Sincronização em Sistemas Distribuídos

Slides baseados no material do prof. FELIPE CUNHA

Sincronização de Processos

Processos Cooperativos:

- Afetam ou são afetados por outros processos.
- Podem compartilhar um mesmo espaço de endereços lógicos, ou um mesmo arquivo.
  - Endereços lógicos: threads, ou processos leves.

Acesso concorrente a dados pode gerar inconsistência de dados.

- Devem ser desenvolvidos mecanismos para garantir a consistência de dados compartilhados por processos cooperativos.

Introdução

O caso produtor/consumidor:

- Suponha processos com memória compartilhada
- Memória consiste em um buffer de tamanho variável que abriga itens em estoque. Variável counter indica o número de itens em estoque (itens no buffer).
- Toda vez que um produto é adicionado incrementa-se o counter.
- Toda vez que um produto é retirado decrementa-se o counter.

O caso produtor/consumidor

Código do produtor:

```
do
    ...
    produce an item in nextp
    ...
    while (counter == n)
      no-op;
    buffer[in] = nextp;
    in = in++ % n;
    counter++;
while (false)
```

O caso produtor/consumidor

Código do consumidor:

```
do
    while (counter == 0)
      no-op;
    nextp = buffer[out];
    out = out++ % n;
    counter--;
    ...
    consume the item in nextp
    ...
while (false)
```

O caso produtor/consumidor

Algoritmos corretos isoladamente.

Podem causar inconsistência se rodados em paralelo.

Problema: **Seção Crítica!!!**

- Região de código onde um processo está acessando dados compartilhados.
- Problemas de inconsistência ocorrem em sessões críticas e devem ser evitados.

O problema da seção crítica

Soluções para o problema da seção crítica devem satisfazer os requisitos:

- Exclusão mútua: Se um processo está executando uma seção crítica (acessando dados compartilhados) os demais processos devem esperar.
- Progressão: Se ninguém está na seção crítica, então apenas processos que querem entrar na seção crítica podem participar da escolha de quem vai entrar.
- Espera limitada: existe um limite no número de vezes que outros processos são permitidos a entrar na seção crítica após um processo requisitar a seção crítica.

Soluções de implementação

Algoritmo da vez:

- Vetor de flags e variável de vez
- Ao terminar de usar a seção, o processo "da vez" passa a permissão para o próximo
- Pode gerar ociosidade quando o próximo não deseja usar a seção.

Algoritmo do padeiro (Bakery algorithm):

- Distribuição de senha
- Exige leitura e escrita atômicas de variáveis compartilhadas em memória global.

Semáforos:

- *wait(S)* => **while** (S <= 0) no-op;
  S--;
- *Signal(S)* => S++;

O caso produtor/consumidor com semáforos

Vamos utilizar dois semáforos contadores (Full e Empty) e um binário (mutex)

- **Full** representa a quantidade de itens no buffer
- **Empty** representa a quantidade de espaços desocupados no buffer.

Inicialização dos semáforos:

- mutex = 1 // permite ou não acesso ao buffer
- Full = 0 // começamos com todos os espaços desocupados, logo a quantidade de itens é zero
- Empty = n // começamos com todos os espaços vazios

O caso produtor/consumidor com semáforos

Código do produtor:

```
...
produce an item in nextp
...
wait(empty);
wait(mutex);
...
add nextp to buffer;
...
signal(mutex);
signal(full);
```

O caso produtor/consumidor com semáforos

Código do consumidor:

```
wait(full);
wait(mutex);
...
remove nextc from buffer;
...
signal(mutex);
signal(empty);
...
consume the item in nextc
...
```

Sincronização em Sistemas Distribuídos

Conceitos importantes no projeto de qualquer aplicação distribuída:

- Comunicação
- Sincronização

Sincronização em aplicações distribuídas:

- Como garantir exclusão mútua no acesso a seções críticas ?
- Como garantir atomicidade na execução de uma transação distribuída ?
- Como alocar recursos evitando a ocorrência de deadlocks ?

Sincronização em Sistemas Distribuídos

Sincronização em sistemas centralizados: utiliza memória compartilhada

- Exemplos: semáforos, monitores, etc

Sincronização em sistemas distribuídos: utiliza troca de mensagens

- Implementada via algoritmos distribuídos
- Mesmo determinar se um evento A ocorreu antes ou depois de um evento B é mais complexo do que em um sistema centralizado

Sincronização Física de Relógios

Sincronização de Relógios

Relógios são essenciais no uso de computação, seja para medir o tempo ou identificar sequências de eventos diversos.

- Em sistemas centralizados, o tempo não é ambíguo, sendo gerenciado em apenas 1 máquina e obtido por chamada ao núcleo;
- Em sistemas distribuídos, obter um horário comum a vários computadores não é trivial.

Sincronização Física de Relógios

Até invenção dos relógios mecânicos (sec. XVII):

- Tempo medido com auxílio dos astros
- Sol nasce no horizonte a leste
- Alcança uma altura máxima no céu, ao meio dia (trânsito solar)
- Sol se põe no horizonte a oeste

Intervalo entre 2 trânsitos solares consecutivos do Sol é um dia solar

Portanto, se um dia possui 24 horas:

- Cada hora possui 3600 segundos
- Um segundo solar é exatamente 1/86.400 de um dia solar

Sincronização Física de Relógios

Em meados de 1940:

- Estabelecido que a rotação da Terra não é constante:
  - Devido à desaceleração gradativa resultante de marés e atrito com atmosfera
- Há 300 milhões de anos o ano tinha 400 dias!
  - A Terra gira mais devagar, mas não alterou sua órbita. Logo, o tamanho do ano aparenta ser o mesmo, mas os dias ficaram mais longos
- Além disso há pequenas alterações ao longo do dia, devido a turbulências no núcleo da Terra
- Necessário definir nova medida para o tempo:
  - Calculado o tempo de vários dias, obtendo uma média do dia
  - Ao dividi-la por 86.400, obtém-se o segundo solar médio

Sincronização Física de Relógios

Em 1948 foi inventado o relógio atômico:

- Calcula o tempo através de contagens de transições do césio 133 (1 segundo solar médio = 9.192.631.770 transições)

Tal fato tornou possível:

- Medir o tempo com maior precisão
- Medi-lo independentemente das condições do globo terrestre e da atmosfera
- Cálculo da hora atômica internacional (TAI)

Sincronização Física de Relógios

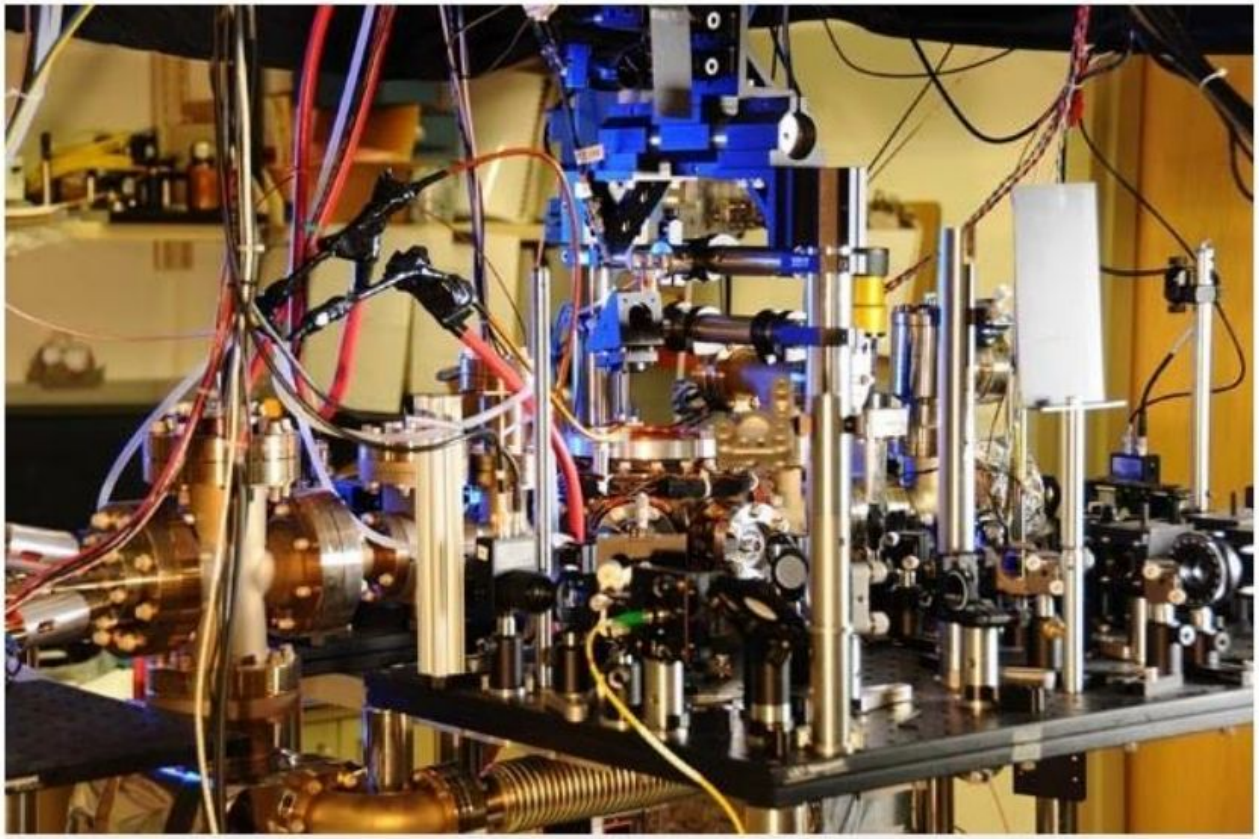


Foto mostra o relógio atômico ultra estável fabricado por cientistas americanos. O relógio de itérbio é mais preciso do que qualquer outro relógio atômico. (Foto: AFP Photo/NIST)

Sincronização Física de Relógios

Com base no TAI com correções de segundos foi estabelecido o sistema UTC (Universal Coordinated Time), que é a base de toda a moderna medição de tempo

- O Nist fornece UTC através de rádios de ondas curtas (WWV).
- Além disso vários satélites fornecem UTC.
- O GPS faz triangulação usando satélites de modo a calcular as diferenças de tempo entre o UTC de cada satélite usado, calculando, assim, a localização geográfica do ponto.

Sincronização Física de Relógios

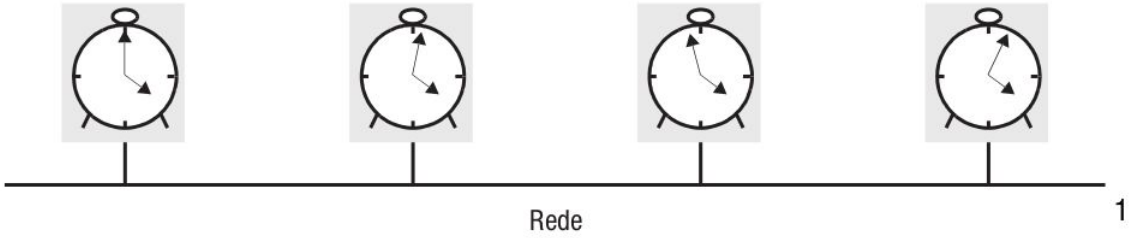
Relógios dos computadores atuais: podem atrasar ou adiantar com o tempo (clock drift)

- Razão: cristal de quartzo oscila a freqüências ligeiramente diferentes
- Clock drift médio: 1 seg a cada 11.6 dias

Sincronização Física de Relógios:

- Objetivo: manter os relógios físicos de um conjunto de máquinas "acertados"
  - $| Ci - Cj | < \varepsilon$, para todo i e j/

Aplicações: sistemas de tempo real ou de missão crítica



Algoritmo de Cristian

Protocolo de tempo de Rede - NTP

Proposto em 1989, baseia-se em clientes consultarem um servidor de tempo.

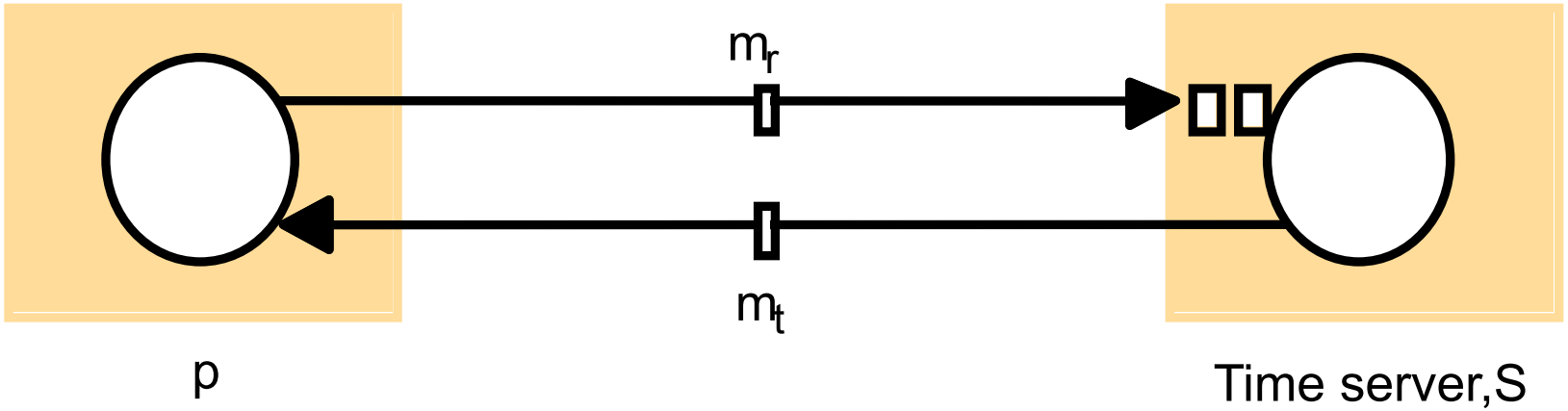
Existem dois tipos de estações:

- Um servidor de tempo (time server)
- Clientes deste servidor de tempo

Idéia básica:

- Clientes periodicamente requisitam o tempo ao servidor
- Tempo do servidor: CUTC (current UTC)

Sincronização usando um servidor de tempo



Algoritmo de Cristian

Problema 1: $C_{UTC} < C_{cliente}$ (ou seja, cliente está adiantado)

- Pode trazer conseqüências inesperadas para algumas aplicações
- Violação da monotonicidade do tempo (tempo sempre crescente)
- Eventos futuros parecerem ocorrer antes de eventos passados (pode causar confusão em logs, ordenação de mensagens e auditorias)
- Erros em sistemas baseados em timestamps

Solução: “atrasar” o relógio gradativamente, até que a correção seja implementada

Algoritmo de Cristian

Problema 2: Como calcular o tempo de propagação da msg do servidor até o cliente ?

- T1: tempo em que o *request* foi enviado
- T4: tempo em que o *reply* foi recebido
- I: tempo médio de processamento no servidor
- TP (tempo de propagação) ≈ (T4 - T1 - I) / 2

- Novo tempo do cliente= C_{UTC} + TP

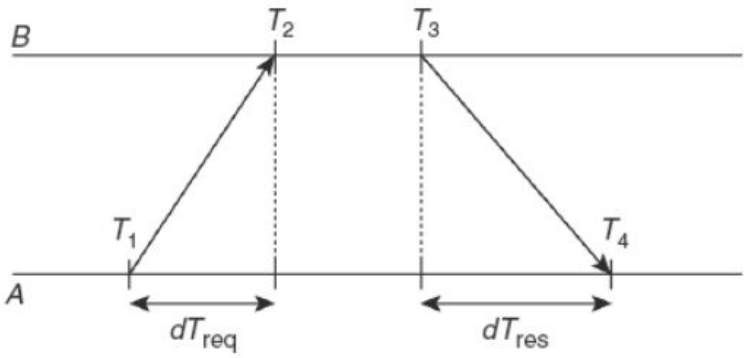


Figura 6.6 Obtenção da hora corrente por meio de um servidor de tempo.

Algoritmo de Cristian

Implementação Prática:

- NTP (Network Time Protocol, 1991)
- Padrão Internet para sincronização de relógios (RFC 1129)
- Baseado em UDP
- Hierarquia de servidores (primários, secundários etc)

NTP – Network Time Protocol

Padrão Internet para sincronização de relógios (RFC 1129).

Desde 1979: O sistema de execução ininterrupta mais longa da internet.

Baseado em UDP.

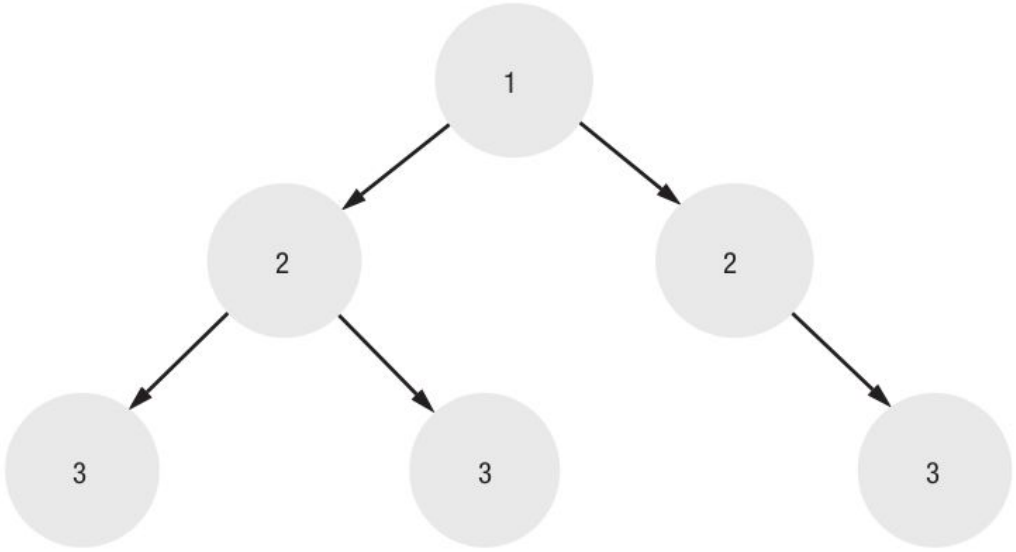
Nos EUA, NIST opera vários serv. NTP primários sincronizados com relógios atômicos de césio e o sistema GPS.

NTPs primários espalhados pelo mundo.

Topologia hierárquica dividida em camadas (strata) numerados de 0 a 16.

- stratum 0 é a referência de tempo: rel. atômico ou GPS, e não faz parte dos servidores NTP.

NTP – Network Time Protocol



Nota: as setas indicam controle de sincronização, os número fornecem o stratum.

4

- Quanto mais próximo da raiz, maior a precisão.

[4] Coulouris (2013), Fig. 14.3

Algoritmo de Berkeley

Ao contrário do NTP, onde o servidor de tempo é passivo, o algoritmo Berkeley consulta todas as máquinas de tempos em tempos, obtendo o horário de cada máquina

- Gera uma média de todas as horas, e informa a todos os computadores o deslocamento de tempo a ser feito
- O servidor muda inclusive a própria hora

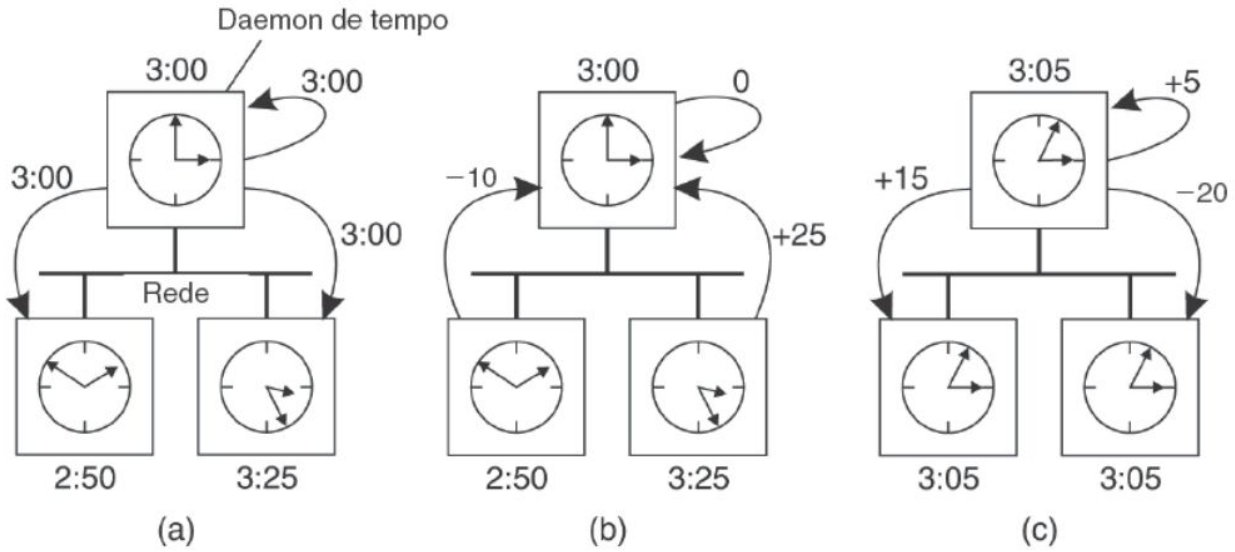


Figura 6.7 *(a) O daemon de tempo pergunta a todas as outras máquinas os valores marcados por seus relógios. (b) As máquinas respondem. (c) O daemon de tempo informa a todas como devem ajustar seus relógios.*

Sincronização em broadcast de referência – RBS

Um dos objetivos é economia de energia;

- Não adota que há dispositivos com hora real
- Visa mera sincronização (como Berkeley)

Não utiliza duas vias para sincronização:

- Nó transmite uma mensagem de referência m
- Cada nó p registra a hora Tp,m em que recebeu m
- Deslocamento de tempo entre nós p e q é dado pela média aritmética das diferenças de tempo de chegada de cada mensagem m trocadas entre p e q
- Cada máquina mantém o deslocamento médio de tempo entre si, não necessitando ajustar o relógio
- Pode-se aplicar outras técnicas como regressão linear padrão para minimizar efeito cumulativo pelo tempo

Sincronização
Lógica de Relógios

Algoritmos Distribuídos

Principais propriedades de um algoritmo distribuído:

- Informações são distribuídas pelos vários nodos do sistema
- Processos tomam decisões baseados apenas nas informações locais
- Sem um ponto único de falhas
- Não existe um relógio global a todo sistema

Ordenação de Eventos

Sistema centralizado: trivial

- Relógio comum a todos os processos

Sistema distribuído: não é nada trivial

Até o momento foi considerada a sincronização de relógios como naturalmente relacionada com a hora real.

- Lamport mostrou que, embora a sincronização de relógios seja possível, não precisa ser absoluta:
- Não é necessário sincronizar processos que não interagem entre si
- Soluciona problemas relativos à ordem de eventos independentemente da hora real.

Relação Happens-Before

Proposta por Lamport em um paper clássico:

- Lamport, L. Time, clocks and the ordering of events in a distributed system, CACM, vol. 21, pp. 558-564, july 1978.

Idéias básicas:

- Definir ordem de eventos apenas entre processos que interagem entre si
- Relógio lógico: o que importa é a ordem dos eventos e não o tempo físico em que os mesmos ocorreram

Relação Happens-Before

Notação: A→ B

- Lê-se que "A ocorreu antes de B"
- Significado: todos os processos do sistema concordam que primeiro ocorreu o evento A e então o evento B

Eventos que não são ordenados pela relação "→" são ditos concorrentes.

- Notação: A \\ B

Relação Happens-Before

Definição:

- Se A e B são eventos de um mesmo processo e A foi executada antes de B, então $A \rightarrow B$
- Se A consiste no evento de enviar uma mensagem para um processo e B é o evento de recebimento desta mensagem, então $A \rightarrow B$.
- Se $A \rightarrow B$ e $B \rightarrow C$, então $A \rightarrow C$ (transitividade)
- Se x e y acontecem em processos diferentes que não trocam mensagens, então tanto $A \rightarrow B$ quanto $B \rightarrow C$ são **falsas**

Implementação da Relação Happens-Before

Algoritmo de Lamport

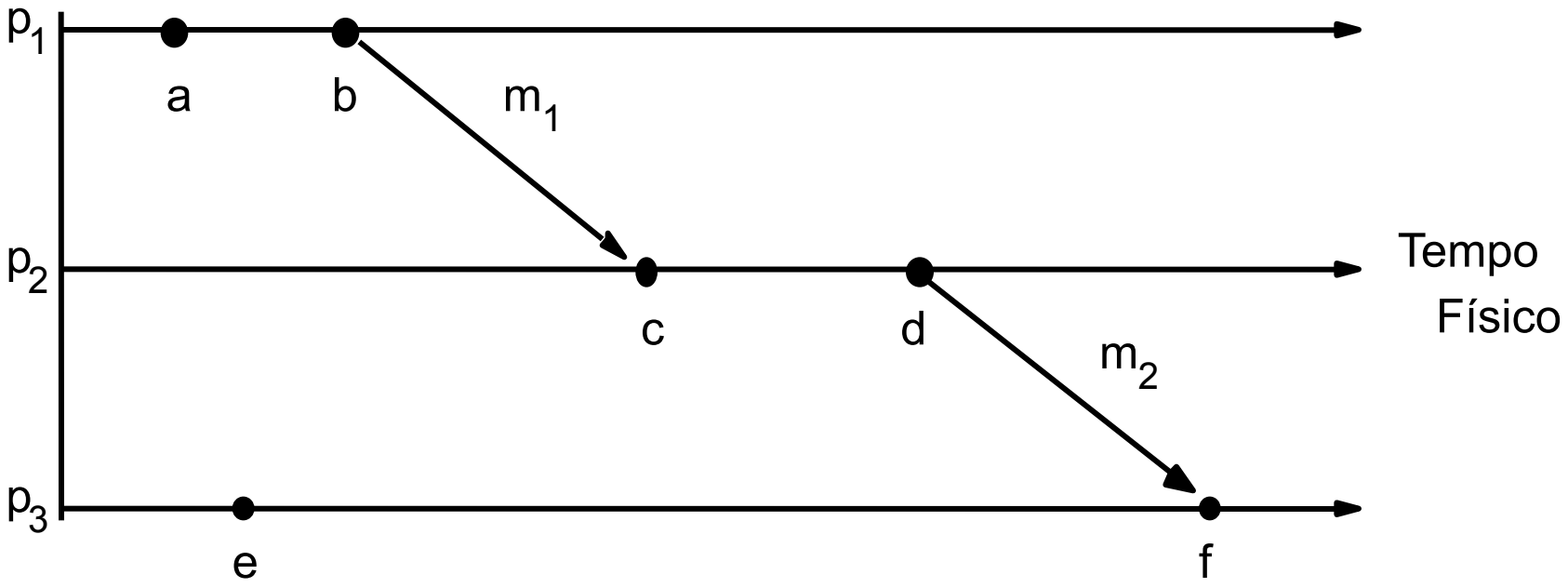
Utiliza o conceito de relógio lógico (C):

- Inteiro monotonicamente crescente armazenado em cada nodo
- Incrementado a cada evento

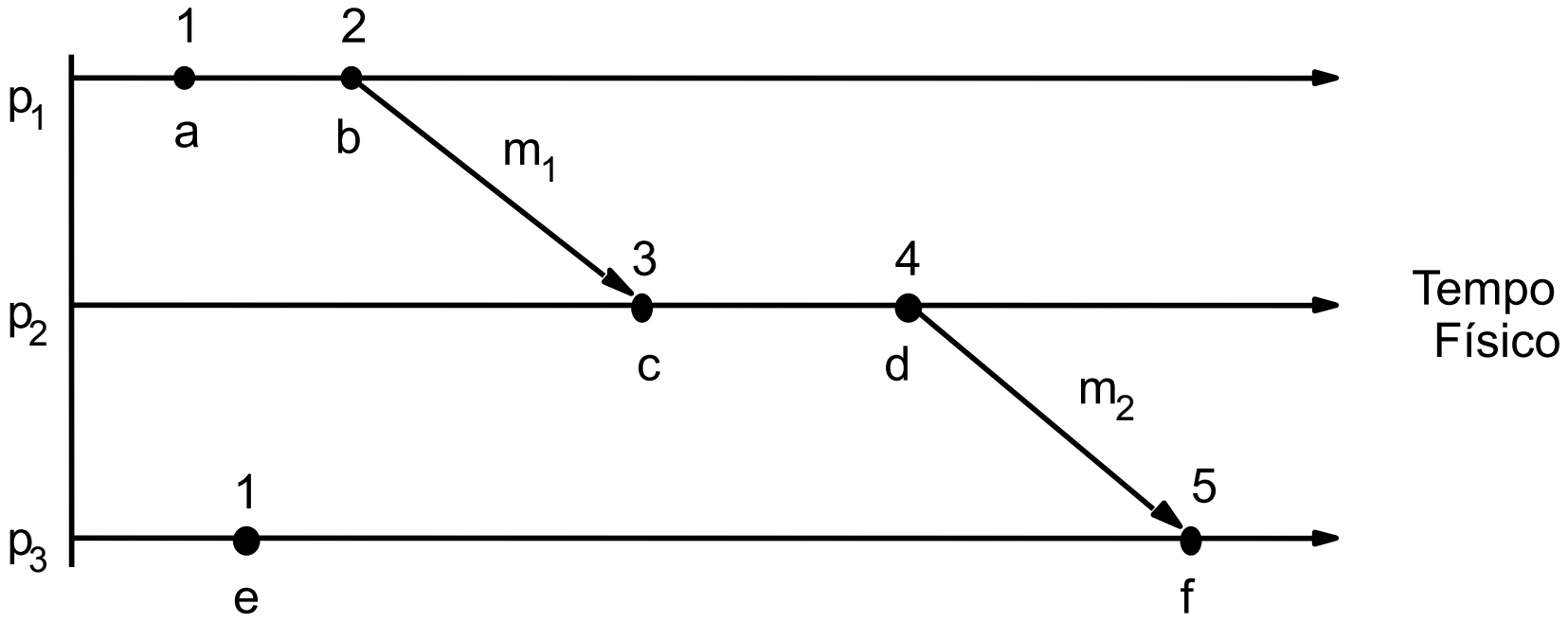
Implementação do conceito de relógio lógico:

- Se $A \rightarrow B$, então $C(A) < C(B)$

Ocorrência de eventos nos processos P1, P2 e P3



Rótulos de tempo do Algoritmo de Lamport para ordenação de eventos



Relógio Lógico

O valor do relógio lógico de um processo P, Cp, é atualizado da seguinte forma:

- Cp é incrementado antes de cada evento de P.
- Quando P envia uma mensagem M para Q, ele insere na mesma um timestamp t= Cp
- Quando Q recebe (M, t), então

  Cq:= max (Cq, t) + 1

  (garante-se assim que Cp < Cq)

Relógio Lógico

Relógio Lógico: como definido, dá origem a uma relação de ordem parcial

- Dois eventos podem ocorrer no mesmo tempo lógico

Ordenação total : adicionar o PID de cada processo no final do relógio lógico

- Exemplo:
  - Processo 1: relógio lógico = 40.1
  - Processo 2: relógio lógico = 40.2

Relógio Lógico

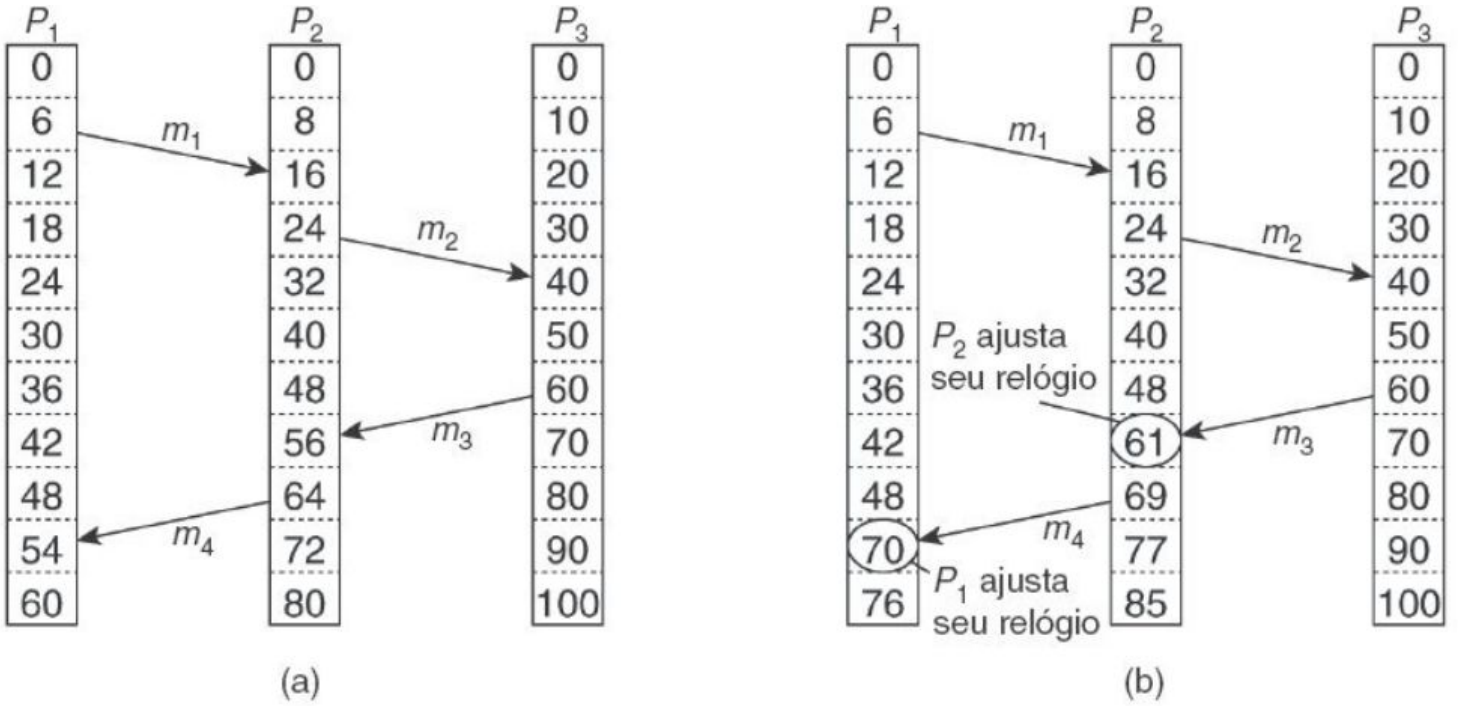


Figura 6.9 (a) Três processos, cada um com seu próprio relógio. Os relógios funcionam a taxas diferentes. (b) O algoritmo de Lamport corrige os relógios.

Relógio Lógico

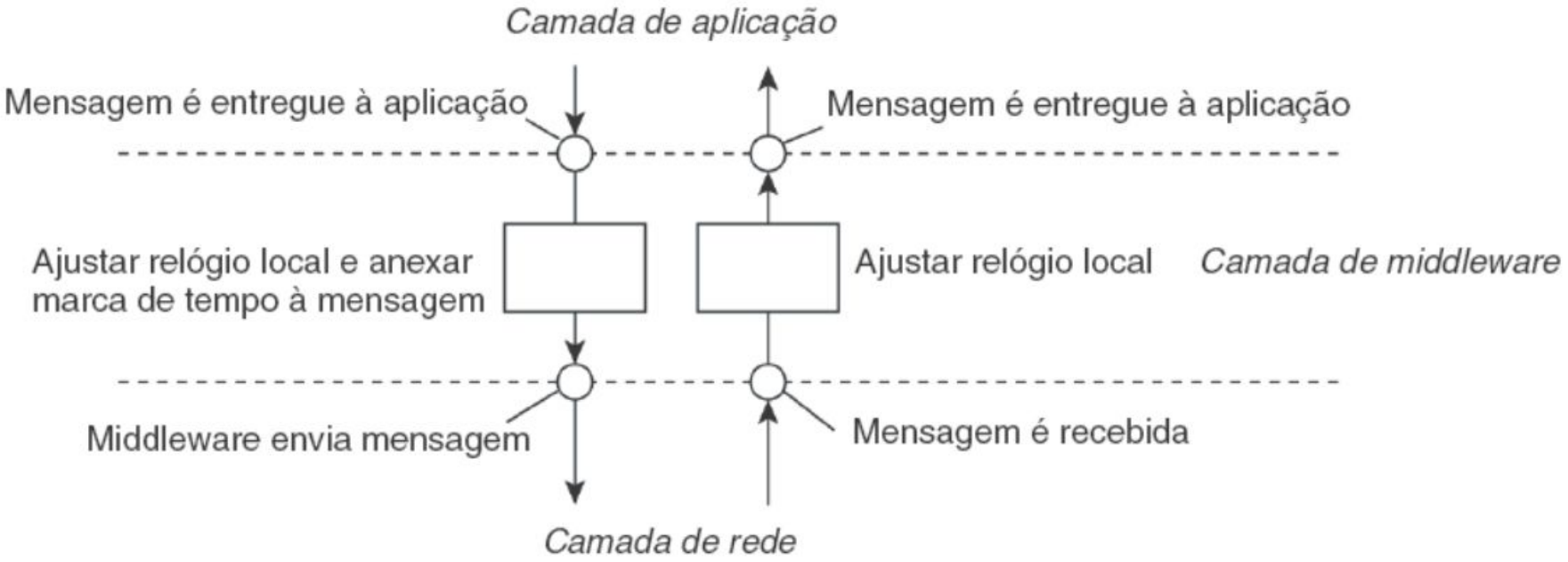


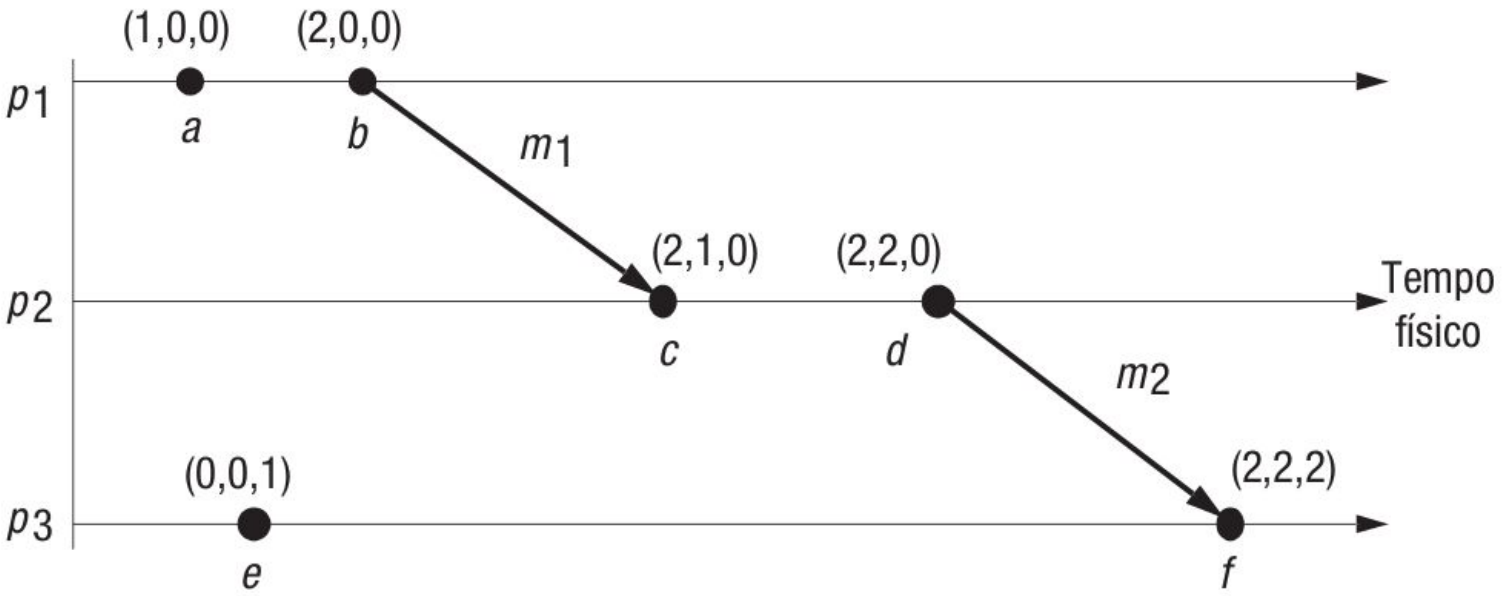
Figura 6.10 Posicionamento de relógios lógicos de Lamport em sistemas distribuídos.

Relógios vetoriais (vector clocks)

Objetivo: eliminar a deficiência de que se CI < CJ nada se pode concluir que I $\rightarrow$ J.
Um relógio vetorial para um sistema com N processos possui N inteiros.
Algoritmo:

- Inicialmente, $V_i[j] = 0$, para i, j = 1, 2..., N.
- Imediatamente antes de p_i gerar o timestamp de um evento, ele configura $V_i[i] := V_i[i] + 1$.
- p_i inclui o valor $t = V_i$ em cada mensagem que envia.
- Quando p_i recebe um timestamp t em uma mensagem, ele configura $V_i[j] := \max(V_i[j], t[j])$, para j = 1, 2..., N.
- Operação de merge (integração).

Relógios vetoriais (vector clocks)



[7] Coulouris (2013), Fig. 14.7

Relógios vetoriais (vector clocks)

Relações entre relógios vetoriais:

- $V = V'$ sse $V[j] = V'[j]$ para $j = 1, 2..., N$
- $V \leq V'$ sse $V[j] \leq V'[j]$ para $j = 1, 2..., N$
- $V < V'$ sse $V \leq V' \wedge V \neq V'$

Se $V(e) < V(e')$, então, $e \rightarrow e'$.

Problema: consumo de memória e overhead de mensagem.

Exclusão Mútua Distribuída

Exclusão Mútua Distribuída

Sistemas Centralizados: memória compartilhada (semáforos, monitores, etc.)

Sistemas Distribuídos: via troca de mensagens

Requisitos:

- Segurança: No máximo um processo pode estar executando na seção crítica (SC)
- Liveness: Um processo que requisita acesso à SC, obtém este acesso em algum momento. Ou seja, não há deadlock ou starvation.

Primeira Solução: Algoritmo Centralizado

Existe um processo coordenado (PC)

- Gerencia o acesso à SC

Processos comuns

- Antes de entrar na SC: enviam request ao PC
- Quando recebem reply: entram na SC
- Quando saem da SC: enviam release ao PC

Processo Coordenador:

- Enfileira request quando a SC ocupada
- Desenfileira request ao receber um release

Primeira Solução: Algoritmo Centralizado

Número de mensagens trocadas para entrar na SC: duas (um *request* e um *reply*)

Saída da SC: uma mensagem (*release*).

Problema: processo que centraliza todas as informações e decisões

Solução: distribuir a fila de pedidos

- Todos os nós recebem todos os pedidos
- Pedidos são ordenados pelos nós da mesma maneira

Primeira Solução: Algoritmo Centralizado

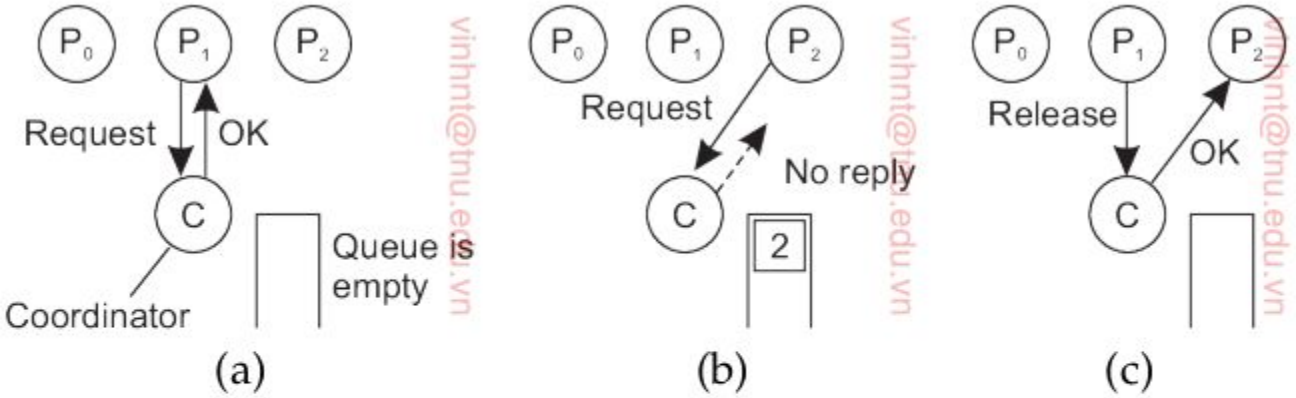


Figure 6.15: (a) Process P_1 asks for permission to access a shared resource. Permission is granted. (b) Process P_2 asks permission to access the same resource, but receives no reply. (c) When P_1 releases the resource, the coordinator replies to P_2.

Algoritmos de Exclusão Mútua Distribuída

Principais Algoritmos:

- Lamport
- Ricart e Agrawala (*)
- Osvaldo e Roucariol (*)
- Maekawa
- Token Ring (*)

(*) Algoritmos que serão estudados a seguir

Algoritmo de Ricart e Agrawala

Algoritmo executado por cada processo Pi, que compartilha uma SC (i <= n):

- Variáveis:
  - estado: estado da seção crítica
  - OSN: relógio lógico do processo
  - HSN: maior valor do timestamp de qualquer mensagem enviada ou recebida
- Inicialização:
  - estado:= livre;
  - HSN:= 0;

Algoritmo de Ricart e Agrawala

Quando processo deseja entrar na SC:

estado:= aguardando;

OSN:= HSN + 1;

“Enviar requisição <OSN, i> para todos processos, exceto Pi”

“Espere até que o número de respostas seja igual a n-1"

estado:= ocupado

Algoritmo de Ricart e Agrawala

Quando processo recebe requisição [k, j]:

HSN:= max (HSN, k)

se (estado = livre)

então “envie reply para Pj”

se (estado = ocupado)

então “enfileire requisição [k, j]”

se (estado = aguardando)

então se [OSN, i] < [k, j]

então “Enfileire requisição [k, j]"

senão “Envie reply para Pj”

Algoritmo de Ricart e Agrawala

Ao sair da SC:

estado:= livre;

"Envie reply para todas requisições enfileiradas (e a retire da fila)"

Demonstração:
https://misterdoom4.github.io/TCC/algoritmos/ricart_agrawala.html

Algoritmo de Ricart e Agrawala

Número de mensagens trocadas para entrar na SC: 2 (n-1)

- n-1: requests
- n-1: replys

Vantagem:

- Distribuído (sem um ponto central de falha)
  - Considerando que exista uma forma de detectar falha em um nó após um certo tempo limite.

Algoritmo de Carvalho e Roucariol

Algoritmo de Ricart & Agrawala:

- Número de mensagens trocadas: 2(n-1)
- "Ótimo, no sentido de que nenhum outro algoritmo distribuído e simétrico pode usar menos mensagens"

Algoritmo de Carvalho e Roucariol (1983):

- Mostraram que a afirmativa acima não é verdadeira
- Número de mensagens trocadas: entre 0 e 2(n-1)

Algoritmo de Carvalho e Roucariol

Idéia básica:

- Se Pi recebeu uma mensagem *reply* de Pj, então a autorização implícita nesta mensagem permanece válida até que Pi receba um novo *request* de Pj
- Enquanto isto não acontecer, Pi pode acessar a SC sem consultar Pj

Algoritmo de Carvalho e Roucariol

Análise:

- Melhor caso: processo que conseguir (n-1) autorizações pode acessar a SC sem enviar nenhuma mensagem (enquanto nenhum outro processo desejar acessar o recurso)
- Pior caso: entre um acesso à SC e o acesso seguinte, processo recebe (n-1) requisições de processos distintos. Logo, deverá enviar (n-1) *requests* para voltar a acessar a SC.

Algoritmo Token Ring

Supõe que os processos são organizados como um anel lógico, por onde circula um *token*

- Não exige que a topologia da rede seja em anel (anel lógico e não físico)
- Cada processo deve armazenar a configuração completa do anel

Ao processo de posse do *token* é garantido o acesso à SC

Algoritmo Token Ring

Idéia básica:

- Se um processo não deseja entrar na SC:
  - Ao receber o token, repassa o mesmo para seu vizinho
- Se um processo deseja entrar na SC:
  - Aguarda a passagem do token e o retém

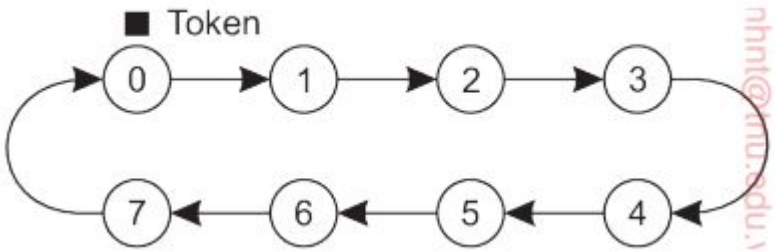
Número de mensagens trocadas: entre 1 e n-1

Algoritmo Token Ring

Vantagens: simplicidade

Desvantagem: tratamento de erros

- Perda da mensagem que contém a token
  - Deve-se gerar a token novamente
  - Quando gerar ? Quem deve gerar ?
- Travamento de processos
  - Deve-se reconfigurar o anel



Algoritmos de Eleição

Algoritmos de Eleição

Objetivo: escolher um líder – processo que seja coordenador dentre um conjunto de processos.

Suposição: cada processo tem um número exclusivo (endereço de rede, por exemplo)

Abordagem geral: procurar processo com número mais alto e designá-lo como líder.

Algoritmos variam na maneira de localizar tal processo.

Algoritmos de Eleição

- Todo processo sabe o número de todos os outros.
- Processos não sabem quais estão funcionando e quais estão inativos.
- **Meta do algoritmo de eleição**: garantir que, quando uma eleição começar, ela terminará com todos os processos concordando com o novo coordenador escolhido.

Algoritmo do Valentão

Proposto por Garcia-Molina (1982)

Processo **P** qualquer nota que coordenador não está mais respondendo, inicia eleição da seguinte forma:

- Envia mensagem **ELEIÇÃO** a todos os processos de número mais alto
- Se nenhum responder, **P** vence a eleição e se torna coordenador
- Se algum responder, este toma o poder e **P** conclui seu trabalho.

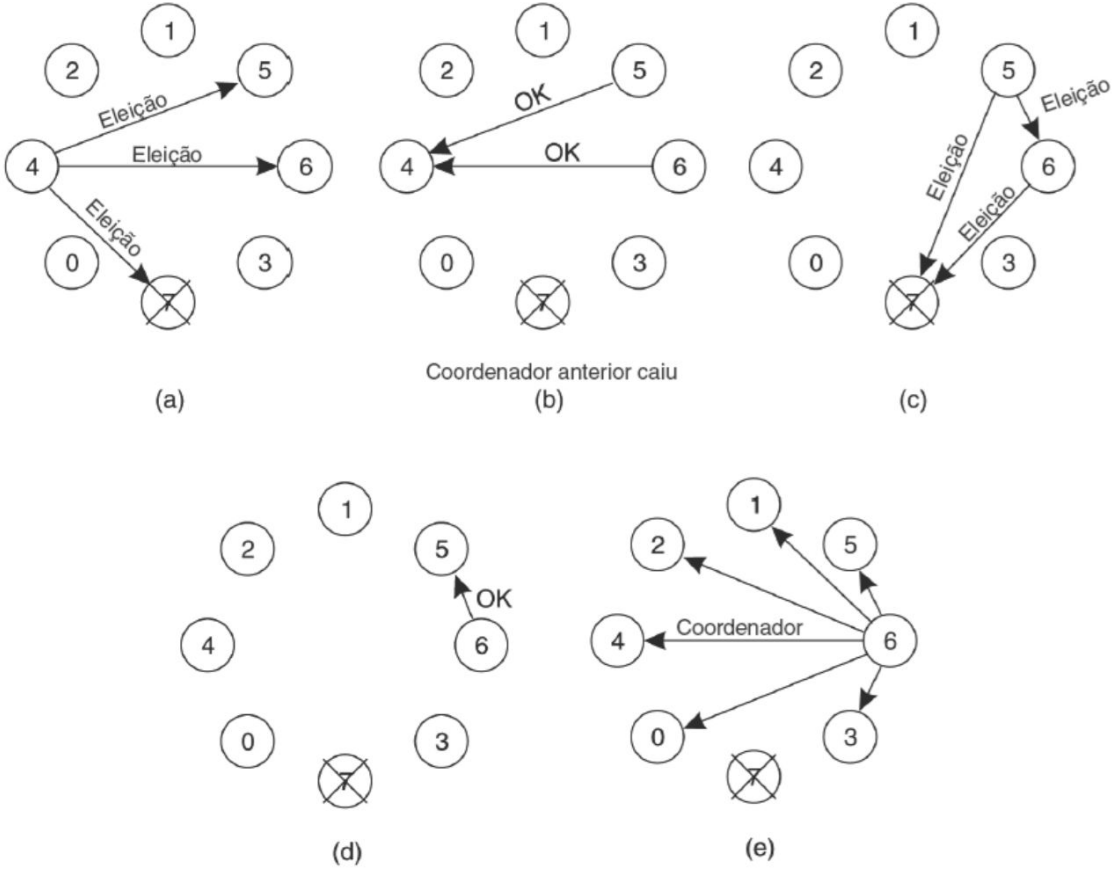
Algoritmo do Valentão

- Processos podem receber mensagem **ELEIÇÃO** a qualquer momento de nós com número mais baixo.
- Receptor envia OK de volta ao remetente, indicando que está vivo e que tomará o poder.
- Receptor convoca uma eleição (a não ser que já tenha convocado uma)
- Converge para situação onde todos desistem, exceto um, que será o novo coordenador.
- Este anuncia a vitória enviando a todos os processos informando que ele é o novo coordenador.

Algoritmo do Valentão

- Processo inativo convoca eleição quando volta.
- Se for processo de número mais alto, ganha
- Mais “poderoso” sempre ganha

Algoritmo do Valentão



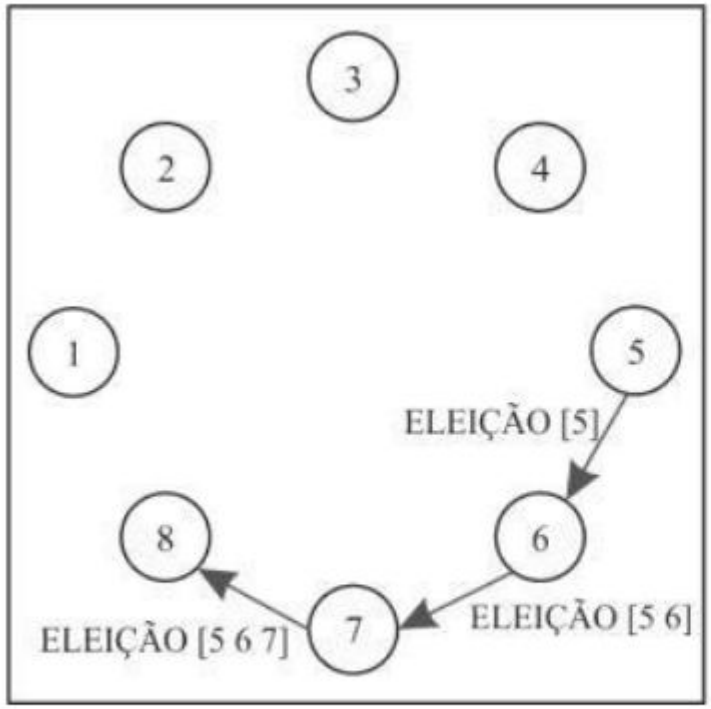
Algoritmo do Anel

- Processos ordenados por ordem física ou lógica
  - Cada um sabe quem é seu sucessor
- Processo nota que coordenador não responde->envia mensagem ELEIÇÃO, contendo seu número de processo
  - Se o sucessor não estiver respondendo, envia ao próximo membro ao longo do anel. Repete até encontrar um processo funcional
- Cada nó adiciona seu número de processo à mensagem, candidatando-se a coordenador

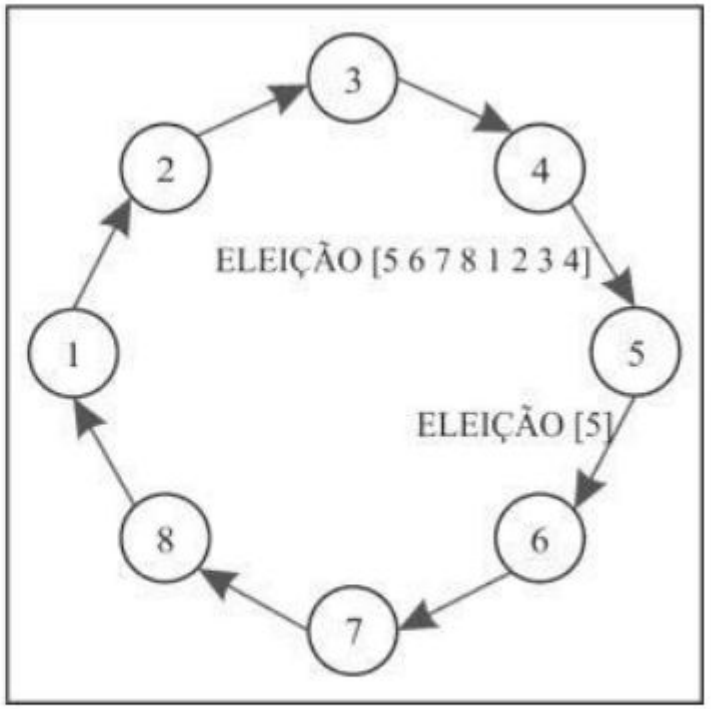
Algoritmo do Anel

- Quando mensagem retornar ao iniciador da eleição:
- Extrai maior número de processo que encontrar na mensagem.
- Circula mensagem COORDENADOR com tal número, além dos participantes do anel.
- Mensagem COORDENADOR chegou ao iniciador, é removida.

Algoritmo do Anel



(a)



(b)

Eleição em ambiente sem fio

Sem premissa de que troca de mensagem é confiável e topologia não muda.

Em especial redes ad hoc.

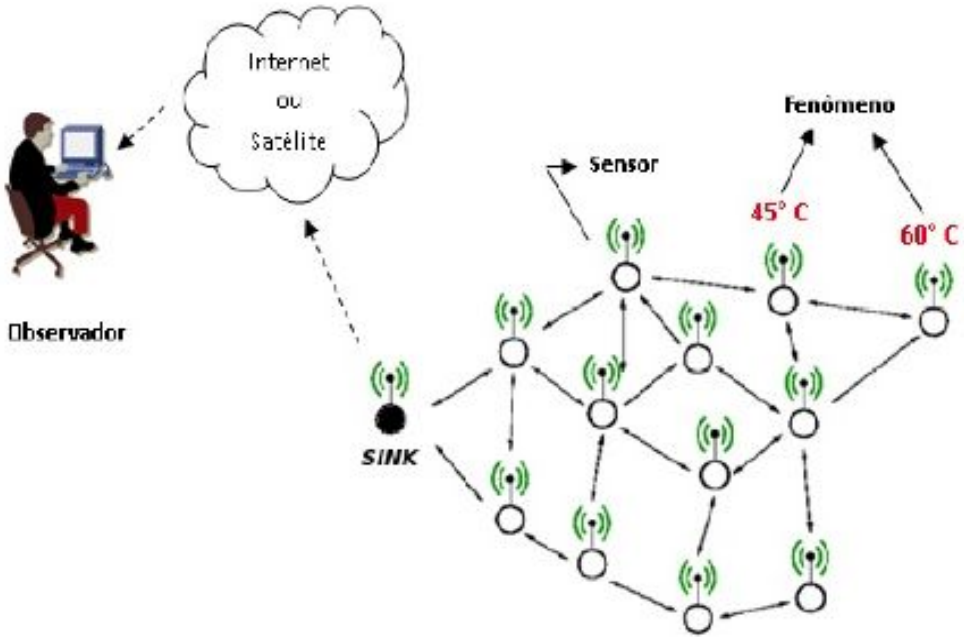
Eleição em ambiente sem fio

Critérios para seleção em redes de sensores sem fio:

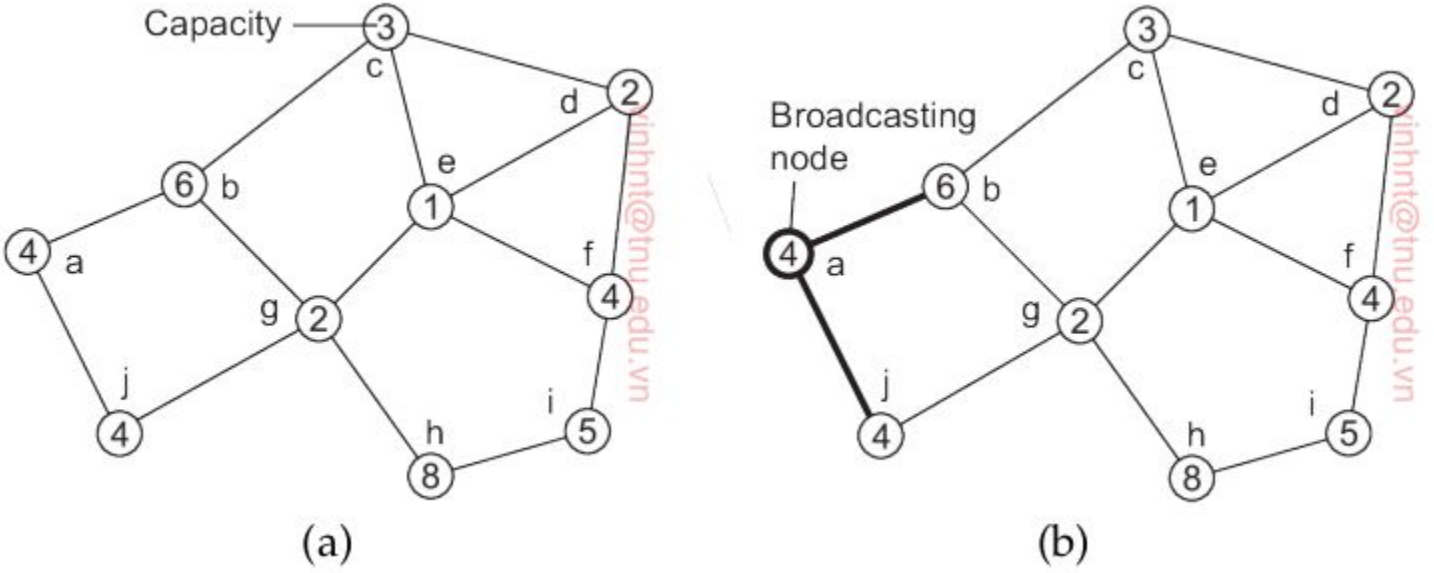
-O nó mais próximo do sink

-O nó com mais capacidade

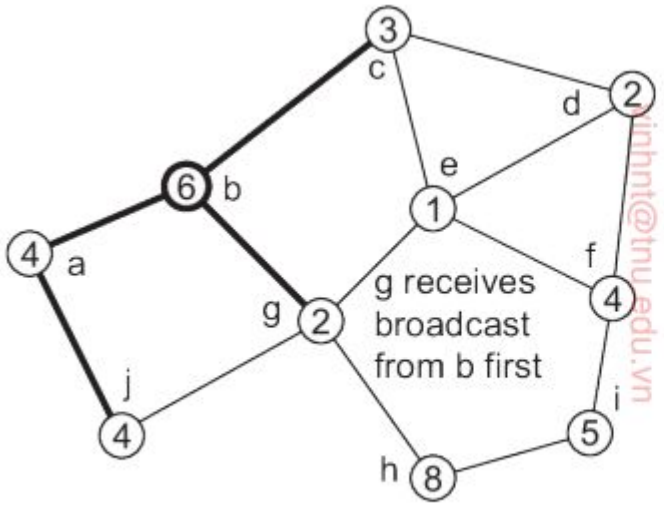
-O nó de maior *id*



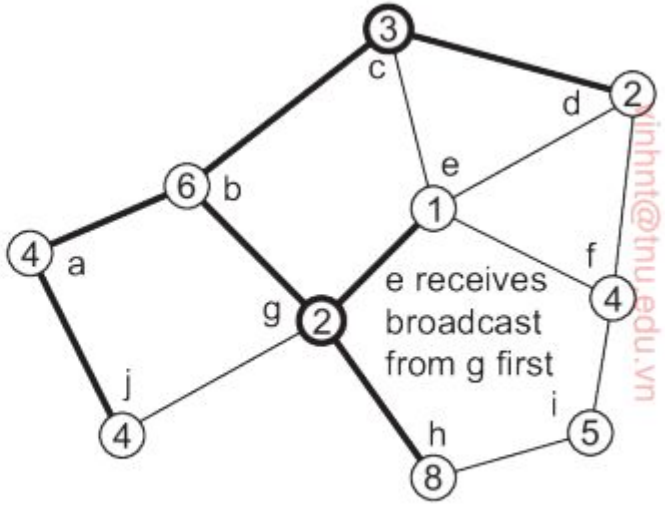
Eleição em ambiente sem fio - Exemplo de algoritmo



Eleição em ambiente sem fio - Exemplo de algoritmo

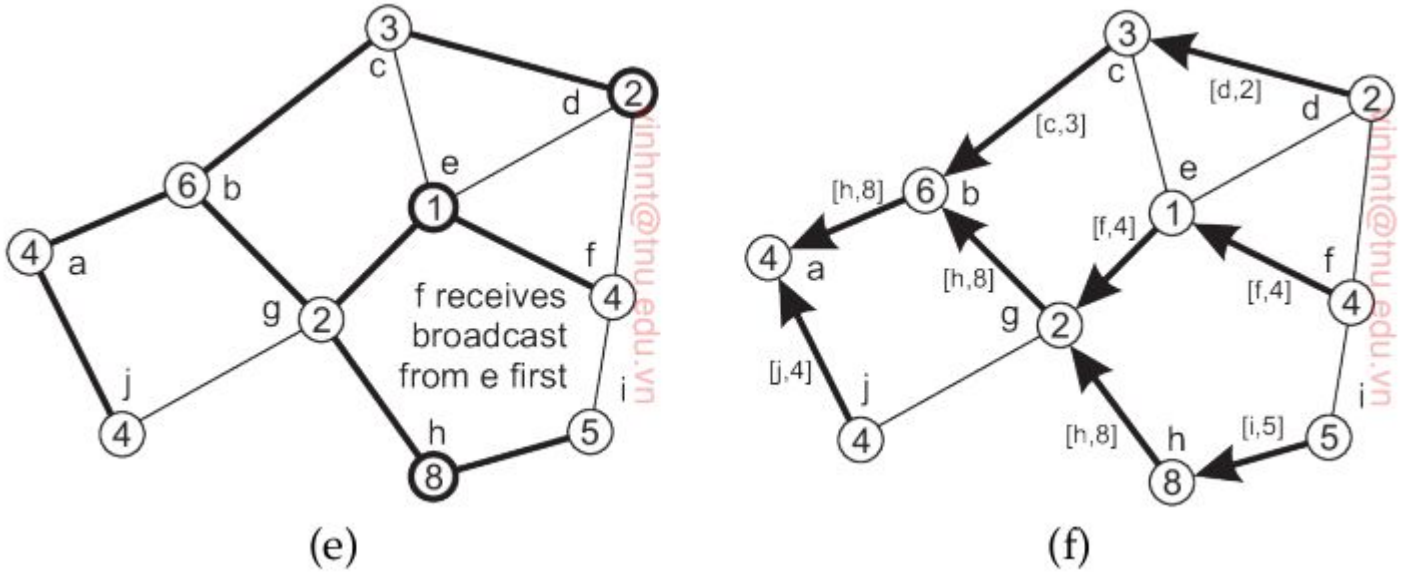


(c)



(d)

Eleição em ambiente sem fio - Exemplo de algoritmo



Vencedor: nó h

Eleição em ambiente sem fio

Tarefas do líder em redes de sensores sem fio:

-Eleger o próximo líder

-Fusão de dados

-Encaminhamento de queries para rede

-Monitoramento das tarefas do cluster

-Sincronização

Algoritmo yo-yo

É um algoritmo de busca mínima, onde o nó com menor identificação (ID) é escolhido como líder.

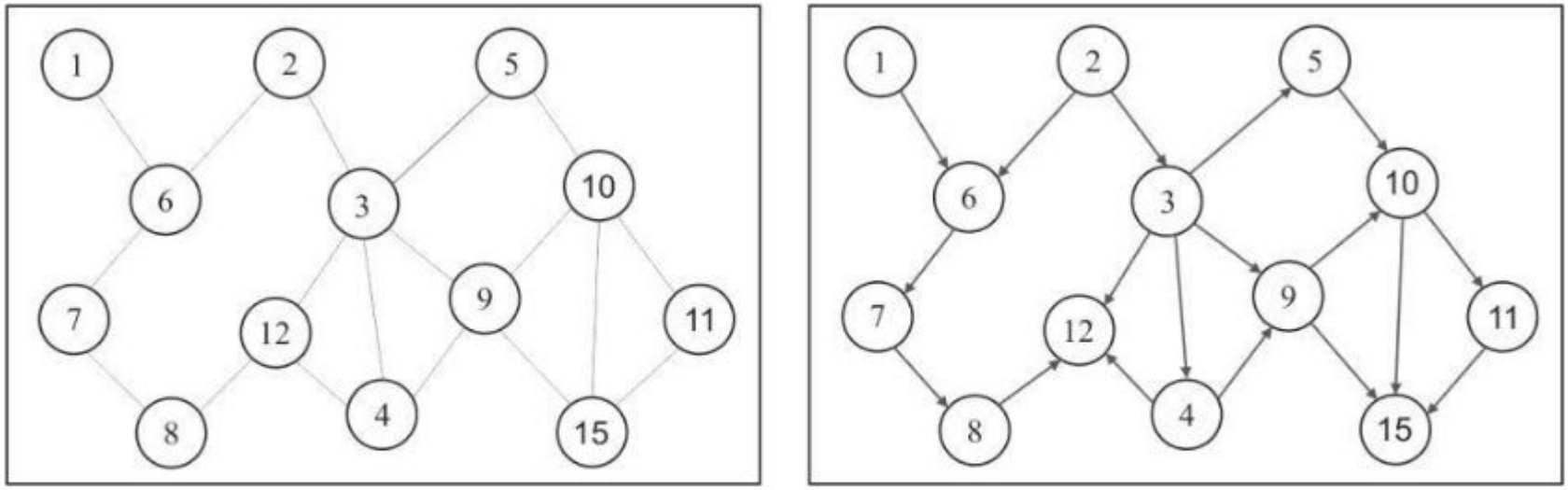


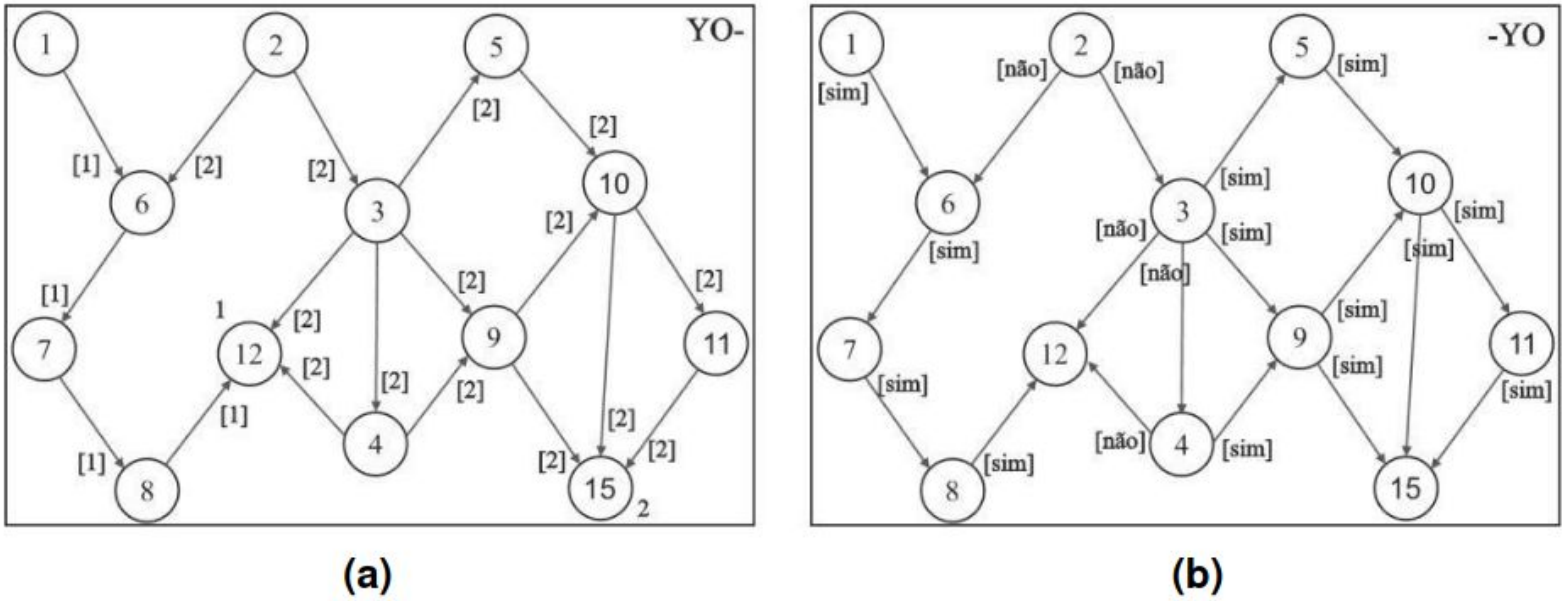
Figura 4. Organização do cenário no *setup*

Algoritmo yo-yo

No exemplo anterior, os nós 1 e 2 serão *sources*, os nós 15 e 12 serão *sinks* e os demais nós serão *internals*

- O source é um nó que é possui ID menor que todos os seus vizinhos e onde os links só saem dele (ele é um mínimo local)
- sink é um nó que é o maior que todos os vizinhos, onde os links só chegam nele e nenhum sai dele (ele é um máximo local)
- internal é um nó que não é um source e nem um sink.

Algoritmo yo-yo



(a) (b)

a) a partir dos *sources*, cada nó encaminha o menor id passado à ele.
b) a partir dos *sinks*, cada nó responde com uma mensagem de "sim" se o menor id veio daquela conexão, e "não" caso contrário. O nó *source* que receber um "sim" ao final vira o líder

Análise de Desempenho

Artigo: Análise e Modelagem de Algoritmos para Eleição de Líder em Sistemas Distribuídos

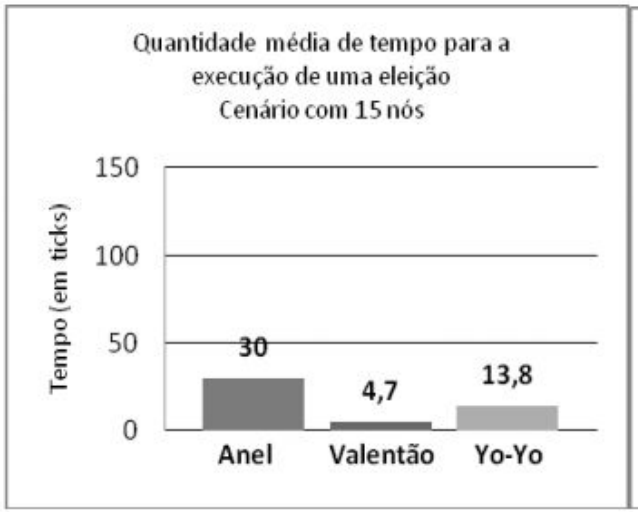
Émerson R. Silva1, Eduardo C. Julião1, Patricia T. Endo1
1GRupo de Estudos Avançados em Tecnologia da Informação e Comunicação (GREAT)
Universidade de Pernambuco (UPE) – Campus Caruaru – Caixa Postal 55014-908 – Caruaru – PE – Brazil

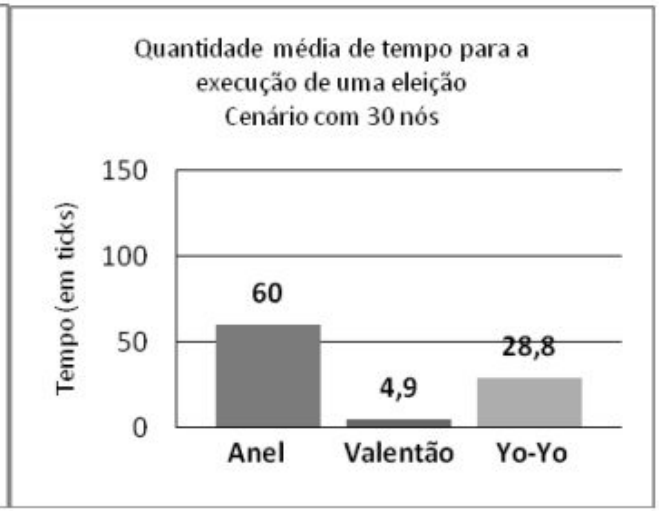
Ferramenta NetLogo

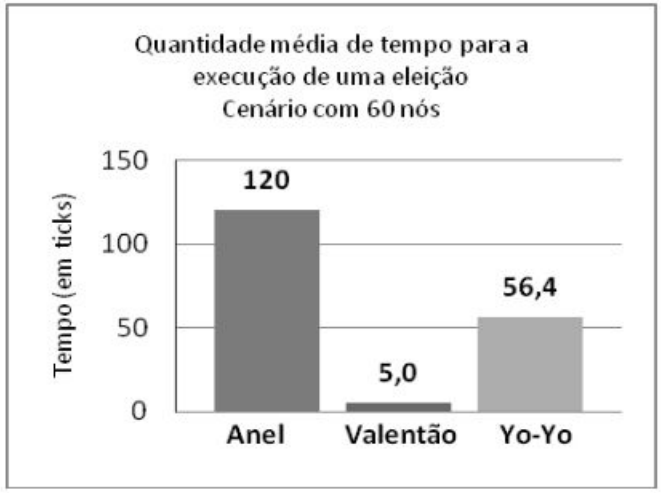
Algoritmos avaliados: Valentão, Anel, Yo-Yo

Topologia de rede com 15, 30 e 60 nós

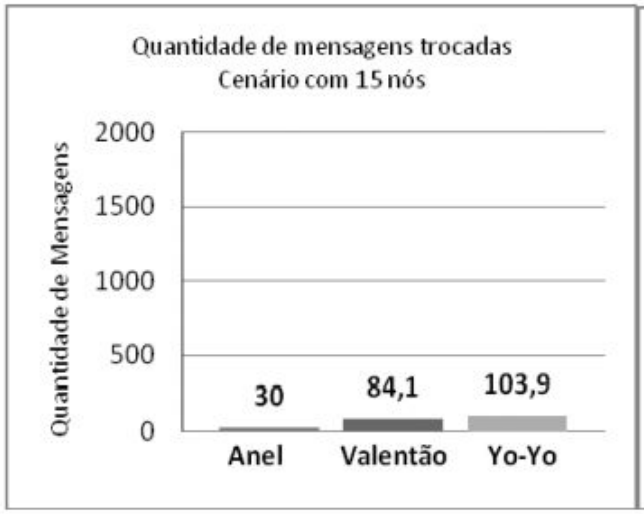
Análise de Desempenho

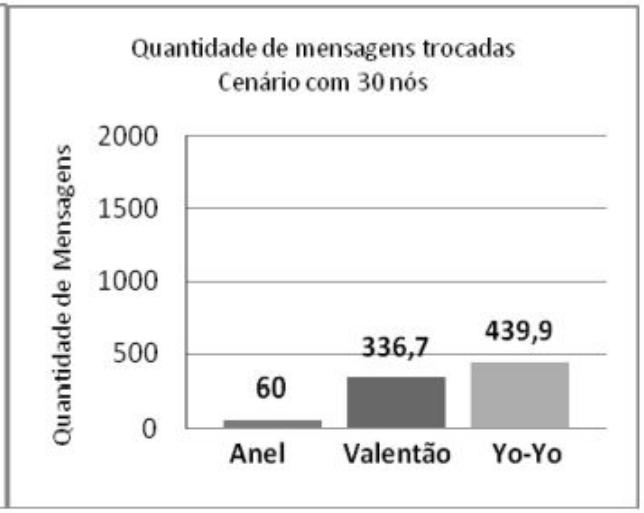


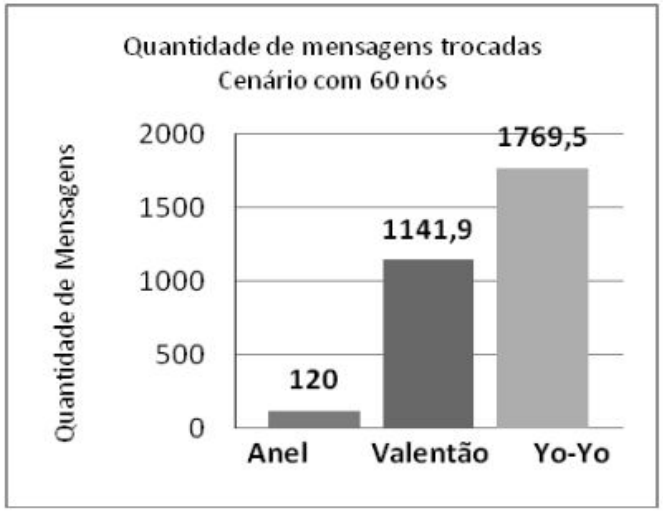




Análise de Desempenho







Deadlocks em Sistemas Distribuídos

Deadlocks em sistemas distribuídos

Mais difíceis de tratar.

Estratégias para lidar com deadlocks:

- Algoritmo de Ostrich (ineficiente)
  - Ignora problemas potenciais com base no fato de que eles podem ser extremamente raros
  - É mais rentável permitir que o problema ocorra do que tentar a sua prevenção
- Detecção - deixa deadlocks ocorrerem e depois tenta recuperar estado
- Prevenção - estaticamente torne deadlocks estruturalmente impossíveis
- Evitar deadlocks alocando recursos com cuidado (dificilmente usado por ser extremamente complexo).

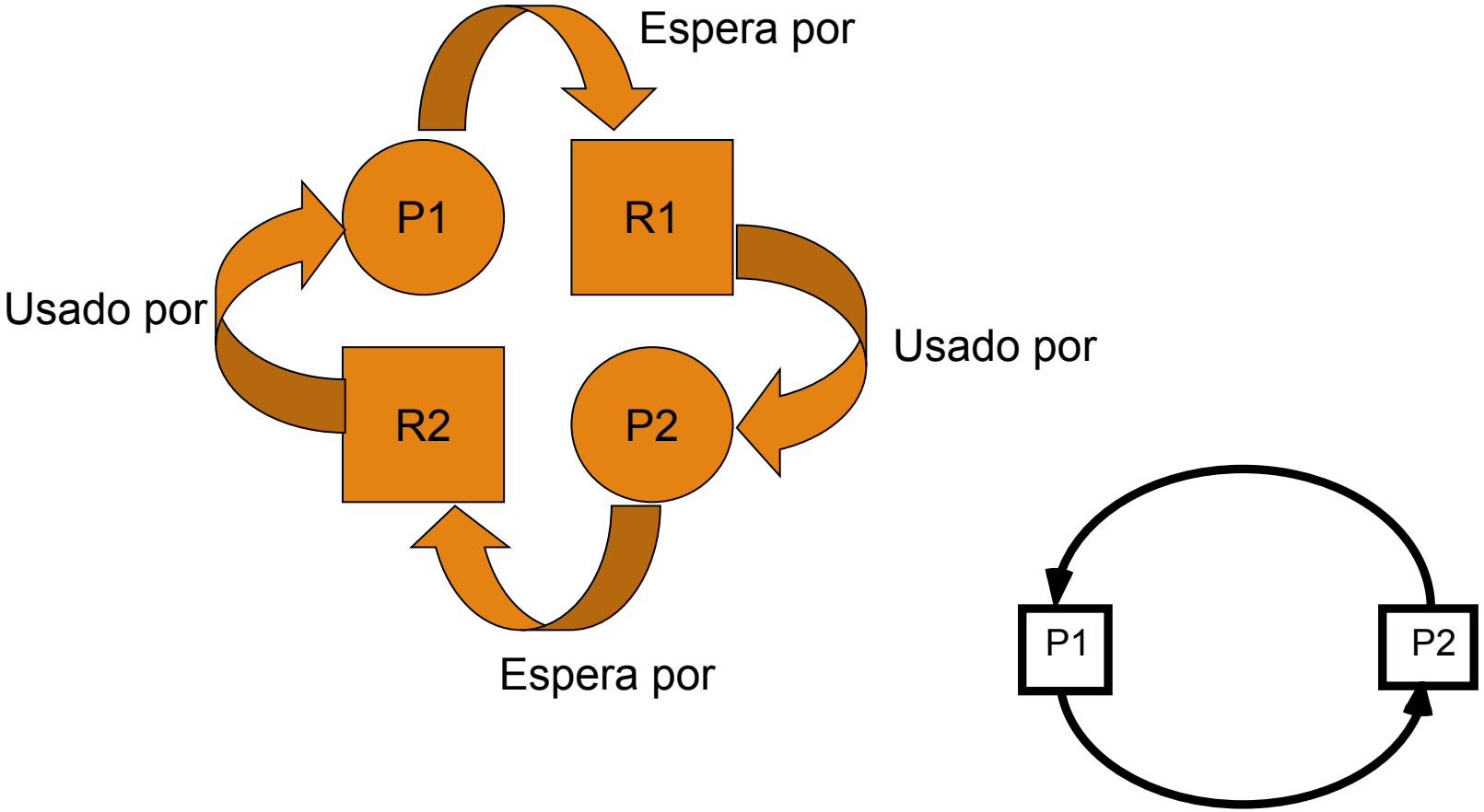
Detecção de deadlocks

Caso um deadlock é detectado, algumas transações atômicas que ocasionaram o deadlock são destruídas.

Método Centralizado:

- Cada máquina possui o seu grafo de alocação de recursos e envia para o coordenador.
- Caso um ciclo seja detectado pelo coordenador, um processo deve cancelar a espera ou utilização de um recurso.

Detecção de deadlocks



Detecção de deadlocks

Método Centralizado:

- Problema: Ocorrência de falsos deadlocks devido a atrasos em mensagens.
- Solução: Usar algoritmo de Lamport para tempo global. Caso ciclo seja detectado, coordenador pede por grafos atualizados.

Detecção de deadlocks

Método Distribuído:

- Algoritmo de Chandy-Misra-Haas
- Processos podem pedir múltiplos recursos.
- Processos enviam mensagens para os processos que estão esperando. Se um loop for detectado, processos são eleitos para liberarem recursos.

Eliminando deadlocks

Processo que iniciou a prova do deadlock comete suicídio.

- Problema: se vários processos iniciam provas em paralelo podem ter morte em massa.

Incluir ID do processo na prova. Processo de maior número (mais novo) deverá cometer suicídio.

Problemas: Mandar e analisar provas quando se está bloqueado não é trivial.

Prevenção de deadlocks

Tornar deadlocks impossível estruturalmente!

Método 1:

- Cada processo só pode usar um recurso de cada vez e deve liberar o recurso a fim de pedir outro.

Método 2:

- Recursos são ordenados de forma global. Processos devem pedir recursos em ordem crescente.

Prevenção de deadlocks

Método 3:

- Algoritmo wait-die
- Utilizar tempo global
- Processos só podem esperar por processos mais novos (tempo global maior).
- Caso um processo jovem precise de um recurso retido por um processo velho ele deverá cometer suicídio.