UNICAMP

INSTITUTO DE COMPUTAÇÃO

RELATÓRIO TÉCNICO

RESUMO DO CONTEÚDO ABORDADO NA EMENTA DA DISCIPLINA: "COMPUTAÇÃO DISTRIBUÍDA – MO441"

2° SEMESTRE/2010

DANIEL CINTRA CUGLER
RA 109231

Este relatório está disponível para download em: http://www.ic.unicamp.br/~dcintra

Este documento foi criado com base no conteúdo exibido na disciplina MO441 (Computação Distribuída), oferecida no departamento de Ciência da Computação da Universidade Estadual de Campinas (UNICAMP), no segundo semestre de 2010.

É importante ressaltar que este documento não passou por nenhum processo externo de revisão. O intuito deste documento é compartilhar parte do conhecimento que adquiri durante o período em que assisti a disciplina MO441.

Daniel Cintra Cugler

SUMÁRIO

RESUMO DA COMPLEXIDADE DOS ALGORITMOS CITADOS NESTE DOCUMENTO

Modelo Síncrono

→ Eleição de líder em rede de anel síncrono

Algoritmos baseados em comparação

- LCR
  - Complexidade de mensagens: $O(n^2)$
  - Complexidade de tempo:O(n)
- HS
  - Complexidade de mensagens: O(n log n)
  - Complexidade de tempo: 3n se n é potência de 2 / caso contrário 5n

Algoritmos não baseados em comparação

- TimeSlice
  - Complexidade de mensagens: n
  - Complexidade de tempo: $n * UID_{min}$
- VariableSpeeds
  - Complexidade de mensagens: o(2n)
  - Complexidade de tempo: $(n*2^{UIDmin})$

→Algoritmos em redes gerais síncronas

- Inundação - Flooding (*Eleição de líder*)
  - Complexidade de mensagens: diâmetro * |E|
  - Complexidade de tempo: diâmetro
  - ◦ Otimização (*economia de mensagens*)
    - Complexidade de mensagens: *menor que* (diâmetro * |E|)
    - Complexidade de tempo: diâmetro
- Busca em largura – SyncBFS
  - Complexidade de mensagens: |E|
  - Complexidade de tempo: O(diâmetro)
  - ◦ Otimização (*vértices conhecem os seus filhos*)
    - Complexidade de mensagens: O(|E|) se bidirecional – |E|*b bits para cada msg
    - Complexidade de mensagens: O(diâmetro*|E|) se unidirecional - |E|*b bits para cada msg
    - Complexidade de tempo: diâmetro
- Caminhos mínimos – Bellman Ford
  - Complexidade de mensagens: (n-1)*|E|
  - Complexidade de tempo: n-1
- Árvore geradora mínima (Minimum Spanning Tree) – SynchGHS
  - Complexidade de mensagens: O((n + |E|) * log n)
  - Complexidade de tempo: O(n log n)

→Falhas de processo

Consenso com falhas

- Falha e pára:
  - Algoritmo FloodSet
    Complexidade de mensagens: $O((f+1) * n^2)$
  - OptFloodSet
    Complexidade de mensagens: $O(2n^2)$
- Falha bizantina
  - $n > 3f$

Modelo Assíncrono

→Exclusão Mútua

- Algoritmo Lamport 78 (Baseado em tempo lógico)
  Complexidade de mensagens: 3(n-1)
- Algoritmo Ricart & Agrawala 81 (Baseado em tempo lógico)
  Complexidade de mensagens: 2(n-1)
- Algoritmo Carvalho & Roucairol (NÃO é baseado em tempo lógico global)
  Complexidade de mensagens: 0 a 2(n-1)
- Algoritmo Ricart & Agrawala 83 (NÃO é baseado em tempo lógico global)
  Complexidade de mensagens: n →(n-1) + 1 mensagem do token = n
- Algoritmo Maekawa (NÃO é baseado em tempo lógico global)
  Complexidade de mensagens melhor caso: √request, √lock, √release
  Complexidade de mensagens pior caso: √request, √enquire, √relinquish, √lock, √release, √lock
- Algoritmo Agrawal & Abadi (NÃO é baseado em tempo lógico global)
  Complexidade de mensagens: O(log n)

→Terminação

- Algoritmo DijkstraScholten
  Complexidade de mensagens: 2m
  Complexidade de tempo: $O(m(t_{execução} + t_{entrega\_mensagem}))$

→Snapshot consistente global

- Algoritmo LogicalTimeSnapshot.
  Complexidade de mensagens: 0 → nenhuma mensagem adicional
  Complexidade de tempo: infinito
- Algoritmo ChandyLamport
  Complexidade de mensagens: $2|E| = O(|E|)$
  Complexidade de tempo: $O(diâmetro(t_{execução} + t_{entrega\_mensagem}))$

→Detecção de Deadlock

Algoritmo Chandy, Misra, Haas

- Resource Model
  Complexidade de mensagens: $O(|E|)$ *do grafo de dependência*
  Complexidade de tempo: $O(diâmetro(t_{execução} + t_{entrega\_mensagem}))$ *do grafo da rede*
- Resource Model
  Complexidade de mensagens: $2|E| = O(|E|)$ *do grafo de dependência*
  Complexidade de tempo: $O(diâmetro(t_{execução} + t_{entrega\_mensagem}))$ *do grafo da rede*

1. Computação distribuída – Alguns conceitos

(Conteúdo deste capítulo pode ser encontrado no capítulo 1 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

- **Computação distribuída:** vários computadores conectados por uma rede. Eles podem estar fisicamente distantes. Cada um possui seu próprio processador e memória.
- **Computação paralela:** uma mesma memória é compartilhada para vários processadores que são empilhados em um hardware de grande porte. Ex: Mainframe
- **Recurso preemptível**: Podem ser retirados do processo proprietário sem nenhum prejuízo. Ex: CPU, memória.
- **Recurso NÃO preemptível**: NÃO pode ser retirado do processo prioritário sem que a computação apresente falha. Ex: Uma unidade gravadora de DVD não pode estar disponível para outro processo enquanto grava um DVD, senão vai gerar um DVD defeituoso.
- **Deadlock**: situação onde existe um ciclo de espera entre os processos. Nesta situação os processos ficam BLOQUEADOS.
- **Livelock:** situação onde os processos se comportam de maneira que impede que o estado do sistema avance. Nesta situação os processos NÃO ficam bloqueados. Ex: duas pessoas em um corredor estreito estão andando em sentidos opostos e de frente. Em determinado momento um quer deixar o outro passar e se movem sempre para o mesmo lado em mesma velocidade. Note que os processos estão trabalhando, mas não evoluem.
- **Justiça**: definição para situação onde todo processo deve ter a oportunidade de executar sua tarefa. A falta de justiça causa STARVATION. Graus de justiça:
    - Todos têm a mesma oportunidade
    - Processo em algum momento indefinido, mesmo que em uma oportunidade menor, terá sua vez de executar.
- **Starvation:** O processo morre de inanição. Ex: o processo tem prioridade baixa e o recurso acaba nunca sendo disponibilizado para o processo. Ex2: Se ao chegar em uma ponte há um carro já atravessando na sua frente e no mesmo sentido você passa logo em seguida e os outros atrás de você também. Se tiver algum carro no outro sentido, pode ser que a vez dele nunca chegue.
- **Grafo fortemente conexo:** grafo direcionado onde há sempre um caminho entre dois vértices.
- **Grafo completo:** todos os vértices estão interligados com todos os outros vértices através de uma aresta. Isto é, o diâmetro de uma grafo completo é 1.

1.1. Técnicas para solução de problemas em computação distribuída

- **Exclusão mútua**: técnica que evita que um recurso compartilhado seja utilizado por mais de um processo por vez. Utiliza-se semáforo binário para dizer que um recurso compartilhado está disponível ou não. Esta técnica pode causar deadlocks!

- **Sincronização de relógios**: sincronizando os relógios de cada nodo da rede, é possível ordenar os eventos em um sistema. A sincronização de relógios permite resolver o problema de **exclusão mútua**. Cada participante mantêm sua própria fila de processos que aguardam um recurso. Esta fila é ordenada pelo tempo que foi gerado o pedido, conforme exibido na figura a seguir:

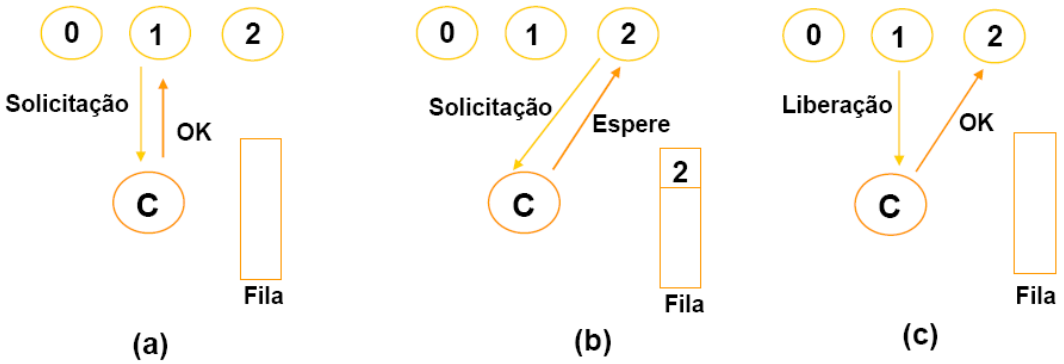


Figura 1- Sincronização de relógios (Figura extraída da internet - fonte desconhecida)

Mas, e se o pedido chegar atrasado?

- **Consenso distribuído** *(problema dos generais bizantinos)***:** consiste em um acordo entre processos para a escolha de um valor v dentro de um conjunto V.

- **Disseminação confiável *(broadcast confiável)*:** uma mensagem é recebida por todos os participantes ou nenhum. As mensagens são recebidas por todos os participantes na mesma ordem. Pode ser utilizada para resolver o problema de exclusão mútua: cada participante faz broadcast de pedido. Cada participante mantêm uma fila contendo os pedidos por ordem de chegada. O participante que está no início da fila pode usar a exclusão mútua. Assim que terminar de usar o recurso o participante faz broadcast para avisar que terminou de usar.

- **Replicação de dados:** consiste em manter múltiplas cópias da mesma informação em dispositivos distintos. Com isso é possível aumentar a eficácia, tolerância a falhas e o desempenho dos serviços distribuídos, uma vez que um dado pode ser coletado em um nodo próximo ao participante.

- **Eleição de líder:** entre um conjunto de processos candidatos, um processo deve se pronunciar líder para uma determinada tarefa. É executado quando o sistema distribuído está sendo iniciado ou quando o líder anterior não consegue se comunicar com os demais processos pela ocorrência de alguma falha. Existem vários algoritmos que realizam a eleição de líder.

1.2. Elementos básicos

- Processos
- Rede de comunicação: Existem vários tipos de topologias:
    - **Anel**
    - **Estrela**
    - **Árvore**
    - **Totalmente conexa** (quando todos os vértices do grafo estão interligados)

1.3. Algoritmos distribuídos

Algoritmos utilizados para comunicação em ambientes distribuídos. Algumas características são levadas em conta no seu desenvolvimento:

1 - Método de comunicação entre processos (Ex: IPC)

2 - Modelo de tempo

3 - Modelo de falhas

4 - Problema a ser resolvido

1 - Método de comunicação entre processos – IPC (Inter Process Communication)

Grupo de técnicas que permite processos transferirem informações entre si.

1. Memória compartilhada: o IPC permite que diferentes processos troquem informações entre si através de memória compartilhada.

2. Mensagem ponto a ponto: operações para enviar mensagens de um determinado processo a outro. Pode ser dividido em:

    - Mensagem síncrona: o processo que envia a mensagem não pode prosseguir até o outro processo receba a mensagem.
    - Mensagem assíncrona: o processo que envia a mensagem não precisa esperar o outro processo receber para prosseguir.
    - RPC – Remote Procedure Call: é um tipo de comunicação entre processos que permite a chamada de um procedimento em outro nodo da rede. É utilizada na implementação do modelo cliente-servidor.

2 - Modelo de tempo

- **Síncrono:** componentes progridem de modo simultâneo, em *rounds* síncronos.
- **Assíncrono:** componentes progridem em passos independentes, em ordem e velocidade arbitrárias.

- **Parcialmente Síncrono:** este modelo não é tão rígido quanto o modelo completamente síncrono. São assumidas algumas restrições na ordem dos eventos. Este modelo é utilizado para evitar alguns problemas de consenso.

3 - Modelo de falhas

- ***Erro:*** É a causa de falha. Um erro pode gerar uma falha, ou simplesmente passar despercebido.
- ***Falha:*** é o desvio de comportamento esperado de um componente da rede. *(fault-tolerance system: sistema com tolerança a falhas – mecanismos permitem a detecção e tratamento de falhas).*
    - ***Fase de desenvolvimento:*** prevenção e eliminação de falhas.
    - ***Fase de operação:*** tolerância a falhas. Essas falhas podem ser classificadas em tipos:
    - **Tipos de falhas (de canal):**
        - **Falha e pára (*fail-stop*):** componente pára totalmente assim que ocorre uma falha.
        - **Omissão:** o canal não entrega parte das mensagens.
        - **Performance:** o canal não respeita os limites de tempo de transmissão das mensagens. As mensagens chegam antes ou depois do esperado.
        - **Maliciosa:** tudo pode acontecer.

4 - Problema a ser resolvido:

Conhecer o tipo que deseja resolver para selecionar corretamente os modelos necessários.

1.4. Métricas para análise de algoritmos distribuídos

- **Tempo para completar execução**
- **Número de bits transmitidos**
    - Número de bits
    - Tamanho das mensagens
- **Espaço de memória necessário em cada nó**
- **Justiça**

2. Algoritmos em modelo síncrono

(Conteúdo deste capítulo pode ser encontrado no capítulo 3 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

2.1. Eleição do líder em uma rede de anel síncrona

Neste problema temos uma rede de n processadores conectados em um anel. Cada nodo da rede consegue distinguir vizinho esquerdo do vizinho direito. O objetivo é eleger um dos n computadores como sendo líder e fazer com que os outros nodos saibam disso.

Algoritmos vistos neste capítulo:

Algoritmos baseados em comparação

1. **LCR:** (Le Lann, Chang, Roberts)
2. **HS:** (Hirschberg e Sinclair)

Algoritmos *NÃO* baseados em comparação

3. **TS:** (Time Slice)
4. **VS:** (Variable speeds)

2.1.1. Algoritmo LCR (Le Lann, Chang, Roberts)

Nesta abordagem, o anel segue um ciclo orientado unidirecional, isto é, as mensagens percorrem uma mesma direção (Ex: sentido horário, por exemplo).

Complexidade de comunicação: $O(n^2)$ - Complexidade de tempo: n rounds

O algoritmo começa em todos os nós da rede. Note que não é necessário que se saiba o tamanho da rede. O objetivo é eleger líder o processo que possui o menor id (Ou o maior, não importa. No livro adotado pelo professor é selecionado o processo com Id maior).

Inicialmente:

- Cada processo possui status={líder, não líder ou desconhecido}

No início cada computador envia o seu Id para o próximo computador do anel. Nos próximos ciclos cada processador executa as seguintes tarefas:

1 – Recebe um Id do processo predecessor no anel;

2 – Compara o Id recebido com o maior Id que o processo conhece (inicialmente ele conhece somente o próprio ID);

3 – Se o ID recebido for maior que o maior ID já visto pelo processo, então ele o envia para o próximo computador do anel, caso contrário, ele não envia mensagem alguma, pois não há necessidade, uma vez que o processo já encaminhou anteriormente um ID maior do que o ID atualmente recebido. Esse detalhe permite economizar mensagens.

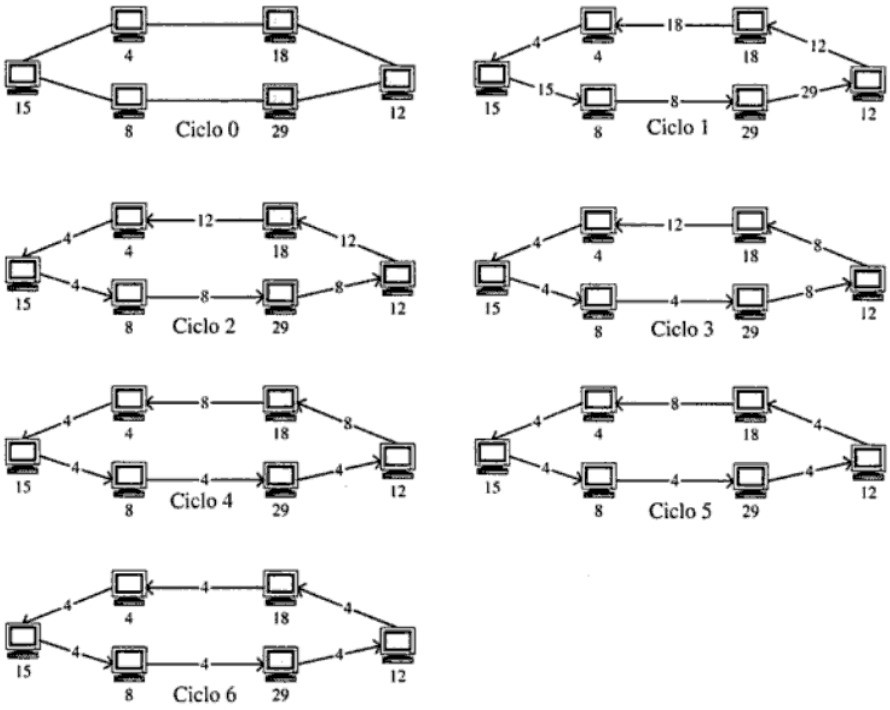


Figura 2 - Algoritmo LCR (Figura extraída da internet - fonte desconhecida)

Versão com Halting (parada)

O algoritmo LCR não sabe a hora de parar. O líder se elege líder quando recebe uma mensagem, mas os outros nodos não sabem disso e continuam enviando mensagens. Para resolver isso, existe outra versão do algoritmo que pára.

Quando um computador receber seu próprio Id é porque ele é o Líder. Então, a próxima etapa, consiste em esse computador enviar uma mensagem para todos os nodos da rede para avisar que ele é o líder.

- Complexidade de mensagens para encontrar o líder: $O(n^2)$
- Complexidade de tempo: 2n rounds
- Complexidade para avisar ao anel quem é o líder → Nesta segunda etapa, o nó que recebe a mensagem com o próprio Id vai enviar uma mensagem para o próximo nodo dizendo que ele é o líder e pára de enviar mensagens. O próximo nodo, ainda está enviando uma mensagem e vai enviar mais essa que recebeu, logo, o próximo nodo vai enviar 2 mensagens. Seu sucessor vai enviar 2 mensagens mais a mensagem avisando quem é o líder, logo, 3 mensagens, e assim por diante. Então, a complexidade para esta segunda fase, onde é enviada uma mensagem dizendo quem é o líder é $O(n^2)$.

2.1.2. Algoritmo HS (Hirschberg e Sinclair)

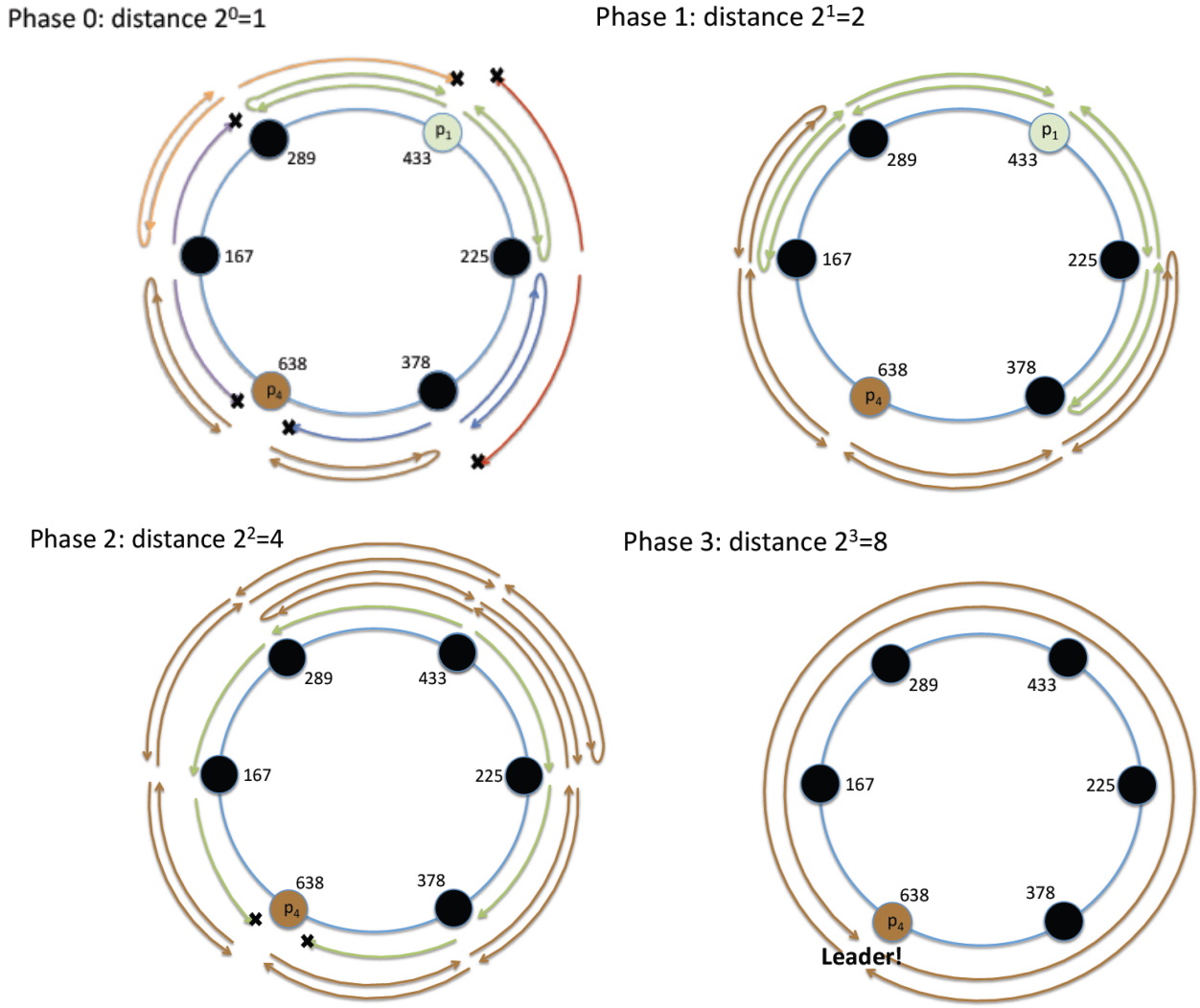
Nesta abordagem, o anel segue um ciclo bidirecional, isto é, as mensagens partem simultaneamente de um nodo e percorrem direções contrárias.

Complexidade de comunicação (mensagens enviadas): O(n log n)

Complexidade de tempo: 3n se n é potência de 2 / caso contrário 5n (Em outras palavras, O(n))

O algoritmo começa em todos os nós da rede e é executado em fases. Para cada fase, um nó envia simultaneamente uma mensagem para cada lado do anel. Cada mensagem vai percorrer a distância 2^{fase} (1,2,4,8,16...).

Cada nodo que enviou a mensagem vai enviar seu Id. Cada nodo que recebe, vai comparar o Id recebido com o seu próprio Id. Se o Id recebido for maior, a mensagem é encaminhada até a distância máxima 2^{fase} (e depois volta), caso contrário, se o Id recebido for menor, então o processo não faz nada, simplesmente pára. Se em determinada fase o nodo que enviou a mensagem não receber ela de volta de um dos lados sequer, então ele não é o maior, e ele cai fora das próximas fases. O algoritmo pára quando um nodo recebe seu próprio Id, isto é, quando o percurso de uma mensagem dá a volta no anel. Vejamos as figuras:



(Figuras extraídas da internet - fonte desconhecida)

2.1.3. Algoritmo TS (Time Slice)

- O tamanho n da rede é conhecido por todos os nodos

- Comunicação unidirecional é suficiente

- A computação ocorre em fases v. Cada fase consiste de n rounds consecutivos.

- Na fase v somente o nodo com Id=v pode enviar mensagem, então, se houver alguma fase v onde não há um nodo com Id=v, nenhuma mensagem é enviada, apenas os n rounds são esperados como sincronia. Na próxima fase todo processo se repete.

- Se um processo i tiver um Id=v, então ele envia uma mensagem ao longo do anel informando que ele é o líder. Logo, neste algoritmo o nodo com menor Id que é o líder.

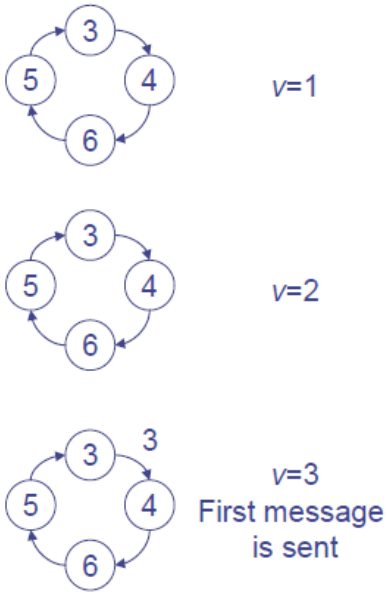


Figura 3 - Algoritmo TimeSlice (Figura extraída da internet - fonte desconhecida)

- Se existe um processo i com Id=v (v significa fases), e a fase (v-1) é executada sem que i tenha recebido uma mensagem anteriormente, então ele se elege líder e envia uma mensagem para cada nó da rede.

Complexidade de comunicação: O(n) – Complexidade de tempo: O(UID_{min} * n)

Obs.: Por exemplo: há uma rede de 5 nodos, onde o nodo de menor Id é 1000. Logo, serão necessários 5 * 1000 rounds para que o nodo de Id=1000 envie uma mensagem para os outros dizendo que ele é líder. Então não dá para prever o tempo que será gasto, já que ainda não se sabe o menor Id até que ele seja eleito. Mesmo em uma rede pequena, como citado no exemplo, a complexidade de tempo pode ser alta, embora a complexidade de mensagens seja apenas n.

2.1.4. Algoritmo VS (Variable Speed)

Originalmente os processos enviam uma mensagem com seu UID. Mensagens de processos diferentes caminham com velocidades diferentes. Cada mensagem carrega UID do processo que a originou e viaja com a velocidade de 1 mensagem a cada 2^{UID_MSG} rounds (ou seja, cada processo no caminho espera 2^{UID_MSG} rounds antes de repassar a mensagem).

Quanto menor o UID, mais rápido a mensagem vai caminhar.

Quando a mensagem com UID_{min} andou a volta completa, a mensagem com o segundo menor UID andou no máximo meia volta. Ou seja, o número de mensagens enviadas com UID_{min} é maior do que todas as outras mensagens somadas. Assim, o número total de mensagens é menor que 2*n*.

Complexidade de comunicação: o(2n) – Complexidade de tempo: $O(n*2^{UIDmin})$

Para avisar aos processos que o algoritmo terminou, e quem é o líder, temos:

Complexidade de comunicação: o(4n) – Complexidade de tempo: $O(n + (n*2^{UIDmin}))$

3. Algoritmos em redes gerais síncronas

(Conteúdo deste capítulo pode ser encontrado no capítulo 4 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

Ao contrário dos algoritmos vistos até este momento neste documento, que são aplicados em redes de topologia em anel, neste capítulo são apresentados algoritmos voltados para aplicação em redes com topologia geral.

Algoritmos vistos neste capítulo:

1 – Inundação (FloodMax)

- Versão otimizada (OptFloodMax)

2 – Busca em largura (SyncBFS)

- Versão otimizada

3 – Caminhos Mínimos (Bellman Ford)

4 – Árvore geradora mínima (Minimum Spanning Tree – Kruskal)

3.1. Algoritmo Inundação (FloodMax algorithm)

→ Algoritmo simples para eleição de líder em uma rede geral

- Processos conhecem o diâmetro "d" da rede

- Grafo possui arestas direcionadas

- No primeiro round cada nodo envia o seu Id para os seus vizinhos.

- No próximo round os nodos comparam os Ids recebidos com o seu próprio Id. Então é repassado para seus vizinhos o maior Id (que pode ser um Id recebido pelos vizinhos, ou ser o próprio Id do nodo).

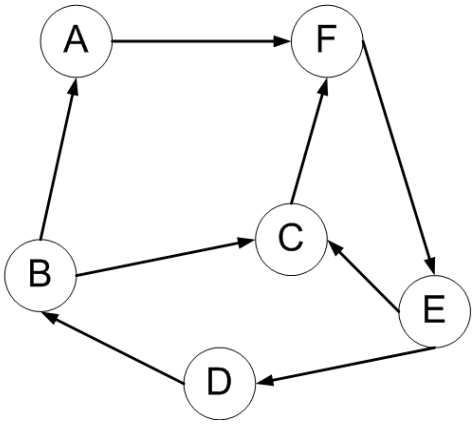
- Após "d" rounds (d = diâmetro da rede), o processo que não recebeu nenhum Id maior que o seu em nenhum momento se elege líder e os outros se elegem não líder.

- É suficiente a execução de "d" rounds, pois em um grafo com diâmetro d, após percorrer uma distância de comprimento d, é possível ter alcançado todos os nodos.

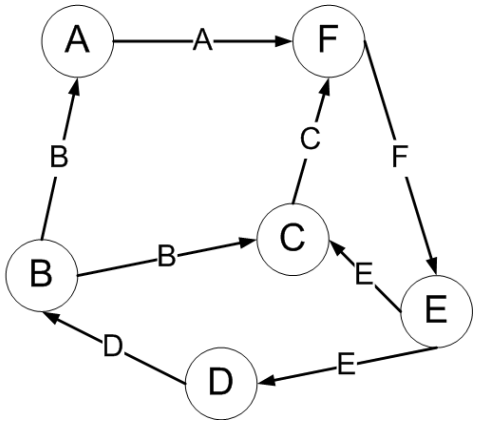
Complexidade de comunicação: diâmetro * |E| *(|E| significa "número de arestas do grafo")*

Complexidade de tempo: diâmetro rounds

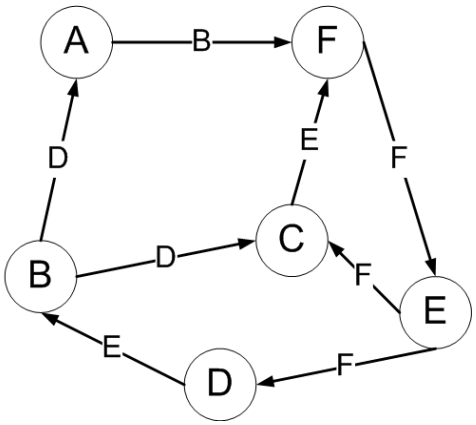
Round 0



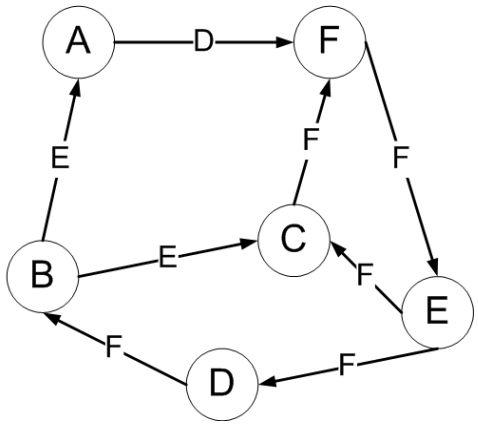
Round 1



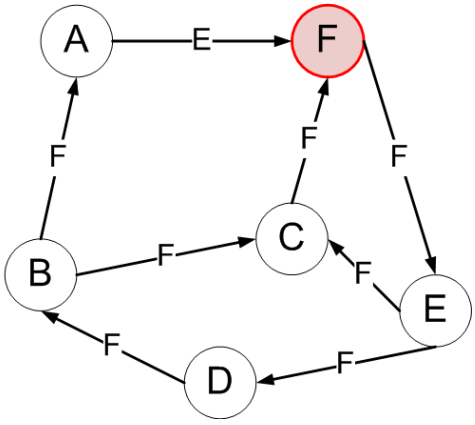
Round 2



Round 3



Round 4



Obs: Se eu não souber o diâmetro da rede, então posso usar o número de nós como condição de parada, no entanto isso vai modificar a complexidade.

3.1.1. Algoritmo de inundação otimizado (OptFloodMax)

Otimizando o algoritmo: Um processo só envia uma mensagem se o ID recebido por ele é maior do que o maior ID que ele já conheceu. Isso não vai mudar a complexidade de tempo, mas vai diminuir a complexidade de comunicação:

Complexidade de comunicação: É MENOR que diâmetro * número de arestas do grafo

Complexidade de tempo: diâmetro rounds

3.2. Busca em largura - Algoritmo SyncBFS

É utilizado para, a partir de um vértice conhecido i_0, "descobrir" (ou alcançar) todos os vértices do grafo. O algoritmo encontra a menor distância entre i_0 e todos os outros vértices do grafo (grafo possui arestas sem peso). Ao final do algoritmo é gerada uma "árvore primeiro na extensão" que, para um vértice v acessível a partir de i_0, o caminho na árvore primeiro na extensão de v até i_0 é o caminho mais curto, isto é, que possui o menor número de arestas. Este algoritmo pode ser utilizado, por exemplo, para:

- Broadcast
- Computação global
- Eleição de líder
- Determinação do diâmetro

Não é necessário conhecer o diâmetro do grafo para executar este algoritmo, no entanto, em sua primeira versão, mesmo sem conhecer o diâmetro o algoritmo vai demorar diâmetro rounds para ser executado.

Cada vértice pode estar pintado com uma das seguintes cores: branco, cinza ou preto. Inicialmente todos os vértices estão pintados na cor branca. O algoritmo de busca em largura não distribuído utiliza uma lista do tipo "primeiro a entrar, primeiro a sair" para gerenciar o conjunto de vértices na cor cinza.

Complexidade de comunicação: Exatamente |E|

Complexidade de tempo: O(diâmetro rounds)

- Grafo pode ser um dígrafo (arestas direcionadas, onde todos os vértices são alcançáveis de qualquer outro vértice)

A cada ponto da computação, existe um conjunto de processos "marcados". Inicialmente, apenas i_0 é marcado. Processo i_0 envia uma mensagem *search*, no round 1, para todos os seus vizinhos. Em qualquer round, se um processo não marcado recebe uma mensagem *search*, ele se marca e escolhe um dos processos dos quais recebeu mensagens *search* para ser seu pai. No primeiro round após ter sido marcado, processo envia mensagem *search* para todos os seus vizinhos de saída. Se um processo já marcado recebe uma mensagem, então ela é ignorada.

O produto deste algoritmo é uma árvore (é árvore pois cada vértice conhece seu pai) onde é conhecido o menor caminho entre um vértice e i_0.

Variação → Permitindo cada vértice conhecer não só seus pais, mas também seus filhos.

A cada ponto da computação, existe um conjunto de processos "marcados". Inicialmente, apenas i_0 é marcado. Processo i_0 envia uma mensagem *search*, no round 1, para todos os seus vizinhos. Em qualquer round, se um processo não marcado recebe uma mensagem *search*, ele se marca e escolhe um dos processos dos quais recebeu mensagens *search* para ser seu pai; envia mensagem *parent* para esse processo, e mensagem *non_parent* para todos os outros. No primeiro round após ter sido marcado, processo envia mensagem *search* para todos os seus vizinhos de saída.

Complexidade de comunicação: O(|E|) se arestas são bidirecionais ou O(diâmetro * |E|) se unidirecional. No entanto o número de bits é grande no caso unidirecional (|E|*b bits para cada mensagem**)**

Complexidade de tempo: O(diâmetro turnos) (as mensagens para avisar quem são os pais, embora demorem diâmetro turnos para dar a volta no grafo, são executadas em paralelo, então, assintoticamente falando, o tempo gasto é O(diâmetro)).

Terminação do algoritmo

Como o nó i_0 sabe que a computação terminou? Através do CONVERGECAST

Começando das folhas, cada folha envia notificação de terminação para seus pais. Processos que não são folhas podem enviar notificação de terminação para seu pai se:

- recebeu respostas para todas as mensagens search (sabe quem seus filhos são)
- recebeu notificação de todos os seus filhos.

Complexidade de comunicação - Bidirecional: O(|E|)

Complexidade de tempo - Bidirecional: O(diâmetro)

Complexidade de comunicação - Unidirecional: O(diâmetro^2 * |E|)

Complexidade de tempo - Unidirecional: O(diâmetro^2) (Para as folhas avisarem seus pais que elas terminaram, são gastos, para cada nível da árvore gerada, diâmetro turnos para avisar ao pai que o algoritmo terminou. Isso é realizado sucessivamente para todos os níveis, logo, diâmetro * diâmetro rounds para realizar a terminação. Em detalhes, seria gasto aproximadamente diâmetro turnos para criar a árvore + diâmetro^2 turnos para realizar a terminação. Como a análise é assintótica, então o tempo total é O(diâmetro^2)).

3.3. Algoritmo BellMan Ford

Problema: Determinar o caminho mínimo de um nó i_0 a todos os outros nós do grafo.

Complexidade de comunicação: (n-1) * |E|

Complexidade de tempo: n-1

- Encontra um caminho mínimo (soma dos valores das arestas) entre i_0 e todos os outros vértices do grafo (*em uma rede geral*)
- Comunicação unidirecional ou bidirecional
- Grafo tem ciclos
- Arestas têm peso
- Cada processo conhece o número total de processos
- Se houver CICLOS NEGATIVOS o algoritmo avisa que não pode encontrar o caminho mínimo
- Utiliza técnica de relaxamento de arestas, isto é, progressivamente vai diminuindo a estimativa de distância até encontrar o menor caminho.

A diferença entre Bellman Ford (caminho mínimo) e o Kruskal (árvore geradora mínima) é que o segundo cria uma árvore de custo mínimo para ligar TODOS os vértices do grafo. BellMan Ford encontra o caminho mínimo entre DOIS vértices do grafo.

Algoritmo: Vamos definir i como o vértice de origem. Todos os vértices têm duas variáveis: *distância* e *pai*. *Distância* guarda o menor valor já visto entre o vértice atual e i. *Pai* guarda quem é o pai do vértice atual para o menor valor já recebido. Inicialmente i guarda *distância* 0 e ele não tem *pai*. Todos os outros vértices inicializam com *distância* igual a infinito e *pai* desconhecido.

Compreendendo o algoritmo: até agora podemos ver que a cada iteração cada vértice guarda o menor caminho que ele já conheceu entre ele e o vértice i.

O algoritmo inicia com o vértice i enviando mensagens para seus vizinhos. Cada vizinho vai receber todas as mensagens (pode ser que ele receba mais de uma mensagem por round). Em seguida o vértice que recebeu pelo menos uma mensagem compara as distâncias recebidas juntamente com a distância que ele armazena (inicialmente a distância de um vértice intocado é igual a infinito, então qualquer mensagem que ele receber vai ter *distância* menor, e ele vai armazenar essa distância). Em seguida o vértice vai atualizar o seu *pai* com o ID da aresta que lhe enviou a menor *distância*. Caso a distância dele for menor que as recebidas, ele não altera suas variáveis *pai* nem *distância.*

A cada nova mensagem recebida que tenha uma distância menor, o valor armazenado é diminuído. Isto é chamado de relaxamento.

Somente quem sofreu modificação vai enviar nova mensagem para seus vizinhos.

Os relaxamentos de arestas permitem que a mínima distância se propague pelo grafo (na ausência de ciclos negativos), e o caminho mais curto vai somente visitar cada vértice no máximo uma vez.

O processo termina quando round = número de vértices – 1. Isso vai garantir que todos os vértices vão conhecer o menor caminho para i, já que com n-1 rounds é possível percorrer todos os caminhos possíveis.

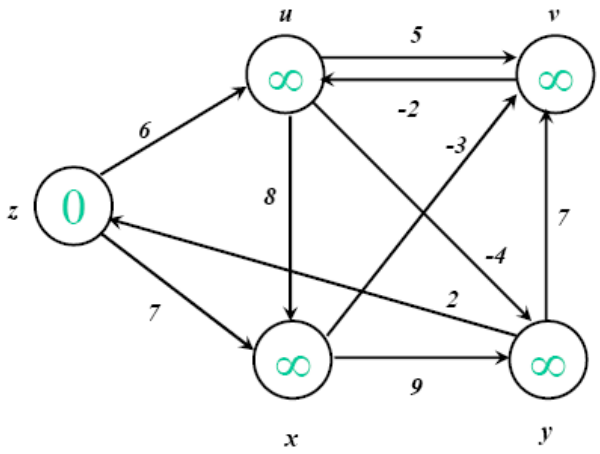


Figura 4 - Algoritmo BellMan Ford - Round 0 (Figura extraída da internet - fonte desconhecida)

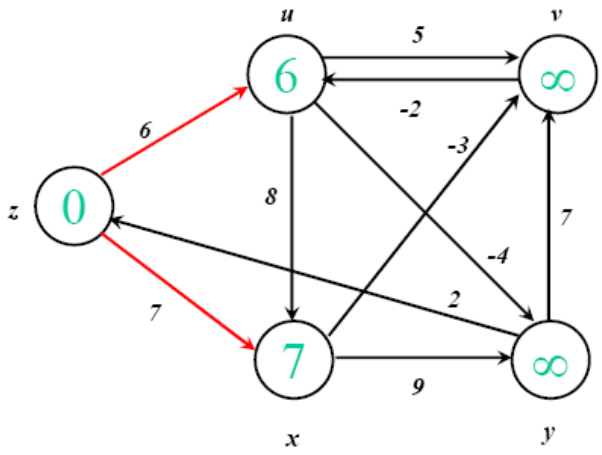


Figura 5 - Algoritmo BellMan Ford - Round 1 (Figura extraída da internet - fonte desconhecida)

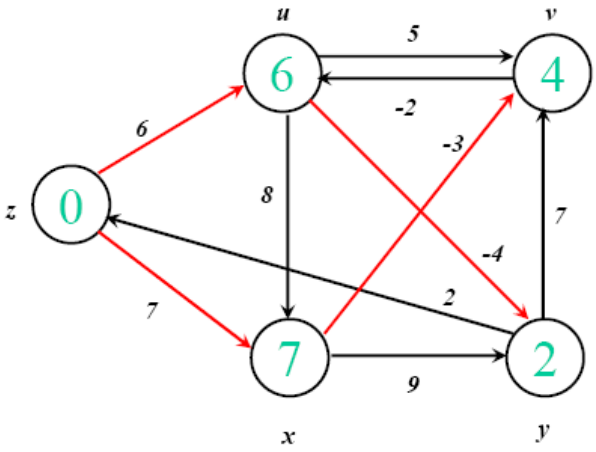


Figura 6 - Algoritmo BellMan Ford - Round 2 (Figura extraída da internet - fonte desconhecida)

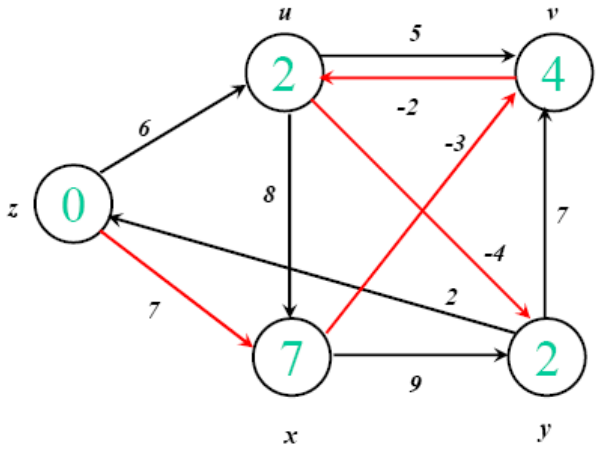


Figura 7 - Algoritmo BellMan Ford - Round 3 (Figura extraída da internet - fonte desconhecida)

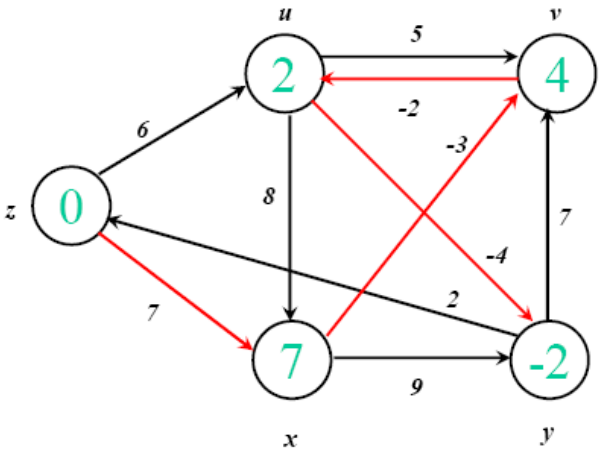


Figura 8 - Algoritmo BellMan Ford - Round 4 (Figura extraída da internet - fonte desconhecida)

→ *Caminhos mínimos (Bellman Ford)* e *busca em largura (BFS)* têm tem uma estrutura conveniente para realizar broadcast

3.4. Árvore Geradora Mínima / Minimum Spanning Tree - SynchGHS

Dado um grafo com arestas não direcionadas, contendo pesos nas arestas, encontrar uma árvore que permita ligar todos os vértices com o menor custo, e de modo que não haja ciclos (O resultado final vai ser o mesmo grafo, no entanto com várias arestas "deletadas").

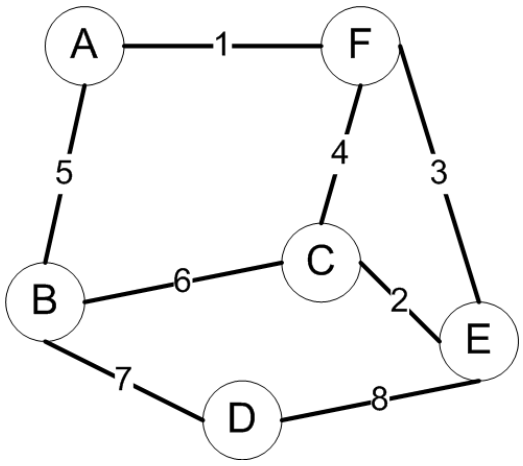


Figura 9 - Grafo original G

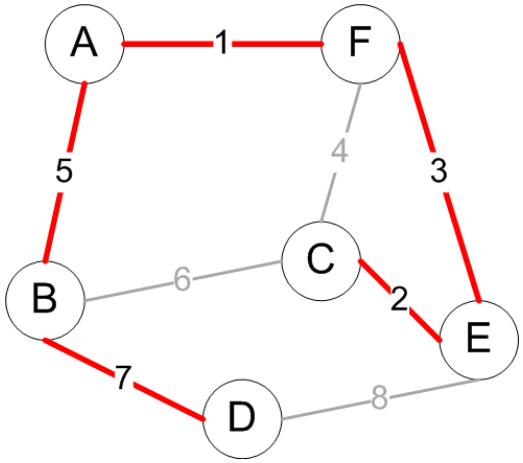


Figura 10 - Árvore geradora de custo mínimo – Minimum Spanning Tree para o grafo G

Dado um grafo, existem dois algoritmos muito conhecidos e não distribuídos para criar a árvore geradora de custo mínimo:

- PRIM
- KRUSKAL

3.4.1. Algoritmo Kruskal (Não distribuído)

Vamos descrever brevemente o funcionamento do algoritmo de Kruskal, antes de introduzirmos o SynchGHS, um algoritmo distribuído para encontrar a árvore geradora mínima.

Kruskal é um algoritmo não distribuído. Para achar a árvore geradora mínima utilizando o algoritmo de Kruskal, pegamos os valores de todas as arestas contidas no grafo e as colocamos em ordem crescente:

1-2-3-4-5-6-7-8

Depois, em ordem crescente vamos pintando cada aresta. Primeiro aresta de valor 1, depois a 2, então a 3. A aresta de valor 4 não pode ser marcada, pois se for marcada vai gerar um ciclo, e isso não é permitido. Depois é pintada a aresta 5. A aresta 6 não pode ser pintada pois geraria um ciclo. Em seguida é pintada a

aresta de valor 7. O valor 8 não é pintado pois geraria um ciclo. Pronto! Uma árvore geradora mínima foi criada (minimal spanning tree), conforme Figura 10.

Acabamos de ver de maneira prática o funcionamento do algoritmo KRUSKAL para criar uma árvore geradora de custo mínimo.

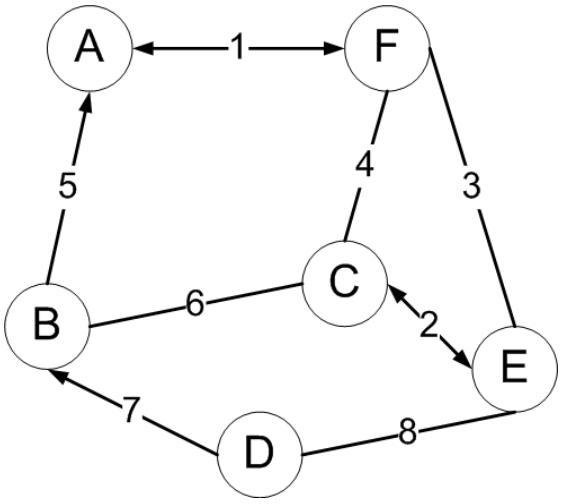
3.4.2. Algoritmo SynchGHS (versão distribuída para encontrar a Minimum Spanning Tree)

Complexidade de comunicação: O(n + |E|) * log n

Complexidade de tempo: O(n log n)

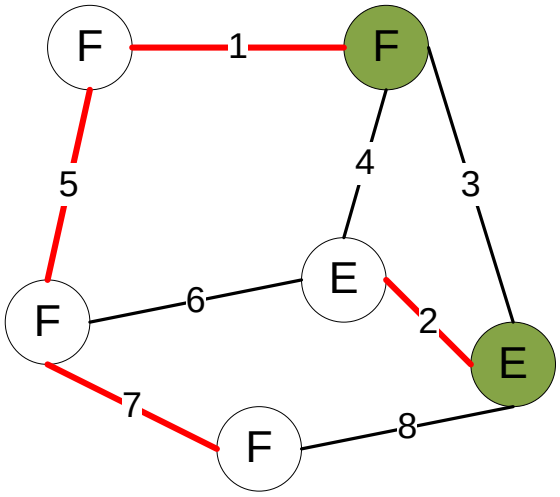
Agora vejamos como funciona o **SynchGHS**, um algoritmo distribuído para criação da árvore geradora mínima (Minimum Spanning tree):

Início: cada nó vai verificar qual a sua aresta de menor valor e enviar uma mensagem.



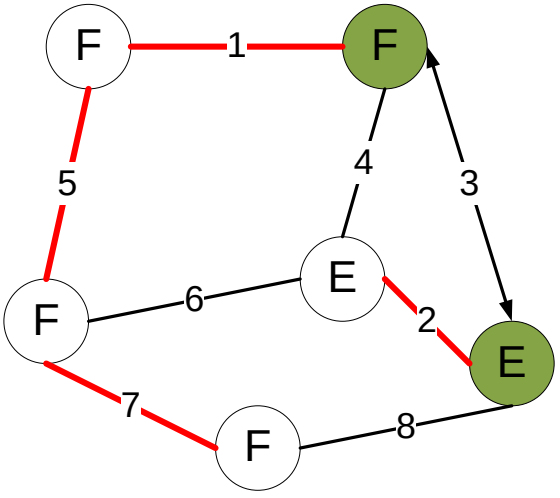
Note que neste round todos os vértices enviaram uma mensagem pela sua aresta de menor valor. Todas essas arestas devem fazer parte de uma nova componente conexa, as quais formarão a minimum spanning tree que queremos encontrar. Note que os vértices A-F e C-E enviaram e receberam mensagens no mesmo round, logo, eles deverão entrar em um consenso e definir qual deles será o líder para cada componente ao qual fazem parte. O importante aqui é saber que quando um vértice envia e recebe mensagem no mesmo round, então ele deve entrar em consenso com o vizinho que enviou e recebeu a mensagem também no mesmo round a fim de decidirem qual dos dois será o líder do componente conexo que estão formando (não vamos entrar em detalhes aqui sobre qual o algoritmo adotado para decidir quem será o algoritmo, vamos utilizar aqui uma definição qualquer, por exemplo, o processo de maior ID será o líder).

Após conectar os vértices pelas arestas selecionadas, então teremos dois diferentes componentes conexos. Cada componente conexo vai ter apenas um ID, para isso é atribuído a todos os nós do componente conexo o ID do maior nodo que pertence a este componente conexo, conforme a figura a seguir:

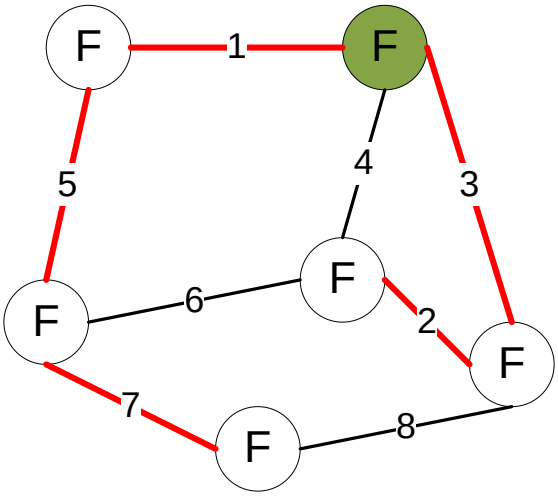


Recordando que A-F e C-E enviaram e receberam mensagem no mesmo round, então A e F entraram em consenso e decidiram que F seria líder da nova componente conexa. O mesmo ocorreu entre C e E, que decidiram que E seria o líder da componente conexa que eles formaram. Os líderes estão representados na cor verde.

No próximo round, cada líder pergunta para todos os vértices de seu componente conexo quais são seus vizinhos de menor valor. Primeiro vejamos a componente F. O líder de F pergunta para todos os vértices quais são as arestas de menor valor (3, 6 e 8). Em seguida, entre os valores retornados para o líder, ele decide qual deles é o menor, e envia uma mensagem para o vértice que lhe enviou esse valor e manda ele se conectar através desta aresta (F precisa ser visto como uma aresta só, e verificamos que ela se liga com 3, 4, 6 e 8. Entre estes valores, o líder escolhe o menor, logo, 3). O mesmo ocorre em E, e a menor aresta que liga este componente também é 3 (Observe que as arestas que estão em vermelho não devem ser consideradas, pois já foram ligadas). Após escolhidos estes valores, são enviadas mensagens para esses vizinhos falando que eles são os menores, conforme exibido na figura:



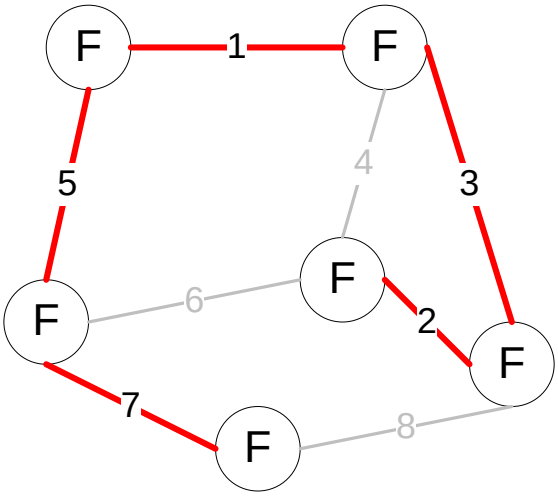
Podemos verificar que F-3-E enviaram e receberam mensagem no mesmo round, logo, eles devem decidir entre eles qual será o líder da nova componente conexa que vão formar. (Por coincidência deste exemplo, a decisão de líder será entre dois líderes, mas pode haver situações onde a decisão de quem será o líder ocorrerá entre vértices que não são os líderes da componente. O importante é ter em mente que a decisão de quem será o líder será sempre entre os vértices que enviaram e receberam mensagem no mesmo round).



Como E e F se conectaram pela aresta 3 e se transformaram em um só componente conexo, então todos os vértices devem ter um mesmo UID para que eles possam ser vistos como uma só componente, então, os vértices E se transformaram em F, e agora todos são uma só componente.

No próximo round o líder de cada componente (só restou uma componente) pergunta para cada vértice de sua componente quais são as menores arestas que cada um conhece (arestas que não fazem parte da componente e que não podem ligar a componente a ela mesma, senão gera ciclos). Nenhum componente retorna nenhuma mensagem, pois não há mais arestas que podem ser ligadas de modo que não haja ciclos. Se ligarmos qualquer uma das arestas 4, 6 ou 8, a componente conexa F será ligada a ela mesma, gerando um ciclo, então, como o líder não recebeu nenhuma mensagem dos vértices que fazem parte da componente conexa, então ele sabe que a computação terminou e que a árvore geradora mínima foi encontrada.

A seguir é exibida a árvore geradora de custo mínimo. As arestas na cor cinza são desprezadas, e estão descritas na figura apenas para facilitar o entendimento.



4. Consenso distribuído com *falhas de links*

(Conteúdo deste capítulo pode ser encontrado no capítulo 5 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

Falhas de link

- Erros ocorrem SEMPRE nos links, e não nos processos.

Definição de FALHA: “Determina-se falha quando uma execução violou a especificação”.

→ Não existe um algorítimo determinístico para solucionar falhas de link.

Terminação: Em algum momento no futuro (eventualmente) todos os processos decidem.

Cenário: Rede é um grafo não direcionado e conexo.

Tipos de falha:

- Falha e pára: mais fácil de tratar (Não fica enviando mensagens inválidas/erradas)
- Falha bizantina: cada processo pode enviar o que quiser. Podem não enviar nada por um momento e em seguida enviar novamente. Podem enviar mensagem errada. O importante é notar que falha bizantina nunca pára.

5. Consenso distribuído com *falhas de processo*

(Conteúdo deste capítulo pode ser encontrado no capítulo 6 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

O consenso em um ambiente distribuído pode ser útil, por exemplo, em uma transação bancária onde vários processos devem entrar em consenso se a transação vai ou não ser realizada em todos os processos que compõem o ambiente.

Tópicos vistos neste capítulo:

-Consenso em falhas de processo

- FALHA E PÁRA (Stopping failures)
    - Algoritmo FloodSet
    - Algoritmo OptFloodSet
- FALHA BIZANTINA

5.1. Falhas de processo (FALHA E PÁRA)

Modelo tradicional: É definido um número F máximo de falhas que podem ocorrer.

- O grafo é completo.

- Uma falha pode ocorrer a qualquer momento da execução.

- Objetivo: Obter o consenso em um valor possível e permitido.

- Consenso: Entre todos os processos que decidem, todos decidem em um mesmo valor.
- Validade: Ex: Se os processos começam com valor V, então eles só podem definir um consenso em V.
- Terminação: Todos os processos não falhos decidem.

5.1.1. Algoritmo FloodSet

FloodSet é um algoritmo simples para resolver o problema de consenso em redes síncronas com falhas de processo do tipo **FALHA E PÁRA.**

"Para garantir que o consenso vai ocorrer com sucesso, é necessário, apenas, executar (F + 1) rounds, onde F é o número máximo de falhas permitidas."

Adotando que a rede é um grafo não direcionado e **completo** (todo processo conectado com todos os outros processos), o *floodSet* funciona da seguinte maneira:

Todos os processos iniciam com uma variável W, que inicialmente possui um valor qualquer de V (Neste exemplo, os processos i, j, k e l escolhem respectivamente os valores 0, 1, 2 e 3. Neste exemplo V={0, 1, 2, 3}. É importante ter em mente que diferentes processos podem escolher valores iguais, desde que eles façam parte do conjunto V. Por exemplo, i, j e k poderiam escolher o valor 1 e "l" escolher o valor 2). Em cada round, cada processo envia os valores de sua variável W para todos os outros processos, que vão armazenando os valores recebidos em sua variável W local. Como é permitido no máximo F falhas, é lógico que em pelo menos um round entre 0 e (F+1) rounds todos os processos (que ainda estão ativos) vão comunicar-se corretamente. Então, após (F+1) rounds cada processo é capaz de decidir. Como?

Terminação: No round (F + 1), se W tem somente um elemento, então a decisão é sobre esse único elemento (Se W tem somente um elemento é porque todos os outros processos falharam logo no primeiro round e não foi recebida nenhuma mensagem, então ele decide no próprio valor), caso contrário, a decisão é sobre um valor default V_0.

Terminação resumida: Decisão é "v" quando W={v}, caso contrário v_0.

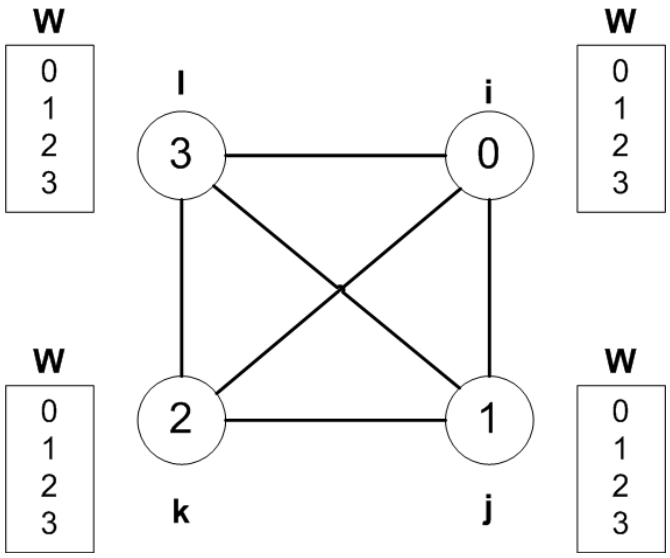


Figura 11 - Ilustração da execução do algoritmo FloodSet - Após r rounds todo processo deve ter sua variável W contendo pelo menos os valores de todos os outros processos ativos.

Lemas:

1 – Se até um round r, que esteja entre o round 1 e ou round (F+1), não ocorrer falha, então $W_i(r)=W_j(r)$. (Claro, pois pelo menos um dos rounds não teve falha, e neste específico round todos os processos enviaram mensagens para todos os outros processos, o que vai fazer as variáveis W de cada processo serem idênticas)

2 – Se $W_i(r)=W_j(r)$, é porque houve pelo menos um round sem falhas. Então, se R está entre 1 e (F + 1), em qualquer outro round entre R e (F + 1) continuará valendo que $W_i(r)=W_j(r)$ (Isso vale para o caso onde i e j continuam ativos até o final. Se houver falha em algum deles, eles ficarão inativos, pois estamos adotando que neste contexto as falhas são do tipo falha e pára).

3 – Se dois processos i e j estão ativos ao final do round F + 1, então $W_i=W_j$.

Complexidade de tempo: F + 1 rounds

Complexidade de mensagens: O((F+1) * n^2)

Complexidade de comunicação: O((F+1) * bn^3) - *Como cada mensagem contêm uma lista de no máximo n elementos, então o número de bits por mensagem é O(n*b). Então foi multiplicado este número pela complexidade de mensagens para achar a complexidade de comunicação.*

5.1.1.1. FloodSet Otimizado (OptFloodSet)

No primeiro round cada processo i faz broadcast do valor desejado (um valor pertencente a V)e adiciona em W esse valor juntamente com os valores recebidos dos outros processos. Nos próximos F+1 rounds, cada processo faz broadcast de apenas mais uma mensagem, no momento em que o processo conhece um novo valor que ainda não conhecia.

O fato de saber que W possui mais que um valor implica em decidir em V_0. Logo, não há necessidade de enviar mensagens repetidas em todos os rounds.

Então, no primeiro round todos enviam (total de n^2 mensagens). Em algum outro round entre 2 e F+1, **se houver** algum round onde o processo conheça um novo valor, então ele envia nova mensagem para todos (envio de mais uma cota de n^2 mensagens) **e não envia mais.** Trocar mais mensagens seria desnecessário, pois só o fato de saber que W possui mais que um elemento é suficiente.

Terminação: Idêntico ao FloodSet: Decisão é "v" quando W={v}, caso contrário v_0.

Complexidade de tempo: F + 1 rounds

Complexidade de mensagens: O($2n^2$)

Complexidade de comunicação: O(n^2b) – O valor não assintótico para essa análise seria ($b*2n^2$).

5.2. Falhas bizantinas

Para solucionar o problema de consenso em uma rede com *falhas bizantinas* é necessário que, o número de vértices da rede seja pelo menos 3 vezes maior que o número de falhas:

$$n > 3f$$

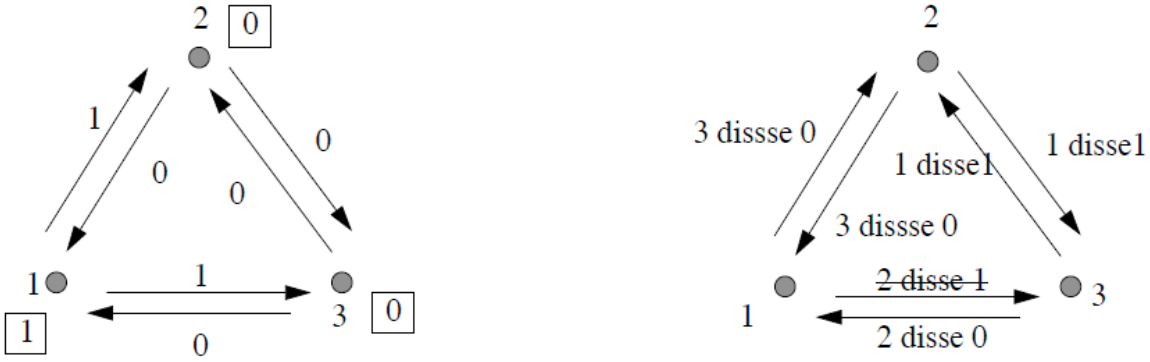


Figura 12 - Exemplo de falha bizantina, apenas para ilustrar que se n <= 3f, então é impossível solucionar o problema de consenso (Figura extraída da internet - fonte desconhecida).

Na Figura 12 é exibido um exemplo para ilustrar que é impossível solucionar o problema de consenso em uma rede com falhas bizantinas onde n <= 3f. Note que o processo 3 não consegue distinguir quais os processos são falhos, processo 1 ou processo 2.

6. Modelo Assíncrono

(Conteúdo deste capítulo pode ser encontrado no capítulo 8 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

Antes de iniciar a introdução do modelo assíncrono, imagine um exemplo hipotético, onde várias pessoas desejam utilizar uma mesma caneta (Figura 13). Neste exemplo, todas as pessoas conseguem se comunicar com todas as pessoas, e devem chegar a um **consenso** sobre qual vai ser a ordem das pessoas que vão utilizar a caneta (a ordem das pessoas que têm interesse em utilizá-las). Tenha em mente que a caneta pode ser utilizada por uma pessoa de cada vez, de modo que haja **justiça**.

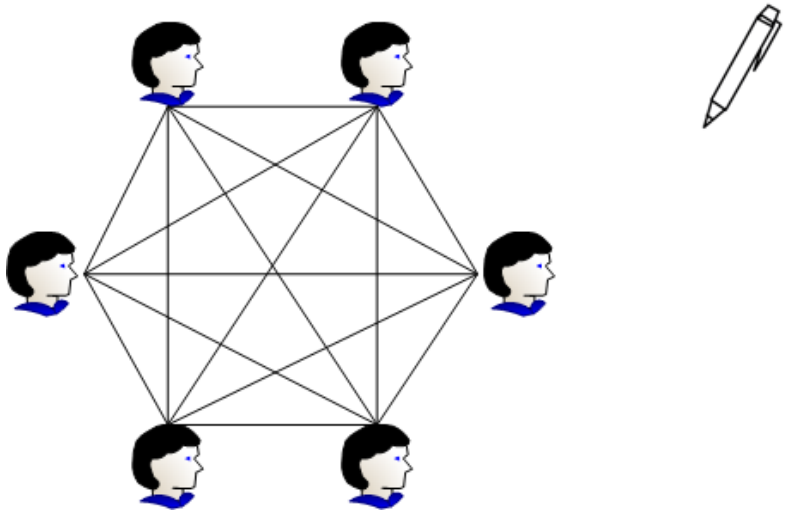


Figura 13 - Exemplo de várias pessoas que querem utilizar uma caneta. Este problema envolve consenso em um ambiente distribuído.

Situações semelhantes a este exemplo ocorrem em um **ambiente distribuído**, onde processos concorrem entre si para acessar um mesmo recurso (garantir **exclusão mútua –** recurso pode ser acessado por apenas um processo de cada vez) com a garantia de **justiça**, ausência de **deadlock** e ausência de **livelock**. Neste capítulo são tratadas soluções para esta categoria de problemas.

A seguir são descritos alguns conceitos necessários para a compreensão do funcionamento do **modelo assíncrono**:

– O envio de mensagens ocorre a qualquer momento, ao contrário do modelo síncrono que envia mensagens em rounds.

– Cada processo possui uma *máquina de estados* (autômato) interna, onde cada *transição* é equivalente a uma *ação*. A ação pode ser de entrada, saída ou interna.

Exemplo de transição: Transição(estado_inicial, ação, estado_final)

Autômato de entrada/saída:

- Assim como em uma representação de autômatos finitos, a representação de uma autômato de entrada/saída de cada processo é feita similarmente através de 5 componentes:

Componentes:

1 – Estados iniciais.

2 – Conjunto de estados.

3 – Relação de transição - Trans(s, p, s') ← Assim como exemplo acima. Para cada estado e para ação de entrada deve haver uma transição.

4 – Assinatura (conjuntos de entrada, conjunto de saída).

5 – Threads de controle para garantir justiça.

"Dizemos que um estado é ***quiescente*** se as únicas ações possíveis nesse estado são ações de entrada. Em outras palavras, o processo não faz nada a não ser quando recebe uma mensagem. Por curiosidade, um algoritmo de eleição termina quando todos os processos estão em estado ***quiescentes***".

Execução:

- Uma execução é uma seqüência de *estados* e *transições*: $est_0 \rightarrow trans_0 \rightarrow est_1 \rightarrow trans_1 \rightarrow ...$

- Um estado é alcançável se ele é o *estado final* de uma execução.

Traço:

- Os processo (autômatos) podem ser vistos como uma caixa preta, sendo observadas apenas as ações externas do processo, as quais chamamos de **traço**. Seguem o mesmo formato de alternância entre *estados* e *transições*, assim como na execução: $est_0 \rightarrow trans_0 \rightarrow est_1 \rightarrow trans_1 \rightarrow \ldots$

Justiça:

- Cada tarefa do autômato (thread) ganha infinitas oportunidades de executar uma de suas ações.

- Como não há rounds no modelo síncrono, então é necessário, de alguma forma, medir a ordem de execução dos eventos, a fim de garantir **justiça.** Para solucionar esse problema, existem algumas abordagens, como por exemplo *sincronização de relógios físicos e lógicos*.

6.1. Sincronização de relógios físicos

→ Em um ambiente <u>não</u> distribuído

Primeiramente, é necessário conhecer a definição para "evento": Evento é qualquer ação importante no sistema. Ex: Manifestar desejo de uso de um recurso, enviar uma mensagem, acionar um alarme, etc.

Do ponto de vista de apenas um computador com um processador, a ordem de ocorrência dos eventos é bem definida:

$$e1 \rightarrow e2 \rightarrow e_{n-1} \rightarrow e_n \ldots$$

Histórico do processo: é uma seqüencia de eventos h<e1, e2, ...e_n> dentro de um mesmo processo.

→ Mas, como atribuir um tempo para cada evento?

Resposta: Uma das maneiras é utilizar a hora do relógio local. Cada evento recebe um *timestamp*, desta forma é possível atribuir um tempo físico para cada evento.

→ Em um ambiente distribuído

Utilizar a hora local de cada computador para atribuir tempo aos eventos pode trazer problemas para computadores ligados em rede, uma vez que cada computador pode ter **horas diferentes**. Logo, atribuir um *timestamp* para os eventos em computadores ligados em rede não é uma tarefa trivial.

Exemplo: Pode não ser possível ordenar todos os eventos ocorridos em todos os computadores de uma rede, pois dois eventos que ocorreram em um mesmo momento em computadores diferentes podem estar com *timestamps* diferentes devido aos computadores terem horas erradas.

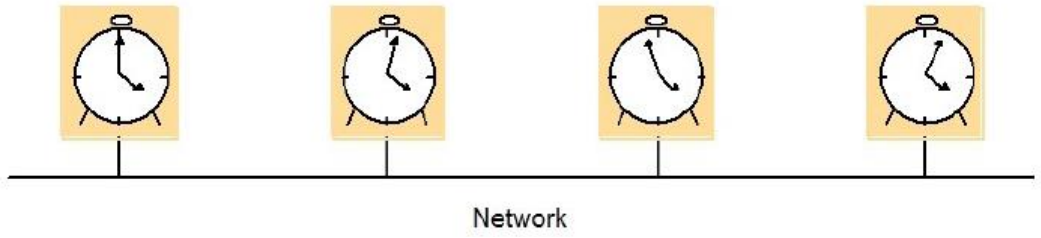


Figura 14 - Diferentes computadores ligados em rede, marcando horas diferentes (Figura extraída da internet - fonte desconhecida).

É interessante calcular a hora que certos eventos ocorreram, em diferentes máquinas, para:

- saber se um evento E1 é anterior ou posterior a outro evento E2
- reconstituir uma sequência ordenada (rollback, auditoria)
- saber se dois eventos são simultâneos

** Para que eu sincronizo relógios? → Para que seja possível ordenar eventos e conseqüentemente garantir* **justiça!**

Exemplo: Para compreender a importância do **sincronismo de relógios** e de **justiça** em uma rede assíncrona, analisemos a Figura 15. Note que o processo A requisita o recurso no tempo A, e a mensagem demora tempo 10 para ser enviada. Um pouco depois, no tempo 42, o recurso B solicita o mesmo recurso, sendo que a mensagem demora tempo 5 para ser enviada.

Neste exemplo, o recurso vai receber a requisição de B no tempo 47, e a requisição de A mais tarde, no tempo 50. Notem que mesmo A solicitando o recurso primeiramente, sua mensagem chega depois da solicitação de B que foi feita em um momento posterior. Por este motivo, é importante tratar o problema de **sincronização de relógios** em uma rede, a fim de garantir **justiça**.

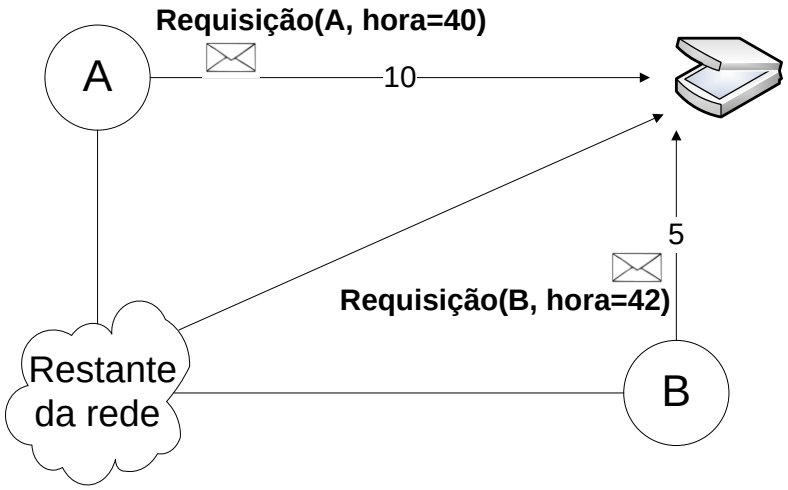


Figura 15 - Problema de justiça em uma rede assíncrona

Para solucionar o problema de horas incorretas em relógios, é possível realizar a sincronização de relógios físicos:

- **Sincronização externa (UTC)**: computador sincroniza seu relógio através de uma fonte externa. Após sincronizado, a hora do computador não pode ter diferença superior a um limite pré-definido D (delta), em relação ao servidor de hora.

- **Sincronização interna**: consiste em sincronizar todos os relógios dos computadores da rede sem considerar nenhuma outra fonte de relógio externa. Neste tipo de sincronização não importa se o tempo dos relógios estão correndo mais rápido ou devagar que a hora do mundo real, o que importa neste caso é apenas que os relógios caminhem juntos. Neste tipo de sincronização, os relógios não possuem diferença superior a um limite pré-definido D (delta).

"Para compreender o problema de sincronização de relógios físicos, imagine uma situação hipotética onde é verificado que o relógio do computador C não está correto. Em seguida é enviado para ele uma função de ajuste de relógio, que vai ser utilizada para gerar a hora correta do sistema, e consequentemente para gerar os *timestamps* corretos para todos os novos eventos". Por exemplo: A hora correta é 1000, no entanto um computador C possui hora 1015. No momento da sincronização, é informado ao computador C uma função para ajuste de hora. Por exemplo:

Hora_correta := hora_do_sistema – 15;

6.1.1. Algoritmo de Berkeley

Este algoritmo é utilizado para **sincronizar relógios físicos** em um ambiente distribuído (Sincronizar relógios físicos prepara a rede para permitir que ela seja síncrona, isto é, baseada em um tempo comum para todos os processos. Em redes que utilizam relógios lógicos o cenário é assíncrono, pois os processos não conhecem uma hora comum global).

Funcionamento:

- A rede possui um mestre, o qual vai ser responsável pela sincronização dos relógios
- O mestre periodicamente solicita o tempo de todos os processos
- Mestre analisa todos os tempos e faz a estimativa das horas recebidas, levando em conta a o tempo que a mensagem demorou para ser chegar.
- Em seguida, exclui os valores que possuem diferença maior que um valor D (delta) pré-determinado. Após isso, é realizada a média sobre os valores recebidos (incluindo o tempo do mestre), exceto àqueles excluídos.
- Em seguida, o mestre estipula que a hora média encontrada por ele é a hora que deve ser atribuída a todos os processos.
- Então mestre envia um fator de ajuste para todos os processos, de modo que cada processo possua hora com limite máximo D aceitável de diferença entre todos os processos (Figura 16).

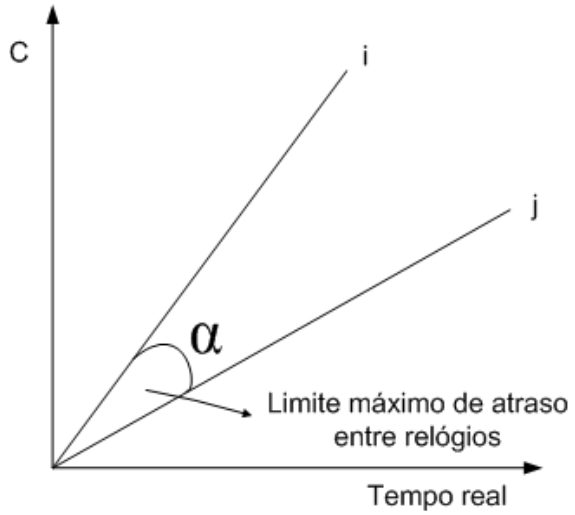


Figura 16 - Limite máximo aceitável de atraso entre os relógios dos processos "i" e "j"

Não é possível simplesmente atrasar o relógio caso ele esteja adiantado, senão vai gerar inconsistência no tempo dos eventos que já ocorreram nele.

Ao final da execução do algoritmo de **Berkeley**, é previsto que em um determinado tempo no futuro, todos os nós estejam **sincronizados**, inclusive levando em conta o tempo de troca de mensagens (Figura 17).

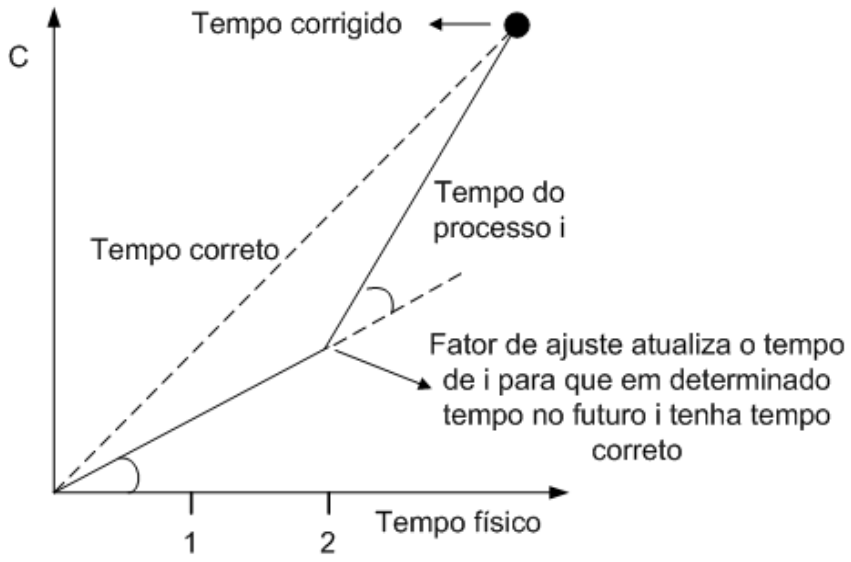


Figura 17 - Fator de ajuste aplicado para permitir que em determinado tempo no futuro o relógio esteja sincronizado.

6.2. Tempo Lógico

O uso do **tempo lógico** é uma outra maneira de **ordenar eventos**. Para compreender o significado de tempo lógico, é necessário compreender o significado da palavra **causalidade**, que está diretamente relacionada à **ordem de ocorrência dos eventos** em um sistema.

Causalidade é a propriedade que permite verificar que se um evento A deu início a um evento B, é porque o evento A ocorre antes de B. Exemplo: Quando uma mensagem é enviada, o evento "envia_msg" ocorre antes do evento "recebe_msg".

Então, utilizando o conceito de **causalidade**, é possível determinar a ordem de execução dos eventos do sistema. Essa ordem, que independe do relógio físico, é o **tempo lógico** (também chamado de **relógio lógico**).

6.2.1. Algoritmo de Lamport

Na Figura 18 é exemplificado o funcionamento do **algoritmo de Lamport**. Nela são exibidos 3 processos: p1, p2 e p3. Cada processo mantêm um contador local para gerenciar os tempos de execução dos eventos. A cada novo evento, o contador é incrementado. Note que p1 executa o primeiro evento, "a", no tempo 1. Em seguida, no tempo 2, manda uma mensagem m1 para o processo p2. O processo p2 precisa atribuir um tempo lógico para o evento que acabou de ocorrer (recebeu a mensagem). Note que a mensagem recebida por p2 é o primeiro evento que ocorre em p2. Teoricamente o seu tempo lógico seria 1 pelo fato de ser o primeiro evento ocorrido, mas como a mensagem m1 que chegou de p1 ocorreu no tempo 2 de p1, então o tempo lógico deste evento em p2 não pode ser menor que o tempo lógico de p1. Então, o tempo lógico do recebimento da mensagem m1 em p2 deve ser 3.

O mesmo ocorre em p2. P2 envia a mensagem m2 para p3. Embora p3 só tenha um evento ocorrido até o momento (evento "e" - no tempo lógico 1), p3 precisa registrar o recebimento de m2 em um tempo lógico superior ao tempo do envio de m2 por p2 (tempo 4), logo, p3 registra o recebimento de m2 no tempo lógico 5.

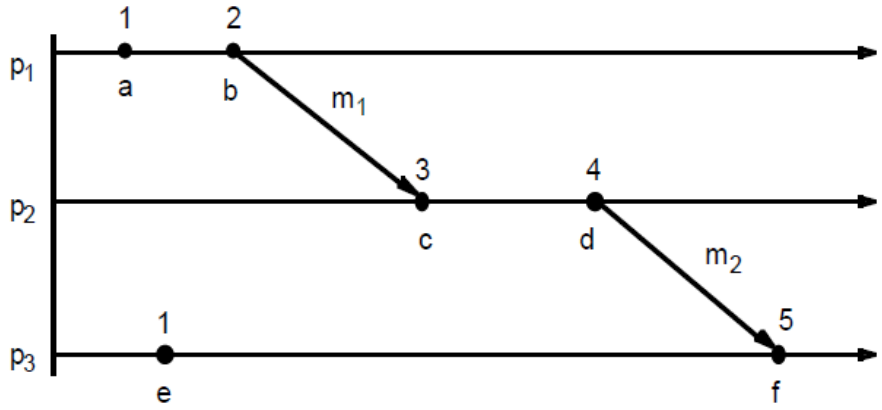


Figura 18 - Princípio do relógio lógico de Lamport (Figura extraída da internet - fonte desconhecida).

Outro detalhe importante a ser analisado na Figura 18 está relacionado aos eventos "a" e "e", que ocorrem em p1 e p3 respectivamente. Note que não são enviadas mensagens em nenhum dos 2 eventos, isto é, os eventos não possuem relação alguma com nenhum outro processo, então, são concorrentes. Logo, com base na premissa da **causalidade**, não importa que "a" ocorreu em um tempo físico anterior à execução de "e" e mesmo assim ambos tenham mesmo tempo lógico (tempo 1), pois "a" e "e" não interagem em nenhum momento com outro processo, isto é, são concorrentes.

*"Quando não for possível definir uma relação de **causalidade** entre dois processos:*

!(e1 → e2) e !(e2 → e1)

então os eventos são concorrentes e seus tempos lógicos não precisam ter relação alguma, assim como os evento "a" e "e" na Figura 18*"*

6.3. Algoritmos para Exclusão Mútua

6.3.1. Algoritmo de Lamport 78 – Solução distribuída

Em um ambiente distribuído, garantir **exclusão mútua** é essencial para permitir **justiça.** Outra característica importante é garantir a **ausência de deadlocks** e **livelocks**, para assegurar que os processos sempre caminhem. A solução distribuída do algoritmo de Lamport garante que essas premissas sejam satisfeitas.

- É necessário cada processo saber o número total de processos na rede.

- Cada processo mantêm uma fila de prioridades para armazenar os pedidos de acesso ao recurso. (**filas distribuídas**)

– Toda a rede tem noção global das filas de prioridade, pois cada processo possui uma fila local, que está sincronizada com qualquer outra fila de qualquer outro processo da rede, o que faz com que cada processo saiba quem é que está acessando o recurso em cada momento.

Funcionamento:

- Um processo P que quer usar um recurso faz broadcast de uma mensagem de requisição (essa mensagem contêm o ID e o tempo lógico de P), com o propósito de pedir autorização para entrar na fila de prioridades.
- Cada nó recebe o pedido de P e responde uma mensagem de ACK para P, informando que recebeu o pedido. Assim que cada processo receber a requisição de P, então P é colocado em sua lista local de prioridades (isso vai permitir que cada processo saiba o que está acontecendo na rede, isto é, quem está acessando atualmente o recurso e quais serão os próximos processos que vão acessar).
- Depois que P recebeu os ACKs de todos os processos, se ele for a requisição da lista de prioridades que possui o menor tempo lógico, então ele utiliza o recurso (mesmo que haja algum outro pedido de acesso ao recurso nesse período intermediário de troca de mensagens, quem tiver tempo lógico menor é quem vai acessar o recurso). Ao terminar de utilizar o recurso, P faz broadcast de uma mensagem de *release* e se deleta da sua lista de prioridades. Assim que os outros processos receberem a mensagem de *release*, então eles deletam o processo P da sua lista de prioridades. Depois disso, o processo da lista que possui menor tempo lógico acessa o recurso.

Exemplo: Processo i quer acessar um recurso, então, em seu tempo lógico 1, dá broadcast de uma mensagem de requisição (neste exemplo a rede é composta pelos processos i e j). Em seu tempo lógico 4, o processo j recebe a requisição e envia um broadcast de uma mensagem de ACK em seu tempo 5. Em seguida, adiciona "i" em sua lista de prioridades (a mensagem de ACK deve levar o maior tempo lógico conhecido pelo processo). No tempo lógico 6 de "i", o processo i recebe a mensagem de ACK de todos os processos da rede (neste exemplo, apenas j corresponde ao resto da rede), e se adiciona na sua lista de prioridades. Em seguida verifica quais são os processos que estão aguardando o recurso na sua lista de prioridades e compara o valor de suas prioridades para ver quem vai ter acesso ao recurso. Como ele está no topo da lista de prioridades, então ele pode utilizar o recurso (Figura 18).

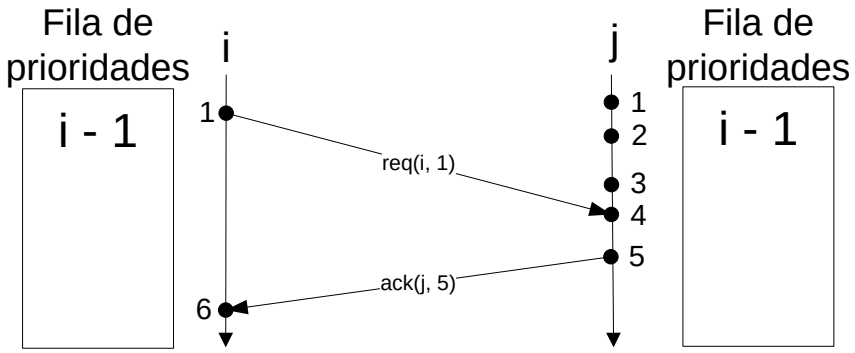


Figura 19 - Algoritmo de lamport - processo i solicita acesso ao recurso, enviando mensagem que é inserida na lista de prioridades

Enquanto o processo i está utilizando o recurso, o processo j envia para toda a rede uma solicitação de acesso ao recurso (Figura 20). Em seguida o processo i responde a mensagem através de um ACK e insere j em sua lista de prioridades. Ao receber a mensagem de ACK, o processo j se insere na sua lista de prioridades, **mas** ainda não pode utilizar o recurso, pois devido ao fato do processo i ainda estar em sua lista, e ao fato de i possuir tempo lógico menor que o tempo lógico de j, então o processo j sabe que ele tem menor prioridade e que i ainda está utilizando o recurso.

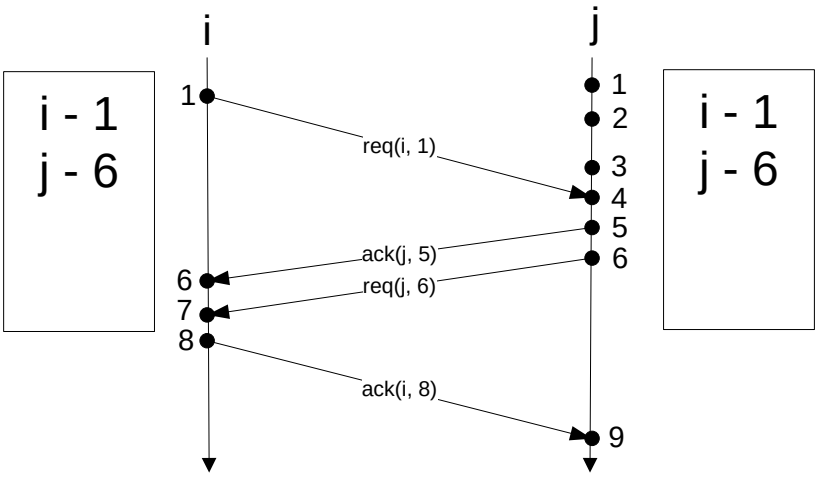


Figura 20 - Algoritmo de lamport - Enquanto i acessa o recurso, j envia mensagem requisitando autorização para entrar na lista de prioridades

Assim que processo i deixa de utilizar o recurso (tempo lógico 9 de i, Figura 21), o mesmo envia uma mensagem de *release* para todos os processos da rede e se exclui da lista de prioridades. No tempo lógico 10 de j, o processo j recebe a mensagem de *release* de i, exclui i da lista de prioridades, e a partir de então ganha acesso ao recurso, pois j é o processo com menor id na fila de prioridades (neste exemplo, ele é o único processo na lista).

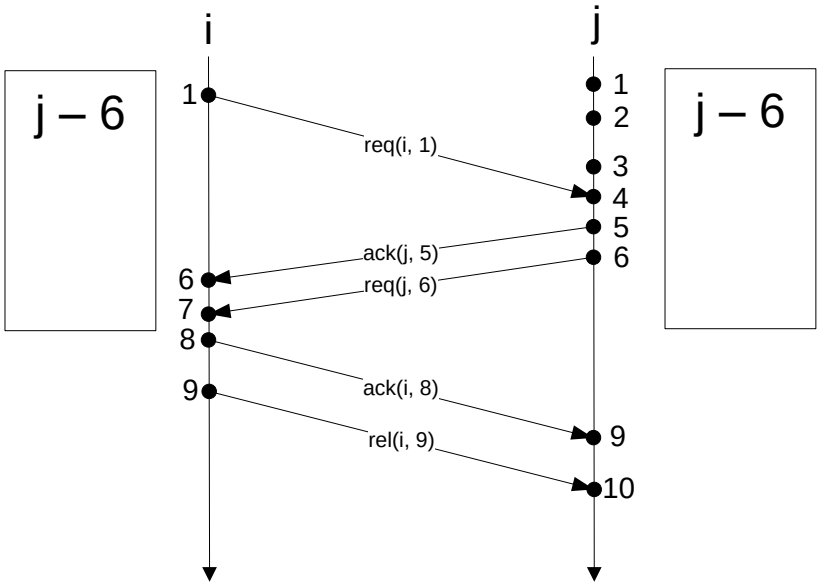


Figura 21 - Algoritmo de Lamport - processo i libera o recurso com uma mensagem de release (rel)

Obs.: Podem ocorrer situações onde há vários processos na lista de prioridades aguardando para ganhar acesso ao recurso. Quando um processo recebe uma mensagem de *release*, ele sabe que o processo foi liberado. Em seguida, ele vai verificar se ele é o processo que está na lista com o menor tempo lógico. Caso seja ele, ele vai acessar o recurso, caso não, deve aguardar uma nova mensagem de *release* para retirar mais

um processo da fila de prioridades, e isso ocorre sucessivamente até o momento em que o tempo lógico dele for o menor de todos.

Correção: a relação de causalidade garante a correção. Se por ventura houver dois processos com um mesmo tempo lógico que solicitam a entrada na região crítica ao mesmo tempo, então é utilizado seus respectivos IDs para realizar o desempate. Outro detalhe é que um processo só se retira da fila quando ele termina de usar o recurso. Enquanto isso, os outros processos não acessam o recurso devido a terem um tempo lógico global maior do que o processo que está acessando o recurso e que ainda encontra-se na lista, logo, é garantida a exclusão mútua.

Complexidade de mensagens: 3(n-1) – Existem três tipos de mensagens: Requisição, ACK (acknowledge) e release. Cada uma das 3 mensagens são enviadas para (n – 1) processos, logo, a complexidade é 3(n-1).

*Não há complexidade de tempo a ser calculado neste algoritmo, pois estamos lidando com um **modelo assíncrono**, onde não há rounds.*

6.3.2. Algoritmo de Ricart & Agrawala 81

Este algoritmo é considerado uma melhoria do algoritmo de Lamport

O algoritmo de "Ricart e Agrawala 81" faz uso da ordenação total dos eventos, portanto faz uso dos relógios de Lamport (**causalidade – tempo lógico**) e funciona da seguinte forma:

- Quando um processo quer acessar um recurso, ele faz broadcast de uma mensagem de requisição contendo seu ID e o valor de seu contador, Ex: Req($Proc_i$, 13). (O ID da primeira mensagem não precisa ser necessariamente 0, pois podem ter sido executados outros processos locais que variaram o valor do contator – mais informações, ler princípio do tempo lógico).
- Quando um processo recebe uma mensagem de requisição de acesso ao recurso, três ações podem ser tomadas:
    - Se o processo não está usando o recurso e também não quer usá-lo, responde a mensagem com um ACK (Ok).
    - Se ele ja está usando o recurso, ele não responde a mensagem ainda. Ao invés disso, ele enfileira a requisição. (em uma fila local)
    - Se ele quer usar o recurso mas ainda não o fez, ele compara a marca temporal da mensagem recebida com a marca temporal da mensagem que ele enviou para todos. Se a mensagem recebida tiver uma menor marca temporal, o processo responde com um ACK, caso contrário, ele enfileira a requisição.
- Após fazer broadcast da requisição para acesso ao recurso, o processo espera as repostas dessas requisições (ACKs). Uma vez que todas os ACKs foram recebidos, o processo usa o recurso.
- Ao terminar de usar o recurso, o processo envia mensagens de ACK para todos os processos que estão na sua fila de requisições e apaga esta fila.

Obs.: Neste algoritmo, os processos não sabem qual processo é que está utilizando o recurso em um determinado momento. Uma vez que um processo perdeu uma ou mais requisições, devido ao fato de ter um tempo lógico superior, então ele responde essas mensagens e não sabe quais processos estão acessando o recurso. Ele só tem a garantia que no futuro ele vai receber os ACKs solicitados e então o recurso estará disponível para ele.

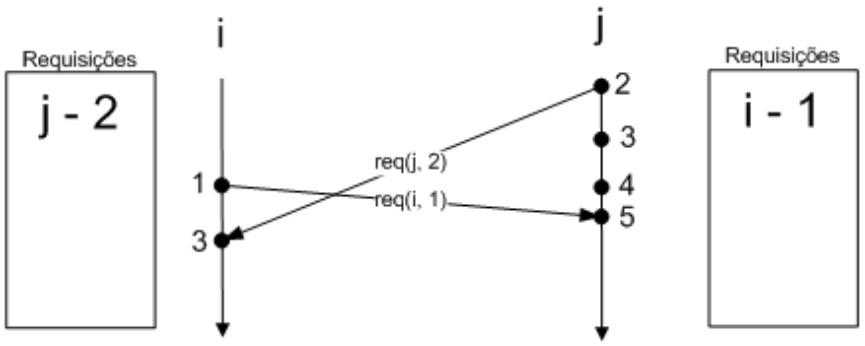


Figura 22 - Algoritmo Ricard & Agrawala 81 - Exemplo

Exemplo: Na Figura 22 dois processos querem utilizar um recurso. No tempo 2_j o processo J envia uma mensagem de requisição ao recurso para toda a rede (neste exemplo, a rede é composta pelos processos i e j apenas). Notem que ocorrem eventos nos tempos 3 e 4 de j, no entanto, eles não tem relação nenhuma com i, por isso não há nenhum tipo de influência. Para que o processo j possa acessar o recurso que ele solicitou, ele deve esperar mensagens de ACK de todos os processos da rede.

Enquanto a mensagem de j "viaja" pela rede, o processo i também envia uma mensagem de requisição de acesso ao recurso. Essa mensagem é enviada para toda a rede (neste exemplo, a rede é composta pelos processos i e j apenas).

Continuando a análise das mensagens, no momento 5 do processo j, o processo j recebe a mensagem do processo i. O processo j insere a requisição do processo i na sua lista local de requisições, incluindo o tempo lógico do envio da mensagem.

É importante destacar que os processos não se auto-inserem na lista de requisições, os processos armazenam nesta lista apenas as requisições pendentes dos outros processos, isto é, as requisições que o processo ainda não enviou a resposta de ACK.

Como j acaba de receber uma mensagem de requisição para acessar o recurso com um tempo lógico menor que o dele, então ele já sabe que vai perder a prioridade de acesso, e envia uma mensagem de ACK para i, conforme Figura 23 (a mensagem de ACK leva consigo o maior tempo lógico já visto para manter o tempo lógico). Quando i receber a mensagem de ACK de j (como j é o restante da rede, então i já terá recebido os ACKs de toda a rede) ele poderá utilizar o recurso.

Notem que a lista de requisições de j está vazia, pois j já enviou a mensagem de ACK para i.

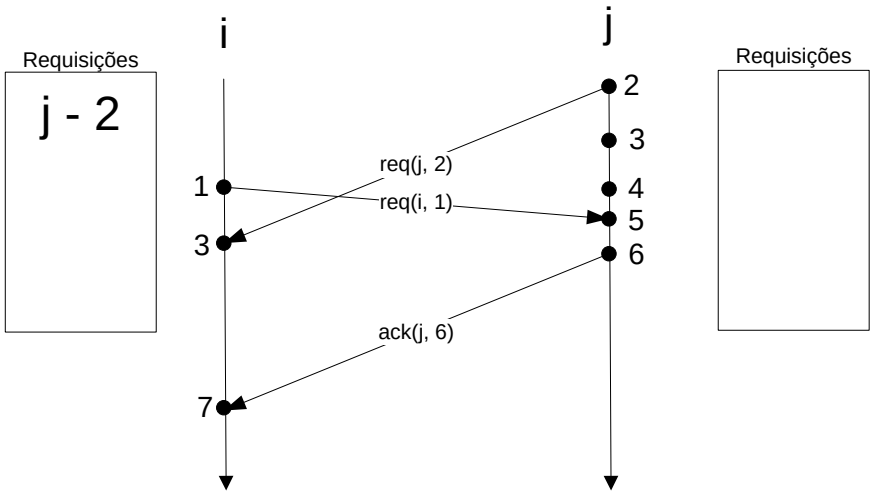


Figura 23 - Algoritmo Ricard & Agrawala 81 - j perde a disputa e envia mensagem de ACK. Em seguida deleta o pedido de i da sua lista local. No momento 6 o processo i utiliza o recurso.

Assim que i termina de usar o recurso ele envia mensagens de ACK para todos os processos que estão aguardando na sua fila de requisições e apaga esta fila, conforme Figura 24.

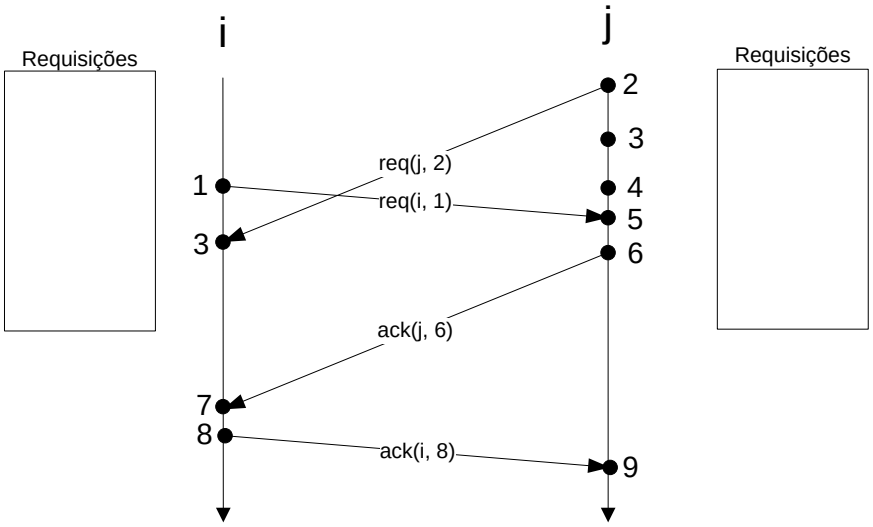


Figura 24 - Algoritmo Ricard & Agrawala 81 - i termina de acessar o recuso no tempo 7 e envia ACK para j.

Por fim j acessa o recurso, pois acaba de receber mensagem de ACK de i. (Não há necessidade de j enviar mensagem para mais ninguém quando ele terminar de usar o recurso, pois não há nenhuma requisição pendente em sua lista de requisições).

Neste algoritmo a exclusão **mútua é garantida** sem **deadlock** ou **starvation**. Além disso, não há ponto único de falha nessa solução.

Complexidade de mensagens: 2(n-1) (Para cada solicitação de acesso ao recurso são enviados dois tipos de mensagens: {(n-1) requisições} + {(n-1) ACKs} → 2(n-1))

Ao contrário de um algoritmo onde há um processo centralizado que gerencia o acesso ao recurso, no algoritmo de "Ricart e Agrawala 81" o ponto único de falha agora é substituído por *n* pontos de falha.

Quando algum processo falha, ele pára de responder às requisições. Essa falta de resposta pode ser interpretada como uma negação de permissão, bloqueando o acesso ao recurso por outros processos. Esse problema pode ser contornando adicionando respostas, inclusive de negação de permissão, a cada mensagem de requisição. Uma falha de entrega agora pode ser detectada por *timeout*.

Outro problema que acaba existindo com o uso do algoritmo distribuído é a necessidade de uma primitiva de comunicação em grupo. Cada processo agora precisa manter uma lista dos membros do grupo, percebendo entradas, saídas e falhas desses mebros.

Dessa forma, na versão destribuída do algoritmo, todos os processos estão envolvidos em todas as decisões de entrada na região crítica.

Correção: Este algoritmo utiliza o conceito de **relógios lógicos** para dar ordem aos eventos. Isso garante parte da correção. **Deadlock** não ocorre pois para que haja um ciclo entre processos, nenhum processo enviou ACK, isto é, vão enviar depois, pois desejam usar o recurso. Mas a decisão "adiar" ou "enviar ACK" é baseado no **tempo lógico**, logo, não é possível ocorrer deadlock. O mesmo ocorre para **starvation**, em algum momento um pedido será o mais antigo e terá a sua vez de ser executado.

6.3.3. Algoritmo Carvalho & Rocairol

Este algoritmo é considerado uma melhoria do algoritmo Ricart & Agrawala 81.

Este algoritmo segue basicamente o mesmo princípio do algoritmo "Ricart & Agrawala 81", no entanto, é realizada uma modificação simples que vai permiti-lo economizar muitas mensagens em algumas situações.

Essa modificação, embora pequena, muda drasticamente o comportamento do algoritmo:

- Processos não têm conhecimento global da rede (os processos não sabem quais serão os próximos processos que vão acessar um recurso).
- Tempo lógico é utilizado para decidir apenas sobre as mensagens que um processo conhece. Pode haver outras mensagens ainda em tráfego e que não foram recebidas que podem ter um ID menor do que um processo que já está acessando um recurso. Por isso o tempo lógico não é global para garantir a exclusão mútua.

No algoritmo de "Ricart e Agrawala 81", a cada vez que um processo quer utilizar um recurso, ele faz broadcast de uma requisição, e só depois de receber uma mensagem ACK de todos é que ele está autorizado a utilizar o recurso.

Figura 25 - Vetor local onde o processo P armazena as autorizações recebidas dos outros processos da rede.

Com base neste detalhe, "Carvalho & Rocairol" sugeriram que cada processo tenha um vetor local com n posições para armazenar um valor booleano que corresponde se o processo possui ou não autorização do processo correspondente (inicialmente cada posição estará preenchida com "false"). A cada ACK recebido de cada processo, a posição correspondente no vetor é modificada para "true", isto é, o processo atual possui autorização do processo P_i para utilizar o recurso.

Na Figura 25 é exibido o exemplo da fila de autorizações do processo P . Este processo requisitou acesso a um recurso e já recebeu uma mensagem de ACK do processo "k". Note que a posição de k no vetor foi atualizada para true. Assim que P receber ACKs de todos os processos, e seu vetor possuir apenas valores "true", então ele terá acesso ao recurso.

A característica de primeiramente receber ACKs de todos os processos para então acessar o recurso é idêntica ao algoritmo "Ricart & Agrawala 81", no entanto, guardar essas informações para serem utilizadas no futuro é o diferencial deste algoritmo.

A grande "sacada" deste algoritmo é a percepção de que se P possui autorização de todos os outros processos, então elas continuam valendo, mesmo que sejam muito antigas. Se a lista de autorizações de P só possui "trues", então ela não precisa solicitar acesso ao recurso para todos, pois desde a última vez que o processo P acessou o recurso, ninguém mais pediu acesso, e solicitar autorização para todos novamente é desnecessário (P pode acessar o recurso quantas vezes quiser, sem ter que enviar nova mensagem de requisição, enquanto nenhum outro processo enviar mensagem solicitando acesso ao recurso).

Caso algum processo tenha pedido autorização para P, então ele marca "false" na sua lista de autorizações e envia um ACK para o processo. Da próxima vez em que ele quiser utilizar um recurso, ele vai enviar mensagem de requisição apenas para os processos que ele não tem mais autorização, isto é, para os processos que estão marcados como "false" em sua lista de autorizações.

Exemplo: Na Figura 26 é exibido o estado inicial da lista de requisições pendentes e das listas de autorizações. Note que cada processo tem autorização dele próprio para acessar o recurso, então, a linha correspondente ao próprio processo na lista de autorizações é sempre true.

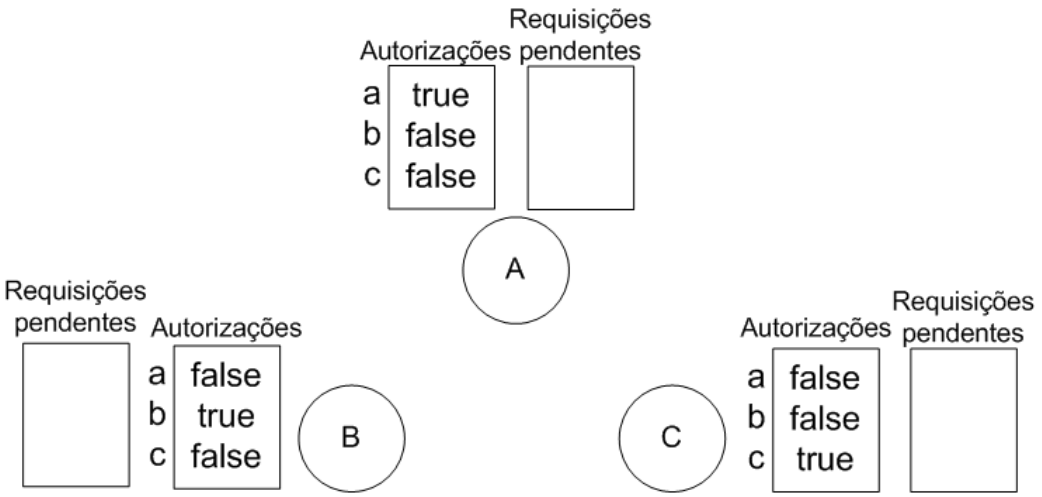


Figura 26 - Algoritmo Carvalho & Rocairol

Neste exemplo, imagine que o processo A solicita o uso do recurso, logo, uma mensagem de requisição contendo o tempo lógico de A é enviada para todos os processos da rede. Como nenhum outro processo quer acessar o recurso, não há requisições pendentes a serem enfileiradas, então os processos B e C respondem uma mensagem de ACK para o processo A.

Como o processo A recebeu autorização de todos os outros processos, ele atualiza sua *fila de autorizações* para "true" para todos os processos. Em seguida, ele pode acessar o recurso tranquilamente, com a garantia de **exclusão mútua**. Esse processo pode ser visualizado na Figura 27.

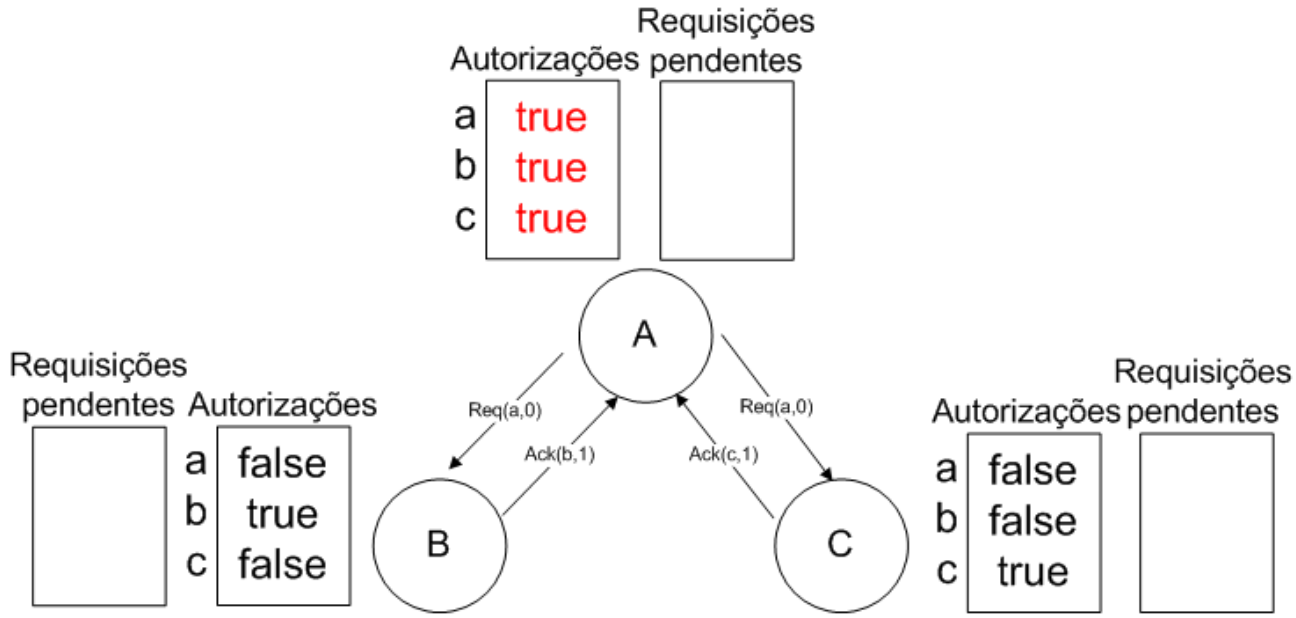


Figura 27 - Algoritmo Carvalho & Rocairol - Processo A solicita acesso a um recurso

Se o processo A, ao terminar de acessar o recurso quiser utilizá-lo novamente, ele poderá fazê-lo sem enviar mensagens requisitando autorizações, pois como ele foi o último processo a acessar o recurso e como a lista de autorizações dele só possui valores "true", então ele pode acessar o recurso sem avisar nenhum outro processo.

Complexidade de mensagens: 0 a 2(n-1) – A complexidade é idêntica ao algoritmo de "Ricart & Agrowala 81", porém, em situações em que um processo já possui autorizações e quer utilizar novamente o recurso, ele pode fazer isso sem ter que enviar qualquer mensagem.

Obs.: Neste algoritmo, os processos não sabem quem está utilizando um recurso. Em situações em que um processo já tinha autorizações de todos, não é possível prever quantas vezes um processo utilizou o recurso, isto é, processos não conhecem estado global da rede (Isto significa que o conhecimento não é simétrico).

6.3.4. Algoritmo Ricart & Agrawala 83

A idéia principal deste algoritmo é permitir que somente o processo que tem a posse do **token** é que possa acessar o recurso.

- Solução não usa tempo lógico.

- Cada processo mantém uma lista local de n posições, a qual vai armazenar o valor dos contadores de cada processo da rede. Cada valor não tem relação alguma com os valores dos outros processos da rede.

- Inicialmente, todas as posições de todos os vetores são "0" (Figura 28).

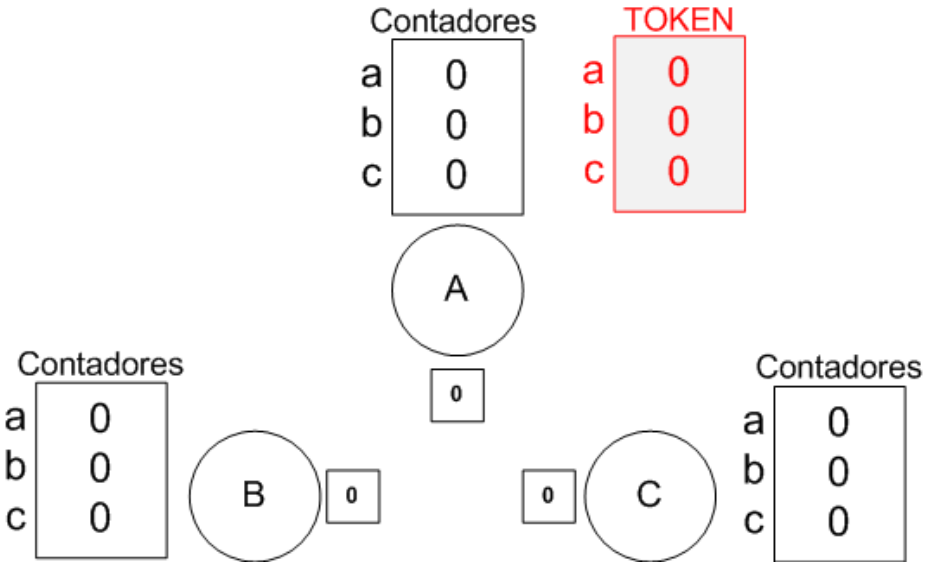


Figura 28 - Algoritmo Ricart & Agrawala 83 - Processos inicialmente possuem contadores com valores 0

- O token leva consigo um vetor, que também possui n posições, as quais armazena os valores dos contadores de cada processo. Estes valores não estão em total sincronia com os vetores de cada processo, pois, baseado na diferença entre os valores do vetor do token e os valores dos vetores de cada processo é que é definida a ordem dos processos que vão acessar o recurso.

- Devido ao fato do token levar consigo o valor dos contadores de cada processo, há um ponto negativo que é o tamanho maior das mensagens de token.

- O token começa em algum processo.

- Quando algum processo quer usar o recurso, faz broadcast de sua requisição e incrementa em 1 o seu contador local.

- Em algum momento, dependendo do tempo que as mensagens demoram para percorrer a rede, cada nó recebe a mensagem de requisição e atualiza em seu vetor local o valor do contador do processo que solicitou acesso ao recurso.

- O processo que está com o token está acessando o recurso. Se durante esse período o processo recebe alguma requisição, ele atualiza o valor do contador correspondente na sua fila local, mas **não** atualiza o token que está com ele.

- Ao terminar de usar o recurso, o vetor local é "varrido" sequencialmente (a partir da posição do processo atual) até ser encontrada uma posição do token que possui valor diferente da lista local do processo atual (A diferença entre o valor da posição P_x do vetor local e seu valor no token, nunca é superior a 1. É claro! Isso ocorre porque cada processo solicita acesso ao recurso uma vez até acessá-lo, e isso incrementa o contador em 1).

- Uma posição P_x do token só é atualizada assim que ele visitar P_x.

Exemplo: Imagine uma rede composta por 3 processos, A, B e C. Inicialmente o token está no processo A, o qual está utilizando o recurso. Em determinado momento C deseja acessar o recurso. Para isso, C incrementa seu contador, atualiza seu valor em sua lista local e então faz broadcast de uma mensagem de requisição. Ao receber a mensagem de requisição, A e B atualizam o valor do contador de C (Figura 29).

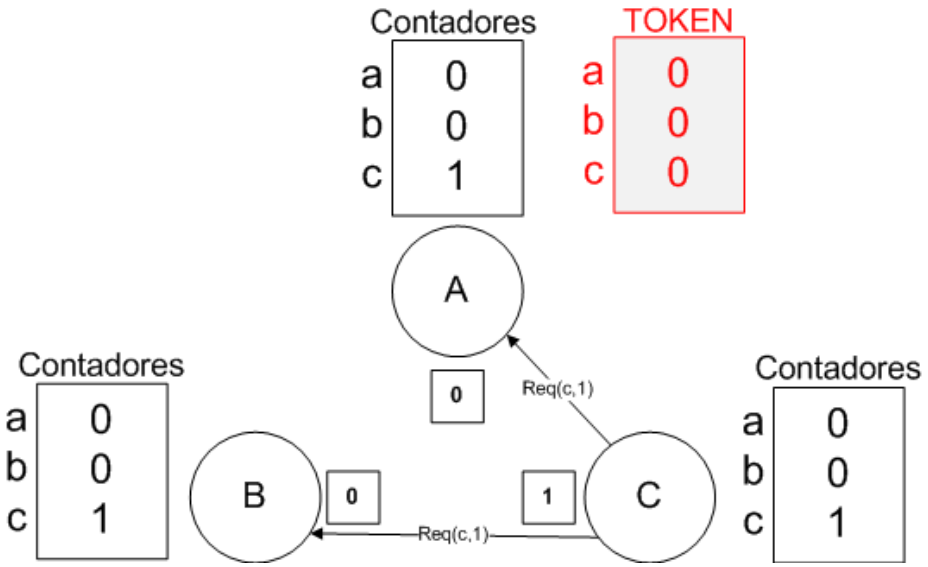


Figura 29 - Algoritmo Ricart & Agrawala 83 - Processo C solicita acesso ao recurso.

Assim que A deixa de utilizar o recurso, ele verifica, no vetor local, qual é o próximo valor depois dele que está diferente do valor do token. Pela ordem alfabética deste exemplo, A verifica se o valor de B em seu vetor local é diferente do valor do token. Como os valores são idênticos, A vai verificar o próximo valor, que é C. Os valores do vetor local e do token que correspondem ao processo C estão diferentes, então, o token é enviado para o processo C, o qual vai poder utilizar o recurso e então atualizar o seu valor o token, conforme Figura 30 (Obs: somente o processo que está com o token pode atualizar o seu valor nele).

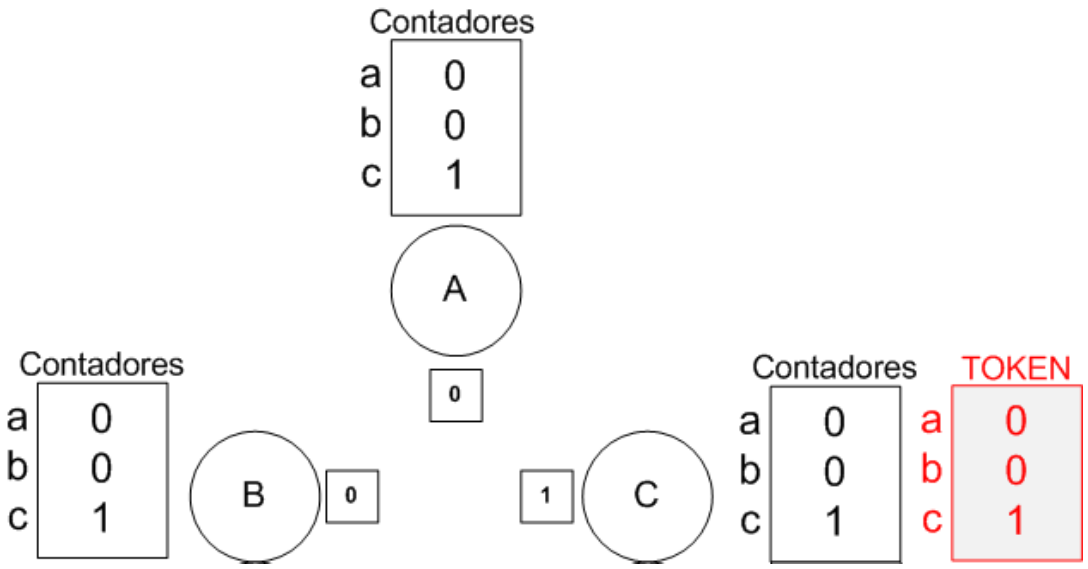


Figura 30 - Algoritmo Ricart & Agrawala 83 - Processo C ganha o token, acessa o recurso e atualiza o valor do seu contador no token.

Complexidade: N mensagens →{(n-1) mensagens de requisições} +{ 1 mensagem do token} = n. Mas, o tamanho da mensagem para enviar o token é maior.

É importante destacar que os valores dos contadores não são utilizados para ordenar os pedidos, mas sim para simplesmente verificar se o processo deseja utilizar o recurso, independente do tempo do pedido.

Corretude: evita **starvation**, pois se as mensagens são entregues dentro de um tempo finito, e conseqüentemente os vetores locais são atualizados, então mais cedo ou mais tarde todos os processos vão ter sua vez de acesso. A **exclusão mútua** também é garantida, uma vez que somente o processo que possui o token é que pode ter acesso ao recurso.

6.3.5. Algoritmo de Maekawa

- O conceito de **Planos Finitos Projetivos** garante a exclusão mútua**:**

Planos Finitos Projetivos: Este conceito é aplicado em conjuntos que possuem um número primo de elementos, logo, este algoritmo é inviável para ser utilizado na prática, é utilizado apenas para foco em estudo. Dado um conjunto S de n elementos | n é primo, então, serão criados n subconjuntos de limite superior de "raiz quadrada de n" elementos. Ex: Se temos um conjunto de 7 elementos, então serão criados 7 subconjuntos de 3 elementos cada.

Cada um desses subconjuntos possui elementos não repetidos, e se comparado esse subconjunto com qualquer outro subconjunto, haverá exatamente um valor em comum. Isso é que garante a exclusão mútua, a seguir será explicado o porquê.

- Todos os processos conhecem todos os outros processos.

- É utilizado tempo lógico para comparar as prioridades das requisições conhecidas por um processo, no entanto não há uma noção de tempo lógico global, apenas tempo lógico para comparar as prioridades das mensagens recebidas.

- Cada processo tem uma “ficha” que é enviada para os processos que desejam ter acesso ao recurso. Essa ficha é devolvida para seu dono assim que o processo libera o recurso.

- Cada processo armazena uma variável local **locked** onde é guardado o ID do processo ao qual ele emprestou a ficha temporariamente.

- Cada processo tem uma fila local chamada **waiting**, onde é armazenado o processo ao qual foi emprestada a ficha, e também são armazenados os processos que pediram autorização e que ainda não a receberam.

- Quando um processo deseja acessar um recurso, é escolhido um dos subconjuntos (normalmente é escolhido o subconjunto que já possui o valor do processo que deseja acessar o recurso), e então são enviadas mensagens de requisição para os processos que fazem parte deste subconjunto.

- Se todos os processos que receberam a mensagem de requisição puderem autorizar este processo acessar o recurso, então elas enviam suas fichas de **Lock** para esses processos. Para realizar a autorização, um processo precisa estar com a posse da sua “ficha”, caso não esteja, é porque ela está emprestada. Nesta situação, o processo armazena a requisição pendente em **waiting**, e assim que receber a ficha de volta, manda a ficha para o processo que tiver menor tempo lógico na sua fila de **waiting**.

- Para cada mensagem de requisição recebida por um processo P, então P armazena o ID do processo que solicitou permissão na sua fila **waiting**.

- Para cada ficha recebida, um processo incrementa a sua variável **locks**.

- A cada **ficha de Lock** que o processo recebe, ele incrementa uma variável local **locks** que armazena todas as autorizações lock que ele já recebeu. Assim que o processo receber **locks** de todos os processos do subconjunto que ele escolheu, então ele pode acessar o recurso (neste caso a variável locks vai ser igual ao número de elementos do subconjunto).

- Assim que ele terminar de acessar o recurso, um processo devolve as fichas para os respectivos donos, utilizando uma mensagem de **Release.** Em seguida, decrementa sua variável **lock.**

- Cada processo que recebe sua ficha de volta através de mensagens de release exclui o ID do processo que lhe devolveu a ficha da sua fila **waiting.**

Seis tipos de mensagens são utilizadas nesse algoritmo:

- Request: o processo solicita acesso ao recurso. Junto com essa mensagem também é enviado o tempo lógico do processo.

- Enquire: Se um processo A enviou uma mensagem de lock para B, isto é, A concedeu uma ficha para B acessar o recurso, porém, pouco tempo depois A recebe uma requisição de C para acessar o recurso com um **tempo lógico** menor que B. Então A percebe que C teria prioridade sobre B, então A envia para mensagem de **enquire** para B, solicitando que ele devolva a ficha, mesmo que ele ainda não tenha acessado o recurso. Caso ele esteja acessando o recurso, ele deve terminar de acessar normalmente, e só depois devolver a ficha.

- Relinquish: é a mensagem de retorno do Enquire. Se um processo recebe uma mensagem de Enquire, então ele devolve a ficha (lock) com uma mensagem de relinquish.

- Lock: É a mensagem onde um processo envia sua ficha dizendo para um processo a autorização dele é concedida.

- Release: Mensagem que é enviada quando um processo termina de acessar um recurso. Essa mensagem é enviada para os donos da ficha (lock) que o processo contêm.

- Fail: Se um processo B pede autorização para A, e a ficha de A neste momento está com um outro processo P que está acessando o recurso, então A envia uma mensagem de **fail** para B, e coloca ele em sua fila **waiting.**

Complexidade de mensagens:

- **Melhor caso: √request, √lock, √release** (Imagine que há uma rede composta por 7 processos, um número primo de processos. Cada subconjunto criado possuirá 3 elementos, isto é, √7 elementos, que arredondando dará 3. Então, para cada mensagem enviada de request, por exemplo, será enviada um mensagem para cada elemento do subconjunto, que é igual a √n = √7 = 3. O número três é o limite superior de √7)

- **Pior caso: √request, √enquire, √relinquish, √lock, √release, √lock** (Ex: Processo A requisita acesso (*request)* para B, mas a ficha de B está com C. Porém, B verifica que o tempo lógico de A é menor e verifica que A tem prioridade, logo, B solicita a ficha de C de volta(*enquire*). Ao receber de volta (*relinquish*), B envia a ficha dele para A (*lock*). Ao terminar de usar, A envia de volta para B (*release)*, que então, agora, deixa C usar, enviando-a novamente para ele(*lock*).)

6.3.6. Algoritmo de Agrawal & Abadi

- Todos os processos conhecem todos os processos

- Todos os processos possuem uma mesma árvore composta por todos os processos da rede. (esta árvore é criada antes de iniciar o algoritmo. A forma como ela é criada não é importante aqui, apena tenha em mente que cada vértice desta árvore é um processo da rede.

- Quando um processo deseja acessar o recurso, ele escolhe um caminho qualquer da árvore, que vá desde a folha até a raiz (qualquer caminho mesmo! Mas... é escolhido normalmente um caminho onde o próprio nó está inserido, assim ele não precisa pedir autorização para ele mesmo).

- O processo deve receber autorização de todos os processos que fazem parte deste caminho para então poder acessar o recurso.

- É flexível: caso haja falha em algum nó da árvore, é possível escolher outro caminho que não tenha falhas para pedir autorizações.

- O número de autorizações é equivalente à altura da árvore.

Se houver falhas:

- Imagine a situação onde um processo seleciona um caminho na árvore e começa a requisitar autorizações, e nesse mesmo período um dos nós desse caminho falha! Então, deve ser escolhido um caminho para cada

um dos filhos desse nó que falhou, e também devem ser feitas requisições para esses novos caminhos, conforme Figura 31.

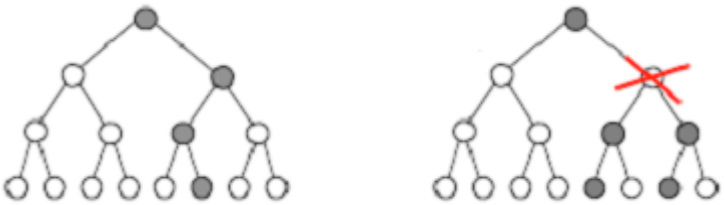


Figura 31 - À esquerda é exibida a árvore original. À direita é exibida a árvore após uma falha e quais os caminhos necessitam ser utilizados para garantir a exclusão mútua (Figura extraída da internet - fonte desconhecida).

Complexidade de mensagens: O(log n)

O número de mensagens só aumenta em caso de falhas

Desvantagem: Quanto mais próximo da raiz, mais um processo é requisitado.

6.4. Terminação: Detecção de terminação para algoritmos de difusão

(Conteúdo deste capítulo pode ser encontrado no capítulo 19.1 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

- Algoritmo é chamado de ***algoritmo de difusão*** quando todas as atividades têm início em um único processo onde um *input* externo ocorre, e então esta atividade se ***difunde*** pela rede através de troca de mensagens.

- **Objetivo:** Dado um algoritmo A, saber quando sua computação terminou, isto é, saber quando todos os processos estão em estado quiescente e garantir que não há nenhuma mensagem trafegando nos canais. Para isso, é utilizado um algoritmo de monitoração B, que deve verificar quando a computação do algoritmo A terminou. Para informar que A terminou a computação, o algoritmo B retorna ***done***.

*Terminação é uma **propriedade estável:** Uma vez que um algoritmo terminou a computação, seu estado perdura para sempre, isto é, nada pode fazer com que ele reinicie a computação, pois já alcançou um estado final.*

Cenário

- Rede é um grafo conexo e não direcionado.

- Inicialmente todos os processos estão em estado *quiescente*, isto é, estão em um estado onde não há ações locais habilitadas e eles podem apenas receber mensagens (Processos não estão processando nada, e nenhuma mensagem está trafegando nos canais. A única ação possível neste estado é um evento externo enviar uma mensagem para algum processo e através dela ser iniciada uma nova atividade.)

- No início o ambiente envia uma única mensagem de *input* a um único processo i. O processo i é então chamado de ***source***.

- O processo i acorda, inicia a computação e envia mensagem para outros processos.

- Outros processos acordam quando recebem mensagens.

- Assim que todos os processos terminarem a computação e não houver mensagens trafegando nos canais, isto é, quando o algoritmo A alcançar o estado quiescente, então o processo i (i é o processo onde a computação iniciou, isto é, i é ***source***) retorna uma mensagem ***done*** informando que a computação terminou.

A seguir é descrito o algoritmo de **DijkstraScholten** que é capaz de detectar quando ocorre a terminação de um algoritmo, conforme citado no cenário acima.

6.4.1. Algoritmo DijkstraScholten

Complexidade de comunicação: 2m (m = número de mensagens enviadas pelo algoritmo A, mais uma mensagem de ACK para cada mensagem, logo, 2m)

Complexidade de tempo: O(m($t_{execução}$ + $t_{entrega\_mensagem}$)) → ($t_{execução}$ é o limite superior para execução de uma tarefa e $t_{entrega\_mensagem}$ é o limite superior para entrega de mensagem)

Neste algoritmo é construída uma **árvore geradora** com os nós ativos no algoritmo A. Esta árvore geradora possui como nó raiz àquele que iniciou a computação, isto é, o nó ***source*** (o nó source é aquele que recebeu a primeira mensagem do ambiente e então deu iniciou a toda computação). Na Figura 32 é exibido um exemplo onde um processo A recebeu uma mensagem de *input* do ambiente e iniciou a computação. Em seguida foram trocadas mensagens com os processos B, C e D, os quais se tornaram ativos. Esses nós ativos, destacados em vermelho, são os nós que compõem a árvore geradora utilizada pelo algoritmo de terminação de DijkstraScholten (Os nós E e F continuam em estado quiescente).

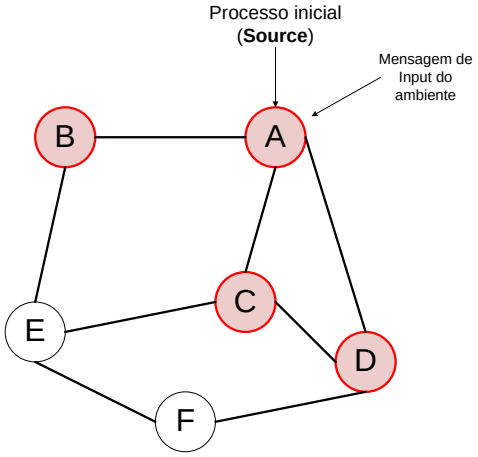


Figura 32 – Execução do algoritmo A - Árvore geradora composta pelos nós ativos do algoritmo A.

Conforme os processos trocam mensagens, novos processos se tornam ativos, e outros se tornam novamente quiescentes. Logo, a árvore geradora utilizada pelo algoritmo de terminação é **modificada em tempo de execução** e pode aumentar e diminuir de tamanho inúmeras vezes antes do fim da execução do algoritmo A. Na Figura 33 é exibido outro momento da execução do algoritmo A. Note que o nó ***source*** continua sendo A, e que agora A, D, E e F estão ativos e fazem parte da árvore geradora. Os nós B e C não fazem parte, pois estão em estado quiescente.

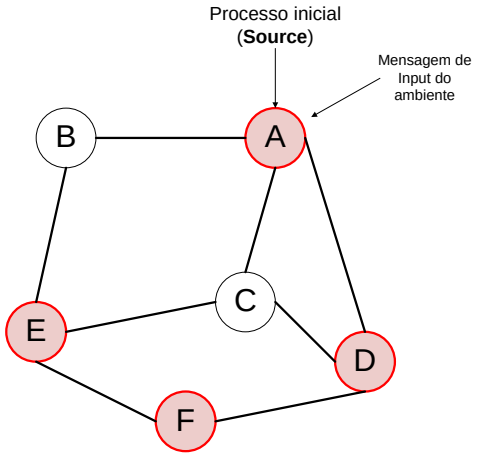


Figura 33 - Execução do algoritmo A - Árvore geradora, analisada em momento diferente ao da Figura 32, composta pelos nós ativos da mesma execução do algoritmo A.

Funcionamento detalhado do algoritmo DijkstraScholten

O algoritmo DijkstraScholten (DS) é executado com base na execução de um algoritmo qualquer, A. O algoritmo DS(A) se baseia no mesmo grafo utilizado em A. Qualquer ação de entrada de DS(A) pode alterar somente DS(A), e nunca o algoritmo original A.

"Dado um algoritmo A, o algoritmo DS(A) para o monitoramento da terminação de A trabalha em paralelo a A, nunca interferindo em seu funcionamento".

No algoritmo para detecção de terminação DijkstraScholten, as mensagens utilizadas são as mensagens do algoritmo A mais uma mensagem de ACK (as mensagens de A são tratadas como as mensagens de *search* no algoritmo *ASynchSpanningTree*). Todos os processos, com exceção do *source*, definem como seu pai da árvore geradora o processo que lhe enviou uma mensagem primeiramente. No exemplo da Figura 34 o processo A envia uma mensagem para os processos B e D. Como B e D ainda não haviam recebido nenhuma mensagem, então A é o processo pai de B e D na árvore geradora criada.

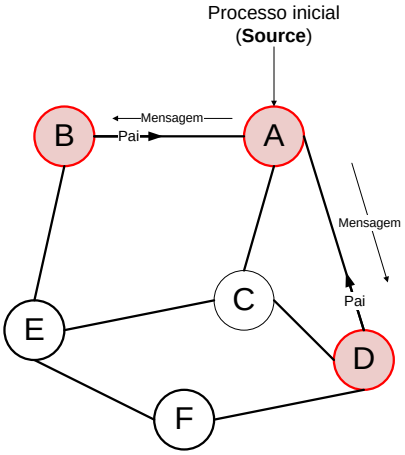


Figura 34 - Algoritmo DijkstraScholten - Processos B e D definem o processo A, que lhe enviou primeira mensagem, como seu pai na árvore geradora.

Para todas as outras mensagens recebidas, são enviadas mensagens de ACK (Somente para a primeira mensagem recebida por um processo que não é enviado ACK, por enquanto. A primeira mensagem recebida é utilizada para definir quem é o pai do processo na árvore geradora). A seguir são exemplificados os passos e estados ocorridos até o momento em que o algoritmo A se torna quiescente.

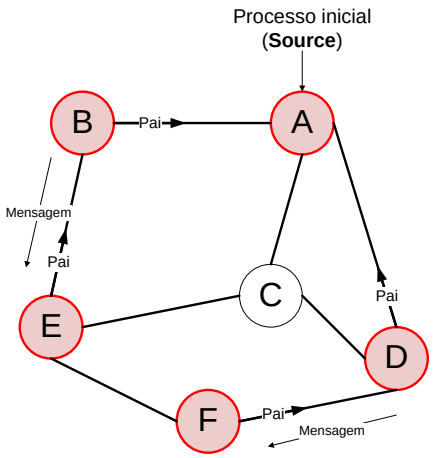


Figura 35

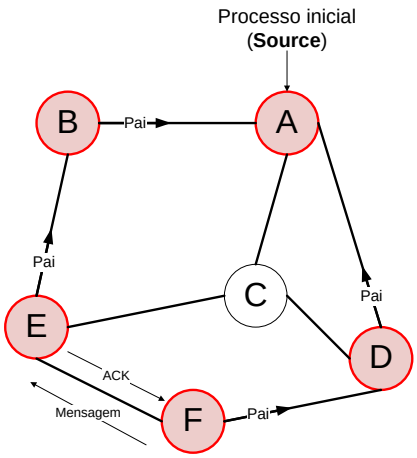


Figura 36

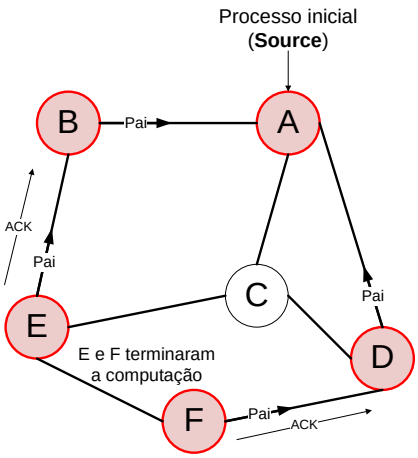


Figura 37

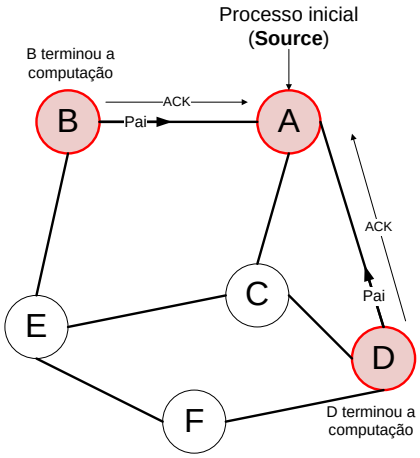


Figura 38

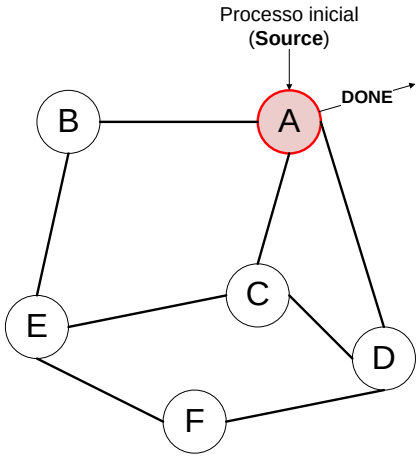


Figura 39

Na Figura 35 os processos B e D enviam mensagens para E e F. Como os processos E e F ainda não haviam recebido nenhuma mensagem, então eles definem o processo que lhe enviou a primeira mensagem como seu pai. Logo, B é pai de E, e D é pai de F (note que não foram enviados ACKs, pois ACKs **não** são enviados para a primeira mensagem que recebem). Note que na Figura 35, a árvore geradora é composta pelos processos A, B, D, E e F, os quais não estão mais em estado quiescente, isto é, os processos que receberam mensagens e que estão ativos. Na Figura 36, F envia mensagem para E. Como E já havia recebido mensagem anteriormente, então ele responde prontamente para F uma mensagem de ACK.

Na Figura 37 os processos E e F terminam a computação. Como ambos se tornaram quiescentes e todas as mensagens que eles enviaram já foram respondidas com ACKs, então eles enviam uma mensagem de ACK para seus pais (Recordando, quando os processos E e F receberam a primeira mensagem e saíram do estado quiescente, eles não enviaram ACK para seus pais naquele momento. Agora é que uma mensagem de ACK é enviada para os pais de E e F). Em seguida, a árvore geradora é modificada, e os processos E e F não fazem mais parte dela, isto é, a árvore geradora foi "encolhida" (*Shrink*).

Na Figura 38 os processos B e D terminam a computação, entram em estado quiescente, constatam que já receberam ACKs para todas as mensagens que enviaram e então enviam a única mensagem de ACK que faltava: um ACK para seus pais. Em seguida a árvore geradora é "encolhida" (*Shrink*). Na Figura 39 é exibido que todos os processos envolvidos na execução do algoritmo A se tornam quiescentes, por esse motivo a

árvore geradora é encolhida até a origem da árvore, o processo *source*, A. Em seguida, o processo A se torna quiescente. Todas as mensagens que A enviou foram respondidas com ACK, logo, DS(A) retorna uma mensagem ***DONE*** para informar que a computação terminou.

É importante destacar que, com exceção da primeira mensagem que o ambiente envia para o processo source que inicia a computação, para todas as outras mensagens que o processo source receber, ele também envia mensagens de ACK. Source também define seu pai para a primeira mensagem que receber (primeira mensagem que não seja a primeira enviada pelo ambiente), assim como ocorre com os outros processos.

6.5. Snapshot global consistente – Detecção de estado global consistente

(Conteúdo deste capítulo pode ser encontrado no capítulo 19.2 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

Problema: Tirar "fotografias" (*Snapshots*) locais de cada processo e formar um ***estado global consistente***. No entanto, para alcançar este objetivo é necessário que as fotografias sejam sincronizadas.

Ex: Processos são agências bancárias. Cada banco mantém uma conta local. Não há depósitos, apenas há transferência de dinheiro entre os bancos (processos) através de troca de mensagens. Em determinado momento é desejado "fotografar" todos os bancos e ter a noção exata do montante que cada um possui. Então, além da dificuldade em fotografar o estado local de cada banco para depois uni-los e gerar um único estado global, há o problema em lidar com as mensagens de transferência de dinheiro que estão em trânsito.

Cenário

- Snapshot global consistente visa detectar um estado global do sistema.

- Estado global do sistema é considerado ***propriedade estável.***

- Rede é um grafo conexo e não direcionado.

- Canais são FIFO (*first in first out*)

- Modelo é assíncrono.

Uma possível solução para este problema seria utilizar relógio físico para combinar o tempo em que os processos fotografariam seus estados. No entanto há o problema de sincronização de relógio a ser considerado, o delta de atraso entre os relógios, o tempo da entrega das mensagens, e seria necessário os processos pararem de mandar mensagens instantes antes do tempo combinado para realizar o snapshot, a fim de evitar que mensagens estejam trafegando no tempo combinado.

> Como é possível verificar, utilizar relógio físico não é uma boa abordagem para solucionar o problema de snapshot global consistente. Uma possível solução é utilizar ***tempo lógico***, como no algoritmo *LogicalTimeSnapshot*, descrito a seguir.

6.5.1. Algoritmo LogicalTimeSnapshot (utiliza tempo lógico)

(O algoritmo *LogicalTimeSnapshot* pode ser encontrado no capítulo 18.3.2 do livro Distributed Algorithms Nancy Lynch, Morgan Kaufmann, 1996)

Complexidade de comunicação: 0 → *não há troca de adicional de mensagens*

Complexidade de tempo: infinito → *é preciso tempo infinito para garantir que também haja troca infinita de mensagens, de modo que cada processo receba pelo menos uma mensagem de cada processo com tempo de envio maior que o tempo t pré-determinado para o snapshot.*

- Ao utilizar relógio lógico para solucionar o problema de detecção de estado global, não é necessário se preocupar com o problema de sincronismo de relógios, assim como na utilização de relógios físicos (pois o uso de relógio lógico já garante um tempo lógico global e por isso o sincronismo é implícito).

- Nesta solução, deve ser pré-determinado no algoritmo qual será o tempo lógico em que o snapshot será realizado por cada processo. Por exemplo, imagine que no tempo lógico 7,5 todos os processos devem realizar um snapshot global consistente (Esta pré-determinação evita o problema de consenso entre os processos para definirem o tempo em que o snapshot vai ser realizado).

- Outra premissa é que haja um número infinito de troca de mensagens, pois caso o tempo pré-determinado para o snapshot seja igual a 1000, deve haver pelo menos 1000 mensagens trocadas para que o tempo lógico alcance este valor (ignorando os eventos locais) e para que os processos tenham o conhecimento do maior tempo lógico envolvido no sistema.

- Um problema a ser resolvido é computar as mensagens que estavam transitando nos canais no momento do snapshot (No instante do snapshot, uma mensagem que trafega no canal ainda não pertence nem ao processo de origem, nem ao de destino).

- Esta solução funciona da seguinte forma: Quando um processo **recebe** ou **envia** uma mensagem que possui tempo lógico maior ao tempo pré-determinado para a realização do snapshot, então o processo armazena o seu estado local (se o tempo lógico recebido da mensagem for maior do que o tempo pré-determinado para o snapshot, então o processo realiza o snapshot antes de computar a mensagem, para não gerar inconsistência).

Exemplo: Na Figura 40 são exibidos 3 processos, P1, P2 e P3. Cada um, analogamente ao exemplo dos bancos introduzido no início desta seção, possui um montante armazenado no início da computação. P1 possui $10, P2 possui $20 e P3 possui $30. O soma total de dinheiro de todos os processos no início da computação é **$60**.

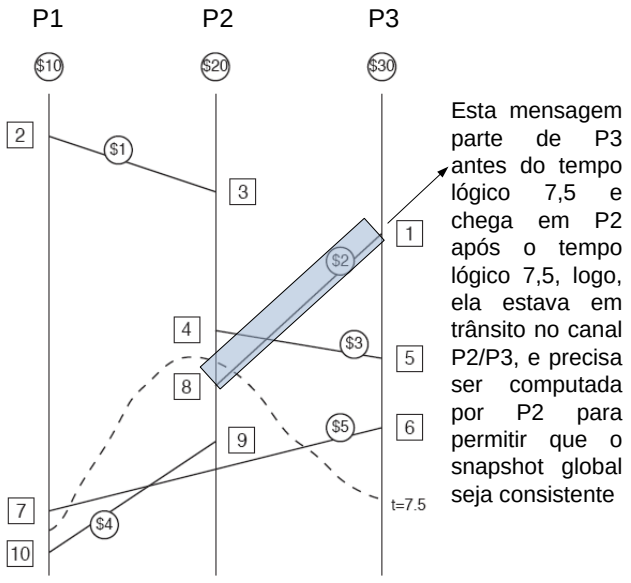


Figura 40 - Detecção de estado global consistente através do uso de relógio lógico

Como foi estipulado que o tempo para realização do snapshot é t=7,5, então todos os processos devem armazenar o seu estado neste instante. Analisando P1, no tempo lógico 2 ele envia $1 para P2 e no tempo lógico 7 recebe de P3 $5. A próxima mensagem recebida por P1 possui tempo lógico 10, então, P1 percebe

que o tempo lógico 7,5 já aconteceu. Logo, P1 realiza um snapshot de seu estado entre os as mensagens recebidas com tempo lógico 7 e 10, pois o tempo lógico 7,5 está entre esses dois valores. Computando o montante de dinheiro que P1 possui no tempo entre 7 e 10, temos que P1=$14.

A mesma análise pode ser feita em P3. Não está representado na figura, no entanto é previsto o ponto onde P3 tem conhecimento do tempo lógico 7,5 (através da linha pontilhada), e então P3 realiza um snapshot local. P3 inicia com $30, transfere $2 para P2 e recebe $3 de P2. Em seguida transfere $5 para P1. Logo, no tempo lógico 7,5 é possível verificar que P3 possui $26.

Por último analisemos P2. P2 inicia com $20 e recebe $1 de P1. Em seu tempo lógico 4 P2 transfere $3 para P3. Em seguida P2 recebe uma mensagem de P3 com $2, no tempo lógico 8. Então P2 percebe que o tempo lógico 7,5 já aconteceu. Logo, P2 realiza um snapshot de seu estado entre os as mensagens recebidas com tempo lógico 4 e 8, pois o tempo lógico 7,5 está entre esses dois valores. Computando os valores enviados e recebidos por P2, temos no tempo 7,5 que P2=$18.

Somando os valores de P1, P2 e P3 nos snapshots tirados por cada processo, temos:

P1=$14 + P2=$18 + P3=$26 → **$58**

Através da soma destes valores, podemos perceber que há algo errado. Inicialmente a soma de todos os valores contidos por todos os processos totalizava $60. Como não ouve adição nem subtração de valores, houve apenas transferência, então a soma deveria ainda totalizar $60, e não $58. ***Por que?***

Isto ocorreu por conta de uma característica da última mensagem recebida por P2 no tempo lógico 8, a qual envolve uma situação extremamente importante nesta solução que envolve tempo lógico. A mensagem enviada por P3 a P2 (com valor $2) saiu de P3 no tempo lógico 1 e chegou em P2 no tempo lógico 8. O tempo lógico pré-determinado para realizar o snapshot (t=7,5) está entre esses dois valores:

1 < 7,5 < 8

Então, é possível P2 verificar que no momento em que ele realizou o snapshot, havia uma mensagem em trânsito no canal e que não foi computada, logo, essa mensagem deve ser computada também por P2. Então, levando em consideração a mensagem que estava em trânsito no canal entre P3 e P2 durante o tempo lógico 7,5, temos o estado global consistente:

P1=$14

P2=$18 **+ $2**

P3=$26

Soma: $60 ← Ok

→ **Terminação:** Cada processo realiza um snapshot local quando receber uma mensagem com tempo lógico maior ou igual ao tempo pré-determinado para realizar o snapshot (t=7,5 neste exemplo). Para ter certeza que não há nenhuma mensagem em trânsito, é necessário que cada processo receba uma mensagem de cada outro processo com tempo de envio **maior** que o tempo pré-determinado para o snapshot. Depois de receber uma mensagem de todos os outros processos com tempo maior que o tempo pré-determinado t, todos os processos devem enviar seus snapshots para um mestre/líder de modo que ele possa uni-los e criar

um snapshot global consistente. Para que o ambiente saiba que a computação chegou ao final, deve ser utilizado um **algoritmo de monitoração** (algoritmo de terminação), como por exemplo, *DijkstraScholten*, conforme descrito na seção 6.4.1.

Desvantagem: Embora este algoritmo não gaste nenhuma mensagem adicional, é necessário um número infinito de mensagens para garantir que vai existir o tempo lógico "t" pré-determinado para realizar o snapshot.

6.5.2. Algoritmo ChandyLamport (sem uso de tempo lógico)

Complexidade de mensagens: (2|E|) = O(|E|)
Complexidade de tempo: O(diâmetro($t_{execução}$ + $t_{entrega\_mensagem}$)) - ($t_{execução}$ é o limite superior para execução de uma tarefa e $t_{entrega\_mensagem}$ é o limite superior para entrega de mensagem)

- ChandyLamport é similar ao algoritmo LogicalTimeSnapshot, no entanto, **não** utiliza tempo lógico para realizar a detecção de estado global consistente (snapshot global consistente).
- Não necessita de um tempo pré-estipulado no algoritmo para realizar o snapshot. Snapshot pode ser retirado a qualquer momento.

- O algoritmo realiza o envio de uma mensagem especial (***marker***) para "varrer e limpar" os canais. Dessa forma é possível verificar quais mensagens estavam em trânsito durante o snapshot.

- O snapshot global consistente (detecção do estado global) é realizado por um ***algoritmo de monitoração*** B(A), como por exemplo, *DijkstraScholten*, conforme descrito na seção 6.4.1.

- Canais são FIFO e bidirecionais.

Exemplo:

- O ambiente envia uma mensagem *Snap* para um processo P_A, solicitando capturar o estado global consistente. Este processo imediatamente armazena seu estado atual e envia mensagens MARKER para todos seus vizinhos nos canais de saída. A mensagem MARKER é também enviada com o propósito de limpar o canal, isto é, de permitir verificar quais as mensagens que estavam em trânsito e que necessitam ser computadas no snapshot. A Figura 41 exibe um canal sendo "varrido" pela mensagem MARKER.

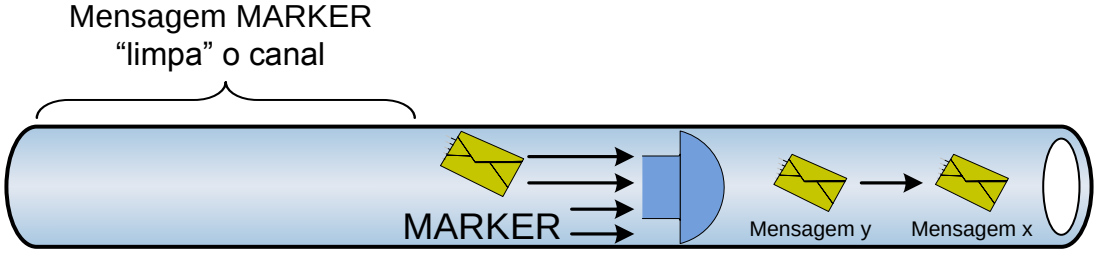


Figura 41 - Mensagem MARKER "limpa" os canais.

Assim que P_A envia MARKER para seus vizinhos, ele monitora todas as mensagens que chegam para ele por cada canal. Esse monitoramento é realizado em cada canal até o recebimento de uma mensagem MARKER. Isto quer dizer que, todas as mensagens que chegaram para P_A entre a mensagem MARKER que ele enviou e a mensagem MARKER que ele recebeu pelo mesmo canal, são mensagens que estavam em trânsito enquanto os snapshots foram realizados, então essas mensagens necessitam ser computadas por P_A para que o snapshot global seja consistente.

Quando um *processo que ainda não recebeu nenhuma mensagem MARKER* receber uma, ele marca o canal por onde recebeu a mensagem como vazio. Em seguida este processo imediatamente armazena seu estado atual (snapshot) e envia mensagens MARKER para todos seus vizinhos nos canais de saída. Em seguida este processo monitora todas as mensagens que receber por cada canal. Esse monitoramento é realizado em cada canal até o recebimento de uma mensagem MARKER. Esse processo é repetido sucessivamente. Por fim, quando um processo termina a computação (enviou e recebeu mensagens MARKER por todos seus canais), então o processo retorna uma mensagem *report* para o ambiente. A mensagem *report* de cada processo contém seu *snapshot* local. Assim que todos os processos retornam *report* para o ambiente, um processo central, por exemplo, pode unir todos os *reports* de cada processo a fim de gerar o *snapshot global consistente.*

Aplicações:

Pode ser utilizado para:

- Detectar deadlocks

- Realizar depuração (tarefa muito difícil)

- Realizar checkpoint

6.6. Detecção de Deadlock

O objetivo desta seção é apresentar algoritmo distribuído para detecção de deadlock em modelo assíncrono.

Cenário

- Não há nó central

- O tempo de entrega das mensagens é finito

- Canais são FIFO (first in first out)

Existem dois tipos de deadlock

- Deadlock de comunicação (Communication Model): chamado de modelo OR, pois um processo pode prosseguir se conseguir APENAS UM dos recursos que solicitou (por exemplo, RPC: remote procedure call)
- Deadlock de recurso (Resource Model): chamado de modelo AND, pois um processo pode prosseguir apenas se conseguir TODOS os recursos que solicitou.

6.6.1. Algoritmo Chandy, Misra e Haas

Os algoritmos de detecção de deadlock propostos por Chandy, Misra e Haas enfocam os dois modelos citados. A seguir os algoritmos para cada modelo são detalhados separadamente.

6.6.1.1. Resource Model

Complexidade de mensagens: O(|E|) do grafo de dependência

Complexidade de tempo: O(diâmetro($t_{execução}$ + $t_{entrega\_mensagem}$)) *do grafo da rede*

→ Visão geral

- Modelo baseado em recursos.

- Um processo não pode executar ao menos que ele tenha acesso a TODOS os recursos que esteja precisando.

- É dito que um grupo de processos está em deadlock quando nenhum processo do grupo possa executar porque cada processo requer um recurso que está em posse de algum outro processo do grupo.

- Cada processo possui um controlador local, C_j.

- Quando processos em uma transação T_i necessitam acessar um recurso administrado pelo controlador C_m, eles se comunicam com o processo P_{im} que é responsável por fazer a requisição para seu controlador C_m.

- Um processo só pode prosseguir sua computação depois que conseguir acesso a TODOS os recursos que ele solicitou. (Por este motivo o *resource model* é chamado AND, pois ele precisa de um consentimento positivo de todos os processos aos quais solicitou acesso ao recurso. No *communication model* um processo só precisa receber mensagem de pelo menos um processo para poder prosseguir, por isso é chamado de modelo OR).

- Um processo está em estado IDLE quando ele está aguardando por um recurso.

- Um processo está em EXECUÇÃO quando ele não está no estado IDLE.

Logo, se um processo NUNCA adquirir um recurso que solicitou, ele ficará permanentemente IDLE.

Ex: $P_a \rightarrow P_b \rightarrow P_c \rightarrow P_D$

É dito que P_a é dependente de P_D se P_a está em estado IDLE e P_a depende de um recurso alocado por P_b. Por outro lado, P_b também está em estado IDLE e está aguardando um recurso alocado por P_c, e P_c está aguardando um recurso que está alocado para P_D. Então é possível deduzir que P_A SEMPRE estará em estado IDLE enquanto P_D também estiver IDLE e aguardando um recurso.

Logo, P_A é DEPENDENTE de P_D.

- P_A estará em deadlock se ele for dependente dele próprio, ou se for dependente de um processo que é dependente dele (P_A). Neste caso, deadlock existe somente se houver um CICLO de processos IDLE, onde cada um é dependente do próximo processo no CICLO.

ALGORITMO

Existem 2 componentes básicos no algoritmo para o modelo de recursos:

1 – Mensagem **PROBE(i, j, k)** – *Probe significa "sonda"*

- i = P_i é o processo quem enviou originalmente a sonda. P_i está aguardando um recurso que está com P_j.
- j = P_j é o processo atual que está repassando a mensagem PROBE "para frente".
- k = P_k é o processo destino que vai receber a mensagem.

2 – Array local **DEPENDENT$_k$(i)=**boolean

- Cada posição do array representa um processo da rede. Se um processo P_K sabe que P_i é dependente dele, então a posição correspondente a P_i no vetor é TRUE, deste modo, todos os processos tem ciência dos processos que são dependentes dele. (*este array é criado para que em algum momento no futuro possa ser utilizado por algum outro algoritmo que resolva o deadlock. Para descobrir se o deadlock existe, somente a sonda PROBE é suficiente*).

- O algoritmo é iniciado por algum processo que entra em estado IDLE e deseja verificar se ele está em deadlock. (A questão de quando um processo demonstra o desejo de saber se ele está em deadlock não é levado em consideração aqui. Poderia ser, por exemplo, após um processo ficar IDLE após x milisegundos.)

- Este processo envia sonda PROBE(i,j,k) para P_k quando:

- P_i está aguardando um recurso em posse de P_j.
- P_j está aguardando um recurso em posse de P_k.

- A sonda pode ser aceita ou descartada por P_k. Ela será aceita se:

- P_k estiver em estado IDLE, isto é, bloqueado e aguardando um recurso.

- $DEPENDENT_k[i]$=FALSE, isto é, P_k não sabia que P_i era dependente dele.
- P_k deve resposta a P_j, isto é, se não respondeu favoravelmente a todas as requisições.

- Quando um processo P_k aceita a sonda PROBE, ele atualiza a sua variável $DEPENDENT_k[i]$=TRUE, e repassa a sonda PROBE para frente, isto é, para os processos aos quais ele é dependente e está aguardando algum recurso.

- Quando um processo receber uma sonda PROBE gerada por ele mesmo, então ele sabe que existe DEADLOCK!

Exemplo:

Suponha que haja uma rede formada pelos processos i, j e k. Em determinado momento o processo i entra em estado IDLE. Neste exemplo i está aguardando a liberação de um recurso que está sendo utilizado por j, e por este motivo i não pode prosseguir sua computação. Então, o processo i (em um determinado momento que não será tratado aqui) envia uma sonda PROBE para j a fim de verificar se ele (i) está em deadlock. Na Figura 42 é exibida a mensagem originada por i, e enviada por i com destino a j.

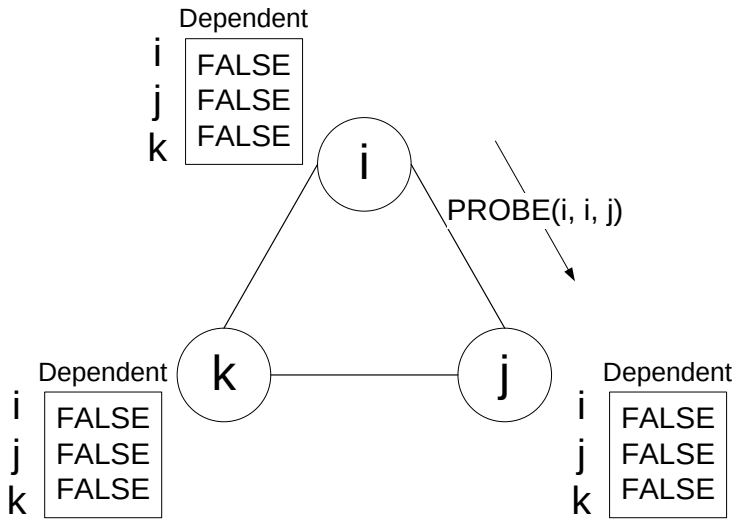


Figura 42 - Sonda PROBE iniciada pelo processo i, enviada por i com destino ao processo j.

Ao receber a mensagem PROBE, j pode aceitá-la ou não. O processo j vai aceitá-la se ele também estiver bloqueado e aguardando a liberação de algum recurso. Como neste exemplo j está aguardando um recurso utilizado por k, então j atualiza sua lista de dependentes, muda o status da posição i da lista para TRUE, e envia uma sonda PROBE para os processos aos quais ele está aguardando liberação de recursos (neste exemplo ele aguarda apenas k liberar o recurso.). Na Figura 43 é exibida a sonda PROBE iniciada por i ser enviada por j com destino a k. Note também que a lista *dependent* do processo j teve o status atualizado para TRUE na posição i, pois agora o processo j sabe que i é dependente dele.

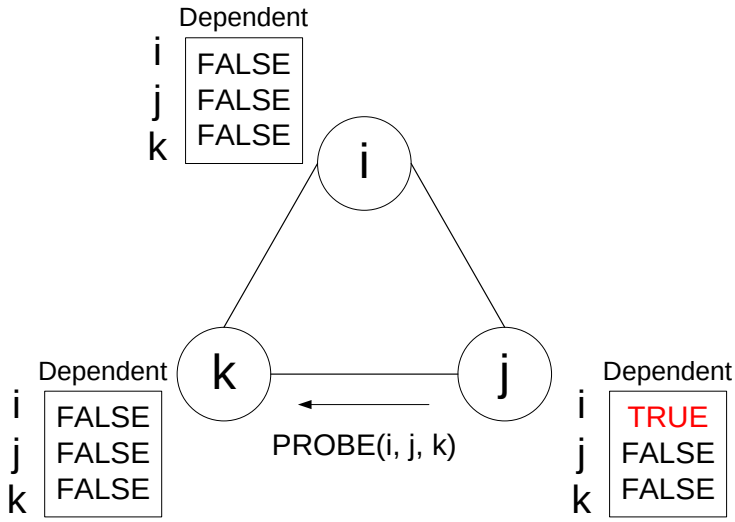


Figura 43 - Processo j envia sonda PROBE iniciada por i para k.

Se neste exemplo o processo j não estive em estado IDLE, isto é, não estivesse aguardando algum processo liberar um determinado recurso para j continuar a computação, então j não aceitaria a mensagem PROBE de i, pois como j está em processamento, então quer dizer que o deadlock não existe, e por isso não há motivo para passar a sonda PROBE para frente.

O processo k também aceita a sonda PROBE, pois ele também está em estado IDLE e aguardando a liberação de um recurso que está alocado para o processo i. Ao aceitar a sonda, k atualiza sua lista de *dependent*, pois agora ele sabe que se o processo i originou a sonda, então i é dependente dele, logo, k atualiza sua lista *dependent* para TRUE na posição i. Em seguida, k encaminha a sonda PROBE para o processo ao qual ele está aguardando a liberação do recurso, o processo i, conforme Figura 44.

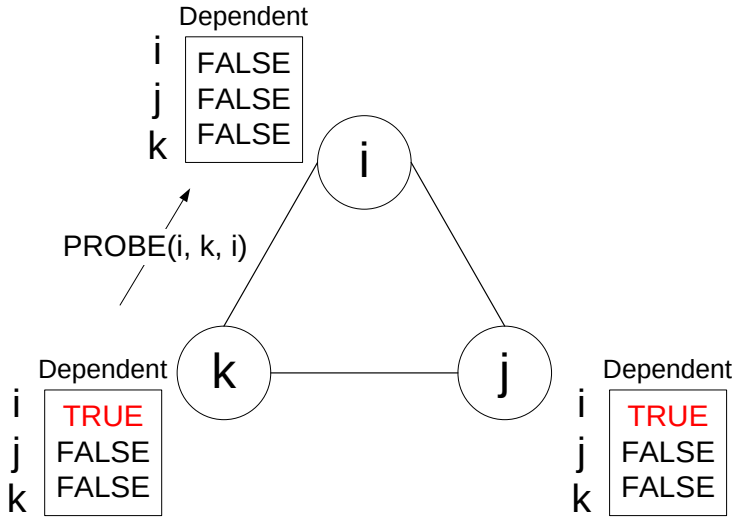


Figura 44 - Processo k envia sonda PROBE originada por i para o processo i.

Como o processo i está em estado IDLE e aguardando um recurso ser liberado por outro processo (j), então i aceita a sonda. Em seguida i atualiza sua lista *dependent*, conforme Figura 45.

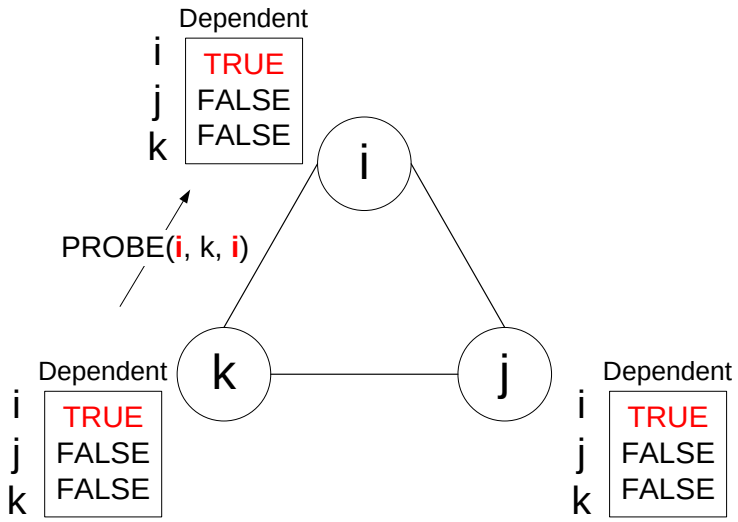


Figura 45 - Processo i recebe uma sonda PROBE originada por ele próprio. Então i detecta que ele é dependente dele mesmo e então há DEADLOCK!

Como i recebe uma sonda PROBE(i, k, i), ele sabe que não só quem originou a sonda, mas também quem a encaminhou são dependentes dele. Então ele atualiza as posições i e k do seu vetor *dependent*. Em seguida, i percebe que ele é dependente dele próprio, então ele sabe que a computação está em **DEADLOCK!**

"Um vetor dependent$_k$[i]=TRUE, apenas se P_i é dependente de P_k e P_k está bloqueado"

Quando um processo recebe uma sonda PROBE que foi originada por ele próprio, então há DEADLOCK!

Obs: Como cada processo que está bloqueado vai eventualmente iniciar o algoritmo de detecção de deadlock paralelamente, então o array Dependent vai sendo atualizado com todos os processos que iniciaram a computação. Logo, no futuro, após detectado que o deadlock existe, todos os processos terão um array atualizado então um algoritmo para solucionar o deadlock pode ser utilizado com base no array.

6.6.1.2. Communication Model

Complexidade de mensagens: 2|E| = O(|E|) *do grafo de dependência*

Complexidade de tempo: O(diâmetro($t_{execução}$ + $t_{entrega\_mensagem}$)) *do grafo da rede*

→ Visão geral

- Um processo está em deadlock neste modelo, se e somente se houver um ***KNOT*** no **grafo de dependência**.

- Definição de ***KNOT:*** um nó é KNOT quando TODOS os vértices que ele alcança, também alcançam ele.

- Este modelo é uma descrição resumida de uma rede de processos que se ***comunicam por mensagens***.

- Ao contrário do *resource model*, aqui não há *controllers* explícitos (controladores). Isso é tratado de forma transparente, pois os *controllers* são simulados pelos processos. (Mesmo no *resource model*, *c*ada processo possui o seu próprio controlador embutido, então tudo é visto como um só processo).

- Não há recursos neste modelo.

- Requisições por recursos, cancelamentos e liberações devem ser implementadas por **mensagens**.

- É assumido que não há qualquer tipo de falha.

- Todo processo em estado IDLE possui a ele associado um grupo de processos chamado DEPENDENT SET.

- A lista DEPENDENT SET (conjunto de dependência) de cada processo armazena os processos que um processo P está aguardando para poder continuar a computação.

- Um processo IDLE pode continuar a sua computação quando ele receber uma mensagem de QUALQUER processo que esteja na sua lista DEPENDENT SET.

- Um processo entra em estado TERMINADO se ele está IDLE e se sua lista DEPENDENT SET está vazia.

- Um grupo de processos S está deadlocked se todos os processos de S estão *permanentemente* IDLE.

- Um processo está *permanentemente* IDLE se ele NUNCA receber uma mensgem de qualquer processo de sua lista DEPENDENT SET.

- Ex: Se um processo A aguarda B, e B está executando um loop demorado, então A não está *permanentemente* IDLE, já que B irá mandar uma mensagem para A eventualmente no futuro. (Já que, por mais que demore, o loop uma hora vai terminar!)

→ Deadlock ocorre no *communication model* quando:

1 - Todos os processos em S estão IDLE;

2 - O grupo DEPENDENT SET de todos os processos em S são um subgrupo de S;

3 - Não há mensagens em trânsito entre os processos em S.

Um grupo S de processos que satisfaçam as 3 condições citadas devem permanecer IDLE permanentemente, pois:

- Um processo IDLE P_i em S só pode iniciar a execução depois de receber uma mensagem de algum processo P_j de sua lista DEPENDENT SET;
- Todo processo P_j na lista DEPENDENT SET de P_i também está em S e não pode enviar mensagens enquanto permanecer IDLE;
- Não há mensagens em trânsito entre P_i e P_j, o que quer dizer que P_i nunca receberá uma mensagem de qualquer processo da sua lista DEPENDENT SET.

"Um processo está em deadlock se ele pertence a algum grupo que está em deadlock"

ALGORITMO

Existem 2 tipos de mensagens utilizadas pelo algoritmo para o modelo de comunicação, mais 5 variáveis locais:

1 – Mensagens **QUERY(i, m, j, k)** e **REPLY(i, m, j, k)**

- i: P_i é o processo que iniciou o a ***query computation*** para verificar se existe deadlock. P_i é também chamado de INITIATOR, pois foi ele quem iniciou a *query computation*.
- m: é um *sequence number* que representa quantas vezes P_i iniciou uma *query computation.*
- j: P_j é o processo atual que está repassando a mensagem "para frente" (SENDER).
- k: P_k é o processo destino que vai receber a mensagem (RECEIVER).

3 – Cinco variáveis locais:

- **DEPENDENT SET**: variável que armazena um conjunto de processos dos quais um processo P_i espera mensagens.
- **LATEST(i):** representa o maior *sequence number* m em qualquer QUERY(i, m, j, k) enviado ou recebido por P_k. Inicialmente LATEST(i)=0, para todo i.
- **ENGAGER(i)**: é o ID do processo que originou LATEST(i), isto é, é o ID de um processo **INITIATOR**.
- **NUM(i)**: é a diferença entre o número de QUERY enviados e REPLY recebidos por P_i. Se NUM(i)=0, quer dizer que P recebeu REPLY para todas as mensagens QUERY que enviou.
- **WAIT(i)**: é TRUE se P é IDLE desde LATEST(i). Inicialmente WAIT(i) é falso para todo i.

- O algoritmo é iniciado por algum processo que entra em estado IDLE e deseja verificar se ele está em deadlock. (A questão de quando um processo demonstra o desejo de saber se ele está em deadlock não é levado em consideração aqui. Poderia ser, por exemplo, após um processo ficar IDLE após x milisegundos.)

- Um processo P_i que se encontra bloqueado (IDLE) inicia uma ***query computation***, enviando mensagens QUERY(i, m, j,k) para todos os processos P_k do seu conjunto de dependência (DEPENDENT SET).

- Todo processo IDLE que recebe uma mensagem QUERY vai propagá-la para os processos que estão nos seus respectivos conjuntos de dependências (DEPENDENT SET), caso ainda não a tenha feito. Então, se existir uma seqüência de processos bloqueados (IDLE) P_a, P_b, P_c e P_d, tal que cada processo na seqüência (exceto o primeiro) está no conjunto de dependência (DEPENDENT SET) do processo anterior, então uma mensagem de QUERY vai ser propagada de P_a até P_d.

Ao receber uma mensagem QUERY de P_i, um processo P_k (P_k se encontra **BLOQUEADO**, isto é, **WAIT(k)=TRUE**) pode estar em uma das três situações descritas a seguir (com base nas variáveis LATEST e "m" (*sequence number*)):

	M < LATEST(i)	M = LATEST(i)	M > LATEST(i)
Descrição	Se a mensagem enviada por P_i tem *sequence number* MENOR ao maior *sequence number* que P_k já viu.	Se a mensagem enviada por P_i tem *sequence number* IGUAL ao maior *sequence number* que P_k já viu.	Se a mensagem enviada por P_i tem *sequence number* MAIOR ao maior *sequence number* que P_k já viu.
Ação	**NADA**	**ENVIA UM REPLY IMEDIATAMENTE**	**PROPAGA A QUERY PARA TODOS QUE ESTAO NA SUA LISTA *DEPENDENT SET* E RESPONDE SOMENTE QUANDO RECEBER TODOS OS REPLIES PARA AS QUERIES QUE PROPAGOU**

*Se um processo **INITIATOR** (que iniciou a **query computation**) recebe uma mensagem REPLY para cada mensagem QUERY que enviou, então este processo está em **DEADLOCK!***

Recorde que se um processo receber uma mensagem de apenas um processo ao qual ele estava esperando, então não há deadlock, pois o communication model é baseado no modelo OR, isto é, não há necessidade de se receber mensagem de todos os processos que ele esteja aguardando para poder continuar a computação.

Exemplo:

- Suponhamos que há uma rede composta por quatro processos, A, B, C e D

- Suponhamos também que os processos A, B e C se encontram bloqueados (IDLE). Então, em um momento aleatório A deseja saber se ele está em deadlock e envia uma mensagem QUERY para os processos que ele aguarda alguma mensagem (os processos aos quais ele é dependente), isto é, para os processos que se encontram na sua lista DEPENDENT SET. Na Figura 46 são exibidos os processos e as mensagens QUERY enviadas por A.

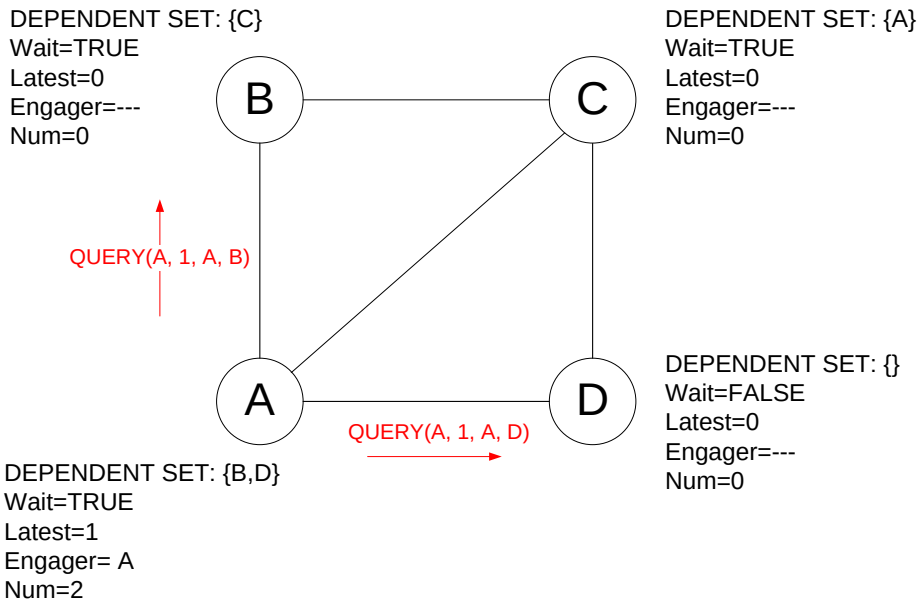


Figura 46 - Processo A envia mensagem QUERY para B e D.

Analisando as variáveis de cada processo, podemos ver que A é dependente de B e D, isto é, o processo A aguarda receber alguma mensagem de A .OR. D para continuar sua computação. A variável WAIT de A é TRUE, pois A se encontra bloqueado. LATEST=1, pois é o maior *sequence number* que A já viu (neste exemplo, 1 é o próprio sequence number gerado por A). ENGAGER=A, pois a ***query computation*** foi gerada por A. NUM=2 pois A enviou duas mensagens QUERY. (Para cada mensagem REPLY que A receber, é decrementado uma unidade da sua variável NUM).

O processo D analisa a mensagem QUERY recebida. Como sua variável WAIT é FALSE, quer dizer que D não está bloqueado, e por isso, ele simplesmente ignora a mensagem QUERY e não a responde. A única ação feita por D é atualizar sua variável LATEST=1, conforme Figura 47.

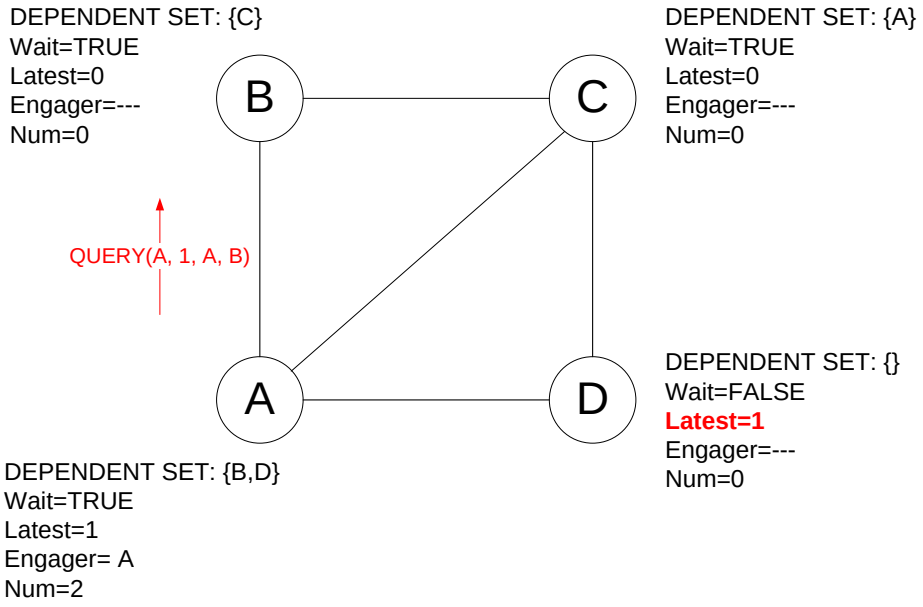


Figura 47 - Processo D não está bloqueado, por isso ele apenas atualiza sua variável LATEST e não envia nenhum REPLY.

Agora vejamos a ação tomada pelo processo B. Ele analisa a mensagem QUERY recebida. Como sua variável WAIT é TRUE, quer dizer que B **está bloqueado**, e por isso**, deve atentar para a variável m da mensagem recebida**. M representa o maior *sequence number* conhecido pelo processo que enviou a mensagem QUERY (A). Como B possui LATEST=0 e M=1, então o *sequence number* da mensagem enviada por A é maior do que o maior *sequence number* conhecido por B. Então B deve enviar mensagens QUERY para TODOS os processos que fazem parte do seu *grupo de dependência* (DEPENTENT SET), neste exemplo, apenas o processo C. Antes de enviar QUERY para C, o processo B deve atualizar suas variáveis locais NUM=1 (pois ele vai enviar 1 mensagem de REPLY), ENGAGER=A (pois A é o INITIATOR da *query computation*) e LATEST=1 (pois 1 é o maior *sequence number* já visto por B), conforme Figura 48.

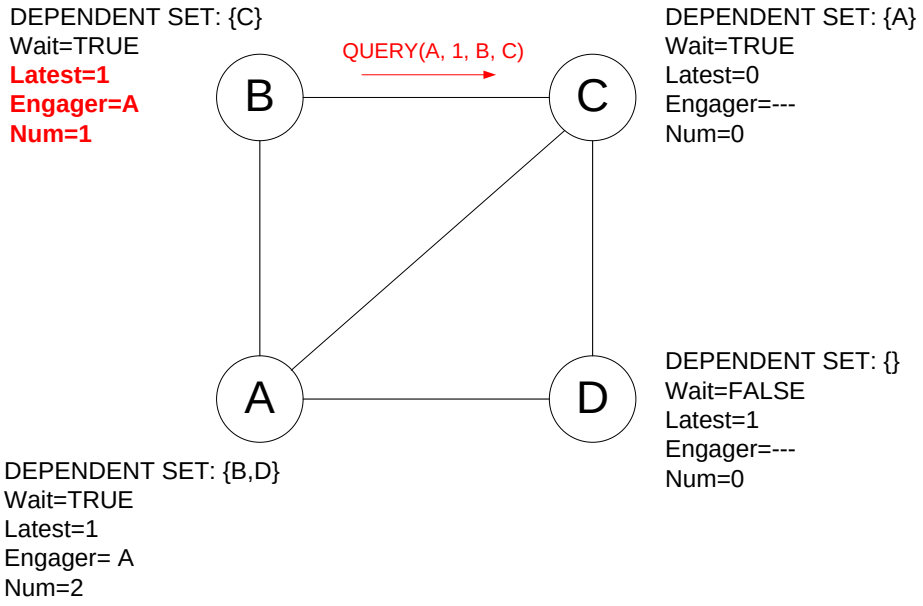


Figura 48 - Processo B atualiza suas variáveis e envia mensagem QUERY para C.

Como o processo C está bloqueado (WAIT=TRUE), quer dizer que B **está bloqueado**, e por isso**, deve atentar para a variável m da mensagem recebida.** Como a mensagem recebida possui m=1 e o processo C possui LATEST=0, então a mensagem QUERY foi gerada com um *sequence number* maior do que ele já conheceu, por isso ele não pode assegurar que não há deadlock e então precisa propagar esta mensagem adiante. Antes de fazer isso C deve atualizar suas variáveis locais, assim como no passo anterior. Na Figura 49 são exibidas as atualizações nas variáveis locais do processo C e a mensagem QUERY enviada de C para A.

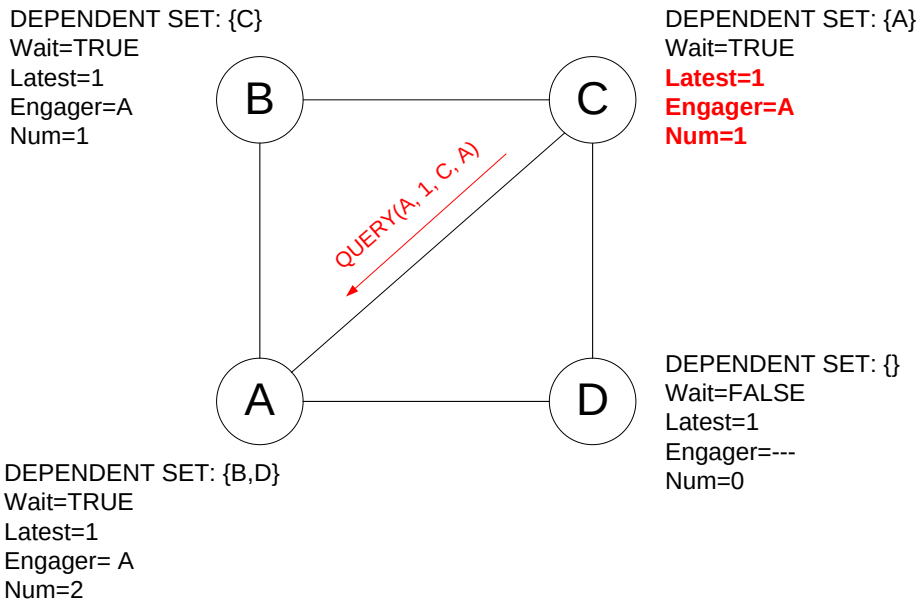


Figura 49 - Processo C atualiza suas variáveis e envia mensagem QUERY para A.

Agora é possível entender a lógica do algoritmo. Por coincidência deste exemplo A é o INITIATOR (quem iniciou a *query computing*), e acabou recebendo uma mensagem QUERY do processo que ele próprio iniciou. Já é possível verificarmos visualmente que há um ciclo de dependências entre os processos. No entanto, vamos continuar a computação do algoritmo. Como o processo A está bloqueado, isto é, WAIT=TRUE e a **variável m=1 da mensagem QUERY recebida é IGUAL a LATEST=1**, então A responde a mensagem QUERY imediatamente através de uma mensagem **REPLY** (neste passo não há alterações a serem feitas em suas variáveis locais), conforme exibido na Figura 50**.**

Obs: A variável NUM do processo A não é decrementada pois isso ocorre APENAS quando A receber mensagens REPLY correspondentes às mensagens QUERY que enviou. O fato do processo A ENVIAR uma mensagem REPLY não afeta o contador NUM.

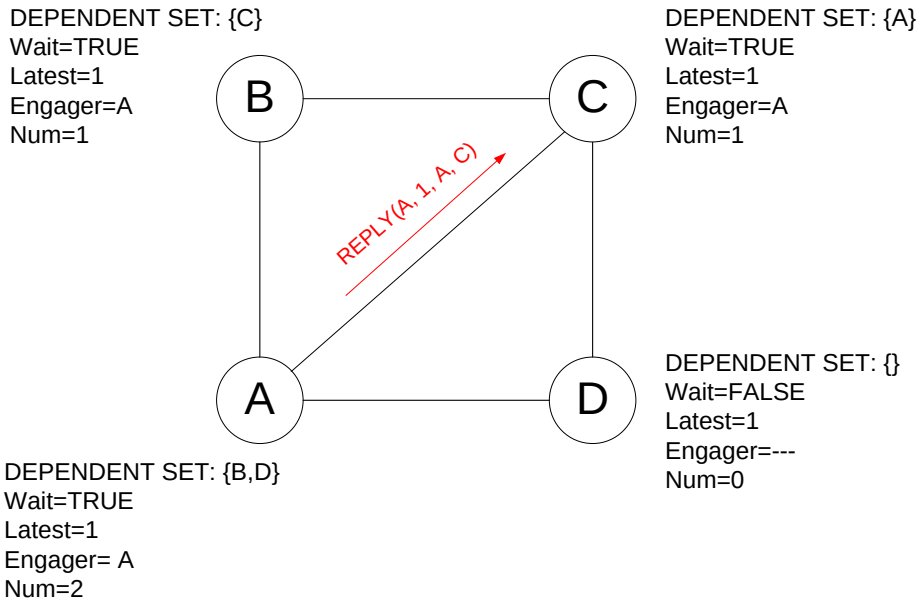


Figura 50 - Processo A responde a mensagem QUERY recebida através do envio de uma mensagem REPLY.

Assim que C recebe a mensagem de REPLY, ele decrementa sua variável NUM. Após o decremento, NUM=0, então quer dizer que C recebeu REPLY para TODAS as mensagens QUERY que enviou, logo, o processo C deve enviar uma mensagem REPLY para B (recordando que foi o processo B que havia enviado uma mensagem QUERY para C), conforme Figura 51.

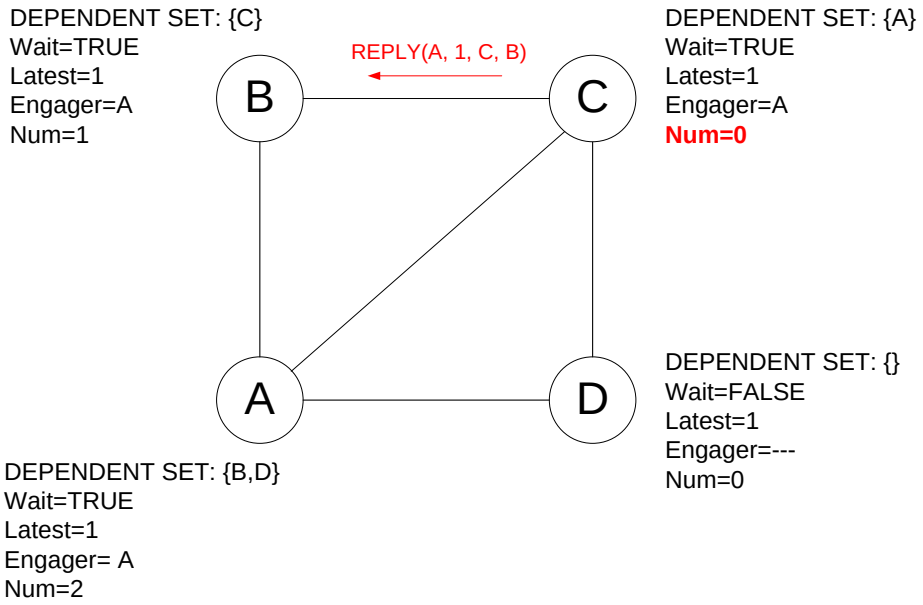


Figura 51 - Processo C envia mensagem REPLY para o processo B.

Agora o processo B recebe a mensagem REPLY. Como sua variável NUM=1, então quer dizer que B estava aguardando apenas uma mensagem de REPLY. Logo, B decrementa NUM, onde NUM agora é igual a 0, e por isso B envia mensagem REPLY para A. O processo B também atualiza suas variáveis locais, conforme Figura 52.

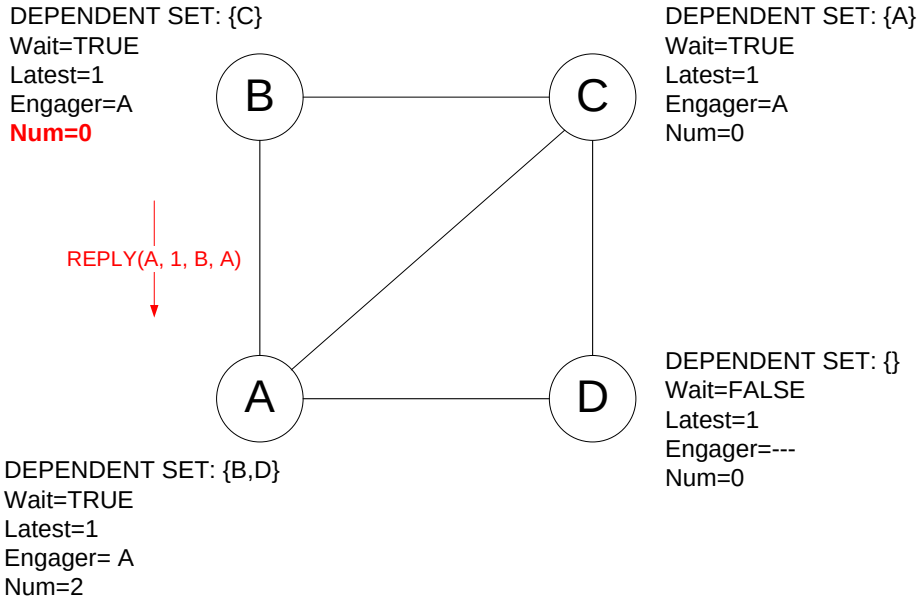


Figura 52 - Processo B envia mensagem REPLY para A.

Agora o processo A recebe a mensagem de REPLY e decrementa em 1 sua variável NUM, logo, agora NUM=1, conforme Figura 53.

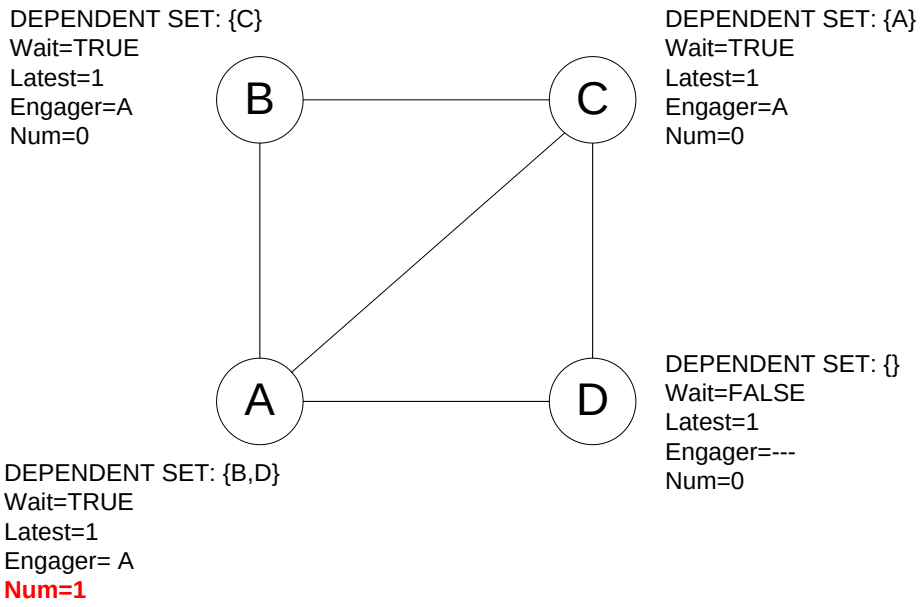


Figura 53 - Processo A decrementa em 1 sua variável local NUM.

CONCLUSÃO

O processo A (INITIATOR) só teria certeza que estaria em DEADLOCK, se e somente se ele recebesse uma mensagem REPLY para cada mensagem QUERY que ele enviou. Analisando a variável local NUM do processo A, podemos ver que ainda há uma mensagem REPLY pendente. Esta mensagem REPLY que está faltando é referente à mensagem QUERY que A enviou para D. Como o processo D não está bloqueado, isto é, WAIT=FALSE, então o processo D ignorou a mensagem QUERY enviada por A e não a respondeu. Logo, este exemplo não caracteriza um DEADLOCK.

Obs: O processo A NUNCA vai saber que ele não está em DEADLOCK. Este algoritmo permite que os processos descubram se eles ESTÃO em deadlock, e não se não estão. Neste exemplo, enquanto o processo A não receber duas mensagens REPLY, ele vai continuar sabendo que ele não está em deadlock.

6.7. REPLICAÇÃO DE DADOS

- O objetivo da replicação de dados é distribuir um mesmo dado em diferentes nós da rede a fim de aumentar a sua disponibilidade, mesmo que alguns nós falhem.

VANTAGENS

- Menor atraso no acesso: o dado pode ser recuperado em um nó próximo.
- Tolerância a falhas: se uma cópia falha, então o dado pode ser recuperado através de alguma cópia em outro nó que ainda esteja ativo.

DESVANTAGENS

- É necessário se preocupar com as versões dos dados salvos em cada nó, a fim de que não haja inconsistências.
- Ocupa mais espaço: cada nó vai armazenar uma cópia fiel do dado.

- Como o modelo adotado é assíncrono, é necessário sincronizar as versões dos dados de cada nó a fim de evitar inconsistências.

OPERAÇÕES SOBRE OS DADOS (possíveis operações que podem ser realizadas com os dados)

- Leitura
- Escrita

COMO GARANTIR CONSISTÊNCIA? Através da ***SERIALIZAÇÃO EM CÓPIA ÚNICA***: Serializar em cópia única significa que mesmo executando várias operações distribuídas sobre uma cópia original de um dado, este dado é replicado em diferentes nós como se todas as cópias fossem um dado só.

- Para garantir a *serialização em cópia única*:

- É assumido que cada cópia do dado que está distribuída em diferentes nós é **carimbada** com um **número de versão**. A cada nova escrita, este número de versão é incrementado, logo, é possível compará-los a fim de encontrar qual cópia é a mais recente.
- Operações de **leitura** devem ser feitas sobre um conjunto de nós (não precisa ser sobre todos) onde pelo menos um deles participou da última operação de **escrita** (intersecção de pelo menos um nó entre a última **escrita** com a atual **leitura**).
- Operações de **escrita** devem ser feitas sobre um conjunto de nós (não precisa ser todos) onde pelo menos um deles também participou da última operação de **escrita** (intersecção de pelo menos um nó entre a última **escrita** com a atual **escrita**).

Para visualizar as situações citadas, na Figura 54 são exibidos os processos de uma rede. Na figura é exibida uma operação de **escrita** sobre o dado. Note que antes da operação de escrita a última versão do dado era a versão 1.0. Após a operação de **escrita** sobre os nós B, D e E, os dados destes nós foram atualizados, e o controle de versão deles incrementado para a versão 2.0.

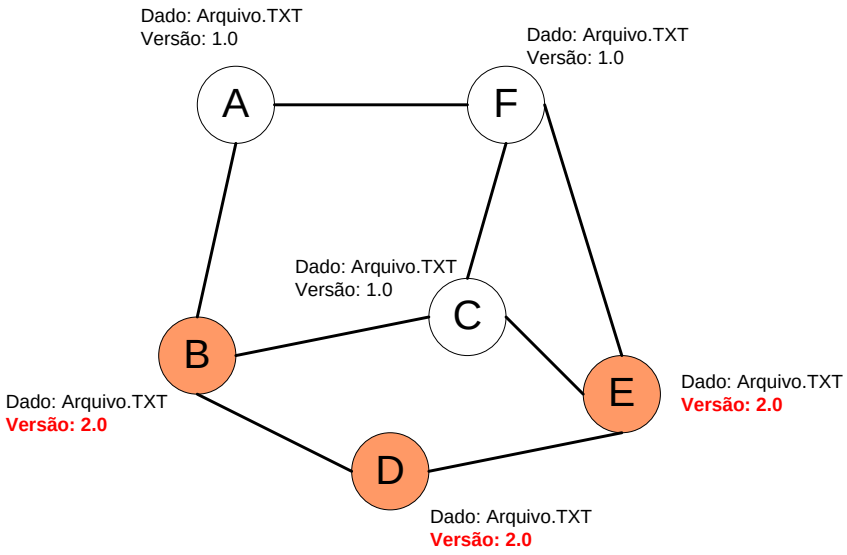


Figura 54 – Operação de escrita sobre o dado foi realizada nos nós B, D e E.

Se a próxima operação for de **leitura** ou **escrita** (não importa), então os nós que vão participar da próxima operação devem ter pelo menos um nó em comum com os nós envolvidos na **última escrita** (intersecção com B ou D ou E), conforme exemplo na Figura 55.

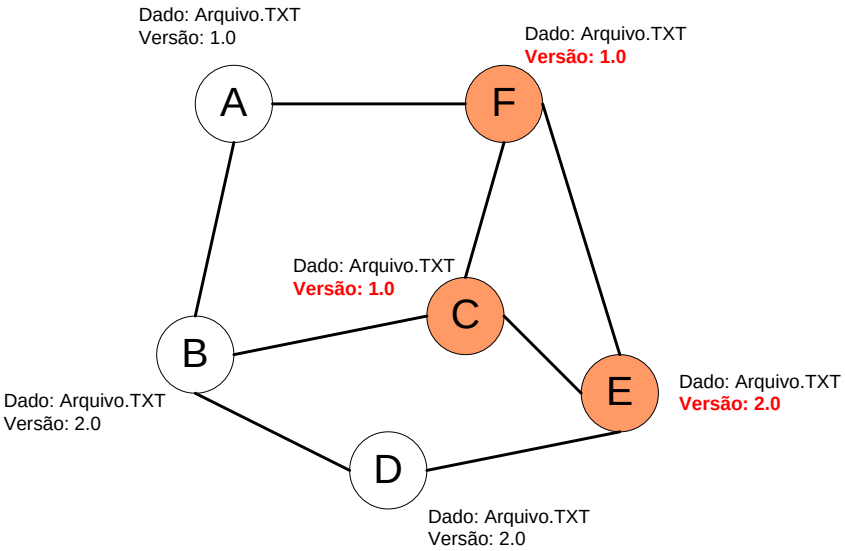


Figura 55 - Operação (escrita ou leitura) deve envolver um conjunto de nós onde haja pelo menos um nó em comum com a última operação de escrita (neste exemplo, o nó E).

Como na Figura 55 há um nó em comum (E) com a **última operação de escrita,** que envolveram os nós B, D e E, então é possível verificar entre os nós F, C e E qual é a versão mais recente do dado que eles possuem (Versão 2.0), e então trabalhar com base nele. Desta forma, é **assegurado** que não importa qual seja a operação realizada após uma **escrita**, sempre será possível verificar qual é a versão **mais recente** do dado.

6.7.1. Protocolos Tradicionais para Replicação de Dados

Existem alguns protocolos tradicionais para realização de replicação de dados. Nas próximas seções esses protocolos são descritos com enfoque nas seguintes características:

- Distribuição de carga entre as cópias
- Custo para cada operação
    - Número de mensagens e cópias envolvidas em uma operação de leitura ou escrita.
- Disponibilidade do dado
    - O quanto um protocolo é tolerante a falhas

6.7.1.1. Protocolo STONEBRAKER, 79 *(Cópia Primária)*

- Utiliza um nó especial (**cópia primária**).

- Todos os nós acessam a ***cópia primária*** quando desejam **escrever** ou **ler** um dado.

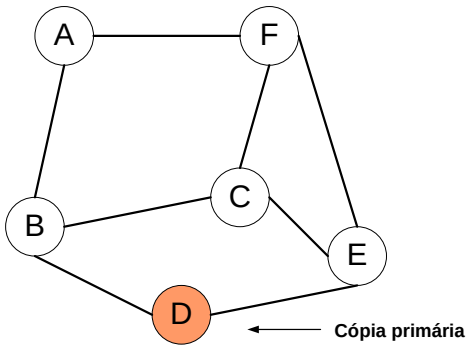


Figura 56 - Todas as operações de leitura e escrita são feitas por intermédio de um nó especial (cópia primária).

- Caso o nó falhe, todas as alterações pendentes são abortadas e é realizada uma ***eleição de líder*** para eleger uma ***nova cópia primária***.

VANTAGEM

- Facilidade em poder ler ou escrever acessando apenas um nó

DESVANTAGEM

- Congestionamento: Todas as operações estão concentradas em um único nó.
- Necessidade de realizar eleição caso a cópia primária falhe
- Demora em atualizar todas as cópias, uma vez que a operação é feita apenas em um nó.

6.7.1.2. Protocolo ROWA *(Lê um, escreve todos – **R**ead **O**ne **W**rite **A**ll)*

- A ***leitura*** de um dado é feito através do acesso a ***um único nó*** (*não importa qual seja. Não é necessário fazer eleição de líder para determinar o nó. Como todos os nós estarão com o mesmo dado, a leitura pode ser feita aleatoriamente em qualquer um dos nós*).

- Para ***escrever*** um dado, é necessário escrevê-lo em ***TODOS*** os nós.

VANTAGEM

- A **leitura** é feita sempre através de uma única cópia.

DESVANTAGEM

- A **escrita** é muito cara, pois é necessário escrever em **TODOS** os nós.
- ***Baixa tolerância a falhas*** para operações de **escrita**.
    - Para que um dado seja replicado para os outros nós, é necessário que cada nó confirme que recebeu o dado corretamente. Se um nó falha, então os outros nós vão ficar esperando uma mensagem de confirmação dele que nunca vai chegar. Logo, após um tempo finito x a ***operação é abortada*** a fim de manter a consistência dos dados.

6.7.2. Protocolos Tradicionais que Utilizam Votação

6.7.2.1. Protocolo THOMAS, 79 *(Votação Simples)*

- Este protocolo é o **mais simples** de todos citados até o momento.

- Esta solução é **totalmente distribuída**.

- Operações são realizadas após a execução dos seguintes passos:

- É realizada uma votação entre todos os nós ativos
- Se mais da metade dos nós autorizarem a operação (***Quorum***), então ela é realizada, caso contrário, ela é abortada.

"Quorum é o número mínimo de votos que uma transação distribuída deve obter a fim de executar uma operação em um sistema distribuído."

- Existe uma **cópia** do dado em **cada nó** da rede.

VANTAGEM

- Solução é totalmente distribuída

DESVANTAGEM

- Custo da **leitura** é **caro**.
- À medida que os nós falham, há uma degradação da disponibilidade.

6.7.2.2. Protocolo GIFFORD, 79 *(Votação Ponderada)*

- Cada nó possui um peso

- Os nós mais confiáveis possuem um peso maior (por exemplo, um nó que possui proteção contra queda de energia pode ter um peso maior, pois é mais confiável)

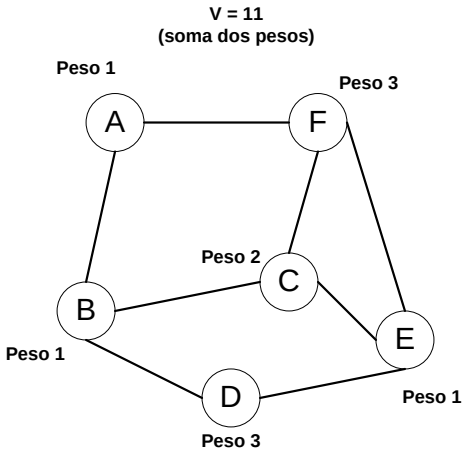


Figura 57 - Cada nó possui um peso diferenciado, o qual é proporcional a sua confiabilidade.

Os quoruns devem satisfazer as seguintes constantes:

- r + w > v
- 2w > v

Exemplo:

É possível realizar uma operação com os nós C, D e F, pois a soma dos pesos deles (8) é maior que a metade da soma dos pesos de cada nó (V=11).

VANTAGEM

- Solução é totalmente distribuída
- É possível **enfocar** as transações nos **nós mais robustos**.

DESVANTAGEM

- À medida que os nós falham, há uma degradação da disponibilidade.
- Quoruns são grandes

6.7.3. Melhorias nos Protocolos que Utilizam votação

6.7.3.1. Votação Dinâmica

A idéia da votação dinâmica é realizar votações com apenas os nós ativos e ***concorrentes***. Na Figura 58 todos os nós encontram-se ativos. Cada nó armazena as seguintes informações:

- Versão do dado armazenado

- Número de nós ativos que participaram da última operação

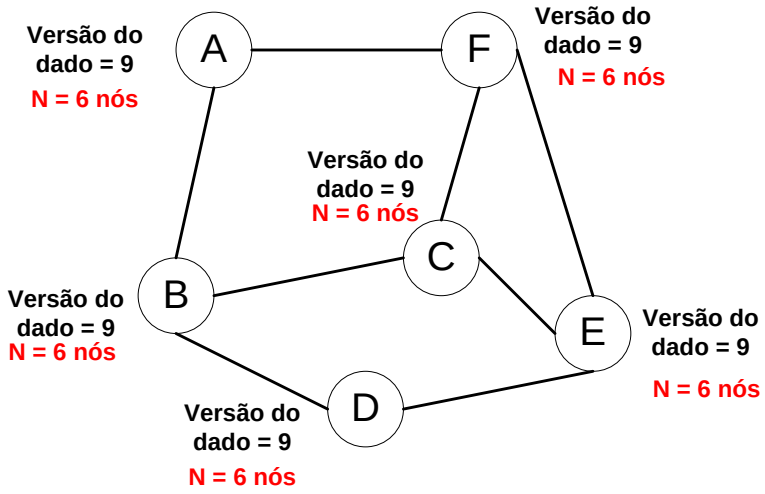


Figura 58 - Todos os nós estão ativos.

Para que seja possível ocorrer uma operação de escrita nesse cenário, é necessário que o quorum seja pelo menos maior que a metade dos nós (Como N=6, então é necessário pelo menos um quorum de 4 nós).

Agora imagine que após alguma falha ocorra um particionamento da rede, conforme Figura 59.

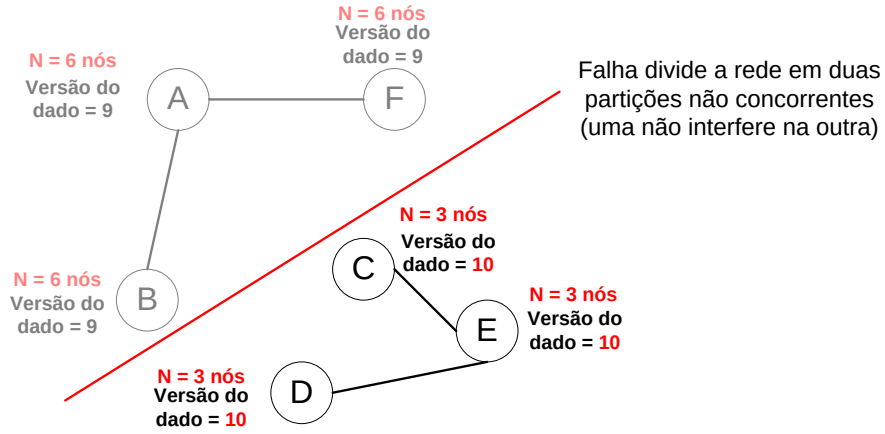


Figura 59 - A ocorrência de uma falha particiona a rede.

Obs: Quando a rede é particionada, a próxima operação é feita sobre o valor antigo de n. Só depois de feita a operação é que n é atualizado.

Como foram criadas duas novas partições não concorrentes, então agora o valor de N é atualizado **dinamicamente**, o qual vai refletir o número de nós ativos. Então, para que seja possível realizar uma operação na partição inferior criada (nós C, D e E), será necessário uma **votação** que tenha um *quorum* com mais que a metade dos nós ativos nesta partição (atualizados **dinamicamente** para N=3), logo, um *quorum* de 2 nós é suficiente para que uma operação ocorra (Após uma votação favorável deste exemplo, então o dado é atualizado e sua versão é incrementada para 10).

"Se o número de nós ativos restantes é par, então o quorum é feito com base na metade dos nós mais um nó especial (para que haja maioria)."

Obs: Quando ocorre o particionamento de uma rede, é necessário garantir que somente uma partição vai sobreviver, pois se uma partição "morre" e depois "retorna", então haverá problemas de inconsistência na versão dos dados.

VANTAGEM

- Mesmo que os nós falhem até restarem 2 nós (por exemplo), operações ainda podem ser realizadas, ao contrário de uma votação tradicional.

DESVANTAGEM

- Se todos os nós falharem, menos um nó C, este ainda poderá atualizar seu dado. No entanto, se os outros nós voltarem a funcionar, mas C falhar, então os outros nós devem aguardar C estar ativo novamente para poderem continuar (para evitar inconsistência de dados, já que C é o nó que está com o dado mais atual até o momento).

6.7.3.2. Votação com Testemunhas

- OBJETIVO: Diminuir o espaço total necessário para o armazenamento dos dados distribuídos, sem afetar drasticamente sua disponibilidade.

- **Testemunhas** são utilizadas para armazenar **apenas a versão** do dado mais recente, e não o dado propriamente dito.

- Dada uma rede composta por N nós, **T** é o número de nós **testemunhas** (que armazenam somente a versão do dado mais novo) e **D** é o número de nós que possuem cópias reais dos **dados**, onde:

$$N = T + D$$

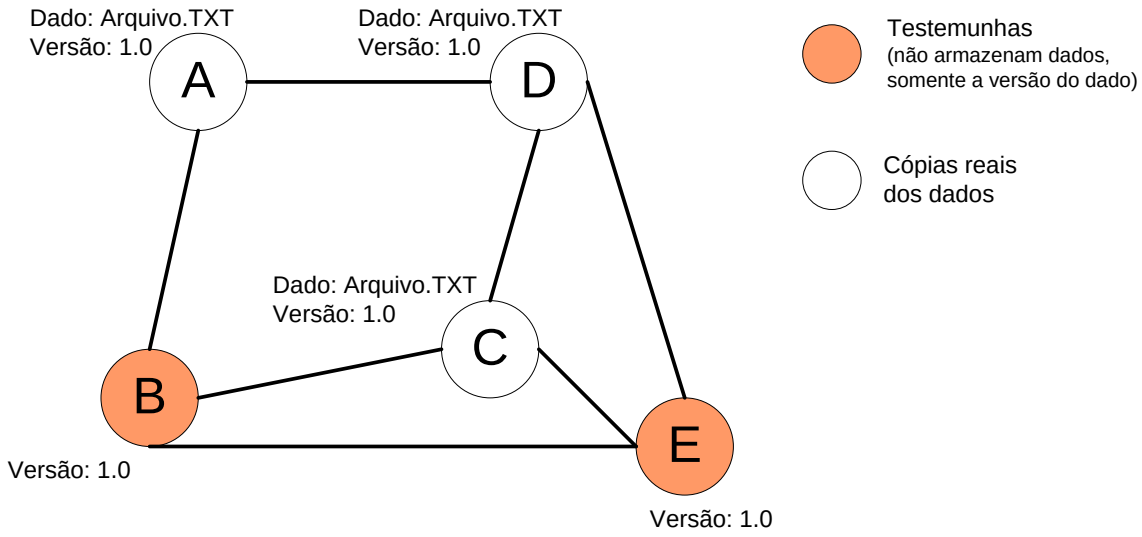


Figura 60 - Representação dos nós que armazenam os dados replicados e das testemunhas que armazenam somente a versão dos dados.

Exemplo:

Imagine duas situações onde há falhas de processo. A primeira delas corresponde a Figura 61.

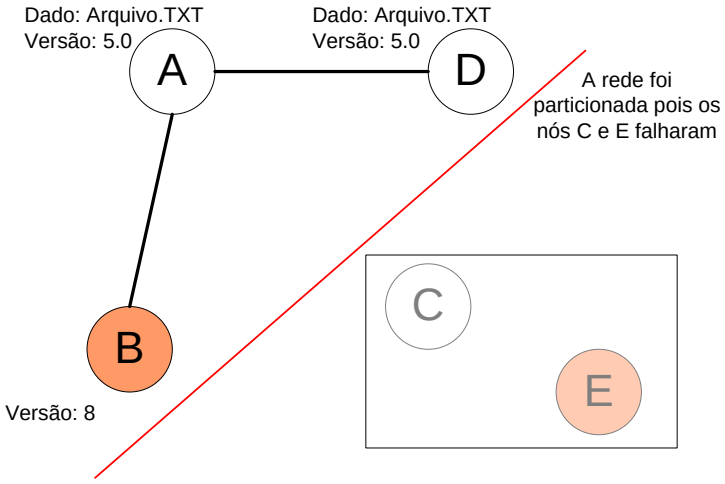


Figura 61 - Impossibilidade de recuperação do dado após a falha de dois nós.

Nesta situação não é possível recuperar o dado, pois ao consultar uma testemunha é possível que o dado mais recente possua versão 8, enquanto que A e D armazenam o dado na versão 5 apenas. Um exemplo onde seria possível recuperar o dado, é com A ou D possuindo o arquivo na versão 8.

VANTAGEM

- Ótima solução em um cenário onde número de cópias é pequeno

6.7.4. Protocolos de votação que utilizam estruturas lógicas

- Em situações onde existem muitos nós contendo cópias dos dados, os protocolos que utilizam votação tradicional são muito caros, pois os *quorums* crescem linearmente com o número de cópias.

- O que fazer para solucionar este problema?

- Organizar os nós (cópias dos dados) em uma **estrutura lógica**.
- Utilizar esta estrutura lógica para **melhorar o custo do protocolo**.

6.7.4.1. ANEL

- Os nós são organizados logicamente em **anel**

- QUORUM DE LEITURA: serão sempre dois nós **vizinhos** (adjacentes)
    - Tamanho do quorum = 2 (*SEMPRE*)

- QUORUM DE ESCRITA: Um nó sim, outro não, ..., até dar a volta no anel. Se o número de nós for par, escolher mais um nó qualquer que não tenha sido escolhido ainda.
    - Tamanho do quorum: N/2 + 1 (um a mais que a metade)

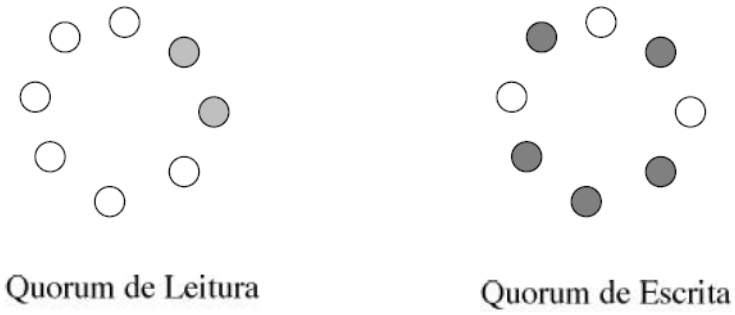


Figura 62 - (Figura extraída da internet - fonte desconhecida)

O anel também pode ser organizado em colunas:

Figura 63 - (Figura extraída da internet - fonte desconhecida)

- QUORUM DE LEITURA: um nó de cada coluna (na mesma linha)
- QUORUM DE ESCRITA: Uma coluna inteira, mais um nó qualquer da outra coluna.

6.7.4.2. GRADE

- Os nós são organizados logicamente em **GRADE**

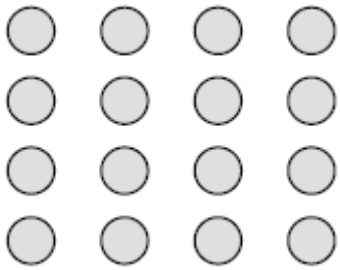


Figura 64 - (Figura extraída da internet - fonte desconhecida)

- QUORUM DE LEITURA: um nó de cada coluna
    - Complexidade de nós: √n

- QUORUM DE ESCRITA: Uma coluna inteira, mais um nó de cada coluna.
    - Complexidade de nós: 2√n - 1

6.7.4.3. GRADE HIERÁRQUICA

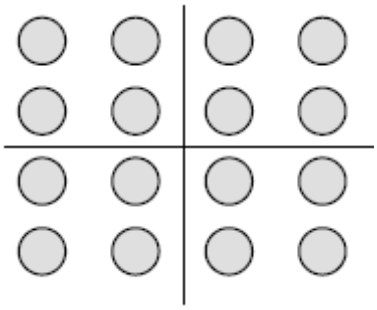


Figura 65 - (Figura extraída da internet - fonte desconhecida)

- Melhor disponibilidade para escrita

- Maior flexibilidade na formação de quorums

TAMANHO DOS QUORUMS: √n

6.7.4.4. QUORUM EM ÁRVORE

- Escrito por Agrawal e Abadi, mesmos autores do algoritmo Agrawal e Abadi para exclusão mútua, citado na seção 6.3.6.

- Cada cópia é um nó da árvore formada.

- Objetivo é formar recursivamente um quorum incluindo a raiz e w filhos. Para cada filho incluído, incluir também w de seus filhos, sucessivamente até uma profundidade l.

- Recursão termina quando comprimento do quorum a ser formado é zero.

- Se não houver altura suficiente, quorum falha.