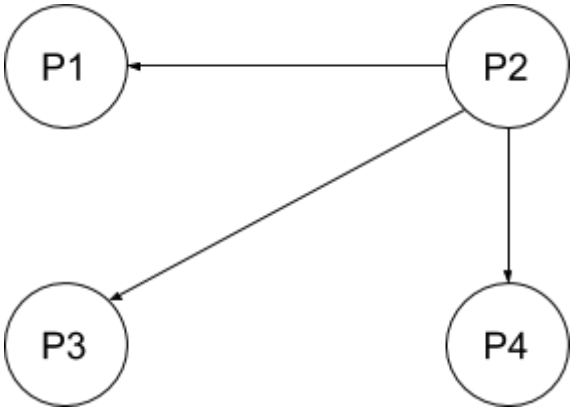
Aula 14/05

Definições Iniciais



- Sistemas Distribuídos = processos + Canais de comunicação
- Execuções
    - Uma execução em sistemas distribuídos consiste de uma sequência de transições
    - Uma configuração é alcançável se existe uma transição que leve até esta configuração

- Estados e Eventos
    - Uma configuração de um SD é como uma “fotografia”, de modo que observa-se os estados dos processos e as mensagens presentes nos seus canais
    - Processos podem realizar eventos (envio/recebimento de mensagens)
    - Um processo é denominado iniciador de seu primeiro evento é interno ou de envio de mensagens
    - Se existe apenas um iniciador no SD, diz-se que o algoritmo é centralizado. Se existe mais de um, então ele é descentralizado

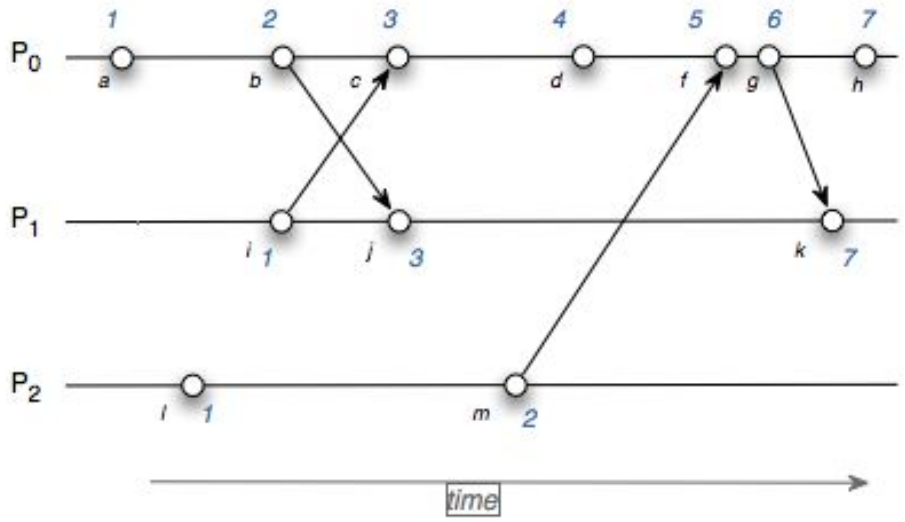
- Asserções
    - Uma asserção é uma propriedade de segurança (*safety*) se ela deve ser verdadeira a cada configuração de cada execução do algoritmo. Ex: Conta bancária
    - Uma asserção é uma propriedade de atividade (*liveness*) se ela é verdadeira em alguma configuração de cada execução do algoritmo

- Ordem Casual
    - A ordem casual ($\prec$) define a ocorrência de eventos em um SD

- Se **a** e **b** são dois eventos no mesmo processo e **a** ocorre antes de **b**, então **a** $\prec$ **b**.
- Se **a** é um send e **b** é um evento de receive correspondente, então **a** $\prec$ **b**. Observe que, nesse caso, **a** e **b** estão em processos distintos.
- Se **a** $\prec$ **b** e **b** $\prec$ **c**, então **a** $\prec$ **c**.
- **a** $\preccurlyeq$ **b** significa **a** $\prec$ **b** ou **a** = **b**
- **a** e **b** são concorrentes se nem **a** $\prec$ **b** nem **b** $\prec$ **a**.
- A permuta da ordem de eventos concorrentes não afetam o resultado da execução.

Relógios Lógicos

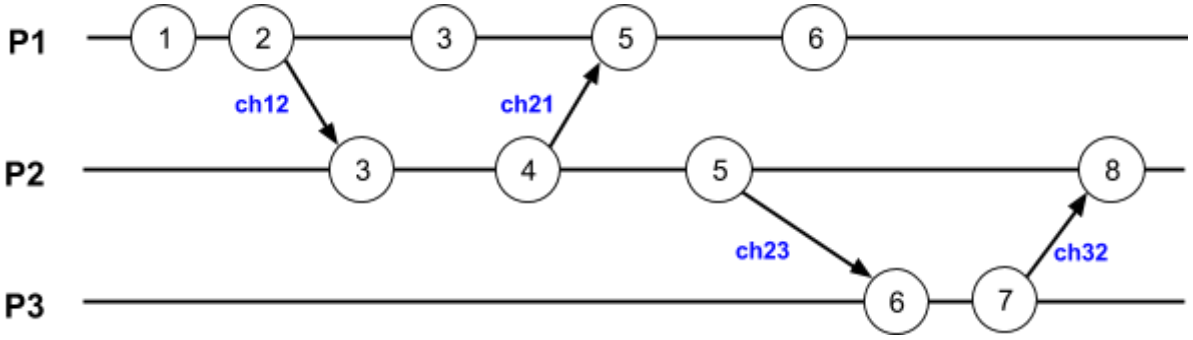
- Relógio de Lamport
  - Condições do algoritmo
    - Considerando C_i o relógio local do processo P_i, então:
      - Cada evento **a** em **i** tem o tempo $C_i(a)$, ou seja, este é o valor do relógio de quando o evento **a** ocorreu.
      - C_i é incrementado em 1 para cada evento em **i**
      - Se **a** é um evento send do processo **i** para **j** então, ao receber **m**, $C_j = max(C_j, C_i(a) + 1)$
  - Exemplo:

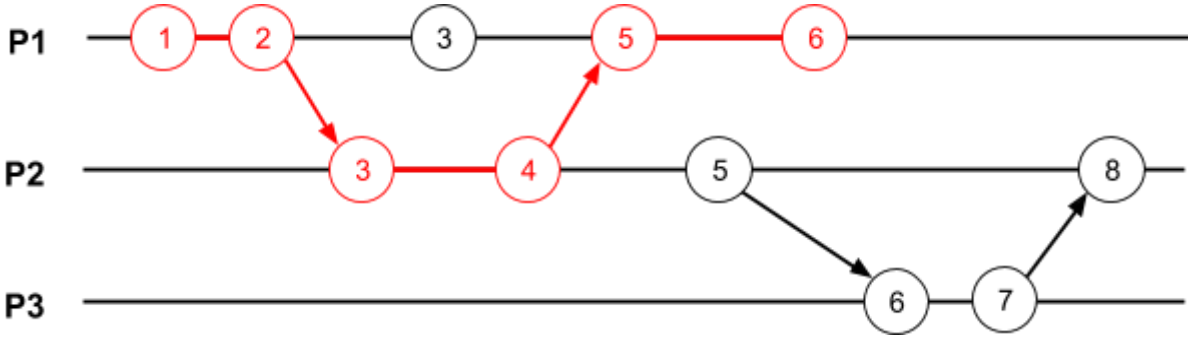


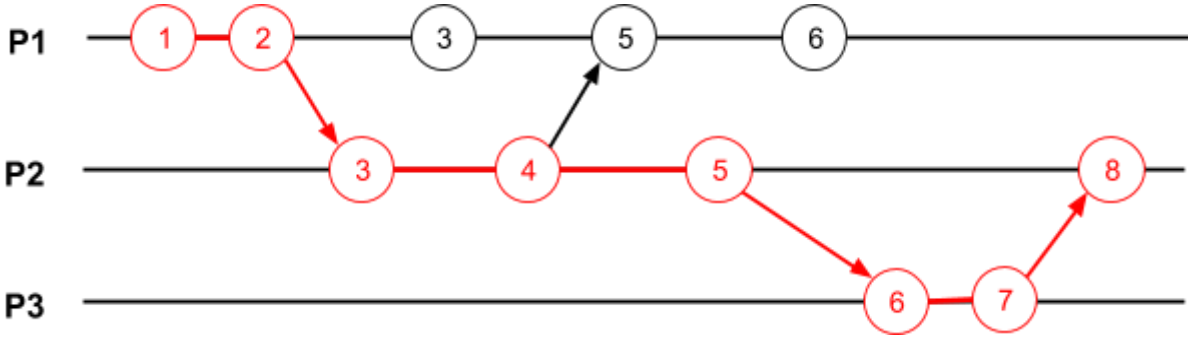
Event *k* in process P_1 is the receipt of the message sent by event *g* in P_0. If event *k* was just a normal local event, the P_1 would assign it a timestamp of 4. However, since the received timestamp is 6, which is greater than 4, the timestamp counter is set to 6+1, or 7. Event *k* gets the timestamp of 7. A local event after *k* would get a timestamp of 8.
With Lamport timestamps, we're assured that two causally-related

events will have timestamps that reflect the order of events. For example, event *c* happened before event *k* in the Lamport causal sense: the chain of causal events is $c \rightarrow d, d \rightarrow f$, $f \rightarrow g$, and $g \rightarrow k$. Since the *happened-before* relationship is transitive, we know that $c \rightarrow k$ (*c* happened before *k*). The Lamport timestamps reflect this. The timestamp for *c* (3) is less than the timestamp for *k* (7). However, just by looking at timestamps we cannot conclude that there is a causal happened-before relation. For instance, because the timestamp for *l* (1) is less than the timestamp for *j* (3) does not mean that *l* happened before *j*. Those events happen to be concurrent but we cannot discern that by looking at Lamport timestamps.

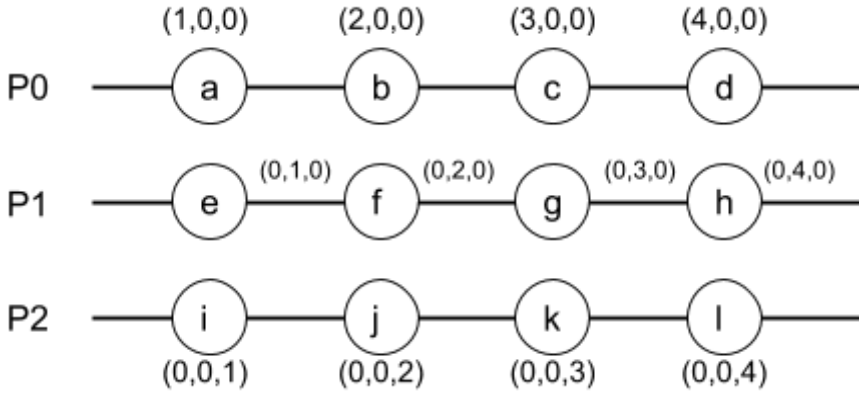
Representação do relógio de lamport para o código do github:



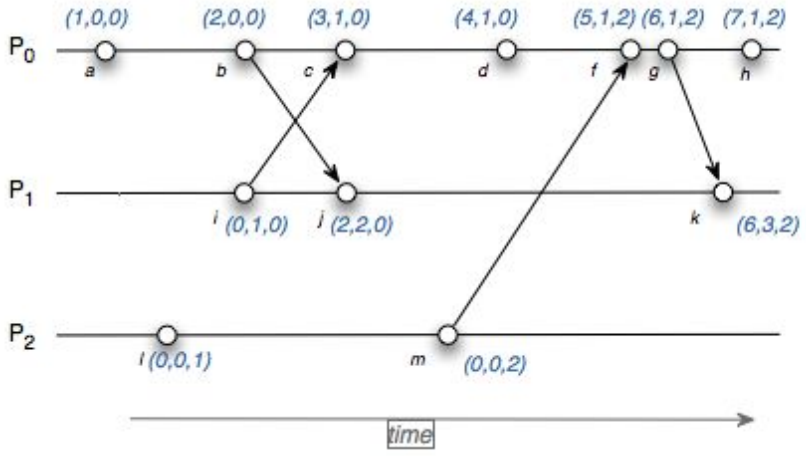




- Relógio Vetorial
  - Neste algoritmo, assumimos que sabemos o número de processos no sistema
  - O clock de cada evento se trata de um array onde cada elemento corresponde a um processo
  - Cada processo tem conhecimento da posição dos demais processos no vetor
  - Supondo que temos os processos (P_0, P_1, P_2), então:

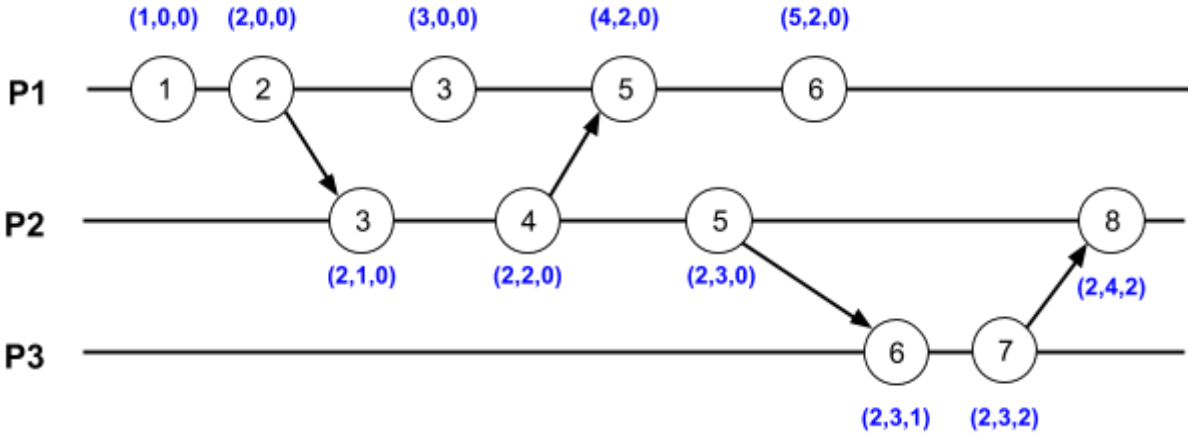


  - Exemplo:



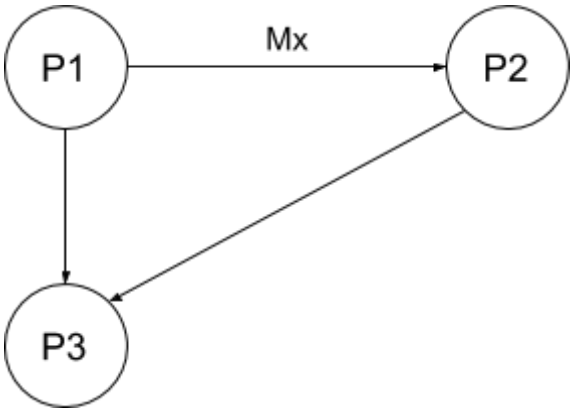
The timestamp for *m* is less than the timestamp for *g* because each element in *m* is less than or equal to the corresponding element in *g*. That is, 0 ≤ 6, 0 ≤ 1, and 2 ≤ 2.
The timestamps for *c* and *j*, on the other hand, are **concurrent**. If we compare the first element, we see that *c* ≥ *j* (3 ≥ 2). If we compare the second element, we see that *c* ≤ *j* (1 ≤ 2). Because of this, we cannot say that the vector for *c* is either less than or greater than the vector for *j*.

Representação do relógio vetorial para o código do github:



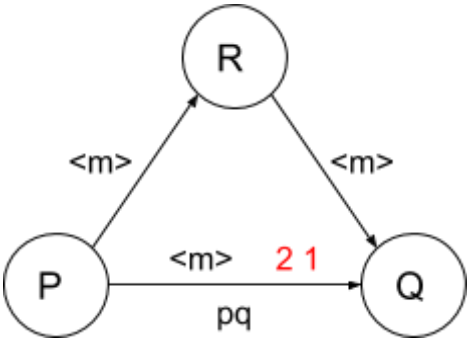
Aula 21/05

Snapshots

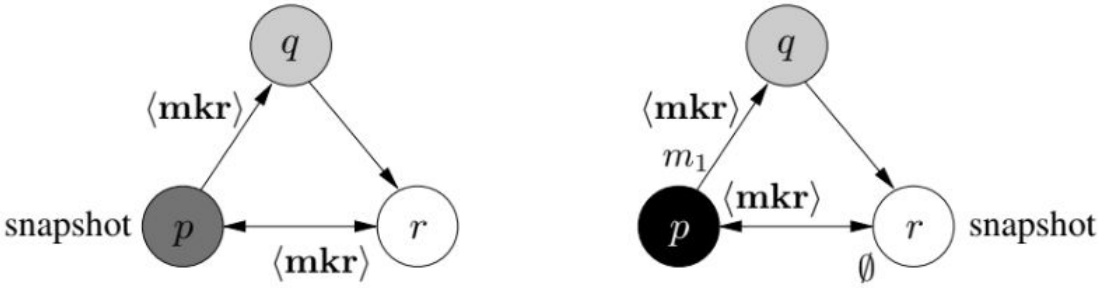


- Uma visão consistente do estado global de um sistema distribuído
- É uma configuração da execução desse sistema compondo:
    - O estado de cada processo
    - As mensagens em trânsito nos canais que ligam esses processos
- Aplicações
    - Detecção de estados imutáveis
        - Deadlocks
        - Terminação (finalização de tarefas de processos)
        - Coleta de Lixo
    - Criação de checkpoints
        - Pontos de reiniciação após falhas
        - Debug
- Desafio
    - Trabalhar em tempo de execução sem parar a execução do sistema distribuído
    - O registro do estado dos processos e das mensagens dos canais podem acontecer em momentos diferentes
- Definições
    - Mensagem básica: é uma mensagem do próprio sistema distribuído
    - Mensagem de controle: mensagem que faz parte do algoritmo de snapshot
    - Um evento em um dado processo é chamado **pré-snapshot** quando ele acontece antes do snapshot, do contrário ele é **pós-snapshot**.
    - Um snapshot é considerado consistente quando:
        - Para cada evento pré-snapshot **a**, todos os eventos que são casualmente anteriores a **a** ($\prec a$) também devem ser pré-snapshot.

        - Uma mensagem básica faz parte do estado de um canal se e somente se o evento **send** é **pré-snapshot** enquanto que o evento correspondente **receive** é **pós-snapshot**.
- Algoritmo de Chandy-Lamport
    - Características
        - Os canais entre os processos têm que ser **FIFO**
        - Qualquer processo iniciador pode dar início ao algoritmo de snapshot
    - O processo iniciador registra um snapshot local do seu estado e, em seguida, envia uma mensagem de controle <*marker*> para todos os seus canais de saída solicitando aos seus vizinhos que também registrem um snapshot
    - Quando um processo que ainda não registrou o snapshot correspondente recebe a mensagem <*marker*>, ele registra um snapshot local do seu estado e envia a mensagem <*marker*> para todos os seus canais de saída
    - Um processo **q** também registra no snapshot, como parte do estado de um canal **pq**, toda mensagem (básica) de entrada que **q** recebe de **p** depois do momento que **q** registra seu snapshot local e antes do momento que **q** recebe <*marker*> de **p**

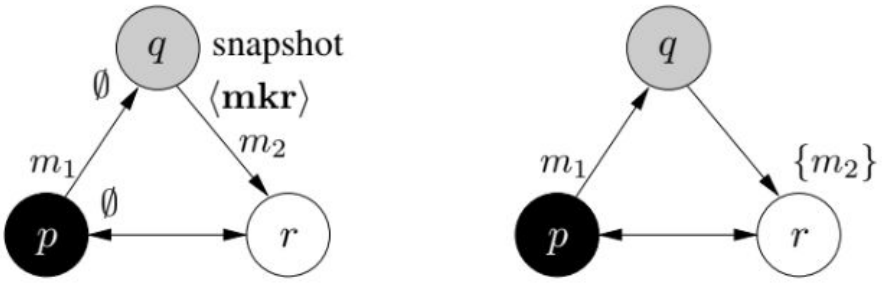


    - *Pela convenção do livro: Estado vazio (Ø) = recebe marcador (<*m*>) => tira snapshot local



    - No exemplo acima temos:
        - Processo iniciador **p** tira um snapshot local e envia marcadores (<*mkr*>) aos processos vizinhos

        - Após o envio de um marcador para o processo **q**, **p** envia uma mensagem básica $\mathbf{m_1}$ para o mesmo
        - Ao receber o marcador, **r** tira um snapshot local e atribui o estado daquele canal, por onde ele recebeu o marcador, como vazio (Ø)
        - **r**, então, envia um marcador para o seu vizinho (**p**)



        - **p** recebe o marcador de **r** e atribui o estado do canal como vazio (Ø)
        - **q** recebe o marcador de **p**, tira um snapshot local e associa o estado do canal como vazio (Ø). Em seguida, ele envia uma mensagem $\mathbf{m_2}$ e um marcador para **r,** nesta ordem.
        - **r** recebe a mensagem $\mathbf{m_2}$ e o marcador. Como a mensagem foi recebida primeiro, o estado do canal é tido como $\{m_2\}$
        - Ao final da execução do algoritmo, verifica-se que a mensagem $\mathbf{m_2}$ foi gravada no processo de snapshot, mas não a mensagem $\mathbf{m_1}$ justamente por causa da ordem de envio marcador-mensagem

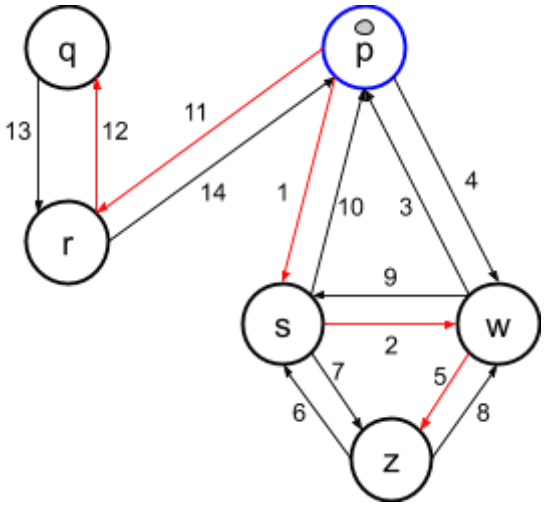
- Algoritmo de Lai-Yang
    - Características
        - **Não** requer canais FIFO
        - Qualquer processo iniciador pode dar início ao algoritmo de snapshot
    - Enquanto um processo não registra o snapshot, ele envia o marcador *false* junto com cada mensagem básica enviada através de seus canais de saída. Depois que o processo registra o snapshot, ele passa a enviar *true*.
    - Quando um processo que ainda não registrou o snapshot recebe o marcador *false*, ele registra o seu snapshot local
      Portanto, o marcador *true* indica que os demais processos que receberem esse marcador devem registrar seu snapshot local
    - Um processo **q** registra como sendo o estado de um canal de entrada **pq** toda mensagem básica recebida com o marcador *false*.
    - Problemas

        - Se o iniciador não enviar nenhuma mensagem básica depois que ele registrar seu snapshot local, como os demais processos vão proceder com o snapshot?
          Pode ser que o processo não tenha mensagens para serem enviadas
        - Como um processo sabe que deve parar de esperar por mensagens de entrada marcadas como *false* no momento de registrar o estado de um canal?
    - Solução
        - Um processo **p** depois de seu snapshot, envia para todos os seus canais de saída uma mensagem de controle especial que contém o número de mensagens básicas com o marcador *false* que foram enviadas através do respectivo canal

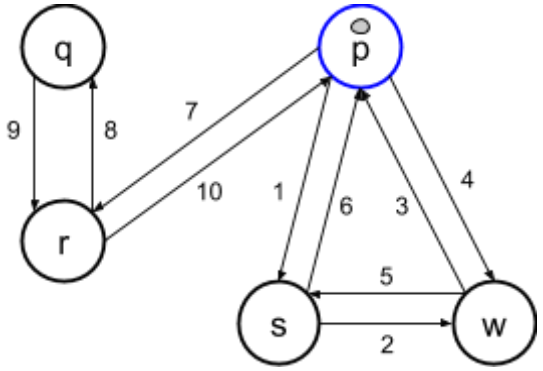
Aula 28/05

Waves

- Os algoritmos tipo wave são aplicados em computações distribuídas **finitas** onde se fazem necessários um ou mais eventos de **decisão**.
- É iniciado por um dos processos (iniciador) que é o responsável pelo evento final de decisão
- Propriedades
    - Computação é finita (ex: não cíclica)
    - Contém um ou mais eventos de decisão
    - Para cada evento de decisão **b** e um processo **p**, **a** $\prec$ **b** para todo evento **a** em **p**.
- Algoritmos de travessia
    - São algoritmos wave centralizados nos quais o iniciador envia um token (mensagem propagada pelos processos) através da rede
    - Depois de visitar todos os processos envolvidos, o token retorna para o iniciador que toma a decisão
        - Exemplo: o algoritmo tipo anel - o token dá uma volta no anel e volta ao iniciador
    - Um algoritmo de travessia pode ser usado para construir uma árvore geradora (spanning tree) da rede tendo o iniciador como raiz
        - Cada processo não iniciador tem como pai o processo do qual ele recebeu o token pela primeira vez
- Algoritmo de Tarry
    - É um algoritmo de travessia para redes não direcionadas
    - Regras
        - Um processo nunca propaga um token pelo mesmo canal duas vezes
        - Um processo só propaga o token para o seu pai se não houver outra opção
    - O pai de um processo é aquele de quem ele recebeu o token pela primeira vez

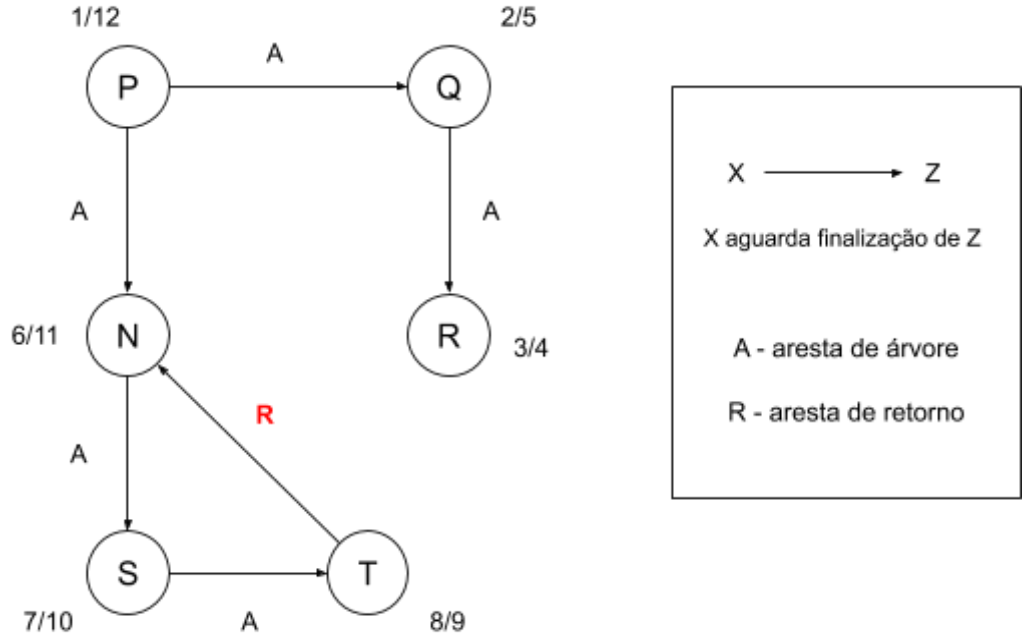


- Representação do algoritmo de Tarry para o caso a seguir:



Detecção de Deadlocks

- Execução de um algoritmo de Snapshot

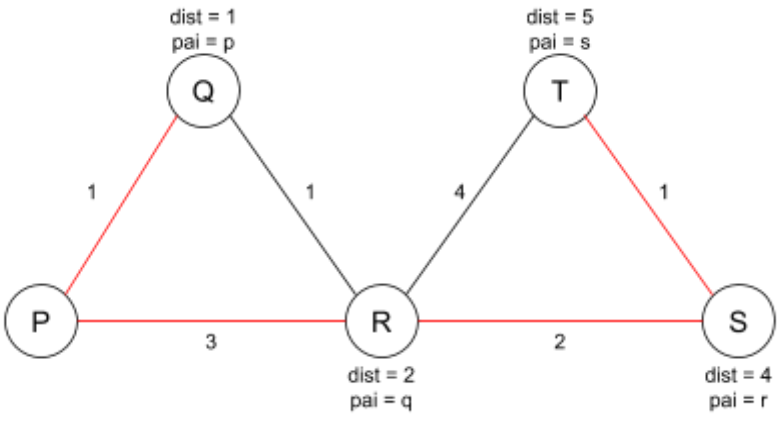


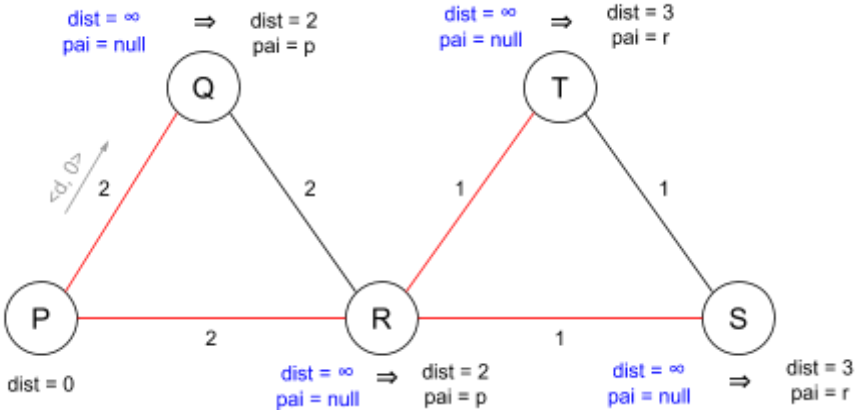
- Ao executar o algoritmo de profundidade, existem arestas de árvore (A) e podem existir arestas de retorno (R). Se uma aresta de retorno for detectada, significa que há ciclos no sistema distribuído e, portanto, foi detectado um deadlock.

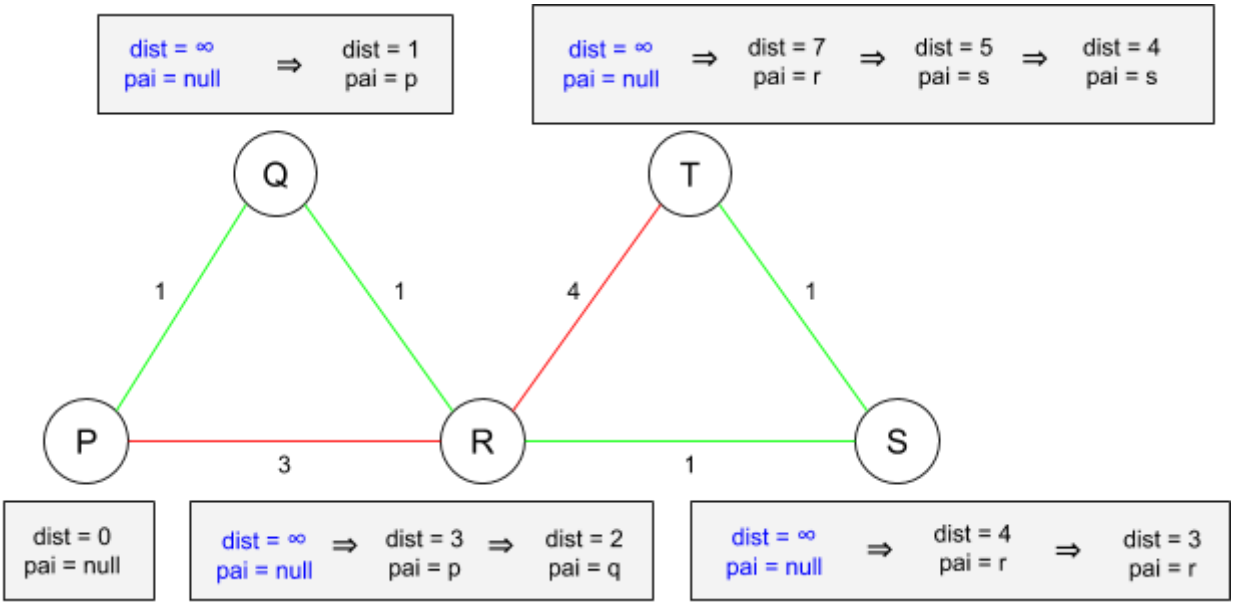
Aula 18/06

Algoritmo de Roteamento

- Chandy-Misra
  - Encontra uma árvore de caminhos mais curtos até o iniciador
  - Cada processo **p** mantém
    - dist **p** → distância mais curta de **p** até o iniciador
    - parent **p** → o pai de **p**
  - No início da execução
    - dist **p** do iniciador = 0
    - dist **p** dos demais processos = ∞
    - parent **p** = null
  - Execução
    - Iniciador envia o dist **p** dele para os vizinhos
    - Quando um processo recebe mensagem do vizinho
      - Se (*distp_receb* + *peso_canal*) < *disp_local*
        - *distp_local* = *distp_receb* + *peso_canal*
        - *parentp* = processo do qual recebeu a *distp_receb*
        - O processo propaga para todos os seus vizinhos (exceto o pai) a sua *distp* atualizada






dist = ∞
pai = null
⇒
dist = 1
pai = p
dist = ∞
pai = null
⇒
dist = 7
pai = r
⇒
dist = 5
pai = s
⇒
dist = 4
pai = s
Q
T
1
1
4
1
P
R
S
3
1
dist = 0
pai = null
dist = ∞
pai = null
⇒
dist = 3
pai = p
⇒
dist = 2
pai = q
dist = ∞
pai = null
⇒
dist = 4
pai = r
⇒
dist = 3
pai = r

Aula 25/06

Algoritmos de Detecção de Término

- Algoritmo de Safra
    - Utiliza um algoritmo de travessia
        - Ex: Algoritmo de Tarry
    - Um processo só repropaga um token quando está no modo passivo
    - Cada processo mantém um contador interno
        - O contador é incrementado toda vez que é enviada uma mensagem básica
        - O contador é decrementado toda vez que uma mensagem é recebida
        - O token acumula os valores dos contadores dos processos
        - O algoritmo termina quando a soma de todos os contadores for 0 (zero)

Aula 09/07

Algoritmos de Eleição

- Eleição em Árvore
    - O objetivo é eleger o próximo processo iniciador com base em um critério (ex: maior valor)
    - No exemplo abaixo, o processos acessam os filhos a fim de identificar aquele com o maior valor. Ao final do algoritmo, P compara os valores resultantes de V(10), eleito no lado direito da árvore, e T(20), eleito no lado esquerdo da árvore. Comparando os processos tem-se que T é o processo eleito naquele sistema e, portanto, o novo processo iniciador.