**CD**

**-Resumo Ch6-**

**Considerações Iniciais**

Este capitulo provem da ideia de que processos num ambiente distribuído necessitam de estarem sincronizados/coordenados para efetivamente trabalharem como suposto. São também abordados outros tipos de coordenação como **Exclusao Mutua** e algoritmos de **Eleiçao.**

**Sincronização:**

* **Process Synchronization:** Garantir que um processo espera por outro para completar a sua operação
* **Data Synchronization:** Garantir que 2 data sets são o mesmo (mais abordado no Ch7 – Replicação)

**Coordenação:**

* Gerir interações e dependências entre atividades num sistema distribuído
* Diz-se que “*encapsula a sincronização*”

**Clock Synchronization**

**-Considerações-**

**Physical Clocks:**

* Os computadores tem um **Timer**
* O timer posusi um cristal de quartzo que oscila a uma frequência +- bem defenida quando sob tensão
* Associado ao cristal estão 2 registos – 1 **Counter** que decrementa a cada oscilação ; 1 **Holding register** que recarrega o counter quando este chega a 0
* **Clock Tick**: Quando o counter chega a 0 é gerado um interrupt
* **Software Clock:** A cada clock tick, e adicionado um ao valor de tempo guardado em memoria (aka atualizamos o software clock)
* **É impossível garantir que cristais em varias maquinas oscilem à mesma frequência…**so that’s why we have problems
* **Clock Skew**: Diferença dos time values de diferentes timers

**UTC – Universal Coordinated Time:**

* Standard mundial
* Basis para menter tempo global
* Estações de radio e satélites oferecem este serviço
* **Receivers** recevem de vários satélites e ground time servers com uma accuracy de 50 nsec
* Relógio Atómico (Baseia-se na oscilação do átomo Celsiu-133)…so it’s precise as heck
* Se um DS tiver um UTC Receiver algures, os outros componentes devem tentar sincronizar-se a este

**Precision:**

* Usado como “bound” da diferença de time values entre 2 timers
* Denota-se por Pi
* 2 relógios, p e q, são preciso se:
  + 
  + Cx(t) -> Time value do relogio x no instante (UTC) t

**Accuracy:**

* Usado como “bound” da diferença de time values entre 1 timer e um UTC Receiver
* Denota-se por Alfa
* 1 relógios, p, é exato se:
  + 
  + Cx(t) -> Time value do relogio x no instante (UTC) t

**Nota**: Usamos:

-Precision para manter **INTERNAL SYNCHRONIZATION**

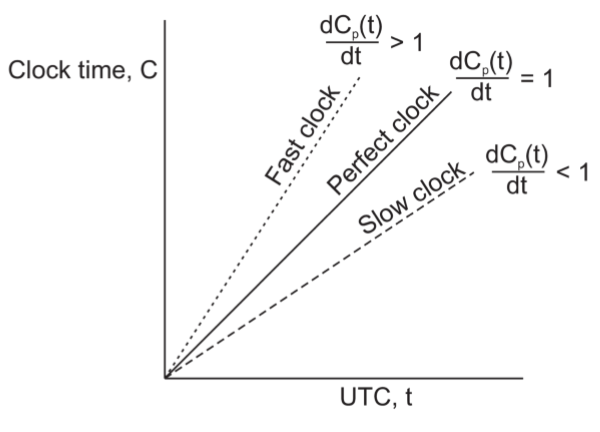
-Accuracy para manter **EXTERNAL SYNCHRONIZATION**

**Nota**: Um set de relógios **accurate em alfa** também será **precise em pi=2\*alfa**

**Nota**: **Caso ideal** seria termos Precision = Accuracy = 0, e Cp(t) = t, sempre…but life ain’t that good :(

**Clock Drift:**

* A frequência dos relógios não e perfeita e e afetada por fatores externos, portanto clocks em diferentes maquinas vao gradualmente começar a mostrar diferentes valores de tempo – **Clock Drift Rate**
* Existe um **Maximum clock drift rate** – **Ró –** dependente do hardware clock
* Devemos procurar manter o Software clock drift rate bounded a Ró
  + 1-Ro <= dCp(t)/dt <= 1+Ro

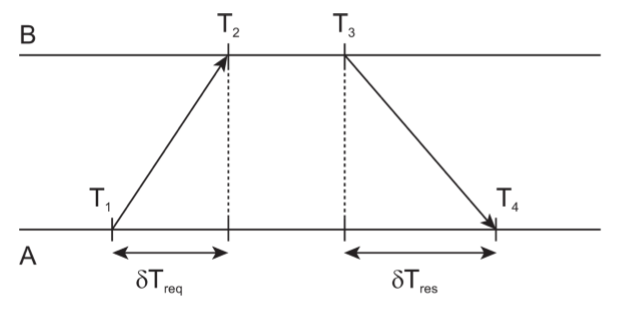


**Nota++**: Para assegurarmos que clocks são **Precisos para pi** (não diferem os seus time values mais de pi), temos que **RESINCRONIZAR** (em software) **os relógios a cada Pi/2(Ro) segundos!**

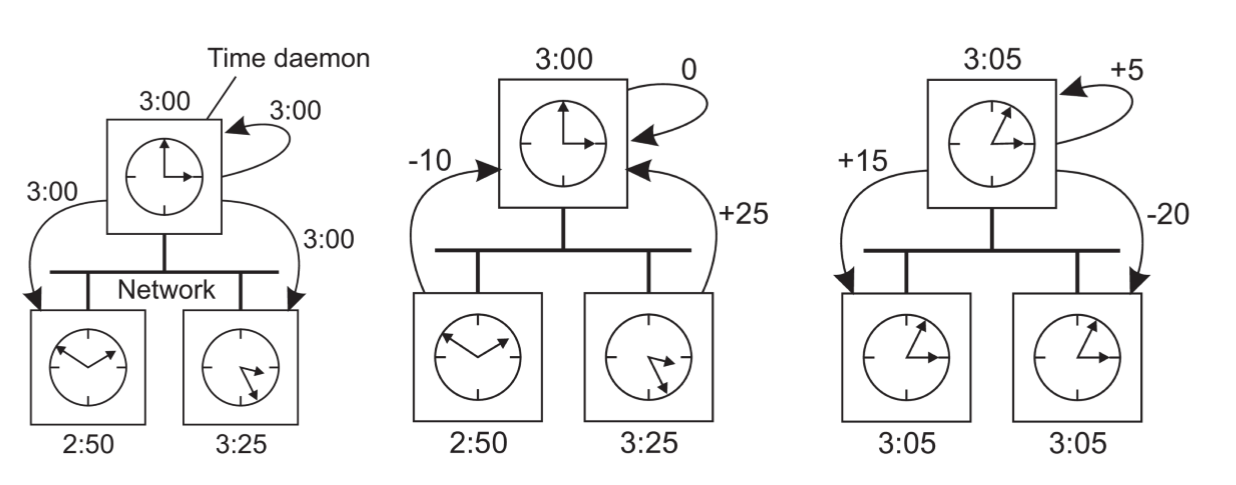
**Clock Synchronization**

**-Algoritmos-**

**Network Time Protocol:**

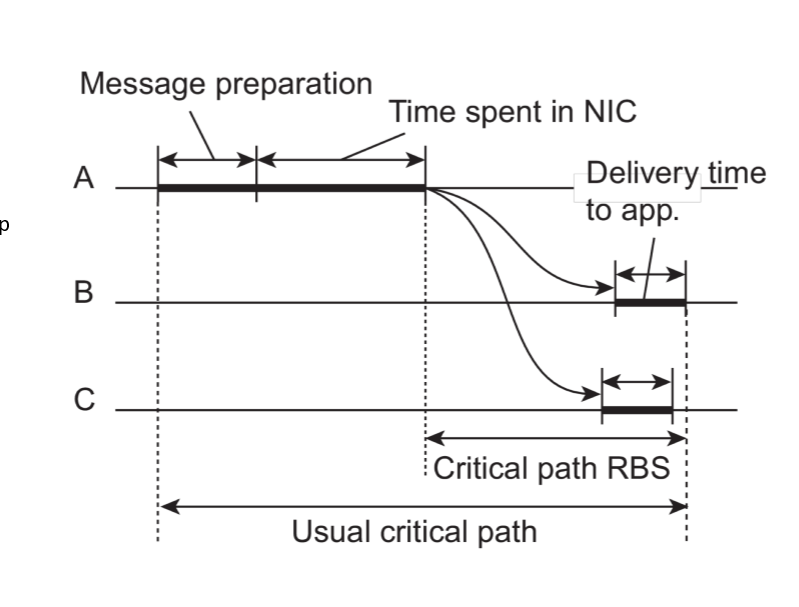
* É **simétrico** (i.e 2 servidores A e B, A pode procurar ajustar o seu time segundo B, ou B segundo A)
* **Ideia**: Procurar sincronizar o tempo segundo um outro servidor (preferencialmente um com UTC receiver), mas tendo em conta os delays que acompanham a troca de mensagens de sincronização.
* **Algoritmo:** Procuramos encontrar uma boa estimação do time delay e obter o offset de tempo entre 2 servers da seguinte forma:
  + **1) A** manda um request a B, timestamped com o valor T1.
  + **2)B** aponta o tempo que que recebeu o pedido, T2, e manda uma resposta a A timestamped com o tempo em que a envia, T3, e com o tempo de receção T2 em piggyback
  + **3)A** aponta o tempo em que recebeu a resposta de B, T4, e com a informação colecionada ate agora consegue calcular o offset:
    - **Offset = ((T2-T1) + (T3-T4))/2**
  + **4)A** vai ajustar o seu internal clock acelerando (caso esteja atras) ou desacelerando (case esteja a frente) a rotina de interrupts ate acertar o tempo (e.g, se antigamente 100 interrupts eram gerados por segundo, e adicionavam 10msec ao tempo, para desacelerar fazemos com que 100 interrupts/segundo gerem adicionem apenas 9msec)
* **Nota**: B também vai calcular o offset a A com a mesma formula, e vai também calcular um delay com a formula **delta = ((T4-T1)-(T3-T2))/2.** Normalmente são guardaos 8 pares (Offset,Delay) e depois escolhe-se o que tiver o valor mínimo do delay como melhor estimação do delay entre 2 servers (e o offset associado como a mais reliable estimation)
* **Nota++:**
  + Não faria sentido B sincronizar-se com A caso B fosse mais accurate (p.ex se tivesse um UTC Receiver).
  + Para evitar situações estupidas, os relógios estão classificados em **Strata**.
  + UTC Receiver clocks são considerados em Stratum-1.
  + Um relógio so se sincroniza por outro, caso esteja num Strata superior (p.ex A so se sincroniza com B se Strata A > Strata B).
  + No caso de A se sincronizar com B, A sobe para (Strata B – 1)

**Algoritmo de Berkley:**

* É **simétrico** (i.e 2 servidores A e B, A pode procurar ajustar o seu time segundo B, ou B segundo A)
* **Ideia**: Ao contrario do NTC, assume-se que não temos nenhum relogio particularmente Accurate, e queremos so manter Internal Synchronization. Para tal um time server, que ao contrario do NTC, é ATIVO, vai pedir o tempo a todas as maquinas e computar uma average. Depois diz a todas as maquinas se devem atrasar ou acelerar para ficarem sincronas
* **Algoritmo:** Os pedidos são tratados por time daemons e vao realizar as seguintes operacoes:
  + **1) A** pede a todos os outros clocks os seus tempos (incluindo a si próprio)
  + **2)A** calcula os time offsets utilizando uma average
  + **3)A** avisa todos os outros clocks se devem, e por quanto, se atrasar/adiantar (incluindo a si mesmo)
  + **4)** Relógios acertam-se e ficam portanto sincronizados entre eles

**Reference Broadcast Synchronization (RBS):**

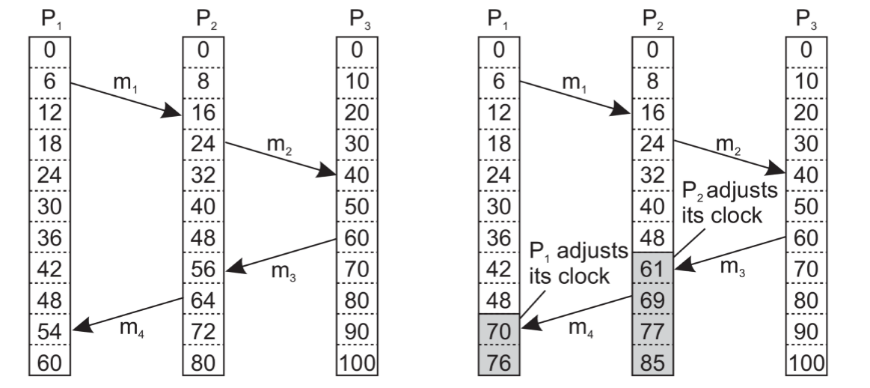
* **Não é simétrico**!
* Usado em wireless networks onde assumçoes como reliable communication entre machines e disseminação de informação simples não existe e onde conservação de energia e multihop routing caro são fatores a considerar
* **Ideia**:
  + Tal como no Berkley’s não se assume que temos um clock particularmente Accurate, procurando portanto sincronização interna.
  + Não é síncrono pois permite que os receivers se sincronizem mantendo o sender out of the loop.
  + Um sender basicamente envia em broadcast uma mensagem de referencia que permitirá aos receivers ajustar os seus relógios.
  + **O tempo de propagação a considerar aqui e medido a partir do momento em que a mensagem sai da network interface do sender!**
  + Quando um node broadcast uma reference message m, cada no p guarda o tempo Tp,m em que recebeu m (este tempo é lido do local clock do p).
  + 2 Nodes p e q conseguem trocar os seus delivery times para estimar o seu mutual relative offset, desta forma p saberá o valor do relogio de q relativo ao seu próprio
  + Devido ao drift porém, uma simples troca de medias não e a melhor forma de calcular o offset, devemos portanto usar uma regressão linear

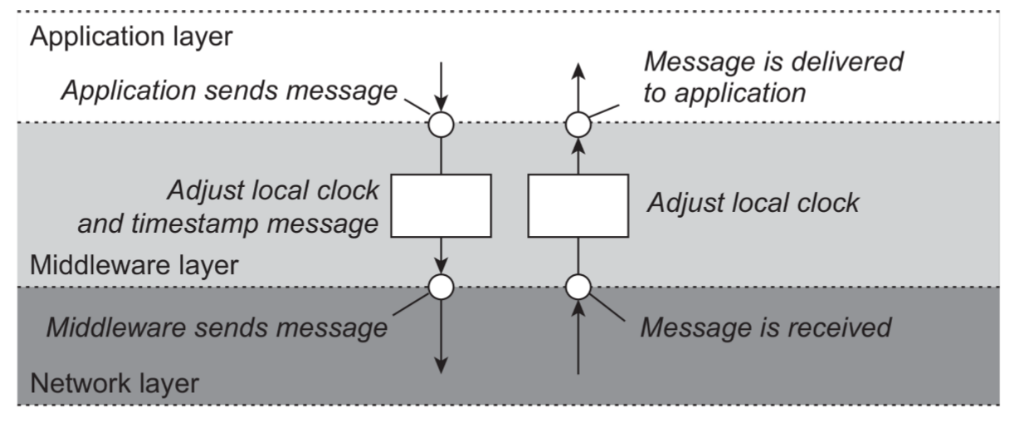


**Logical Clocks**

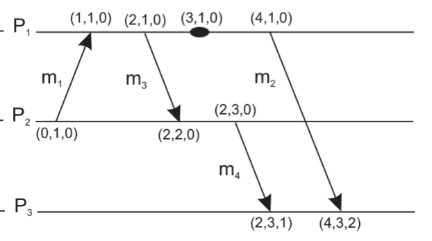
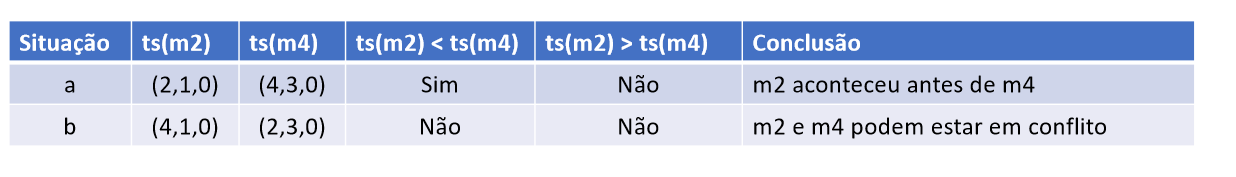
**“*Por vezes, não nos interessa que os clocks em si estejam sincronizados, queremos apenas que as operações sejam executadas ordenadamente*”**

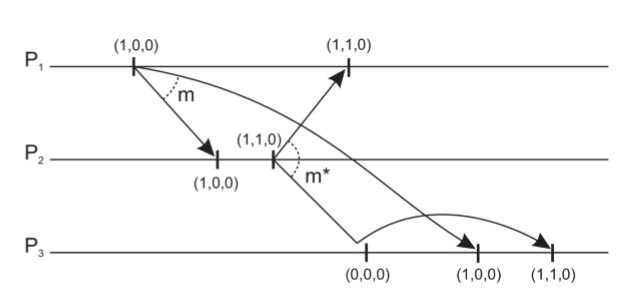
**Lamport’s Logical Clocks:**

* Baseia-se na ideia da relação **happens before:**
  + **A->B**
  + Lê-se: “A acontece antes de B”
  + Significa **que TODOS OS PROCESSOS** concordam que o evento A acontece antes do evento B
  + Observa-se diretamente em 2 situações:
    - Se a e b **são 2 eventos no mesmo processo**, e **a acontece antes de b**, então a->b==True
    - Se a é o evento correspondente ao **envio de uma mensagem num processo**, e b o evento correspondente a **receção dessa mensagem noutro processo**, então a->b==True
  + Eventos em que a->b == False e b->a == False dizem-se **Concorrentes**
* Queremos garantir que, **se a->b então C(a) < C(b)** – Onde C(x) corresponde ao time value em que ocorre x – Independentemente do processo em que a e b ocorrem
* **Algoritmo:**
  + **1)Antes de executar um evento**(i.e mandar uma mensagem)**,** processo **Pi** incrementa **Ci++**
  + **2)**Quando processo **Pi manda a mensagem**,m, **inclui** nela o **timestamp**, ts(m), com o seu **Ci** atual
  + **3)**Quando um processo **Pj recebe a mensagem, ajusta** o seu **Cj = max{Cj, ts(m)}**
  + **4)Pj** incrementa **Cj++**
  + **5)Pj** entrega a mensagem a sua aplicação.
* **Nota++**: Temos 3 camadas: Application, Middleware e Network: A camade de middleware é a que executa o algoritmo de lamport, e depois entrega a mensagem a application para fazer o que quiser com ela. A network layer e responsável pelo envio e receção das mensagens
* **Nota:** Cada processo, Pi, mantem local counter, Ci



**Vector Clocks:**

* Relógios de Lamport can’t actually capture, se, caso C(a) < C(b), a tenha efetivamente acontecido antes de b
  + Lamport’s Clocks: **C(a)<C(b) não implica a->b**
* Não é capturada **CAUSALIDADE**
* Podemos facilmente capturar causalidade ao associa a cada evento um nome único e usando um contador local em que pk e o kth event que aconteceu no processo P.
* O problema ai é keeping track das **Causal Histories** (e.g se p1 e p2 são 2 eventos que aconteceram sucessivamente no processo P, então o causal history de p2 – H(p2) = {p1,p2})
* Ao enviarmos uma mensagem (i.e evento p3 – envio de mensagem) de P ate Q, vamos ter a causal history de q1 – evento de receção da mensagem em Q – H(q1) = {p1,p2,p3,q1}
* Para verificarmos se um evento p causalmente precede um evento q precisamos apenas de verificar se H(p) está contido em H(q) – Neste exemplo, p2 causalmente precede q1, p.ex
* O problema com esta tática e que causal histories são merdosas em termos de representação eficiente. So fuck that we’re gonna use Vector clocks baby
* **Algoritmo:**
  + **1)** Antes de executar um evento (i.e mandar uma mensagem para outro processo ou entregar uma mensagem à application layer ou outro qualquer evento interno), Pi incrementa VCi[i]++ (equivale a dizer que aconteceu um novo evento em Pi)
  + **2)**Quando Pi envia uma mensagem m a Pj, inclui o seu VCi como timestamp na mensagem, ts(m) (apos ter claro incrementado VCi[i]++ visto ir realizar um evento de envio)
  + **3)**Quando Pj recebe uma mensagem, para todo o k, vai fazer VCj[k] = max{VCj[k], ts(m)[k]}
  + **4)** Pj incrementa VCj[j]++
  + **5)** And we go back to step 1
* **Nota++**: Para verificar se um evento causalmente precede outro temos apenas que comparar o ts(m1)
* **Nota:** Cada processo, Pi, tem um vector clock VCi com um length = numero de processos e cujo valor de VCi[i] corresponde ao numero de eventos que aconteceram em i a cada momento.
* **Nota++:** também que o incremento é feito, não quando se recebe a mensagem per se, mas quando ela é entregue à application layer
* **Nota++: Se um evento tem um timestamp ts(a), então ts(a)[i]-1 == Numero de eventos processados em Pi que causalmente precedem a**
* **Enforcing Causal Communication**
  + Podemos, usando vector clocks, garantir que uma mensagem so é entregue se a que a mensagem que lhe causalmente precede também tiver sido recebida
  + Causal Ordered Multicasting (kind of)
  + Assumindo que os clocks são ajustados apenas ao enviar e receber mensagens, então, quando Pj recebe uma mensagem, a entrega desta à application layer sera delayed ate que:
    - 1) ts(m)[i] = VCj[i] + 1
    - 2) ts(m)[k] <= VCj[k] para todo k!=i
  + **A primeira condição diz que m tem de ser a próxima mensagem que Pj estava a espera de receber de Pi**
  + **A segunda condição garante que Pj entregou todas as mensagens anterio** **res que foram entregues por Pi quando mandou a mensagem m**



**Mutual Exclusion**

**-Considerações-**

Por vezes, para garantir que acessos concorrentes por vários processos ao mesmo recurso não o corrompam, temos que garantir que o acesso é feito por exclusão mutua (i.e apenas um de cada vez)

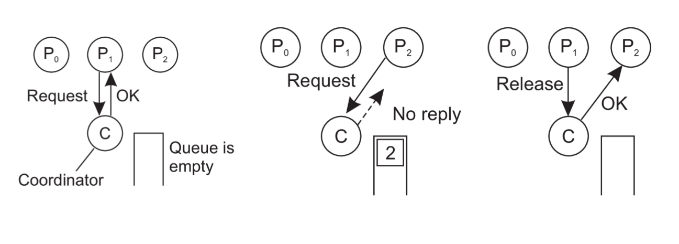
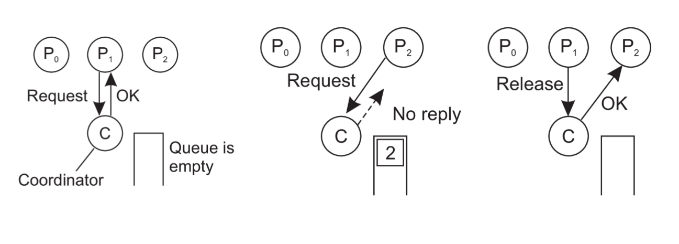
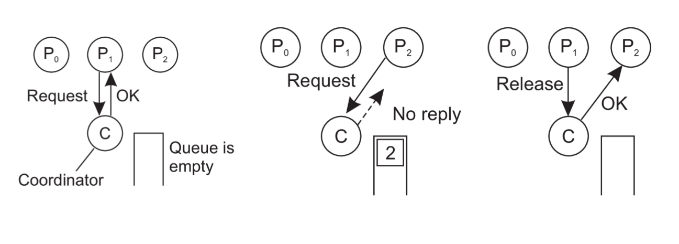
Temos 2 tipos de algoritmos para o fazer:

* Token Based Solutions
  + Existe um token que é passado pelos processos.
  + Apenas o processo que tem o token tem acesso ao recurso partilhado
  + Quando o acesso e terminado, o token e passado ao próximo processo
  + Caso o processo não queira acesso, passa simplesmente o token
  + Previnem Starvation e Deadlocks!
  + Mas…Se o token se perder temos problemas porque precisamos de gerar um novo and thats big oof
* Permission Based Approach
  + Procesos que quer aceder a um recurso pede permissão aos outros recursos primeiro

**Mutual Exclusion**

**-Permission Based Algorithms-**

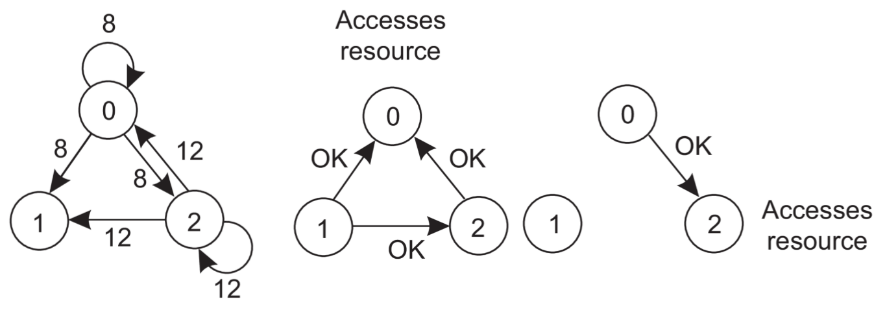
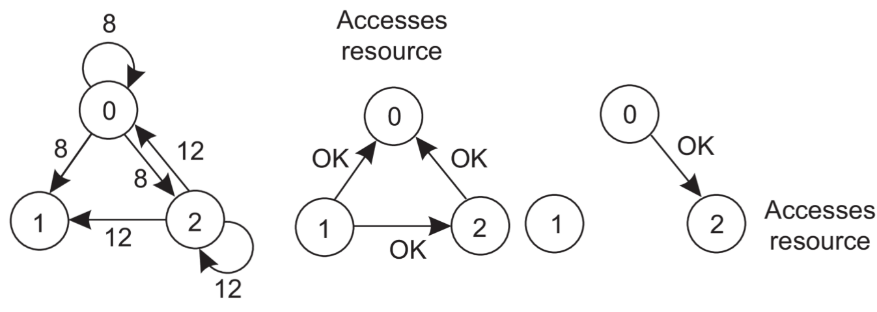
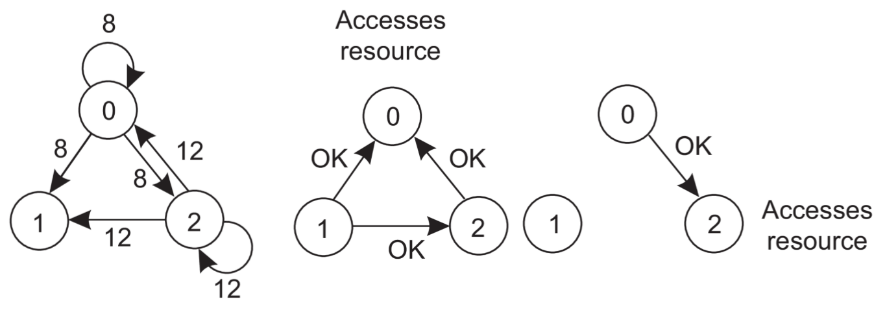
**Centralized Algorithm:**

* **Algoritmo**:
  + Basicamente, temos um processo que é elegido como **Coordinator**
  + Quando um processo quer aceder a um recurso partilhado, pede permissão ao coordinator
  + Se mais nenhum processo estiver a aceder ao recurso, o coordinator envia uma permissão, else pode ou não responder ou mandar um deny e adiciona-o a uma **queue** de processos que querem acesso
  + Quando um processo que esteja a usar o recurso tiver terminado, avisa o coordinator para que este possa dar permissão a outro processo que tenha na queue de requests
* +Evitamos **Starvation** (visto que o algoritmo é justo) e **Deadlocks**
* **+Facil de Implementar**
* **-Coordinator e um single point of failure**
* **-Dificil distinguir coordinator morto de um deny (o que pode levar a bloqueios de processos)**
* **-Coordinator pode tornar-se num bottleneck**
* **Still..solucoes distribuídas não são necessariamente melhores… sad**

**Decentralized Algorithm:**

* **Assume-se que cada resource está replicado N vezes e que cada replica tem um coordenador que controlo o acesso concorrente por processos!**
* Quando um processo quer aceder a um recurso, pede aos coordenadores, e tem de receber uma maioria de votos de aceitação m>N/2 (i.e pelo menos metade dos coordenadores deixam que ele aceda)
* Quando um coordenador não der permissão de acesso ao recurso (i.e já deu a um outro) vai dizer ao requester
* Assume-se que quando um coordenador crasha recupera rapidamente mas vai se ter esquecido do voto dado antes de crashar
* Corremos o risco de um coordenador dar incorretamente permissão de acesso a vários processos (apos ter recuperado) mas…
* Em geral a probabilidade de acontecer merda mesmo apos o coordenador crashar, é tao pequena que podemos ignora-la
* Um processo que seja denied assume que passado algum tempo poderá ter acesso, logo voltara a pedir mais tarde
* **-Se muitos nodes tentarem aceder ao mesmo recurso a utilização baixa rapidamente pois existem tantos recursos a competir peloa cesso que eventualmente, nenhum sera dado permissão pela maioria dos coordinators :(**

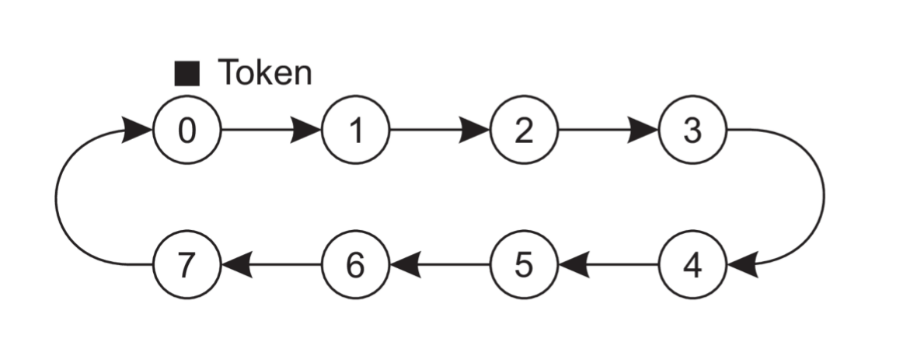
**Decentralized Algorithm:**

* Baseado no *total ordering of all events in the system* (i.e, para todos os pares de eventos, nao Podemos ter duvidas sobre qual aconteceu primeiro)
* **Algoritmo**:
  + 1) Processo que quer aceder ao recurso constrói uma mensagem (que contem o nome do recurso), o seu process number/name e o seu **current logical time**
  + 2) Essa mensagem é enviada a todos os outros processos que também acedem ao recurso, e assume-se que o envio da mensagem é reliable
  + 3) Quando um processo recebe um pedido de acesso ele tem 3 opcoes:
    - **Não está a aceder ao recurso**, **nem quer acesso** – Envia uma resposta de OK
    - **Já esta a aceder ao recurso** – Não responde mas regista numa queue o processo que mandou a mensagem
    - **Não está a aceder ao recurso MAS quer acesso –** Compara o timestamp da incoming message com o contido na mensagem que enviou a todos os outros. Se a mensagem que recebeu tiver um timestamp menor, o receiver manda um Ok ; Else, o receiver guarda numa queue o pedido e não manda nada
  + 4)Depois de enviar os pedidos, um processo espera ate receber um Ok de todos os outros
  + 5) Quando um processo termina o acesso ao recurso, envia um Ok a todos os processos que tenha na queue
* **-N points of failure (precisamos de um Ok de todos os processos…)**
* **-Temos de usar multicast communcation ou cada processo tem de manter uma group membership list**
* **-Todos os processos estão envolvidos em Todas as decisões de acesso…xtra burdon :(**
* **Podemos modificar o algoritmo para o tornar melhor, p.ex fazer com que o acesso seja permitido apos se ter recebido um Ok da maioria dos processos**

**Mutual Exclusion**

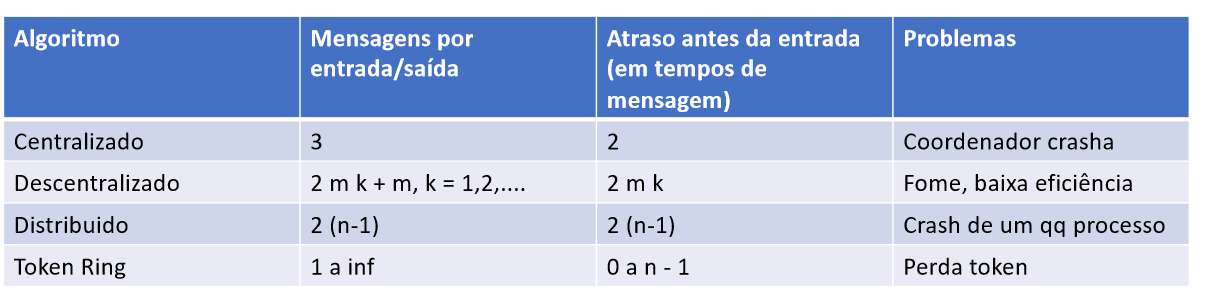
**-Token Based Algorithms-**

**Token-Ring:**

* **Os processos estão organizados numa overlay network com a forma de um logical ring, na qual cada processo é dado uma posição no ring!**
* Cada processo conheço qual o processo que lhe sucede no ring
* Quando o ring é iniciado, é dado um Token a um processo arbitrário (p.ex P0)
* O token circula pelo ring sendo passado de node em node
* Quando um processo recebe o token, verifica se quer aceder ao shared resource.
  + Se sim: Acede ao recurso e depois liberta o token, passando-o ao seu sucessor
  + Se não: Passa o token para o seu sucessor
* +Evitamos **Starvation** (visto que o algoritmo é justo) e **Deadlocks**
* **+Pior dos casos**, um processo tem de esperar que o token passe por todos os outros processos antes de poder aceder ao recurso
* -Se perdermos o token temos que criar um novo (pode acontecer caso um processo com o token crashe)
* -Detetar a perda do token não é fácil pois nunca temos certeza se o token se perdeu ou se um processo esta a demorar muito tempo a usa-lo
* -Garantir que um processo recebeu um token implica que este envie uma mensagem de receive. Caso seja detetado que o processo morre, para reconstruir o rign e preciso que todos os processos mantenham a configuração atual do ring guardada…
* **Nota:** Cada processo, ao adquirir o token, pode aceder ao recurso partilhado apenas uma vez (não pode entrar, sair e voltar a entrar logo, pois tem que passar o token para a frente)

**Mutual Exclusion**

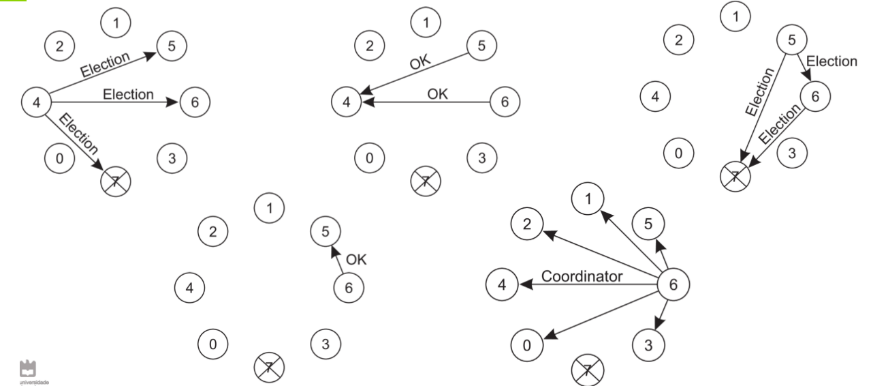
**-Comparação de algoritmos-**



**Election Algorithms**

**-Algorithms-**

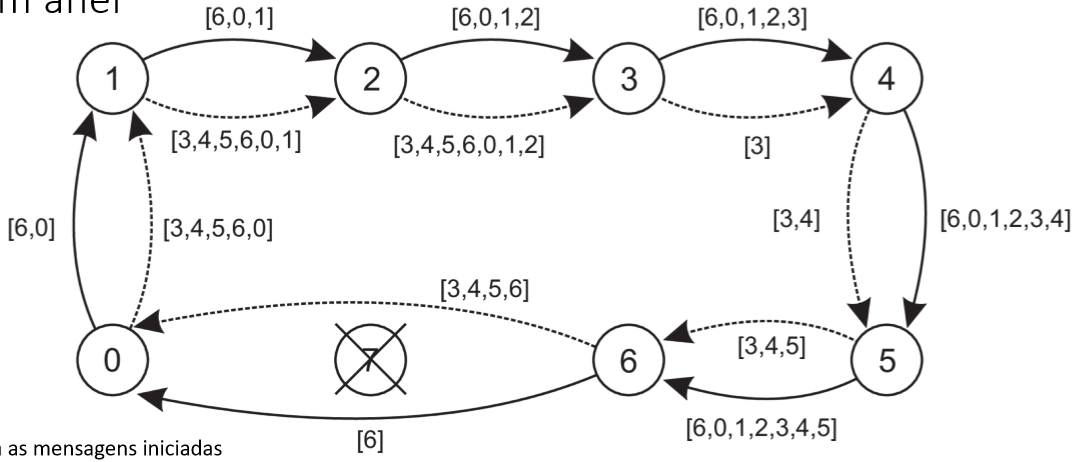
**Bully Algorithm:**



* Consideram-se N processos, P0 ate Pn-1, cada processo com um id(Pi) = i. (onde i é um numero)
* **Algoritmo**:
  + 1) Processo Pi repara que o coordinator/leader não está a responder a pedidos, logo vai iniciar a eleição
  + 2) Pi vai enviar uma ELECTION message a todos os processos cujos identifiers sejam maiores do que o seu (Nota: também poderia mandar aos que fossem menores)
  + 4) Se ninguém responder, Pi ganha e torna-se o coordinator enviando uma mensagem a avisar todos os outros
  + 5) Se alguém responder, repete o algoritmo começando no passo 1)
* A qualquer momento, um processo tem de estar pronto para receber uma ELECTION message e responder com um Ok (caso tenham um id maior do que o sender), passando assim a realizar uma eleição (CASO AINDA NÃO ESTEJA A FAZER UMA)
* Eventualmente todos os processos desistem exceto um
* Se o coordenador que morreu recomeçar, avisa todos os outros que se tornara de novo o coordenador

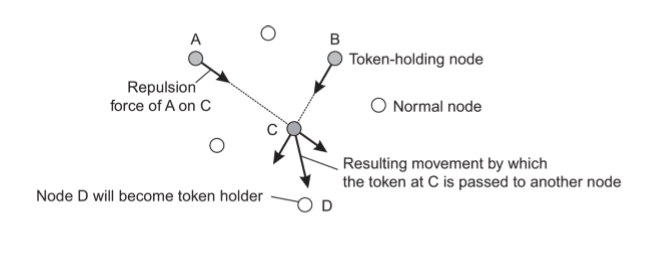
**Ring Algorithm:**

* Baseado no uso de um ring logico mas sem uso de token.
* Assume-se que cada processo conhece o seu sucessor
* **Algoritmo**:
  + 1) Processo Pi repara que o coordinator/leader não está a responder a pedidos, logo vai iniciar a eleicao
  + 2) Pi vai enviar uma ELECTION message ao seu sucessor contendo o seu próprio id (caso o sucessor esteja morto, o sender salta esse manda para o próximo)
  + 4) Processo Pj recebe a mensagem, da append do seu próprio id à lista contida na mensagem e repete o paço 2)
  + 5) Quando a mensagem voltar ao no que a iniciou (i.e quando um no reparar que o seu id já esta na lista), este ve qual e o maior id na lista, e escolhe esse como coordinator
  + 6) Manda mensagem COORDINATOR a circular pelo no, para que todos registem quem e o novo coordinator



**Election in large-scale systems:**

* Temos situações em que VARIOS nodes devem ser selecionados como especiais (p.ex no caso dos **Super Peers** em P2P Networks)
* Requirements para seleção de super peer:
  + Normal nodes devem ter pouca latência de acesso aos super peers
  + Super peers devem estar distribuídos uniformemente
  + Deve haver uma porcao predefenida de super peers relativa ao total numero de nodes na overlay network
  + Cada super peer não deve servir mais do que um fixo numero de nodes
* **Algoritmo (baseado no positioning de nodes em m-dimensional geometric spaces – Queremos colocar N super peers evenly pela overlay)**:
  + 1) São espalhados N tokens por N randomly chosen nodes na overlay network (um node pode ter no máximo 1 token)
  + 2) Cada token representa uma repelling force que inclina outros tokens a se afastarem (requer que nodes que tenham tokens saibam que outros nos também tem tokens)
  + 4) Quando um node tiver um token durante x tempo, promove-se a super peer



**Location Systems**

**-Actual Positioning-**

Num sistema distribuído de larga escala, em que os nós se encontram dispersos numa WAN é comum o sistema ter em atenção a noção de proximidade e/ou distância

Para tal é necessário determinar a localização do nó.

**Global Positioning System - GPS:**

* Usa vários satélites que circulame em orbita
* Cada satélite tem 4 clocks atómicos regularmente calibrados
* Cada satélite continuamente faz broadcast da sua posição e time stamps cada mensagem com o seu local time
* Com isto qualquer receiver na terra consegue facilmente computar a sua posição usando, em principio, apenas 4 satelites
* **Um nó precisa de d+1 marcos de referencia (satélites) para calcular a sua posição num espaço d-dimensional**

**WiFi Access Points:**

* Usa-se quando não podemos usar o GPS (p.ex em ambientes fechados)
* Se tivermos acesso a uma base de dados de conhecidos pontos de acesso (e as suas coordenadas), podemos estimar a nossa distancia a um ponto de acesso e, fazendo isto com apenas 3 pontos, conseguimos computar a nossa posição
* Problemas ocorrem em determinar as coordenadas de um ponto de acesso (para isso usa-se War Driving)
* Memso assim so conseguimos estimações das coordenadas, portanto a accuracy vai sofrer

**Location Systems**

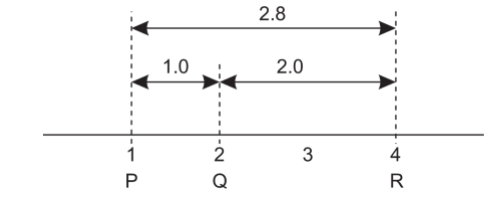
**-Logical Positioning-**

Podemos, em vez de encontrar uma localização absoluta de um node, tentar capturar apenas uma logical, prxoimity based location.

Distancia, nestes casos, pode p.ex corresponder a latência entre 2 nodes

Alguns usos uteis incluem encontrar optimal replica placement

**Centralized Positioning:**

* Como já vimos, para posicionar um node num m-dimensional geometric space precisamos de m+1 marcos de distancia a nodes com posições já conhecidas
* Com apenas 1 teriamos um circulo no qual o nosso node se encontra, com 2 teriamos a interseçao de 2 circulos que nos daria 2 pontos possíveis, e com 3 temos então um único ponto correspondente a localização atual
* O problema com usar latência como distancia é que não tende a ser estável, o que leva a diferentes posicionamentos de P sempre que se calcula a sua posição.
* Por outro lado, nodes que usem P como marco vao ter um erro ainda maior caso este contenha erros
* *Measured distances between a set of nodes will generally not be consistent*