Comunicação entre Sistemas Distribuídos

Programação distribuída

Sockets:

Uma interface local, criada e possuida pelas aplicações, controlada pelo S.O. atráves do qual os processos podem trocar mensagens.

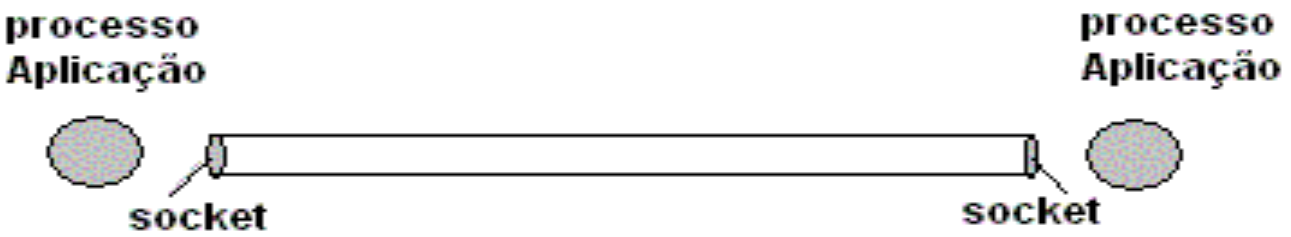
Comunicação entre processos que se utiliza das funcionalidades dos protocolos de rede para realizar a troca de informações entre emissor e receptor.

A programação por portas (sockets) se utiliza dos serviços de redes, sejam eles orientados ou não orientados a conexão.

Programação distribuída

Sockets:

Um socket pode ser entendido como uma porta de um canal de comunicação que permite a um processo executando em um computador enviar/receber mensagens para/de outro processo que pode estar sendo executado no mesmo computador ou num computador remoto.



Programação distribuída

Sockets:

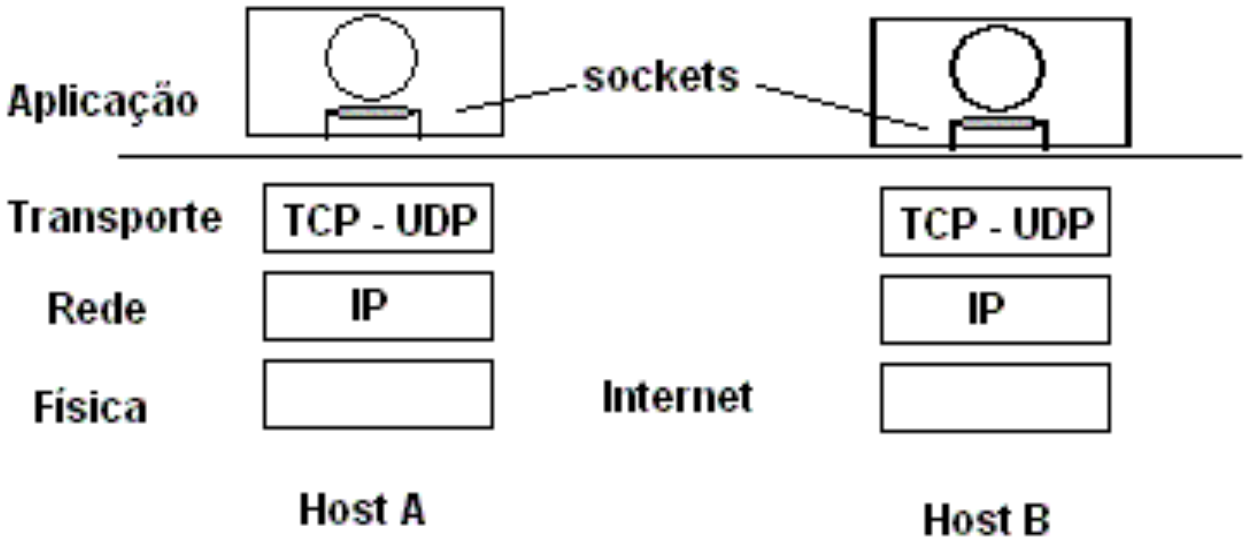
Tipos de serviço de transporte:

- Datagrama - transporte não orientado a conexão e sem controle de erros (protocolo UDP)
- DataStream - transporte orientado a conexão com controle de erros (protocolo TCP)

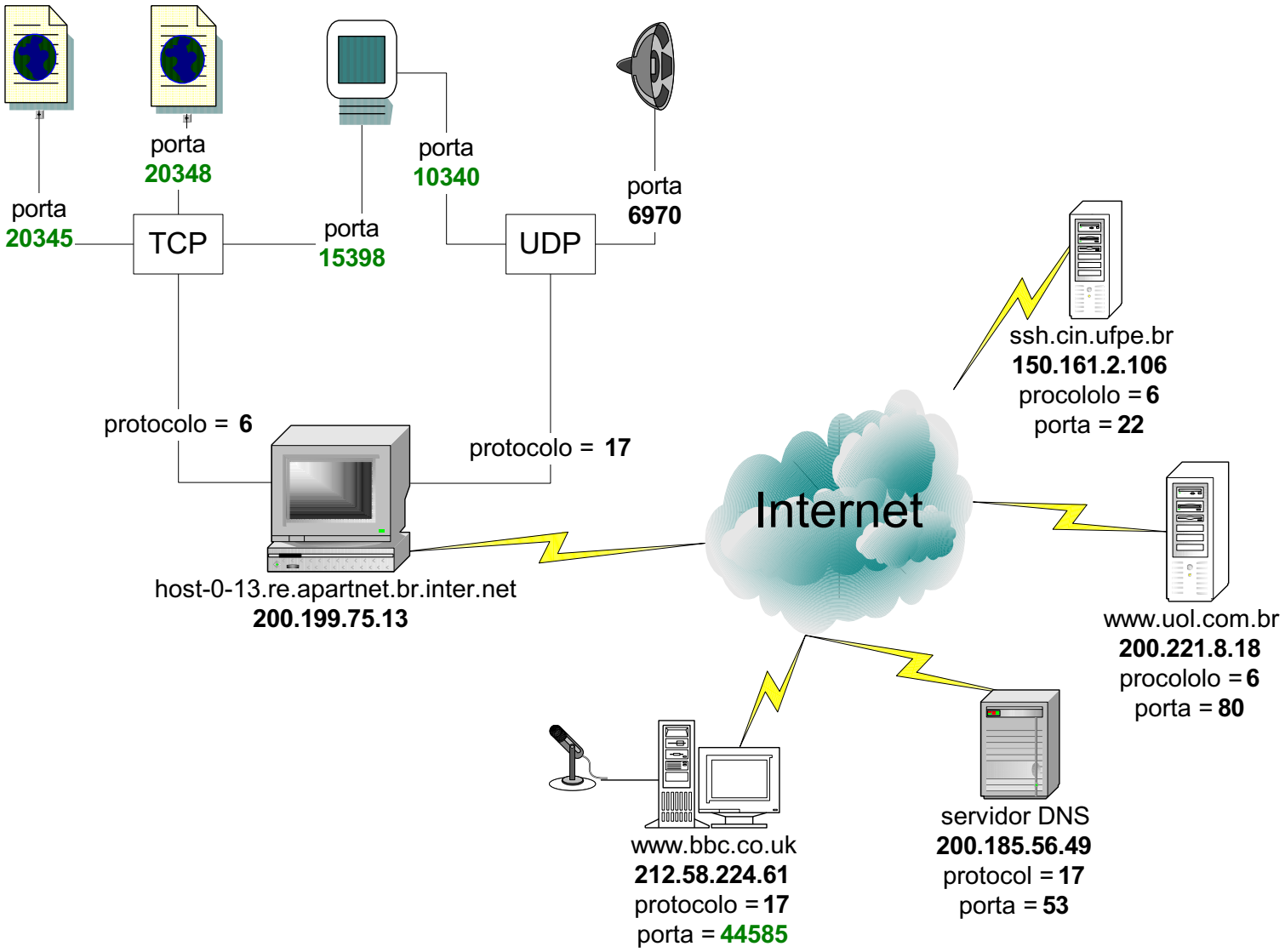
Programação distribuída

Sockets:

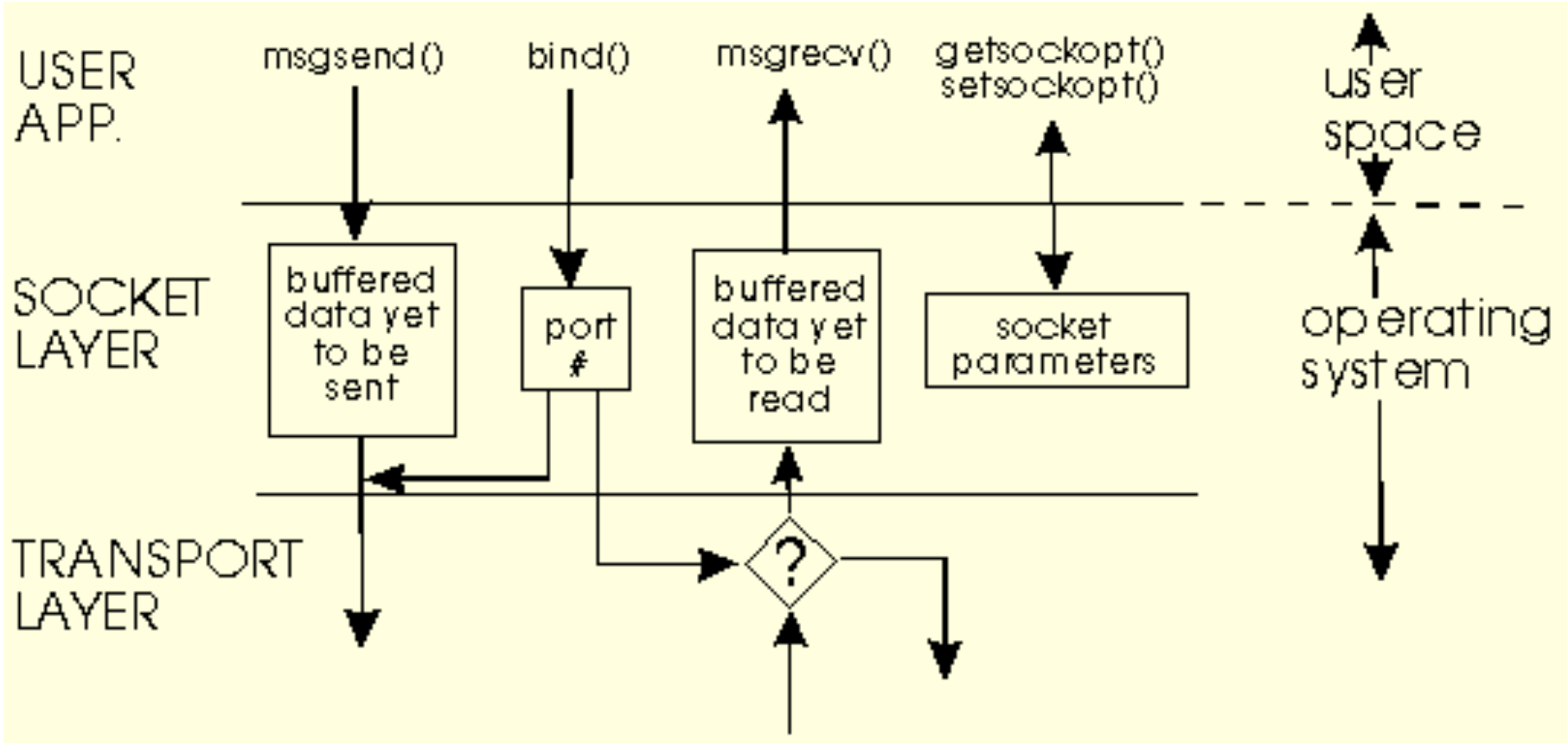
Abaixo temos uma figura com que representa a comunicação de sockets e a pilha TCP/IP



Identificação de aplicações



Sockets - visão conceitual



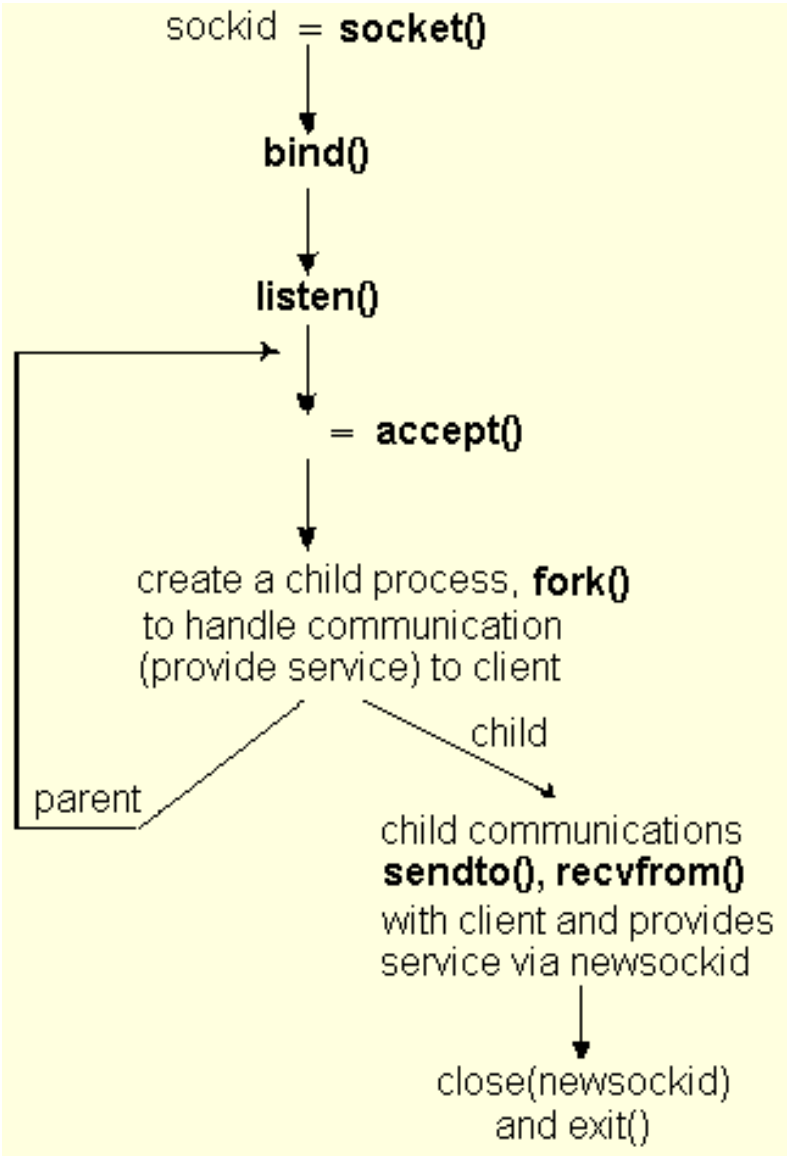
Tipos de sockets

- Serviço com conexão
  - Implementa um *stream* de dados (SOCK_STREAM)
  - Protocolo TCP (tipicamente)
- Serviço sem conexão
  - Implementa um serviço de datagramas (SOCK_DGRAM)
  - Protocolo UDP (tipicamente)
  - Acessa diretamente a camada de rede (SOCK_RAW)
- Serviço de baixo nível
  - Protocolo IP (tipicamente

Principais funções da API

socket	Cria um novo descritor para comunicação
connect	Iniciar conexão com servidor
write	Escreve dados em uma conexão
read	Lê dados de uma conexão
close	Fecha a conexão
bind	Atribui um endereço IP e uma porta a um socket
listen	Coloca o socket em modo passivo, para “escutar” portas
accept	Bloqueia o servidor até chegada de requisição de conexão
recvfrom	Recebe um datagrama e guarda o endereço do emissor
sendto	Envia um datagrama especificando o endereço

Estrutura Típica de um Servidor



Números de portas

- 1-255 reservadas para serviços padrão portas “bem conhecidas”
- 256-1023 reservado para serviços Unix
- 1-1023 Somente podem ser usadas por usuários privilegiados (super-usuário)
- 1024-4999 Usadas por processos de sistema e de usuário
- 5000- Usadas somente por processos de usuário

Sockets em Java

- Java modernizou a API para trabalhar com sockets
- O programador não precisa chamar todas as funções, algumas chamadas são automáticas
- Exemplos
  - Socket: equivalente a *socket* e *bind*
  - ServerSocket: equivalente a *socket*, *bind* e *listen*
- Sockets são implementados no pacote ***java.net***
- A transmissão e o envio de dados são feitos através de classes do pacote ***java.io*** de maneira semelhante à escrita e leitura em arquivos
  - Classes *DataInputStream*, *DataOutputStream*, etc.,

Sincronização em Sistemas Distribuídos

Eventos e relógios

- A ordem de eventos que ocorrem em processos distintos pode ser crítica em uma aplicação distribuída (ex: *make*, protocolo de consistência de réplicas).
- Em um sistema com *n* computadores, cada um dos *n* cristais terá uma frequência própria, fazendo com que os *n* relógios percam seu sincronismo gradualmente.

Relógios lógicos

- Princípios:
  1. Somente processos que interagem precisam sincronizar seus relógios.

     » Ordenação parcial de eventos

  2. Não é necessário que todos os processos observem um único tempo absoluto; eles somente precisam concordar com relação à ordem em que os eventos ocorrem.

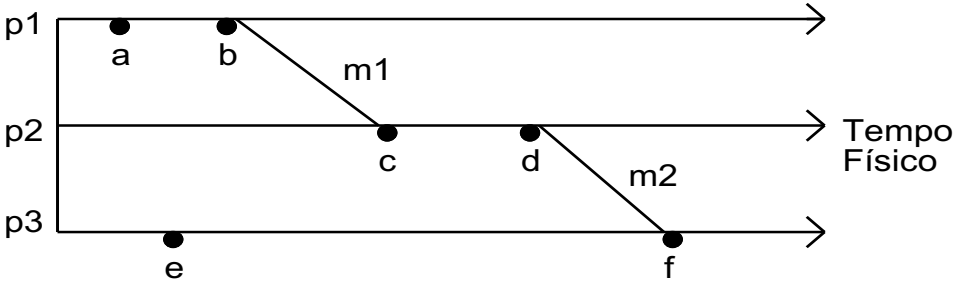
     » Ordenação causal potencial

Relógios lógicos (cont.)

- Relação **acontece-antes** (*-»*):
  1. Sejam *x* e *y* eventos num mesmo processo tal que *x* ocorre antes de *y*. Então *x -» y* é verdadeiro.
  2. Seja *x* o evento de uma mensagem a ser enviada por um processo, e *y* o evento dessa mensagem ser recebida por outro processo. Então *x -» y* é verdadeiro.
  3. Sejam *x, y* e *z* eventos tal que *x -» y* e *y -» z.* Então *x -» z* é verdadeiro.

Relógios lógicos (cont.)

Eventos ocorrendo em três processos:



- a -» b, c -» d, e -» f, b -» c, d -» f
- a -» f
- Os processos "a" e "e" são concorrentes: a || e

Relógios lógicos (cont.)

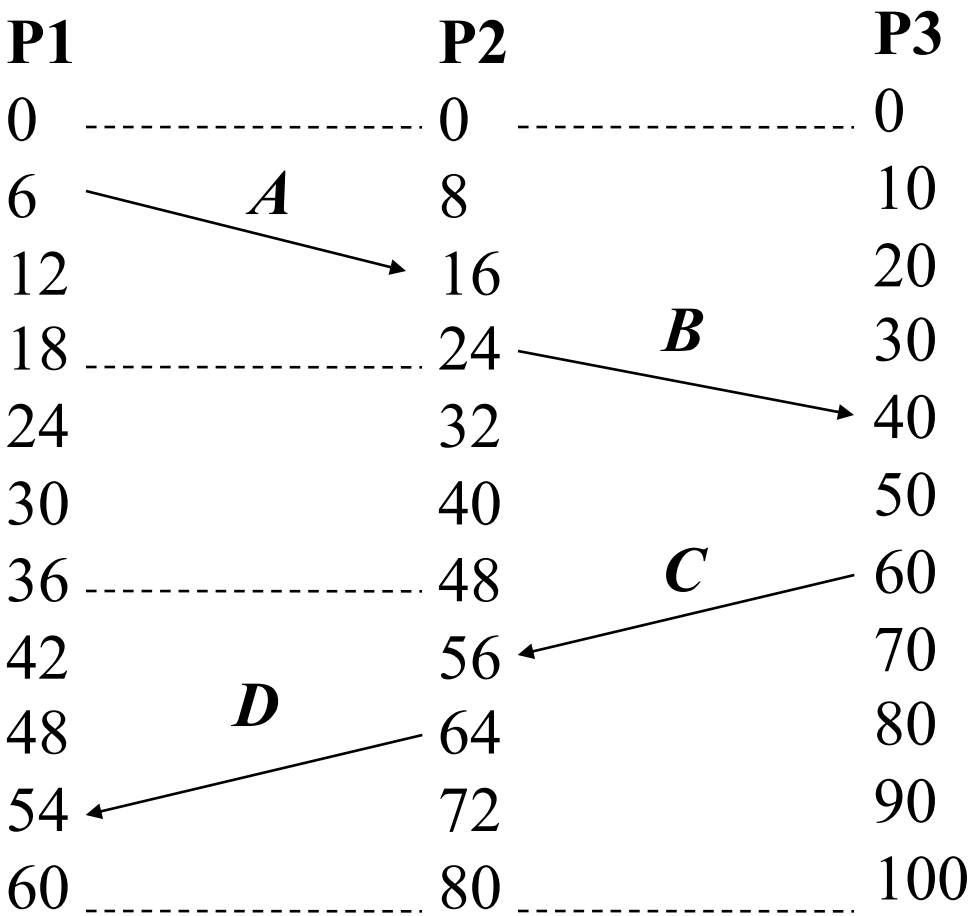
- Implementação: Cada processo *p* mantém seu próprio relógio lógico (um contador, por software), *Cp*, usado para fazer timestamp de eventos. *Cp(x)* denota o timestamp do evento *x* no processo *p,* e *C(x)* denota o timestamp do evento *x* em qualquer processo.

LC1: *Cp* é incrementado antes de cada evento em *p*.

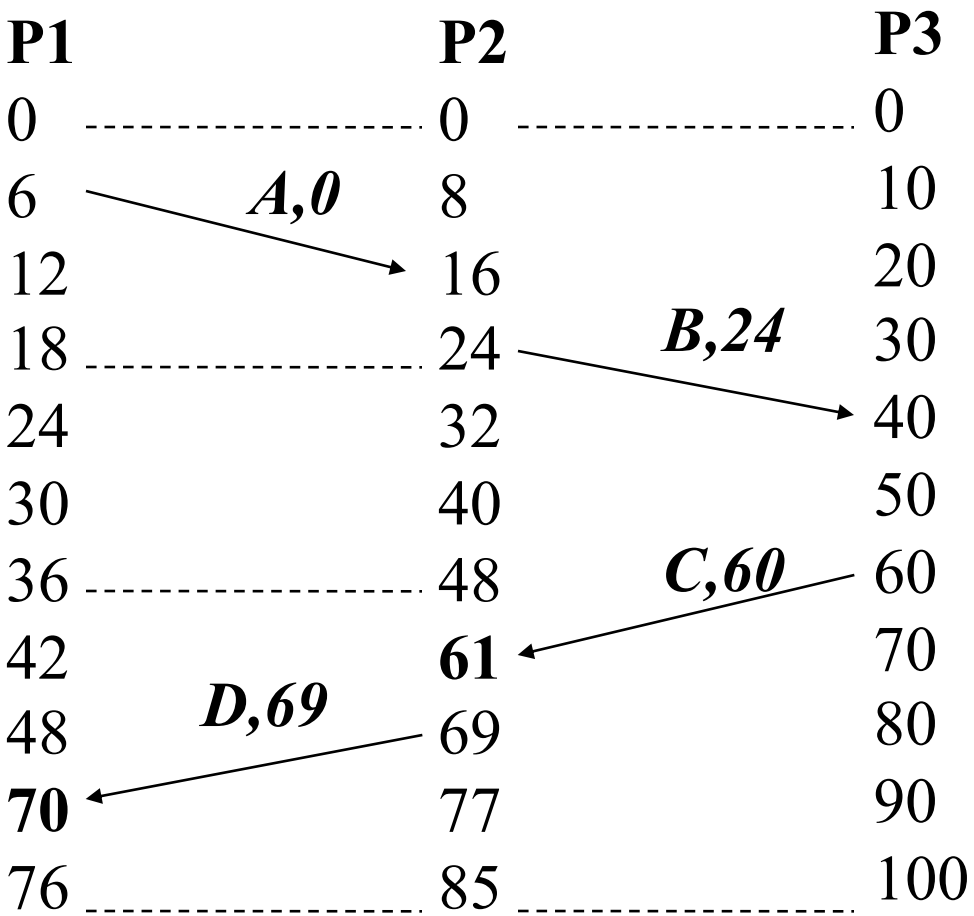
LC2: (a) Quando um processo *p* envia uma mensagem *m*, ele concatena a informação *t=Cp* a *m*, enviando *(m,t)*.

(b) Quando um processo *q* recebe a mensagem *(m,t)*, ele computa *Cq := max(Cq, t)* e aplica **LC1** antes de fazer timestamp do evento de recebimento da mensagem.

Exemplo de aplicação do algoritmo de relógios lógicos



Exemplo de aplicação do algoritmo de relógios lógicos



Relógios lógicos (cont.)

- **Ordenação total de eventos**: dois eventos nunca ocorrem exatamente no mesmo instante de tempo.
  1. Se *x* ocorre antes de *y* no mesmo processo, então *C(x)* é menor que *C(y).*
  2. Se *x* e *y* correspondem ao envio e ao recebimento de uma mensagem, então *C(x)* é menor que *C(y).*
  3. Para todos os eventos *x* e *y*, *C(x)* é diferente de *C(y).*

Implementação: concatenar o número do processo ao timestamp.

Relógios físicos

- GMT: Greenwich Mean Time
- BIH: Bureau Internacional de l'Heure
- TAI: International Atomic Time
- UTC: Universal Coordinated Time
- NIST: National Institute of Standard Time
- WWV: estação de rádio de ondas curtas
- GEOS: Geostationary Environment Operational Satellite

Relógios físicos (cont.)

- Algoritmo de Berkeley:
  - A rede não dispõe de uma máquina com um receptor WWV
  - A rede dispõe de um **time server** que faz polling nas outras máquinas a fim de obter a hora marcada por cada uma, fazer uma média entre essas horas e divulgar essa média para todas as máquinas.
- NTC: Network Time Protocol
  - Sub-rede hierárquica de sincronização
  - Servidores primários (WWV) e secundários

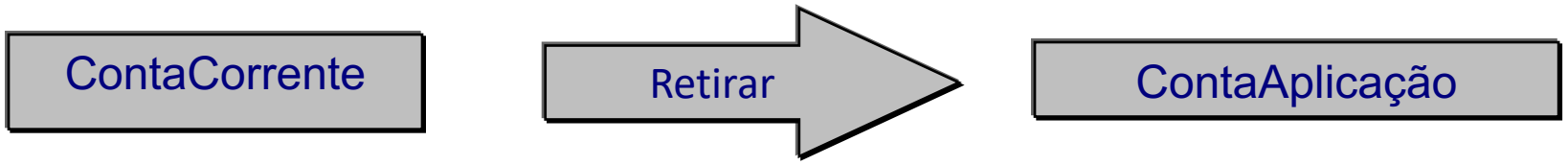
Relógios físicos (cont.)

- Algoritmo de Cristian:
  - A rede dispõe de um **time server** (receptor WWV)
  - Uma máquina cliente envia uma mensagem pedindo a hora certa ao **time server**
  - Ao receber a mensagem resposta do **time server,** o cliente adiciona o tempo médio de envio de mensagens à hora recebida. Esse tempo médio é calculado pelo próprio cliente considerando as horas de envio e recebimento das mensagens e ainda o tempo gasto pelo **time server** para processar o pedido.

Segurança de Dados em Transações Distribuídas

O que é Transação?

É uma unidade lógica de processamento que tem por objetivo preservar a integridade e a consistência dos dados de um sistema.



Este requerimento de “ou faz tudo ou não faz nada” é chamado de **atomicidade**.

Transações

Para que você realize esta unidade de processamento com atomicidade, você deve abrir a transação, realizar as operações com dados, verificar se algum problema ocorreu. Se todas as operações com dados tiverem sido realizadas com sucesso, você deve confirmar a operação. Caso algum problema tenha ocorrido, você deve garantir que nada seja feito.

Sintaxe

BEGIN TRAN [SACTION] [<Nome_Transação>) | @variável]

.

COMMIT TRAN [SACTION] [<Nome_Transação>) | @variável]

.

COMMIT [WORK]

.

ROLLBACK TRAN [SACTION] [[<Nome_Transação>) | @variável | NomeSavePoint | @variável_SavePoint]

.

ROLLBACK [WORK]

.

SAVE TRAN [SACTION] (Nome_savapoint) | @variável_savepoint)

Observe o esquema em seguida:

Criar a unidade de processamento
realizar o **DÉBITO**
checar a ocorrência de erro
se ocorreu algum erro:
1 – desfaça qualquer operação que tenha sido feita até este ponto.
2 – interrompa o processamento aqui.

Realizar **CRÉDITO**
checar a ocorrência de algum erro.
se ocorreu erro
1 – Desfaça qualquer operação que tenha sido feita até este ponto.
2 – interrompa o processamento aqui.

Se não ocorrer nenhum problema
Confirme a operação

Transação

Para realizar o processamento anterior, você precisa utilizar três comandos:

1. Begin Transaction – cria uma transação, ou seja, cria uma unidade de processamento lógico;
2. Roolback Transaction – encerra a transação e desfaz qualquer operação que tenha sido realizada com dados;
3. Commit Transaction – encerra a transação e efetiva qualquer operação que tenha sido realizada com dados.

Observe agora o esquema seguinte:

BEGIN TRANSACTION
realizar o débito
checar a ocorrência de erro
se ocorreu algum erro:
1 – **ROLLBACK TRANSACTION**
2 – **RETURN**

Realizar **CRÉDITO**
checar a ocorrência de algum erro.
se ocorreu erro
1 – **ROLLBACK TRANSACTION**
2 – **RETURN**

Se não ocorrer nenhum problema
COMMIT TRANSACTION

@@ERROR

É uma variável global (função) “alimentada” pelo próprio SQL Server após a realização de qualquer comando da linguagem Transact_SQL.

Se não ocorrer erro @@ERROR = 0

Caso ocorra erro

@@ERROR = n• erro (existente na tabela sysmessages)

Sendo assim é com a variável @@ERROR que você verifica a ocorrência de erros durante o processamento dos seus dados. Observe o seguinte esquema:

@@ERROR

```
BEGIN TRANSACTION
  Realizar o DÉBITO
      IF @@ERROR <> 0
          BEGIN
              ROLLBACK TRANSACTION
              RETURN
          END
Realizar o CRÉDITO
      IF @@ERROR <> 0
          BEGIN
              ROLLBACK TRANSACTION
              RETURN
          END
COMMIT TRANSACTION
```