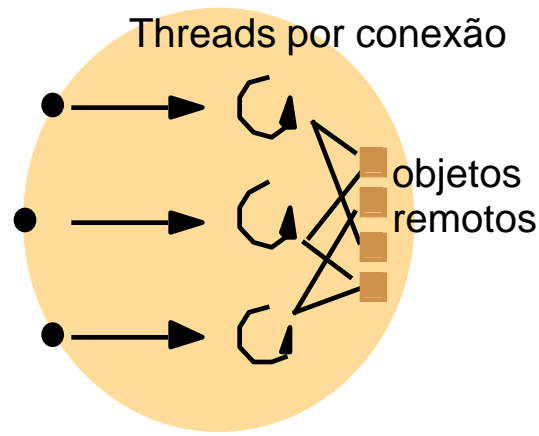


Cliente e Servidores com threads. O uso de threads aumenta a throughput (taxa de rendimento) dos servidores.

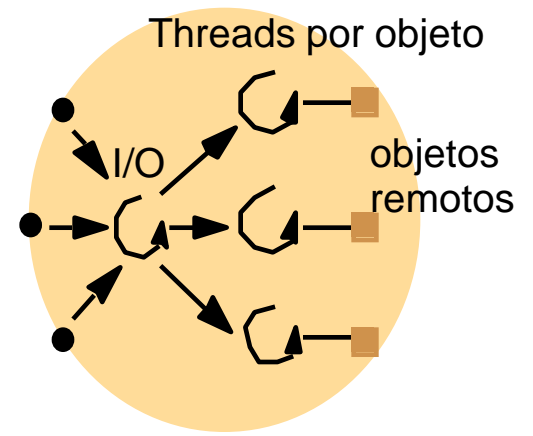
Processos e Threads



a. Thread por pedido



b. Thread por conexão



c. Thread por objeto

Cliente com múltiplas threads

- Clientes com múltiplas threads tornam a comunicação com o servidor transparente.
- Enquanto 1 thread espera a resposta do servidor (estando bloqueada), as demais threads continuam a execução.
- É possível construir um browser de tal forma que cada arquivo componente de uma página é recebido por uma thread diferente. Ao receber um arquivo a thread começa a mostrá-lo na tela, enquanto outras threads esperam pelos demais arquivos, em diferentes conexões com o servidor.

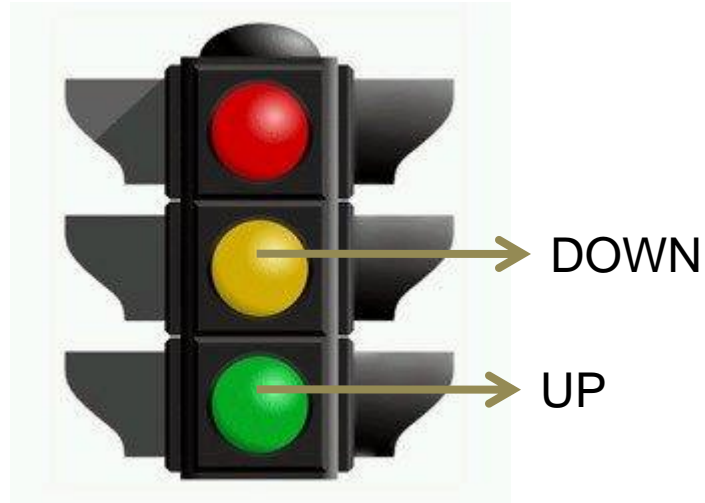
Sincronização de Threads

- Threads possuem acesso a mesma região da memória (região crítica)
- Problema de acesso a região crítica: bloquear acesso, acessar e liberar novamente o acesso
- Uso de variável mutex de acesso a região crítica:
 - ▣ Uso de TSL: Test, Set and Lock
- Uso de semáforos

Sincronização de Threads

□ Semáforos:

```
Semaphore S = 1;  
DOWN(S);  
CriticalSection();  
UP(S);
```



- DOWN(S): Se $S > 1$, $S--$, e acessa região, senão aguarda
- UP(S): $S++$, acorda um processo que aguarda para DOWN

Sincronização de Threads

- Barreiras: pontos de sincronização entre threads
 - ▣ Um conjunto de threads deve alcançar o ponto de sincronização (Barreira) antes de prosseguir na computação.
- Problemas clássicos:
 - ▣ Produtor-Consumidor
 - ▣ Barbeiro
 - ▣ Jantar dos Filósofos
 - ▣ The Little Book of Semaphores

Métodos de construção e gerenciamento de threads-Java

Thread(ThreadGroup group, Runnable target, String name)

Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()

Set and return the thread's priority.

run()

A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()

Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)

Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()

Enter the *READY* state and invoke the scheduler.

destroy()

Destroy the thread.

Chamadas de sincronização de threads - Java

thread.join(int millisecs)

Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), *object.notifyAll()*

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Hands on



- Produtor Consumidor MultiThread
- Produtor Consumidor com Semáforos

COMUNICAÇÃO INTERPROCESSOS



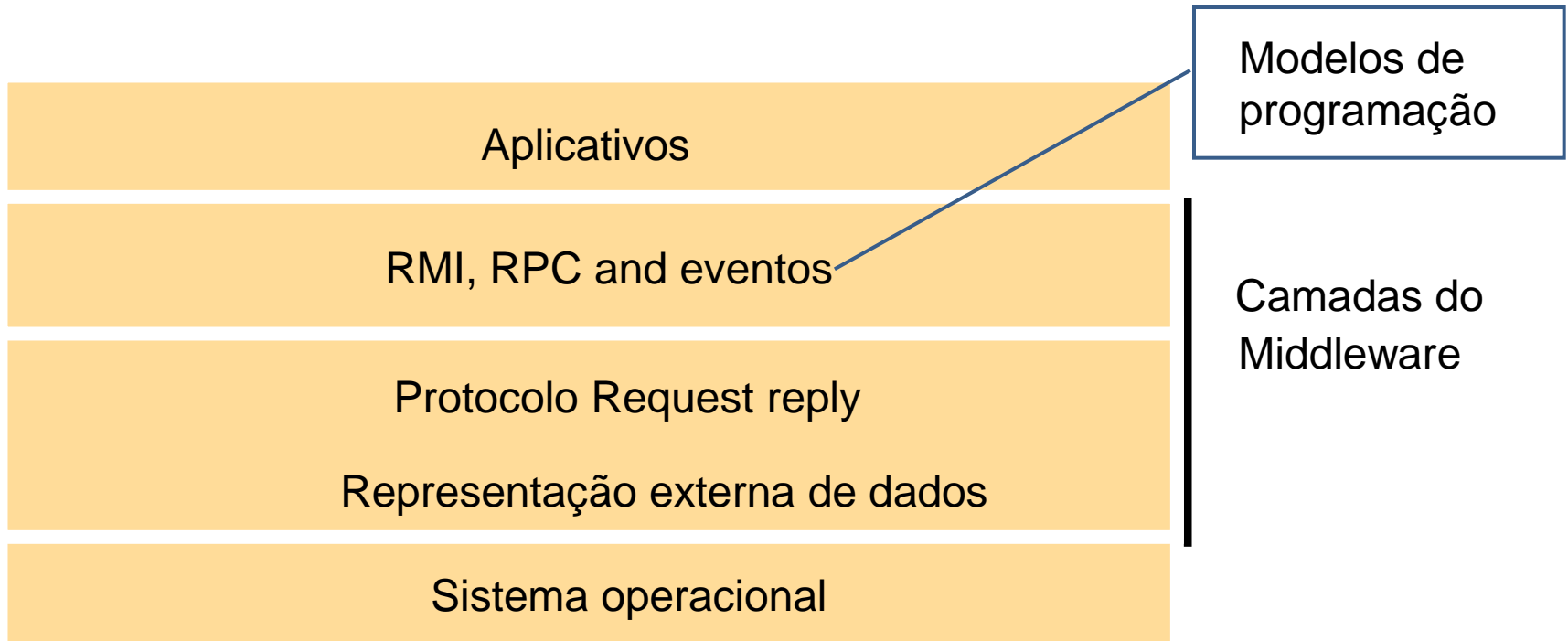
Introdução

- Em sistemas centralizados, a comunicação interprocessos se dá por chamadas a funções do processo, sob gerência do Sistema Operacional, ou por memória compartilhada
- Em sistemas distribuídos, não há um SO único que permite acesso, nem os processos podem utilizar memória compartilhada

Introdução

- Podemos utilizar troca de mensagens diretamente:
Programação de Sockets
- Ou prover uma abstração de chamada de funções do processo ou invocação de método de um objeto que abstraia os detalhes dos sockets
 - ▣ Ex: RPC (procedural), RMI (objeto) e CORBA (objeto)
 - ▣ Encapsulamento/desencapsulamento de parâmetros e resposta (“Marshalling”/ “Unmarshalling”)

Introdução



- Middleware - software que fornece um modelo de programação acima dos blocos de construção básicos de processos e de passagem de mensagens.
 - ▣ Fornece abstração de alto nível para a comunicação entre processos.

Middleware de IPC

- Transparência de localização – cliente desconhece a localização do procedimento chamado por RPC ou do método invocado por RMI – objetos não conhecem a localização dos objetos que geram ou recebem eventos;
- Protocolos de comunicação – protocolos que suportam as abstrações do middleware independem dos protocolos de transporte (TCP ou UDP);

Middleware de IPC

- Hardware de computador – padrões de empacotamento de mensagens ocultam as diferenças entre as arquiteturas de hardware.
- Sistemas operacionais – abstrações do middleware independem do SO;
- Linguagens de programação – alguns middlewares, como o CORBA, permitem que os aplicativos utilizem diversas linguagens.

Interfaces para IPC

- Especificam os procedimentos e as variáveis que podem ser acessadas a partir de outros módulos.
- Sistemas Distribuídos
 - Não se pode acessar diretamente dados remotos;
 - Parâmetros não podem ser passados por referência – seu valor é copiado para o módulo remoto, sendo utilizados como argumentos para a operação chamada, e os resultados retornados com resultado da chamada, ou substituindo valores das variáveis de chamada.

Interfaces para IPC

- Interfaces de serviço – procedimentos oferecidos por um servidor, disponíveis para uso pelos leitores;
- Interfaces remotas – especifica os métodos de um objeto disponíveis para invocação remota;

IDL - Linguagens de definição de interface

```
// In file Person.idl  
struct Person {  
  string name;  
  string place;  
  long year;  
};  
interface PersonList {  
  readonly attribute string listname;  
  void addPerson(in Person p) ;  
  void getPerson(in string name, out Person p);  
  long number();  
};
```

IDL do CORBA

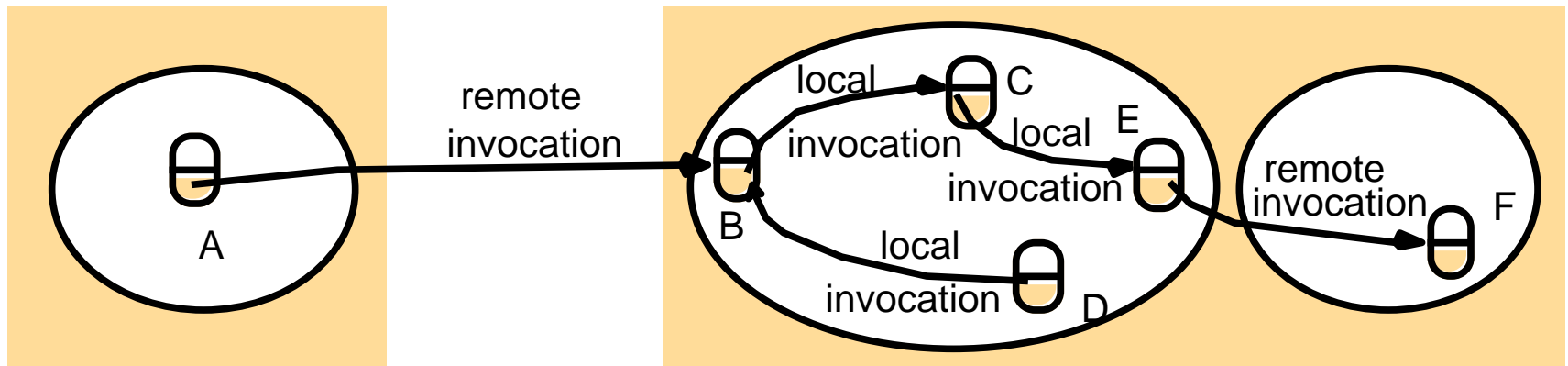
- Permitem que objetos implementados em diferentes linguagens invoquem uns aos outros;

Comunicação entre objetos distribuídos

□ Modelo de objeto –

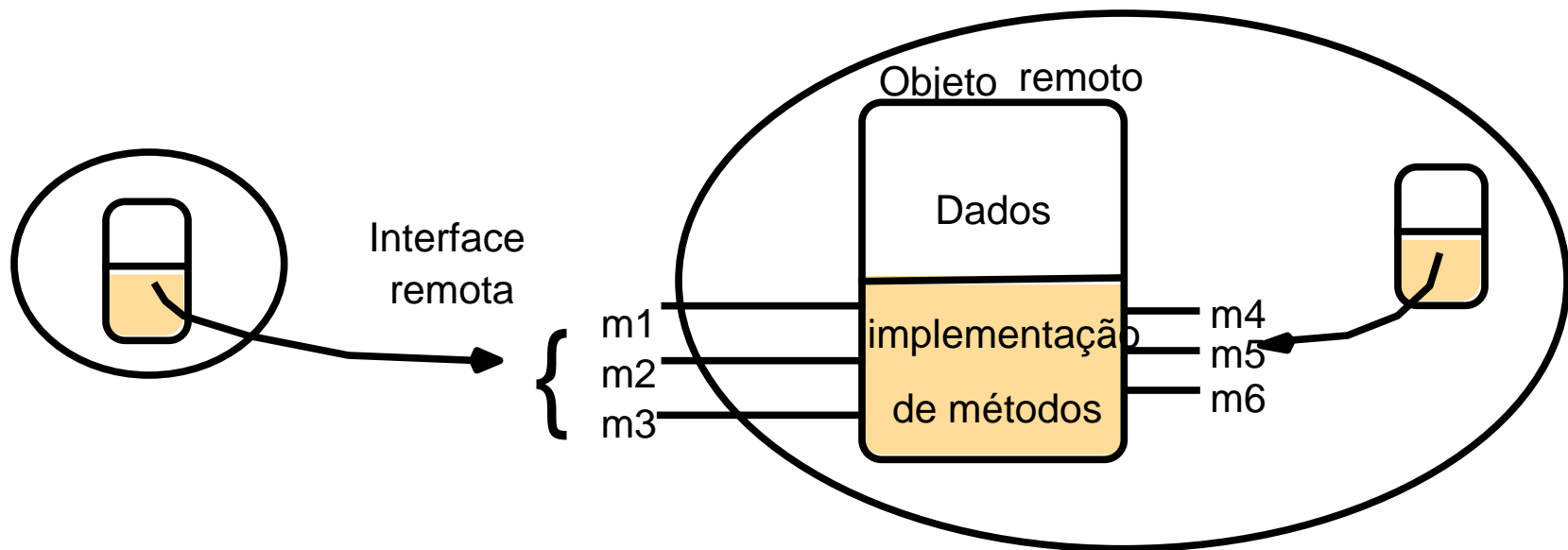
- Referência de objeto – variável de instância – contém uma referência ao objeto;
- Interface – definição das assinaturas de um conjunto de métodos – não especifica a implementação;
- Ações – iniciadas pela invocação de um método;
- Exceções – especificam como tratar condições de erro;
- Coleta de lixo (*garbage collection*) – liberar memória não utilizada – objetos não alcançáveis.

Modelo de objeto distribuído



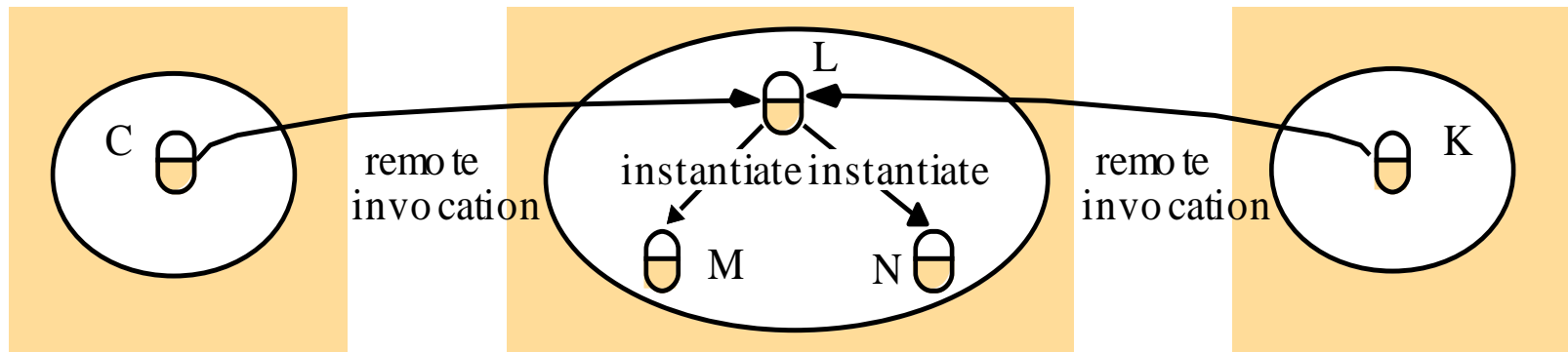
- Referência de objeto remoto – identificador utilizado por todo o sist. distribuído para se referir a um objeto remoto;
 - ▣ Interface do objeto B deve estar disponível para o objeto A;

Modelo de objeto distribuído



- Interfaces remotas – apenas métodos da interface remota podem ser acessados em objetos remotos - localmente se pode invocar todos os métodos do objeto;

Modelo de objeto distribuído



- Ações – iniciada pela invocação a um método – a instanciação de um objeto é feita no processo remoto;

Remote Method Invocation

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;      1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;  2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

- Interfaces remotas *Shape* e *ShapeList* – definidas por herança da classe *Remote* – métodos disparam a exceção *Remote Exception*;

Remote Method Invocation

□ RMIregistry –

- Vinculador da RMI Java – uma instância de RMIregistry deve ser executada em cada computador servidor que contenha objetos remotos;
- Tabela mapeia nomes textuais (URLs) em referências para objetos remotos contidos no computador;
- Acessado por métodos da classe *Naming*, passando argumentos *string* (URL)
 - `//nomeComputador:porta/nomeObjeto`

Remote Method Invocation

void rebind (String name, Remote obj)

Este método é usado por um servidor para registrar o identificador de um objeto remoto pelo nome.

void bind (String name, Remote obj)

Este método pode ser usado alternativamente por um servidor para registrar um objeto remoto pelo nome, mas se o nome já estiver vinculado a uma referência de objeto remoto, uma exceção será disparada.

void unbind (String name, Remote obj)

Este método remove um vínculo.

Remote lookup (String name)

Este método é usado pelos clientes para procurar um objeto remoto pelo nome. É retornada uma referência de objeto remoto.

String [] list()

Este método retorna um array de Strings contendo os nomes vinculados no registro. *Class Naming do RMIregistry*

Exemplo de RMI: Programa Servidor

```
import java.rmi.*;
public class ShapeListServer{
public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());
    try{
        ShapeList aShapeList = new ShapeListServant();
        Naming.rebind("Shape List", aShapeList );
        System.out.println("ShapeList server ready");
    }catch(Exception e) {
        System.out.println("ShapeList server main " + e.getMessage())
    }
}
```

- Classe *ShapeListServer* Java com o método *main*.

Exemplo de RMI: Programa Servidor

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;           // contains the list of Shapes      1
    private int version;
    public ShapeListServant() throws RemoteException {...}
    public Shape newShape(GraphicalObject g) throws RemoteException {    2
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }
    public Vector allShapes() throws RemoteException {...}
    public int getVersion() throws RemoteException { ... }
}
```

- A classe *ShapeListServant* Java que implementa a interface *ShapeList*.

Exemplo de RMI: Programa Cliente

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList")    ;
            Vector sList = aShapeList.allShapes();
        } catch(RemoteException e) {System.out.println(e.getMessage());}
        } catch(Exception e) {System.out.println("Client: " + e.getMessage());}
        }
    }
```

□ Cliente Java para *ShapeList*.

ARQUITETURAS DE SISTEMAS DISTRIBUÍDOS



Introdução

- Dificuldades e ameaças para os sistemas distribuídos:
 - ▣ Modos de uso que variam muito
 - As partes componentes dos sistemas estão sujeitas a amplas variações na carga de trabalho
 - Alguns componentes de um sistema podem estar desconectados ou mal conectados em parte do tempo
 - Alguns aplicativos têm requisitos especiais de necessitar grande largura de banda de comunicação e baixa latência

Introdução

- Dificuldades e ameaças para os sistemas distribuídos (cont.):
 - ▣ Ampla variedade de ambientes de sistema
 - Um sistema deve acomodar hardware, sistemas operacionais e redes heterogêneas.
 - As redes podem diferir amplamente no desempenho – as redes sem fio operam a uma taxa de transmissão inferior a das rede locais cabeadas.
 - Os sistemas computacionais podem apresentar ordens de grandeza totalmente diferentes – variando de dezenas até milhões de computadores – devendo ser igualmente suportados.

Introdução

- Dificuldades e ameaças para os sistemas distribuídos (cont.):
 - ▣ Problemas internos
 - Relógios não sincronizados, atualizações conflitantes de dados, diferentes modos de falhas de hardware e de software envolvendo os componentes individuais de um sistema.
 - ▣ Ameaças externas
 - Ataques à integridade e ao sigilo dos dados, negação de serviço, etc.

Arquitetura

□ Arquitetura de sistema

- ▣ Estrutura em termos de componentes especificados separadamente;
- ▣ Tornar o sistema confiável, gerenciável, adaptável e rentável.
- ▣ Distribuição de responsabilidades e atribuição de componentes a computadores.

□ Modelo de arquitetura

- ▣ Posicionamento dos componentes em uma rede de computadores;
- ▣ Inter-relacionamentos entre os componentes.
- ▣ O modelo de Arquitetura abstrai os detalhes de implementação e apresenta uma abstração de arranjo dos componentes

Arquitetura

□ Modelo Cliente-Servidor

- ▣ Processos clientes e processos servidores;

- ▣ Variações

 - Código móvel – reduz atrasos e minimiza o tráfego;

 - Adição e remoção de computadores e dispositivos móveis.

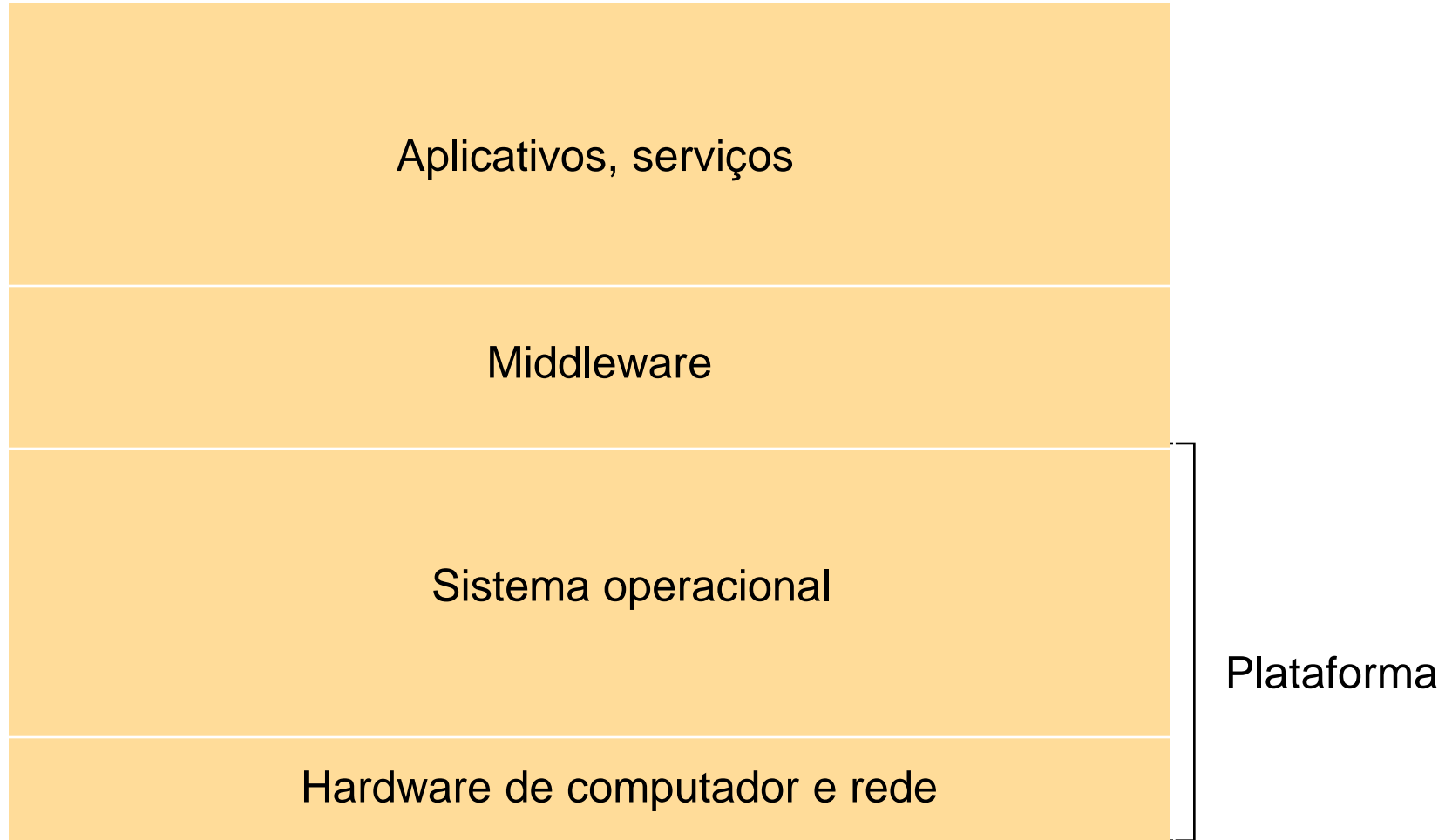
□ Modelo Peer-to-peer

- ▣ Processos peer-to-peer.

Camadas de software

- Estruturação do software em camadas ou módulos em um único computador, ou em termos de serviços oferecidos e solicitados entre processos localizados em um mesmo computador ou em computadores diferentes.
- Servidor – processo que aceita pedidos de outros processos.
- Serviço distribuído – pode ser fornecido por um ou mais processos servidores.

Camadas de software



Camadas de software e hardware em serviços de sistemas distribuídos

Camadas de software

- Plataforma
 - Camadas de hardware e software de mais baixo nível.

Camadas de software

□ Middleware

- Camada de software cujo objetivo é mascarar a heterogeneidade e fornecer um modelo de programação conveniente para os programadores de aplicativos.
- É composto por um conjunto de processos ou objetos em um grupo de computadores.
- Implementa a comunicação e oferece suporte ao compartilhamento de recursos para aplicativos distribuídos.

Camadas de software

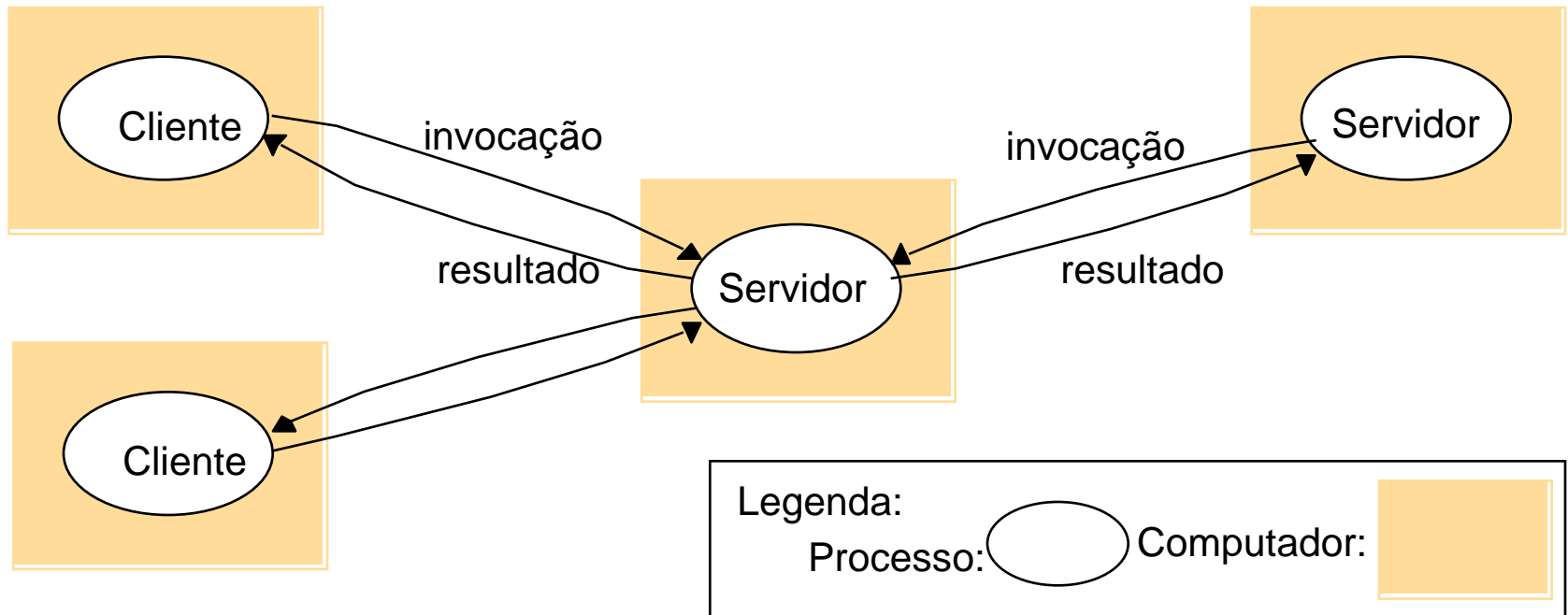
□ Middleware (cont.)

- Fornece uma abstração ao programador de aplicativos para a comunicação entre processos.
- Exemplos:
 - RPC, Isis (sistema de comunicação em grupo), CORBA, RMI Java, serviços web, DCOM (*Distributed Component Object Model*) da Microsoft, RM-ODP (*Reference Modelo for Open Distributed Processing*) do ISO/ITU-T, etc.
- As vezes, soluções mais simples diretamente implementadas nas aplicações, são mais eficientes e adequadas do que utilizar os serviços oferecidos por um middleware.

Cliente-servidor

- Processos clientes interagem com processos servidores, localizados em computadores hospedeiros distintos, para acessar os recursos compartilhados que estes gerenciam.
- Um processo servidor pode ser cliente de outros servidores (ex. servidores web são clientes do serviço DNS).

Cliente-servidor



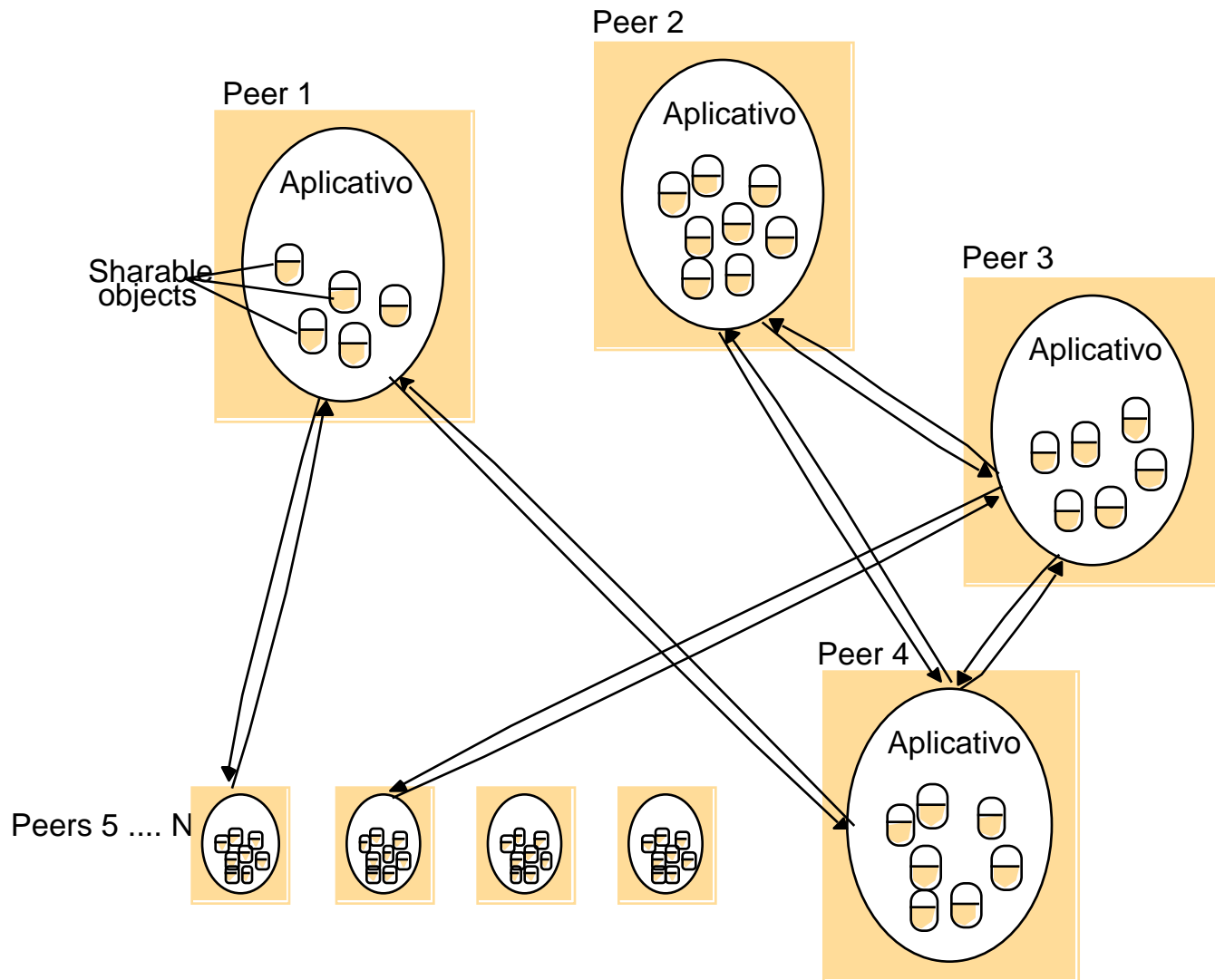
Clientes realizam pedidos a servidores.

Peer-to-peer

- Todos os processos envolvidos em uma tarefa ou atividade desempenham funções semelhantes, interagindo cooperativamente como pares (peers).
- Computadores fornecem acesso a dados e a outros recursos que eles armazenam e gerenciam coletivamente.

Peer-to-peer

Um aplicativo distribuído baseado na arquitetura *peer-to-peer*.

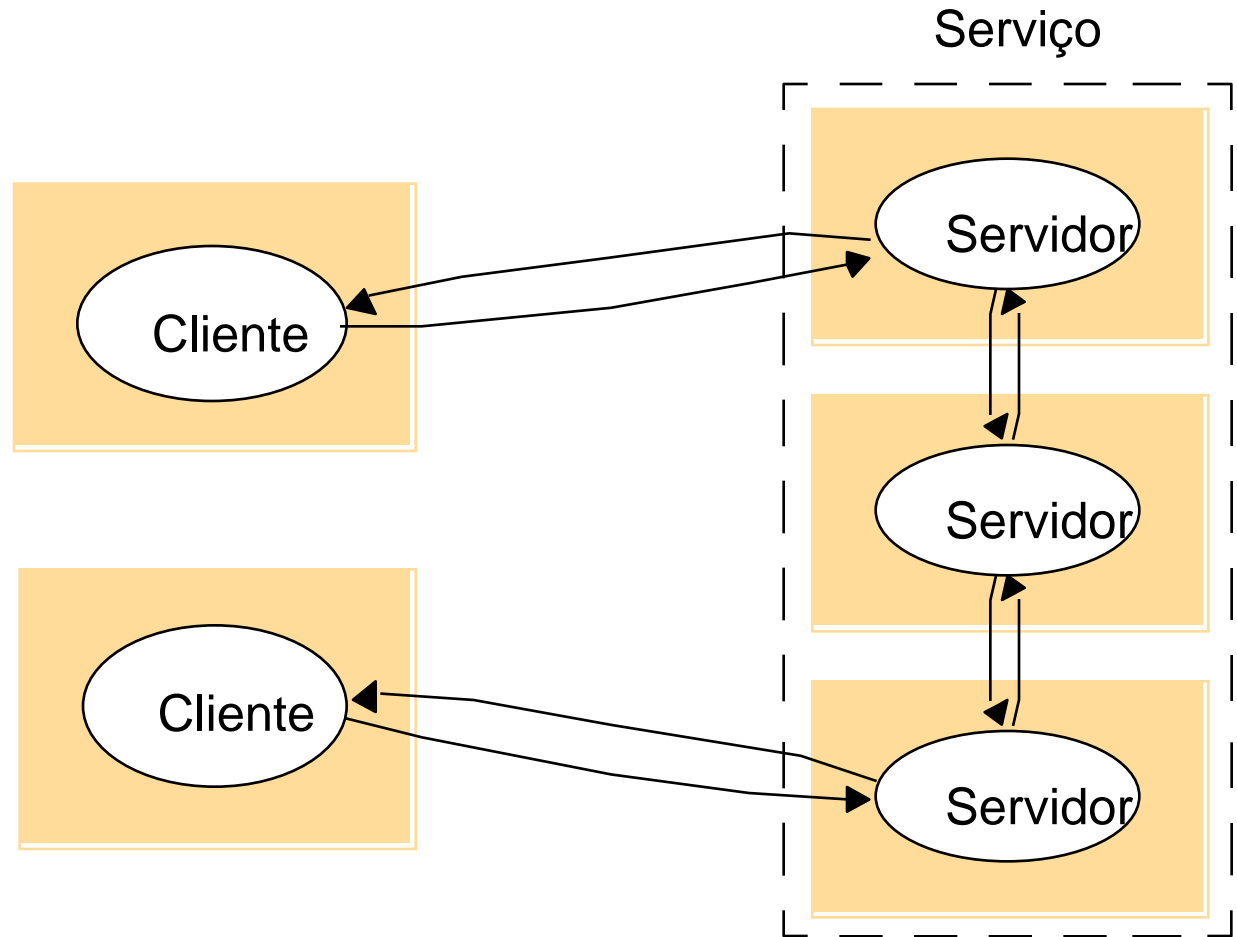


Vários servidores

- Serviços implementados como vários processos servidores em diferentes computadores hospedeiros.
- O conjunto de objetos nos quais o serviço é baseado podem ser particionados entre os servidores, ou duplicados.
 - Particionamento de dados – web;
 - Dados replicados – NIS (*Network Information Service*)
- Cluster – arquitetura com forte interação entre servidores.

Vários servidores

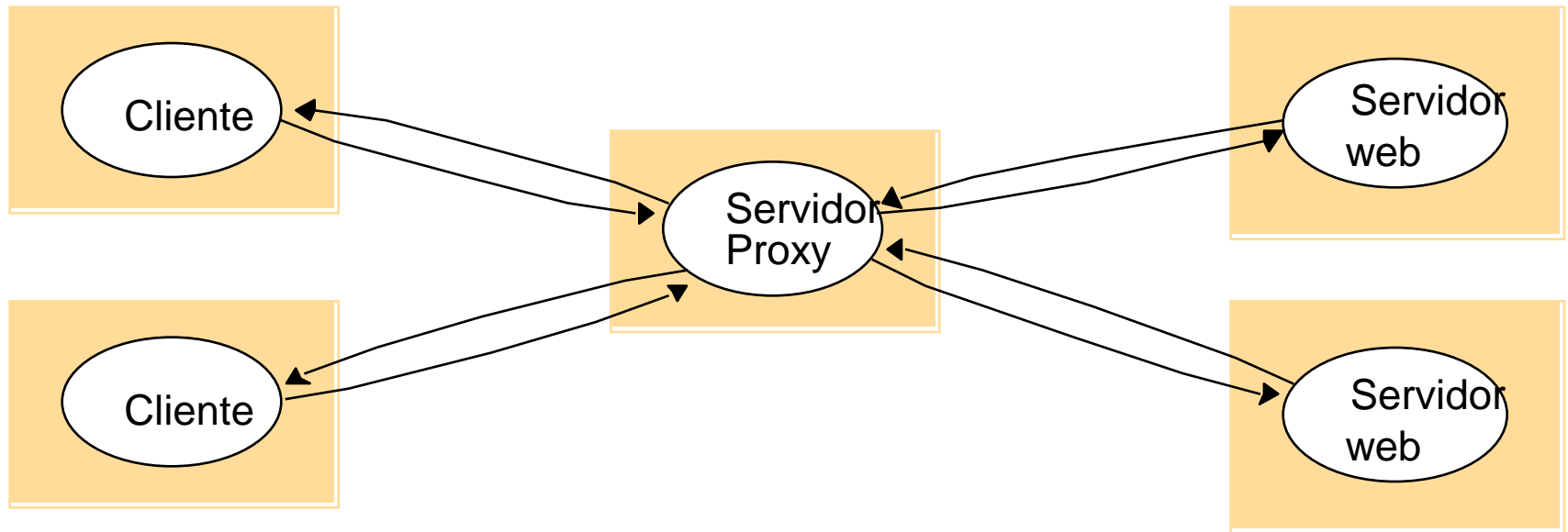
Um serviço
fornecido por
vários servidores



Servidores *proxies* e caches

- Cache – armazenar objetos de dados recentemente usados em um local mais próximo que a origem real dos objetos.
- Acessos posteriores a este objeto são realizados na cache.
- Caches são mantidas nos clientes ou em um servidor *proxy* compartilhado.
- Servidor *proxy* web – fornece uma cache compartilhada de recursos web; aumenta a disponibilidade e o desempenho do serviço.

Servidores *proxies* e caches



Servidor proxy web

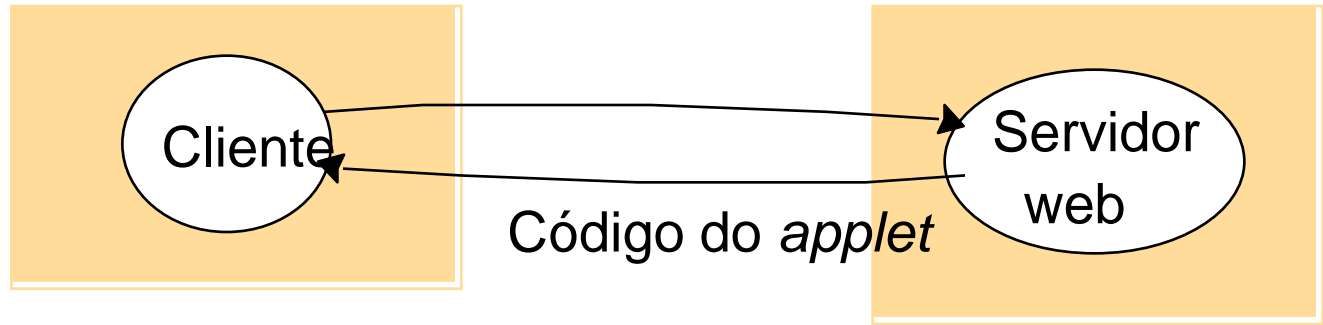
Código móvel

□ *Applets*

- Ficam armazenados em servidores, e são carregados ao selecionar um link em um navegador.
- O código é carregado no navegador e executado posteriormente, localmente.
- Por questões de segurança, os navegadores dão aos *applets* acesso limitados a seus recursos locais.

Código móvel

a) Requisição do cliente resulta no *download* do código *applet*



b) O cliente interage com o *applet*



Applets web

Agentes móveis

- Programa em execução (processo) que passa de um computador para outro em um ambiente de rede.
- Retorna com os resultados para o usuário.
- Realiza requisições de serviços localmente em cada site onde é executado.
- A identidade do solicitante do agente móvel deve ser verificada para não executar código inseguro.

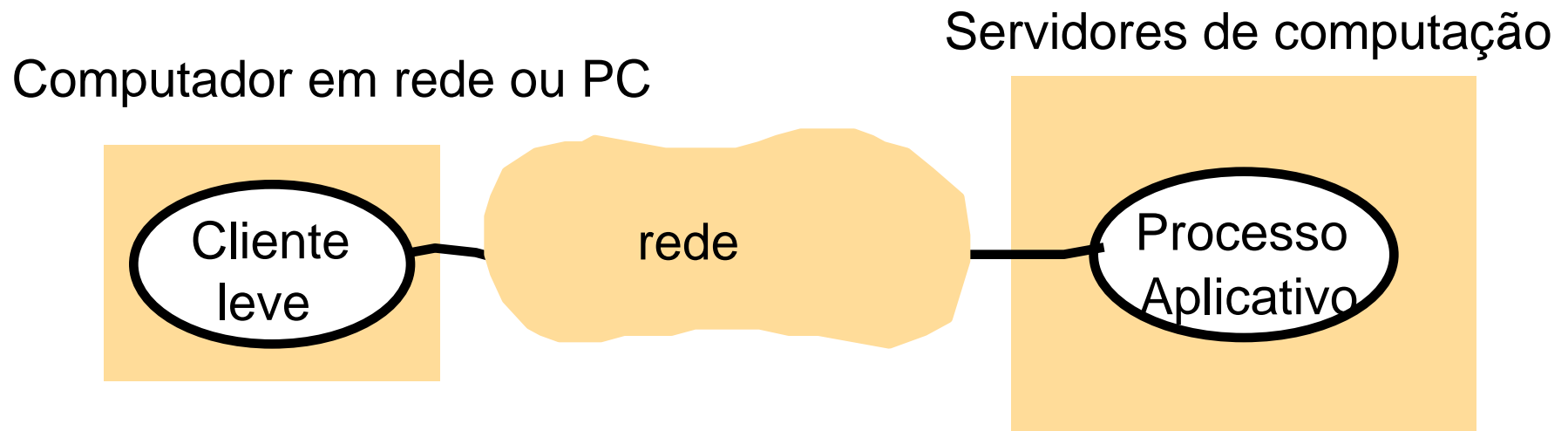
Computadores em rede

- Aplicativos e dados podem ser armazenados em um servidor de arquivos, sendo acessados de qualquer máquina na rede.
- Um usuário executa seus programas e acessa seus dados de qualquer computador.
- O SO também pode ser carregado da rede.

Clientes leves

- ❑ Camada de software que oferece ao usuário uma interface baseada em janelas para que este possa executar programas aplicativos em um computador remoto.
- ❑ Terminais remotos – X-window.
- ❑ Unix – X-11

Clientes leves



Clientes leves e servidores de computação

Dispositivos móveis

- Dispositivos computacionais que se movem entre locais físicos – notebooks, equipamentos de mão (PDAs), telefones móveis, etc.
- Clientes podem se mover entre redes, acessando um servidor fixo, ou os servidores podem também ser móveis, junto com os clientes.
- Interoperabilidade instantânea – associações entre dispositivos são rotineiramente criadas e destruídas.

Requisitos de projeto

- Problemas de desempenho
 - ▣ Limitação da capacidade de recursos (compartilhados) de processamento e de comunicação.
 - ▣ Reatividade
 - Usuários exigem respostas rápidas e consistentes.
 - Atraso na resposta – carga e desempenho do servidor e da rede + atrasos nos componentes de software envolvidos (serviços de comunicação dos sistemas operacionais do cliente e do servidor, *middleware*, código do processo que implementa o serviço, escalonamento dos processos cliente e servidor).

Requisitos de projeto

□ Problemas de desempenho (Cont.)

▣ Taxa de rendimento (*throughput*)

- Velocidade com que o trabalho computacional é feito.
- Afetado pelas velocidades de processamento nos clientes e servidores e pelas taxas de transferência de dados.

▣ Balanceamento de carga computacional

- Aplicativos e servidores devem executar sem disputar os mesmos serviços, explorando os recursos computacionais disponíveis.

Requisitos de projeto

□ Problemas de desempenho (Cont.)

■ Qualidade do serviço

- Confiabilidade, segurança e desempenho. Adaptabilidade e disponibilidade.
- QoS – atender requisitos temporais – o sistema deve garantir a disponibilidade dos recursos necessários no momento apropriado.

Requisitos de projeto

- Problemas de desempenho (Cont.)
 - ▣ Uso de cache e replicação
 - Usar protocolos para manter a consistência das caches.
 - O protocolo verifica se as cópias nas caches estão atualizadas, permitindo o seu uso, ou atualizando o conteúdo destas caches.

Requisitos de projeto

□ Problemas de desempenho (Cont.)

▣ Dependabilidade

- Correção, confiança e confiabilidade.
- Tolerância a falhas – os aplicativos devem continuar a funcionar corretamente na presença de falhas no hardware, no software e na comunicação.
- Confiabilidade – redundância (dispendiosa e complexa);
- Segurança -

MODELOS DE SISTEMAS DISTRIBUÍDOS



Modelos de Sistemas Distribuídos

- Tornar explícitas todas as suposições relevantes sobre o sistema.
- Fazer generalizações a respeito do que é possível e do que é impossível
 - ▣ Propriedades desejáveis ou necessárias;
 - ▣ Algoritmos.

Modelos de Sistemas Distribuídos

- Discutir aspectos sobre
 - Interação – processos interagem trocando mensagens, resultando na comunicação e na coordenação (sincronização e ordenação de atividades); o modelo de interação reflete as propriedades temporais do sistema;
 - Falha – software e hardware podem falhar prejudicando a operação do sistema; o modelo de falhas define e classifica as falhas;
 - Segurança – modelo de segurança define e classifica as formas que os ataques aos sistemas podem assumir.

Modelo de Interação

- Processos interagem de acordo com a arquitetura adotada
 - ▣ Servidores cooperam para fornecer um serviço (DNS) / processos peer cooperam para atingir um objetivo comum (teleconferência);
- Programa simples seqüencial – descrito por um algoritmo – seqüência de passos – seqüência de estados
 - ▣ Estado – valor das variáveis em um momento da execução do sistema.

Modelo de Interação

- Sistema distribuído – comportamento e estado é definido por um algoritmo distribuído
 - ▣ Estado do sistema – estado de cada processo do sistema, que é privativo do processo;
 - ▣ Fatores relevantes:
 - Comunicação e
 - Noção global de tempo.

Modelo de Interação

- Desempenho da comunicação:
 - Latência – atraso
 - Latência da transmissão (tempo para envio dos bits) + atraso no acesso à rede (espera pela disponibilidade do meio) + tempo de processamento dos serviços de comunicação do SO;
 - Largura de banda – volume total de informações que pode ser transmitido em determinado momento;
 - Jitter – variação do atraso.

Modelo de Interação

- Relógios e tempo:
 - Cada computador tem o seu relógio interno;
 - Relógios de computadores se desviam de uma base de tempo (relógio de referência perfeito), e as taxas de desvio dos computadores são diferentes;
 - Taxa de desvio do relógio (drift) – quantidade relativa do desvio de um relógio de computador do relógio perfeito;
 - Solução
 - GPS- Global Positioning System
 - Sincronização entre os relógios.

Sistemas distribuídos síncronos

- Tempo para executar cada ação de um processo tem limites inferior e superior conhecidos;
- Tempo para a transmissão de mensagens em um canal tem limite conhecido;
- Cada processo tem um relógio local cuja taxa de desvio do tempo real tem um limite conhecido.

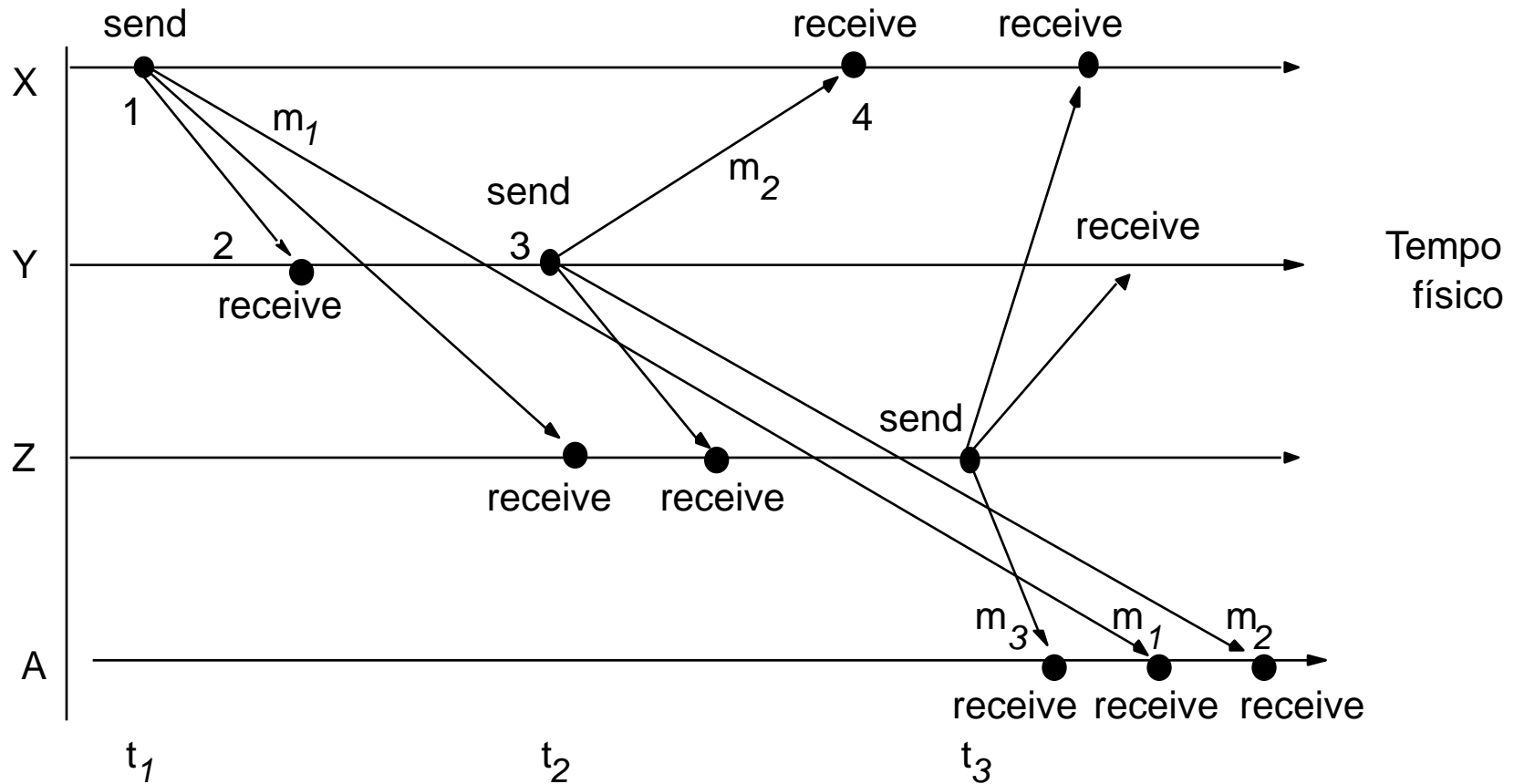
Sistemas distribuídos síncronos

- Necessário reservar recursos para garantir as restrições temporais impostas à execução dos processos – SO de tempo real;
- Reservar recursos para garantir as restrições temporais para a comunicação – rede de controle síncrona ou QoS;
- Usar GPS ou algoritmos para sincronizar os relógios, mantendo o drift dentro de um limite conhecido.

Sistemas distribuídos assíncronos

- Não existem limites de tempo conhecidos para a execução de ações dos processos nem para a transferência de mensagens nos canais de comunicação, e as taxas de desvio dos relógios locais não são limitadas.
- Ex. Internet.
- Alguns sistemas utilizam tratamentos para o atraso em sua execução – web browsers

Sistemas distribuídos assíncronos



Ordenação de eventos no tempo físico.

Sistemas distribuídos assíncronos

- Ordenação de eventos no tempo físico
 - Utilizar relógios sincronizados permitiria que as mensagens fossem exibidas para todos os processos na mesma ordem.
 - Em sistemas assíncronos pode-se utilizar o conceito de relógios lógicos (Lamport 1978), proporcionando uma ordenação dos eventos relacionados, que ocorrem em diferentes processos.

Modelos Parcialmente Síncronos

- Modelos em que há informação de limites temporais parcial em tempo ou espaço:
 - ▣ Sincronismo Parcial (Dwork, Lynch e Stckmeyer 1988) ;
 - ▣ Assíncrono Temporizado (Cristian e Fetzer 1999);
 - ▣ Base de Computação Temporizada – TCB (Veríssimo e Casimiro 2002);
 - ▣ HA (Gorender, Macêdo e Raynal 2007).
- Motivação: Existência de limitações de resolução de problemas em ambientes assíncronos

Modelo de falhas

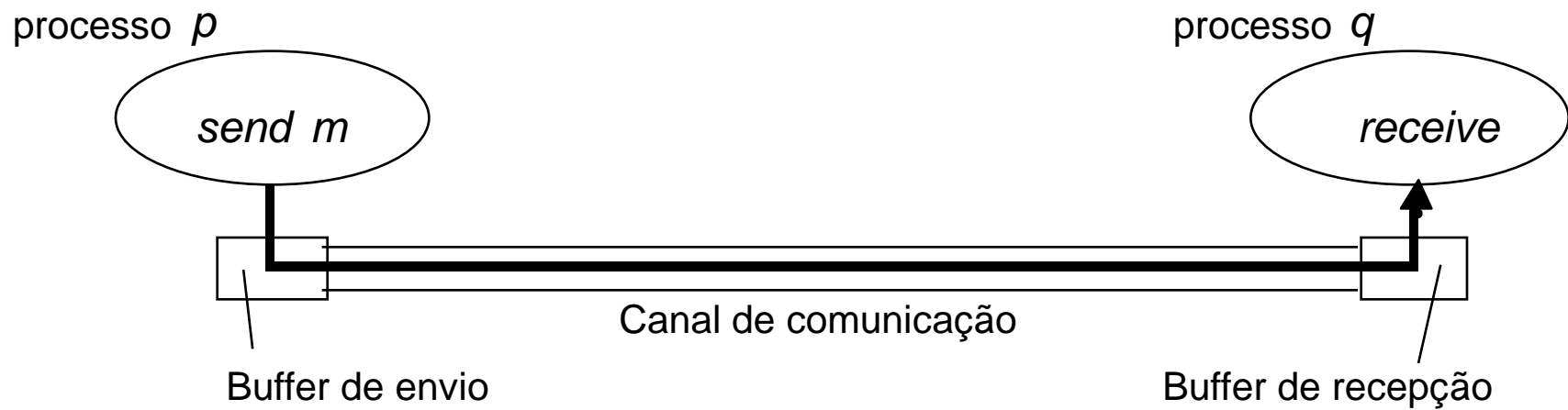
- Processos e canais de comunicação podem falhar – divergir do que considerado um comportamento correto ou desejável.
- Define como uma falha pode se manifestar em um sistema.

Modelo de falhas

□ Falhas por omissão

- Um processo ou canal de comunicação deixa de executar as ações que deveria.
- Processo – colapso (crash) – interrompe a execução.
- Comunicação – uma mensagem é perdida, não sendo transferida para o seu destino –
 - Falha por omissão de envio – entre o processo remetente e o buffer de envio;
 - Falha por omissão de recepção – entre o buffer de recepção e o processo destino;
 - Falha por omissão de canal – no meio de comunicação.

Modelo de falhas



Processos e canais de comunicação

Modelo de falhas

□ Falhas arbitrárias

- Falha bizantina – qualquer tipo de erro pode ocorrer.
- Processos podem informar valores incorretos aos demais.
- Processo – omite arbitrariamente passos desejados do processamento ou efetua processamento indesejado;
- Canais de comunicação – corromper o conteúdo de uma mensagem, enviar uma mensagem inexistente ou entregar mensagens mais de uma vez.

Modelo de falhas

<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
Fail-stop (Parada por falha)	Processo	Processo para e permanece parado. Outros processos podem detectar esse estado (sistema síncrono).
Crash (Colapso)	Processo	Processo pára e permanece parado. Outros processos podem não detectar esse estado.
Omissão	Canal	Uma mensagem inserida em um buffer de envio nunca chega no buffer de recepção do destinatário.
Omissão de envio	Processo	Um processo conclui um envio, mas a mensagem não é colocada em seu buffer de envio.
Omissão de recepção	Processo	Uma mensagem é colocada no buffer de recepção de um processo mas esse processo não a recebe.
Arbitrária (Byzantine)	Processo ou canal	Processo/canal exibi comportamento arbitrário: ele pode enviar/transmitir mensagens arbitrárias em qualquer momento, cometer omissões; um processo pode parar ou realizar uma ação incorreta.

Falhas por omissão e arbitrárias.

Modelo de falhas

- ❑ Falha de temporização
 - ▣ Aplicáveis aos sistemas distribuídos assíncronos temporizados.

<i>Class of Failure</i>	<i>Affects</i>	<i>Description</i>
Relógio	Processo	O relógio local do processo ultrapassa os limites de sua taxa de desvio.
Desempenho	Processo	O processo ultrapassa os limites do intervalo de tempo entre duas etapas.
Desempenho	Canal	A transmissão de uma mensagem demora mais do que o limite definido.

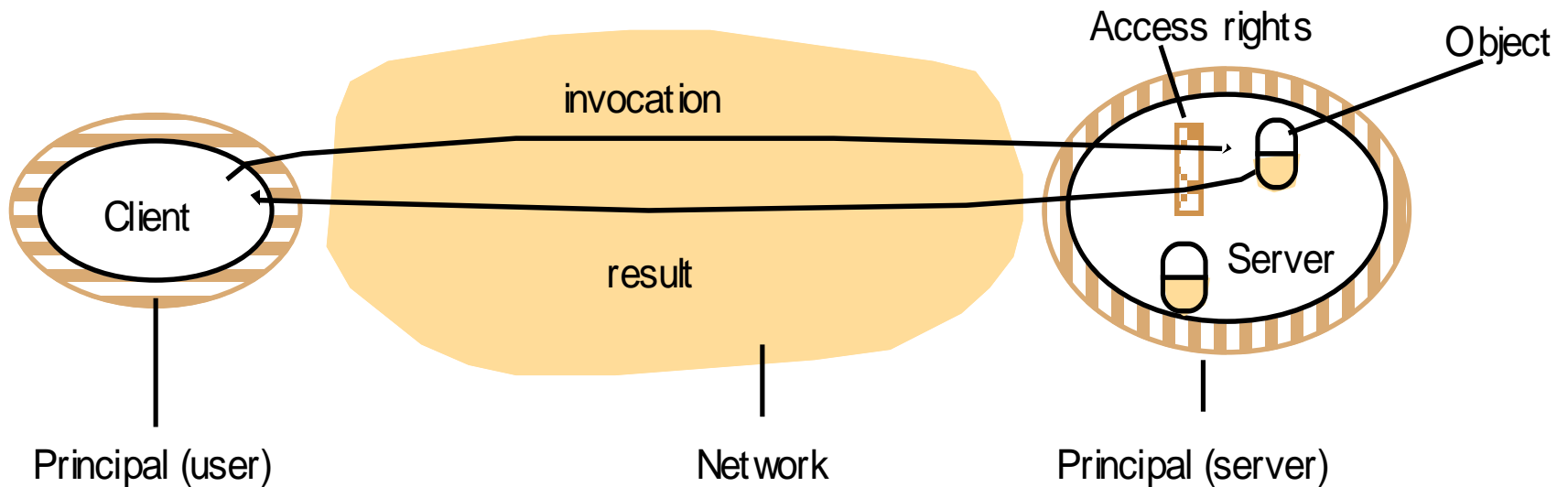
Modelo de falhas

- Mascaramento de falhas
 - ▣ Um serviço mascara uma falha ocultando-a completamente ou convertendo-a em um tipo de falha aceitável.
 - Protocolos de retransmissão de mensagens ocultam falhas de omissão do canal.
 - Replicação – diversas réplicas de um servidor ocultam a falha de uma das réplicas.

Modelo de falhas

- Confiabilidade da comunicação de um para um –
 - ▣ Comunicação confiável:
 - Validade – qualquer mensagem do buffer de envio é entregue ao buffer de recepção de seu destino, independente do tempo necessário para tal;
 - Integridade – a mensagem recebida é idêntica à enviada e nenhuma mensagem é entregue duas vezes.
 - ▣ Ameaças:
 - Protocolos inconsistentes e ataques.

Modelo de Segurança

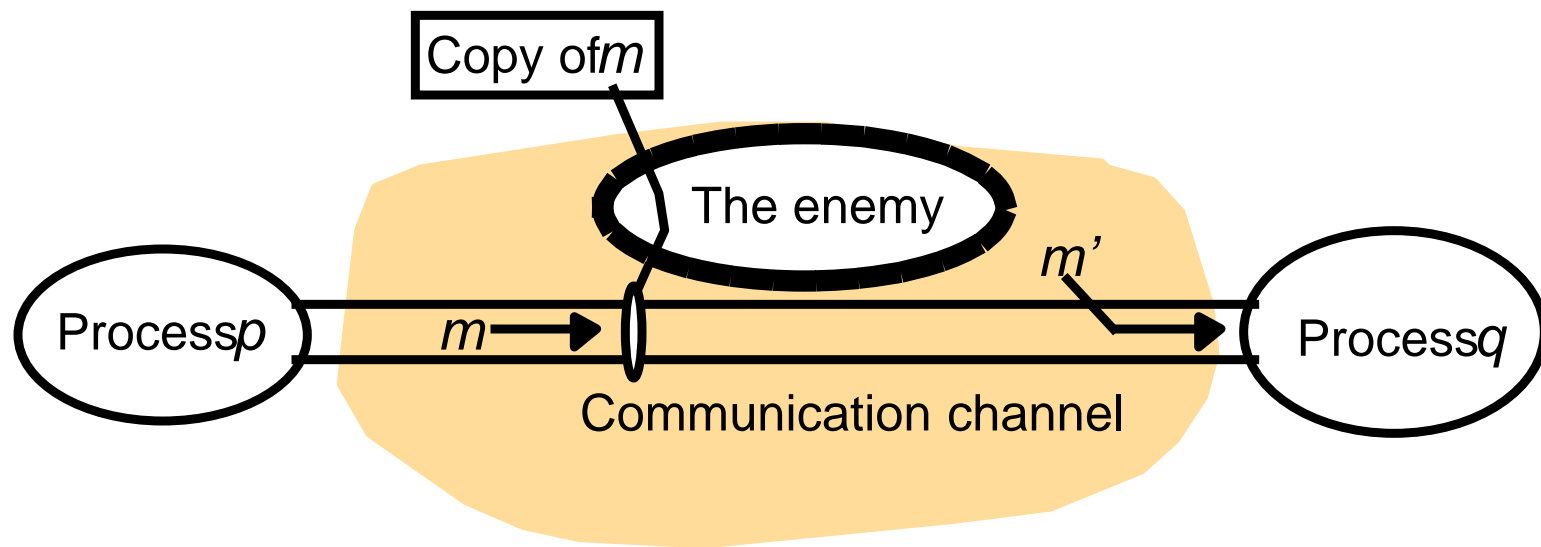


Objetos e principais

Modelo de Segurança

- Direitos de acesso especificam quem pode executar determinadas operações sobre um objeto.
- Principal – autorização de quem executa uma solicitação, que pode ser um usuário ou um processo.
- Servidores e processos peers precisam verificar a identidade e autorização de quem efetua uma solicitação
 - Interfaces dos servidores e dos processos peers são públicas.

Modelo de Segurança



O invasor (atacante).

Um Processo capaz de enviar qualquer mensagem para qualquer processo e ler ou copiar qualquer mensagem entre dois processos

Modelo de Segurança

- Ameaças aos processos:
 - Um processo pode receber mensagens (pedidos) de outros processos e não ser capaz de determinar a identidade do remetente.
 - Um servidor pode receber solicitações com identidade e localização falsas.
 - Um cliente pode receber o resultado de uma solicitação que venha de um invasor, e não ser capaz de identificar a origem da mensagem.

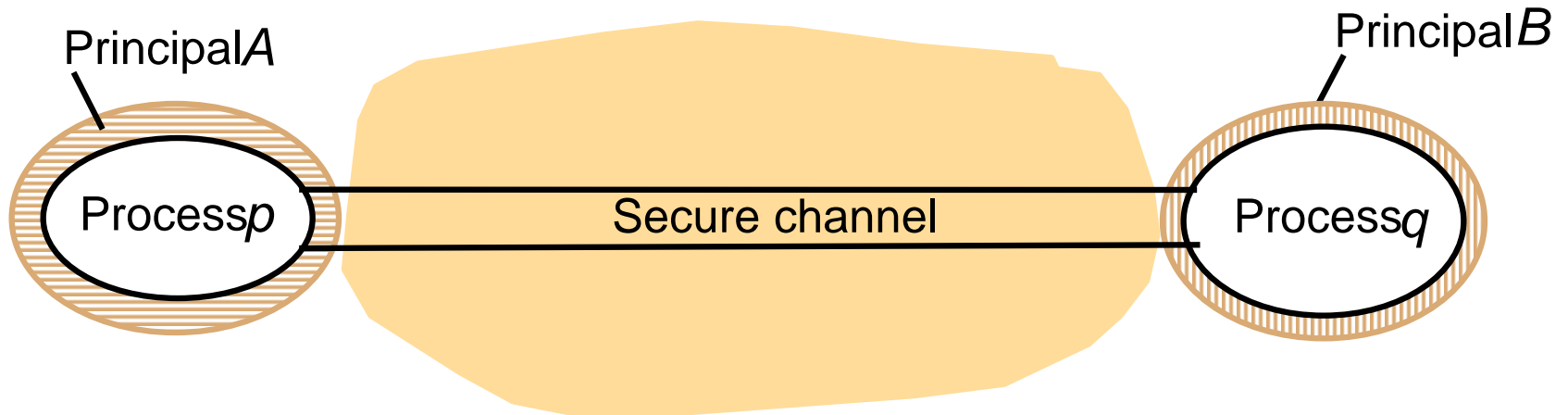
Modelo de Segurança

- Ameaças aos canais de comunicação:
 - Um invasor pode copiar, alterar ou injetar mensagens na rede.
 - Mensagens privadas podem ser tornadas públicas e mensagens podem ser copiadas para uso posterior.

Modelo de Segurança

- Técnicas de segurança:
 - Criptografia: ciência de manter as mensagens seguras
 - Cifrar – processo de embaralhar uma mensagem de maneira a ocultar seu conteúdo.
 - Autenticação: provar as identidades dos remetentes das mensagens.
 - Incluir em uma mensagem uma parte cifrada que possua conteúdo suficiente para garantir a sua autenticidade.

Modelo de Segurança



Canais Seguros - canal de comunicação entre dois processos que garanta a privacidade e a integridade dos dados transmitidos por eles. Cada um dos processos conhece a identidade do principal em nome de quem o outro processo está executando.

TEMPO E ESTADOS GLOBAIS



Introdução

- Problemas que dependem da sincronização do relógio:
 - ▣ Manutenção da consistência dos dados distribuídos;
 - ▣ Verificação da autenticidade de uma requisição enviada para um servidor;
 - ▣ Eliminação do processamento de atualizações replicadas
- O tempo é relativo ao observador!

Introdução

- Não existe um tempo global absoluto!
- Dificuldade de se definir o tempo da ocorrência de eventos em diferentes nós em um SD.
- Entretanto é necessário definir a relação de causa e efeito de diferentes eventos!

Relógios, eventos e estados

□ Modelo –

- ▣ Um sistema distribuído é definido como um conjunto Π de N processos p_i , $i = 1, 2, \dots, N$. cada processo é executado em um único processador e os processadores não compartilham memória. Cada processo p_i tem um estado s_i que ele transforma ao ser executado. O estado de um processo inclui todos os objetos do seu ambiente operacional.

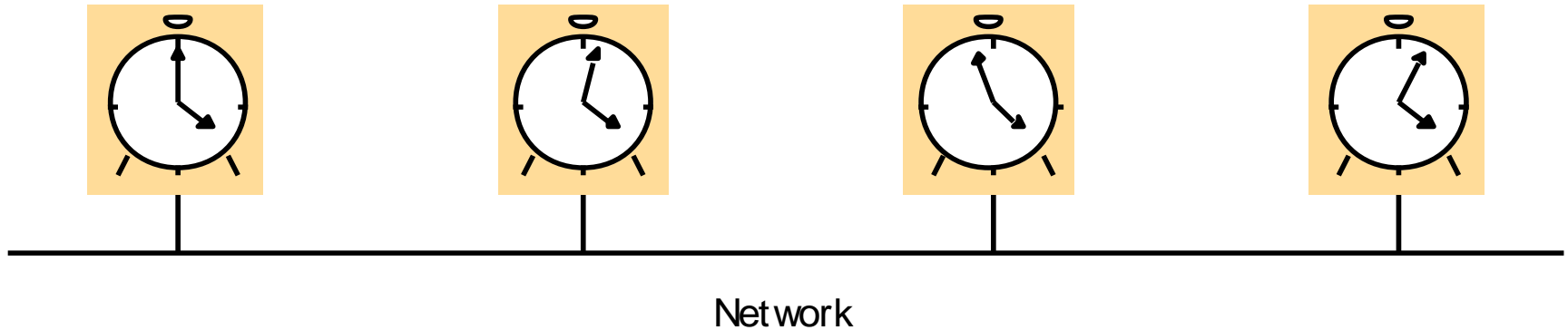
Relógios, eventos e estados

- Processos se comunicam por troca de mensagens. Ações dos processos enviam ou recebem mensagens, ou transformam o seu estado.
- Um evento é definido como uma única ação que um processo realiza ao ser executado. A seqüência de eventos em um único processo p_i pode ser colocada em uma ordem total única.

Relógios

- Cada computador contém seu próprio relógio físico – cristal que emite pulsos e emitem interrupções.
- Relógio de software – o SO atualiza o relógio de software $C_i(t)$ para o processo p_i a partir da leitura do relógio de hardware $H_i(t)$. $C_i(t)$ é a leitura do relógio de software do processo p_i no momento real t .

Desvio de relógio



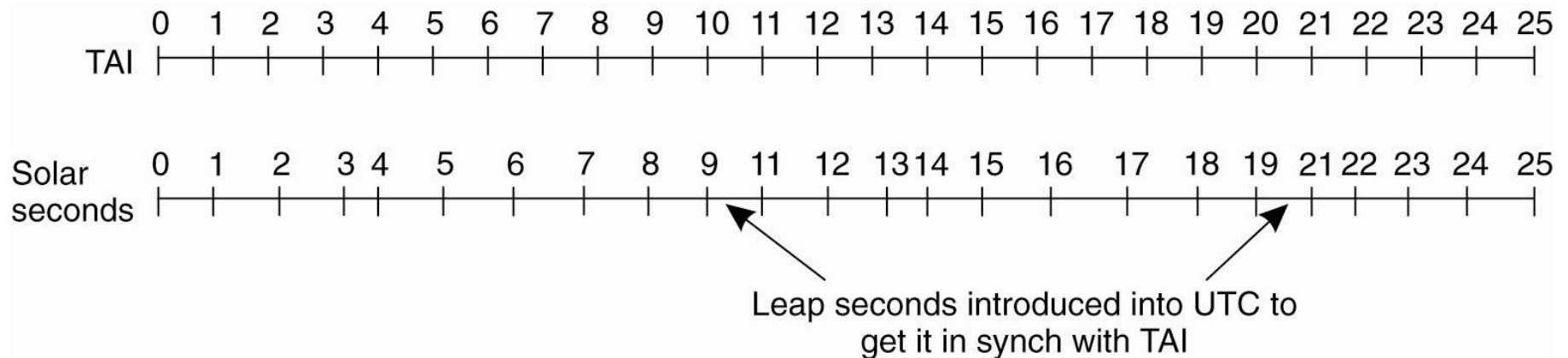
- Distorção – diferença instantânea entre as leituras de quaisquer dois relógios;
- Derivação (*drift*) – relógios divergem, contam o tempo com diferentes velocidades.
- Taxa de derivação (*drift rate*) – deslocamento entre o relógio local e um relógio de referência nominal perfeito.

Tempo universal coordenado

- Tempo atômico internacional (*International Atomic Time* - TAI) – relógios físicos precisos que usam osciladores atômicos.
- Tempo universal coordenado (UTC) – baseado no tempo atômico com correções – GPS.

Tempo universal coordenado

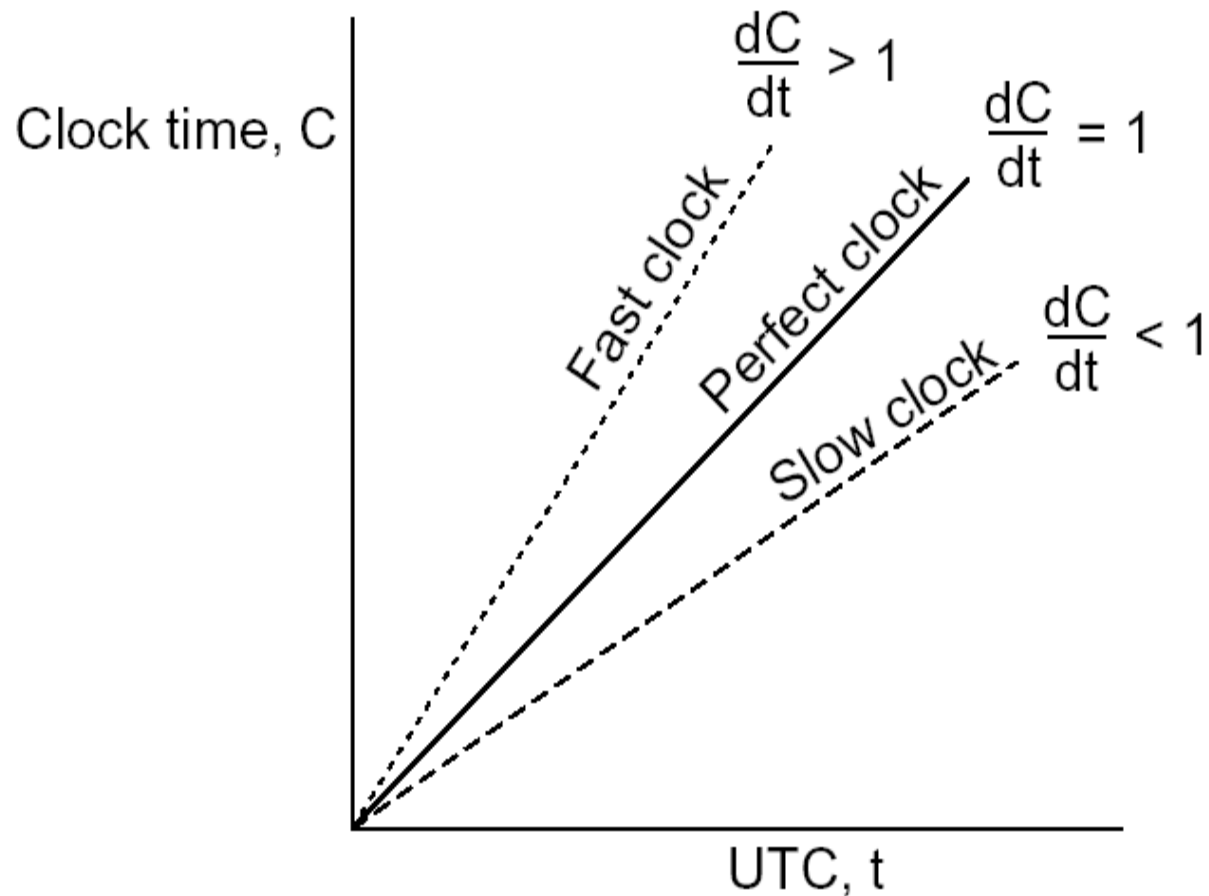
- Segundos TAI têm comprimento constante, diferente dos segundos solares



Sincronizando relógios físicos

- Sincronização externa – relógios C_i são sincronizados com uma fonte de tempo de referência externa.
- Sincronização interna – dois relógios são sincronizados com um grau de precisão conhecido. Podem apresentar um desvio com relação ao tempo real.
- Um sistema sincronizado externamente com um limite D estará sincronizado internamente com um limite $2D$.

Sincronizando relógios físicos



Sincronizando relógios físicos

- Relógio correto (C ou H) – a taxa de derivação é inferior a um limite conhecido – o erro na medição é limitado.
- Monotonicidade – condição de que um relógio C apenas sempre avance.
- Condição de correção mista – monotonicidade + taxa de derivação limitada entre pontos de sincronização.

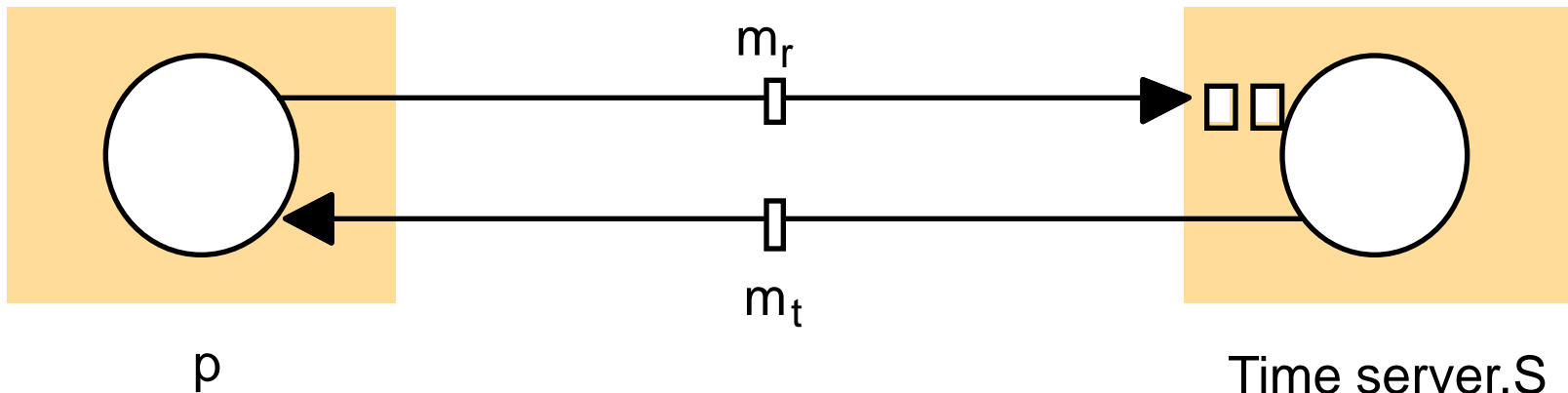
Sincronizando relógios físicos

- ❑ Relógio falho – não aplica condições de correção.
- ❑ Falha de colapso – pára completamente;
- ❑ Falha arbitrária – qualquer outra falha.

Sincronização em sistema síncrono

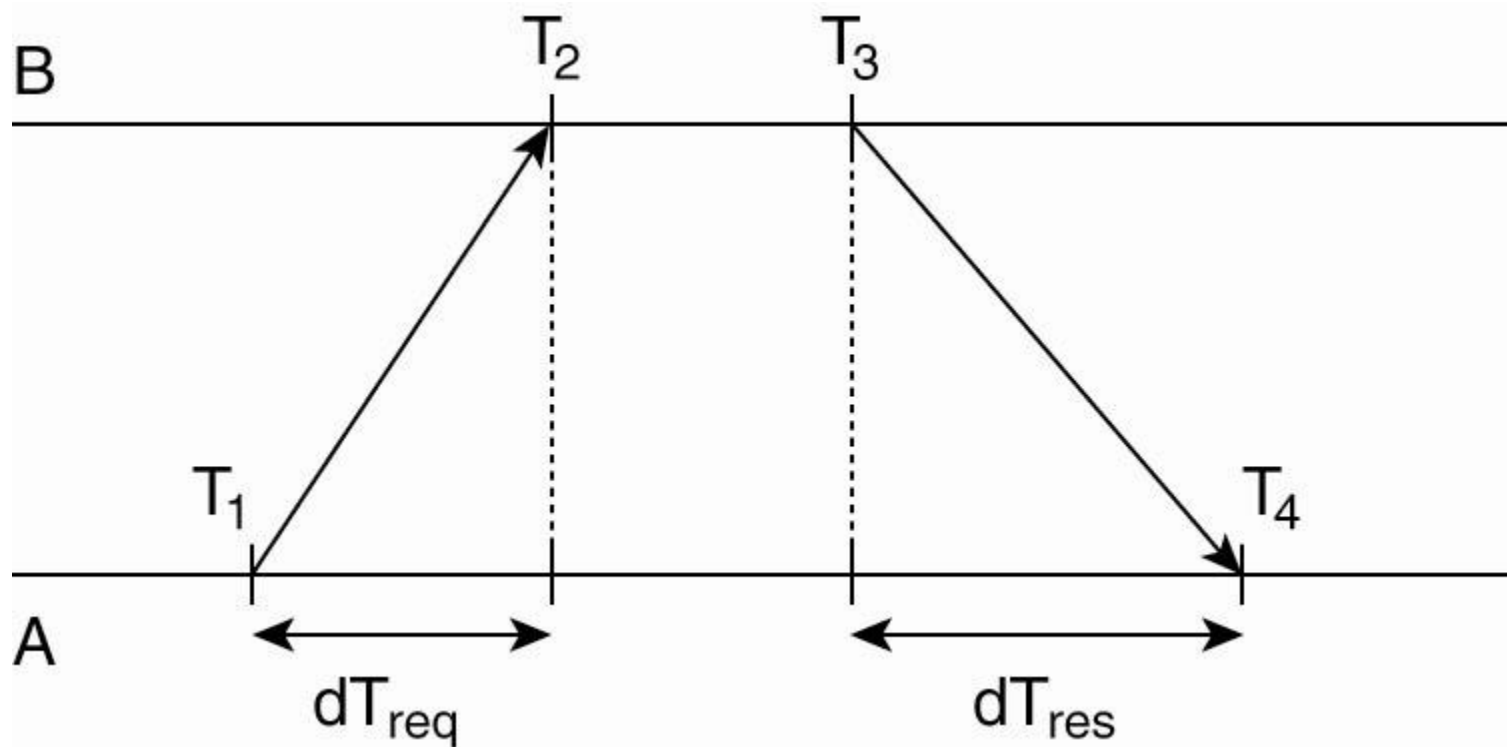
- Um relógio envia o tempo t de seu relógio para outro.
- O receptor atualiza seu relógio para $t + (max + min)/2$, sendo max o tempo máximo de transmissão e min o tempo mínimo de transmissão entre os dois processos. Considerando $u = max - min$, o desvio máximo entre os dois relógios será igual a $u/2$.

Método de Flaviu Cristian - 1989



- Utiliza um servidor de tempo conectado a um dispositivo UTC, sincronizando externamente.
- Processos solicitam o tempo ao servidor.
- Para tolerar a falha do servidor diversos servidores de tempo sincronizados com receptores UTC podem ser usados.

Método de Flaviu Cristian - 1989



Obtenção da hora corrente por meio de um servidor de tempo.

Método de Flaviu Cristian - 1989

$$T_{viagem} = T_4 - T_1$$
$$t_i = t + T_{viagem} / 2$$

Se for conhecido o tempo mínimo para viagem – *min*:

Tempo para a chegada da mensagem pelo relógio do servidor: $[t +$

$min, t + T_{viagem} - min]$ – largura $T_{viagem} - 2min$

Precisão - $\pm(T_{viagem}/2 - min)$

A



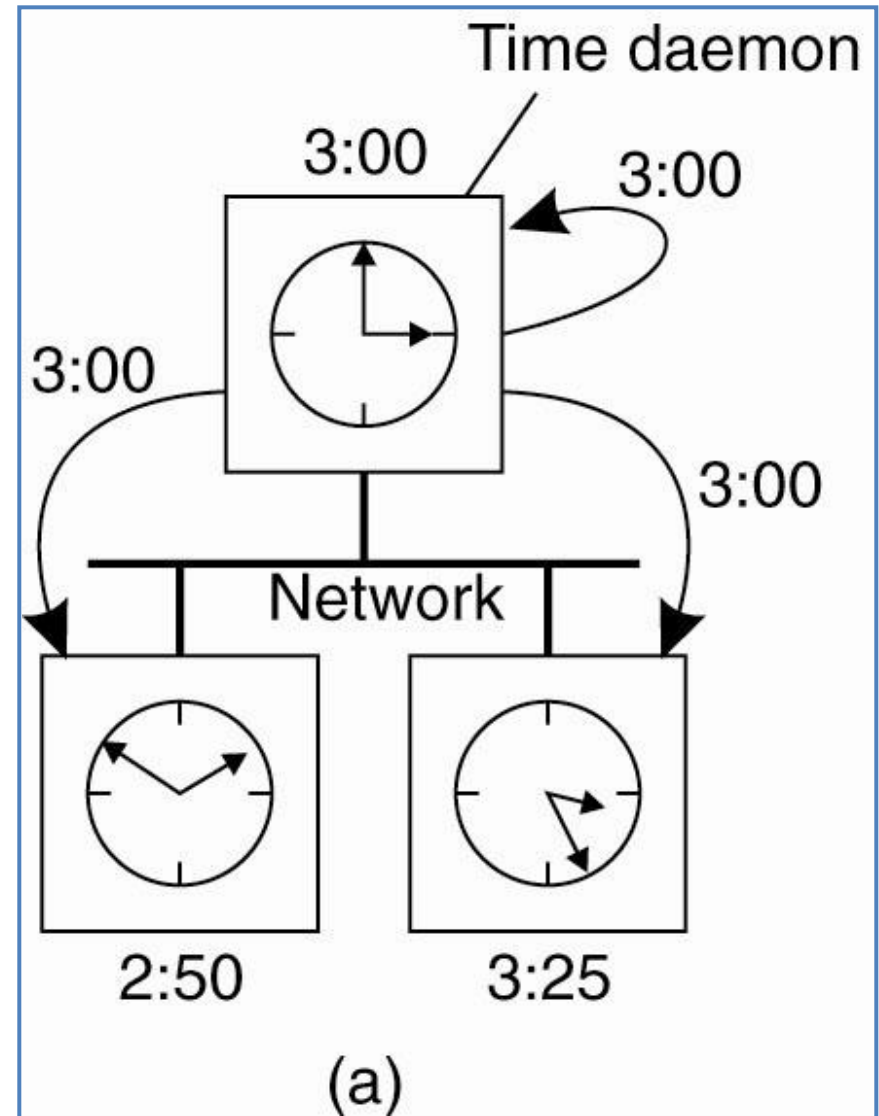
Obtenção da hora corrente por meio de um servidor de tempo.

Algoritmo de Berkeley

- Unix Berkeley.
- Um coordenador (mestre) solicita o tempo de cada relógio C_i , estima os tempos locais a partir dos tempos de transmissão (RTT), faz a média, e indica a cada relógio o seu valor de ajuste.

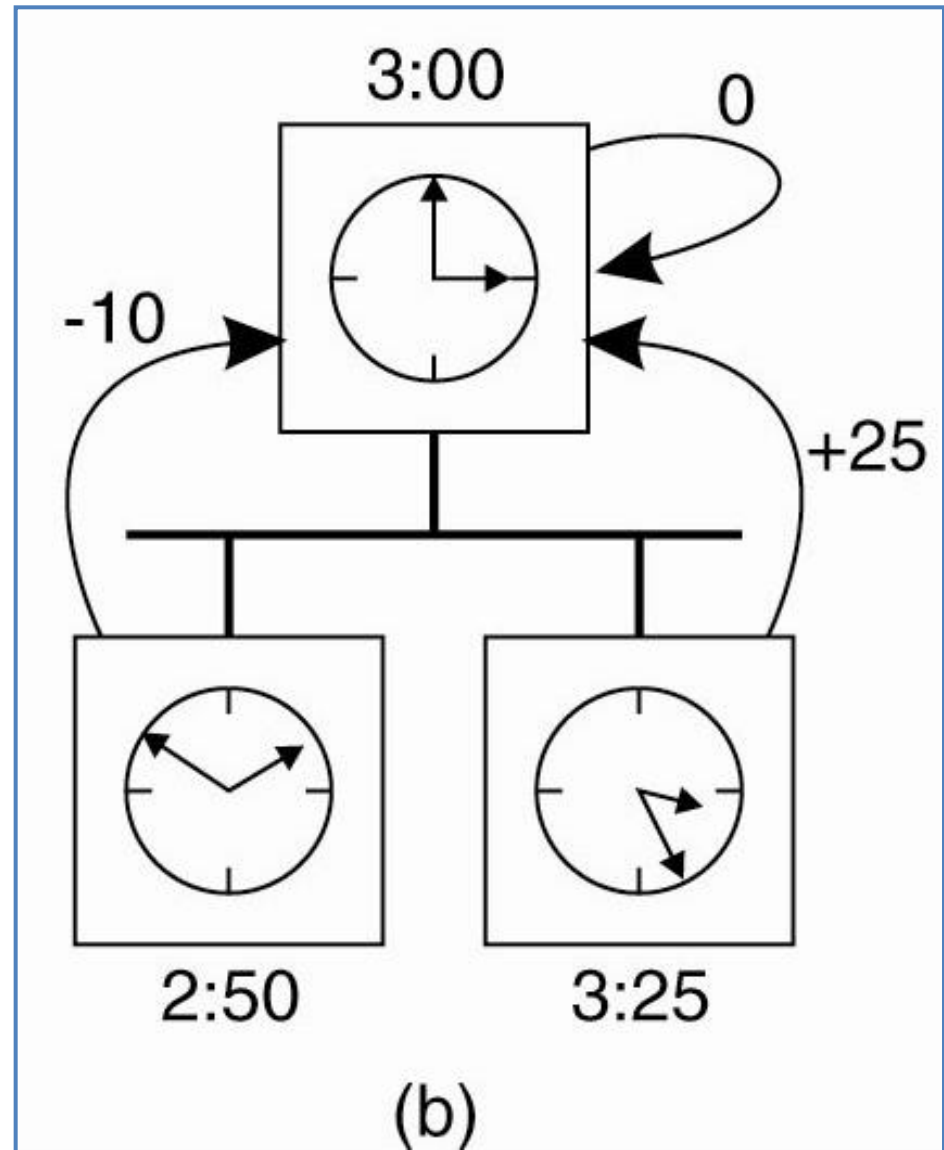
Algoritmo de Berkeley

O *daemon* de tempo pergunta a todas as outras máquinas os valores marcados por seus relógios.



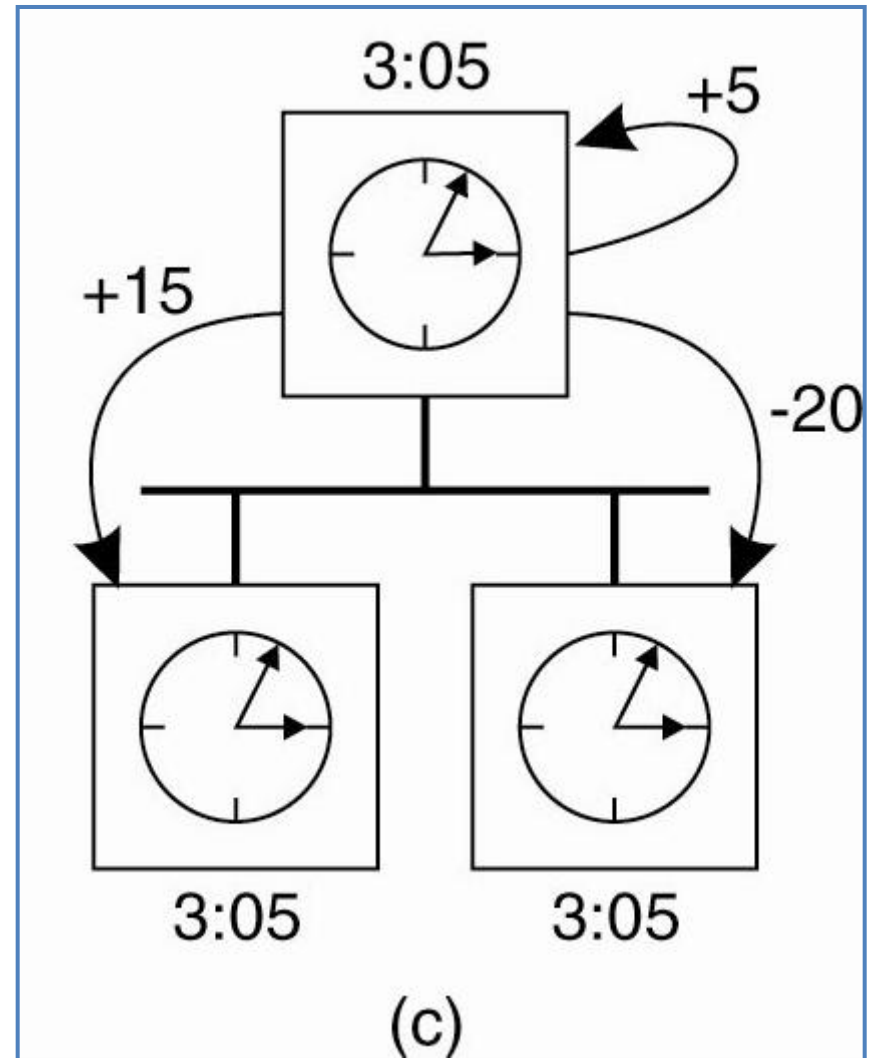
Algoritmo de Berkeley

As máquinas respondem.



Algoritmo de Berkeley

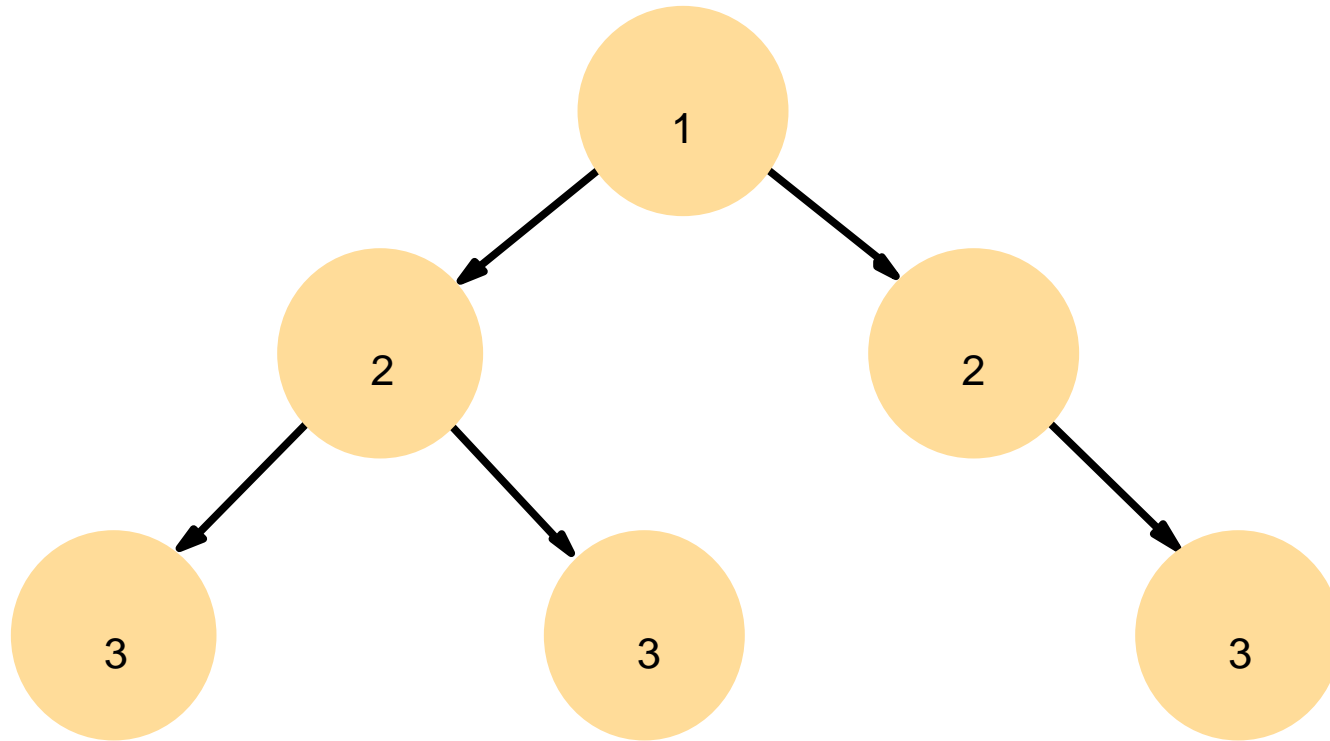
O *daemon* de tempo informa a todos como devem ajustar seus relógios.



NTP – Network Time Protocol

- Serviço de tempo para Internet.
- Servidores são conectados em uma hierarquia lógica chamada sub-rede de sincronização – níveis são chamados de *strata*:
 - ▣ Stratum 1 – servidores primários, nível raiz – conectados diretamente a uma fonte de tempo - dispositivo UTC);
 - ▣ Stratum 2 - sincronizados com os servidores primários.

NTP – Network Time Protocol



Nota: setas indicam controle de sincronização, números fornecem o stratum.

NTP – Network Time Protocol

□ Características:

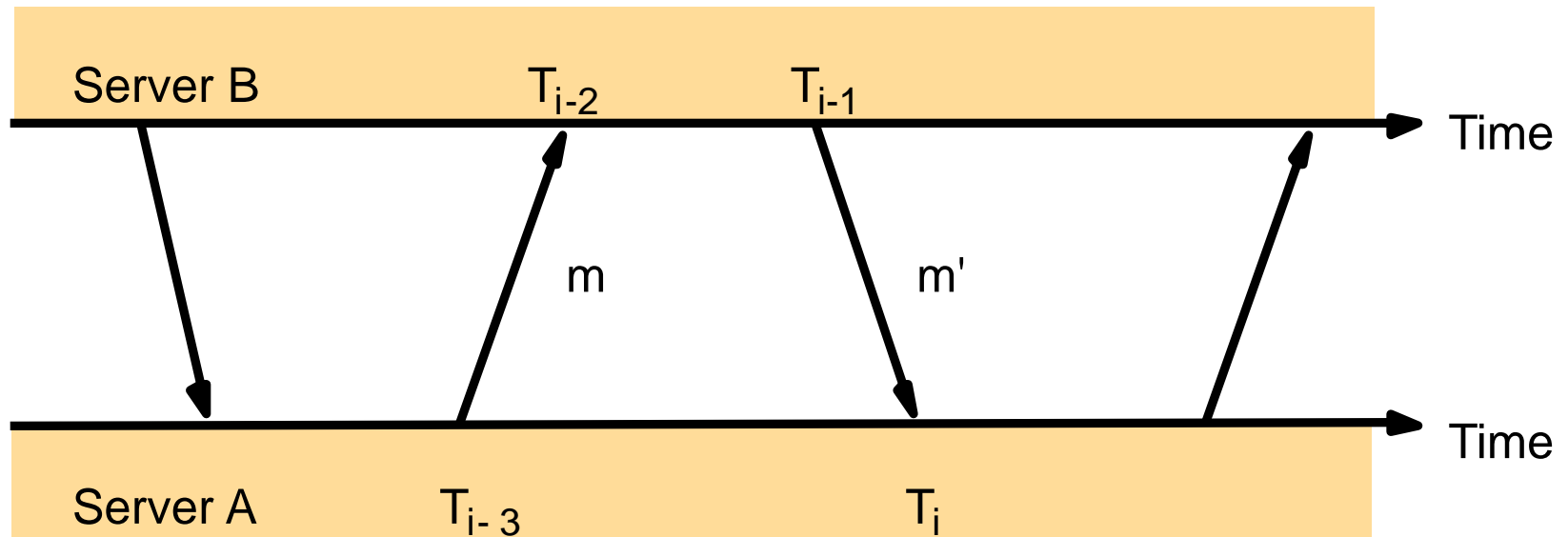
- Fornecer um serviço que permita aos clientes da Internet serem sincronizados precisamente com a UTC.
- Fornecer um serviço confiável que possa sobreviver a longas perdas de conectividade.
- Permitir que os clientes sejam sincronizados de forma suficientemente freqüente para compensar as taxas de derivação encontradas na maioria dos computadores.
- Fornecer proteção contra interferência no serviço de tempo, seja mal-intencionada ou acidental.

NTP – Network Time Protocol

□ Formas de sincronização:

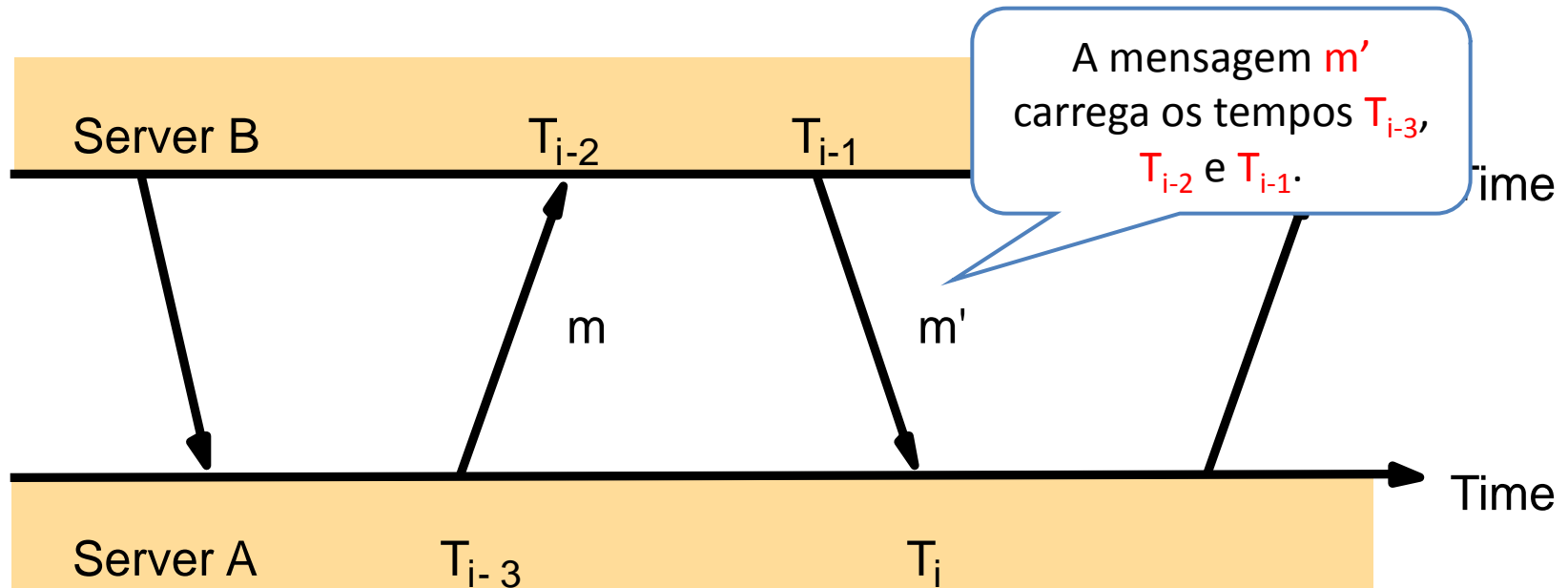
- Multicast: uso em rede local de alta velocidade; um ou mais servidores enviam a informação de tempo via multicast; precisão relativamente baixa;
- Modelo de chamada de procedimento: um servidor aceita requisições de outros computadores; precisão melhor do que com multicast;
- Modo simétrico: em redes locais e nos servidores de hierarquia alta; dois servidores no modo simétrico trocam mensagens com informação de temporização.

NTP – Network Time Protocol



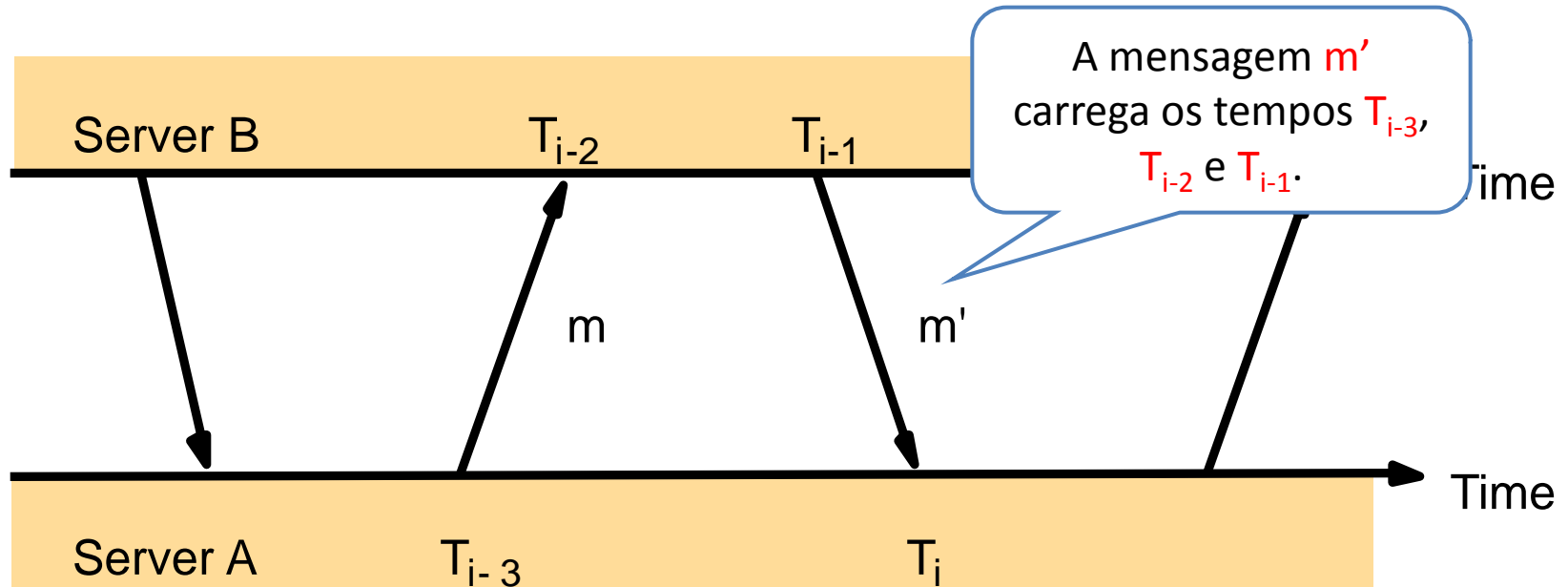
- Mensagens trocadas entre dois pares NTP.

NTP – Network Time Protocol



- Mensagens trocadas entre dois pares NTP.

NTP – Network Time Protocol



Tempo de transmissão total das duas mensagens d_i :

$$T_{i-2} = T_{i-3} + t + o \text{ e } T_i = T_{i-1} + t' - o;$$

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1};$$

Compensação entre os dois relógios o_i :

$$o = o_i + (t' - t)/2, \text{ onde } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2;$$

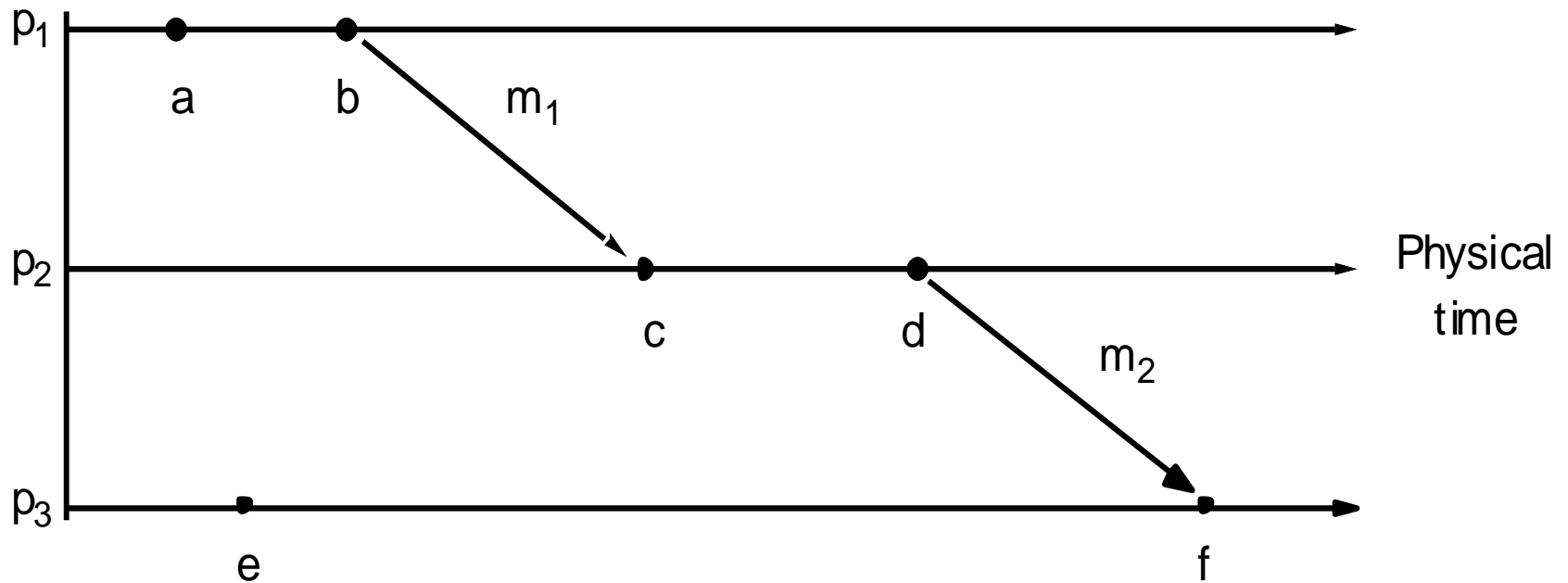
Tempo lógico e relógios lógicos

- Lamport 1978 – como não podemos sincronizar corretamente relógios, não podemos usar adequadamente estes relógios para ordenar eventos.
- Ordenação causal – ordenar eventos dependentes em diferentes processos:
 - ▣ Dois eventos em um processo p_i ocorrem em ordem - \rightarrow_i ;
 - ▣ O envio de uma mensagem ocorre antes de sua recepção.

Tempo lógico e relógios lógicos

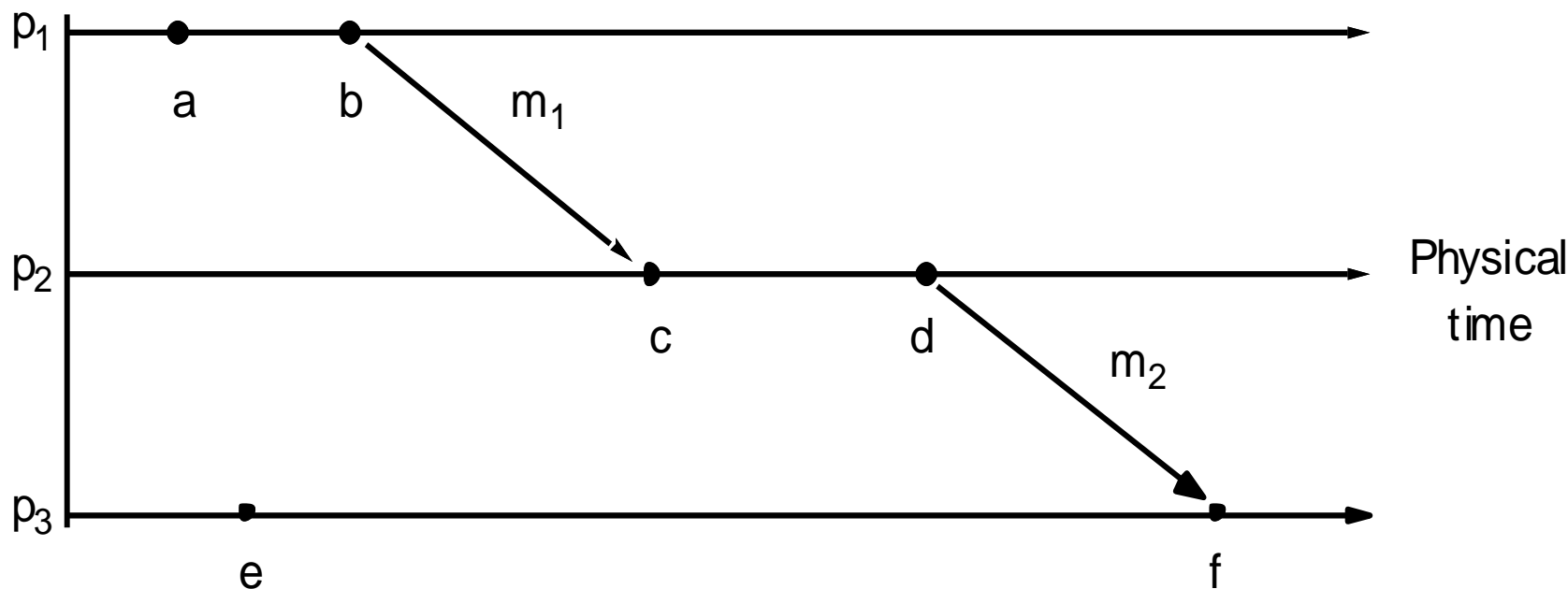
- Relação *happened-before* - \rightarrow :
 - ▣ Se $e \rightarrow_i e'$, então $e \rightarrow e'$;
 - ▣ Para qualquer mensagem m , $send(m) \rightarrow receive(m)$;
 - ▣ Se e , e' e e'' são eventos tais que $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Tempo lógico e relógios lógicos



□ Eventos ocorrendo em três processos

Tempo lógico e relógios lógicos



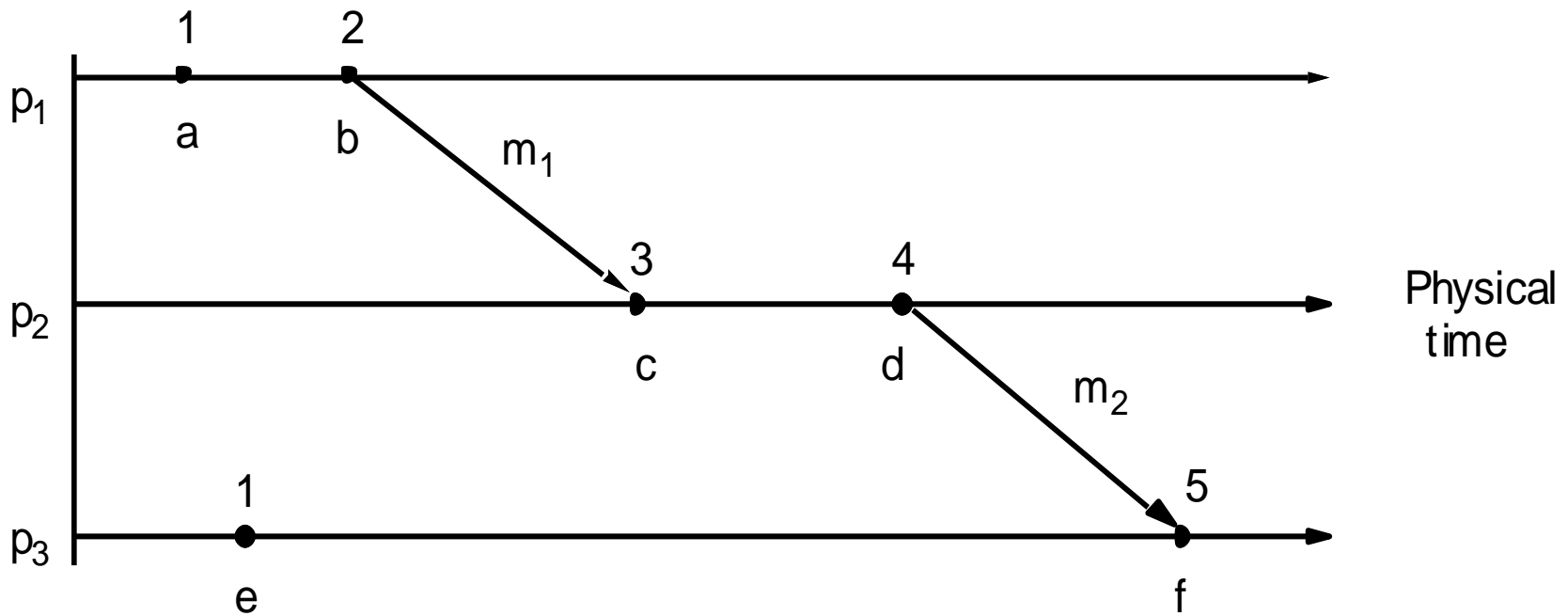
□ Eventos ocorrendo em três processos

$a \rightarrow_i b$ e portanto $a \rightarrow b$; $c \rightarrow d$; $b \rightarrow c$; $d \rightarrow f$; $a \rightarrow f$;

Relógios lógicos

- Contador de software monotônico – atualização dos relógios lógicos L_i :
 - ▣ L_i é incrementado antes da ocorrência de um evento no processo p_i ;
 - ▣ Mensagens enviadas por p_i levam o valor $t = L_i$;
 - ▣ Ao receber uma mensagem o processo calcula o maior valor entre seu relógio lógico local e o valor recebido, e depois incrementa.

Relógios lógicos



- Indicações de tempo de Lamport para os eventos mostrados na figura anterior.

Relógios vetoriais

- Propostos para superar deficiências dos relógios lógicos de Lamport.
 - ▣ $L(e) < L(e')$ não garante que $e \rightarrow e'$;
- Relógio vetorial para um sistema de N processos – vetor de N inteiros;
- Todos os processos possuem o seu relógio vetorial V_i , que são enviados nas mensagens trocadas.

Relógios vetoriais

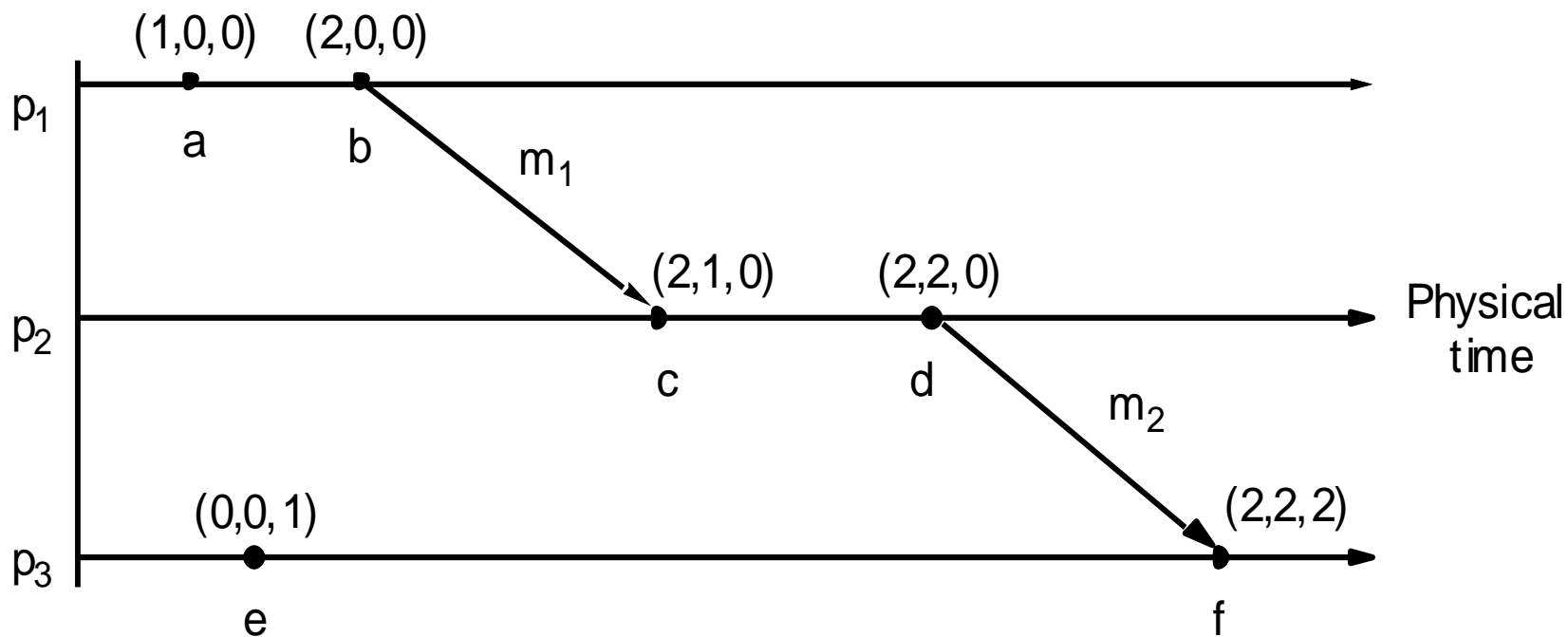
□ Regras para atualizar os relógios:

- RV1: Inicialmente, $V_i[j]=0$, para $i, j=1, 2, \dots, N$.
- RV2: imediatamente antes de p_i indicar o tempo de um evento, ele configura $V_i[i]:=V_i[i]+1$.
- RV3: p_i inclui o valor $t=V_i$ em cada mensagem;
- RV4: Quando p_i recebe uma indicação de tempo t em uma mensagem, configura $V_i[j]:=max(V_i[j], t[j])$, para $j = 1, 2, \dots, N$.

Relógios vetoriais

- $V_i[i]$ é o número de eventos em que p_i participou e $V_i[j]$ é o número de eventos ocorridos em p_j nos quais p_i foi potencialmente afetado.
 - ▣ $V=V'$ sse $V[j]=V'[j]$ para $j=1, 2, \dots, N$.
 - ▣ $V \leq V'$ sse $V[j] \leq V'[j]$ para $j=1, 2, \dots, N$.
 - ▣ $V < V'$ sse $V \leq V' \wedge V \neq V'$.

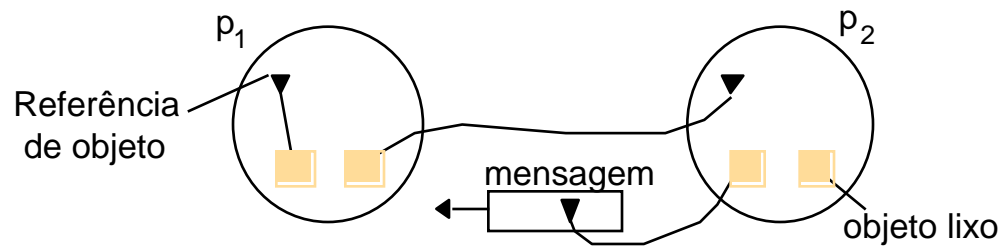
Relógios vetoriais



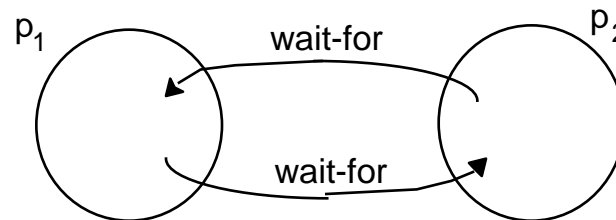
Indicações de tempo vetoriais.

Estados globais

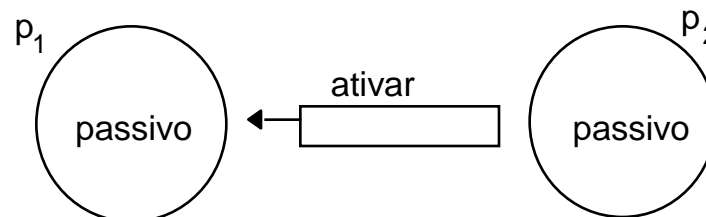
a. Garbage collection –
coleta de lixo distribuída



b. Deadlock - Impasse



c. Terminação –
detecção de término distribuída



Estados globais

- Difícil devido à ausência de tempo global.
- Estado global a partir dos estados locais dos processos.
- No sistema Π o histórico da execução de um processo é a seqüência de eventos que ocorre no processo

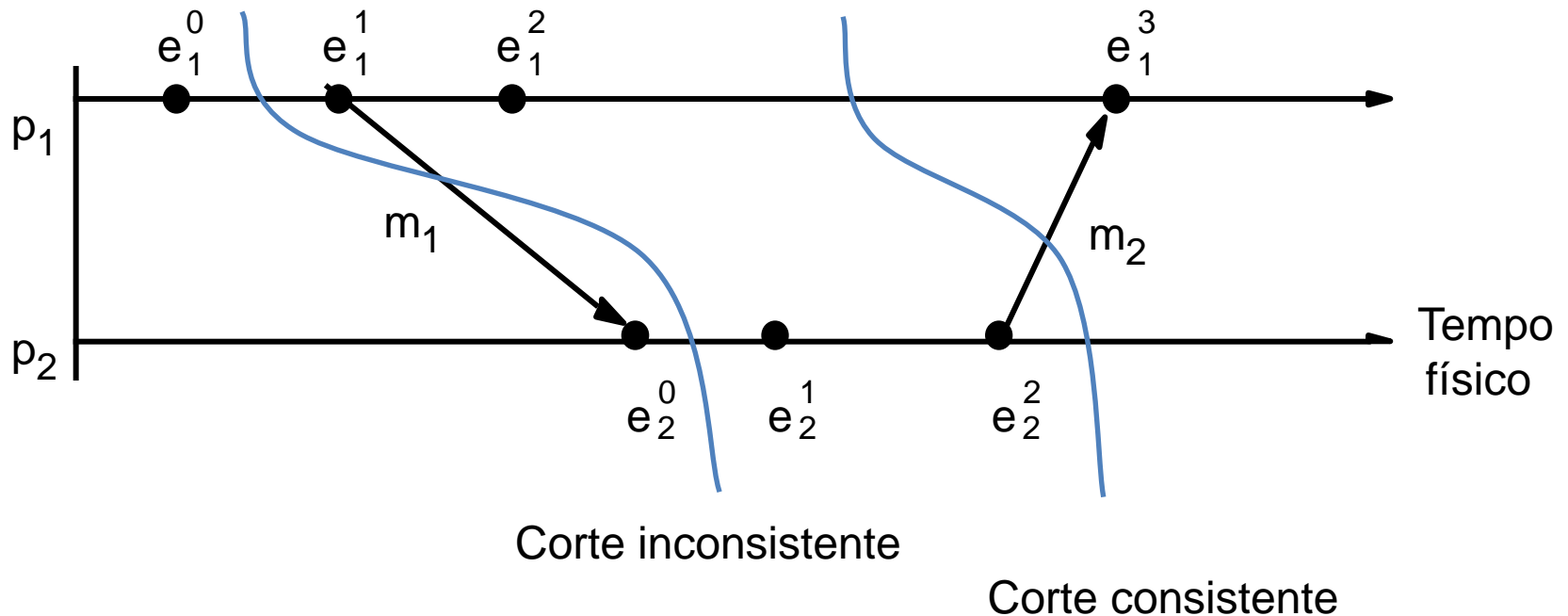
$$\textit{histórico}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

Estados globais - cortes

- Prefixo de histórico: $h_i^k = \langle e_i^0, e_i^1, \dots, e_i^k \rangle$
- Corte – subconjunto do histórico global de um sistema que é uma união de prefixos de históricos de processo.
 - ▣ $C = h_1^{c1} \cup h_2^{c2} \cup \dots \cup h_n^{cn}$
- Fronteira do corte – conjunto de eventos processados por cada processo logo antes do corte.

Estados globais - cortes

- Corte inconsistente - $\langle e_1^0, e_2^0 \rangle$
- Conte consistente - $\langle e_1^2, e_2^2 \rangle$



Estado global

- Estado global consistente – corresponde a um corte consistente.
- Execução do sistema – série de estados globais consistentes.
- Série (*run*) – ordenação total de todos os eventos em um histórico que é consistente com cada ordem do histórico local.

Predicado de estado global

- Função que faz o mapeamento do conjunto de estados globais de processos no sistema para $\{verdadeiro, Falso\}$.
 - ▣ Ex. condições de impasse ou término.

Predicado de estado global

- *Safety* (segurança) – em relação a uma propriedade α é a afirmação de que α é avaliado como *Falso* para todos os estados S que podem ser atingidos a partir do estado inicial S_0 .
- *Liveness* (terminação) – para uma propriedade desejável β a garantia de que em qualquer *run* β será avaliado como Verdadeiro em algum estado S_L atingido a partir de S_0 .
- Um algoritmo de estado global baseado em corte consistente em algum momento (a termo) detecta a ocorrência de um estado global estável de forma segura.

Chandy e Lamport

Marker Sending Rule for process i

- 1 Process i records its state.
- 2 For each outgoing channel C on which a marker has not been sent, i sends a marker along C before i sends further messages along C .

Marker Receiving Rule for process j

On receiving a marker along channel C :

if j has not recorded its state **then**

Record the state of C as the empty set
Follow the "Marker Sending Rule"

else

Record the state of C as the set of messages received along C after j 's state was recorded and before j received the marker along C

APLICAÇÕES EM SISTEMAS DISTRIBUÍDOS: REPLICAÇÃO E COMUNICAÇÃO EM GRUPO



Replicação

- Replicação é um mecanismo de Mascaramento de Falhas
- Usa de Redundância Espacial
- Um serviço existe em mais de uma “réplica”

Replicação

- Replicação NÃO é para aumentar escalabilidade
 - É possível que aumente a escalabilidade ao permitir mais de um ponto de processamento de requisições, contudo tem que SINCRONIZAR um ESTADO ÚNICO

Replicação

- Replicação aumenta disponibilidade
 - Suponha um servidor com disponibilidade 3 9s (0.999)
 - Para replicação tolerante a CRASH:
 - Qual a disponibilidade com 2 réplicas?
 - Qual a disponibilidade com 3 réplicas?
 - Qual a disponibilidade com 5 réplicas?
 - Hipótese de falhas independentes

Replicação

- Replicação deve ser transparente:
 - ▣ Uso de único ponto de acesso
 - ▣ Uso de dispatcher para fazer round-robin de requisições entre as réplicas
 - ▣ Uso de multicast para um grupo de réplicas

Replicação

- Replicação é para serviços STATEFULL
- A estratégia de replicação pode mudar de acordo com o nível de serviço a ser oferecido e o tipo de falha a ser tolerado
- Exemplos:
 - Practical Byzantine Fault Tolerant: Falhas BIZANTINAS – suporta intrusão
 - PRIMARY-BACKUP: Falhas por CRASH – suporta defeito que torna o servidor principal indisponível

Replicação

- Replicação Ativa: todo o estado é replicado em tempo de execução
- A replicação ativa é custosa. Uma possibilidade é a replicação passiva.

Replicação

□ Replicação Passiva:

- Conhecido por PRIMARY-BACKUP
- Quem realiza o serviço é o primário
- O primário mantém atualizadas réplicas com o backup do Estado

Replicação

- O Primário recebe requisições, as executa, atualiza o estado dos backups e retorna o resultado ao cliente
- Em caso de falhas do Primário, um dos Backups assume (Qual? Algoritmo de Eleição de Líder)



Replicação Ativa

- Cada requisição é processada por todo o grupo
- O estado é atualizado por todos e todos enviam a resposta ao cliente
- Sem falhas não é necessário sincronizar o estado



Replicação Ativa

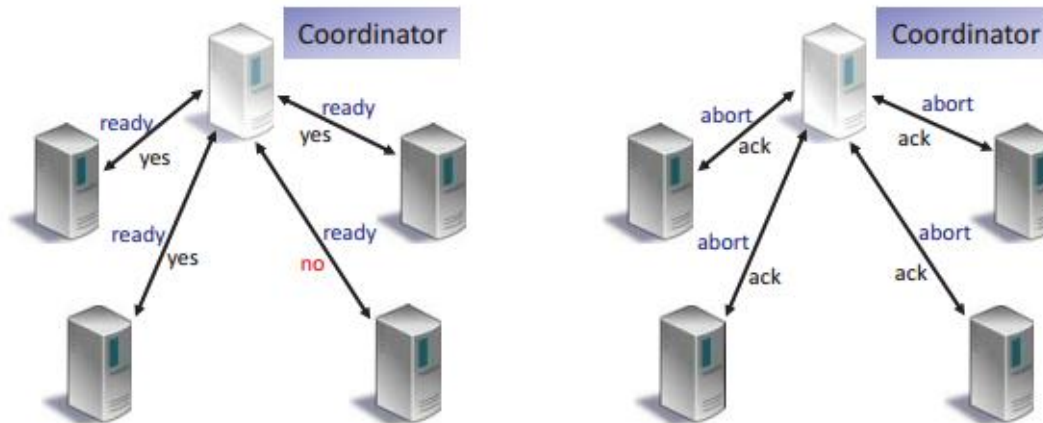
- Em caso de falhas por crash, se a réplica falha for restaurada ela deve consultar o estado das réplicas ativas
- É importante:
 - ▣ Detectar quando uma réplica falha, e excluir do grupo
 - ▣ Processar o retorno da réplica, resincronizando o seu estado
 - ▣ Esta resincronização pode ser por um conjunto de AÇÕES mantidas em LOG, de forma similar a recuperação de transações em BD

Replicação Ativa

- Se a resincronização implicar por simplesmente copiar todo o estado, a mesma será ineficiente em termos de performance
- Se a resincronização implicar em repetir os passos que as réplicas ativas executaram, uma réplica antes de falhar não pode nunca executar uma ação que as demais não executaram previamente
- As réplicas devem concordar de forma UNIFORME no conjunto de ações antes de processar e responder as requisições

Replicação Ativa

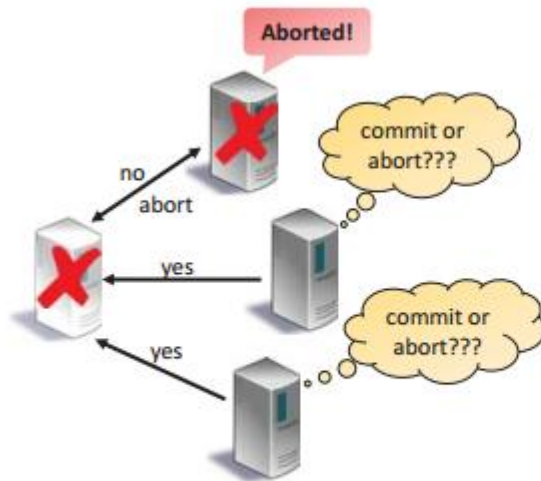
- Acordo na Replicação ATIVA:
 - ▣ Usamos o conhecido TWO Phase Commit:



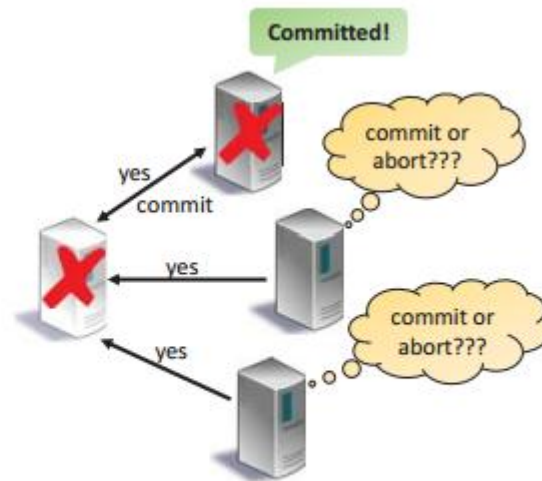
- ▣ Uma variante do CONSENSO

Replicação Ativa

- Acordo na Replicação ATIVA:
 - ▣ Mas o TWO Phase Commit tem problemas:
 - 'Se O coordenador falha, podemos eleger um outro coordenador, mas e se:



The nodes cannot commit!

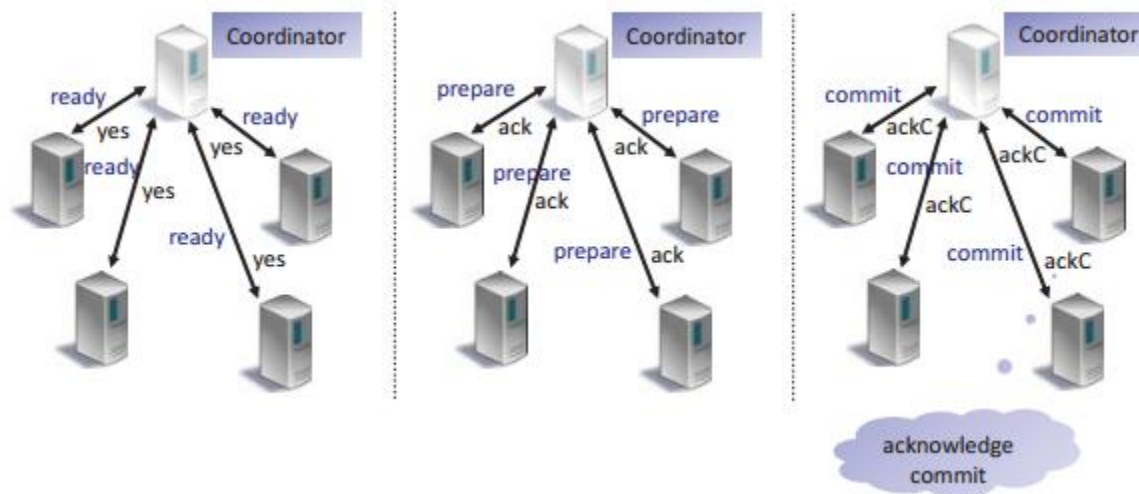


The nodes cannot abort!

Replicação Ativa

□ THREE Phase Commit:

- Acrescentamos uma nova fase, o PREPARE



- O objetivo é informar a todos os nós que todos estão aptos a efetuar o COMMIT (ou não)
- Ao fim desta fase, cada nó sabe quando todos os nós pretendem fazer COMMIT (ou não) antes de efetuar de fato

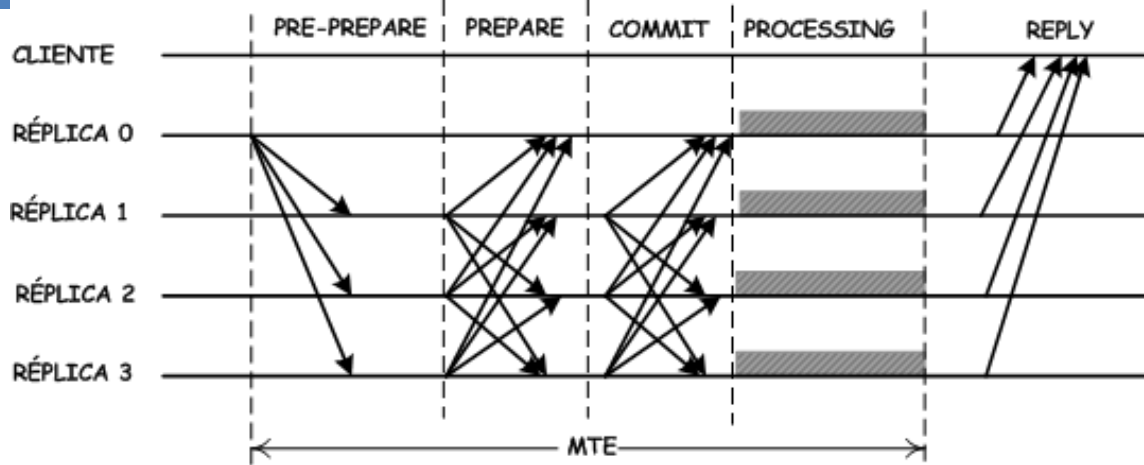
Replicação Ativa

- 2-PHASE COMMIT: Acordo Não Uniforme
- 3-PHASE COMMIT: Acordo Uniforme
- Possibilidade também por meio de:
 - ▣ Difusão Atômica
 - Acordo Não Uniforme na Entrega de Mensagens: 2PC
 - Acordo Uniforme na Entrega de Mensagens: 3PC

Replicação Ativa

- No caso de falhas BIZANTINAS, o acordo no conjunto de requisições é tolerante a falhas bizantinas
- Acordo BIZANTINO: O Problema dos Generais Bizantinos – LAMPORT
- Tolerar F réplicas bizantinas com um total de N réplicas
 - ▣ $N > 3 F$

Replicação Ativa



- Pre-prepare: Primário difunde uma requisição atribuindo uma sequência e assinando-a digitalmente
- Prepare: Réplicas enviam ACCEPT da requisição
- Commit: Se ao menos $2F+1$ ACCEPTS forem recebidos (incluindo ele mesmo), a réplica envia o COMMIT
- Processing: Se $2F+1$ COMMITs forem aceitos a requisição é enviada
- Reply: O CLIENT aceita como resposta uma mesma resposta enviada por $2F+1$ réplicas

Comunicação em Grupo

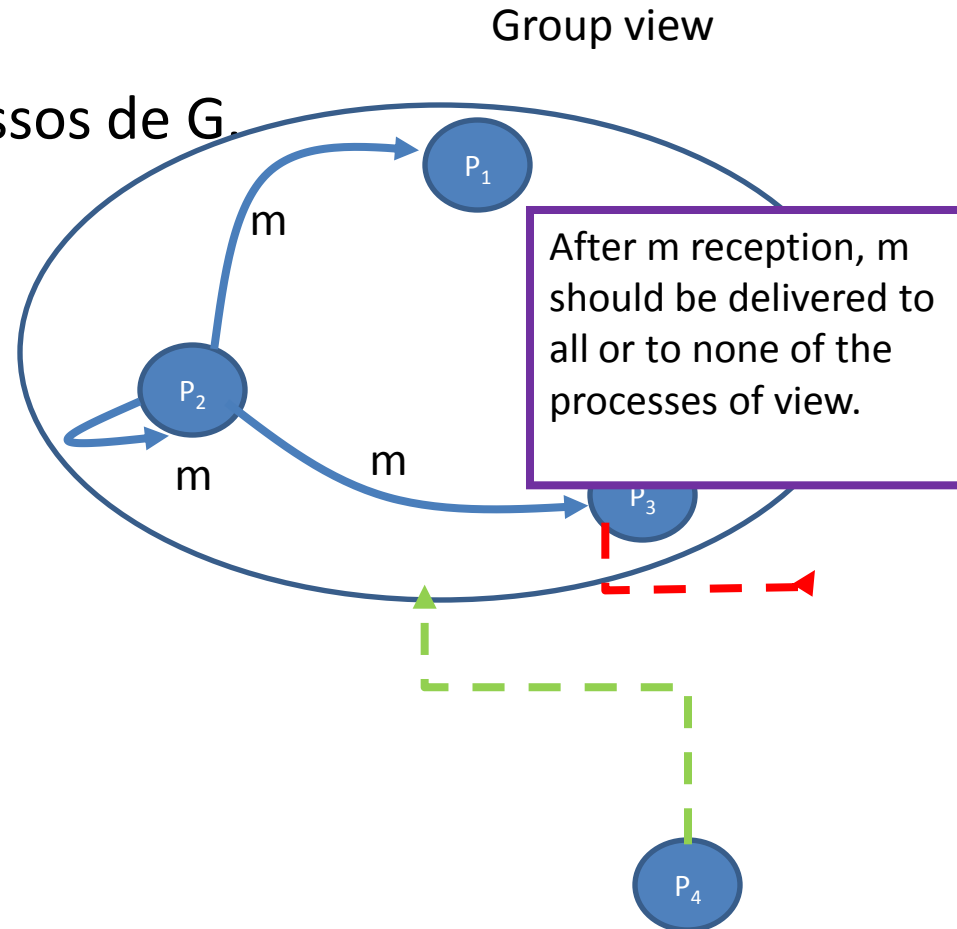
- Idéia da Comunicação em Grupo:
 - Manter um grupo G de processos que trocam mensagens entre si
 - Garantir que este conjunto de mensagens M seja o mesmo e na mesma ordem:
 - Ordem Causal: respeita happens-before
 - Ordem Total: a ordem é a mesma para todo o grupo

Comunicação em Grupo

- Problema de Comunicação em Grupo se divide em:
 - ▣ Entregar o conjunto de mensagens M na ordem a todo o grupo G : **Difusão Atômica Confiável**
 - ▣ Manter a visão V do grupo G : **Sincronia de Visões**

Comunicação em grupo

- $G = \Pi$, inicialmente
- $v_i^k(G)$ é visão de p_i dos processos de G .
- Primitivas:
 - $\text{join}(g)$
 - $\text{leave}(g)$
 - $\text{send}(g, m)$
 - $\text{deliver}(g, m)$
- Join e Leave alteram visão de $v_i^k(G)$ para $v_i^{k+1}(G)$



Difusão Atômica

- O problema da difusão atômica confiável é garantir que a mensagem chegue, apesar de falhas do meio ou de nós a TODOS nós
- Após usamos o ACORDO NA ENTREGA
 - ▣ TODOS ou NENHUM e na mesma ordem

Tipos de Difusão

- broadcast

- ▣ envio de mensagens a todos os nodos do sistema

- multicast

Envolve o conceito de grupos

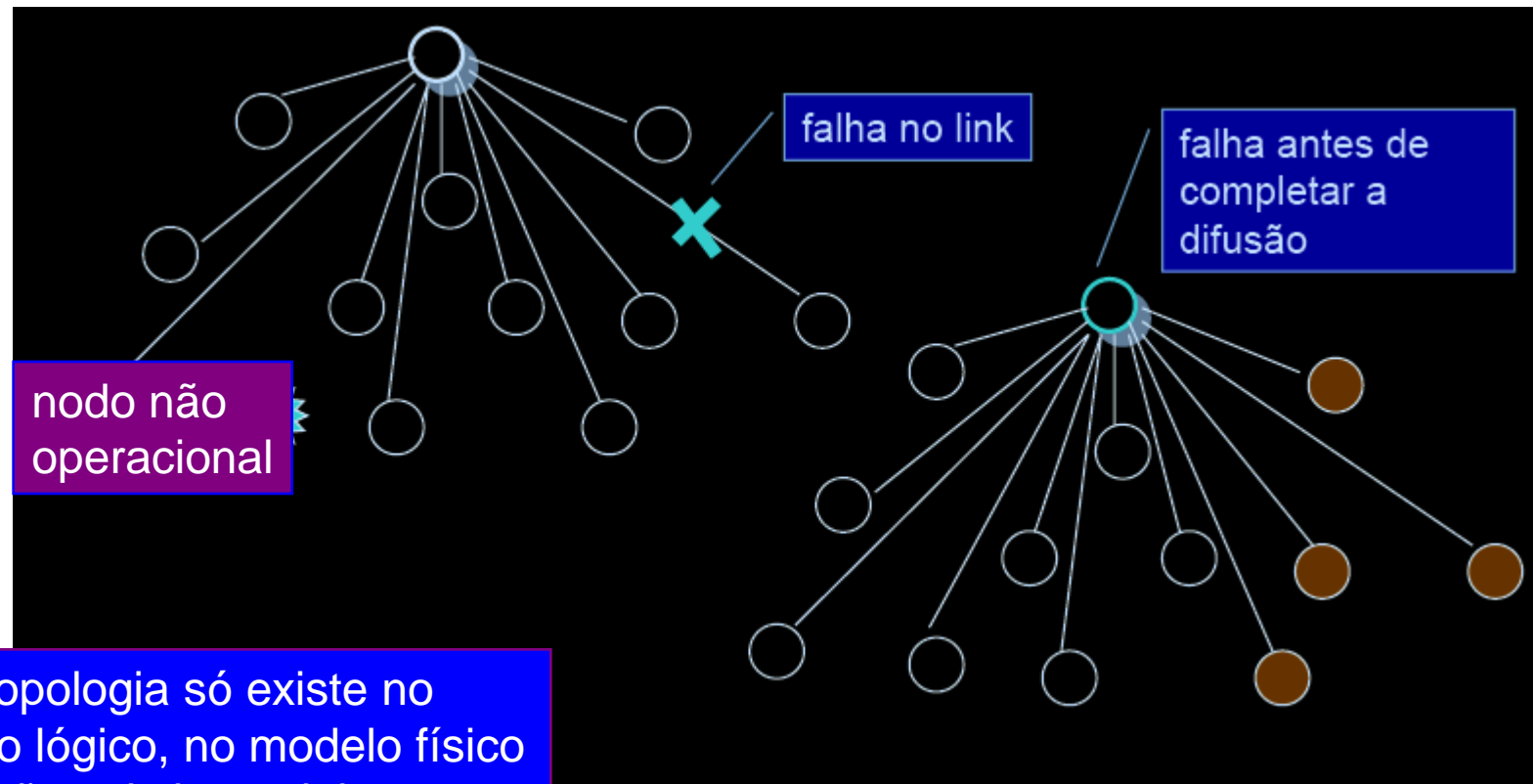
- ▣ envio de mensagens a alguns nodos do sistema

- infraestrutura de rede

- ▣ podem ser baseados em broadcast não confiável
 - ▣ ou em comunicação ponto a ponto
 - sensível a falhas de nodo e comunicação

nodo pode falhar após ter iniciado broadcast, assim alguns nodos podem ter recebido a mensagem e outros não

Problemas possíveis



essa topologia só existe no modelo lógico, no modelo físico pode não existir caminhos entre alguns nodos

Propriedades da difusão

valem tanto para broadcast como multicast

- confiabilidade

- ▣ mensagem deve ser recebida por todos os nodos operacionais

- ordenamento consistente

ordenamento consistente
é diferente de
ordenamento temporal

- ▣ diferentes mensagens enviadas para nodos diferentes são entregues na mesma ordem em todos os nodos

- preservação de causalidade

- ▣ a ordem na qual mensagens são entregues é consistente com a relação causal de envio das mensagens

mensagens sem relação causal poderiam ser entregues em qualquer ordem

Primitivas

- difusão confiável

- uma mensagem enviada é recebida em todos os nodos não falhos na rede, mesmo na presença de falhas

- difusão atômica

- suporta difusão confiável e ordenação

- difusão causal

- assegura ordenação causal
- cada primitiva tem sua aplicação
 - mensagens isoladas: difusão confiável
 - banco de dados: difusão atômica
 - uma mensagem depende de outra: difusão causal

Possibilidades da Literatura

- Sistemas síncronos:
 - ▣ Periodic Group Creator (Uso de relógio físico)
- Sistemas assíncronos:
 - ▣ Amoeba (Sequenciador)
 - ▣ Isis, Causal Blocks (Relógios Vetoriais)

Propriedades de interesse

- Message Delivery Liveness: Validade
- Message Delivery Safety: **Uniform** Total and Causal Order, **Uniform** Agreement
- View Installation Liveness: Terminação na detecção de falhas e suspeitas
- View Installation Safety: sequência única de visões

Propriedades **Uniform** são mais forte pois exigem comportamento **não** somente para **processos corretos**, e são necessárias em aplicações como de replicação ativa.

Exemplo: Periodic Group Creator

- Difusão por meio de “flooding”
 - ▣ Complexidade de mensagens $O(N^2)$
- Uso de sincronia de relógios
 - ▣ Com precisão ε suficientemente pequena
 - ▣ A entrega é baseada no timestamp dos relógios físicos
- Periodicamente os processos devem indicar seu interesse em continuar no grupo

Exemplo: Periodic Group Creator

```
1 task Membership;
2   var group: Time; members: set-of-P;
3     joined: Boolean initially false;
4   broadcast("new-group",myclock+ $\Delta$ );
5   cycle
6     when receive("new-group",V)
7     do if myclock > V then abort fi;
8       cancel(Renewal-Time);
9       broadcast("present",V,myid);
10      schedule(Renewal-Time,V+ $\pi$ ) at V+ $\pi$ - $\eta$ 
11    when receive("present",V,M)
12    do if myid  $\in$  M then joined  $\leftarrow$  true fi;
13      members  $\leftarrow$  M; group  $\leftarrow$  V od;
14  endcycle;

15 task Renewal-Time(V:Time);
16   if myclock > V then abort fi;
17   broadcast("present",V,myid);
18   schedule(Renewal-Time,V+ $\pi$ ) at V+ $\pi$ - $\eta$ ;
```

Exemplo: Periodic Group Creator

```
1. task Start;
2. const  $\Delta = \pi(\delta + \epsilon) + D_{\pi,\lambda} + \epsilon$ ;
3. var t: Time;  $\sigma: \Sigma$ ; s: P;  $\kappa: 1 \dots n$ ;
4. cycle take( $\sigma$ );
5.   t  $\leftarrow$  clock;
6.    $\kappa \leftarrow 1$ ;
7.   s  $\leftarrow$  myid;
8.   send-all(t,s, $\kappa$ , $\sigma$ );
9.    $H \leftarrow H \oplus (t,s,\sigma)$ ;
10.  schedule(End,t+ $\Delta$ ,t);
11. endcycle;
```

```
1. task Relay;
2. const  $\Delta = \pi(\delta + \epsilon) + D_{\pi,\lambda} + \epsilon$ ;
3. var t, $\tau$ : Time;  $\sigma: \Sigma$ ; l: link;  $\kappa: 1 \dots n$ ;
4. cycle receive(t,s, $\kappa$ , $\sigma$ ,l);
5.    $\tau \leftarrow$  clock;
6.   [ $\tau < t - \epsilon\kappa$ : "too early" iterate];
7.   [ $\tau > t + \kappa(\epsilon + \delta)$ : "too late" iterate];
8.   [ $\tau > t + \Delta$ : "too late" iterate];
9.   [ $t \in \text{dom}(H) \ \& \ s \in \text{dom}(H(t))$ : "deja vu" iterate];
10.  send-all-but(l,(t,s, $\kappa + 1$ , $\sigma$ ));
11.   $H \leftarrow H \oplus (t,s,\sigma)$ ;
12.  schedule(End,t+ $\Delta$ ,t);
13. endcycle;
```

```
1. task End(t: Time);
2. var p: P; val:  $P \rightarrow \Sigma$ ;
3. val  $\leftarrow H(t)$ ;
4. while  $\text{dom}(\text{val}) \neq \emptyset$ ;
5. do p  $\leftarrow \min(\text{dom}(\text{val}))$ ;
6.   deliver(val(p));
7.   val  $\leftarrow \text{val} \setminus p$ ;
8. od;
9.  $H \leftarrow H \setminus t$ ;
```

Exemplo: Amoeba

- Uso de paradigma de Sequenciador
- Ambientes Assíncronos

Exemplo: Amoeba

- Envio da mensagem ao sequenciador
- Mensagem é enviada ao sequenciador
- Sequenciador define relógio lógico
- Envia a mensagem a todos

Exemplo: Amoeba

- Problemas?
 - ▣ Falha do sequenciador?
 - **Eleição de novo Líder**
 - ▣ Perda de mensagens?
 - Uso de NACK para retransmissão, se falta mensagem na sequencia

- ▣ Pedidos de mudanças de visão são enviados ao sequenciador, o qual pode enviar uma nova mensagem ao grupo na sequencia indicando a nova visão

Exemplo: Amoeba

□ Problemas?

▣ Entrega Uniforme?

- A entrega só pode acontecer quando todos sabem que a mensagem foi recebida por todos
 - Uso de ACK para confirmar o recebimento e de COMMIT para entregar a mensagem

▣ Em caso de falha do líder, além da eleição deve-se entregar as mensagens não entregues ainda a todos do grupo: **discussão em detalhes em outro protocolo!**

▣ Uso de **Detector de Defeitos**

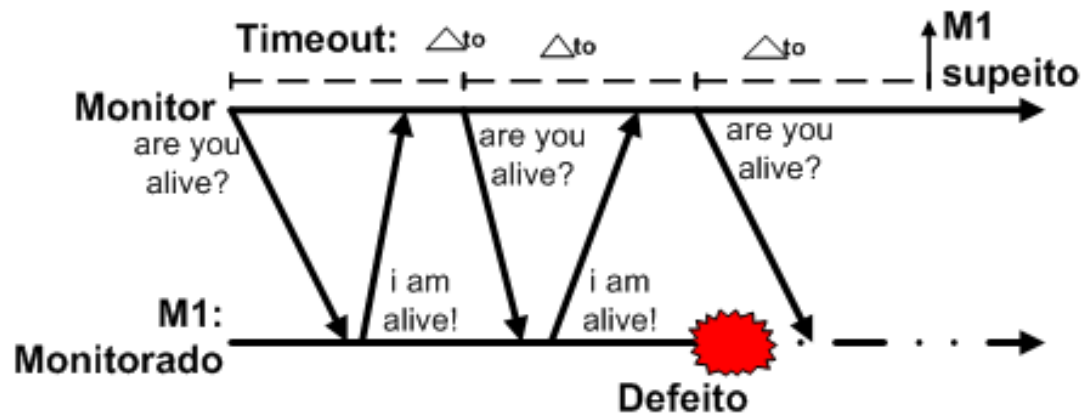
PIGGYBACKYING

APLICAÇÕES EM SISTEMAS DISTRIBUÍDOS: DETECTORES DE DEFEITOS E ELEIÇÃO

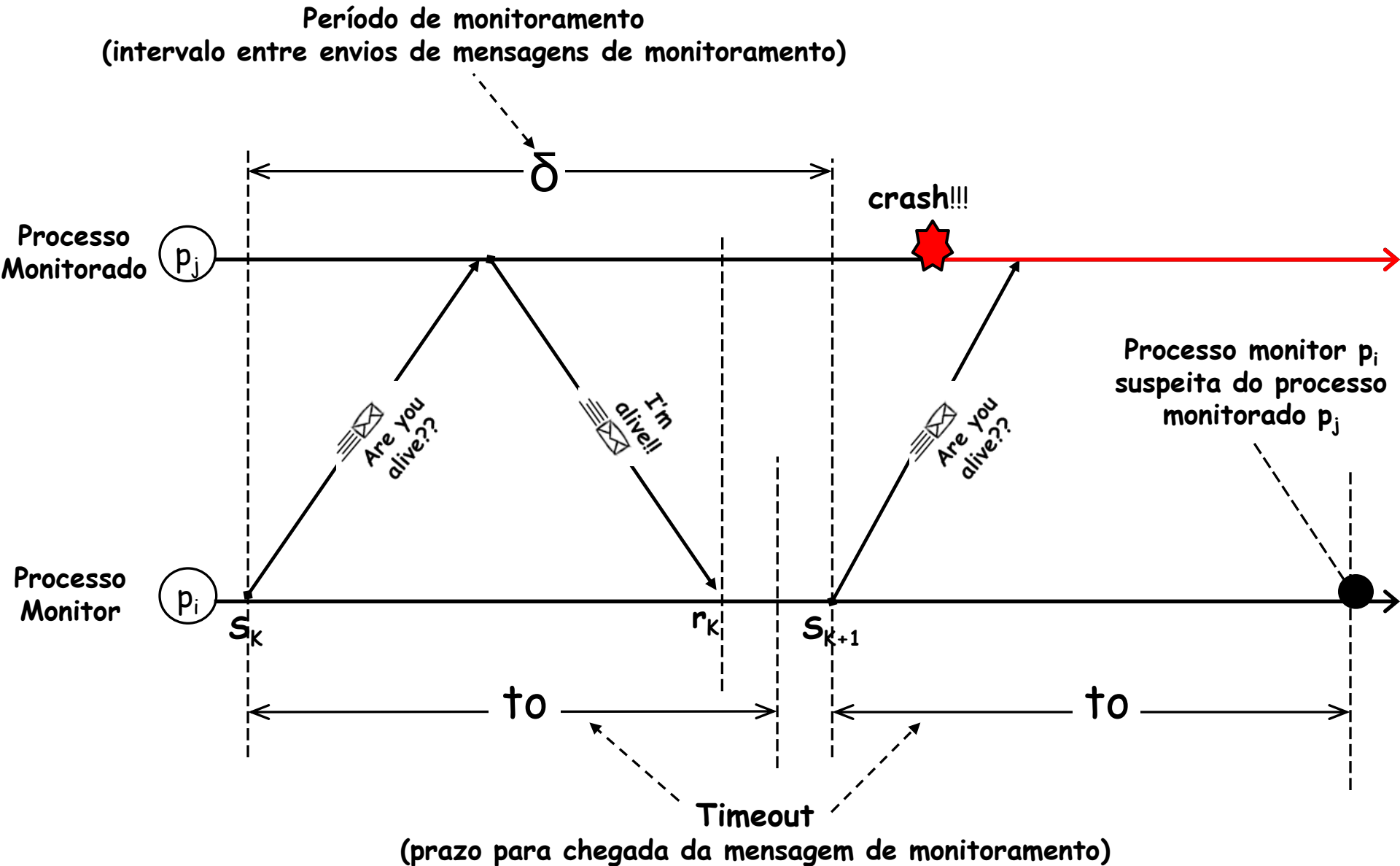


Detectores de Defeitos

- Detectores de defeitos são blocos básicos na construção de mecanismos de tolerância a falhas:
 - O serviço de detecção de defeitos é fundamental para ativar procedimentos de recuperação e/ou reconfiguração do sistemas no caso de falhas de componentes.
- Em ambientes distribuídos a detecção é realizada através de troca de mensagens.

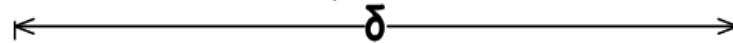


Response	Percentage
Yes	85%
No	15%



Detecção de defeitos em sistemas distribuídos

PERÍODO DE MONITORAMENTO
(INTERVALO ENTRE ENVIOS DE MENSAGENS DE MONITORAMENTO)



Tempo de detecção no pior caso :

TD depende de δ e t_o

Em ambientes abertos (e.g. Internet) determinar o **timeout de detecção** ou o **período monitoramento** é um problema:

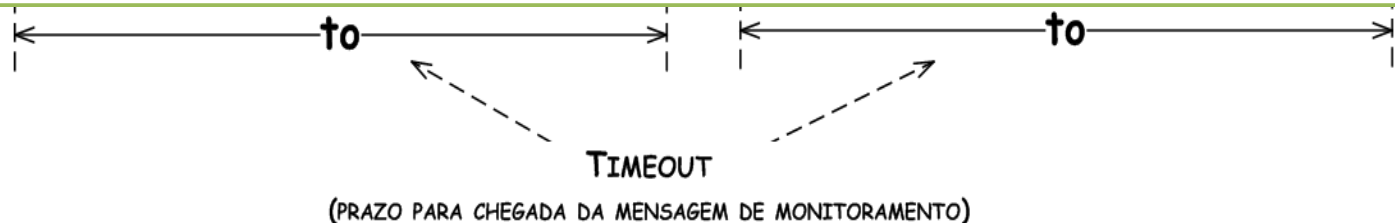
Não se conhece com certeza os tempos de processamento e viagem das mensagens ou recursos disponíveis

timeout pode ser muito curto (\Rightarrow falsas suspeitas);

timeout pode ser muito longo (\Rightarrow detecção lenta);

período pode ser muito curto (\Rightarrow consome mais, pode ter mais suspeitas);

período pode ser muito longo (\Rightarrow detecção lenta);



Detectores de defeitos

- Os parâmetros do detector de defeitos dependem da percepção do ambiente:
 - ▣ Podem ser calculados em tempo de projeto
 - ▣ Ou adaptados de forma dinâmica, conforme monitoramento do ambiente
- Chandra e Toueg (1996) apresentam 8 classes de Detectores de Defeitos Não Confiáveis, baseado na Qualidade de Serviço que o Detector provê, de acordo com o ambiente
- Destacaremos o de classe P e o de classe $\leftrightarrow S$

- Detectores de Defeitos Não Confiáveis
 - ▣ Definidos por meio de 2 propriedades:
 - Abrangência – quantidade de falhas detectadas
 - Exatidão – quantidade de falsas suspeições cometidas
 - ▣ *Aplicações são definidas em função da especificação do Detector de Defeitos*
 - ▣ Detectores de falhas **perfeitos**
 - ✗ Abrangência forte – em algum momento, todo processo falho será considerado suspeito, permanentemente, por qualquer processo correto;
 - ✗ Exatidão forte – nenhum processo correto será suspeitado por outro processo correto.

- A semântica forte nem sempre pode ser obtida:

EM TERMOS DE ABRANGÊNCIA:

- × Abrangência fraca – em algum momento, todo processo falho será considerado suspeito, permanentemente, por *algum* processo correto;

EM TERMOS DE EXATIDÃO:

- × Exatidão fraca – *algum* processo correto nunca é suspeitado;
- × Exatidão forte eventual – *em algum momento*, o detector garante a exatidão forte;
- × Exatidão fraca eventual – *em algum momento*, o detector garante a exatidão fraca.

Detectores de Defeitos

- Um ambiente síncrono permite o uso de detectores perfeitos com propriedades da classe P
- Sistemas puramente assíncronos não permitem a rigor garantir exatidão e abrangência
- Em ambientes parcialmente síncronos, com a hipótese GST, os detectores de defeitos tem propriedades da classe $\langle \rangle S$, ou seja:
 - *em algum momento do tempo (GST) algum processo correto nunca é suspeito e todo processo falho é considerado suspeito por este processo correto.*
 - **Isto é suficiente para o progresso de algoritmos como o Consenso.**

Eleição de Líder

- Algoritmo do Bully (Valentão)
- Algoritmo em Anel

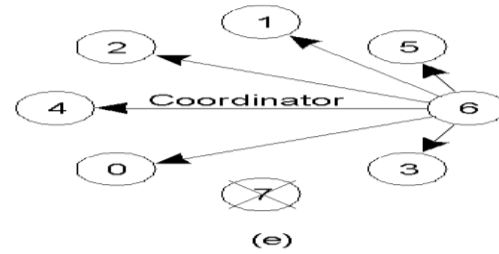
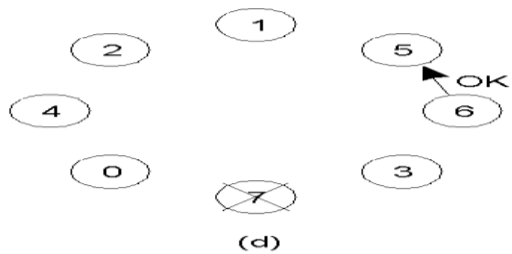
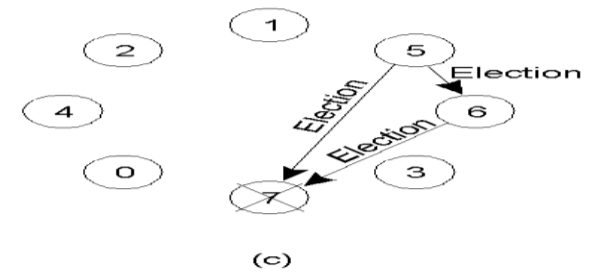
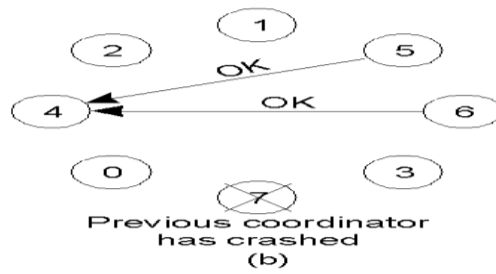
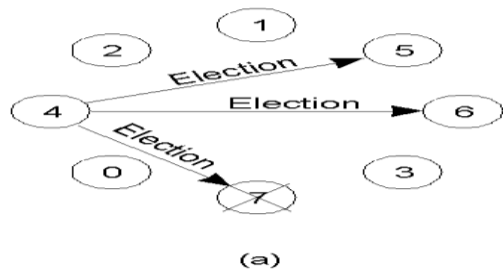
O algoritmo do Bully

- Hipóteses de falhas
 - ▣ Crash-Recovery
 - ▣ Sem falhas de canais (canais confiáveis)

O algoritmo

- Cada processo tem um ID único, o de maior ID deve ser o líder
- Quando um processo suspeita que o líder é falho, inicia uma eleição para os demais processos
- Se um processo de maior ID recebe pedido de eleição de um de menor ID, o retira da eleição e se candidata em uma nova eleição
- Se um processo propõe uma eleição e não é retirado da mesma, torna-se o líder e informa ao grupo

Exemplo de Execução



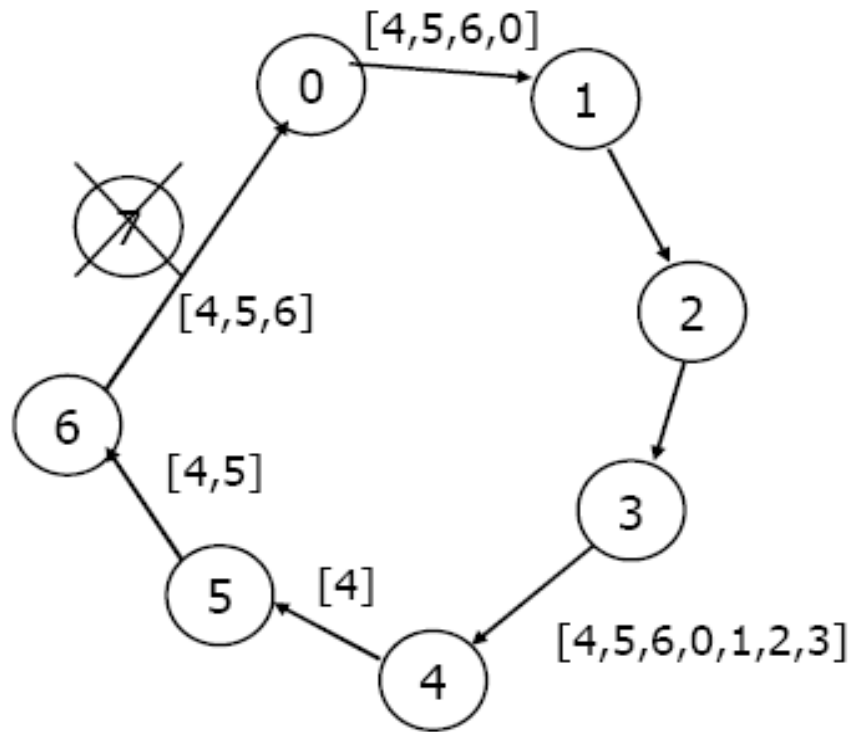
Propriedades do algoritmo

- Em um grupo de n processos
 - ▣ Pior Caso:
 - O processo de menor ID inicia a eleição
 - $O(n^2)$ mensagens
 - ▣ Melhor Caso:
 - “Eventual leader” inicia a eleição
 - Requer $(n-1)$ mensagens
- ▣ Em sistemas assíncronos quais as implicações deste algoritmo?

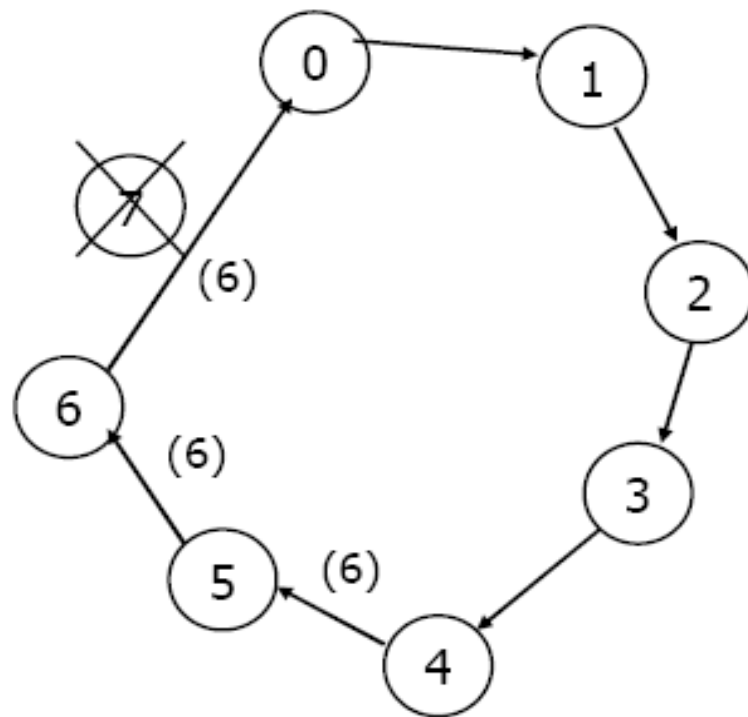
Algoritmo em Anel

- Os IDs dos processos são obtidos pela organização dos mesmos em um anel lógico. O processo de maior ID deve ser eleito líder.
- Cada processo possui um sucessor
 - ▣ Initiation:
 - Um processo envia uma ELECTION message ao seu sucessor (**ou ao próximo processo ativo no anel**) with its ID
 - Cada processo adiciona seu próprio ID e encaminha a ELECTION message
 - ▣ Leader Election:
 - A ELECTION message retorna a quem iniciou a eleição
 - Quem inicia a eleição anuncia por meio de uma nova mensagem o líder ao anel

Algoritmo em Anel: Initiation



Algoritmo em Anel - Election



Algoritmo em Anel: Propriedades

- Se apenas 1 processo inicia a eleição:
 - ▣ Requer $2n$ mensagens
- Dois ou mais processos podem iniciar eleições simultâneas
 - ▣ Mensagens extras
 - ▣ Mantém o mesmo resultado do líder

APLICAÇÕES EM SISTEMAS DISTRIBUÍDOS: CONSENSO



Consenso

- O propósito do Consenso é obter um valor comum para a computação distribuída
- Ex:
 - ▣ Nova Visão do Grupo
 - ▣ Executar o COMMIT ou não?
- Depende do modelo do sistema.

Consenso

- Um conjunto de processos deve concordar com um mesmo valor proposto ao final do consenso, sendo que cada processo pode ter um valor proposto inicial diferente
 - Terminação – todos os processos terminam o consenso;
 - Validade – o valor acordado é o valor proposto inicial de algum processo;
 - Acordo – todos os processos que terminam o consenso sem falhar decidem pelo mesmo valor proposto.

Consenso

- Coordenação, ordenação, eleição de líderes, definição de membros de grupo de comunicação (*membership*); terminação atômica; etc.
- Sistemas Síncronos – existem soluções para o consenso.
- Sistemas Assíncronos – o consenso é impossível na ocorrência de falhas, mesmo que apenas um processo falhe por colapso (Fisher, Lynch e Paterson 1985).

Consenso

- Consenso Assíncrono

- ▣ Assume em geral que há uma maioria que não falha

- Consenso Síncrono

- ▣ Como há uma detecção perfeita, pode-se excluir exatamente o processo defeituoso

Consenso Síncrono

- F processos que podem falhar
- Rounds síncronos
- Utilizando difusão atômica confiável (rmcast), o processo p_i no round i difunde a sua proposta de valor v_i

Consenso Síncrono

- Agrupa os valores propostos em V (o rmcast garante que todos receberam o mesmo conjunto de propostas)
- Após $F+1$ rounds, há ao menos 1 processo correto que enviou a sua proposta para V recebida por todos

Consenso Síncrono

- `ROUND = 1`
- `WHILE ROUND <= f+1 {`
 - ▣ `If (round = i) && (myID == i)`
 - `Reliable-multicast(v_i)`
 - `Wait until timeout: round ++`
- `}`
- `valor = min(V)`

- `On receive(v_i):`
 - ▣ `V = V U v_i`

Consenso Assíncrono

- Paxos - Lamport
 - ▣ Tem um artigo chamado “Paxos made simple”
- Consenso para sistemas “crash-recovery” sem limite temporal
- Assume uma maioria de processos corretos.

- Envia a Proposta
- A proposta deve ser aceita por uma maioria (PREPARE REQUEST)
- Após ser aceita é confirmada (ACCEPT REQUEST)
- Papéis: Proponente, Acceptor, Learner

Paxos

- Um accepter concorda (ACK) com o PREPARE de uma proposta X, se não concordou previamente com nenhuma proposta maior que X
- Somente com maioria de ACKs é enviado o ACCEPT da proposta X
- Um accepter aceita o valor de um ACCEPT da proposta X se não aceitou previamente nenhuma proposta maior que X

Algoritmo do Paxos

- Fase 1a: Preparar

Proposer seleciona proposta X e envia mensagem PREPARE para os Acceptors.

- Fase 1b: Promise

Acceptor promete aceitar proposta X se NÃO SE COMPROMETEU com nenhuma proposta $\leq X$, senão envia NACK.

Algoritmo do Paxos

□ Fase 2a: Accept

Se Proposer recebe maioria de ACKs para proposta X, escolhe um valor para consenso.

Se algum Acceptor do quórum aceitou algum valor previamente, o Proposer deve escolher um desses valores. Senão, pode escolher qualquer valor.

Proposer envia mensagem Accept para os Acceptors, com o valor escolhido.

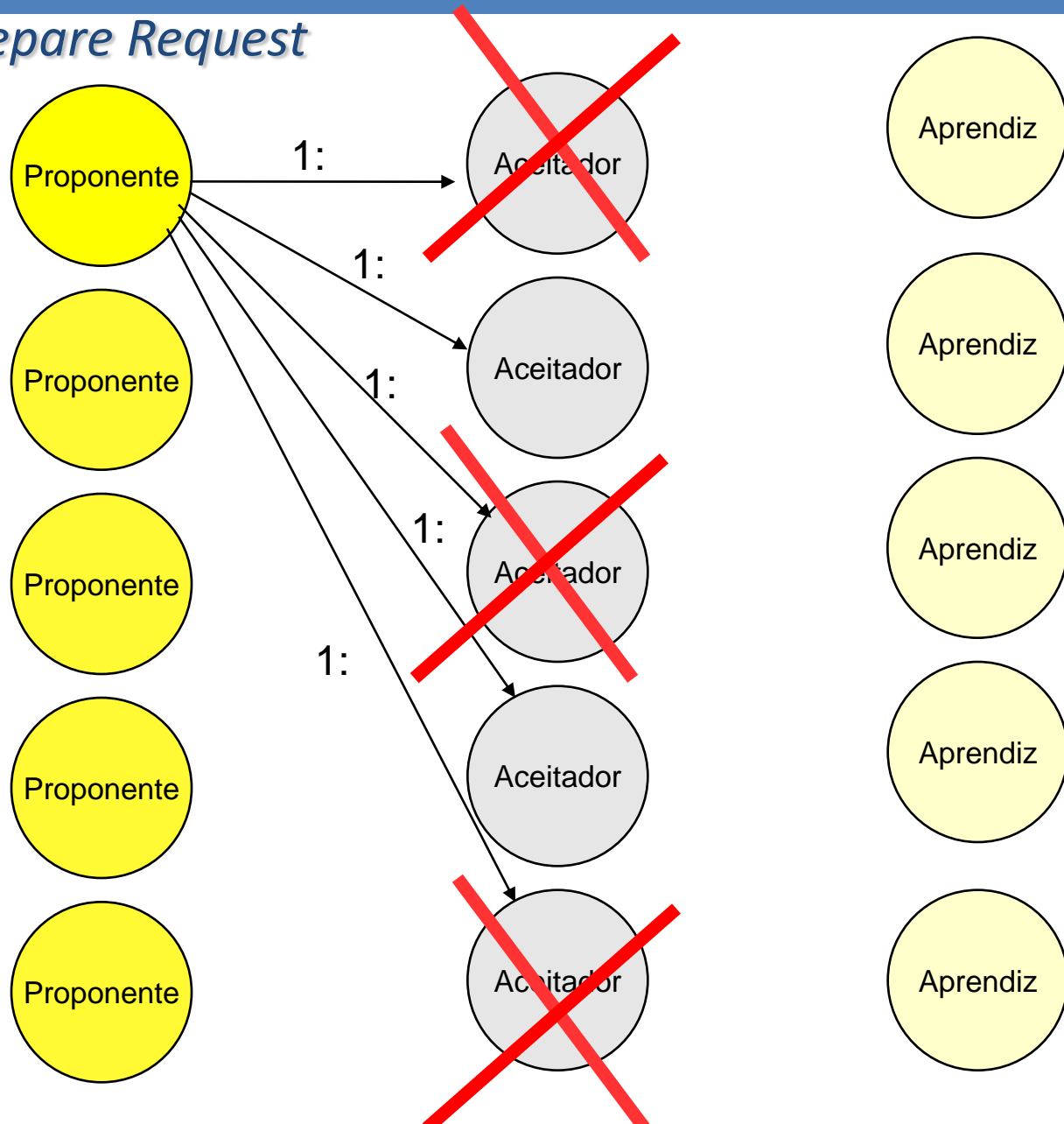
□ Fase 2b: Accepted

Acceptor aceita o valor.

Cada Acceptor envia uma mensagem Accepted para o Proposer e para cada Learner.

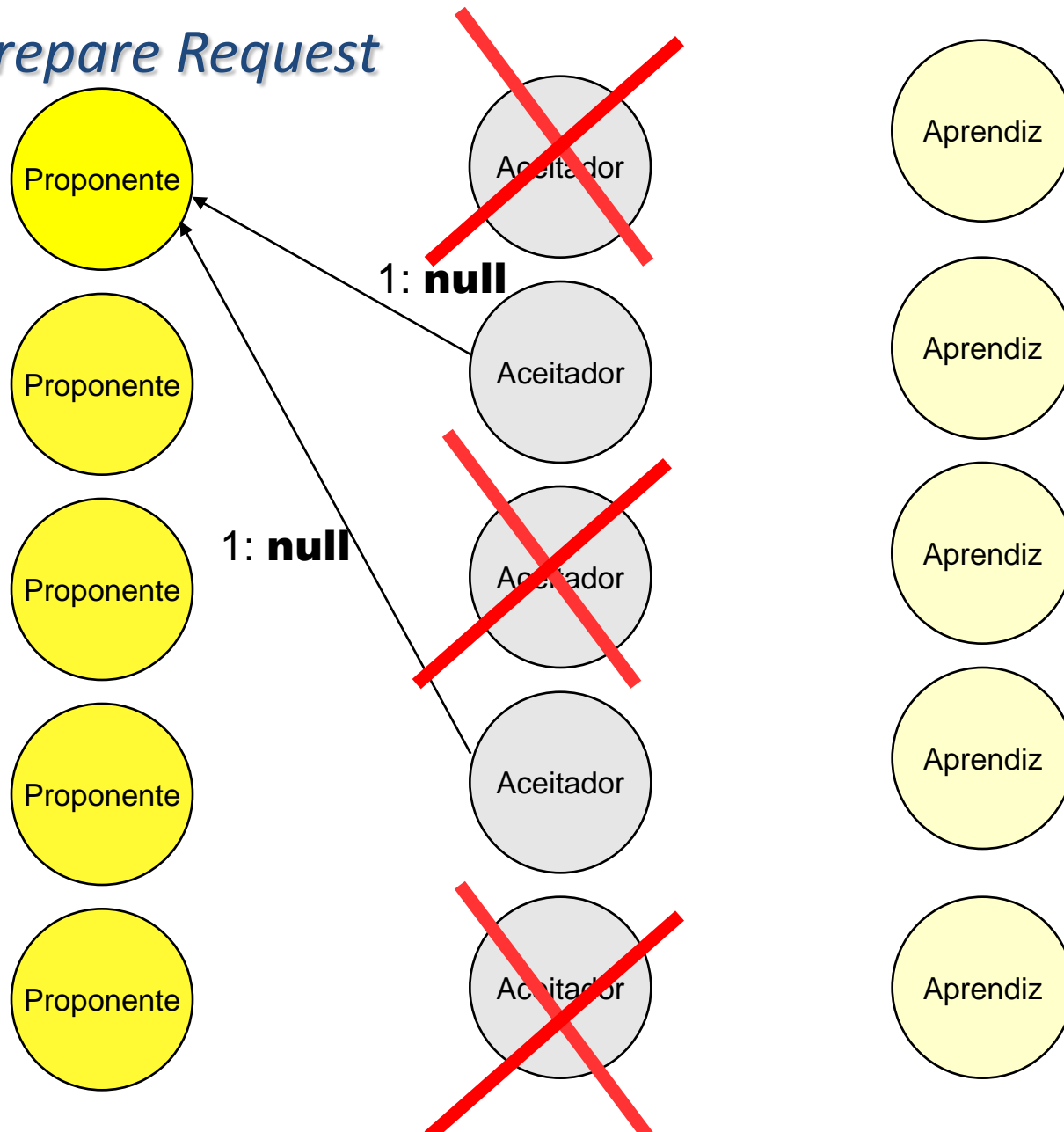
1ª Tentativa para o Consenso

1ª Fase: Prepare Request



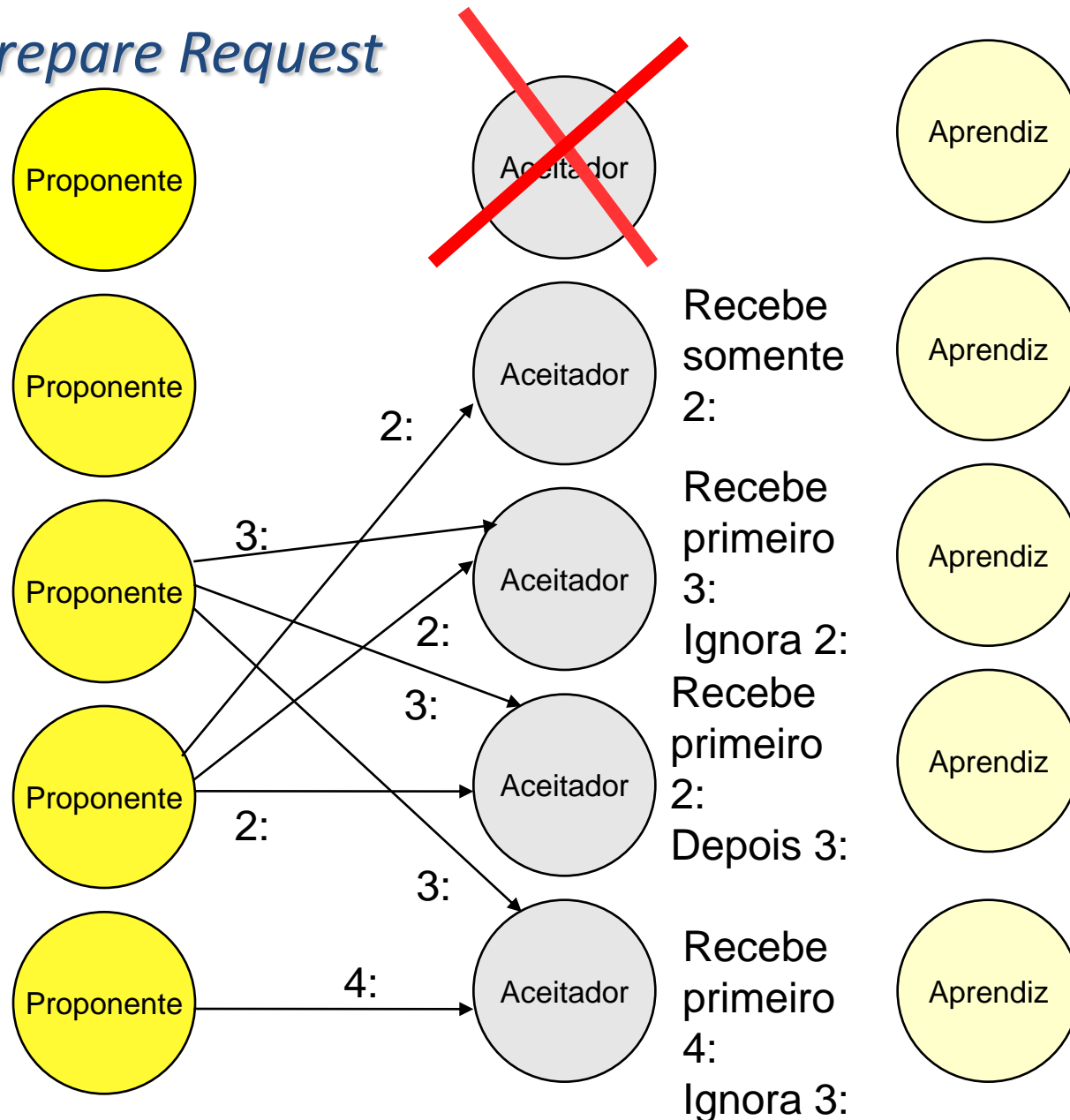
1ª Tentativa para o Consenso

1ª Fase: Prepare Request



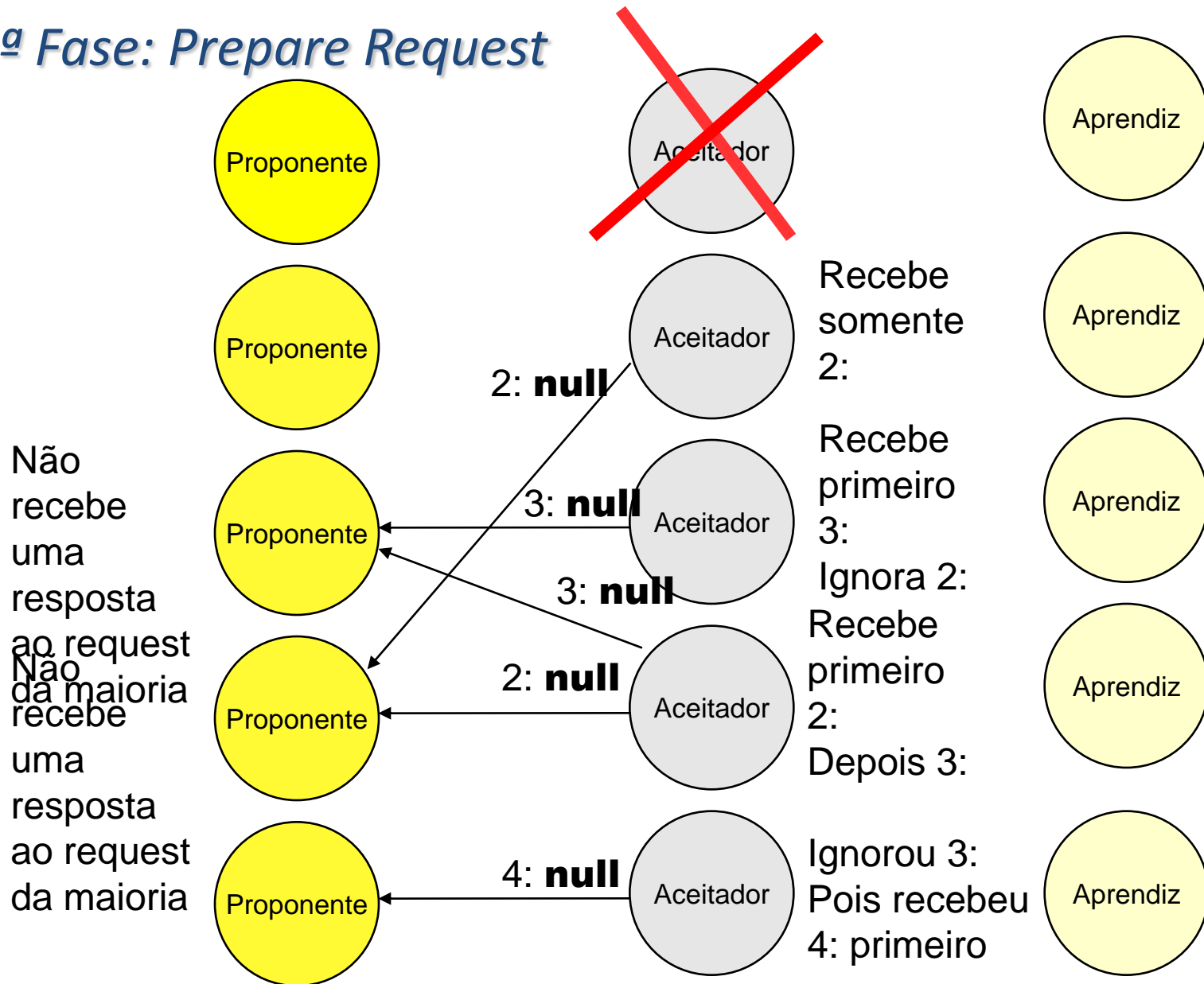
2ª, 3ª Tentativa para o Consenso (e início da 4ª)

1ª Fase: Prepare Request



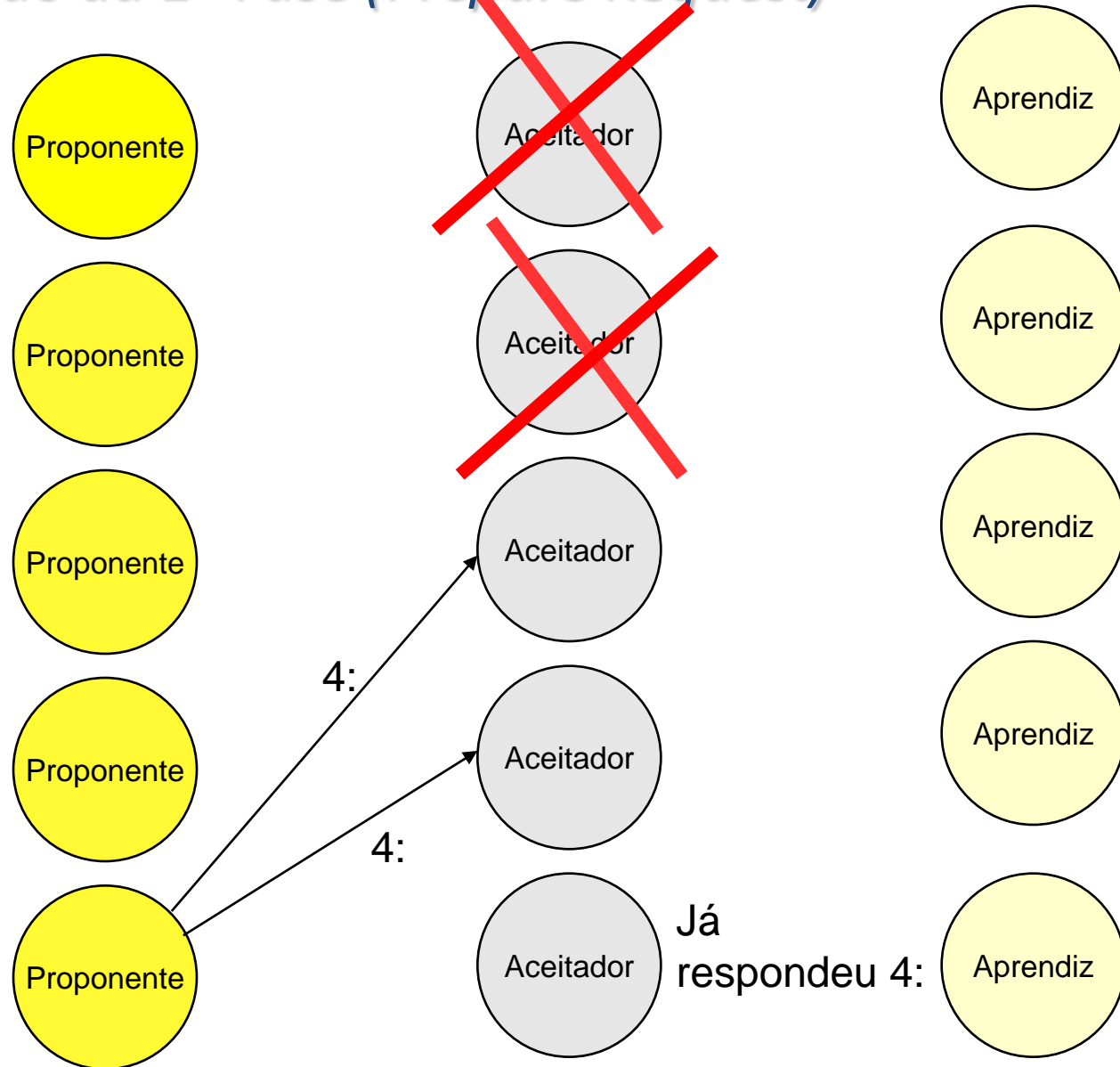
2ª e 3ª Tentativa para o Consenso (e início da 4ª)

1ª Fase: Prepare Request



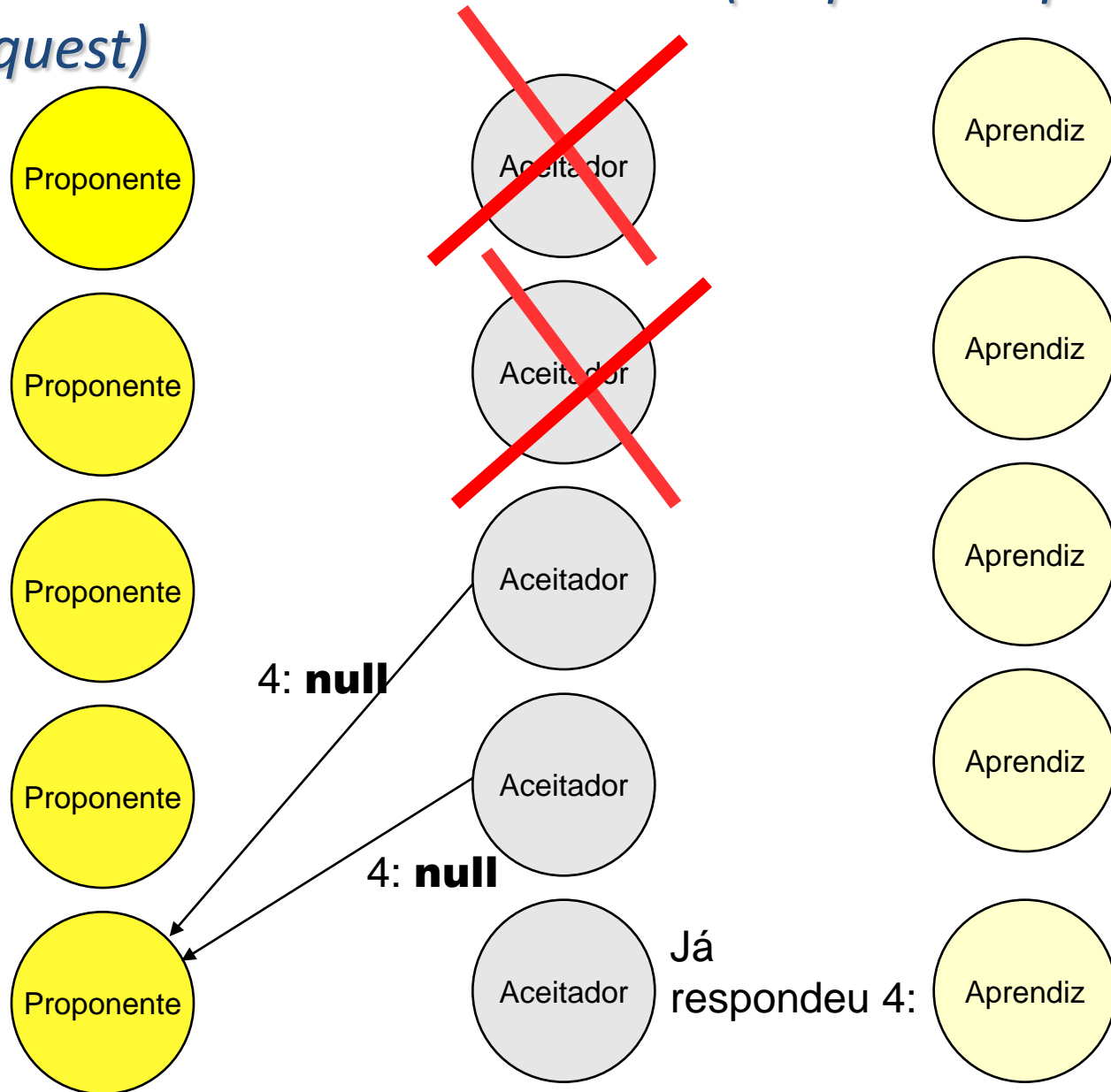
4ª Tentativa para o Consenso

Continuação da 1ª Fase (Prepare Request)



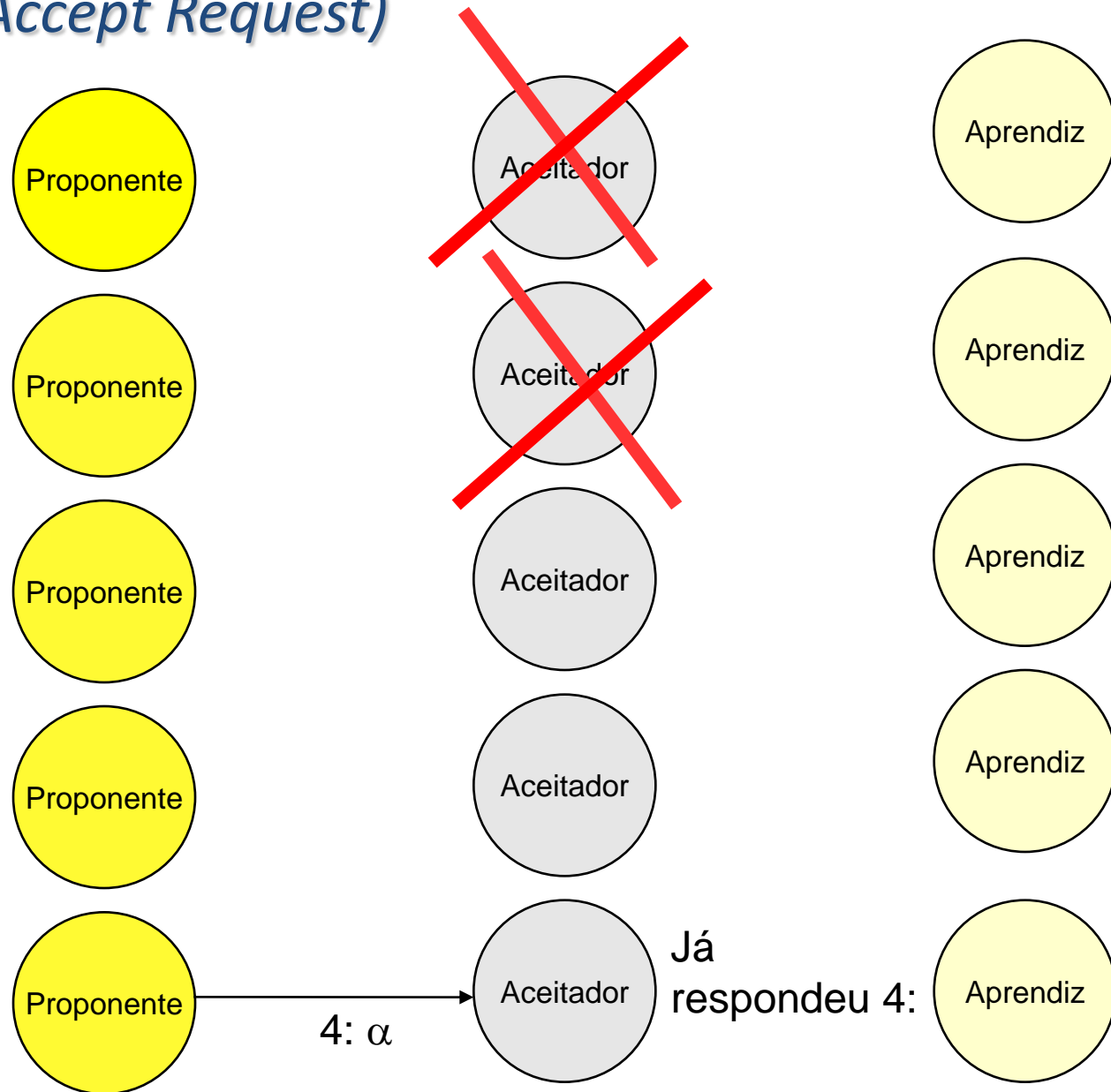
4ª Tentativa para o Consenso

Continuação da 1ª e início da 2ª Fase (Prepare Request e Accept Request)



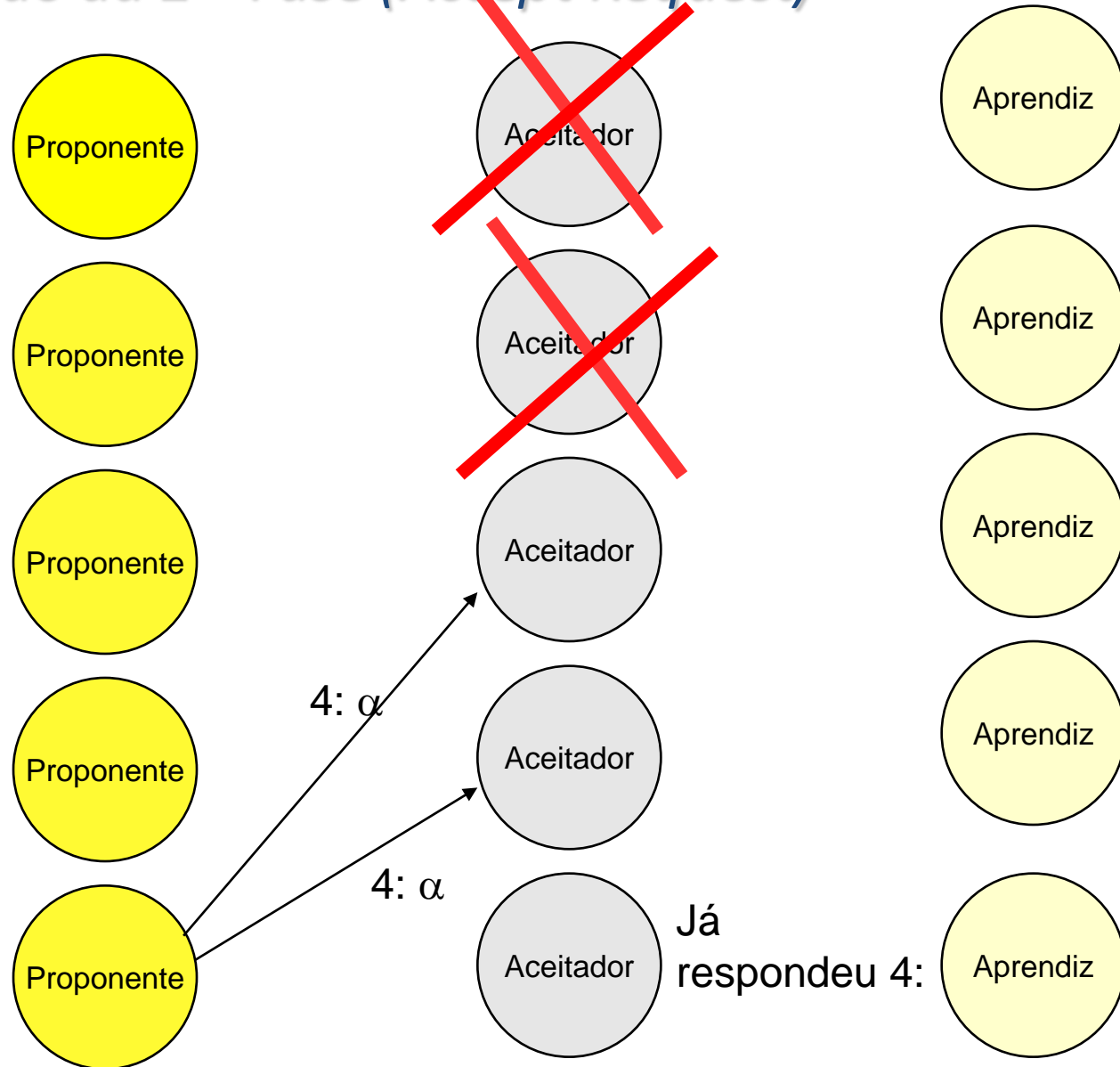
4ª Tentativa para o Consenso

2ª Fase (Accept Request)



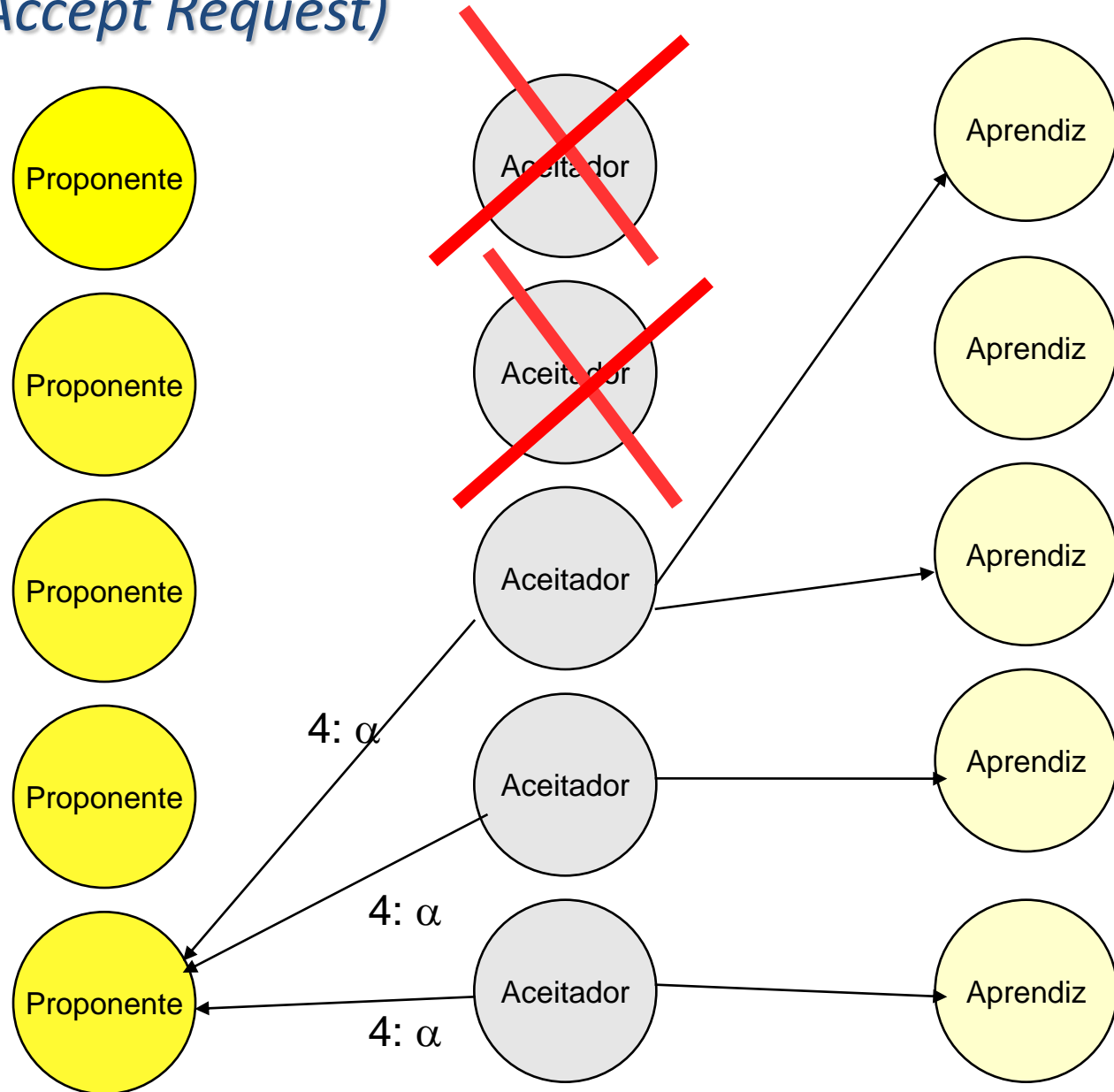
4ª Tentativa para o Consenso

Continuação da 2ª Fase (Accept Request)



4ª Tentativa para o Consenso

2ª Fase (Accept Request)



Referências

- Parte dos slides (AULA 1, 2 e 3) foram cedidos pelo prof. Sérgio Gorender
- Parte dos slides (AULA 4: Difusão Atômica Confiável) foi baseada em apresentação: ÂNGELO, Santo. URI/DECC. “Difusão de Mensagens: Broadcast confiável, atômico e causal”
- Parte dos slides (AULA 5: Eleição) foi baseada em apresentação: SRINIVASAN, Adith. “Election Algorithms”
- Parte dos slides (AULA 5: detectores de defeitos) foi cedido pelo prof. Alírio Sá e parte dos slides de Detectores de Defeitos Não Confiáveis foi baseada em apresentação: SAMPAIO, Livia. LSD/UFCG. “Unreliable Failure Detectors for Reliable Distributed Systems” 2005