Tempo, Sincronização e Coordenação em Sistemas Distribuídos

Tempo, Sincronização e Coordenação

Motivações:

- **Necessidade de coordenar e sincronizar a tomada de decisões num ambiente em que os múltiplos processos se executam em processadores diferentes, interligados por um sistema de comunicações que apresenta atrasos de transmissão não desprezáveis;**
- **Dois eventos ocorridos no sistema podem ser conhecidos, em sítios diferentes, segundo uma ordem de ocorrência diferente.**

Características dos Algoritmos Distribuídos:

- **A informação relevante está disseminada por vários sítios (máquinas);**
- **Os processos tomam decisões baseadas unicamente em informação local;**
- **Não existe um relógio global comum a todas as máquinas;**
- **Deve ser evitado um ponto de falha único (processo com estado global do sistema)**

Sumário

Tempo em Sistemas Distribuídos

- Necessidade de ter uma relação Aconteceu-Antes (*happened-before*), entre dois eventos:
  - RELÓGIOS FÍSICOS - sincronização dos relógios de cada máquina a partir de uma referência temporal global - UTC (*Universal Time Coordenation*);
  - RELÓGIOS LÓGICOS - Técnica de ordenação de eventos sem conhecimento preciso do tempo em que os mesmos ocorreram.

Comunicação por Grupos

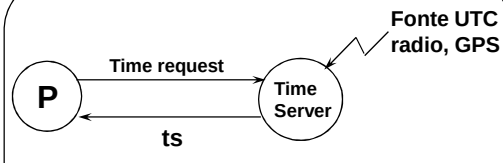
- Conceito de Grupo
- Comunicação *Multicast* fiável com ordenação de eventos

Coordenação Distribuída

- Coordenação entre processos distribuídos
  - Acesso a recursos partilhados - Algoritmos de exclusão mútua distribuída;
  - Coordenação de actividades dos vários processos - Algoritmos de eleição.

Sincronização de Relógios Físicos

Algoritmo Cristian [1989]



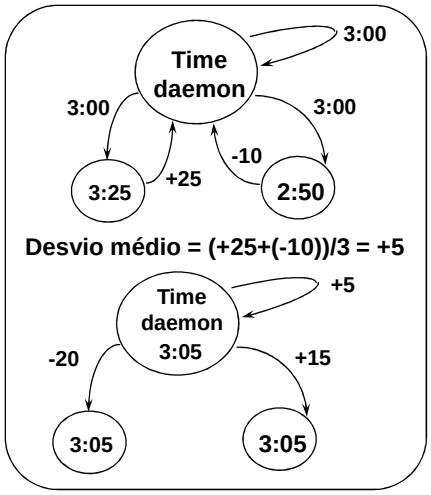
P - Acerta o relógio para: ts + $T_{round}/2$

T_{round} - tempo médio calculado por P para enviar o pedido e obter a resposta.

- Dependente de falhas do(s) servidor(es);
- Dependente da precisão do relógio local de P

Algoritmo Berkeley [Gusella,1989]

- sincronização de máquinas Unix



Sincronização de Relógios Físicos (cont.)

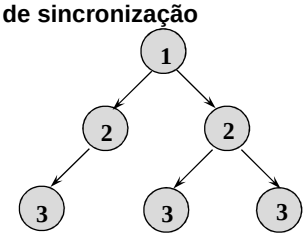
Objectivos do Network Time Protocol (NTP) - (Mills 1985-1995-…)

- Sincronização de clientes distribuídos pela Internet com base no *Universal Time Coordinated* (UTC);
- Providenciar um serviço fiável tolerante a falhas de ligação;
- Permitir aos clientes um ritmo de sincronização frequente (da ordem dos desvios (*drift*) dos relógios dos clientes);
- Protecção contra interferências ao serviço de sincronização

O NTP é baseado em:

- Servidores primários
  - ligados directamente a fontes UTC
- Servidores secundários
  - sincronizam-se a partir dos primários
- Os servidores encontram-se ligados segundo uma estrutura hierárquica

ntp.isel.ipl.pt (strata 3)
http://www.oal.ul.pt



Os níveis são designados de *strata*

- o servidor raiz ocupa o *stratum* 1
- quanto maior é o *stratum* menor é o rigor na exactidão do relógio

Artigos de referência sobre NTP

The Network Time Protocol

Cristian's method and the Berkeley algorithm are intended primarily for use within intranets. The Network Time Protocol (NTP) [Mills 1995] defines an architecture for a time service and a protocol to distribute time information over the Internet.

NTP's chief design aims and features are as follows.

To provide a service enabling clients across the Internet to be synchronized accurately to UTC: Despite the large and variable message delays encountered in Internet communication. NTP employs statistical techniques for the filtering of timing data and it discriminates between the quality of timing data from different servers.

Pag. 441 – Livro Coulouris

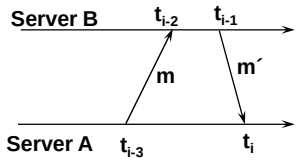
D. L. Mills, "Improved algorithms for synchronizing computer network clocks," *IEEE/ACM Trans. Network.*, vol. 3, pp. 245–254, June 1995.

D. L. Mills, "A brief history of NTP time: memoirs of an Internet timekeeper," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 9–21, April 2003

Sub-rede de sincronização (rede NTP)

Modos de sincronização do NTP

- ***Multicast*** **- Usado em LANs de alta velocidade. Periodicamente, um ou mais servers enviam mensagens com um valor de relógio para os outros servidores que acertam o seu relógio.**
- ***Procedure-call*** **- Usado em LANs vizinhas. Similar ao algoritmo de *Cristian*.**
- **Modo simétrico (*symmetric mode*) - os servidores, organizados em pares, trocam mensagens. Cada mensagem transporta os tempos da mensagem anterior e o tempo da mensagem enviada (na figura m´ transporta t_{i-3}, t_{i-2} e t_{i-1}).**



$t_{i-2}=t_{i-3} + t + o$

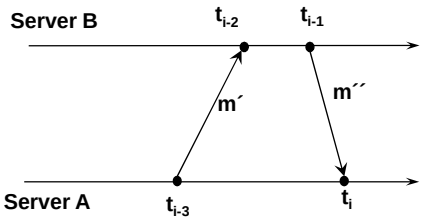
$t_i=t_{i-1} + t´ - o$

Para cada par de mensagens enviadas entre dois servidores, o NTP calcula o offset (o) que é estimado existir entre o relógio de B e o relógio de A e um delay (d_i) que traduz o tempo de transmissão (t + t´) das duas mensagens.

Demonstra-se que o *offset* é aproximadamente: $o = (t_{i-2} - t_{i-3} + t_{i-1} - t_i)/2$

Demonstração do *offset* no protocolo NTP

- **Dado que os tempos dos servidores A e B não são idênticos, existe um *offset*, que será positivo ou negativo consoante se veja a diferença do servidor A para o servidor B ou se veja do servidor B para o servidor A. No entanto, o módulo do *offset* será sempre o mesmo.**



Demonstração do *offset* no protocolo NTP (cont.)

- **Assumindo que o valor do relógio físico do servidor B está adiantado em relação ao relógio físico do servidor A é possível obter o seguinte par de equações:**

$$\begin{cases} t_{i-2} = t_{i-3} + t' + o \\ t_i = t_{i-1} + t'' - o \end{cases}$$

- **Assumindo que em média o tempo de transmissão da mensagem (t') é igual ao tempo de recepção (t'') podemos dizer:**

$$\begin{cases} t_{i-2} = t_{i-3} + T + o \\ t_i = t_{i-1} + T - o \end{cases}$$

$$\begin{cases} t_{i-2} = t_{i-3} + T + o \\ T = t_i - t_{i-1} + o \end{cases}$$

$$\begin{cases} t_{i-2} = t_{i-3} + (t_i - t_{i-1} + o) + o \end{cases}$$

$$\begin{cases} t_{i-2} - t_{i-3} - t_i + t_{i-1} = o + o \end{cases}$$

$$\begin{cases} 2o = t_{i-2} - t_{i-3} + t_{i-1} - t_i \end{cases}$$

$$\begin{cases} o = \dfrac{t_{i-2} - t_{i-3} + t_{i-1} - t_i}{2} \end{cases}$$

Relógios Lógicos

As acções de um processo podem ser modeladas através de três tipos de eventos:

- **Internos - afectam só o processo onde ocorrem e podem ser ordenados linearmente de acordo com o relógio local;**
- **Envio/Recepção de mensagens - definem fluxos de informação entre processos e estabelecem dependências causais entre eventos.**

***Lamport[1978]* - Propôs um método para ter um "tempo" não ambíguo, mostrando que a sincronização dos relógios, não necessita ser absoluta.**

→ relação de precedência Aconteceu-Antes

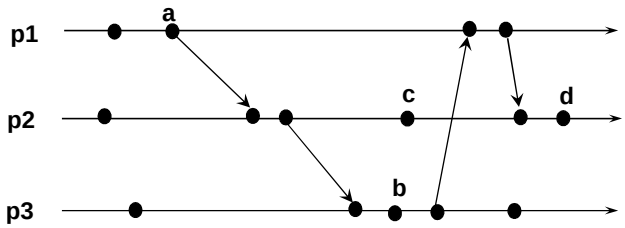
Dados dois eventos a e b

Se a → b então R(a) < R(b) : R_i - Relógio do processo i

[Precedência Local] - Se a e b são eventos do mesmo processo então a → b é verdade se a aconteceu-antes de b.

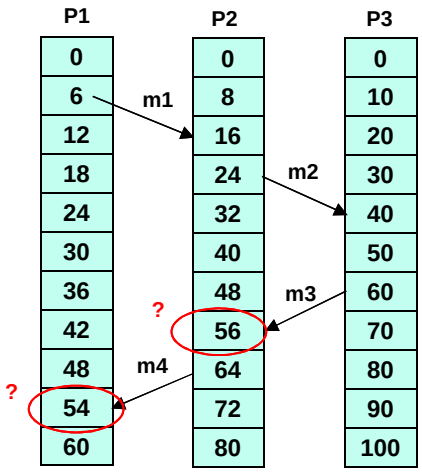
[Precedência na Transmissão] - Send(m) → Receive(m) - uma mensagem não pode ser recebida antes de ser enviada e tem um tempo finito de envio.

- A relação a → b é transitiva: a → b e b → c então a → c
- Se dois eventos a e b acontecem em processos diferentes que entretanto não trocam mensagens então (a → b) e (b → a) são falsas.
  Neste caso os eventos a e b são designados concorrentes a // b.
- Numa execução distribuída, dados dois eventos a e b temos:
  a → b ou b → a ou a // b

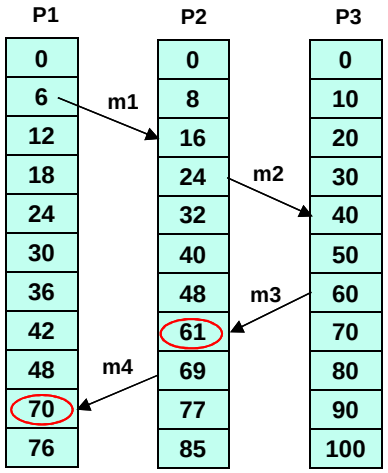


a → b
b → d
b // c

Ajustamento de Relógios Lógicos

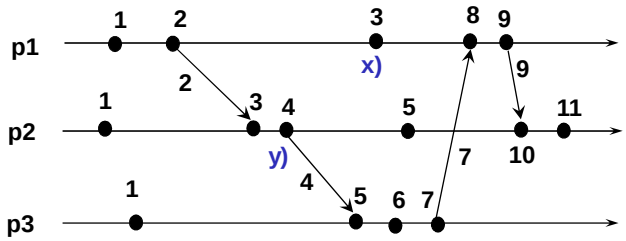


Três processos com o seu próprio relógio e com ritmos diferentes



O algoritmo de *Lamport* permite que os processos ajustem os relógios

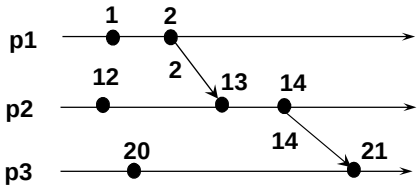
Relógios Lógicos (Algoritmo de *Lamport*)



1. Antes de cada evento um processo incrementa o valor do relógio: Ri= Ri+1;

2. Por cada mensagem enviada é associado à mensagem o *Timestamp* Ri;

3. Na recepção da mensagem:
   - Ri = max(Ri, TSmsg)
   - executa 1, isto é, Ri=Ri+1
   - a mensagem é entregue



Problemas com os relógios de *Lamport*

Os relógios de *Lamport* não resolvem todas as situações:

- Entre processos existem eventos com o mesmo relógio impossibilitando uma ordenação global de eventos;
- Eventualmente, Associando o número do processo ao valor do relógio evita-se esta situação:
  - O relógio 3 em p1 passa a 3.1 e em p2 o relógio 3 passa a 3.2.
  - Com este método consegue-se as seguintes condições:
    1. Se a aconteceu antes de b (a → b) no mesmo processo então R(a) < R(b);
    2. Se a e b representam, respectivamente, os eventos de enviar e receber uma mensagem então R(a) < R(b);
    3. Para eventos distintos a e b, R(a) ≠ R(b)

- Falta de consistência: R(x) < R(y) e no entanto y → x (ver slide anterior)
- Ao ter um relógio global que esmaga os relógios locais, perde-se a dependência causal do conjunto de eventos entre diferentes processos.

Relógios Lógicos Vectoriais

- Os relógios de *Lamport* não capturam dependências causais (a ocorrência de um evento ser consequência de outro);
- A solução é baseada em relógios vectoriais proposta independentemente por Fidge e Mattern em 1988.

- Cada processo `Pi` tem um vector de relógios `Vi` com o seguinte significado:
  - `Vi [i]` contém o número de eventos que ocorreram em `Pi`
  - Se `Vi[j] = k` então `Pi` sabe que ocorreram `k` eventos em `Pj`
  - Os relógios são actualizados com as seguintes regras:

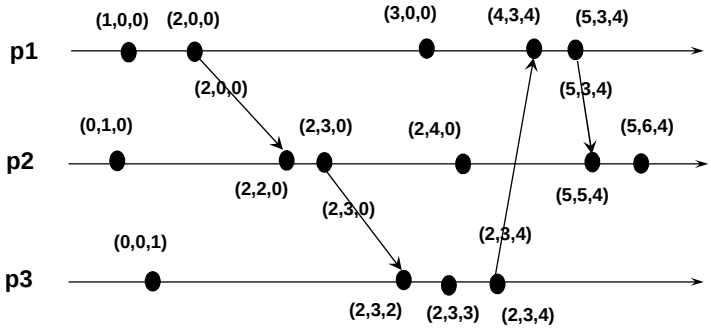
    R1: Por cada evento (interno, send, receive) em `Pi` é incrementado o relógio no vector `Vi[i]=Vi[i]+1`

    R2: Quando `Pi` envia uma mensagem `m` anexa-lhe o seu vector `m.Vi`

    R3: Quando `Pi` recebe uma mensagem actualiza o seu vector, em que para cada `x` temos `Vi[x]=max(Vi[x], m.V[x])`
  - Assim, `Vi[j]` representa o número de eventos que `Pj` produziu e que pertencem ao passado causal de `Pi`

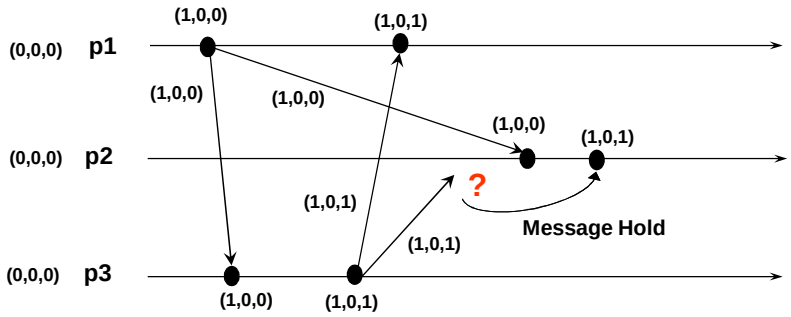
Relógios Lógicos Vectoriais

Permitem captar as dependências causais entre quaisquer eventos



multicast com ordenação causal

- Para captar dependências causais com multicast é necessário alterar o mecanismo de actualização dos relógios vectoriais.
- Uma solução simples é considerar que o Vector `Vi[i]` só é incrementado quando `Pi` envia uma mensagem, isto é não se incrementa o relógio na recepção de mensagens.



Relógios Lógicos Vectoriais com multicast

Implementação do mecanismo *multicast* com ordenação causal

vectores existentes nos outros processos L_k

	p0	p1	p2	p3	p4	p5
0	4	3	3	3	2	3
1	6	7	5	6	6	7
2	8	8	8	8	8	8
3	2	2	2	2	2	3
4	1	1	1	1	1	1
5	5	5	5	5	5	5

V_m - vector na mensagem enviada pelo p0.

OK Hold OK Hold OK

Condições que garantem ordenação causal

```
j - processo emissor
Vm[j] = Vk[j]+1
Vm[i] <= Lk[i]  para todo i <> j
```

Comunicação por Grupos - MULTICAST

- **O Multicast pode ser vantajoso em:**
  - Tolerância a falhas baseada na replicação de serviços
  - Localização de objectos em serviços distribuídos
  - Melhor desempenho através da réplica de dados
  - Actualizações em múltiplos repositórios

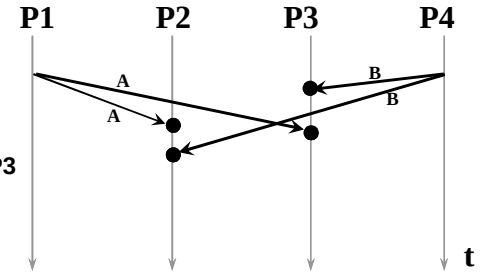
- **Atomicidade**
  - **Multicast-fiável** - método que tenta fazer o "melhor esforço" para entregar as mensagens a todos os membros, não garantindo que isso aconteça.
  - **Multicast-atómico** - existe quando as mensagens são recebidas por todos os membros do grupo ou então nenhum recebe a mensagem.

Ordem das mensagens

Mensagens:
- A - de P1
- B - de P4

↓

Multicast para P2 e P3

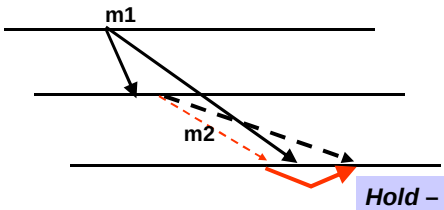


- **Multicast com ordenação total** – Garantir que todas as mensagens chegam a todos os membros pela mesma ordem.

- **Multicast com ordenação causal** – todas as mensagens com relação causal, chegam a todos os membros pela mesma ordem.

Aspectos da implementação na comunicação por grupos

- **Eficiência**
  - **utilização da capacidade do Broadcast da ethernet**
- **Fiabilidade**
  - **uma das mensagens pode perder-se**
  - **o processo emissor pode falhar após o envio de algumas mensagens**
- **Multicast atómico e fiável**
- ***Hold-back***
  - **o módulo de comunicação responsável pela atomicidade e ordenação de mensagens, retarda o envio para a aplicação de mensagens fora de ordem.**



***Hold* – assegura a ordem (m1, m2)**

Grupos de Processos

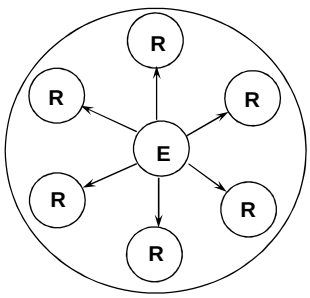
Grupo - Colecção de processos que se relacionam entre si.

- **Tolerância a falhas (serviços replicados);**
- **Localização de objectos ou serviços num sistema distribuído;**
- **Serviços de notificação e difusão de informação;**
- **Processamento paralelo;**
- **Modularidade das aplicações distribuídas (Dividir para Reinar);**

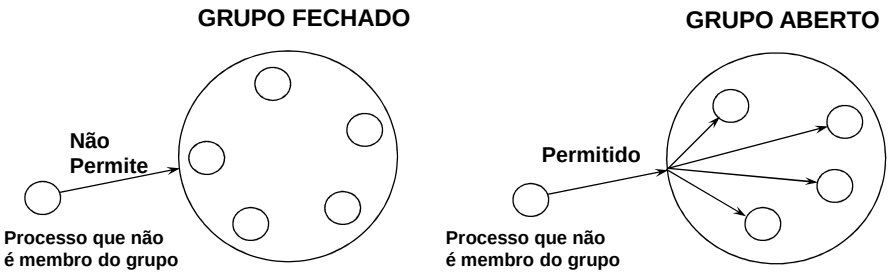
Comunicação por Grupos

Quando uma mensagem é enviada para o grupo todos os membros do grupo a recebem (*multicasting*).

A abstracção de grupo de processos, permite que um processo cliente envie uma mensagem para um grupo de servidores, sem saber quantos são e onde estão localizados



Tipos de Grupo



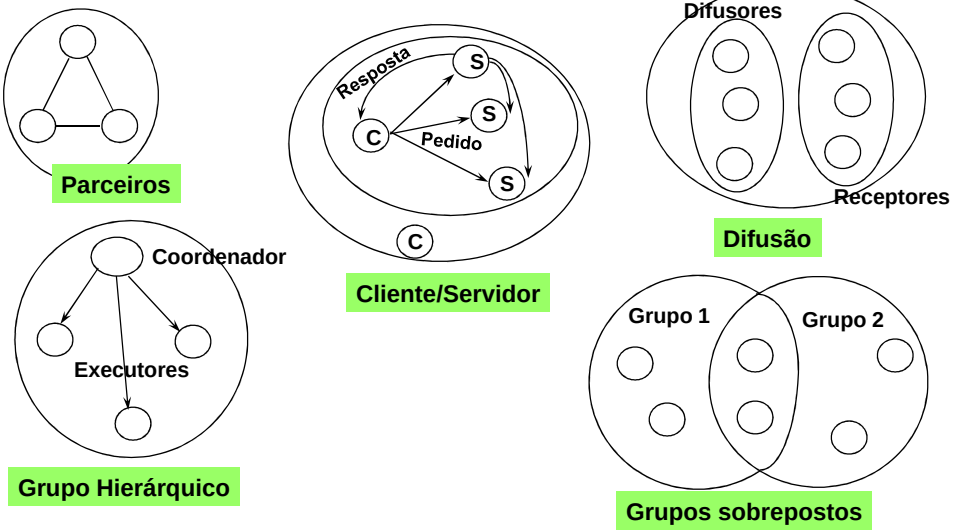
A distinção entre grupos fechados e abertos é muitas vezes feita por questões de implementação. No entanto, há situações que caracterizam os dois tipos de grupo:

Grupos Fechados - Usados em processamento paralelo, por exemplo: colecção de processos de um jogo de xadrez, aplicações de cálculo científico, etc

Grupos Abertos - Grupo de servidores replicados, em que um processo cliente (fora do grupo), pode enviar pedidos ao grupo.

Estrutura dos grupos

Definida de acordo com o padrão de comunicação interno



Características desejáveis da comunicação por Grupos

- Cada grupo deve ter um identificador (endereço de grupo) único;
- Reconfiguração dinâmica. Um processo pode entrar/sair (join/leave) de um grupo;
- Fiabilidade no envio de mensagens
  (sem preocupações de mensagens perdidas ou duplicadas)
- Critérios consistentes de ordenação das mensagens;
- Possibilidade de manter um estado global consistente;
- Transferência de estado para os novos membros que chegam ao grupo;
- Possibilidade de manter o historial de eventos vistos pelo grupo;
- Atomicidade ou *Atomic Multicast*: Propriedade do tudo-ou-nada. Quando uma mensagem é enviada ao grupo, ou todos a recebem ou ninguém a recebe

Critérios de ordenação das mensagens

Ordenação Total no tempo: Se um processo envia uma mensagem A e mais tarde outro processo envia uma mensagem B, então todos os membros do grupo recebem primeiro A e depois B; (Difícil de implementar e pouco eficiente).

Ordenação Total Consistente: O sistema impõe uma ordem total consistente independente do tempo. Duas mensagens concorrentes (juntas no tempo) A e B, são ordenadas pelo sistema, garantindo que ou todos os processos recebem primeiro A e depois B ou todos recebem primeiro B e depois A.

Ordenação Causal: As mensagens relacionadas por fluxos de informação são recebidas pela mesma ordem. Se existirem duas mensagens A e B, em que B é consequência de A, então o sistema garante que todos os membros do grupo recebem primeiro A e depois B.

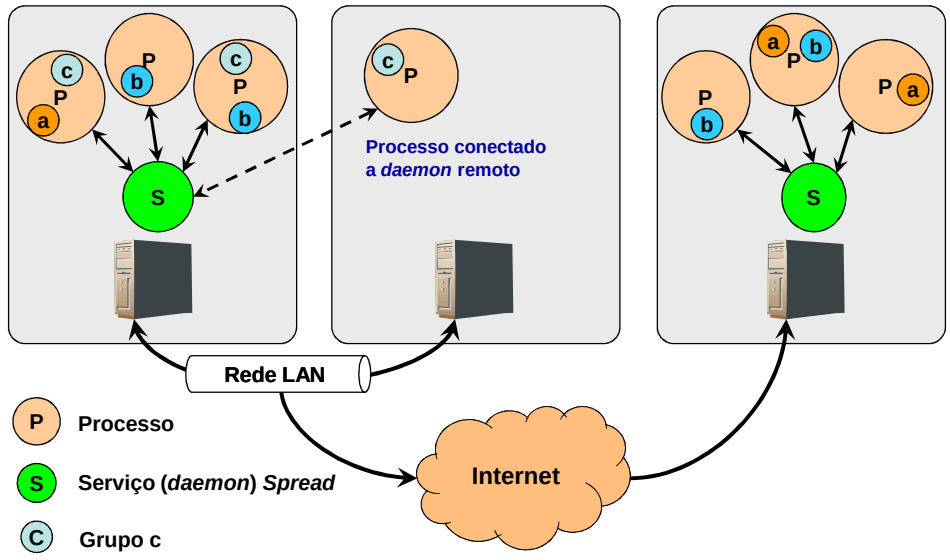
The Spread Toolkit –

http://www.spread.org/

- O *Spread* funciona como um *Bus* de mensagens para aplicações distribuídas de alto desempenho, proporcionando comunicação por grupos (*multicast*) fiável (garantia de entrega das mensagens) e com ordenação total das mensagens, mesmo na presença de falhas (tolerante a falhas);

- O *Spread Toolkit* consiste num serviço (*daemon*), uma biblioteca (API) que as aplicações utilizam para usarem o *Spread*, vários utilitários e aplicações de demonstração.

Arquitectura



Características fundamentais

1. **Abstracção de Grupo (nome representando um conjunto de processos que receberão qualquer mensagem enviada para o Grupo);**
2. ***Multicast* de mensagens para um Grupo;**
3. **Conceito de *Membership* de um Grupo,**
4. **Mensagens fiáveis para um Grupo;**
5. **Ordenação de mensagens para um Grupo;**
6. **Detecção de Falhas de membros de um Grupo;**
7. **Modelo com semântica forte de como as mensagens são tratadas quando ocorre uma mudança de *Membership* num Grupo;**

Mensagens – Tipo de mensagens

Tipo de Mensagem	Ordenação	Grau de Confiança na entrega
UNRELIABLE_MESS	Sem ordenação	Sem confiança
RELIABLE_MESS	Sem ordenação	Com confiança
FIFO_MESS	Ordem FIFO na emissão	Com confiança
CAUSAL_MESS	Causal (Relógios de *Lamport*)	Com confiança
AGREED_MESS	Ordem Total (consistente com ordem causal)	Com confiança
SAFE_MESS	Ordem Total	Confiança Total

Mensagens – Ordenação de mensagens

- **Sem ordenação** (UNRELIABLE_MESS; RELIABLE_MESS)

  Este tipo de ordenação não garante qualquer ordem de entrega das mensagens. Qualquer mensagem m1 que seja enviada com este tipo de ordenação pode ser entregue antes ou depois de uma outra mensagem m2.

- **Ordem FIFO na emissão** (FIFO_MESS)

  Todas as mensagens deste tipo que forem enviadas por um dado emissor têm a garantia de serem entregues pela ordem de envio. Neste contexto define-se emissor como uma ligação ao *daemon spread*., isto é, se um processo tiver 3 conexões a diferentes *daemon*, então temos três emissores distintos.

  No caso de se enviar uma mensagem do tipo RELIABLE_MESS depois de uma mensagem do tipo FIFO, é possível que a mensagem RELIABLE_MESS seja entregue antes.

Mensagens – Ordenação de mensagens (cont.)

- **Ordem Causal** (CAUSAL_MESS)

  Todas as mensagens, enviadas por todos os emissores, serão entregues por uma ordem consistente com a definição de ordem causal, proposta por *Lamport*. Este tipo de ordenação é consistente com a Ordem FIFO.

- **Ordem Total** (AGREED_MESS; SAFE_MESS)

  Todas as mensagens, enviadas por todos os emissores, serão entregues pela mesma ordem para todos os destinatários. Este tipo de ordenação é consistente com o tipo Ordem Causal.

Mensagens – Grau de confiança das mensagens

- **Sem confiança**

  As mensagens enviadas podem ou não ser entregues. A plataforma não faz nenhuma tentativa de recuperação.

- **Com confiança**

  As mensagens enviadas serão sempre entregues a todos os membros do grupo. A plataforma efectua recuperação de mensagens para ultrapassar perda de mensagens devido a problemas de rede.

- **Confiança Total**

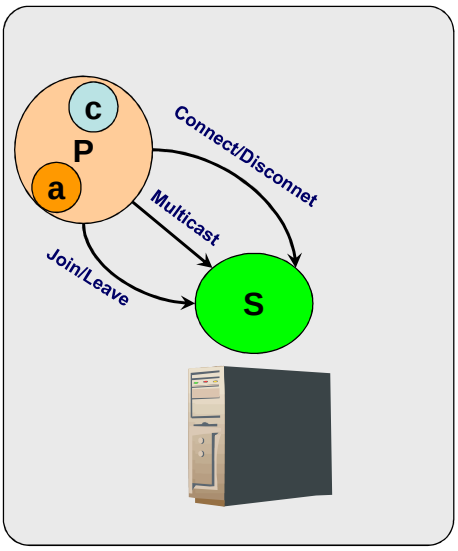
  Mensagens enviadas só são entregues se o *daemon spread* associado ao receptor da mensagem tiver a garantia que todos os *daemons* receberam a mensagem.

API – 3 Conceitos – 3 Classes

– **Ligação** (Classe ***SpreadConnection***)
- Permite ligar o processo a um *daemon*
  - ***connect()***
  - ***disconnet()***
- Enviar mensagens para grupo
  - ***multicast()***
- Receber Mensagem
  - ***receive()***

– **Grupo** (Classe ***SpreadGroup***)
- Representa o conceito de grupo
  - ***join()***
  - ***leave()***

– **Mensagem** (Classe ***SpreadMessage***)
- Criar mensagem
  - ***spreadMessage()*** –
- Adicionar à mensagem os grupos destino
  - ***addGroup()***



4.3.1 SpreadConnection

```
import spread;
SpreadConnection SpreadConnection();
connect(InetAddress spread_name,        int port,        String privateName,
                boolean priority, boolean groupMembership);
disconnect();
SpreadGroup getPrivateGroup();
multicast(SpreadMessage message);
multicast(SpreadMessage messages[]);
SpreadMessage receive();
SpreadMessage[] receive(int numMessages);
boolean poll();
add(BasicMessageListener listener);
add(AdvancedMessageListener listener);
remove(BasicMessageListener listener);
remove(AdvancedMessageListener listener);
```


4.3.3 SpreadGroup Class

```
SpreadGroup SpreadGroup();
join(SpreadConnection connection, String groupname);
leave();
String toString();
boolean equals(Object object);
```

4.3.2 SpreadMessage Class

```
import spread;
SpreadMessage SpreadMessage();
boolean isIncoming()
boolean isOutgoing()
int getServiceType();
boolean isRegular();
boolean isMembership();
boolean isUnreliable();
boolean isReliable();
isFifo();
isCausal();
isAgreed();
isSafe();
isSelfDiscard();
SpreadGroup[] getGroups();
SpreadGroup getSender();
byte[] getData();
Object getObject();
Vector getDigest();
short getType();
boolean getEndianMismatch();
setServiceType(int serviceType);
setUnreliable();
setReliable()
setFifo();
setCausal();
setAgreed();
setSafe();
setSelfDiscard(boolean selfDiscard);
addGroup(SpreadGroup group);
addGroup(String group);
addGroups(SpreadGroup groups[]);
addGroups(String groups[]);
setData(byte[] data);
setObject(Serializable object);
digest(Serializable object);
setType(short type);
MembershipInfo getMembershipInfo();
Object clone();
```


4.3.4 MembershipInfo Class

```
boolean isRegularMembership();
boolean isTransition();
boolean isCausedByJoin();
boolean isCausedByLeave();
boolean isCausedByDisconnect();
boolean isCausedByNetwork();
boolean isSelfLeave();
SpreadGroup getGroup();
GroupID getGroupID();
SpreadGroup[] getMembers();
SpreadGroup getJoined();
SpreadGroup getLeft();
SpreadGroup getDisconnected();
SpreadGroup[] getStayed();
```

A partir do conceito de *Membership*, cada membro do grupo é notificado sobre todos os acontecimentos no grupo, por exemplo, se um membro falhar todos os restantes são notificados dessa situação:

Linguagens

- **Oficialmente o *toolkit* oferece suporte para C/C++ e para a linguagem Java;**
- **No entanto, existem portes feitos por várias pessoas para outras linguagens:**

Title	Description	Notes	Location
OcamlSpread	Objective Caml bindings	Early release, looking for maintainer.	http://ocamlspread.sourceforge.net
Ruby Spread	Ruby bindings		www.omniti.com/~george/rb_spread/
Squeak Spread	Squeak Smalltalk bindings		http://bike-nomad.com/squeak/
PHP for Spread	PHP bindings		http://www.omniti.com/~george/php_spread_ or http://pecl.php.net/package/spread
Python Spread	Python bindings	Maintained by Python/Zope team.	http://www.python.org/other/spread/
C# Spread	C# bindings and sample programs	Developed by Darin Peshev dpeshev@axarosenberg.com	cspread.zip
Lua Spread	Lua bindings	Developed by Taj Khattra	http://pobox.com/~taj.khattra/luaspread.html
Scheme Spread	Scheme bindings	Developed by Felix Winkelmann	http://www.call-with-current-continuation.org/eggs/spread.html
TCL Spread	TCL Bindings	Early release, looking for maintainer.	http://ocamlspread.sourceforge.net

Configuração

- **A configuração do *spread* é baseada num ficheiro de texto que permite definir:**
  - **Configuração da rede - São indicados quantos *daemons spread* existem bem como os troços de rede onde estão localizados na rede**
  - **Opções de execução - São indicados vários parâmetros tais como nível de *log* dos *daemons*, local onde os *logs* são guardados, etc.**

spread.config

```
Spread_Segment 192.168.1.255:4803 {
        portlass        192.168.1.100
        sidney          192.168.1.101
}
# PRINT:info that should always be printed;
# EXIT:Errors or other events that cause Spread to quit.
DebugFlags = { PRINT EXIT }
#EventLogFile = testlog.out
#EventTimeStamp = "[%a %d %b %Y %H:%M:%S]"
#Enable ou Disable comandos da aplicação sptmonitor.exe
DangerousMonitor = false
# Liberta o socket TCP/IP se o servidor faz crash
SocketPortReuse = AUTO
```

Sincronização e Coordenação

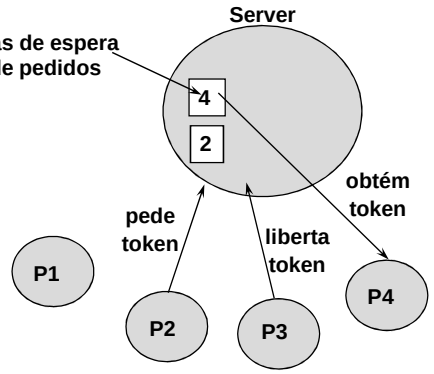
- Algoritmos de Exclusão Mútua Distribuída
- Algoritmos de Eleição

Exclusão mútua

Características da exclusão mútua:

- Consistência - apenas um processo na região critica;
- Um processo obtém sempre acesso
- As entradas devem acontecer pela ordem Aconteceu-Antes

Algoritmo Centralizado



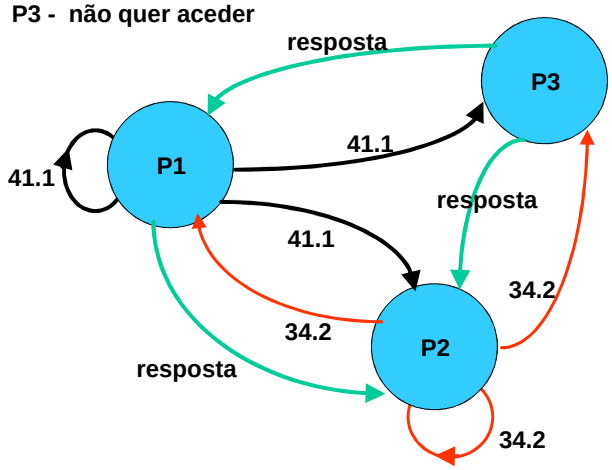
Algoritmo centralizado:

Entrar( ) // bloqueia se necessário acesso à região crítica

Sair( ) // outro processo pode entrar

Algoritmo Distribuído [*Ricart, Agrawala*]

P1 e P2 – pretendem aceder
P3 - não quer aceder



P2 – Obtém autorização por ter o menor *TimeStamp*

- Cada processo mantém um relógio lógico actualizado segundo as regras 1,2 e 3 definidas por Lamport.
- Cada *TimeStamp* tem a forma (T, p_i), onde T é o valor do relógio do emissor e pi um identificador único do processo
- O processo que apresentar a *TimeStamp* menor ganha o acesso à região critica
- Pressupõe existência de *multicast*

Algoritmo Distribuído (cont.)

Estados do Token = {PRETENDIDO, LIBERTADO, POSSUIDO}

Inicialização:

estado = LIBERTADO;

Obtenção do Token:

estado = PRETENDIDO;
Multicast com pedidos para todos os processos;
T = Time-stamp do pedido;
Espera até (número de respostas == (N – 1)); // N igual ao número de processos
estado = POSSUIDO;

Algoritmo Distribuído (cont.)

O processo Pj recebe o pedido (Ti, Pi) (i <> j)

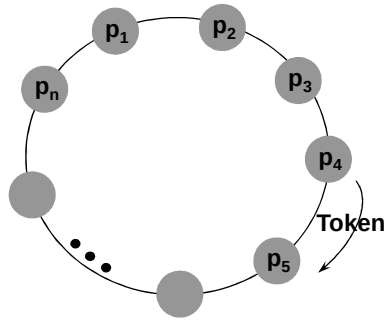
Se (estado == POSSUIDO ou (estado == PRETENDIDO e (Tj, Pj) < (Ti, Pi)))
então
coloca numa fila de espera o pedido de Pi sem resposta
senão
responde OK imediatamente a Pi;

Libertar o Token:

estado = LIBERTADO;
Responde OK para todos os pedidos pendentes na fila de espera;

Algoritmo em Anel

- **Em cada momento só um processo possui o *Token*;**
- **Um processo que não queira entrar na região critica envia o *token* para o vizinho;**
- **Um processo que queira o *token*, espera até o receber. Quando o recebe acede à região critica e após sair envia o *token* ao vizinho seguinte**



- **Não obedece à condição Aconteceu-Antes;**
- **Necessidade de (n-1) mensagens para obter o token;**
- **Dificuldade de repor o anel após a falha de um processo;**
- **Se o processo que tem o *token* falha, é necessário eleger o processo que o volta a ter;**

Comparação dos algoritmos de exclusão mútua

Algoritmo	Mensagens entrar/sair	Mensagens antes de entrar	Problemas
Centralizado	3 Ped, Ack, Sair	2 Ped, Ack	Falha do Servidor Falha do processo com *token*
Distribuído	2(n-1)	2(n-1)	Falha de um nó qualquer
Anel	1 – todos ∞ - ninguém	0 – *token* local (n-1) – acabou de sair	Falha do processo com *token*

Algoritmos de Eleição

Eleger um processo que actue como:

- **Iniciador;**
- **Sequenciador;**
- **Coordenador;**
- **Outro papel especial coordenar a terminação**

- **Todos os membros do grupo conhecem o número (identificador, endereço) dos outros;**
- **Assegurar que quando uma eleição se inicia todos os processos envolvidos vão concordar qual passa a ser o novo coordenador;**
- **Assume-se que a comunicação é fiável, mas que os processos podem falhar**

Algoritmo *Bully* [*Garcia e Molina, 1982*]

O coordenador (eleito) é o processo com o número maior

Quando um processo descobre que o coordenador actual não responde a um pedido (ou falhou) inicia uma eleição:

1. **Envia uma mensagem ELECTION para todos os processos que têm número maior;**
2. **Se ninguém responde o processo assume-se como coordenador (ganha eleição);**
3. **Se um processo com número mais alto responde, o processo que iniciou a eleição fica à espera que lhe seja anunciado quem é o vencedor (novo coordenador);**
4. **O processo que ganha anuncia a sua eleição a todos os outros;**

<u>Tipos de mensagens</u>

- **mensagem de eleição**
- **mensagem de resposta à eleição**
- **mensagem do coordenador confirmando a sua eleição**

Algoritmo *Bully* (cont.)

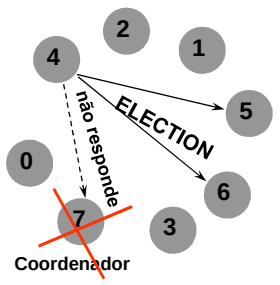


fig. a)

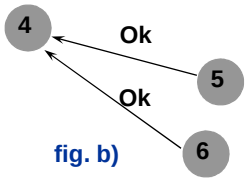


fig. b)

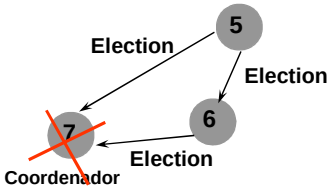


fig. c)

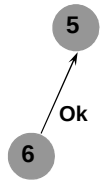


fig. d)

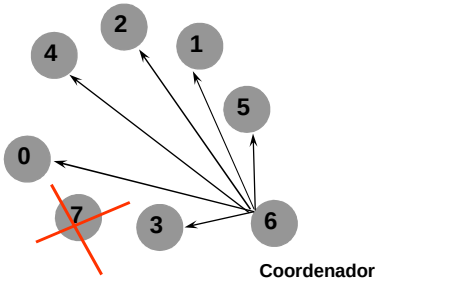
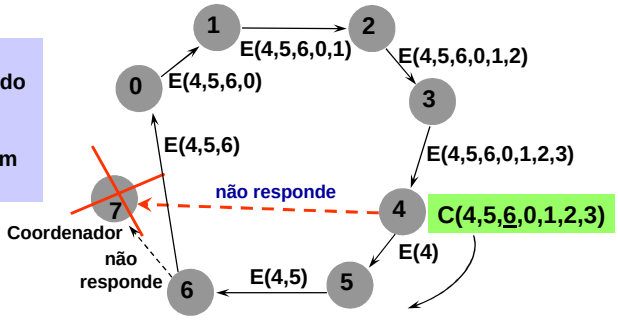


fig. e)

Algoritmo em Anel [*Tanenbaum 1992*]

- Os processos comunicam em anel;
- Quando um processo descobre que não há coordenador, envia para o seu sucessor no anel uma mensagem de ELEIÇÃO contendo o seu número;
- Em cada passo os processos adicionam o seu número, declarando-se participantes;
- Quando a mensagem que chega a um processo já inclui o número deste, a mensagem é alterada para COORDENADOR, continuando a circular novamente para que todos os participantes conheçam o novo coordenador;

O processo 4 descobre que o coordenador (7) falhou iniciando uma Eleição.
O processo 4 é também quem em primeiro lugar conclui quem é o novo coordenador (6)



Exercício

Considere um sistema que usa comunicação por grupos, por exemplo, baseado no *toolkit Spread*. Proponha um algoritmo de eleição que eleja um membro do grupo como coordenador sempre que o grupo sofre alterações:

- ✓ Entrada de membros (*Join*)
- ✓ Saída de membros, incluindo o coordenador corrente (*Leave*)
- ✓ Falha de um membro (sem ter feito *Leave*)

Leituras complementares para este Tópico

[Livro] **G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems, Concepts and Design, Fourth Edition, ISBN 0321263545*, Addison-Wesley, 2005; Capítulos 11 e 12**

[Artigo] **"Time, Clocks, and the Ordering of Events in a Distributed System",** Leslie Lamport, Communications of the ACM, July 1978, Volume 21, Number 7

[Artigo] **"Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems", Roberto Baldoni • Universita di Roma, Italy, Michel Raynal • IRISA, France**

[Artigo] **"Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications",** Tobias Landes, Institut f¨ur Informati, Technische Universit¨at M¨unchen, Germany

[site do Spread ToolKit] - **http://www.spread.org/**
"A Users Guide to Spread", Jonathan R. Stanton, October 21, 2002