# KONSENSUS V DISTRIBUOVANÝCH SYSTÉMECH

František Zbořil ml., Petr Hanáček

GTs – 2026

# Synchronizace

- **Synchronizace** zaručuje (částečné) uspořádání mezi událostmi
- Pokud máme systémy se sdílenou pamětí, pak můžeme použít nám již známé mechanismy, jako jsou semafory nebo monitory
- Pokud máme sdílený prostředek v distribuovaných systémech, jako například nástěnku, synchronizaci také již dokážeme realizovat, například pomocí systému LINDA
- Ale pokud chceme zajistit synchronizaci pouze předáváním zpráv, potom se budeme zabývat
  1. *Synchronizací globálním (reálným) časem*
  2. *Synchronizací řízenou master uzlem*
  3. *Synchronizací založenou na shodě mezi procesy*

# Synchronizace v distribuovaných systémech, úvodní poznámky

- Požadavky:
  - kauzalita: uspořádání v reálném čase ~ uspořádání dle logických hodin nebo časových razítek (”korektní chování z pohledu uživatele”)
  - všechny procesy uspořádávají události v tom samém pořadí
  - *Pokud bychom měli, jakože nemáme, přesné hodiny ve všech uzlech, pak bychom byli schopni požadavky splnit. Dokážeme je ale splnit i bez nich*
  - *Vždy se totiž nemůžeme spolehnout na skutečný čas a někdy i nemusíme znát centrální řídící uzel*

# SYNCHRONIZACE FYZICKÝM A LOGICKÝM ČASEM

FYZICKÝ ČAS

NTP

# Synchronizace fyzického času, Berkley algoritmus

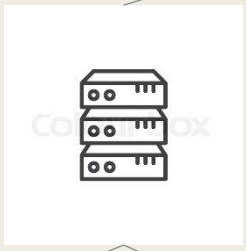
- Předpokládá se, že komunikace dotaz - odpověď (návratový čas) je dostatečně krátká
- Hlavní uzel si vyžádá od všech hodnotu posunu vůči svému aktuálnímu času
- Následně vypočte průměrnou hodnotu posunu a z té posuny pro jednotlivé uzly

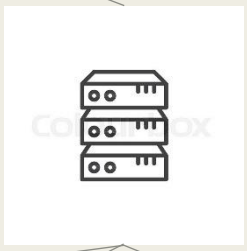
| | 03:14 | | | 03:06 | |
|---|---|---|---|---|---|
| -13 | -16 | +5 | +5 | +8 | -13 |
| 03:01 | 02:58 | 03:19 | 03:01 | 02:58 | 03:19 |

# Network Time Protocol (NTP)

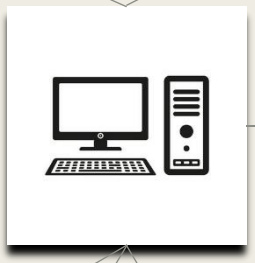
- Synchronizace v síti na různých úrovních (stratech)
- Komunikace přes UDP
- Universal Coordinated Time (UCT)
- Různé módy synchronizace
  - ***Multicast*** – opakované vysílání aktuálního času v rámci skupiny. Vhodné pro malé sítě s vysokou rychlostí přenosu dat
  - ***Klientský přístup*** – volání procedury na serveru klientem, použitelné v případech, kdy není možný multicast
  - ***Párový přístup*** – synchronizováno s velkou přesností

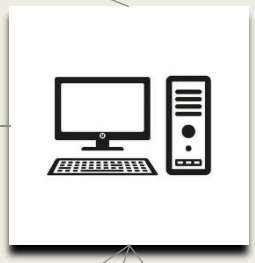
UTC



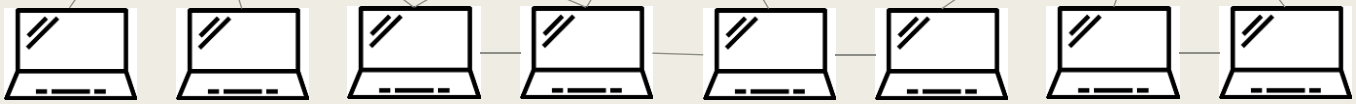


Strata 1 (Time Servers)



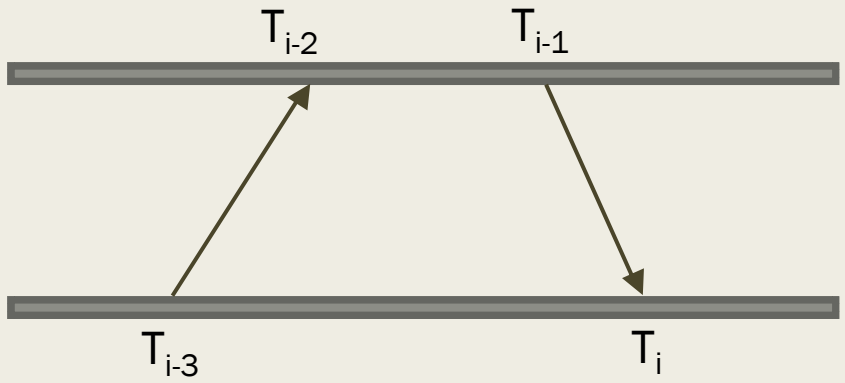


Strata 2



Strata 3

# NTP – trvání komunikace a zpoždění

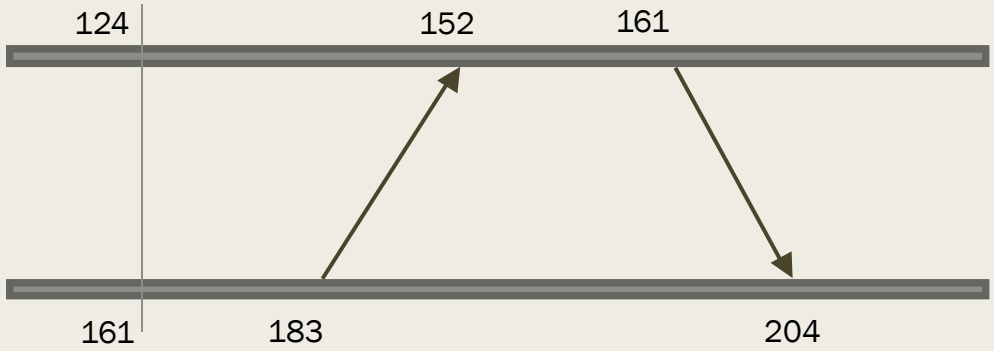



- Zpoždění se spočte jako

  $d_i=T_{i-2}-T_{i-3}+T_i-T_{i-1}$
- Posunutí (offset) je

  $o_i=1/2(T_{i-2}-T_{i-3}+T_{i-1}-T_i)$
- Obojí je spočteno pro všechny kontaktované NTP servery
- Filtrování (Marzullo-ův algoritmus)

# NTP – trvání komunikace a zpoždění, příklad




- Skutečný offset je **37**, zpoždění jedné zprávy je **6**

$d_i=T_{i-2}-T_{i-3}+T_i-T_{i-1}$

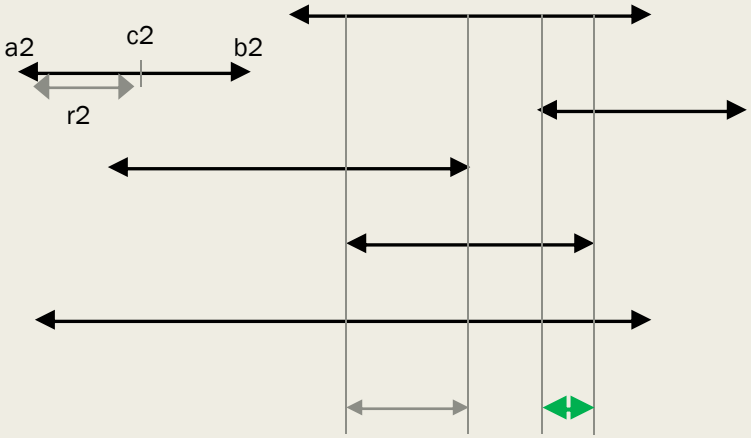
$d_i=152-183+204-161=-31+43=$**12**

$o_i=1/2(T_{i-2}-T_{i-3}+T_{i-1}-T_i)$

$o_i=1/2(152-183+161-204)=1/2(-31-43)=1/2(-74)=$**-37**

# Marzullo-ův algoritmus

- Pro množinu intervalů hledáme nejmenší interval, který spadá do největšího možného počtu intervalů z této množiny.
- tj. pokud by takové intervaly existovaly pro n intervalů a pro n+1 intervalů žádný průnik již nebylo možné nalézt, vezme se ten nejmenší z nich.



# Marzullo-ův algoritmus

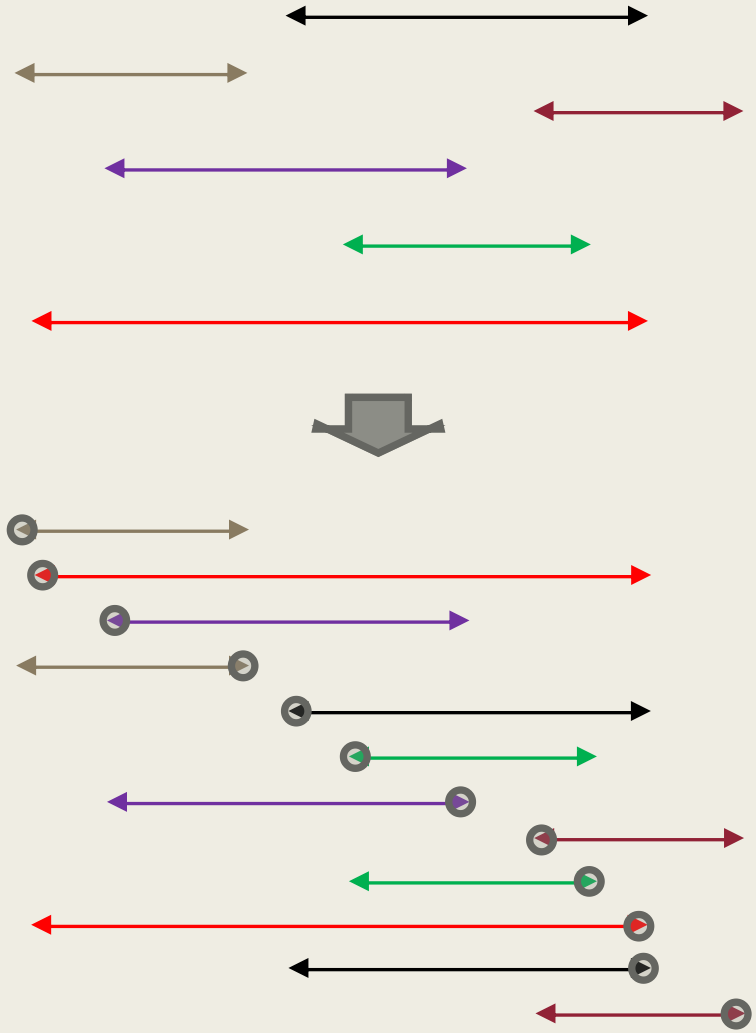
- Každý interval <a,b>, resp. <c-r,c+r> reprezentují dvě dvojice

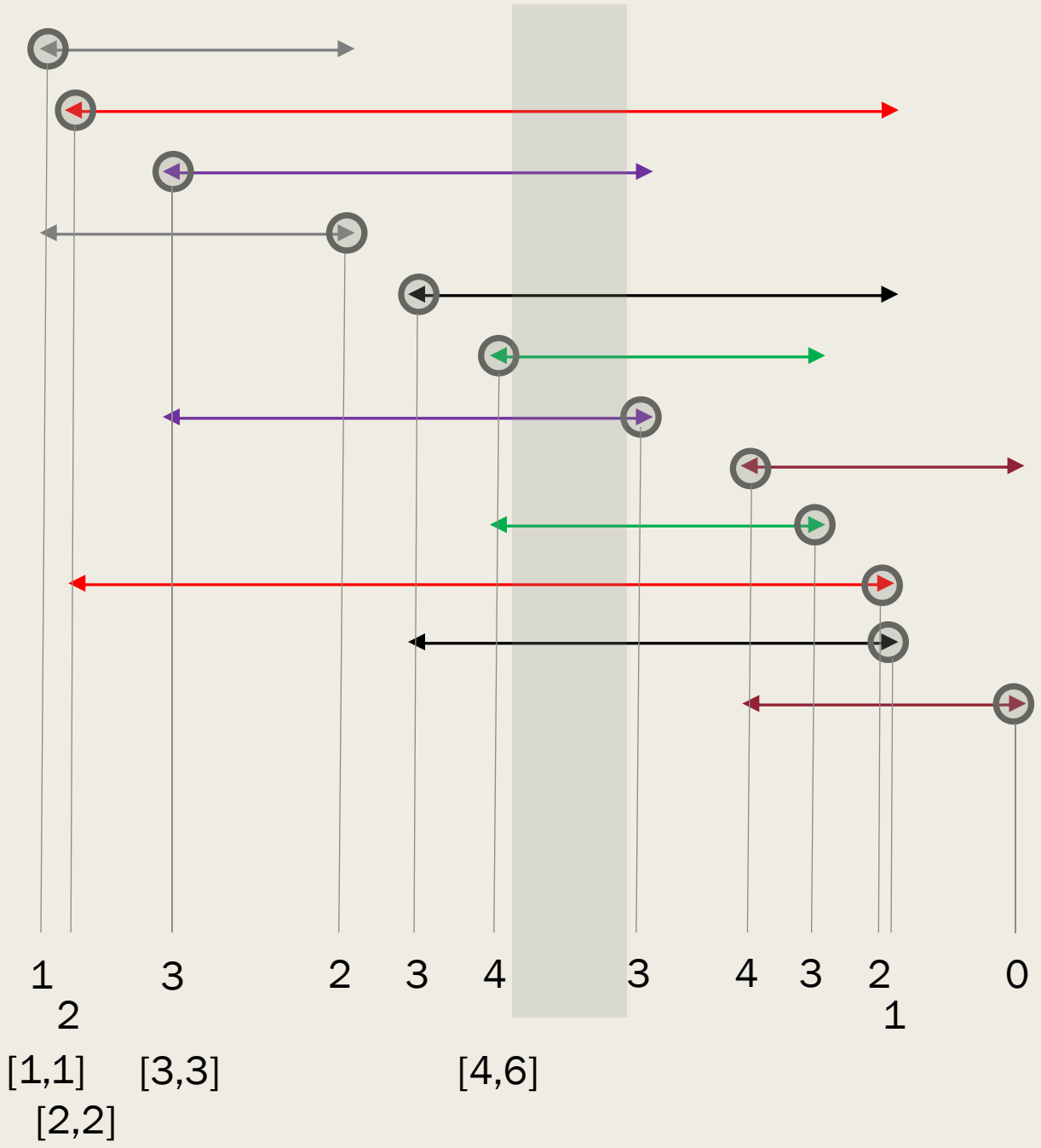
  (a,1), (b,-1) resp. (c-r,1), (c+r,-11)

1. Seřaď intervaly podle *offsetů* počátků a konců intervalů. Počáteční okamžiky označte ci=1, koncové ci=-1
2. Best = 0, Count = 0
3. Postupně pro každý *offset* i od nejnižší hodnoty po nejvyšší

   Count += ci

   Pokud Count > Best pak Best = Count, c = ci




1
2
3
2
3
4
3
4
3
2
1
0
[1,1]
[2,2]
[3,3]
[4,6]
[4,6]

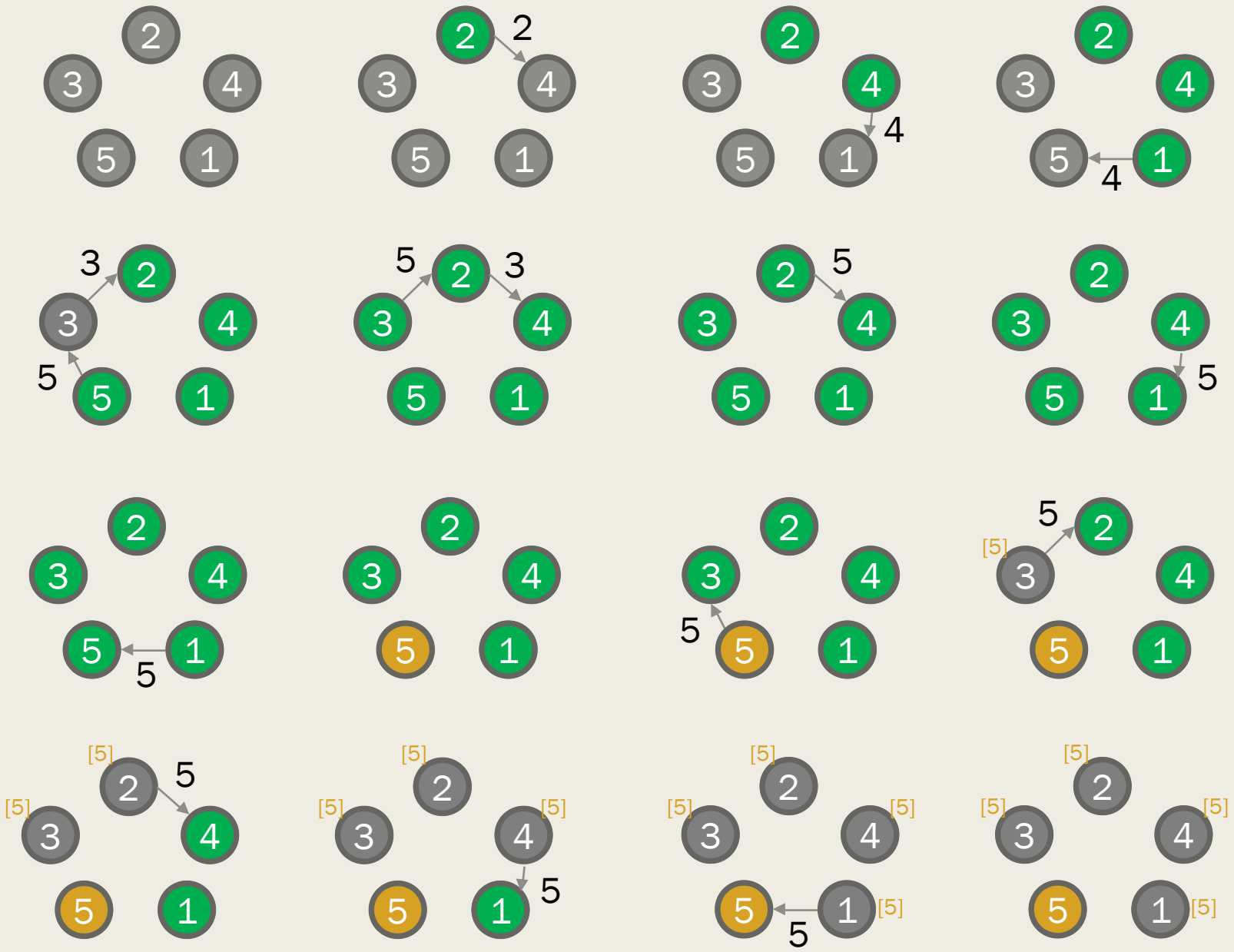
# VOLBA HLAVNÍHO UZLU

# Volba hlavního (master) uzlu, algoritmus Chang a Roberts

- Procesy jsou korektní po celou dobu (pro tuto a ostatní metody, než se přejde k nekorektním procesům)
  - *Jejich chování definované a je správně, nikdy neselžou – nepřestanou fungovat ani na přechodnou dobu*
- Procesy jsou propojeny v **kruhové topologii** a každý má přiděleno unikátní číslo UID
- Algoritmus hledá maximální hodnotu ze všech hodnot těchto uzlů
- Zprávy jsou posílány po směru hodinových ručiček

# Volba master uzlu, Chang and Roberts, algoritmus

- Procesy se shodnou na procesu, který má největší UID takto:
  - *Uzel zahájí komunikaci, označí se za účastníka a pošle zprávu se svým UID následujícímu*
  - *Pokud uzel přeposílá zprávu, označí se za účastníka*
  - *Každý uzel po obdržení zprávy*
    - Pokud UID ve zprávě je větší než UID tohoto uzlu, je zpráva přeposlána dále tak jak je
    - Pokud je číslo ve zprávě menší než má uzel UID, potom
      - *Pokud již je účastník, zahodí zprávu*
      - *Pokud není účastník, nahradí hodnotu původní za svoje UID a přepošle zprávu dále*
    - Pokud je číslo ve zprávě stejné jako má uzel UID, potom tento uzel volbu vyhrál. Potom zahájí druhou část algoritmu
  - *Vítěz volby se odznačí jako účastník a pošle svoje číslo dále*
  - *Každý, kdo obdrží zprávu a je stále účastníkem, si číslo poznačí, odznačí se jako účastník a pošle zprávu dále.*
  - *Pokud zprávu obdrží vítěz volby, zprávu zahodí*

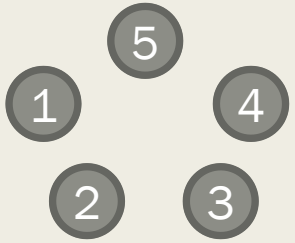

2
3
4
5
1
2
2
4
4
5
4
1
3
2
3
4
5
5
1
5
3
2
3
4
5
5
1
2
5
2
3
4
5
1
5
2
3
4
5
1
2
3
4
5
1
5
5
2
[5]
3
4
5
1
[5]
5
2
[5]
[5]
3
4
5
1
5
2
[5]
[5]
3
4
5
1
[5]
5
2
[5]
[5]
3
4
5
1
[5]

# Volba master uzlu, Chang and Roberts, analýza algoritmu

- Nejlepší případ – hlasování zahájí pouze uzel s nejvyšším UID. Pak proběhne 2n zpráv
- Nejhorší případ – uzel s maximálním indexem je první za iniciátorem hlasování a každý uzel zahájí hlasování před odesláním první zprávy některým z uzlů
  - *V nejhorším případě se musí přeposlat 3(n-1) zpráv pro jeden uzel*
  - *Celkově je ale možné pracovat až s $O(n^2)$ zprávami*

    *(n*(n-1)/2)+n*

    - Uzly od největšího po nejmenší (ve směru posílání zpráv
    - Každý uzel inicializuje volbu
    - Pak v první fázi 1+2+3+4+5=15 zpráv



# Volba master uzlu, Hirschberg-Sinclair

- Počet zpráv i časová složitost O(n*log n)

Všechny uzly začínají výpočet zároveň, zašlou **ELECTION**(UID,0,1) pravému i levému sousedovi

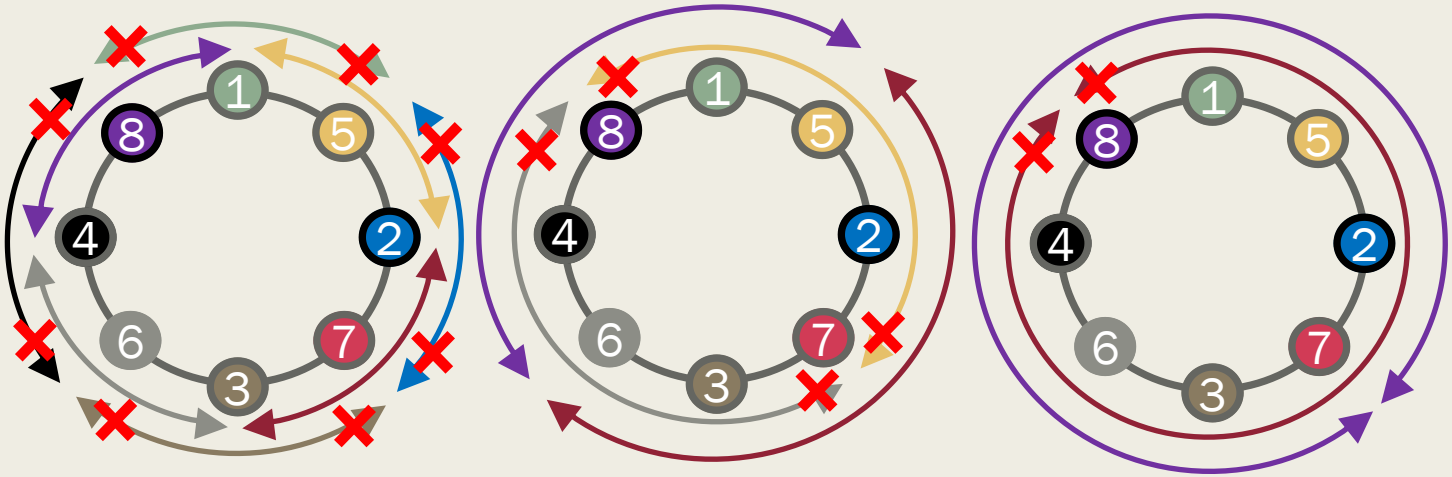
Po přijetí **ELECTION**(UID r, d) procesem *i* s $UID_i$
- if UID < $UID_i$ then zahoď zprávu
- elseif UID = $UID_i$ then zašli levými sousedovi **ELECTED**(UID)
- elseif d < $2^r$ zašli **ELECTION**(UID, r. d+1) po směru zprávy (‘pošli dál’)
- elseif d >= $2^r$ zašli **REPLY**(UID, r) směrem, odkud přišla zprávu (‘odpověz’)

Pří přijetí zprávy **REPLY**(UID, r)
- if UID ≠ $UID_i$ then zašli **REPLY**(UID, r) po směru zprávy
- elseif uzel obdržel **REPLY**(UID, r) z obou směrů, then pošli **ELECTION**(UID r+1, d) po obou směrech

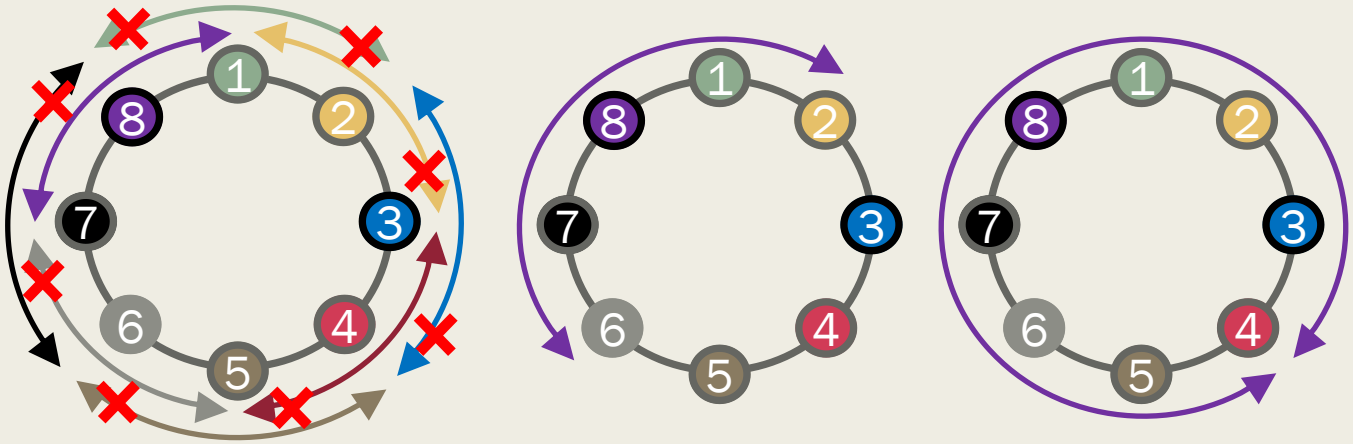
Po přijetí zprávy **ELECTED**(UID)
- if UID ≠ $UID_i$ then poznač si UID a přepošli zprávu **ELECTED**(UID) po směru zprávy

# Volba master uzlu, Hirschberg-Sinclair

Příklad 1




Příklad 2



# Volba master uzlu, Hirschberg-Sinclair

- V jednom běhu usne minimálně polovina uzlů, maximálně až na jeden všichni
- Jediný z uzlů zůstane zaručeně po $\log_2 n$ bězích
- V posledním z běhů bude zasláno 4n zpráv
  - *V příkladech na předchozím slidu chybí poslední běh*
  - *A také informování ostatních uzlů o výsledku volby*
- Proto je zaručeno, že třída počtů zpráv je O(n*logn)

# SYNCHRONIZACE LOGICKÉHO ČASU

LAMPORTOVY HODINY, LAMPORTŮV A RA ALGORITMUS
MEAKAWŮV ALGORITMUS
RAYMONDůV ALGORITMUS
CHANG A ROBERTSŮV ALGORITMUS

# Požadavky na vzájemné vyloučení

- Vzájemné vyloučení (Mutual exclusion) – v kritické sekci smí být v jednom okamžiku nejvýše jeden proces.
- Konzistence uspořádání (Ordering consistency) – systém musí zachovat definované pořadí konkurenčních požadavků.
- Bez uváznutí (Deadlock freedom) – systém se nesmí dostat do stavu, kdy žádný proces nemůže pokračovat.
- Bez vyhladovění (Starvation freedom) – každý proces žádající o vstup musí být nakonec obsloužen.
- Férovost (Fairness) – procesy mají získávat přístup ke kritické sekci spravedlivým způsobem.

# Mechanismy vzájemného vyloučení v distribuovaných systémech

- Obecně existují dva druhy mechanismů (algoritmů) pro zajištění vzájemného vyloučení
- Algoritmy založené na časových razítcích
  - *Lamportův algoritmus dist. vzájemného vyloučeni*
  - *Ricard-Agrawala-ův algoritmus*
  - *Meakawův algoritmus*
- Algoritmy založený na tokenech
  - *Suzuki-Kasami vysílací algoritmus*
  - *Raymondův stromový algoritmus*

# Lamportovy hodiny

- Nepotřebuje fyzický čas v distribuovaných systémech
- Absence globálních hodin, synchronizovaných hodin, nebo master uzlu
- Relace Happened-before (stalo se před *tím*) / **kauzalita**

**R**(e1,e2) iff e1 předchází e2 v rámci jednoho procesu

e1 je **send**(p2,m,ix) v procesu p1 a e2 je **receive**(p1,m,ix) v procesu p2

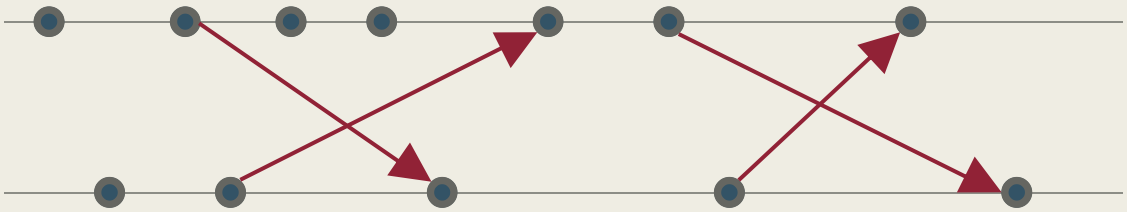
**R**(e1,e3) a **R**(e3,e2) // tranzitivita

# Synchronizace s Lamportovými hodinami

- Lamportovy logické hodiny
  - *Každý proces i má jedny logické hodiny Ci*
  - *Log. hodiny před každou událostí e zvyšují čítač (monotónně, tj. např o 1)*
  - *C(e) zobrazuje pro každou událost odpovídající logický čas*
  - *Požadujeme, aby*

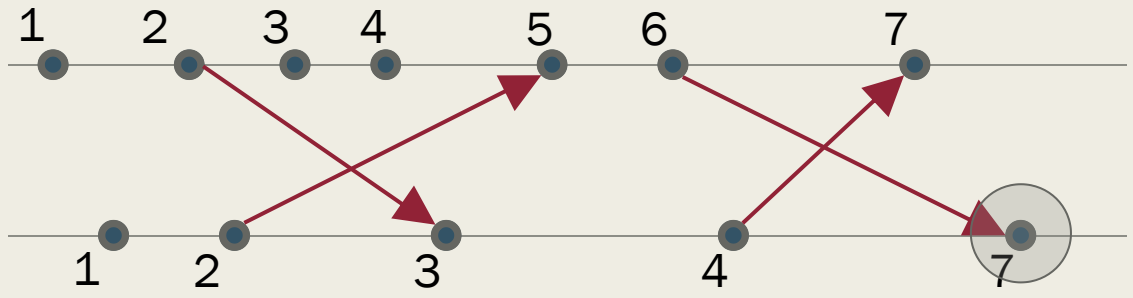
$$a \blacktriangleright b \Rightarrow C(a) < C(b)$$

# Implementace logických hodin



- Spolu se zprávou se posílá i časové razítko dle logického času vysílacího procesu
- Při přijetí zprávy *rec(m,p,tp)* příjemce *i* nastaví/aktualizuje svůj logický čas Ci:=max(Ci+1, tp+1)

# Implementace logických hodin




- Spolu se zprávou se posílá i časové razítko dle logického času vysílacího procesu
- Při přijetí zprávy *rec(m,p,tp)* příjemce *i* nastaví/aktualizuje svůj logický čas Ci:=max(Ci+1, tp+1)

# Poznámky k logickým hodinám

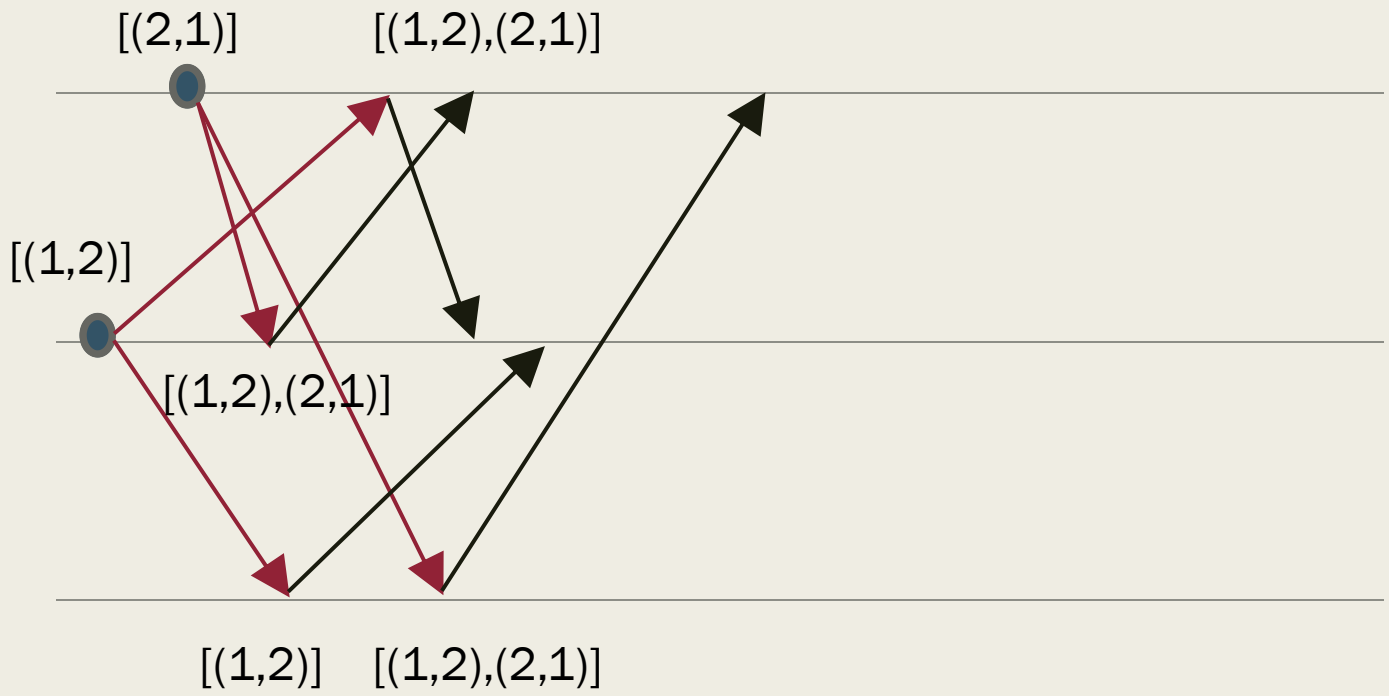
- “Happens-before” je nereflexivní relace částečného uspořádání
- Abychom dosáhli úplného uspořádání je třeba dodat další informaci, zde lze použít ID procesů, pak procesy s nižším číslem ‘mají přednost’ a jejich události, pro které nemůžeme uspořádání původně určit, předcházejí událostem procesů s vyšším číslem.
- Nerozlišujeme zvýšení logického času na základě toho, jestli k němu došlo vnitřní událostí, nebo na základě komunikace

# Lamportův algoritmus

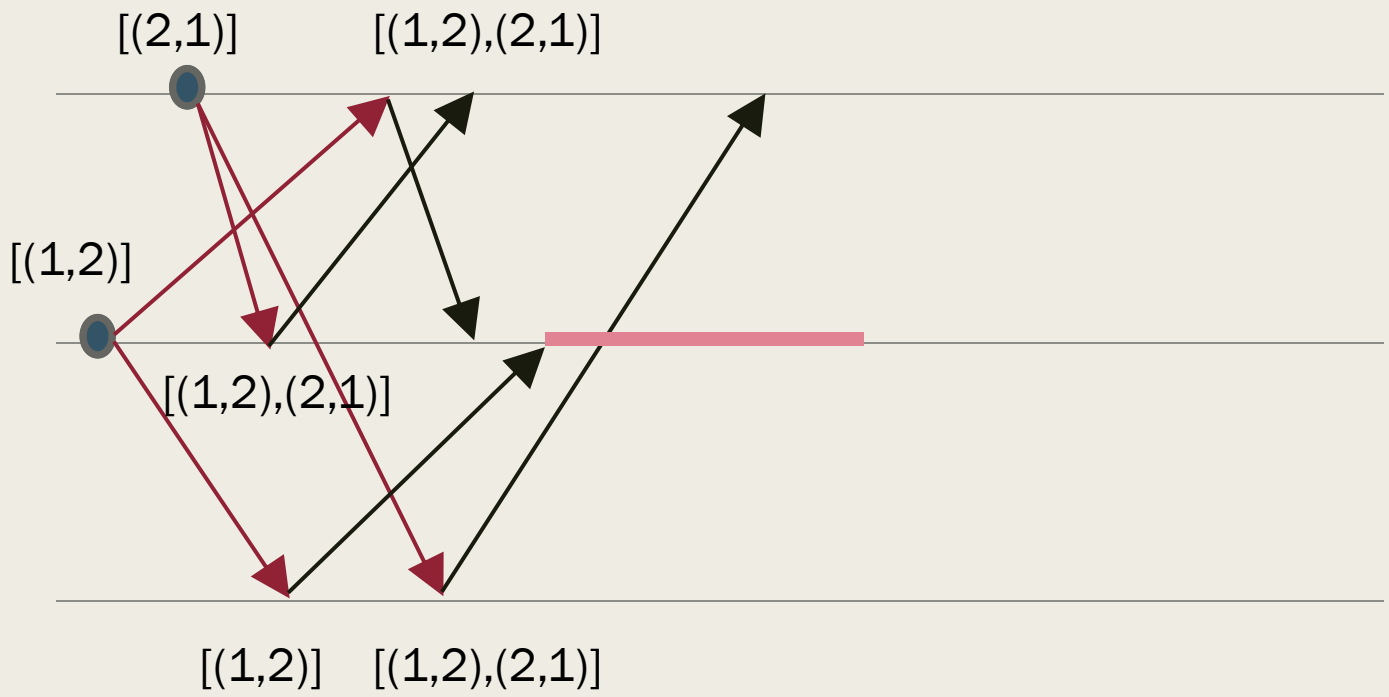
- Procesy hledají shodu navzájem na tom, který z nich získá kritickou sekci
- Založeno na FIFO doručování zpráv
- Udržuje lokální prioritní frontu zpráv ve které priority jsou úplně uspořádány podle předávaných časových razítek

# Lamportův algoritmus (požadavek)

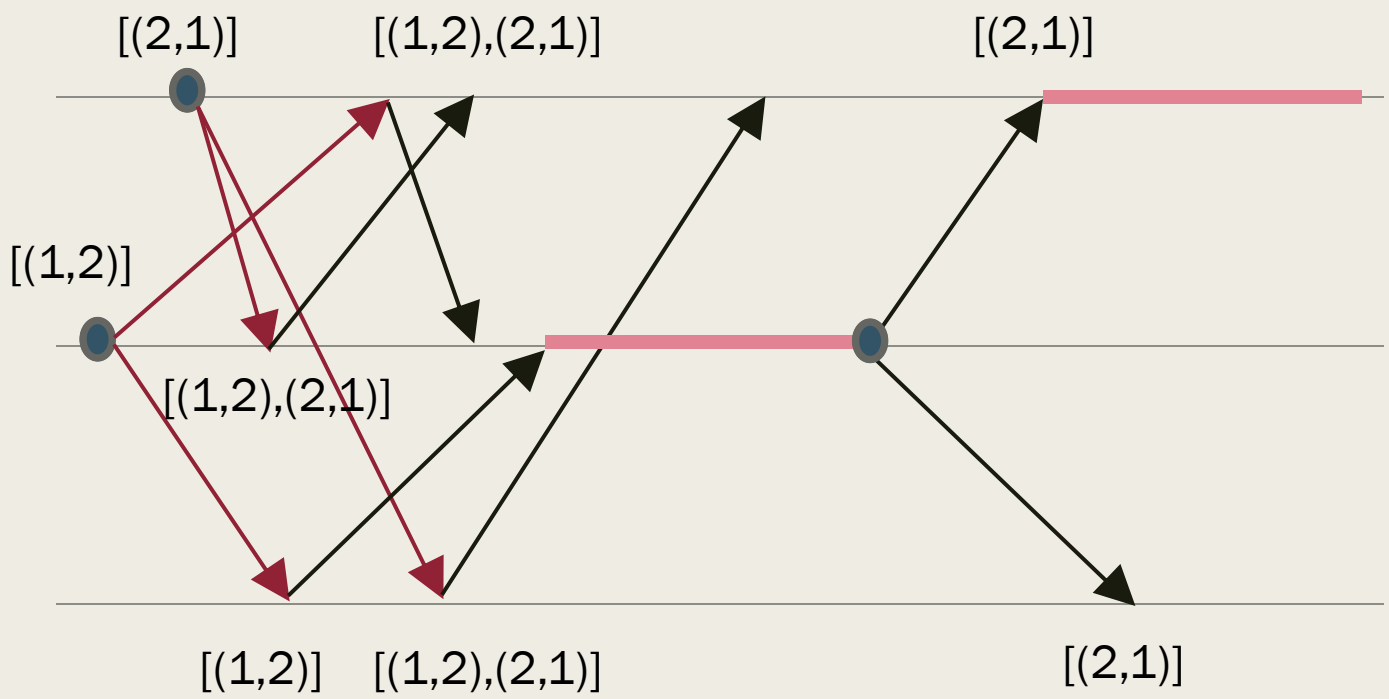
# Lamportův algoritmus (odpověď)



# Lamportův algoritmus (kritická sekce)



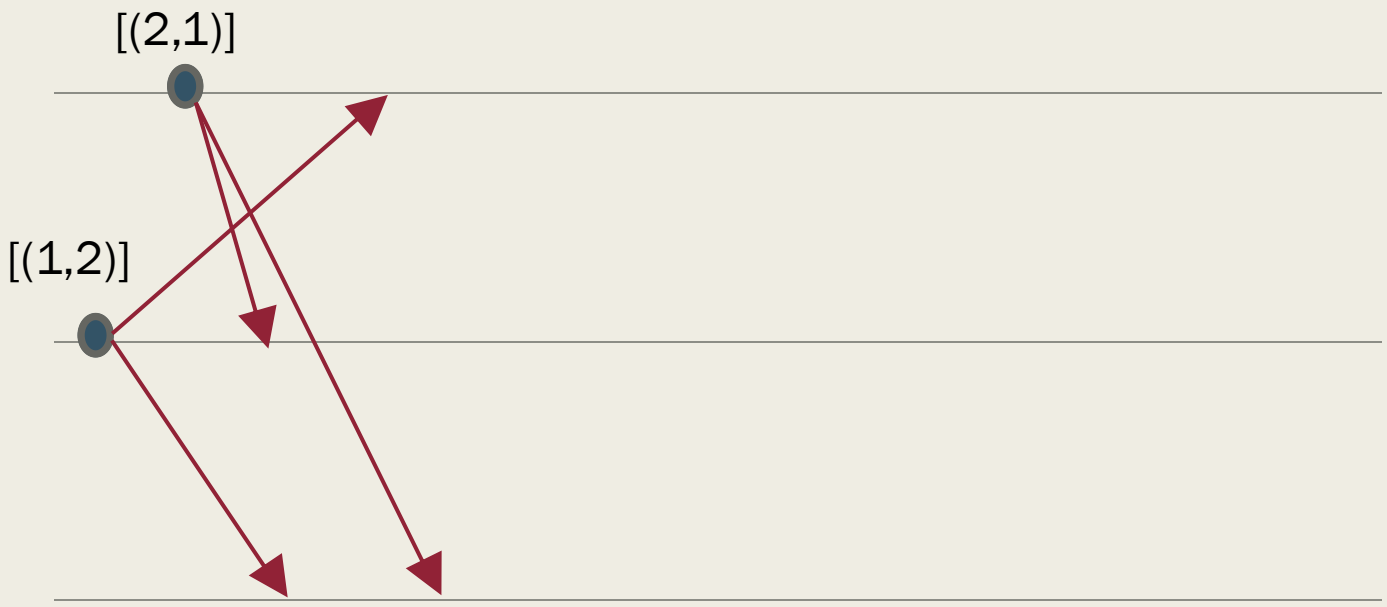
# Lamportův algoritmus (uvolnění kritické sekce)



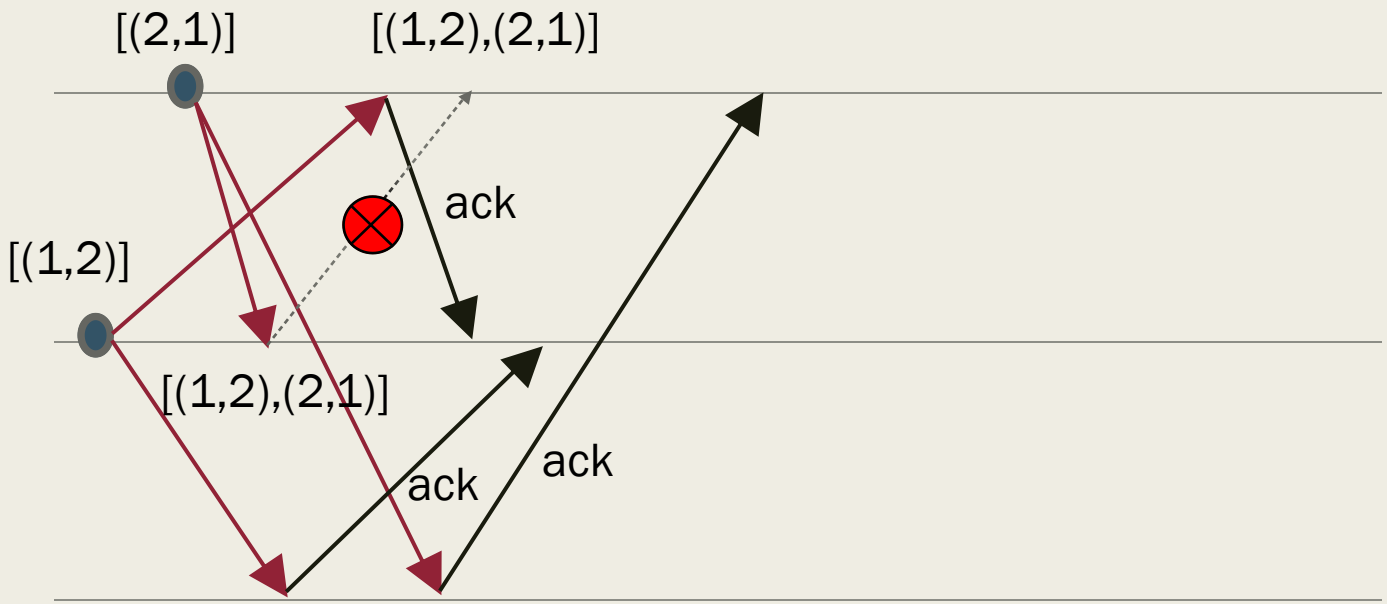
# Algoritmus Ricart-Agrawala

- Lamportův algoritmus vyžaduje 3(n-1) zpráv
  - *(n-1) požadavků*
  - *(n-1) odpovědí*
  - *(n-1) uvolnění*
- Algoritmus Ricart-Agrawala
  - *Jedná se o optimalizaci Lamportova algoritmu*
    - Slučuje dohromady zprávy odpovědi a uvolnění
  - *Celkem se tak posílá pouze 2(n-1) zpráv*
  - *Synchronizační zpoždění je opět jedna zaslaná zpráva*

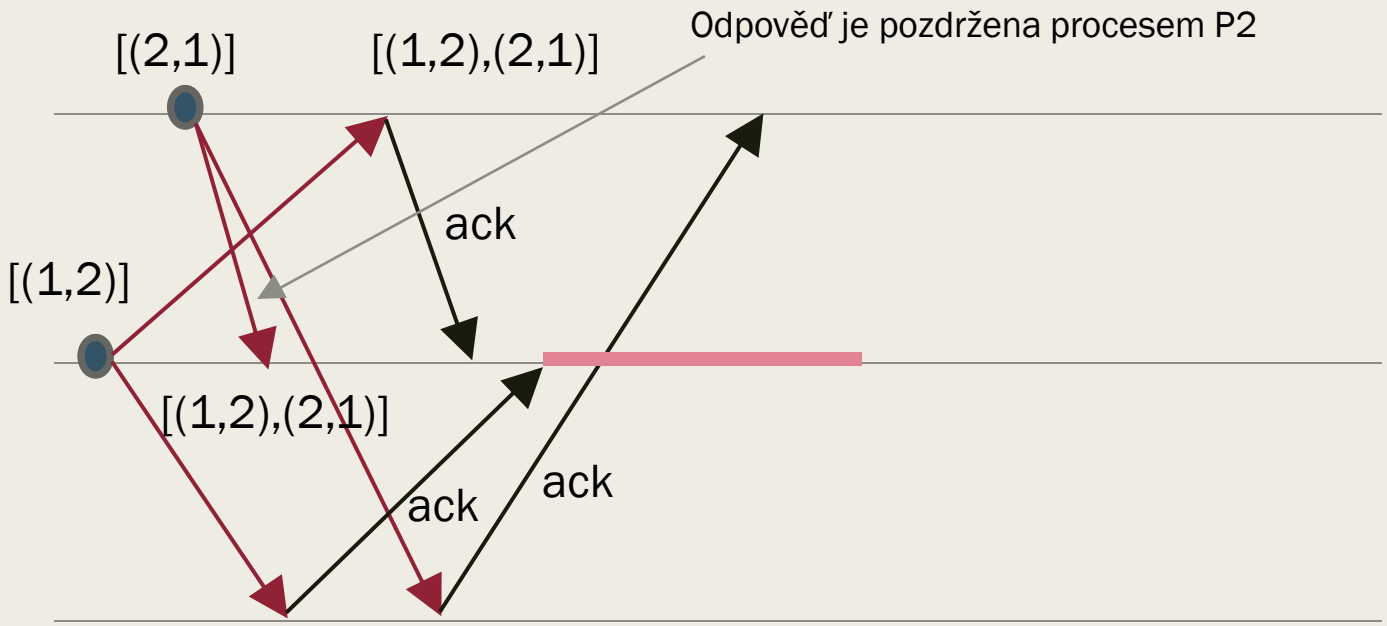
# Algoritmus Ricart-Agrawala (požadavek)



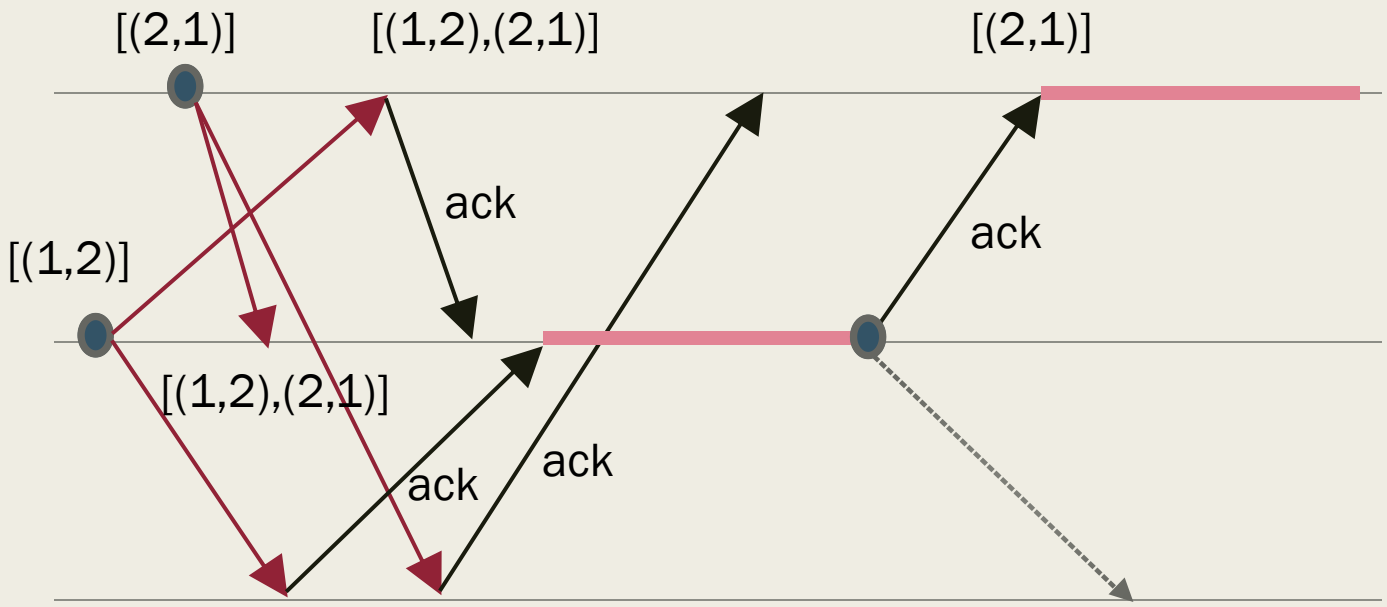
# Algoritmus Ricart-Agrawala (odpověď)



# Ricart-Agrawala algoritmus (kritická sekce)



# Ricart-Agrawala algoritmus (uvolnění kritické sekce)

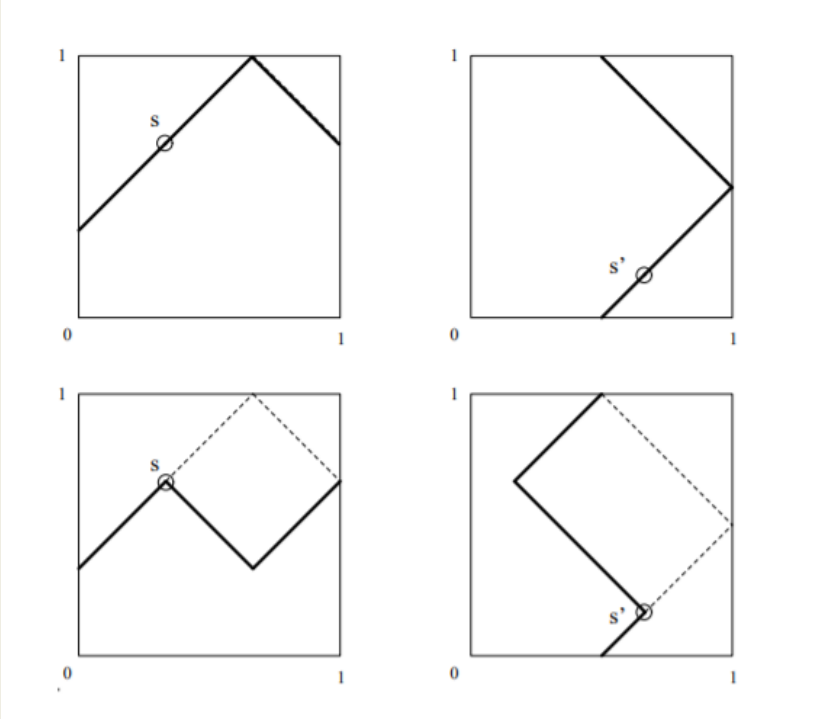


# Maekawův algoritmus

- Povolení je třeba jen od podmnožiny procesů
- Každý proces 1<=i<=N si udržuje seznam procesů Ri tak, že
  - *Každé dva procesy sdílí nějaký proces, tedy Ri∩Rj≠{}*
  - *Každý proces i je v Ri*
  - *Velikost těchto množin je K, K=* $\sqrt{2 * N + 1}$
  - *Každý proces je přítomen celkem K-krát v seznamech všech procesů*

# Maekawův algoritmus, vytvoření kvór

- Biliardová metoda pro vytváření kvór
- Hrajeme šťouch vždy v tom samém směru, například směrem dopravo a nahoru
- Přímý x zalomený šťouch (zamezí té samé posloupnosti pro všechny čísla na trase šťouchu před odrazem).



# Billiard pro 2*24+1=7^2

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | |
| 4 | | 5 | | 6 | | 7 |
| | 8 | | 9 | | 10 | |
| 11 | | 12 | | 13 | | 14 |
| | 15 | | 16 | | 17 | |
| 18 | | 19 | | 20 | | 21 |
| | 22 | | 23 | | 24 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | |
| 4 | | 5 | | 6 | | 7 |
| | 8 | | 9 | | 10 | |
| 11 | | 12 | | 13 | | 14 |
| | 15 | | 16 | | 17 | |
| 18 | | 19 | | 20 | | 21 |
| | 22 | | 23 | | 24 | |

# Zalomený billiard pro 24 uzlů

- V místě odpovídajícímu číslu procesu, pro který hledáme kvórum, zalomíme na opačnou stranu.
- Zalomíme zpět tak, aby cílové pole bylo stejné jako v nezalomeném případě.

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | |
| 4 | | 5 | | 6 | | 7 |
| | 8 | | 9 | | 10 | |
| 11 | | 12 | | 13 | | 14 |
| | 15 | | 16 | | 17 | |
| 18 | | 19 | | 20 | | 21 |
| | 22 | | 23 | | 24 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | |
| 4 | | 5 | | 6 | | 7 |
| | 8 | | 9 | | 10 | |
| 11 | | 12 | | 13 | | 14 |
| | 15 | | 16 | | 17 | |
| 18 | | 19 | | 20 | | 21 |
| | 22 | | 23 | | 24 | |

# Zalomený billiard pro 24 uzlů

- V místě odpovídajícímu číslu procesu, pro který hledáme kvórum, zalomíme na opačnou stranu.
- Zalomíme zpět tak, aby cílové pole bylo stejné jako v nezalomeném případě.

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | |
| 4 | | 5 | | 6 | | 7 |
| | 8 | | 9 | | 10 | |
| 11 | | 12 | | 13 | | 14 |
| | 15 | | 16 | | 17 | |
| 18 | | 19 | | 20 | | 21 |
| | 22 | | 23 | | 24 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | |
| 4 | | 5 | | 6 | | 7 |
| | 8 | | 9 | | 10 | |
| 11 | | 12 | | 13 | | 14 |
| | 15 | | 16 | | 17 | |
| 18 | | 19 | | 20 | | 21 |
| | 22 | | 23 | | 24 | |

# Maekawaův algoritmus, základní verze

- Žádající proces
  - *Pokud chce proces i vstoupit do kritické sekce ve stavu svých logických hodin ts, pošle request(ts,i) všem procesům z Ri*
  - *Pokud obdrží grant(j) od všech j z Ri, vstoupí do kritické sekce*
  - *Po opuštění kritické sekce pošle release(i) všem z Ri*
- Žádaný proces. Pokud proces i obdrží *request(ts,j)* od nějakého procesu *j*, pak
  - *pokud nevydal žádný aktuální grant jinému procesu, pošle grant(i) procesu j*
  - *pokud již vydal grant jinému procesu, pošel failed(i) a zařadí jej do fronty*
  - *Pokud obdrží zprávu release(j), pak odstraní proces j ze své fronty a pošle grant(k) případnému aktuálně prvnímu procesu ve frontě.*

# Maekawaův algoritmus, základní verze

- Výhoda - posílá $k\sqrt{N-1}$ zpráv kde k je mezi 3 a 6 (minimálně request, grant, release, // request, fail, grant, release, // request, inquiry, yield, grant, release, grant – ve verzi bez **uváznutí**)
- Problém – může nastat **uváznutí**

# Maekawaův algoritmus, uváznutí

- Uváznutí pro procesy P0 – P6 v základní verzi,
- Vytvoříme si quora R0-R6 (např billiardovou metodou)

R0={**0**,1,2}, R1={**1**,3,5}, R2={**2**,4,5}, R3={0,**3**,4}, R4={1,**4**,6}, R5={0,**5**,6}, R6={2,3,**6**}

- 0,1,2 chtějí do KS – co obdrží od svých Ri?
  - *0 <- od 0 a 2 dostane grant, od 1 ne (garantuje sebe)*
  - *1 <- od 1 a 3 dostane grant, od 5 ne (garantuje dvojku)*
  - *2 <- od 4 a 5 dostane grant, od 2 ne (garantovala nulu)*

Příklad:

*grant 0->0, grant 1->1, grant 3->1, failed 1->0, grant 2->0, failed 2->2, grant 4->2, grant 5->2, failed 5->1*

# Maekawaův algoritmus

- Žádající proces
  - *Pokud chce proces i vstoupit do kritické sekce ve stavu svých logických hodin ts, pošle request(ts,i) všem procesům z Ri*
  - *Pokud obdrží grant(j) od všech j z Ri, vstoupí do kritické sekce*
  - *Po opuštění kritické sekce pošle release(i) všem z Ri*
- Žádaný proces. Pokud proces i obdrží *request(ts,j)* od nějakého procesu j, pak
  - *pokud nevydal žádný aktuální grant jinému procesu, pošle grant(i) procesu j*
  - *pokud vydal grant procesu s vyšší prioritou, pošle failed(i) procesu j a zařadí si tento proces do fronty*
  - *jinak*
    - optá se procesu *k*, kterému dal grant zprávou *inquire(i)*
    - pokud obdrží zprávu *yield(k),* pak dá *grant(i)* a proces *k* si uloží do fronty

# Maekawaův algoritmus

- Spolupracující proces
  - *Proces i, který dostane zprávu inquire(j) od procesu j pošle zpět zprávu yield(i), pokud dostal na svoji žádost fail(k) od nějakého procesu ze svého Ri, nebo poslal dříve někomu jinému yeld(i) a neobdržel od něj grant(i) zpět*
  - *Pokud proces i obdrží zprávu release(j), pak odstraní j ze své fronty a pošle grant(k) případnému aktuálně prvnímu procesu ve frontě.*

# Maekawaův algoritmus, uváznutí vyřešeno

R0={0,1,2}, R1={1,3,5}, R2={2,4,5}, R3={0,3,4}, R4={1,4,6}, R5={0,5,6}, R6={2,3,6}

R0 > R1 > R2 ...

*grant 0 -> 0, grant 1 -> 1, grant 3 -> 1,*

*?? grant(1 -> 0): // vydal grant 1, R1<R0*

*inquire 1*

*grant 2 -> 0*

*failed 2->2 // vydal grant R0, R0>R2*

*grant 4-> 2*

*grant 5 -> 2*

*?? grant 5 -> 1: inquire 2 // vydal grant R2, R2 < R1*

*yeld(2) -> 1, grant 5->1*

***proces 1 je v kritické sekci***

*release 1->5, release 1->1 release 3->1*

*grant(1->0)*

***proces 0 je v kritické sekci***

*release 0 -> 0, release 0 -> 1, release 0 -> 2*

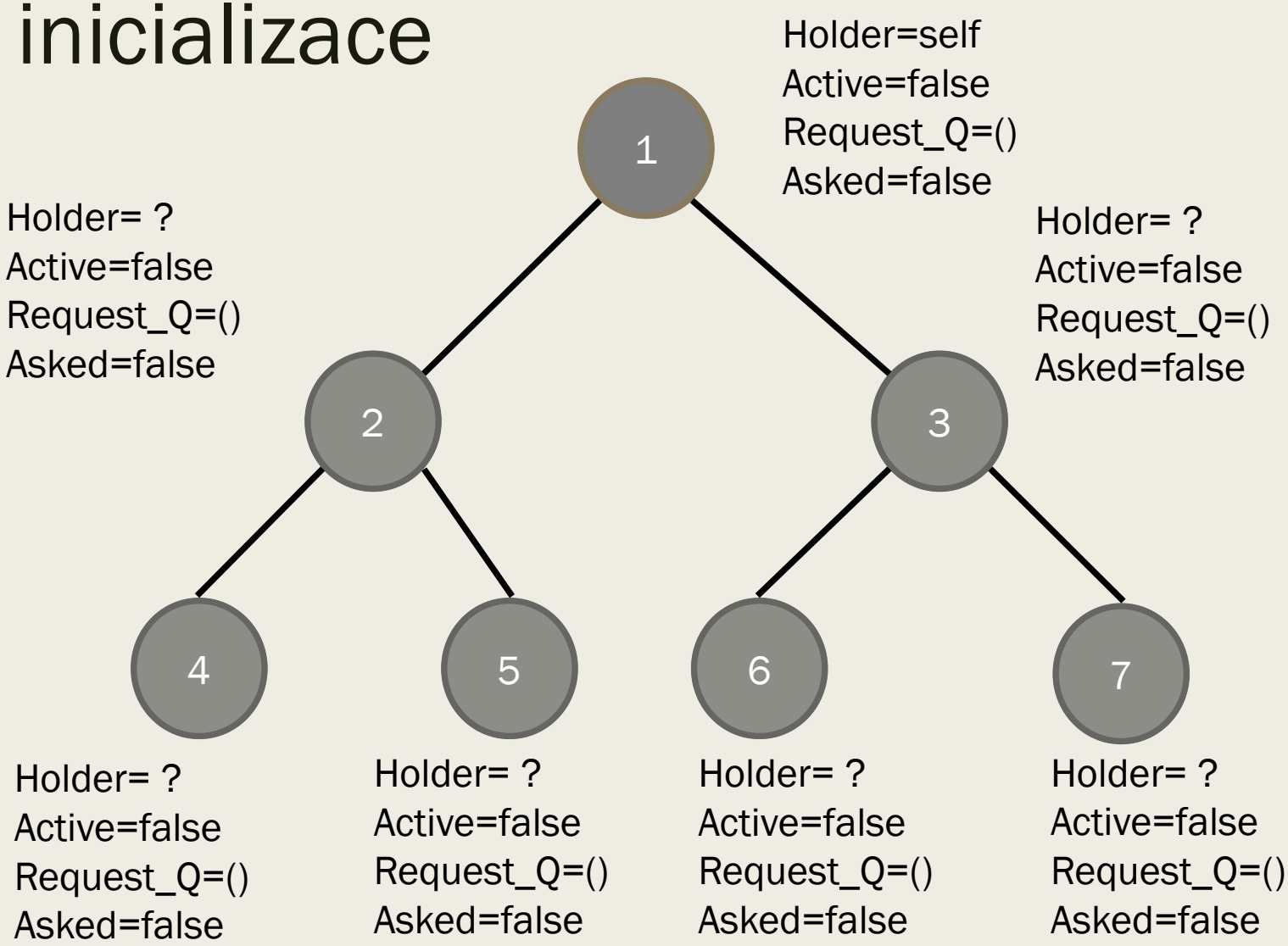
*grant 5 -> 2 // před tím yeld, je ve ftontě*

***proces 2 je v kritické sekci***

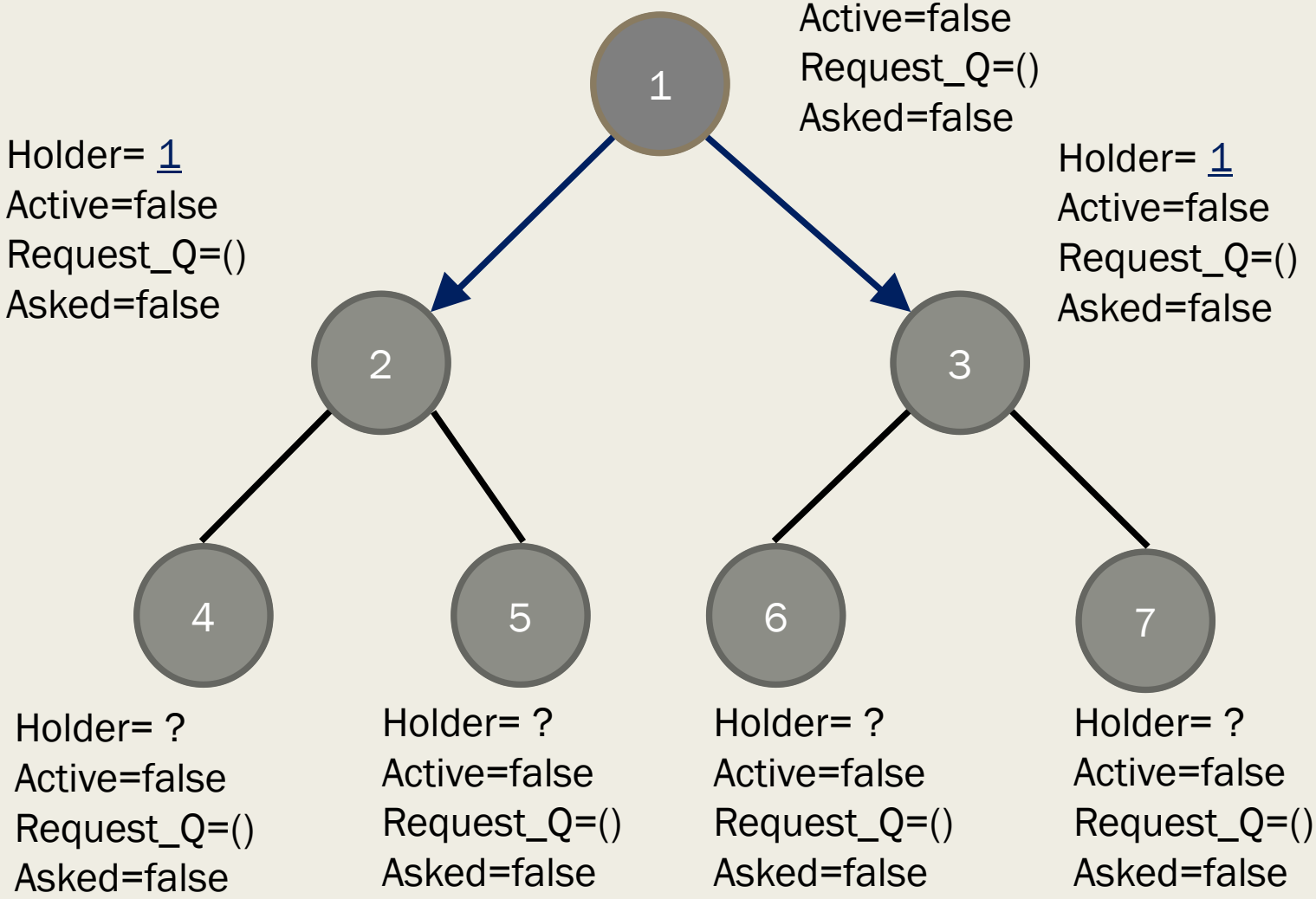
# Raymondův algoritmus

- Myšlenka
  - *Jedna virtuální značka/token reprezentuje v systému poukázku na kritickou sekci.*
  - *Proces může vstoupit do kritické sekce, pokud obdrží token*
  - *Důkaz o vzájemném vyloučení procesu je triviální (token může držet jen jeden proces a ten jej předává, pokud není v kritické sekci)*

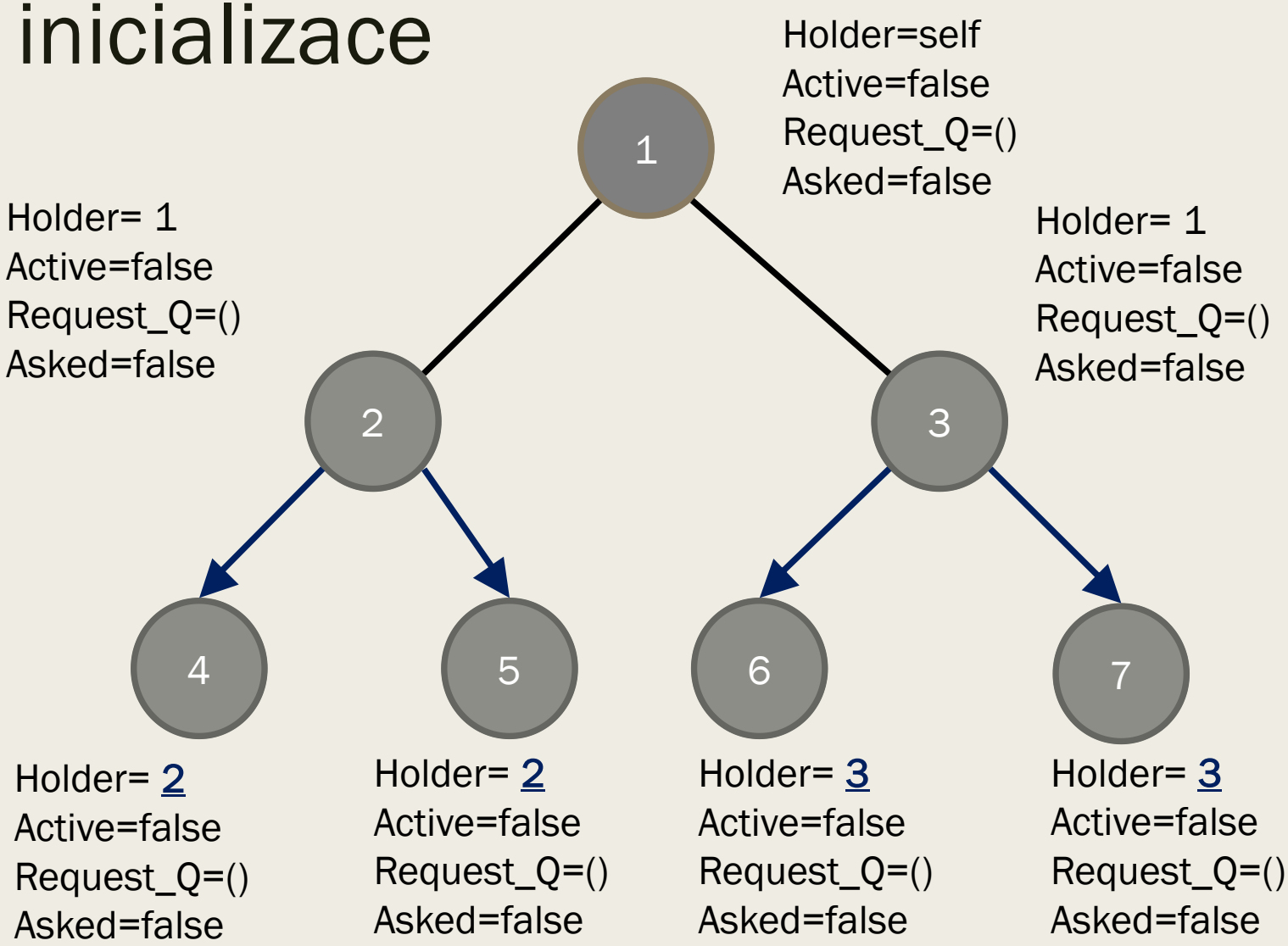
# Raymondův algoritmus, inicializace



# Raymondův algoritmus, inicializace



# Raymondův algoritmus, inicializace



# Raymondův algoritmus

- Proces je povinen předat token žádajícímu procesu, pokud jej drží, pokud jej právě nepoužívá a pokud takový požadavek je (včetně sebe, pak pouze zahájí používání kritické sekce)

```
ASSIGN-PRIVILEGE:
if HOLDER = self and not USING and REQUEST-Q # empty
    then
        HOLDER := dequeue (REQUEST-Q)
        ASKED := false
        if HOLDER = self
            then
                USING := true (initiate entry into critical section)
            else
                send PRIVILEGE to HOLDER
```

# Raymondův algoritmus

- Agent, pokud chce přístup do kritické sekce, vloží hodnotu ‘self’ do Request_q
- Agent odešle požadavek, pokud je žádán procesem (včetně sebe) o token, a nemá poznačeno, že si již ve směru ‘Holder’ zažádal

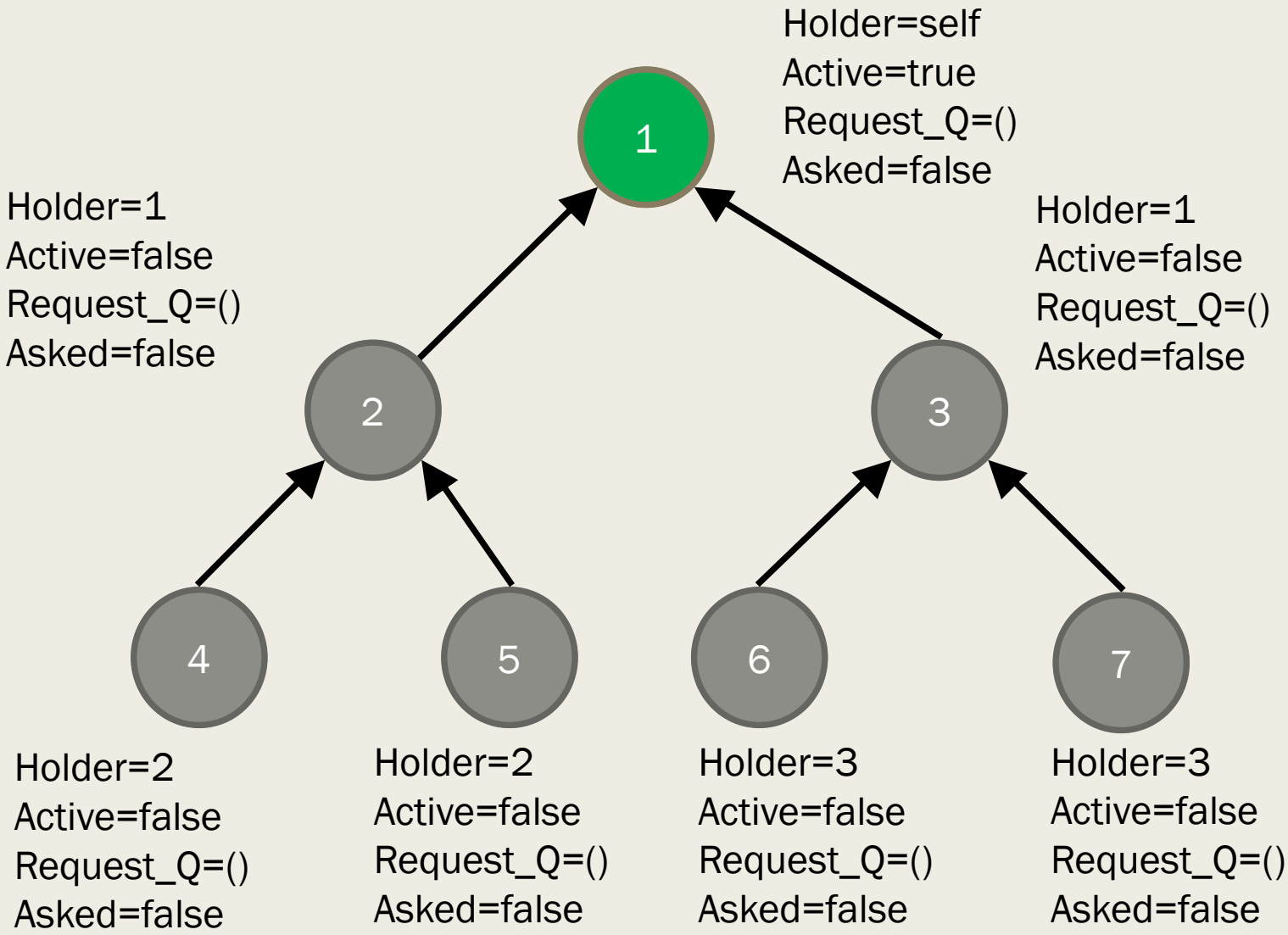
**MAKE- REQUEST:**

if HOLDER # self and REQUEST-Q # empty and not ASKED
then

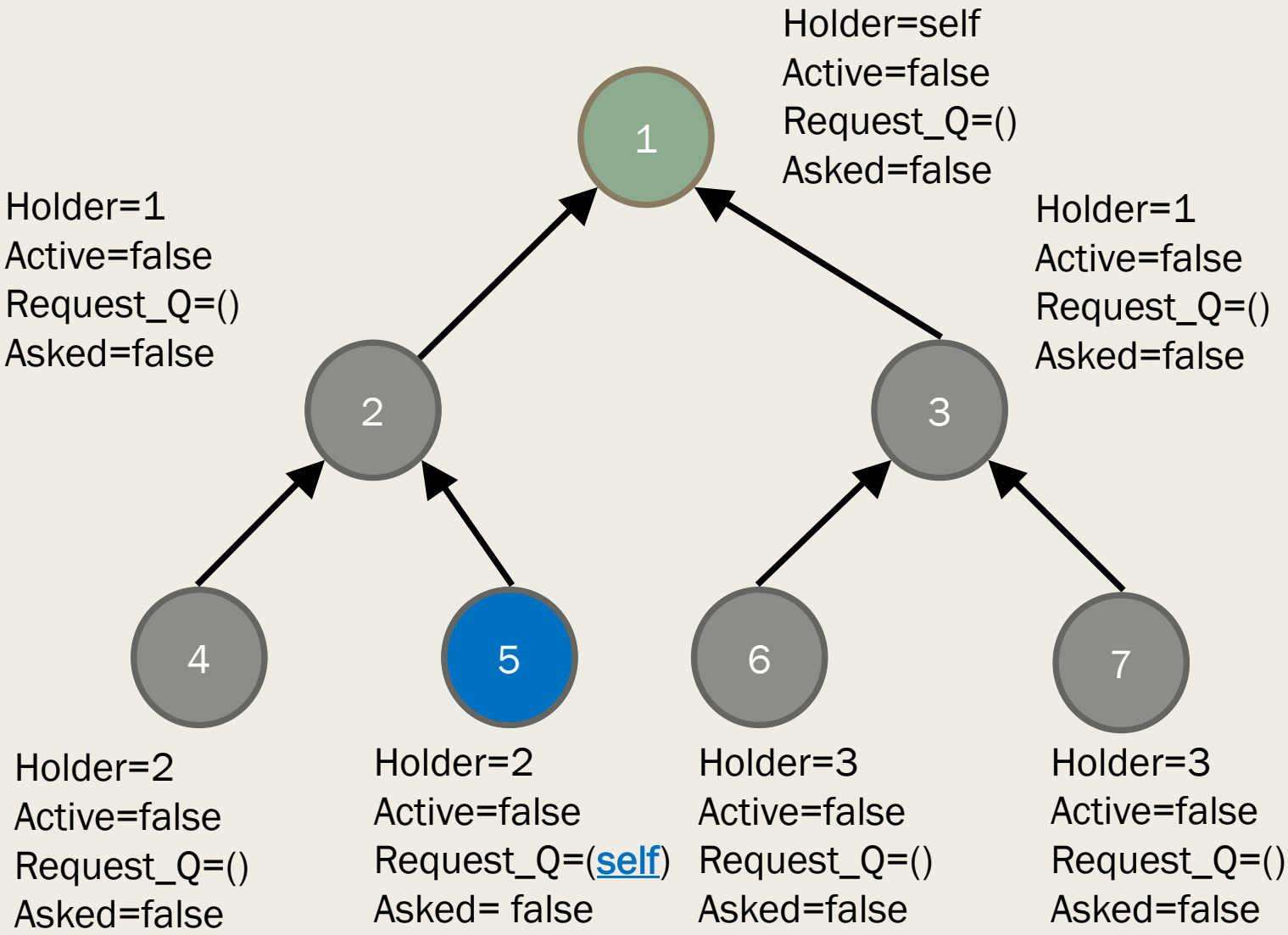
*send REQUEST to HOLDER*

*ASKED := true*

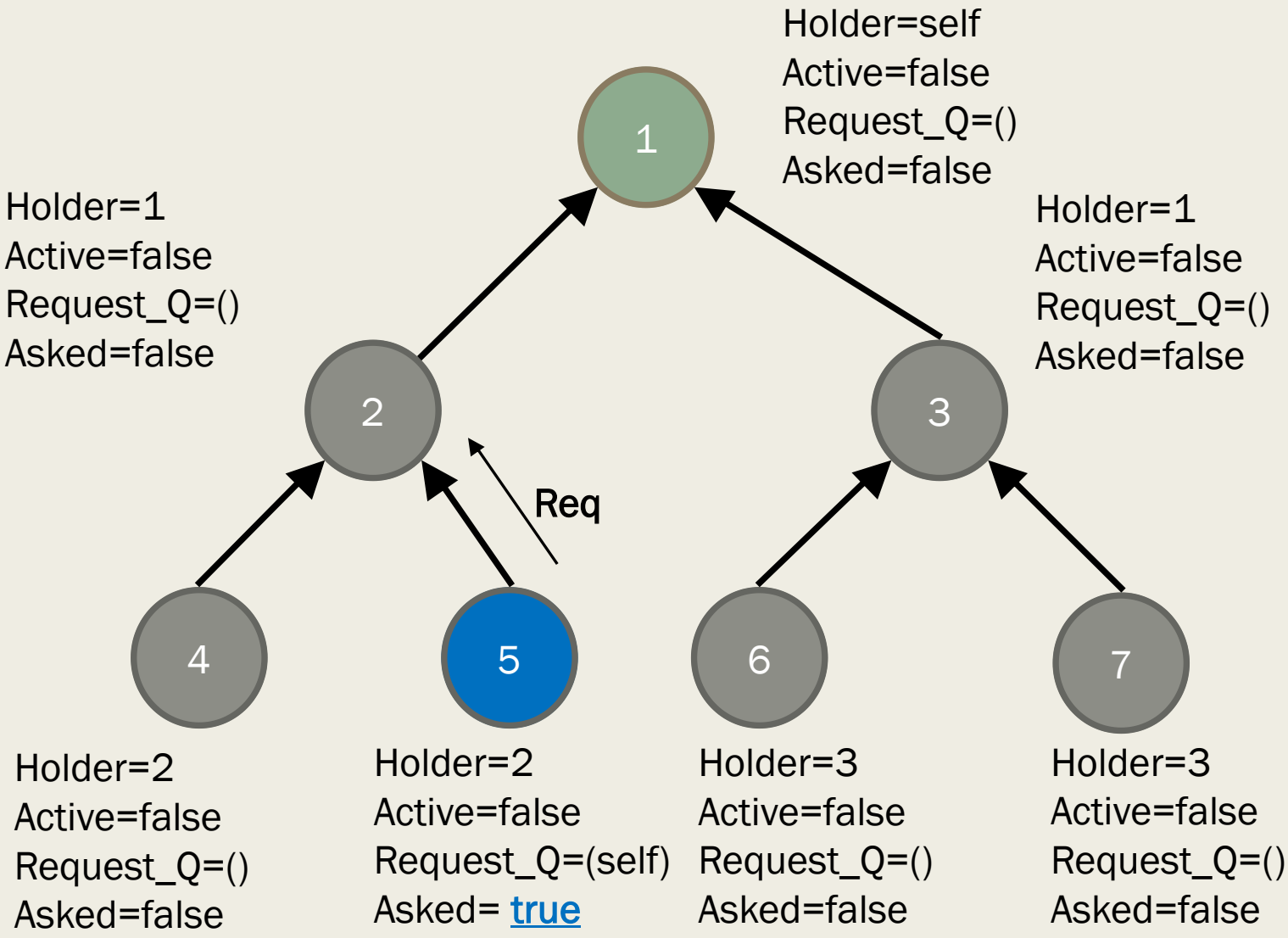
# Raymondův algoritmus



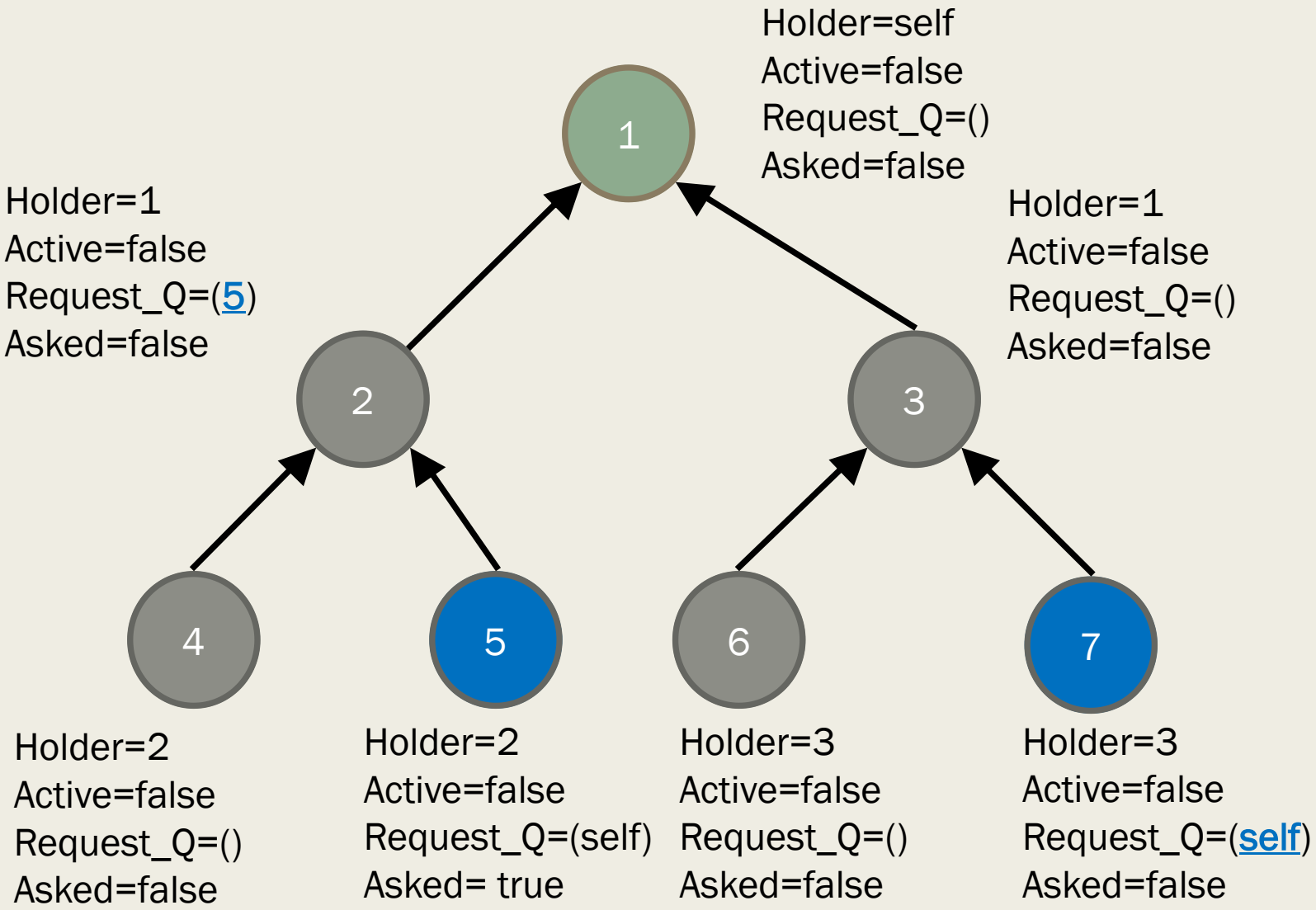
# Raymondův algoritmus



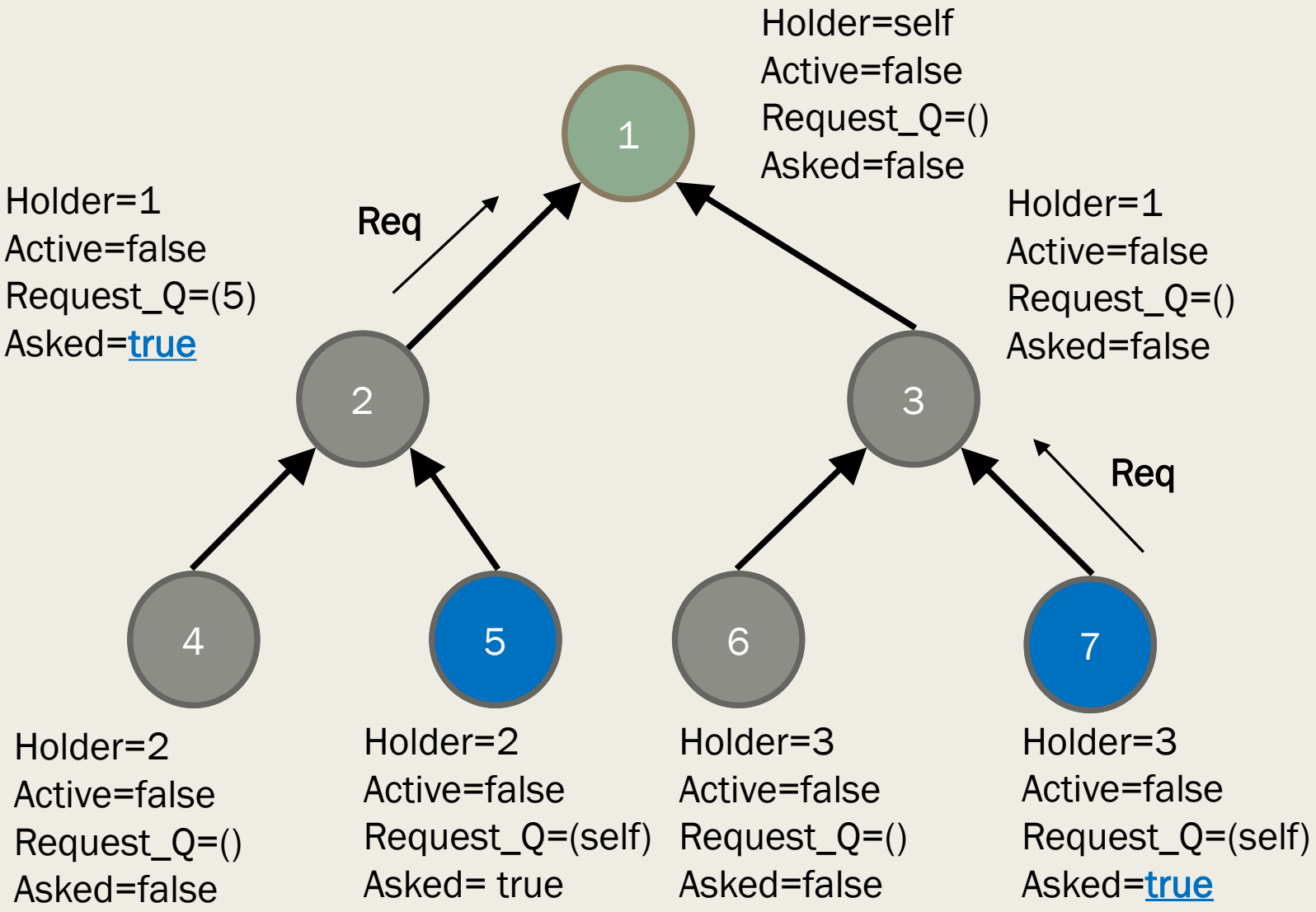
# Raymondův algoritmus



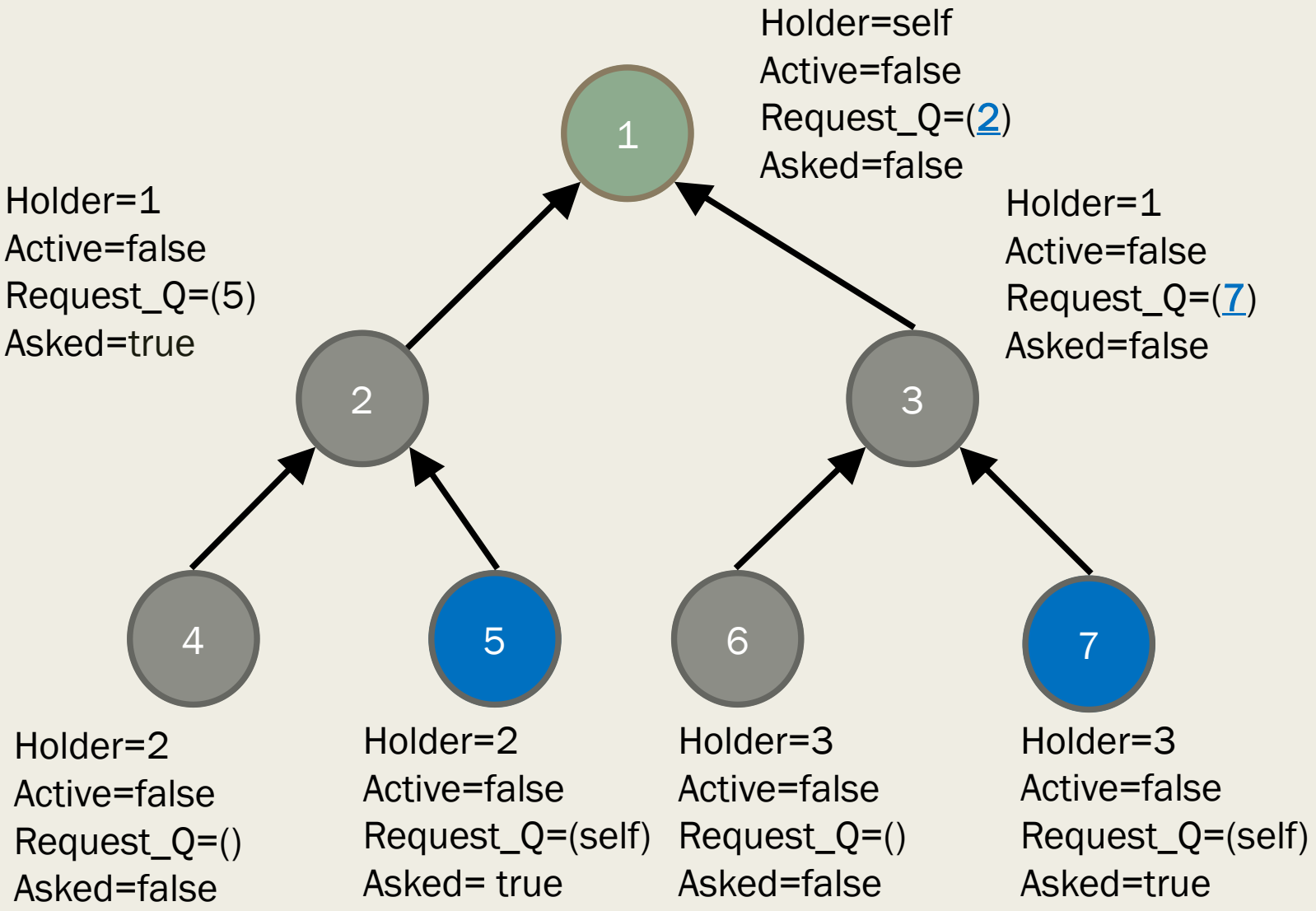
# Raymondův algoritmus



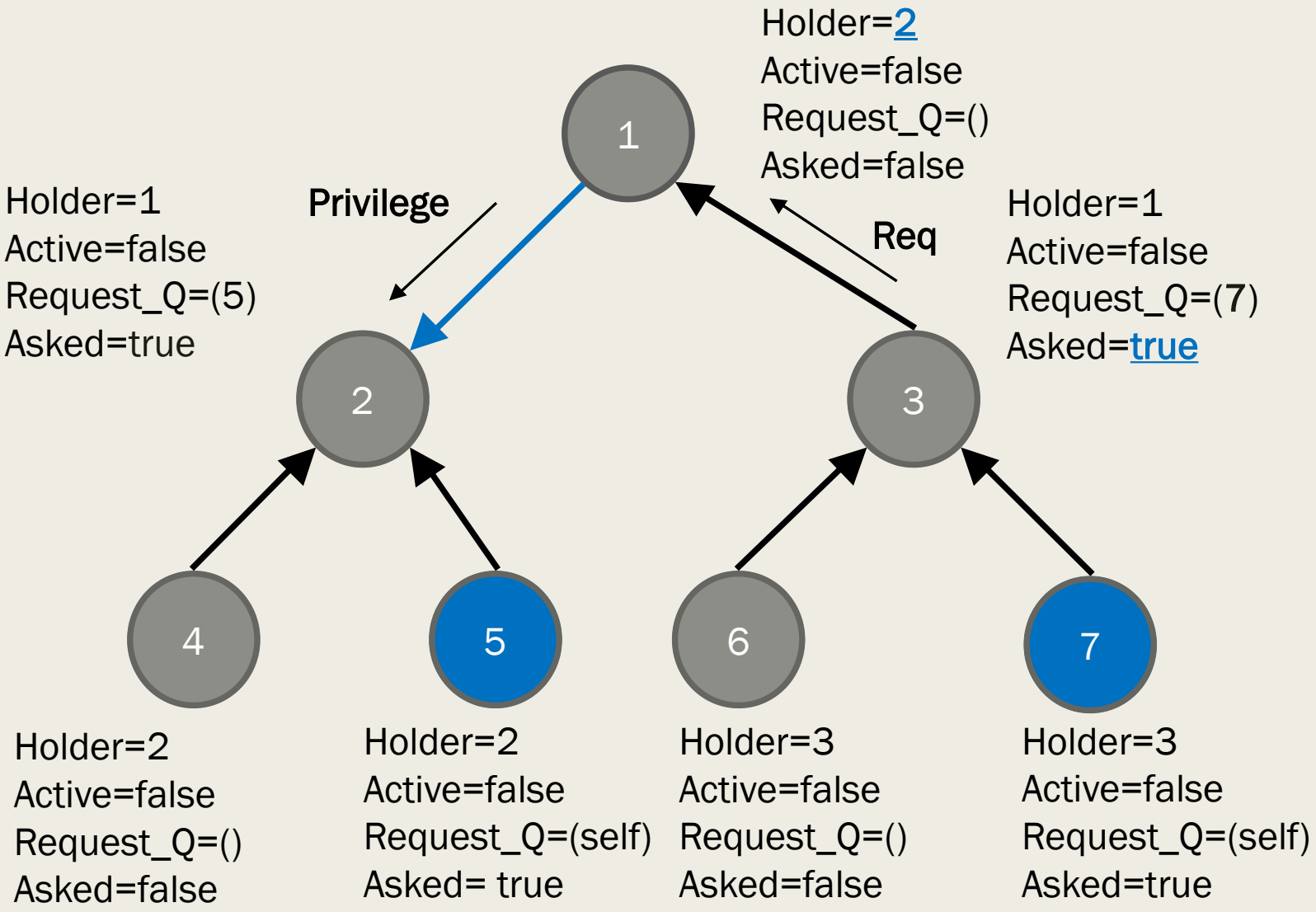
# Raymondův algoritmus



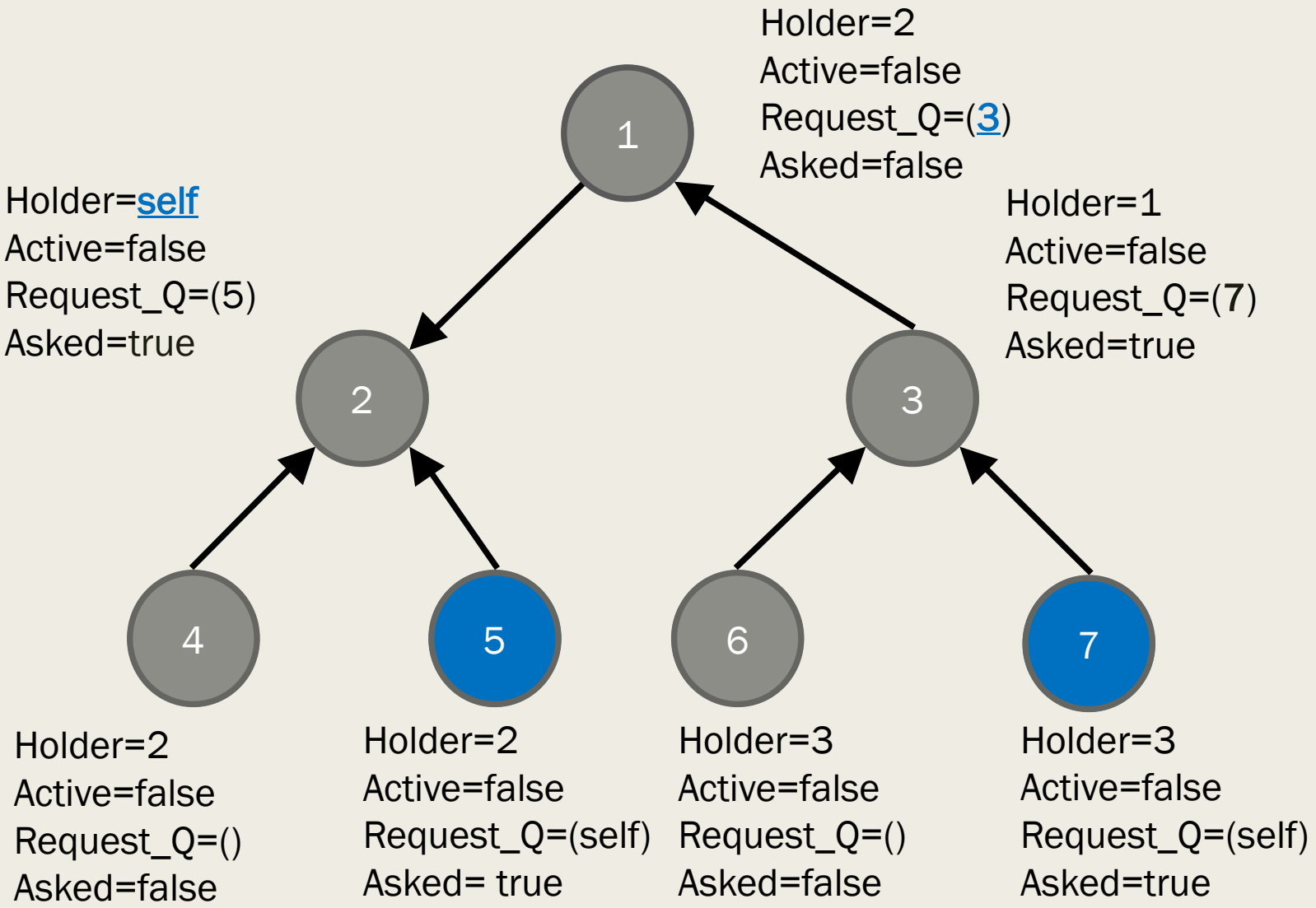
# Raymondův algoritmus



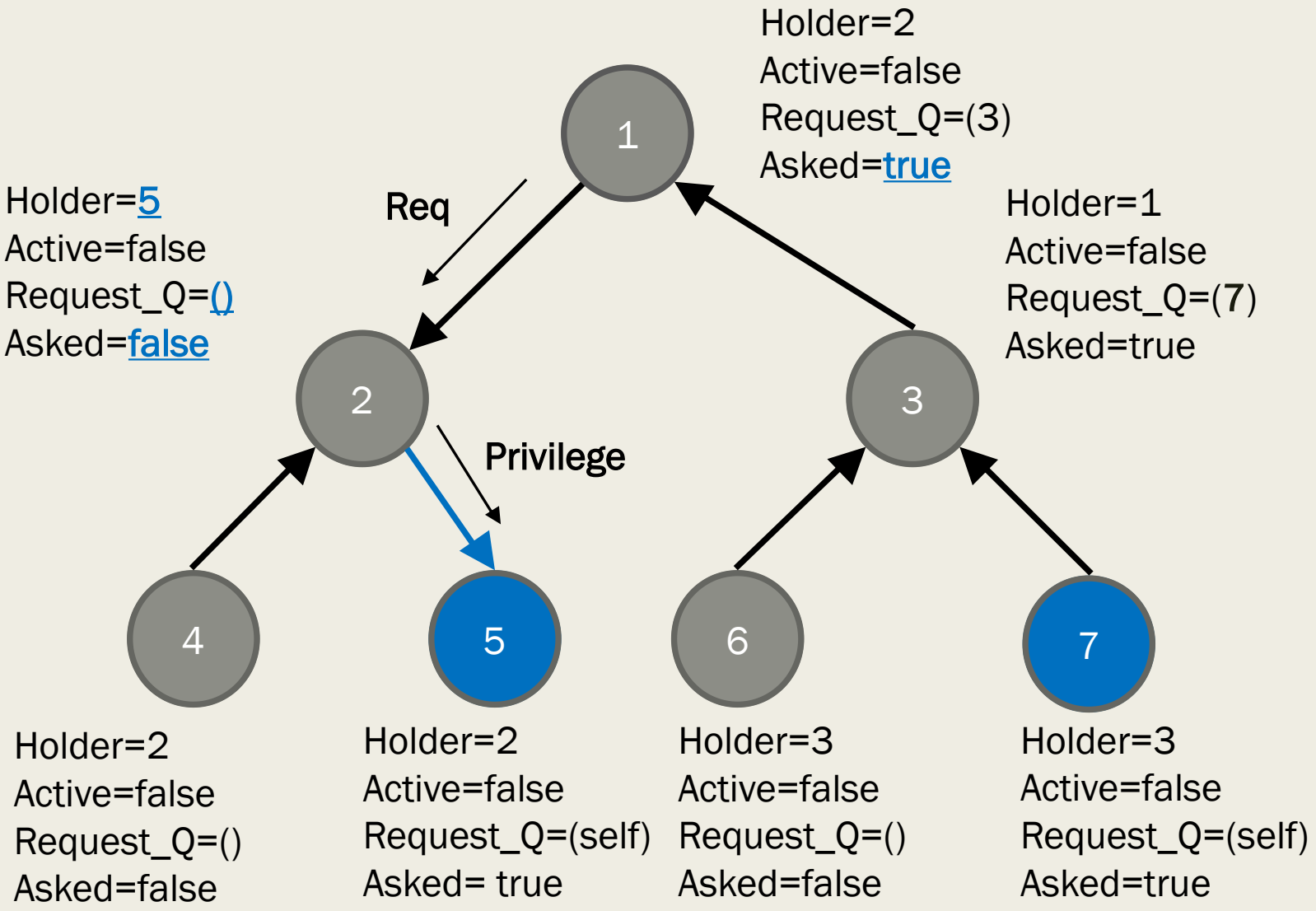
# Raymondův algoritmus



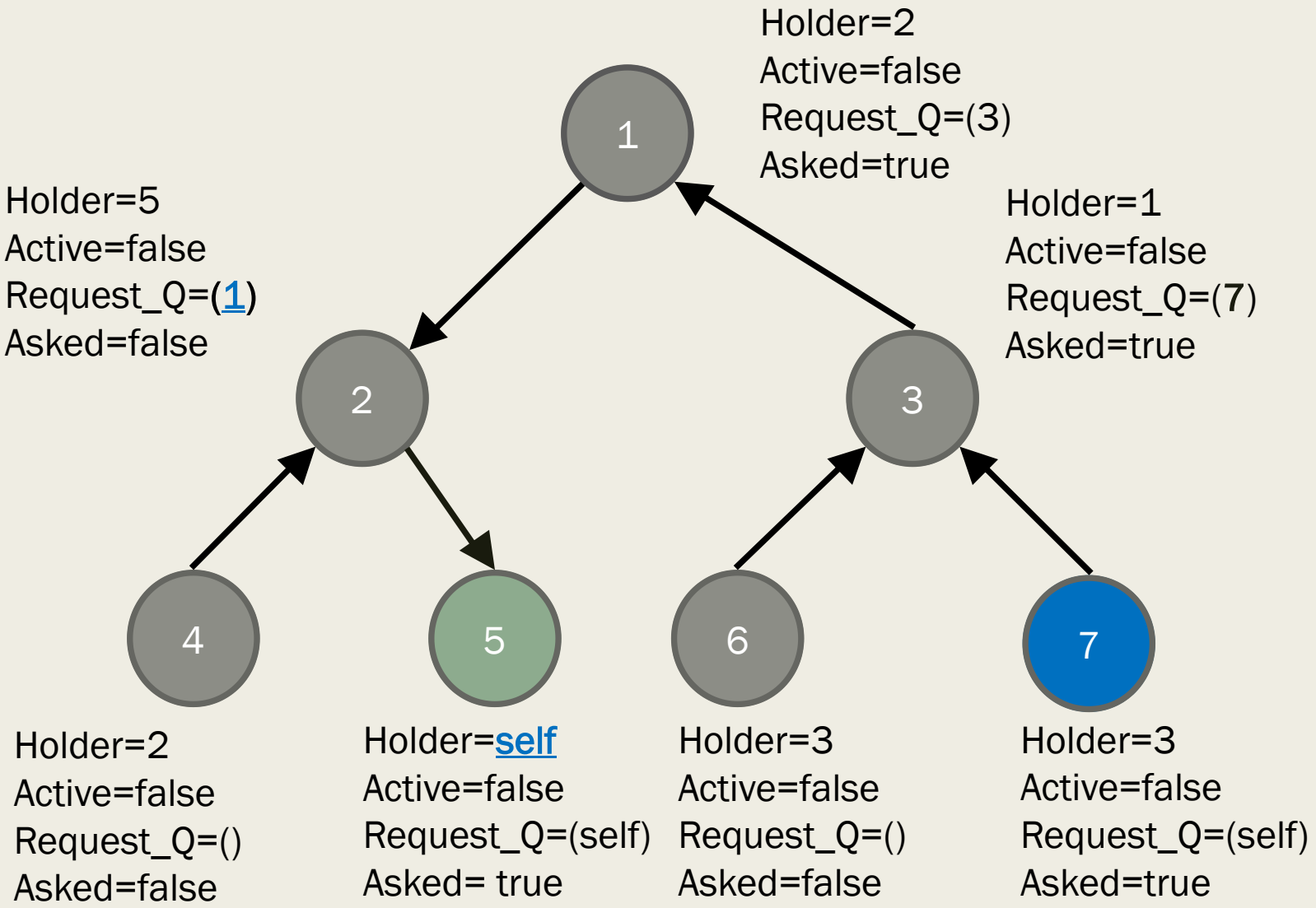
# Raymondův algoritmus



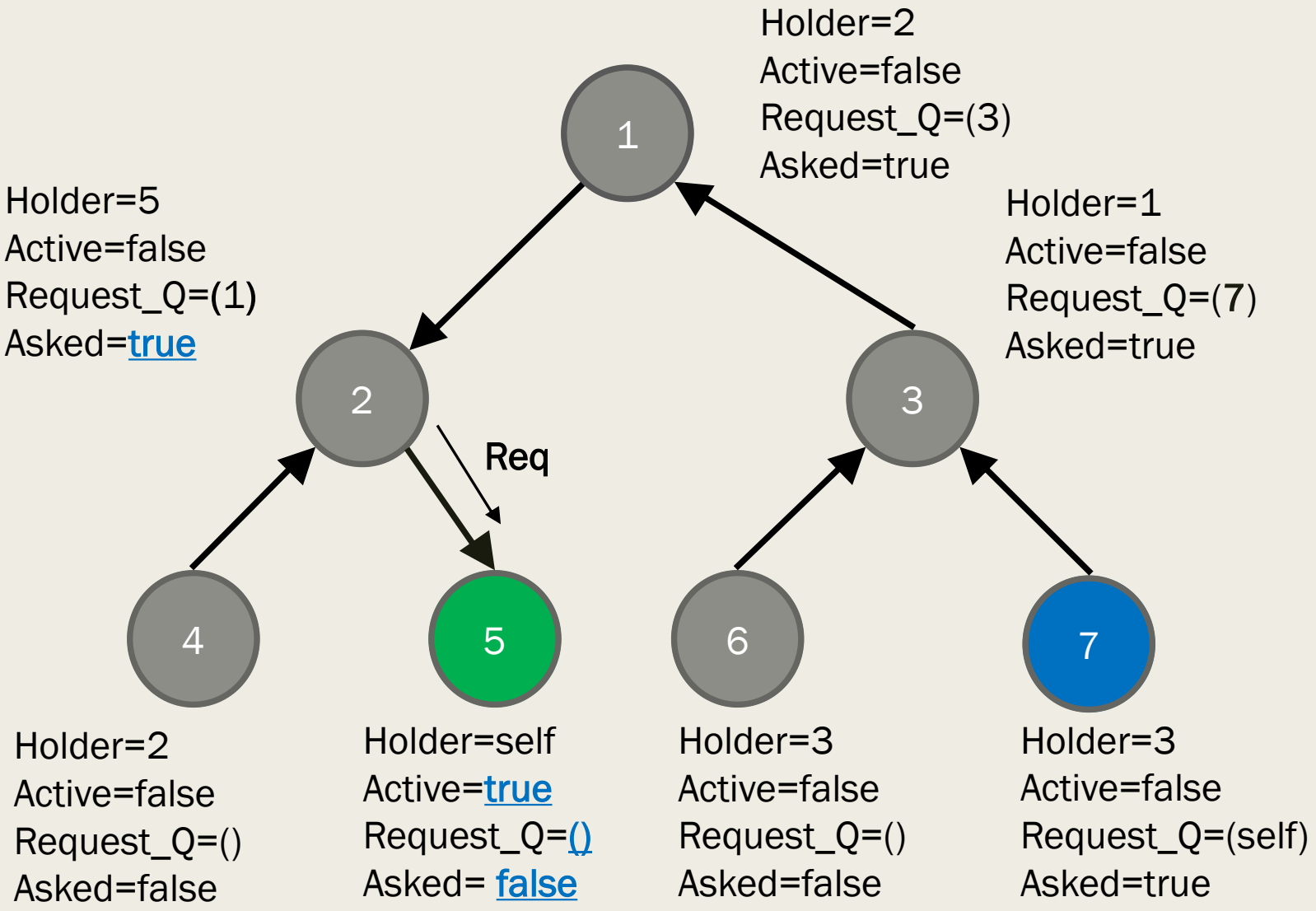
# Raymondův algoritmus



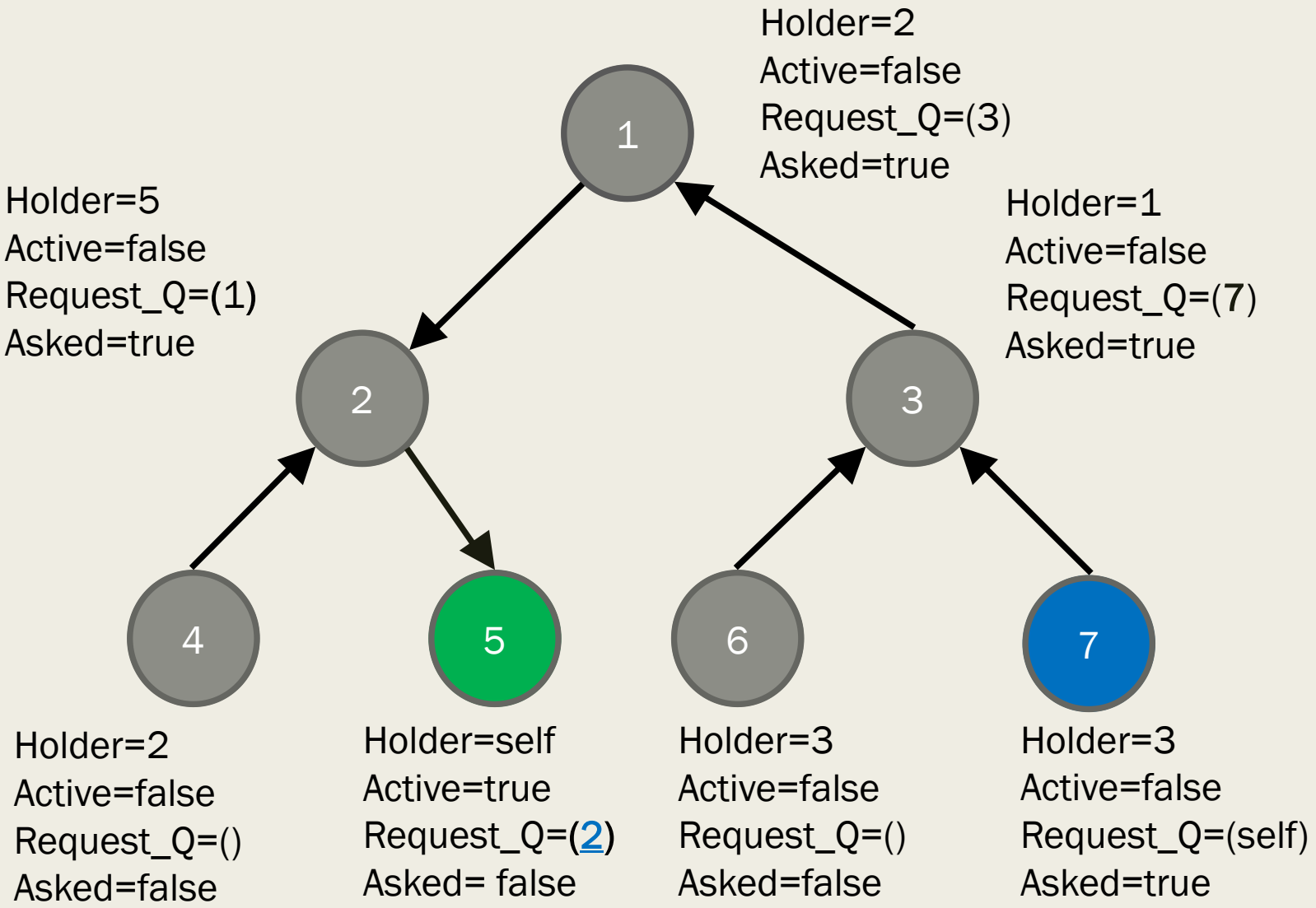
# Raymondův algoritmus



# Raymondův algoritmus



# Raymondův algoritmus



# Raymondův algoritmus, zhodnocení

- Vlastnosti
  - *(+) Nezpůsobuje vyhladovění*
  - *(+) Nezpůsobuje uváznutí*
  - *(+) Fault tolerance*
  - *(-) Přerušení jednoho komunikačního kanálů může odpojit i všechny procesy (konektivita = 1)*

# Analýza Raymondova algoritmu

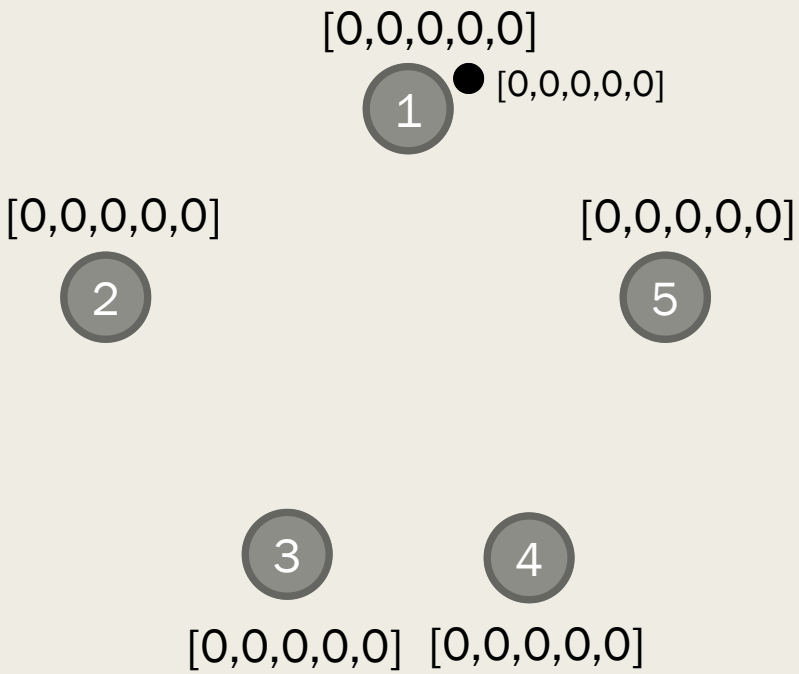
- Počet zaslaných zpráv – třída složitosti O(lg N)
- Průměrné synchronizační zpoždění je lg N/2
- Přenosnost se snižuje při zahlcení sítě zprávami
- Může použít greedy strategii - možnost hladovění

# Suzuki – Kasami vysílací algoritmus

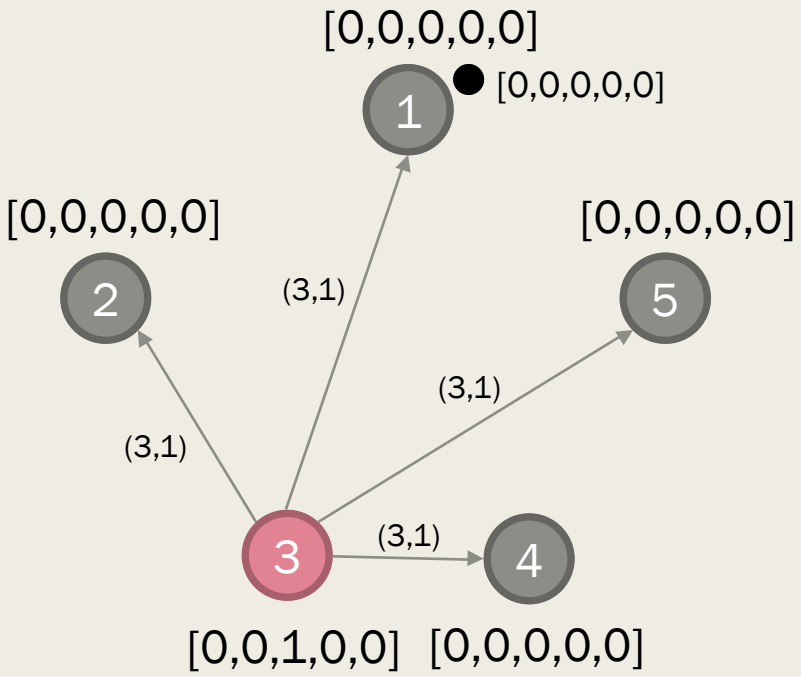
- Token obsahuje frontu procesů a pole čítačů LN[i], v tomto poli je uvedeno, kolikrát byl token kterému procesu **přidělen**
- Každý uzel *i* také udržuje pole čítačů Ri[i] pro každý proces, což značí, že na každém procesu je uloženo, kolikrát který proces o token **žádal**
- Uzel vyžadující vstup do kritické sekce, tedy uzel *i*, se snaží získat token. Zvýší svůj čítač ve svém poli a rozešle zprávu všem ostatním
- Každý proces si upraví čítač u daného procesu (po obdržení zprávy) na výšší číslo ze svého aktuálního čítače a čítače ve zprávě – pro vyřešení případných zpožděných zpráv.
- Proces, který má volný token, nebo ho právě uvolnil, zkontroluje svoje čítače, a všechny čítače, které mají o jedna větší číslo a nejsou ve frontě tokenu do této fronty umístí. Následně pošle token prvnímu procesu z fronty a ten z fronty odstraní

# Suzuki – Kasami vysílací algoritmus

V systému je pět plně propojených procesů. Proces 1 drží token.

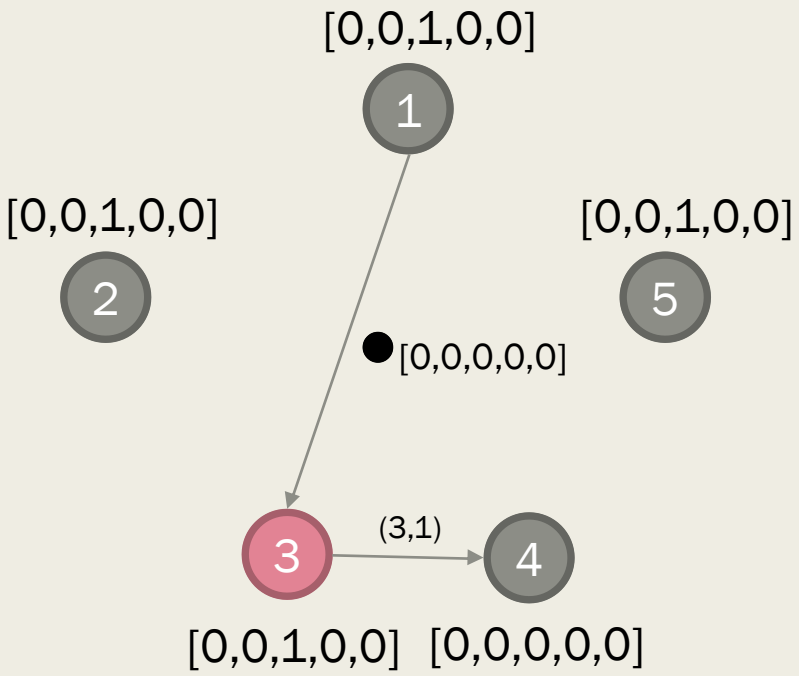


# Suzuki – Kasami vysílací algoritmus



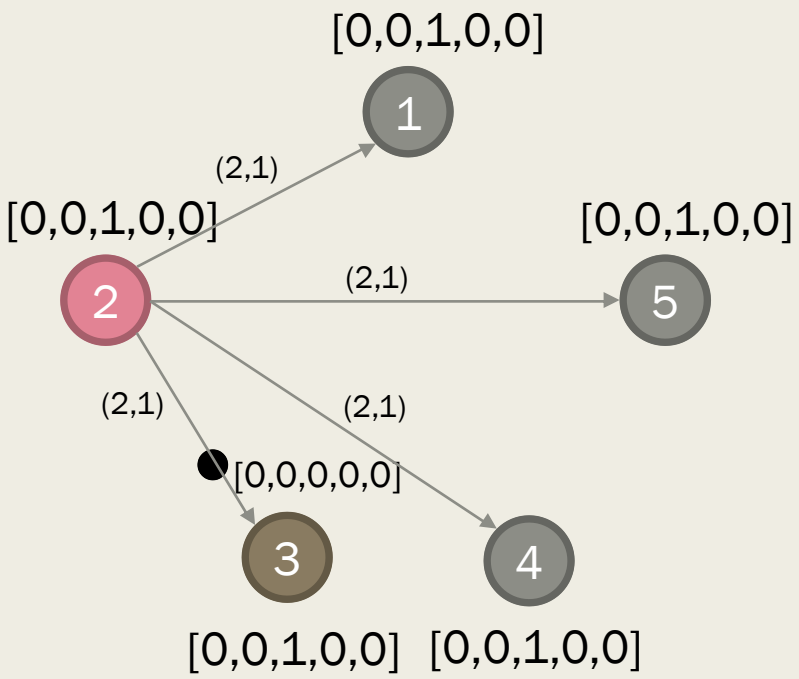

Proces 3 zažádá o token, protože chce vstoupit do kritické sekce

# Suzuki – Kasami vysílací algoritmus



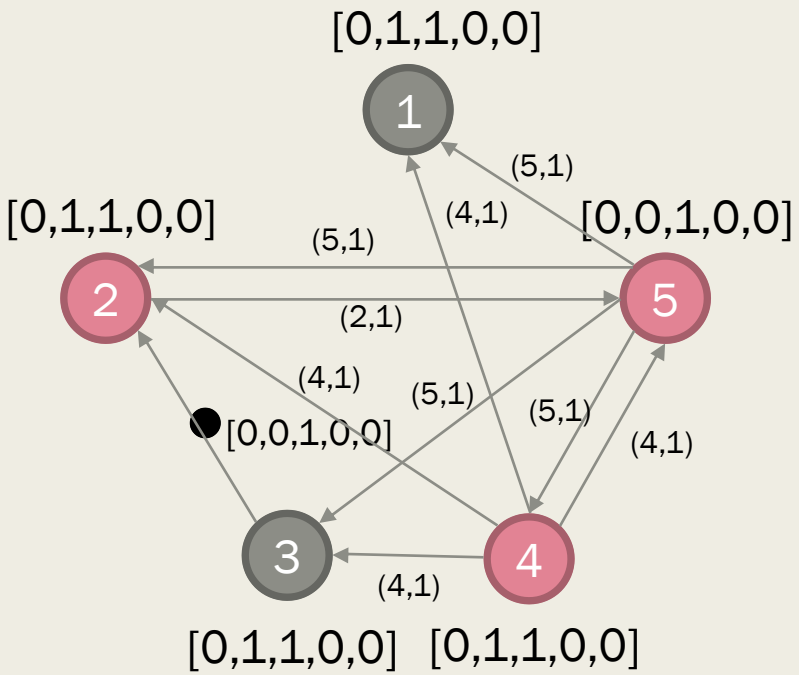

Procesy 1,2 a 5 zprávu zpracují. Proces jedna drží token, nepotřebuje jej a jelikož je třetí proces jediným, jehož hodnota v čítači je větší, než hodnota v čítači tokenu, zašle jej tomuto procesu

# Suzuki – Kasami vysílací algoritmus



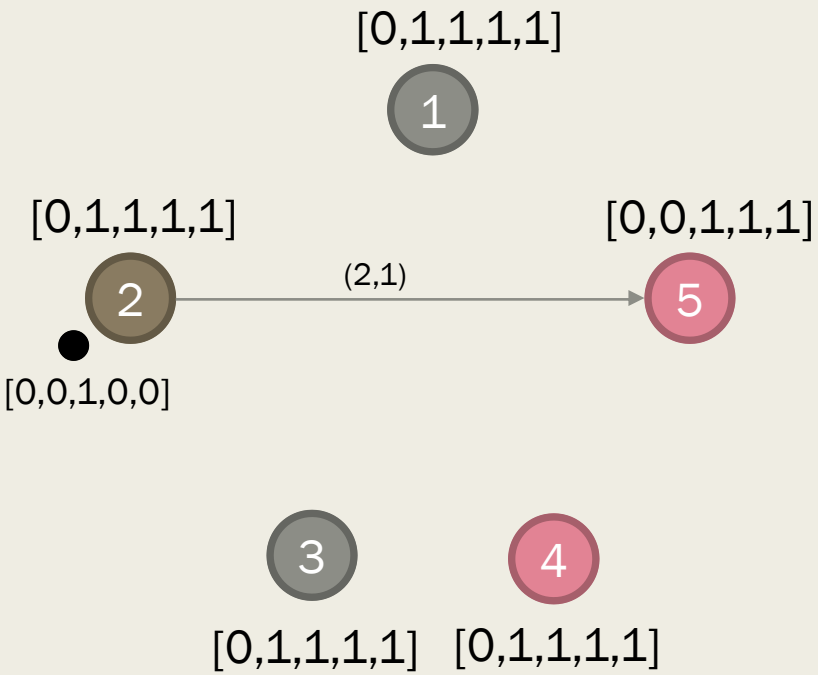

Proces 3 se nachází v kritické sekci. Proces 4 také zpracoval zprávu a upravil si patřičně svůj čítač. V ten samý okamžik žádá o token proces 2. Vyšle všem zprávu, že žádá poprvé a je odložen, čekajíc na token.

# Suzuki – Kasami vysílací algoritmus



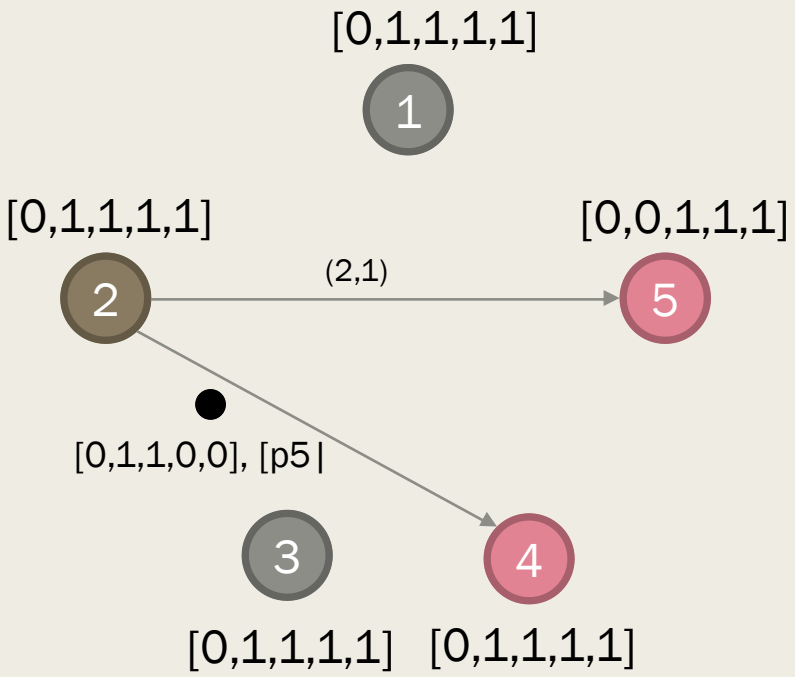

Všechny procesy až na pátý si zpracují žádost od druhého procesu, upraví své čítače a jelikož proces 3 dokončil svoji činnost v kritické sekci, posílá token jedinému procesu, o kterém ví, že jeho počet žádostí je vyšší, než je v tokenu uložený jeho počet přidělení tomuto procesu. Zároveň ale žádají procesy 4 a 5 o token. Procesu 5 stále nedorazila první žádost od procesu 2.

# Suzuki – Kasami vysílací algoritmus



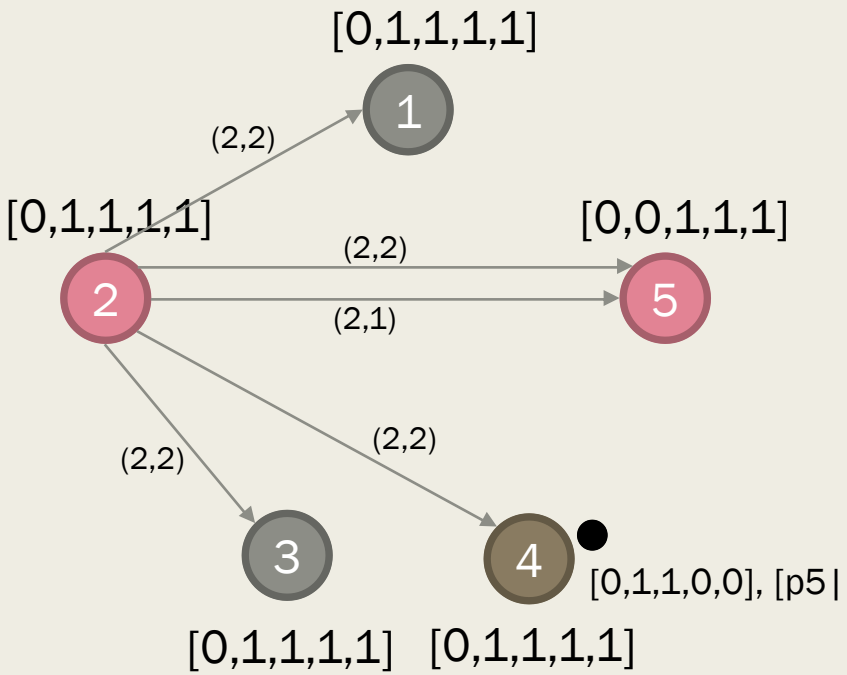

Proces 2 si užívá kritické sekce, procesy 4 a 5 čekají na token a zpráva uzlu pět od uzlu nebyla stále doručena, přestože již žádající uzel token má!

# Suzuki – Kasami vysílací algoritmus



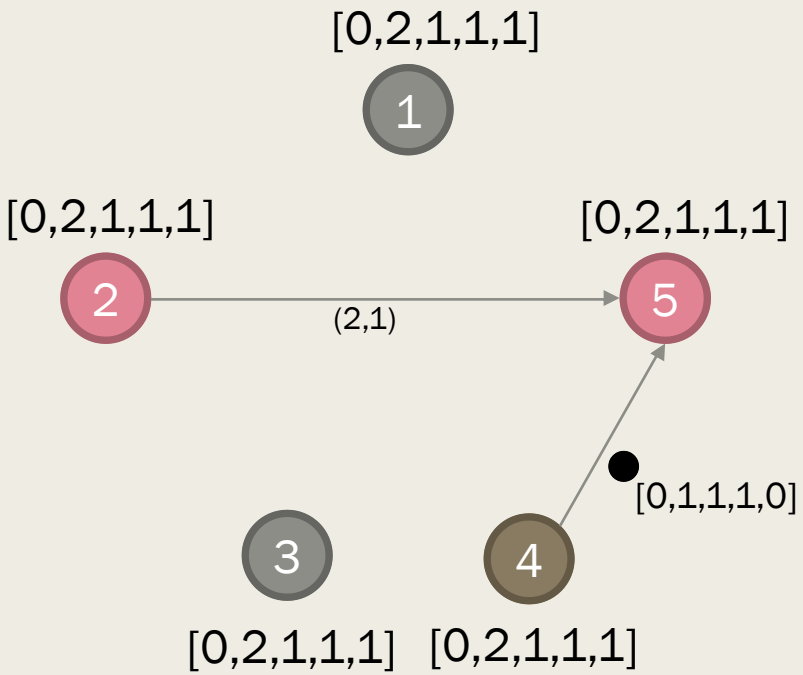

Stále máme jednu starou bloudící zprávu v systému. Proces dva dokončil svoji činnost v kritické sekci a zjistil, že hned dva uzly čekají na token, protože v čítači vidí, že mají vyšší počet žádostí, než je počet přidělení v záznamech tokenu. Vybere jeden, například čtvrtý, tomu pošle token a pátý proces umístí do frontu tokenu.

# Suzuki – Kasami vysílací algoritmus



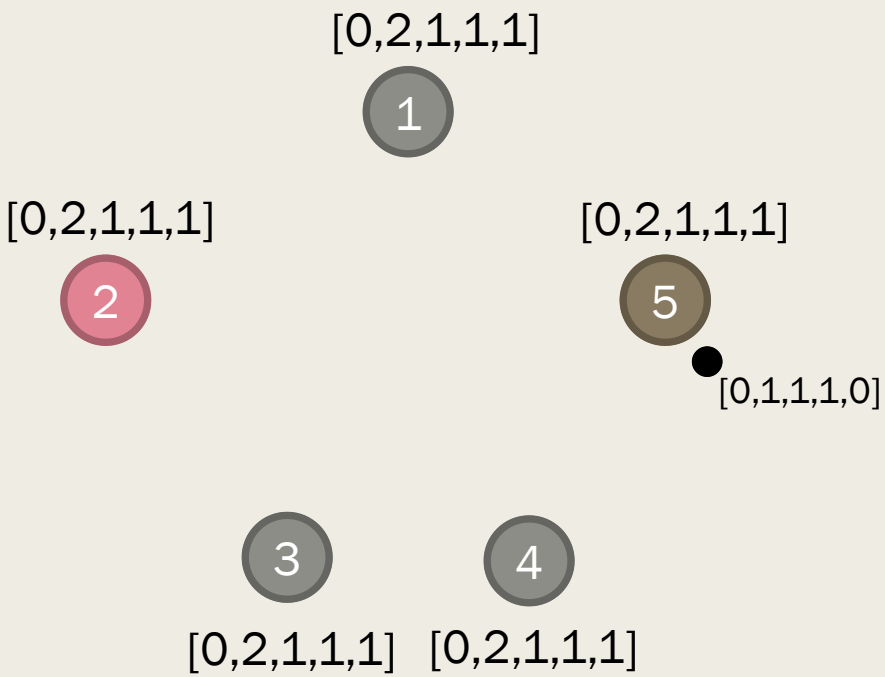

Proces 4 je v kritické sekci a proces 2 znovu žádá o vstup do této sekce. Pošle patřičné zprávy.

# Suzuki – Kasami vysílací algoritmus




Ty jsou zpracovány všemi procesy, i pátým, kterému nedošla první žádost od dvojky. Proces 4 opustil kritickou sekci a posílá dále token procesu uloženému na čele frontu procesů tokenu. Stejně by v tomto případě zjistil, že pětka čeká na token na základě čítače, ale díky frontě by věděla o žádosti pětky, i kdyby od ní nebyla zde žádost ještě zpracována.

# Suzuki – Kasami vysílací algoritmus




Proces 5 obdrží token a vstupuje do kritické sekce. Také konečně dorazila první zpráva od procesu 2. Ta ale neupraví čítač, protože ten je Již pro proces 2  u procesu 5 nastaven na aktuální hodnotu 2. Pokud by v době, kdy proces 2 žádal poprvé byl token na uzlu 5, pak by nebyl kvůli zdržení zprávy odeslán.

# Analýza

- Počet zpráv pro jeden vstup do kritické sekce $O(N)$
- Zaručeno ***vzájemné vyloučení*** a bez možnosti ***uváznutí*** a ***vyhladovění***
- Díky frontě požadavků garantována ***férovost***

# DETEKCE UKONČENÍ

# Detekce ukončení běhu v distribuovaném systému

- V systému se nachází *n* procesů, které komunikují předáváním zpráv
- Pokud neuvedeme jinak, předpokládáme plné propojení procesů komunikačními kanály
- Procesy komunikují s různou dobou zpoždění v přenášení zpráv.
- Detekce ukončení musí být schopna nalézt stav, ve kterém již není žádná zpráva přenášena nějakým kanálem a nikdy v budoucnu pro tento běh ani nebude (stabilní vlastnost).

# Atomický model komunikujících procesů

- Komunikace je zahájena jedním procesem
- Pokud proces obdrží zprávu, může odeslat atomicky zprávy ostatním procesům
- Pokud v systému není žádná zpráva, je běh systému ukončen

# Algoritmus čtyř čítačů

- Myšlenka je založena na kauzalitě odesílané a přijímané zprávy
- Pokud v nějakém čase ***t1*** obdrží pozorovatel od všech procesů počet přijatých zpráv a poté se dotáže na počet odeslaných zpráv => pokud se v odpovědi v čase ***t2*** tyto počty rovnají, musely v okamžiku ukončení odpovědi na první dotaz být obdrženy všechny odeslané zprávy
  - *Pokud by v čase **t1** nějaká dříve odeslaná zpráva ještě nebyla doručena, musí být zahrnuta v oznámených odeslaných zprávách v čase **t2***

# Algoritmus čtyř čítačů

Při přijetí zprávy

$recv_i = recv_i + 1$

$send_i = send_i + x$ (x je počet zpráv odeslaných od přijetí poslední zprávy doposud)

Při odeslání zprávy

Detekce ukončení

loop

dotaz všem procesům na počet přijetí a odeslání zpráv

po obdržení všech odpovědí

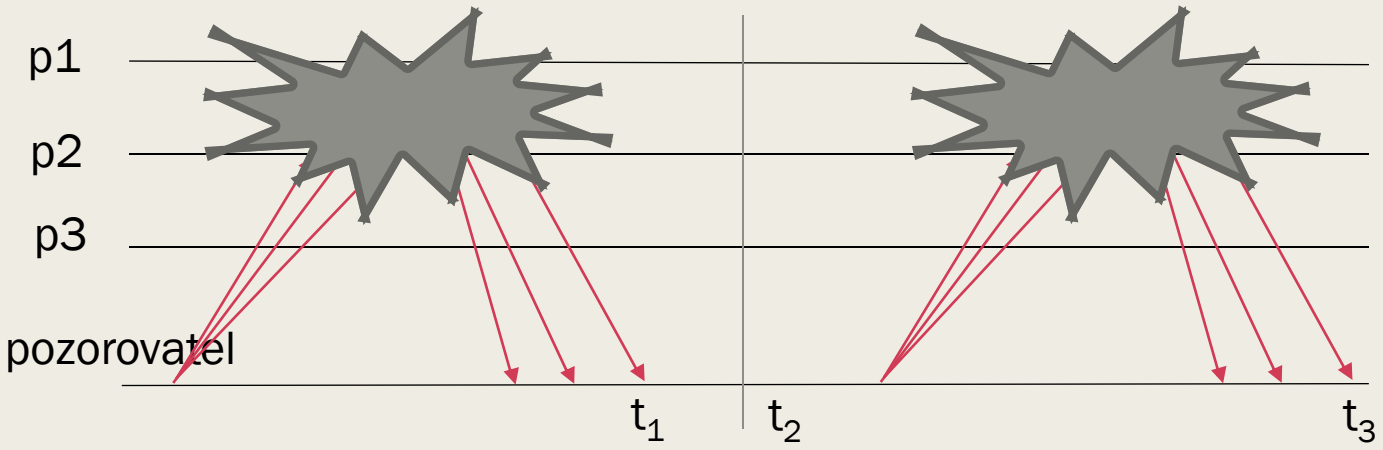
$\mathbf{R_2}$ = suma všech avizovaných přijatých zpráv

$\mathbf{S_2}$ = suma všech avizovaných odeslaných zpráv

pokud $\mathbf{R_1} = \mathbf{S_2}$, došlo v řezu k přijetí všech odeslaných zpráv (detekce ukončení), jinak

$\mathbf{R_1} = \mathbf{R_2}$, $\mathbf{S_2} = \mathbf{S_1}$

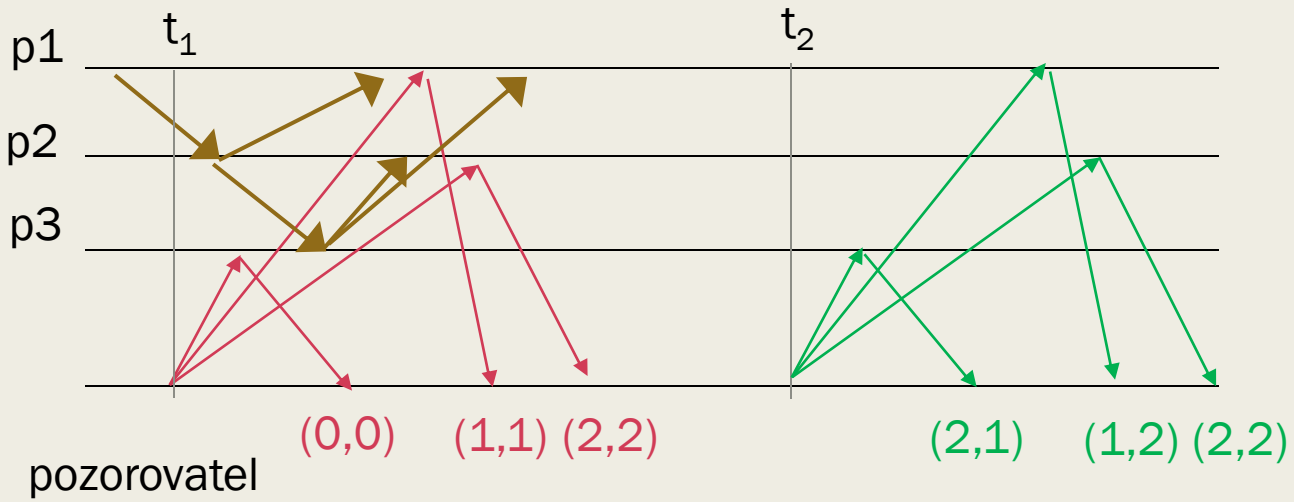
# Algoritmus čtyř čítačů



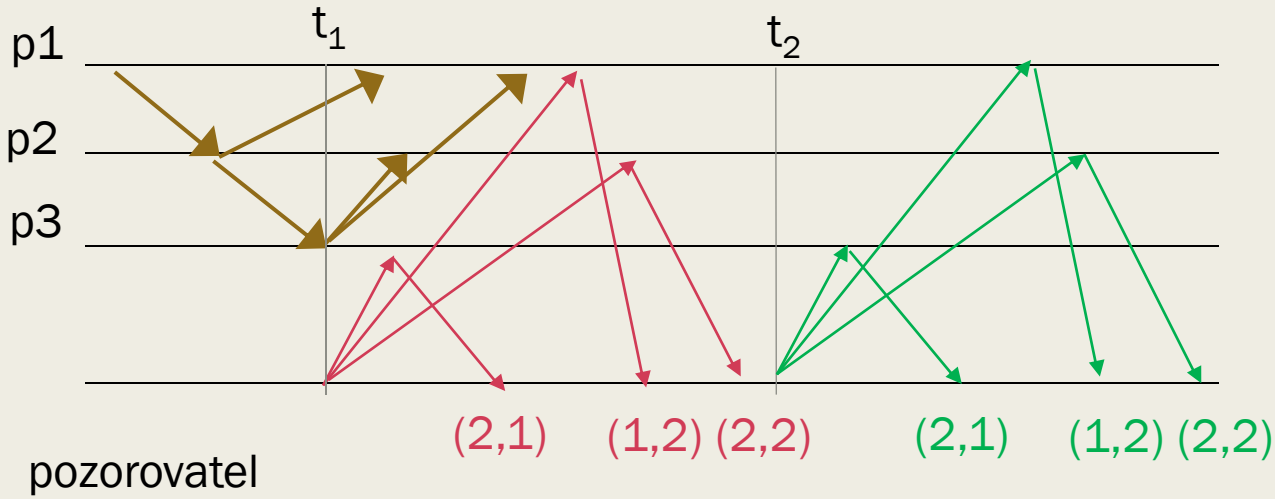

V čase $t_1$ pozorovatel zjistí (minimálně kolik) $S_1$ a $R_1$
V čase $t_3$ pozorovatel zjistí (minimálně kolik) $S_2$ a $R_2$
$S_2$ jistě zahrnuje zprávy odeslané před $t_2$
$R_1$ jistě nezahrnuje zprávy přijaté po $t_2$
Pokud $R_1 = S_2$, pak v čase $t_2$ byly všechny komunikační kanály prázdné (všech $R_1 = S_2$ zpráv bylo přijato před $t_2$ a muselo být i odesláno před ---$t_2$) → detekce ukončení před časem $t_2$ dle předpokladů

**R* >= R1 = S2 >= S*, tedy R* >= S*, ale podle kauzality R*=S* v čase $t_2$**
**R* a S* je skutečný počet přijatých a odeslaných zpráv před časem $t_2$**

# Algoritmus čtyř čítačů, příklad



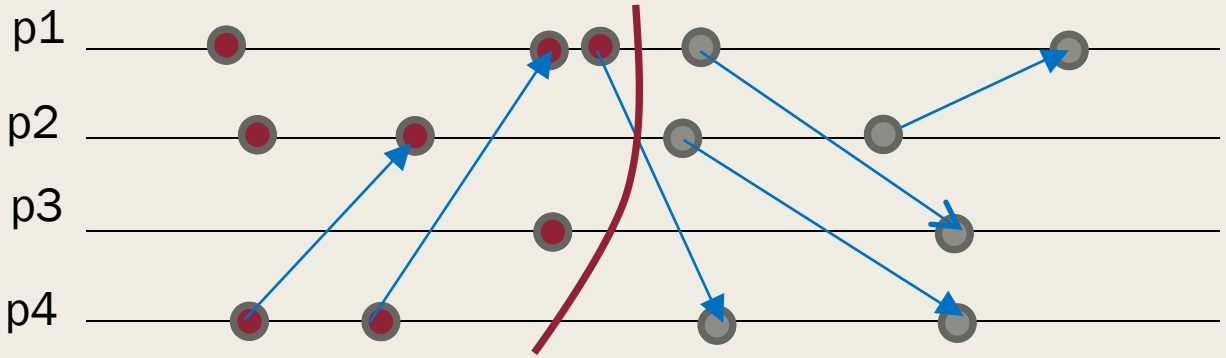

**R1=3**, S1=3
R2=5, **S2=5**




**R1=5**, S1=5
R2=5, **S2=5**

# Řez procesů (obecně)

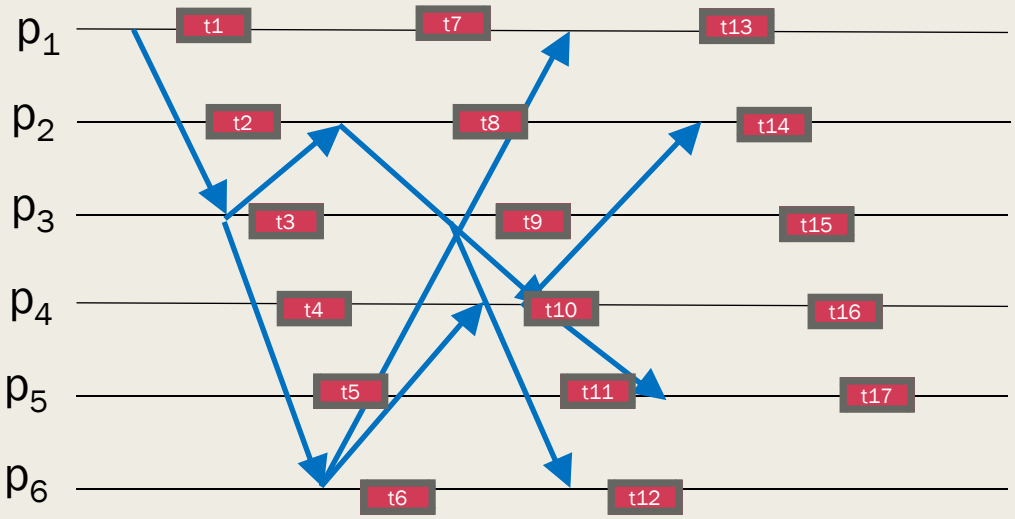
- Pro každou událost *e* v řezu C je každá událost *f*, která kauzálně předchází události *e* také v řezu C



# Detekce ukončení, vektor

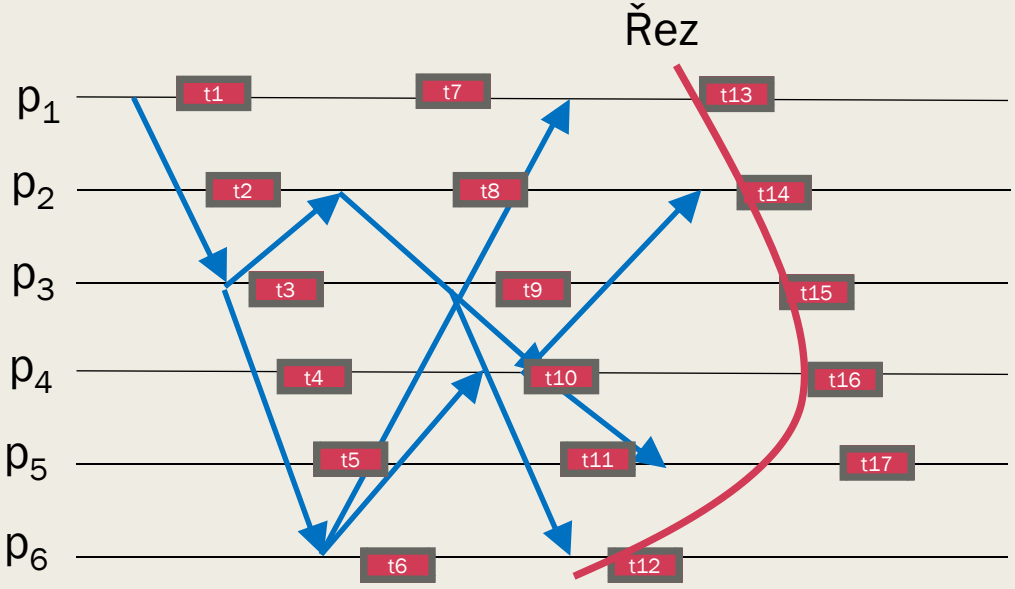
- Pro n procesů $p_1$, $p_2$ … $p_n$ máme token s vektorem [$m_1$,$m_2$ … $m_n$]
- Token se postupně předává po jednotlivých procesech
- Pokud process obdrží token, pak $r_j$ je počet zpráv, které obdržel od procesu *i* od posledního držení tokenu a $s_i$ počet zpráv, které od té doby procesu *i* odeslal
- Ve vektoru sníží hodnotu u procesu, od kterých přijal zprávy o $r_i$ a u sebe zvýší hodnotu o $s_i$

# Vektorový algoritmus, příklad




$t_1$=(0,0,1,0,0,0)
$t_2$=(0,0,1,0,0,0)
$t_3$=(0,1,0,0,0,1)
$t_4$=(0,1,0,0,0,1)
$t_5$=(0,1,0,0,0,1)
$t_6$=(1,1,0,1,0,0)
$t_7$=(1,1,0,1,0,0)
$t_8$=(1,0,0,1,1,0)
$t_9$=(1,0,0,1,1,1)
$t_{10}$=(1,1,0,0,1,1)
$t_{11}$=(1,1,0,0,1,1)
$t_{12}$=(1,1,0,0,1,0)
$t_{13}$=(0,1,0,0,1,0)
$t_{14}$=(0,0,0,0,1,0)
$t_{15}$=(0,0,0,0,1,0)
$t_{16}$=(0,0,0,0,1,0)
$t_{17}$=(0,0,0,0,0,0)

# Vektorový algoritmus, příklad




$t_1$=(0,0,1,0,0,0)
$t_2$=(0,0,1,0,0,0)
$t_3$=(0,1,0,0,0,1)
$t_4$=(0,1,0,0,0,1)
$t_5$=(0,1,0,0,0,1)
$t_6$=(1,1,0,1,0,0)
$t_7$=(1,1,0,1,0,0)
$t_8$=(1,0,0,1,1,0)
$t_9$=(1,0,0,1,1,1)
$t_{10}$=(1,1,0,0,1,1)
$t_{11}$=(1,1,0,0,1,1)
$t_{12}$=(1,1,0,0,1,0)
$t_{13}$=(0,1,0,0,1,0)
$t_{14}$=(0,0,0,0,1,0)
$t_{15}$=(0,0,0,0,1,0)
$t_{16}$=(0,0,0,0,1,0)
$t_{17}$=(0,0,0,0,0,0)

# Detekce ukončení, diseminační procesy

- Komunikace je zahájena jedním procesem
- Ostatní procesy začínají v klidovém stavu
- Pokud proces obdrží zprávu, může
  - *provádět (tichou činnost)*
  - *odesílat zprávy ostatním procesům*
  - *přejít do klidového stavu*
- Pokud v systému není žádná zpráva a všechny procesy jsou v klidovém stavu, je běh systému ukončen

# Použití kostry grafu komunikace, pseudokód

**Při přijetí zprávy receive($p_i$,$p_j$,m) procesem $p_i$ od procesu $p_j$**

```
stav_i =aktivni
if(predek_i=null) then
      predek_i=p_j
else
      send(p_i,p_j,ACK)
```

**Při přechodu do nečinnosti**

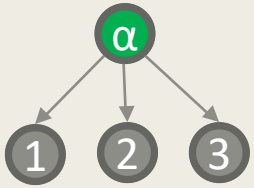
```
stav_i=pasivni
if(deficit=0)
      send(p_i,predek,ACK)
      predek=null
```

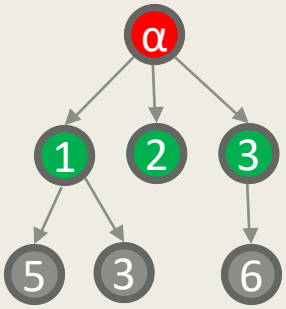
**Při odeslání zprávy send($p_i$,$p_j$,m)**

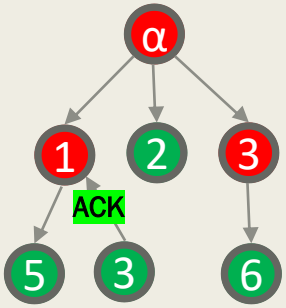
```
deficit++
```

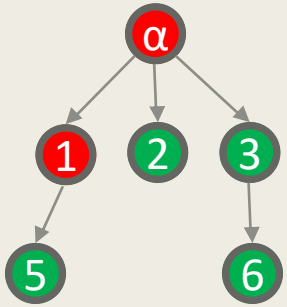
**Při přijetí zprávy receive($p_j$,$p_i$,ACK)**

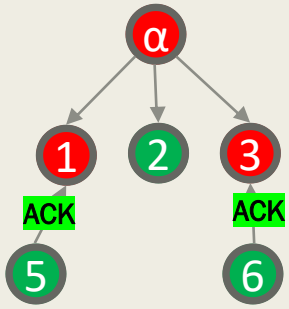
```
deficit--
if(deficit=0)&(stav=pasivni)
      send(p_i,predek,ACK)
      predek=null
```

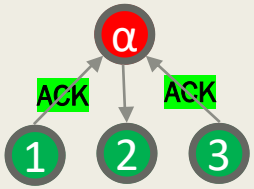

α
1
2
3

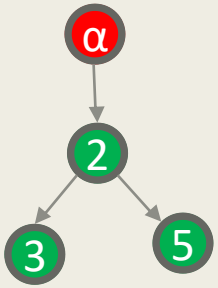

α
1
2
3
5
3
6

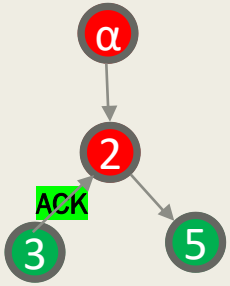

α
1
2
3
ACK
5
3
6

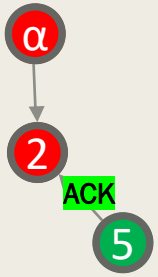

α
1
2
3
5
6


α
1
2
3
ACK
ACK
5
6


α
ACK
ACK
1
2
3


α
2
3
5


α
2
ACK
3
5


α
2
ACK
5


α
2


α
ACK
2


α


α

UKONČENÍ

# NEKOREKTNÍ PROCESY

# Asynchronní / Synchronní distribuovaný systém

**Asynchronní systém (Model reálného světa)**

- V systému neexistuje žádná pevná horní mez Δ pro doručení zprávy. Zpráva může dorazit za milisekundu, za hodinu …
- Standardní model pro internet, cloudy a většinu distribuovaných aplikací – síť je nepředvídatelná.
- Zprávy se mohou předbíhat (zpráva B odeslaná po zprávě A může dorazit dříve), mohou se ztratit nebo být duplikovány

Synchronní systém

- Definuje existenci horní meze Δ (latence). To znamená, že máme matematickou jistotu, že zpráva vždy dorazí do určitého času.
- Klíčový předpoklad pro řešitelnost konsenzu. V reálném světě (asynchronní systémy) tento předpoklad neplatí, proto jej simulujeme pomocí *Failure Detectorů*

Částečně synchronní systém

- Na počátku asynchronní systém se po čase se systém ustálí do synchronního stavu

# Abstrakce a Modely Selhání

**Abstrakce chování procesu (Co se s ním stane?)**

- **Crash & Stop** (Havárie a zastavení): Proces přestane fungovat a *nikdy* se neobnoví.
- **Crash & Recovery** (Havárie a obnova): Proces selže, ale po čase se *obnoví* a pokračuje. Musí mít přístup k perzistentní pamětii k obnově stavu.
- **Byzantine** (Byzantské selhání): Libovolné chování. Proces může odesílat špatná data, lhát nebo se chovat škodlivě. Nejtěžší model na řešení.

**Abstrakce observace systému (Jak selhání vidíme?)**

- **Fail-Silent**: Při selhání proces prostě zmlkne. Ostatní procesy to nemohou spolehlivě odlišit od pomalé sítě bez použití timeoutů (často vede k nutnosti *eventual consistency* modelů).
- **Fail-Stop** (vyžaduje *Perfect Failure Detector*): Selhání je okamžitě a spolehlivě detekováno všemi ostatními. Ostatní procesy *vědí*, že daný proces selhal a už se nevrátí.
- **Fail-Noisy**: Ostatní procesy jsou informovány o selhání (např. obdrží signál nebo vyprší spolehlivý timeout), ale detekce může mít zpoždění nebo být dočasně nepřesná (falešná pozitiva, je podezírán se selhání proces, který se jen opozdil).

# Detekce selhání, požadavky

- Pro model crash-stop abstrakci a synchronní distribuovaný systém máme perfektní detektor selhání
- Vlastnosti
  - Silná úplnost (Strong completeness): Eventuálně (dříve nebo později) je každý selhaný proces permantně detekován
  - Silná přesnost (Strong accuracy): pokud je proces detekován jako selhaný, tak selhal
- Pro model crash-stop a částečně synchronní distribuovaný systém máme eventuálně perfektní detektor selhání
  - Silná úplnost
  - Eventuálně silná přesnost (Eventual strong accuracy): Eventuálně žádný korektní proces není podezřívaný jako selhaný

# Detekce selhání, perfektní detektor selhání P

```
upon event <P,Init> do
        active:=Π;
        detected:=0;
        start_timer();

upon event (Timeout) do
        forall p ∈ Π
          if(p ∉ alive) ∧ (p ∉ detected) then
                detected= detected ∪ {p};
                trigger<P,Crash|p>;
          trigger<send|p,[HeartbeatRequest]>;
        alive:=0;
        start_timer();

upon event <deliver, q,[HeartbeatRequest]> do
        trigger <send | q,[HearbeatReply]>

upon event <deliver | p,[ HeartbeatReply]> do
         active:= active ∪ {p}
```

Proces neposlal HeartbeatReply a ani není mezi dříve detekovanými

Signalizujeme, že daný proces (asi) selhal (v systému může být obsluha této události)

Každému, i domněle selhanému procesu pošleme žádost o heartbeat

Začneme novou etapu, žádný proces není považován teď za živý

# Detekce selhání, nakonec perfektní detektor selhání

- Procesy, které se odmlčí, nakonec obnoví svoji činnost
- Po ukončení čekání jsou procesy považovány buď za živé, pokud odpověděly, nebo podezřelé – generuje událost $< \Diamond P, Suspect | p >$
- Pokud nějaký podezřelý proces se ozve později
  - *je odstraněn z množiny podezřelých*
  - *zvětší se interval čekání (pravděpodobně nebyl dostatečný)*
  - *generuje se událost* $< \Diamond P, Restore | p >$

# Detekce selhání, nakonec perfektní detektor selhání

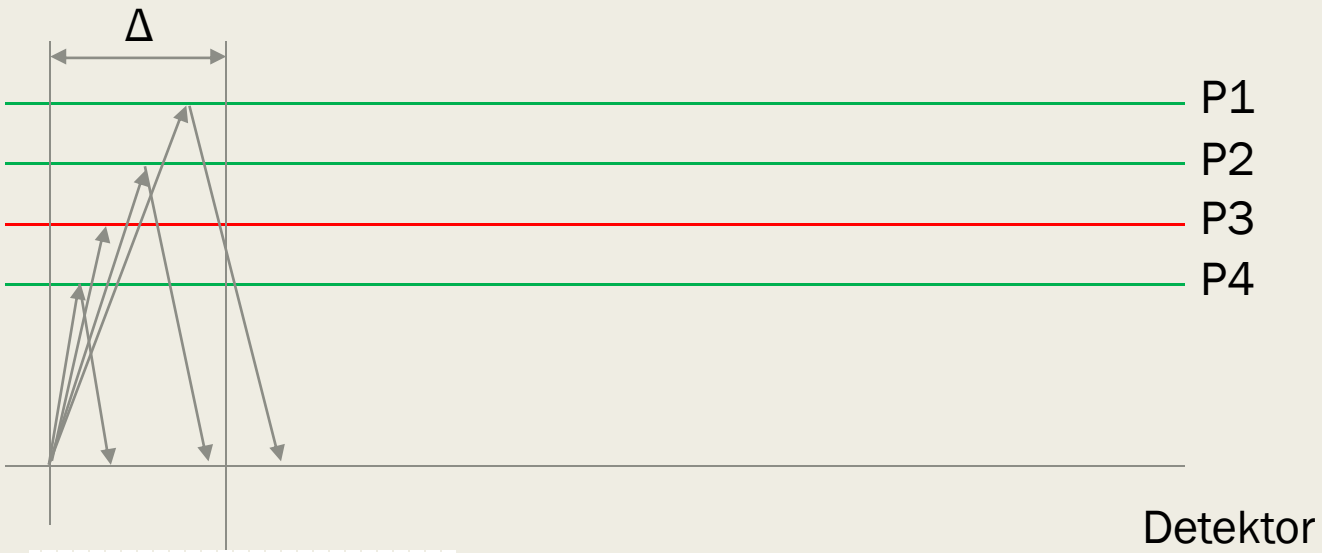
upon event <◊P,Init> do
    active:=Π; suspected:=0;
    delay:=Δ;
    start_timer(delay);

upon event (Timeout) do
    if($alive \cap suspected \neq 0$) *then*
        delay:=delay+ Δ;
    forall $p \in \Pi$
        if($p \notin alive$) ∧ ($p \notin suspected$) then
            suspected= suspected ∪ {p};
            trigger< ◊P, Suspect|p>;
        else if($p \in alve$) ∧ ($p \in suspected$) then
            suspected= suspected − {p};
            trigger< ◊P, Restore|p>;
        trigger<send|p,[HeartbeatRequest]>;
    alive:=0;
    start_timer(delay);

upon event <deliver, q,[ HeartbeatRequest]> do
    trigger <Send, q,[HearbearReply]>

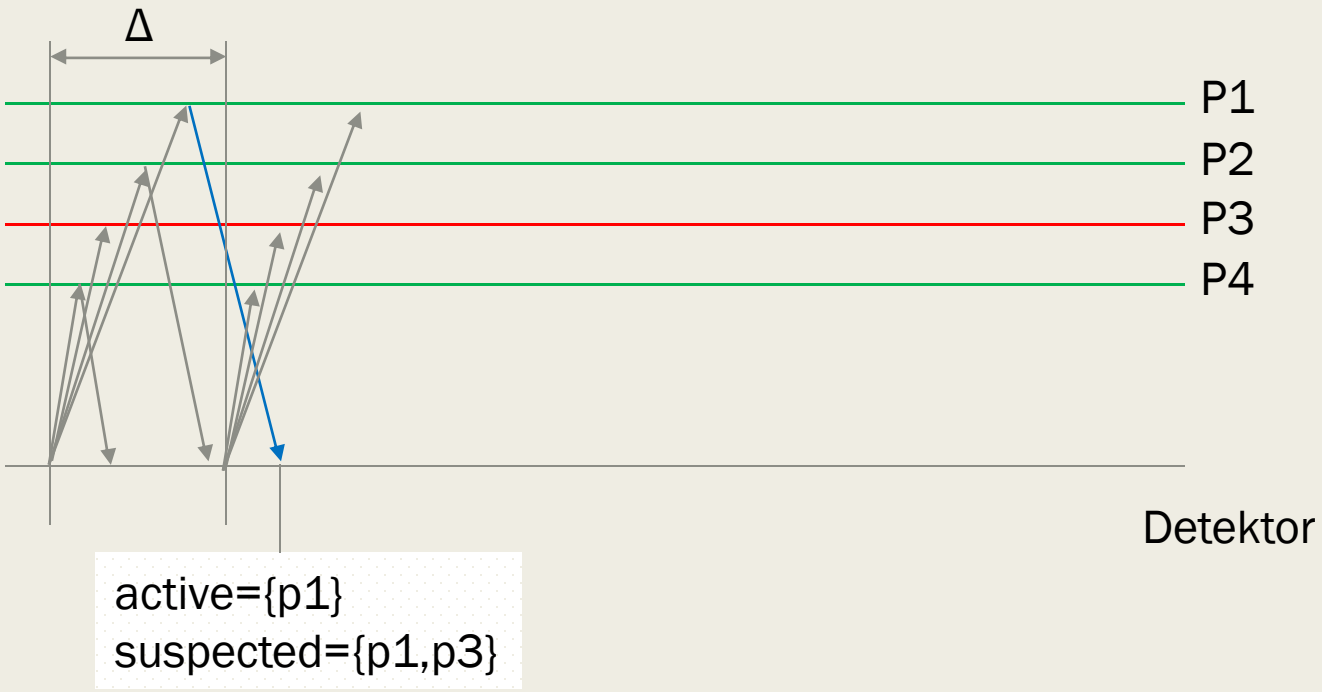
upon event <deliver, p,[ HeartbeatReply]> do
    $active$:= active ∪ {p}

# Detekce selhání, nakonec perfektní detektor selhání

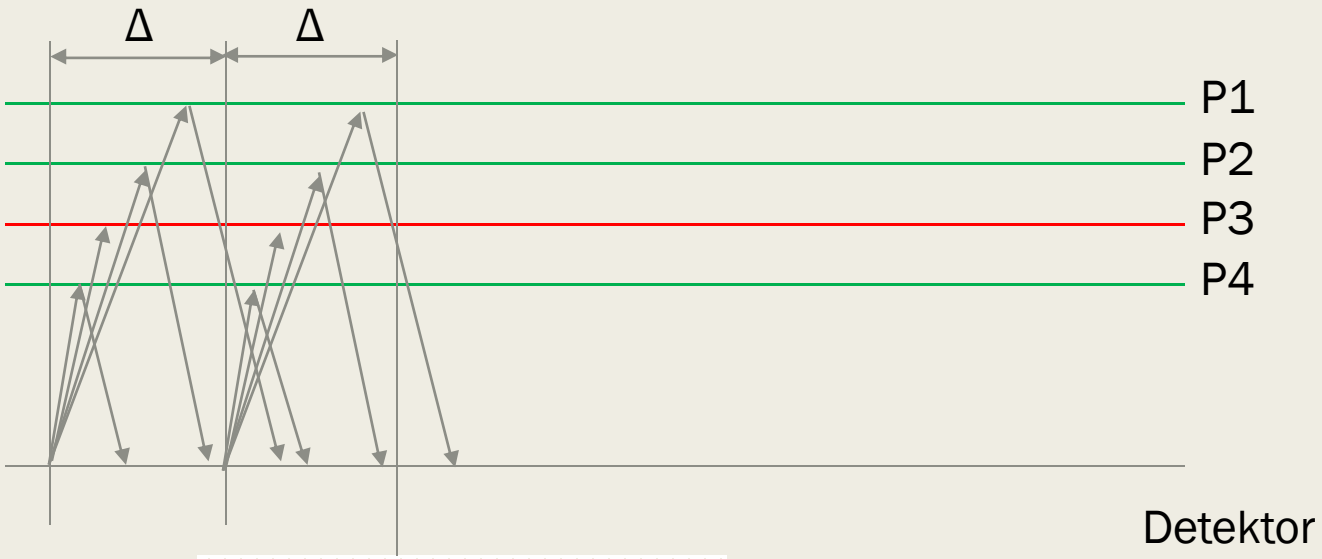



TIMEOUT
active={p2,p4}
suspected={p1,p3}

# Detekce selhání, nakonec perfektní detektor selhání

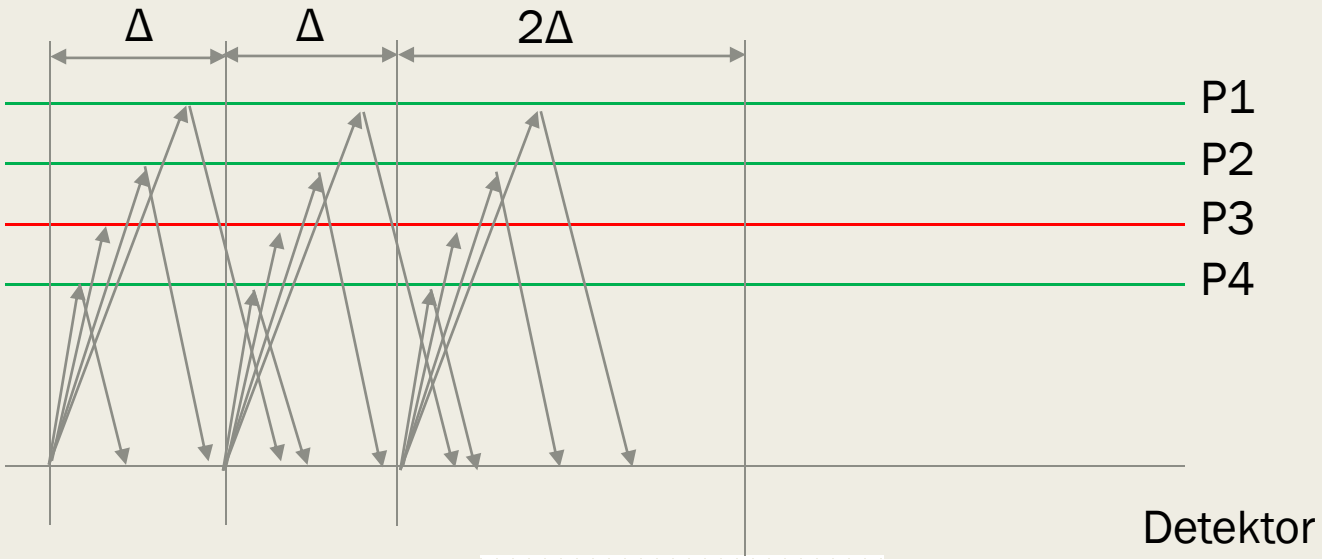


# Detekce selhání, eventuálně perfektní detektor selhání



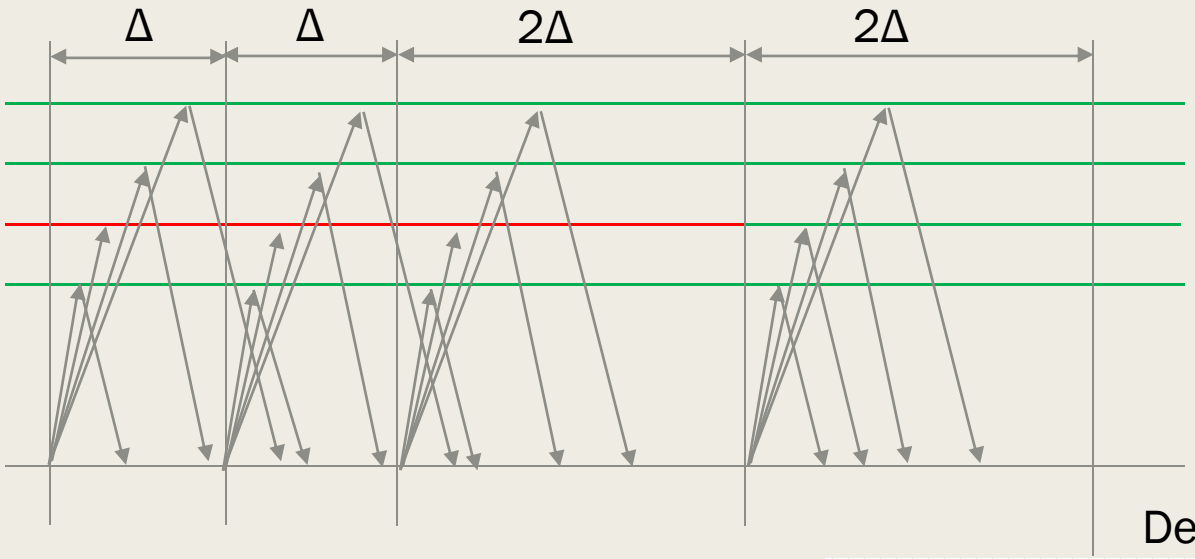

active={p2,p4,p1}
suspected={p1,p3}
$(alve \cap suspected \neq 0)$

# Detekce selhání, eventuálně perfektní detektor selhání




active={p2,p4,p1}
suspected={p3}

# Detekce selhání, eventuálně perfektní detektor selhání




active={p1,p2,p3,p4}
suspected={p3}
$(alve \cap suspected \neq 0)$
… příště 3Δ

# Volba lídra - v Monarchickém systému / Eventual Leader Detector Ω

- Lze použít i pro crash-recovery abstrakce procesů
- Udržuje množinu podezřelých procesů, které v daném čase neodpověděli a na základě této množiny volí lídra
- Vládne (a je mu důvěřováno) živý process (Monarcha), resp. process nepodezíraný z neživosti, s největší prioritou

```
upon event <Ω,Init> do
        suspected:=0
        leader:=0
upon event <◊P, Suspect|p> do
        suspected := suspected ∪ {p}
upon event <◊P, Restore|p > do
        suspected := suspected \ {p}
upon leader ≠ maxrank(Π \ suspected) do
        leader := maxrank(Π \ suspected)
        trigger< Ω , Trust | leader>;
```

# Nakonec perfektní detektor selhání Ω

- Předpokládá se, že existuje aspoň jeden proces, který nikdy neselže, nebo někdy selhává, ale po čase a zotavení již nikdy neselže
- Tento proces je vybrán algoritmem, který si počítá epochy, ve kterých byl aktivní
  - *Při inicializaci se epocha nastaví na 1 (a uloži se, aby byla dostupná při zotavení se procesu)*
  - *Při obnovení po pádu procesu se epocha inkrementuje o jedna*

```
upon event <Ω,Init> do
        epoch := 0; sotre(epoch); candidates :=0;
        trigger< Ω, Recovery>

// událost Recovery nastává i při zotavení
upon event <Ω,Recovery> do
        leader:=maxrank(Π);
        trigger< Ω, Trust | Leader>
        delay:= Δ;
        retrieve(epoch); epoch:=epoch+1; store(epoch);
        forall p ∈ Π trigger< Send | p,[HEARTBEAT, epoch] >;
        candidates:=0;
        start_timer(delay);
```

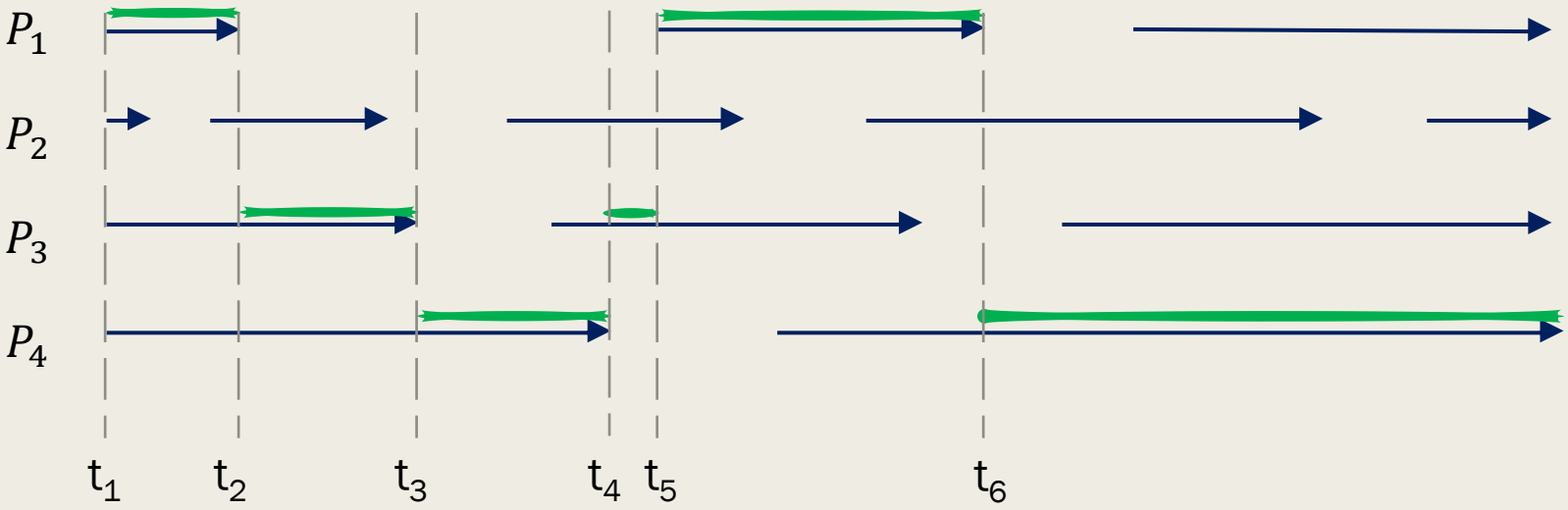
# Nakonec perfektní detektor selhání Ω

- V každém cyklu je procesem za lídra označen proces, který má odpověděl v nastaveném intervalu, a má nejmenší počet epoch (selhal nejméněkrát). Pokud je více takových, volí se podle ranku (provede **select**)
- Pokud se lídr změnil od minulého kola je nový lídr avizován událostí < Ω , Trust | leader> a je zvýšen delay o hodnotu Δ

```
upon event <Timeout>  do
        newleader:=select(candidates);
        if newleader≠leader then
                delay:=delay+ Δ;
                leader:=newleader;
                trigger< Ω, Trust | Leader>
        forall p ∈ Π trigger< Send | p,[HEARTBEAT, epoch] >;
        candidates:=0;
                start_timer(delay);

upon event <Deliver, q, [HEARTBEAT, ep] >  do
        if (s,e) ∈ candidates & s=q & e<ep then
                candidates:=(candidates – (q,e)) U (q,ep)
```

# Volba lídra pro částečně synchronní distribuovaný systém, příklad




($t_1$,$t_2$): candidates= ((P$_1$,0), (P$_2$,0), (P$_3$,0), (P$_4$.0)) / ((P$_1$,0), (P$_3$,0), (P$_4$.0)) /((P$_1$,0), (P$_2$,1), (P$_3$,0), (P$_4$.0))
($t_2$,$t_3$): candidates= ((P$_2$,1), (P$_3$,0), (P$_4$.0)) / ((P$_3$,0), (P$_4$.0))
($t_3$,$t_4$): candidates= ((P$_4$.0)) / ((P$_2$,2), (P$_4$.0)) / ((P$_2$,2), (P$_3$,1), (P$_4$.0))
($t_4$,$t_5$): candidates= ((P$_2$,2), (P$_3$,1))
($t_5$,$t_6$): candidates= ((P$_1$,1), (P$_2$,2), (P$_3$,1)) / ((P$_1$,1), (P$_3$,1)) / ((P$_1$,1), (P$_3$,1), (P$_4$,1)) / ((P$_1$,1), (P$_2$,3), (P$_3$,1), (P$_4$,1)) / ((P$_1$,1), (P$_2$,3), (P$_4$,1))
($t_6$,$t_+$): candidates= ((P$_2$,3), (P$_4$,1)) / ((P$_2$,3), (P$_3$,2), (P$_4$,1)) / ((P$_1$,2), (P$_2$,3), (P$_3$,2), (P$_4$,1)) / ((P$_1$,2), (P$_3$,2), (P$_4$,1)) / ((P$_1$,2), (P$_2$,4), (P$_3$,2), (P$_4$,1))

# Volba lídra pro asynchronní distribuovaný systém, vlastnosti

- Vlastnosti
  - ***Nakonec přesnost*** *(Eventual Accuracy): Po nějakém čase každý proces věří nějakému korektnímu procesu jako lídrovi*
  - ***Nakonec shoda*** *(Eventual Agreement): Po nějakém čase žádné dva procesy nevěří různým procesům jako lídrům*

Systém nakonec konverguje k jedinému, permanentně korektnímu lídrovi (bez ohledu na počáteční stav nebo přechodné chyby)

**Abstrakce zotavení:** Algoritmus umožňuje procesům po restartu (po pádu) integrovat se zpět do rozhodování, aniž by byla narušena integrita lídra

# Distribuovaný konsensus

- Procesy navrhují volbu / hodnotu a kolektivně se shodnou na nějakém návrhu
- Operace pro konsensus
  - **Návrh** (Proposal), *každý proces může vznést návrh a ten rozešle všem ostatním procesům*
  - **Rozhodnutí** (Decision), *proces se rozhodl pro návrh na základě všeobecné shody – všechny procesy se musí shodnout na stejném návrhu*
- Vyžaduje synchronní systém, nebo částečně synchronní systém
  - *Omezené zpoždění zpráv, je garantované zpoždění do nějakého limitu* $\Delta$
  - *Omezený čas zpracování události procesem*
  - *Algoritmy pracují v kolech*

# Distribuovaný konsensus

- Základní požadavky
  - *Ukončení (Termination) Každý korektní proces se nakonec pro nějakou hodnotu rozhodne.*
  - *Platnost (Validity) Pokud se proces rozhodne pro hodnotu ‚V', pak ‚V' musela být navržena některým z procesů.*
  - *Integrita (Integrity) Žádný proces se nerozhodne více než jednou.*
  - *Shoda (Agreement) Žádné dva korektní procesy se nerozhodnou pro odlišné hodnoty.*
  - *Uniformní shoda (Uniform Agreement): Žádné dva procesy se nerozhodnou pro odlišné hodnoty*
- Poslední vlastnost odlišuje regulérní konsensus od uniformního konsensu – v druhém případě je vyžadována, v prvním ne

# Distribuovaný konsensus

- Pro asynchronní systém neexistuje algoritmus pro zaručení konsensu
- FLP Věta (Ficher, Lynche, Paterson, 1985)

  „V **čistě asynchronním systému** nelze dosáhnout konsensu (shody) deterministickým algoritmem, pokud může dojít k selhání byť jen **jednoho jediného procesu**.“

  Protože nedokážeme rozlišit mezi tím, jestli proces selhal, nebo je jen extrémně pomalý
- Pokud máme algoritmus pro distribuovaný konsensus, dokážeme na něj převést ostatní úlohy (volba leadera, detekce chyb ...)

  Jedná se o nejtěžší problém v distribuovaných systémech

# Regulérní konsensus, záplavový algoritmus

- Algoritmus pracuje v kolech, ve kterých procesy sdílejí návrhy, které podaly, nebo o nich dostali zprávu od ostatních
- Předpokládá **crash-stop** model selhání, selhání lze detekovat
- Používá **Best Effort** všesměrové vysílání a **Perfect Failure Detector** → jeho použití zaručí shodu i pro **Best Effort**, jednodušší, než použít **Reliable Broadcast**
  - Přenáší menší počet zpráv při jednom vysílání
  - Nepřeposílá zprávy
  - I neodeslané zprávy představují režii
- Zaručuje **platnost**, **shodu**, **integiritu**, **ukončení**

# Regulérní konsensus, záplavový algoritmus

Princip:

- Procesy si uchovávají

  seznam procesů, od kterých obdrželi zprávu v přechozím kole

  *seznam procesů, od kterých obdrželi zprávu v přechozím kole*

  *seznam návrhů, i kterých se doslechly v aktuálním kole*

- Proces v každém kole vysílá zprávu se všemi mu známými návrhy
- Kolo skončí pro proces v okamžiku, když obdrží zprávu od všech ostatních korektních procesů – nekorektní co selhaly jsou detekovány
- Po skončení kola proces rozhodne pro nejmenší (největší) z jemu známých návrhů, pokud obdržel informace o návrzích od stejného počřu procesů jako v minulém kole

# Regulérní konsensus, záplavový algoritmus

Algoritmus si udržuje seznam procesů, které selhaly, dále číslo kola, zvolený návrh, pokud učinil rozhodnutí, seznam procesů, od kterých v daném kole obdržel návrhy a všechny návrhy, které v daném kole zaznamenal

```
upon event <c, init> do          // c - regulerní záplavový konsensus
   correct:=Π; round:=1 decision:=nill;
   recievedFrom:=[]; proposals:=[]
```

Perfektní detektor v synchronním systému dokáže detekovat, že proces p nepracuje správně a pokud byl učiněn návrh tímto procesem, vysílá tento návrh společně se všemi mu doposud známými návrhy dál (možně jen v prvním kole

```
upon event <crash,p> do              // detekováno selhání procesu p
   correct:=correct-{p};

upon event <propose,c, v> do
   proposals[1]:=proposals U {v}
    trigger<beb, broadcast | [Proposal, 1, proposals[1]]);
```

Pokud je mu doručen seznam návrhů od procesu proces, obojí si pro aktuální kolo zaznamená

```
upon event <deliver(process, [Proposal, round, ps])> do
     receivedfrom[round]:=receivedfrom[round] U {process}
     proposals[round]:=proposals[round] U ps
```

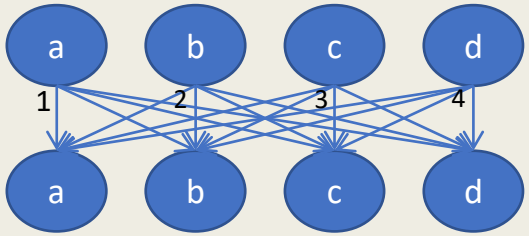
# Regulérní konsensus, záplavový algoritmus

- Pokud proces obdrží v kole *k* odpovědi od stejné množiny procesů, jako v kole předchozím, má informace o všech navržených ‚aktivních' hodnotách
  - *Žádný proces, který postoupil do kola k neví o jiném návrhu*
- Pokud proces nebyl doposud schopen sám rozhodnout na základě shod návrhů z aktuálního a předchozího kola, ale jiný korektní proces již rozhodl, následuje takové rozhodnutí

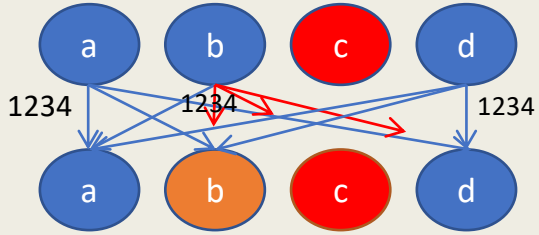
```
when correct ⊆ receivedfrom[round] ∧ decision = nill
   if receivedfrom[round]=receivedfrom[round − 1]
      decision := min(proposals[round]);
       trigger<beb, broadcast([DECIDED,decision])
        trigger<c, decide(decision)>
   else round := round +1;
      trigger<beb,broadcast([PROPOSAL,round,proposals
                                        [round − 1]])
upon event deliver(p, [DECIDED,v]) do
   if p ∈ correct ∧ decision = nill then
      decision := v
      trigger<beb, broadcast([DECIDED,decision])>
      trigger<c, decide(decision)>
```

Předpokládáme perfektní detektor selhání, nemůže být označen za korektní proces, který u tohoto broadcastu selal

Není zaručena atomicita! Decide může nastat i když broadcast selže

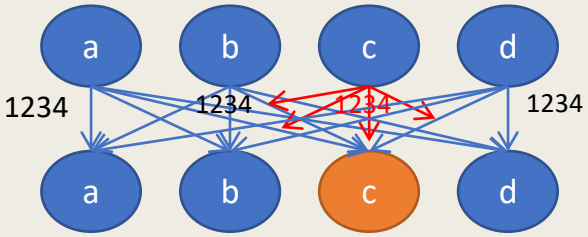



| | | | | |
|---|---|---|---|---|
| RecFrm[0] | [] | [] | [] | [] |
| RecFrm[1]: | [abcd] | [abcd] | [abcd] | [abcd] |
| Correct: | [abcd] | [abcd] | [abcd] | [abcd] |
| proposals: | [1234] | [1234] | [1234] | [1234] |

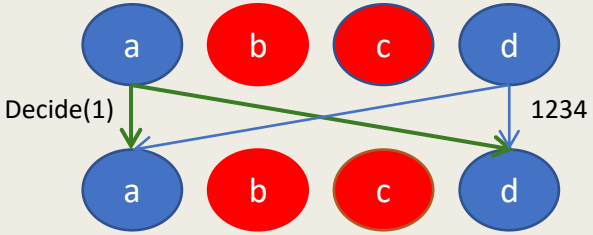



| | | | | |
|---|---|---|---|---|
| RecFrm[2] | [abd] | [abd] | - | [abd] |
| RecFrm[3]: | [abd] | - | - | [ad] |
| Correct: | [abd] | - | - | [ad] |
| proposals: | [1234] | - | - | [1234] |

**a** doručí od **b** dříve než detekuje jeho selhání -> učini rozhodnutí
(abc byly v minulém kole korektní a všichni znají jejich návrhy [1234])




| | | | | |
|---|---|---|---|---|
| RecFrm[1] | [abcd] | [abcd] | [abcd] | [abcd] |
| RecFrm[2]: | [abd] | [abd] | - | [abd] |
| Correct: | [abd] | [abd] | - | [abd] |
| proposals: | [1234] | [1234] | - | [1234] |




| | | | | |
|---|---|---|---|---|
| Decide | 1 | - | - | 1 |

**d** obdrží informaci, že **a** rozhodl a rozhodnutí následuje

# Analýza

- Algoritmus skončí po *f*+1 kolech kde *f* je počet selhání
- Počet přenesených zpráv každé kolo $O(N^2)$, stejný je i počet Decided zpráv
- Počet kol nejvýše $N \rightarrow$ celkem zpráv $O(N^3)$
- Je zaručena integrita, platnost (zřejmé z algoritmu), shoda, ukončení

# Analýza, korektnost

- **Shoda** (Agreement): Všechny korektní procesy rozhodnou stejně.

  V jednom kole nemohou dva procesy rozhodnout odlišně

  Pokud rozhodují po obdržení stejného počtu zpráv, rozhodují stejně

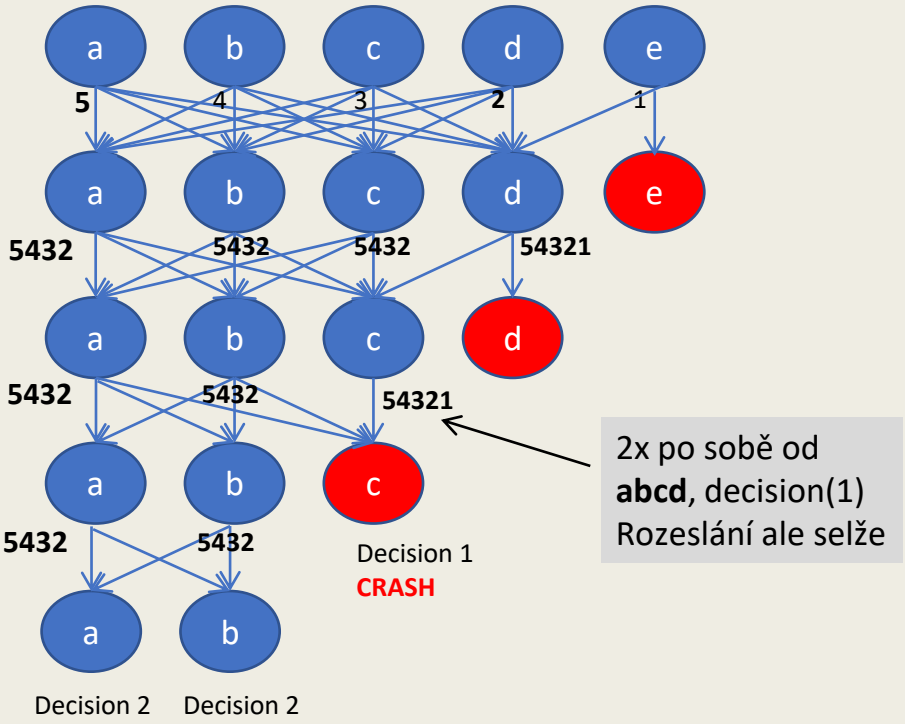
  Pokud obdrží Decision, buď jej obdrželi všechny korektní, nebo některé korektní obdrží méně zpráv a nemohou rozhodnout

  Jak rozhodl první korektní, rozhodnou všechny korektní v následujícím kole

- **Ukončení** (Termination): Nové kolo pokračuje, když alespoň jeden proces selže, nejpozději po *f* kolech nemůže již žádný selhat

# Analýza, uniformní shoda ?

- Není zaručena uniformní shoda
- Pokud proces udělá rozhodnutí a nešíří jej (selže před odesláním), toto rozhodnutí je ‚zapomenuto‘
- Ostatní procesy mohou udělat rozhodnutí jiné



# Záplavový algoritmus, uniformní shoda

- Procesy rozhodnou až v **N-tém** kole
- Udržují procesy, od kterých slyšeli návrh jen pro aktuální kolo
- Udržují globální, napříč koly, seznam doslechnutých návrhů

→ V **N-tém** kole mají všechny korektní (přeživši) procesy stejnou množinu doslechnutých návrhů

(pokud by proces věděl o návrhu o kterých jiný proces neví, musely by jej šířit v každém kole nekorektní procesy → těch je ale méně než N)

→ Rozhodnou jen oni a rozhodnou stejně (**uniformní shoda**)

# Hierarchický konsensus

- Algoritmus s $O(N^2)$ zprávami
- Používá **Best Effort** všesměrové vysílání a **Perfect Failure Detector** pro **crash-stop** model procesů
- Zaručuje **platnost**, **shodu**, **integritu**, **ukončení**
- V následující verzi nezaručuje uniformní shodu
- **Princip:**

  Pracuje v kolech, proces učiní rozhodnutí v kole, které odpovídá jeho ranku

  Rozhoduje se podle své aktuální hodnoty ***props*** *a vysílá ji všem ostatním procesům*

  Na počátku nastaví svůj návrh do své lokální proměnné ***props***

  Pokud proces obdrží hodnotu od procesu s vyšší prioritou, než má on sám, ale s nižší než je hodnota procesu, od kterého akceptoval ***propavlue***, uloží si ji a poznačí si i tento proces

# Hierarchický konsensus, algoritmus / implementace

- Algoritmus si udržuje seznam procesů, které selhaly, dále číslo kola, aktuálně zvolený návrh a jeho navrhovatele, příznak, zdali od procesu s daným rankem již návrh obdržel a příznak, zdali již návrh vysílal.
- Detekuje procesy které selhaly
- V případě podání počátečního návrhu, tento šíří jako návrh v rámci prvního kola

```
upon event <c,init> do   // c – hierarchický konsensus
     detected:={}; round:=1; proposal:=null; proposer:=0;
     delivered={}; broadcast:=false

upon event <crash|p> do // detekováno selhání procesu p
     detected:=detected U {rank(p)}

upon event <c,propose|v> do
     proposal:=v;
```

# Hierarchický konsensus, algoritmus

Proces rozhodne, pokud aktuální kolo odpovídá jeho ranku

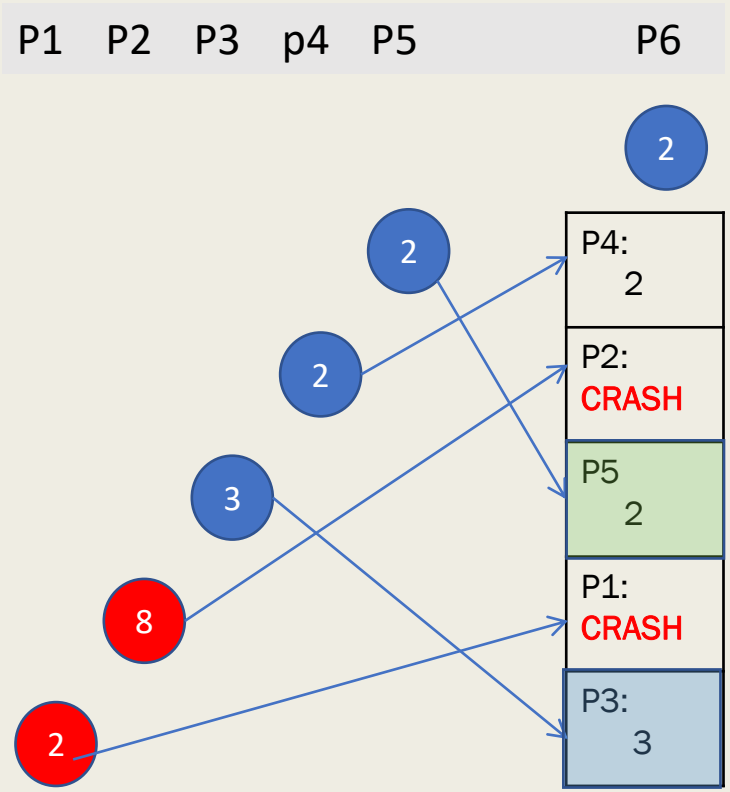
```
upon round=rank(self) ∧ proposal≠nill ∧ broadcast=false do
       broadcast=true
       trigger<beb, Broadcast|[DECIDED, proposal]>
       trigger<c,Decide|proposal>
```

Index kola každý proces zvýší, pokud pro od procesu s rankem round obdrží zprávu o jeho rozhodnutí, nebo detekuje jeho selhání

```
upon round ∈ detected ∨ delivered[round]=true do
       round=round+1
```

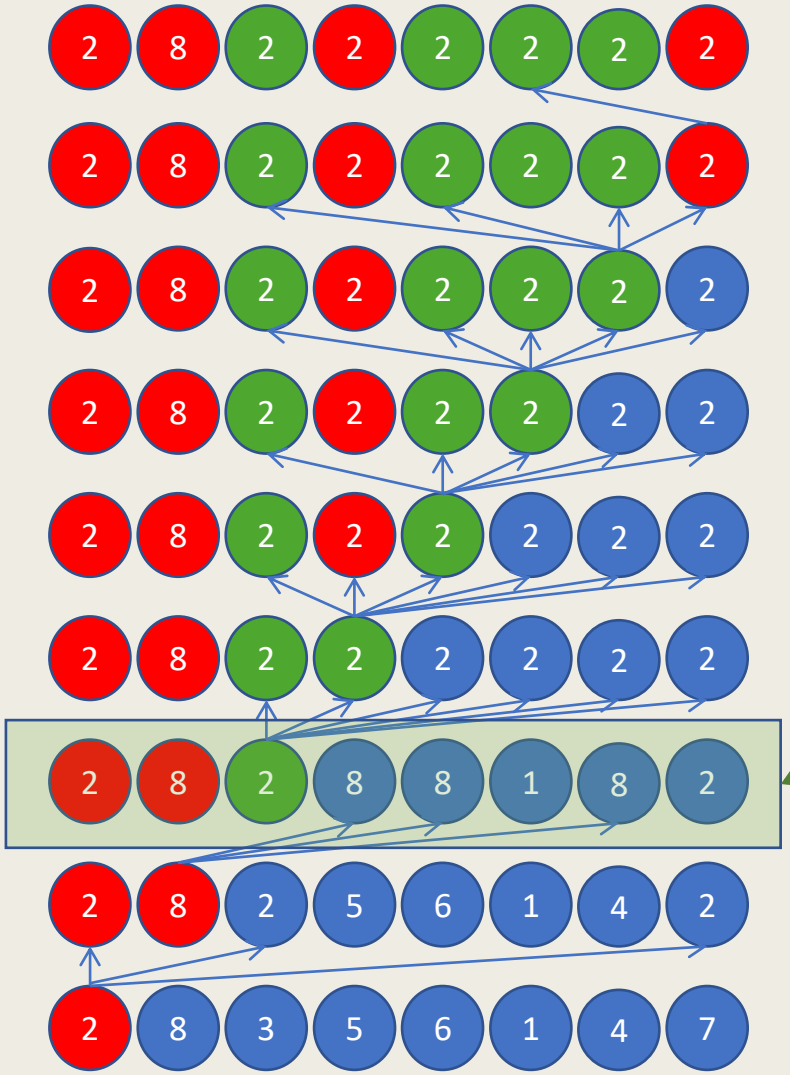
Po obdržení zprávy od procesu *p* si zprávu uloží jako aktuální kadidátní rozhodnutí, včetně informace o doručení od daného procesu, pokud rank odesilatele je menší než jeho, ale větší než rank procesu aktuálně uloženého kadndidátního rozhodnutí

```
upon event <beb, Deliver| p,[DECIDED,v])>
       if rank(p)<rank(self) ∧ rank(p)>proposer then
              proposal=v, proposer=p
       delivered(rank(p)):=true
```

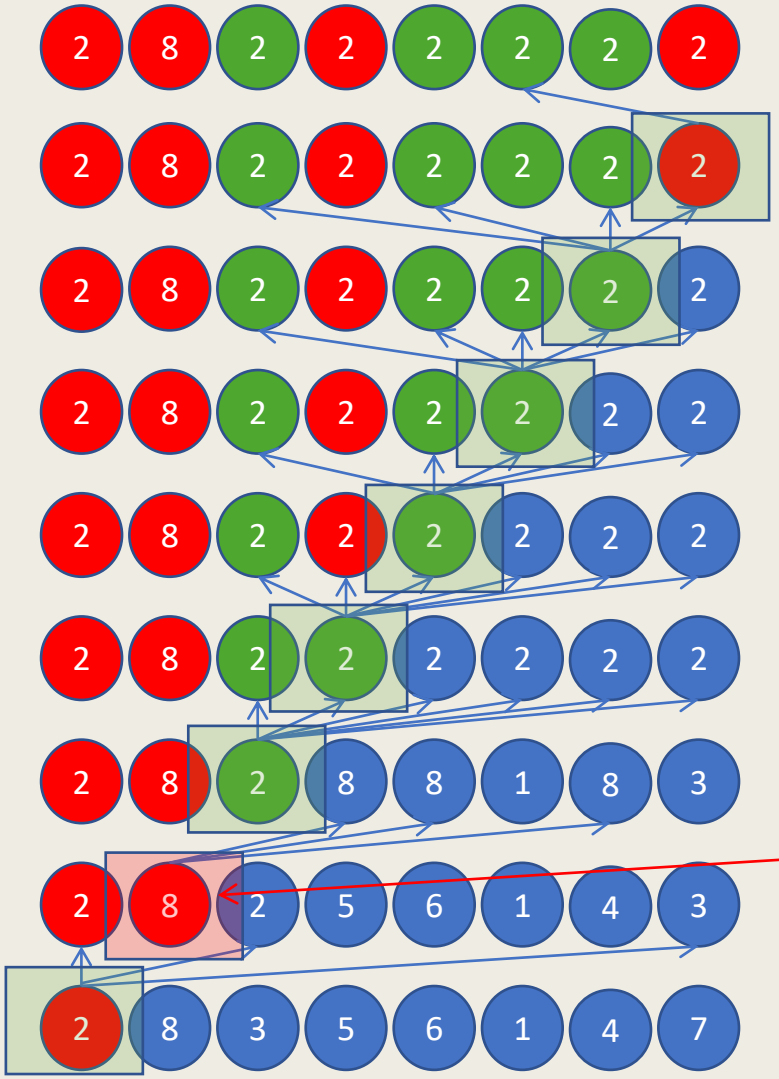



Adaptuje nejprve hodnotu 3 od procesu P3, pak hodnotu 2 od procesu 5. Hodnou 2 od procesuP4 ignoruje

- Žádný proces nerozhodne a nevysílá dříve, než obdrží zprávu nebo detekuje selhání všech procesů s nižším rankem.
- **Nezáleží na pořadí, v jakém proces obdrží v rámci jednoho kola zprávy od procesů s nižším rankem nebo detekuje jejich selhání → vždy rozhodně stejně**
- Všechny korektní procesy rozhodnou s hodnotou, kterou rozhodl korektní proces s nějnižším rankem
- Není zaručena uniformní shoda (druhý proces rozhodl pro hodnotu 8)




- Žádný proces nerozhodne a nevysílá dříve, než obdrží zprávu nebo detekuje selhání všech procesů s nižším rankem.
- Nezáleží na pořadí, v jakém proces obdrží v rámci jednoho kola zprávy od procesů s nižším rankem nebo detekuje jejich selhání → vždy rozhodně stejně
- **Všechny korektní procesy rozhodnou s hodnotou, kterou rozhodl korektní proces s nejnižším rankem**
- Není zaručena uniformní shoda (druhý proces rozhodl pro hodnotu 8)




- Žádný proces nerozhodne a nevysílá dříve, než obdrží zprávu nebo detekuje selhání všech procesů s nižším rankem.
- Nezáleží na pořadí, v jakém proces obdrží v rámci jednoho kola zprávy od procesů s nižším rankem nebo detekuje jejich selhání → vždy rozhodně stejně
- Všechny korektní procesy rozhodnou s hodnotou, kterou rozhodl korektní proces s nějnižším rankem
- **Není zaručena uniformní shoda (druhý proces rozhodl pro hodnotu 8)**

# Hierarchický konsensus, Analýza

- Je zaručena **platnost, shoda** a **integrita**
- **Ukončení** vyplývá z vlastnosti silné úplnosti perfektního detektoru (odhalí selhání procesu)
- Není zaručena **uniformní shoda**
- Ukončení je garantováno po N kolech, v každém se přenese $\mathrm{O(N)}$ zpráv
- Počet vysílaných zpráv je $O(N^2)$

# Uniformní Hierarchický konsensus

- S použitím spolehlivého broadcastu zajistíme uniformní shodu
- Spolehlivý broadcast je složitější a zahrnuje více zpráv, použijeme jen v okamžiku, kdy je jisté, že všechny uzly ví o návrhu, leadera jehož rank odpovídá aktuálnímu kolu
- Princip:

  Leader, odešle návrh a neuloží dokud nedostane '***ack***' od ostatních (nebo nedetekuje jejich pád)

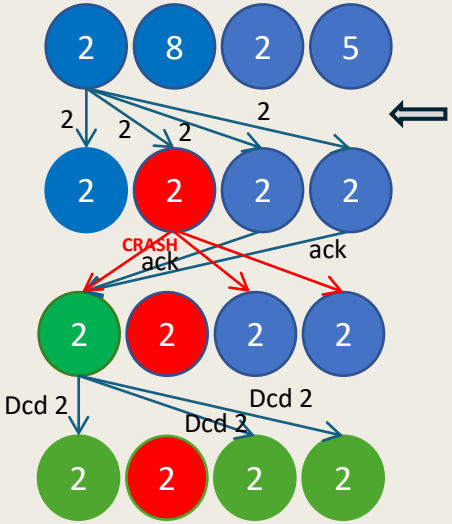
  Pokud má všechna potvrzení, ukládá a spolehlivým broadcastem posílá informaci o doruční

  Účastnící pokud obdrží návrh od lídera, přijmou jej za aktuální a pošlou '***ack***'

  Pokud účastníci obdrží informaci o doručení od lídera, informaci doručí

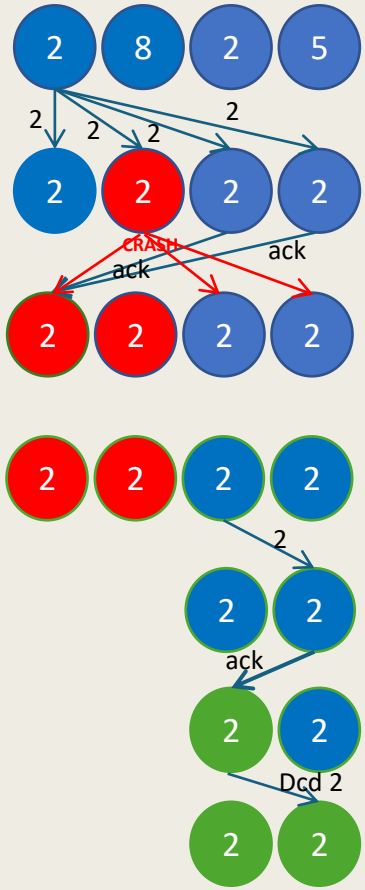
  Pokud účastníci detekují pád lídera, přechází do dalšího kola

# Uniformní hierarchický konsensus, příklady




Procesy na výzvu odpoví ‘ack’ až na jeden, který selhal. Ten je detekován a vzápětí proces 1 odešle spolehlivým broadcastem rozhodnutí, všechny korektní procesy jej doručí a rozhodnou.

Procesy na výzvu odpoví ‘ack’ až na jeden, který selhal. Ten je detekován a vzápětí proces 1 zkusí odeslat spolehlivým broadcastem rozhodnutí, ale Selže. Rozhodnutí nedoručil žádný korektní proces, detekují selhání procesu 1 a pokračuje proces 3



# Epocha lídra

- Epocha je v distribuovaném systému definována jako časový úsek od okamžiku volby lídra po jeho sesazení
- Epocha definována unikátní dvojicí: $(ts, leaderID)$
  - **ts** – Časové razítko epochy, Zajišťuje **monotonii a pořadí**.
    - Vyšší **ts** vždy „přebíjí“ nižší (logická hodiny/čítač).
  - **LeaderID (leaderID):** Identifikátor – rank lídra této epochy, zajišťuje **autoritu**.

- Společně tyto parametry vytvářejí **jednoznačný kontext**, který procesům říká:
  *"Věřím tomuto lídrovi v této konkrétní fázi vývoje systému.„*

- **Vznik nové epochy** nastává, když detektor **Ω** nahlásí nového důvěryhodného lídra
- **Akceptace (Validační pravidlo):** Proces přijme novou epochu $(ts, leaderID)$ pouze tehdy, pokud je $ts$ vyšší než jeho ***lastts*** (předchozí časové razítko)
- Tímto se automaticky „degraduje“ starý lídr – jeho zprávy už systém neakceptuje, protože mají nižší $ts$.

# Střídání epoch lídra: Epoch change

- Modul ec (epoch change) signalizuje započetí nové epochy událostí

$$< ec, StartEpoch | ts, lid >$$

- Garantuje následující vlastnosti
  - ***Monotonie**: pokud korektní proces začne epochu* $(ts, \mathrm{lid})$ *a později* $(ts', lid')$ *pak* $ts' > ts$
  - ***Konzistence**: pokud jeden korektní proces započne epochu* $(ts, \mathrm{lid})$ *a jiný* $(ts', lid')$ *a pokud* $ts = ts'$ *pak* $lid = \mathrm{lid}'$
  - ***Nakonec leader**: od určitého okamžiku všechny korektní procesy započaly stejnou epochu s korektním leaderem a další epochu nezahájí*

# Střídání epoch lídra: Epoch change, algoritmus

- Inicializuje aktuálního leadera, časové razítko aktuální epochy a vlastní časové razitko, které bude případně používat při nárokování si vlády

```
upon event <ec,init> do
        trusted:=l0, lastts=0; ts=rank(self)
```

- Detektor leadera Ω detekuje nového leadera a procesy si jej poznačí. Pokud je leaderem on, uchází se o převzetí vlády zasláním zprávy s časovým razítkem o N procesu větším než posledně (tak má každý proces unikátní sekvenci časových razítek)

```
upon event <Ω,trust|p> do
        trusted:=p
        if p=self then
                ts:=ts+N
                trigger<beb, Broadcast|[NEWEPOCH, ts]>
```

# Střídání epoch lídra: Epoch change, algoritmus

- Proces akceptuje nárok odesilatele na vládu, pokud dle jeho informací je odesilatel aktuálním lídrem a časové razítko ve zprávě je větší než posledně mu známé. Jinak odmítne

```
upon event <beb, Deliver | led, [NEWEPOCH, newts]> do
   if trusted:=led & newts>lastts then
      lastts:=newts;
      trigger <ec, StartEpoch | newts, led>
   else
      trigger <pl, Send | led, [NACK] > // NACK - not ack
```

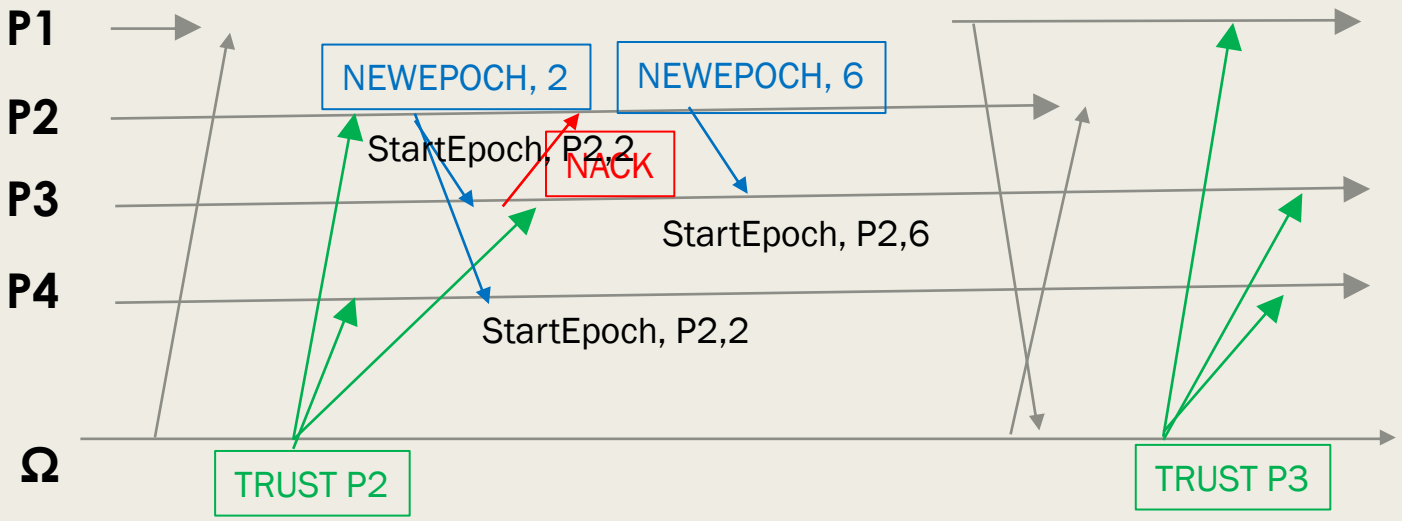
- Pokud je proces odmítnut (buď ještě nezná nového leadera, nebo jej nezná odesilatel odmítnutí) a stále věří, že leaderem je on, je tvrdohlavý a požadavek opakuje se zvýšeným časovým razítkem

```
upon event <pl, Deliver | p, [NACK] > do
    if trusted:=led then
      ts:=ts+N
      trigger<beb, Broadcast|[NEWEPOCH, ts]>
```

# Střídání epoch lídra: Epoch change, analýza

- Leader doručí svůj nárok všem procesům (platnost Best Effort Broadcast)
- Pokud nedoručí, selhal a leaderem se stane jiný proces
- Dříve nebo později, pokud neselže, doručí nárok s největším časovým razítkem
- **Monotonie** a **konzistence** plyne z algoritmu
- Vlastnost **nakonec silná přesnost** detektoru a volby lídra **Ω** zaručuje, že **nakonec** bude zvolen **lídr**, který je lídrem permanentně

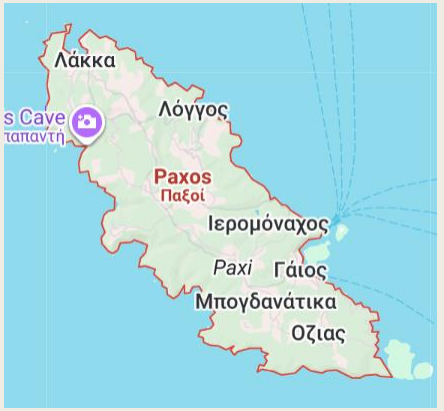
# Střídání epoch lídra: Epoch change, příklad




P3 a P4 doposud Neselhali, P3 má Vyšší prioritu

# Konsensus v asynchronním systému

- Přestože platí **FLP**, existuje algoritmus, díky kterému může dojít ke konsensu … ale(!)
- Platí v něm jen ***safety*** podmínky, ***platnost***, ***integrita*** a ***shoda***
- Neplatí ***liveness*** podmínka ***ukončení***
  - Nezaručuje, že někdy shody bude dosaženo



# PAXOS

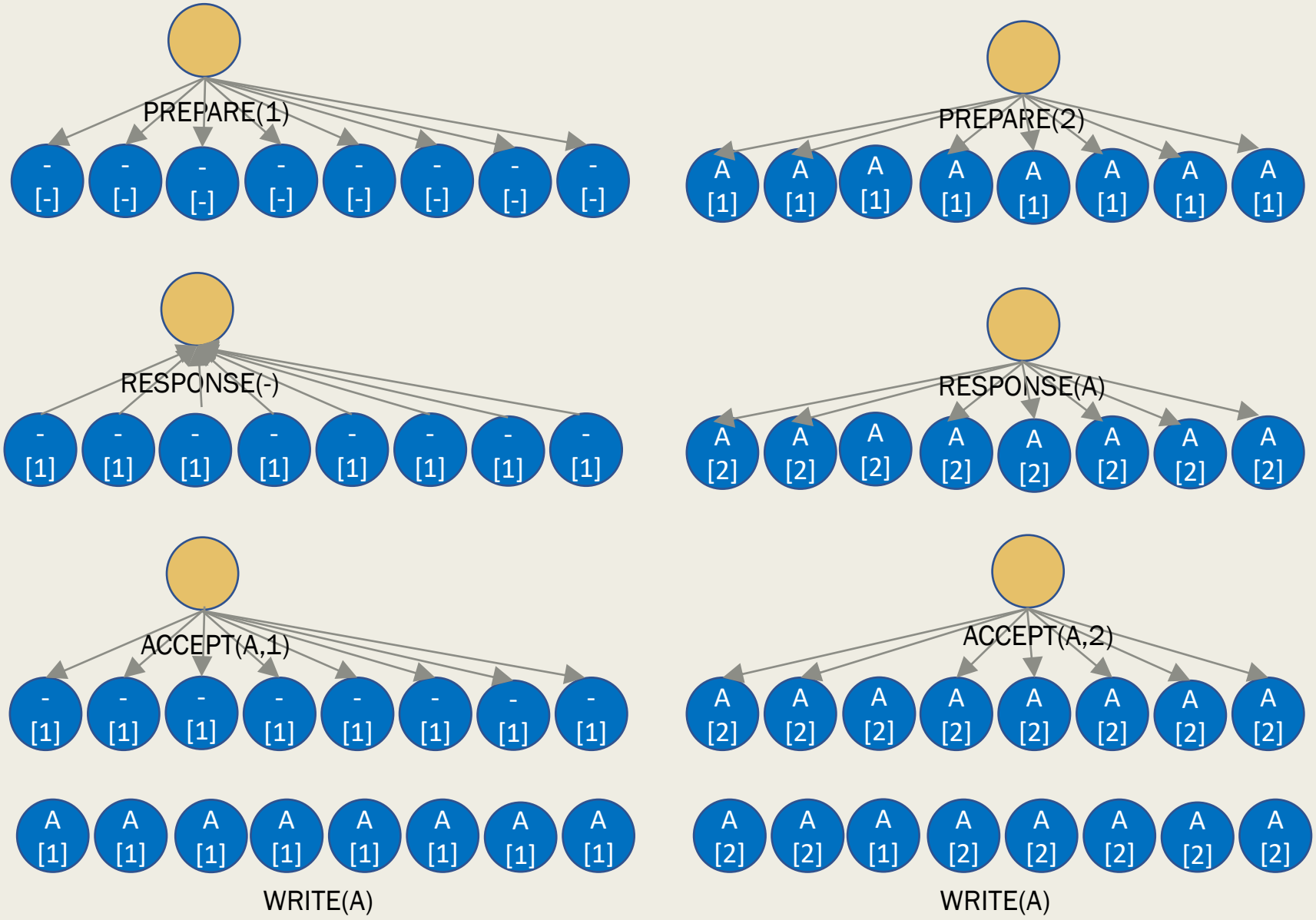
- Obecný princip
  - *Algoritmus využívá **kvórum** (nadpoloviční většinu) procesů a unikátní čísla návrhů.*
  - ***Navrhovatel** (proposer) předkládá vybrané majoritě návrh obsahující hodnotu a unikátní číslo návrhu.*
  - ***Akceptoři** (acceptors) návrh přijmou nebo odmítnou na základě porovnání čísla návrhu s těmi, která již dříve zaznamenali.*
  - *Pokud je hodnota potvrzena majoritou, stává se neměnným konsenzem a všechny budoucí návrhy musí tuto hodnotu respektovat.*
    - N/2+1 proces zapsal stejný návrh

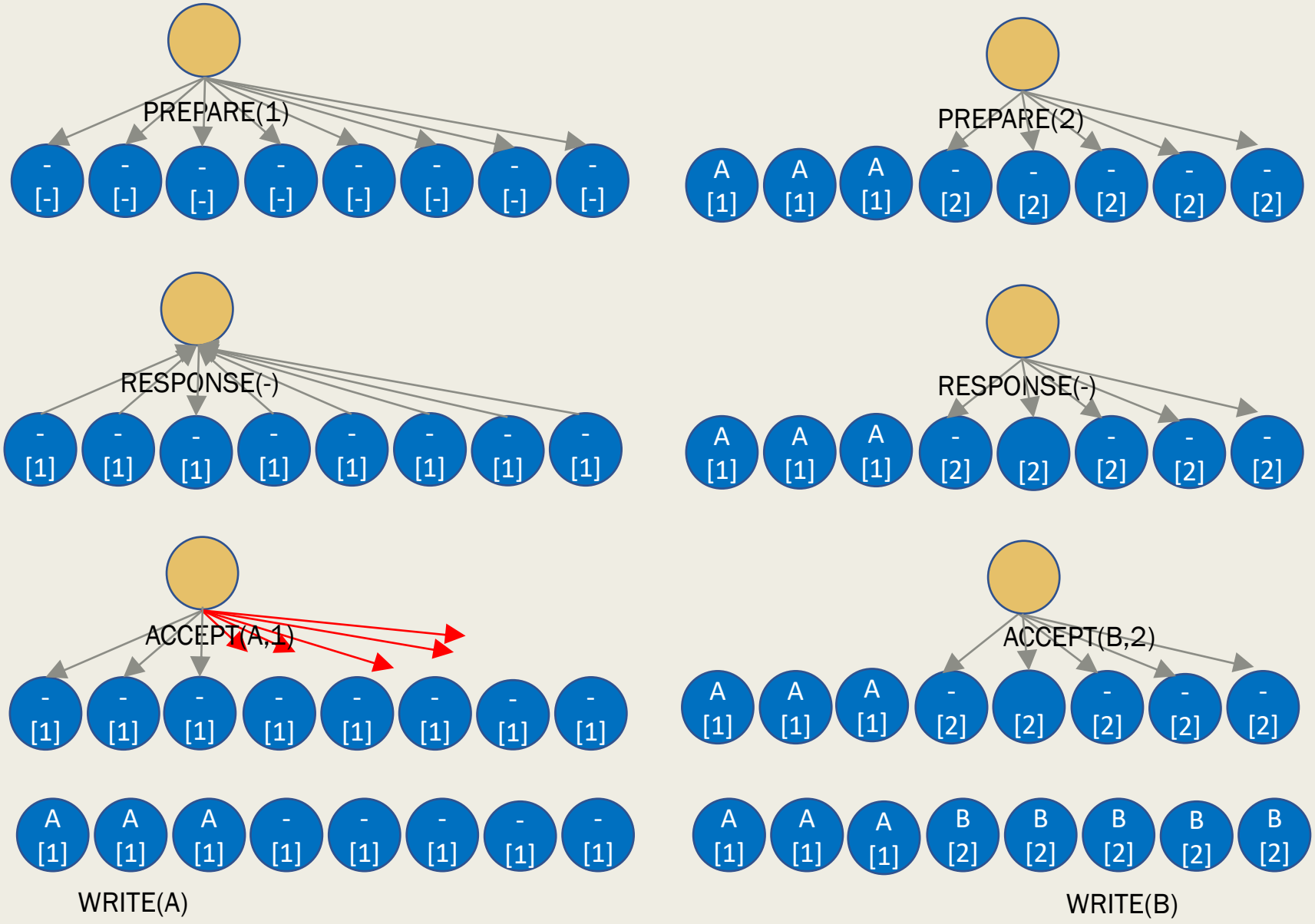
# PAXOS, Algoritmus
## Fáze přípravy

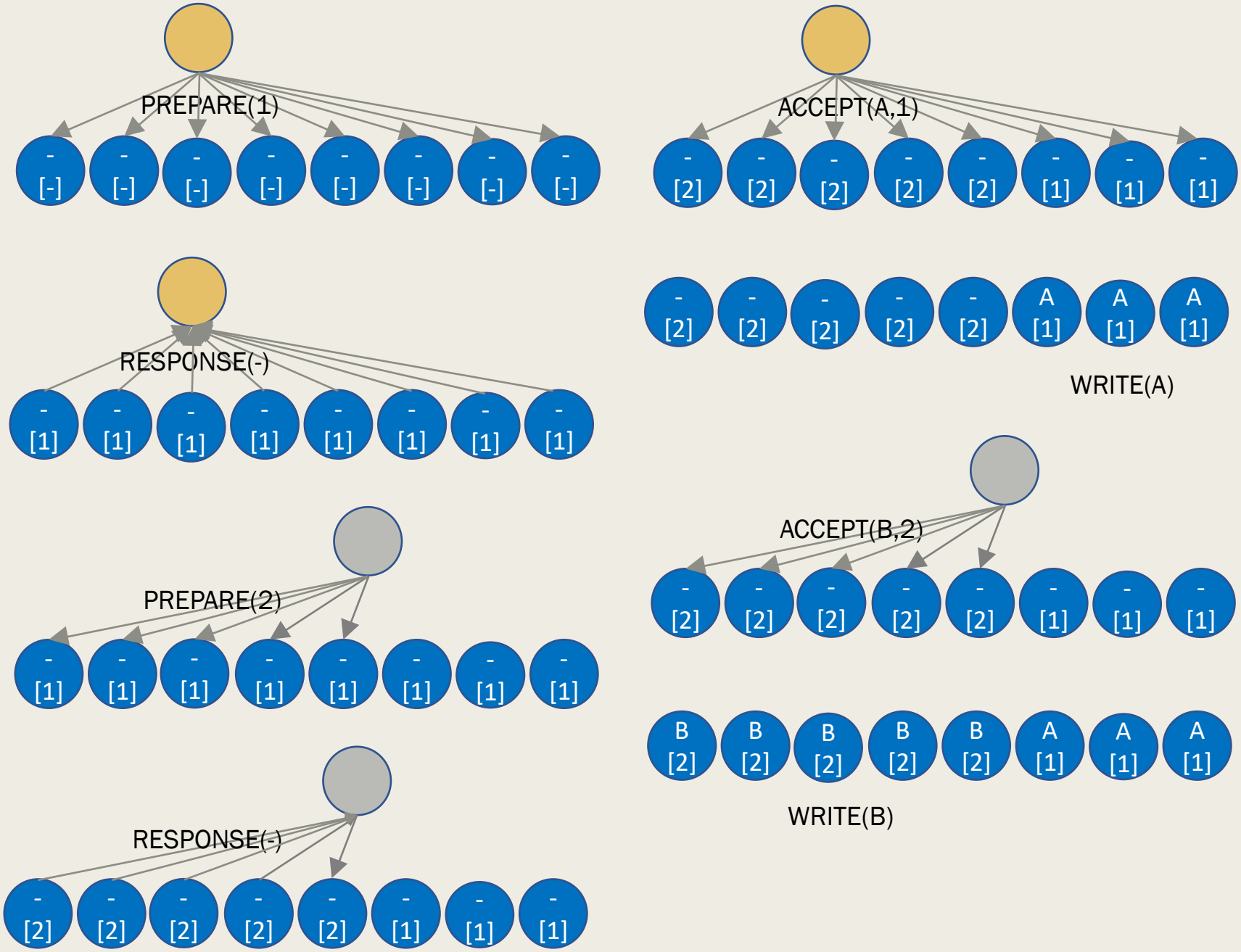
- **Navrhovatel (Proposer) připravuje půdu:**
  - Zvolí si unikátní číslo návrhu (vyšší než jakékoliv doposud použité).
  - Odešle zprávu PREPARE(číslo) na majoritu (quorum) procesů.
- **Akceptoři (Acceptors) reagují:**
  - **Ignorují:** Pokud už slíbili účast jinému návrhu s vyšším číslem.
  - **Slibují (Promise):** Pokud je číslo návrhu dostatečně vysoké, odpoví: *„Slibuji, že nebudu akceptovat žádný nižší návrh.“*
  - **Posílají info:** Pokud už dříve nějakou hodnotu „odsouhlasili“, přiloží ji k odpovědi (aby Navrhovatel věděl, co musí respektovat).

# PAXOS, Algoritmus
# Fáze přijetí

- **Navrhovatel konsoliduje:**
  - Pokud obdržel sliby od majority, může konečně poslat svůj návrh.
  - Pokud mu akceptoři v první fázi poslali zprávy, že už mají uloženou nějakou předchozí hodnotu, **navrhovatel musí tuto hodnotu převzít** (místo té své původní). Tím se zajišťuje, že se konsenzus nepřepíše něčím starším.
  - Odešle zprávu `ACCEPT(číslo, hodnota)` všem akceptorům.
- **Akceptoři rozhodují:**
  - **Přijmou:** Pokud mezitím nepřišel jiný navrhovatel s vyšším číslem, akceptor hodnotu „uloží“ (zapíše do svého logu) a potvrdí to navrhovateli.
  - **Odmítnou:** Pokud mezitím slíbili účast někomu jinému (kdo má vyšší číslo návrhu), tento ACCEPT prostě zahodí.

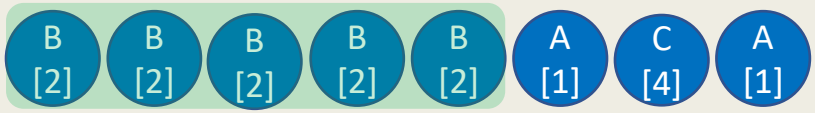

PREPARE(1)
- [-]
PREPARE(2)
A [1]
RESPONSE(-)
- [1]
RESPONSE(A)
A [2]
ACCEPT(A,1)
ACCEPT(A,2)
WRITE(A)
WRITE(A)


PREPARE(1)
- [-]
PREPARE(2)
A [1]
- [2]
RESPONSE(-)
- [1]
RESPONSE(-)
[2]
ACCEPT(A,1)
ACCEPT(B,2)
WRITE(A)
WRITE(B)
B [2]


PREPARE(1)
- [-]
- [-]
- [-]
- [-]
- [-]
- [-]
- [-]
- [-]
ACCEPT(A,1)
- [2]
- [2]
- [2]
- [2]
- [2]
- [1]
- [1]
- [1]
RESPONSE(-)
- [1]
- [1]
- [1]
- [1]
- [1]
- [1]
- [1]
- [1]
- [2]
- [2]
- [2]
- [2]
- [2]
A [1]
A [1]
A [1]
WRITE(A)
PREPARE(2)
- [1]
- [1]
- [1]
- [1]
- [1]
- [1]
- [1]
- [1]
ACCEPT(B,2)
- [2]
- [2]
- [2]
- [2]
- [2]
- [1]
- [1]
- [1]
RESPONSE(-)
- [2]
- [2]
- [2]
- [2]
- [2]
- [1]
- [1]
- [1]
B [2]
B [2]
B [2]
B [2]
B [2]
A [1]
A [1]
A [1]
WRITE(B)

# PAXOS, Algoritmus, důkaz shody

Majorita M nakonec uloží stejnou hodnotu se stejným číslem návrhu
Je možné, aby v systému byla později uložena hodnota s vyšším číslem návrhu a jinou hodnotou, než je hodnota konsensu?

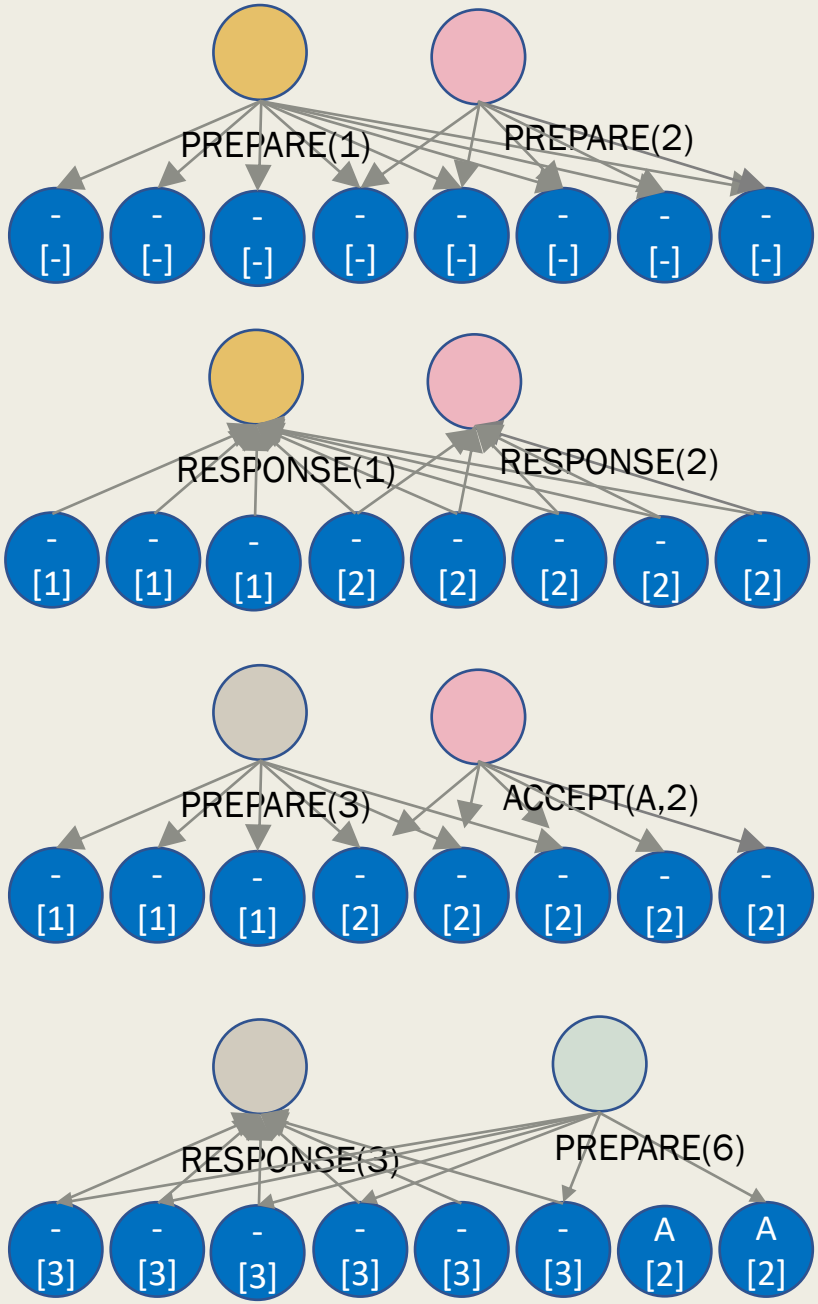



K uložení (C,4) musel navrhovatel získat majoritu procesů. Z principu majority musel jeden z procesů z M dát příslib k návrhu s časovým razítkem 4
**Předpokládejme, že je to první hypotetické uložení jiné hodnoty než B s číslem návrhu větším než 2**
Tento proces již musel uložit (B,2)
- pokud by dal souhlas k PROPOSE(4) dřív, nemohl by (B,2) uložit
- pokud jej dal později, musel připojit hodnotu B
  - ta může mít vyšší číslo návrhu, konkrétně 3, pokud v rámci jiného návrhu akceptoval návrh a uložil (ale musí jí byt B, jinak by neplatil předpoklad)
- jiný proces nemohl přislíbit s připojenou jinou hodnotou a s vyšším číslem návrhu než je 2, protože se jedná o první uložení jiné hodnoty s časovým razítkem vyšším než 2

Nemůže totiž dojit k prvnímu uložení hodnoty mimo konsensus

# PAXOS, LIVELOCK

- Pokus je mnoho navrhovatelů, kteří chtělí svůj návrh prosadit, může dojít k **livelock** (**uváznutí v neustálé aktivitě**)
- Navrhovatelé podávají návrhy se zvyšujícími se čísly návrhu, a nikdy nevznikne majorita pro jeden návrh
- **Řešení**? -> využít lídra



# Paxos s Eventually Leader-em

- **Princip:** Delegování právomoci na jednoho navrhovatele
  - Všichni účastníci věří, že existuje Leader díky *Ω (epocha lídra)*
  - Ostatní Proposeři "čekají" (nebo se stáhnou), pokud detekují aktivního Leadera.
  - Konflikty zmizí, protože návrhy podává pouze jeden proces.

- **Shoda**, **Integrita**, **Platnost:** Stále platí – i když Leader Detector selže (např. v jednu chvíli máme dva leadery), Paxos je díky své podstatě stále bezpečný (safety neztrácíme).
- **Ukončení:** Obnovena v momentě, kdy Detector stabilizuje na jednom správném Leaderovi

# BYZANTSKÉ PROCESY

FZjr 2019 - 2024

# Byzantské procesy

- Nesprávné ukončení procesu nemá žádná pravidla a chování procesu není nijak vymezené
- Proces dokonce může, pokud je ukončen napadením, předstírat korektní chování, přitom poskytovat nepravdivé informace
- Detekce uvedené v předchozích částech jsou nepoužitelné

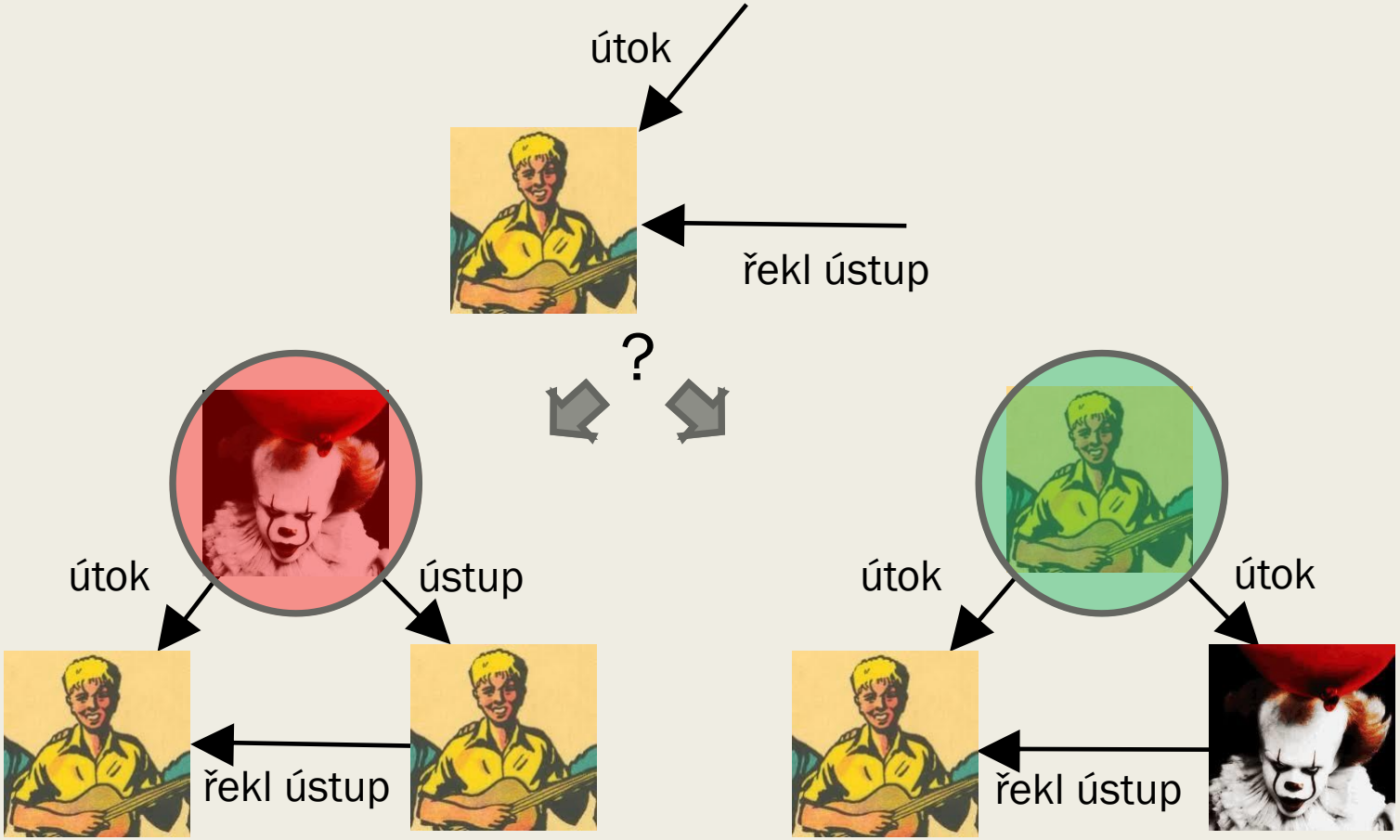
# Problém Byzantských generálů – příběh

- Vojsko Byzantské říše složené z několika armád dobývá město.
- Každý generál velí své armádě a komunikuje s ostatními generály pomocí seržantů.
- Generálové nebo seržanti mohou být agenty nepřítele a jejich záměr jde proti zájmům Byzantských. Nepřátelský agent - generál může vydávat falešné zprávy, aby věrní generálové nedosáhli konsensu,
- Útok na město se blíží a všechny armády musí zaútočit ve stejný okamžik, jinak je téměř jistá porážka útočících vojsk s velkými ztrátami
- Věrní generálové se musí dohodnout na době útoku a to navzdory možným nepřátelským agentům ve svých řadách.

# Problém Byzantských generálů

- Dohody nelze dosáhnout, pokud byzantských procesů je jedna třetina nebo více.
- **Důkaz**: Ukážeme pro tři procesy, kde je jeden generál a dva seržanti, že v případě jednoho zkorumpovaného člena nejde dosáhnout dohody
- Pokud general není loajální, pak pro způsobení ztrát vydá odlešné rozkazy oboum seržantům …
- Požadujeme:
  - *IC1: Všichni věrní seržanti vykonají ten samý příkaz*
  - *IC2: Pokud příkaz vydal věrný generál, vykonají jej takto všichni věrní seržanti*

# Problém Byzantských generálů



# Pro *n* účastníků a *m* zrádců

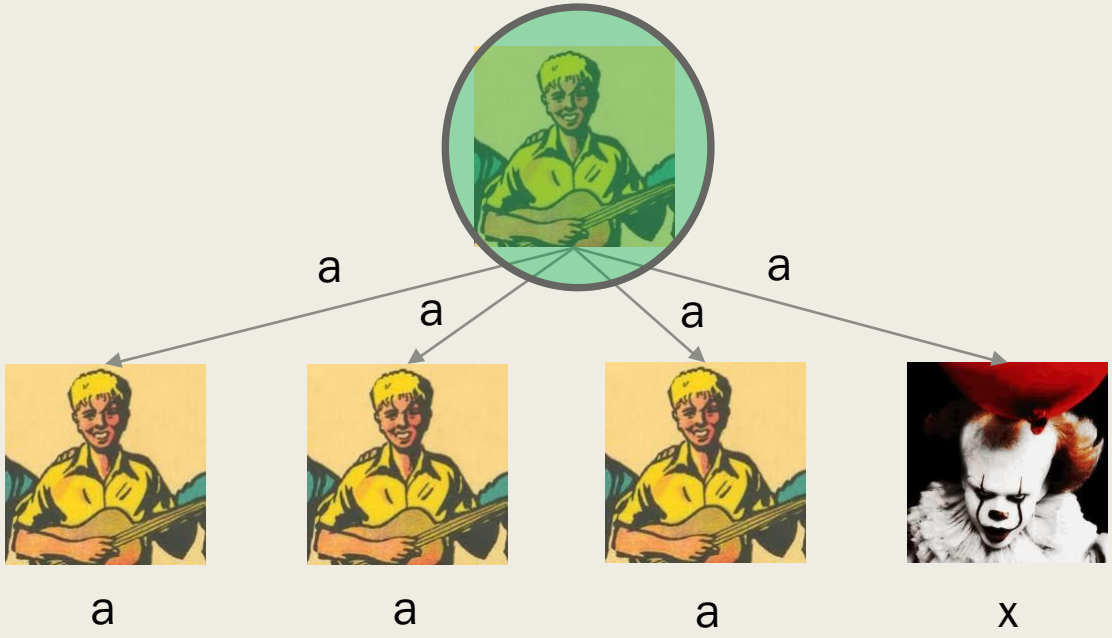
- *n* Albánských generálů a z toho *m* zrádců
- Pokud **n<=3*m** problém nemá řešení
- **Spor**
  - *Dobrá, umíme vyřešit problém pro n=3*m Albánských generálů …*
  - … pak bychom ale měli být schopni řešit i problém tří Byzantských generálů s jedním zrádcem!
  - Protože bychom redukovali Albánce po skupinách na Byzanťany, zlé do skupiny jedné, zbylé do ostatních dvou skupin a výstupem skupin by byla zjištěná shoda.

# Algoritmus pro řešení Problému Byzantských generálů

- Algoritmus OM přijímá parametr *m*
- Algoritmu OM(0)
  1. *Generál posílá jeho hodnotu každému ze seržantů*
  2. *Každý seržant použije tuto hodnotu, pokud je mu doručena, pokud ne, použije defaultní hodnotu (například RETREAT)*
- Algoritmu OM(m), pro m>0
  1. *Generál posílá jeho hodnotu každému ze seržantů*
  2. *Pokud seržant i obdrží hodnotu, označíme ji vi, pokud žádnou neobdrží, pracuje s defaultní hodnotou. Posléze provede algoritmus OM(m-1), ve kterém nahradí na pozici generála původního generála a zašle tuto hodnotu všem ostatním seržantům*
  3. *Pro všechny i a pro každého j seržant platí, že seržant i obdrží zprávu vj po ukončení kroku 2 (výsledek OM(m-1) pro tohoto seržanta), nebo Defaultní hodotu, Pak je jím zvolenou hodnotou (na této úrovni rekurze) hodnota majority(v1 … vn-1)*

# Generál věrný, OM(0)

- Pokud jsou všichni věrní, není co řešit a OM(0) funguje pro IC1 a IC2
- Algoritmus OM(0) funguje vždy, když je generál věrný, tj. pro libovolný počet zrádců mezi seržanty
- Pravidlo IC2 je splněno tím, že věrní seržanti splní generálův rozkaz



# Generál věrný, OM(m)

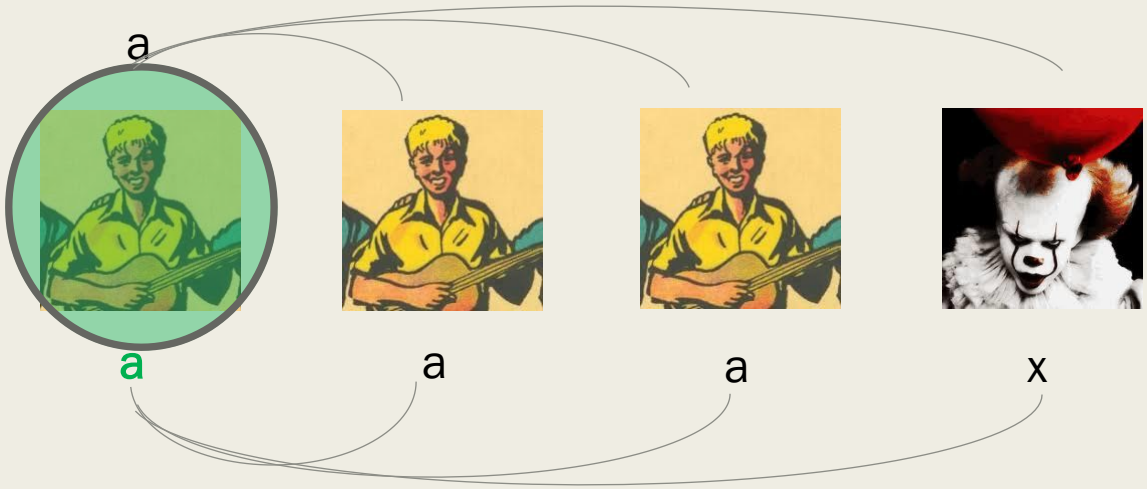
- Budeme potřebovat, aby fungoval i algoritmus OM(m), m>0
- Pro případ věrného generála je větší *m* komplikací, ale bez toho to pro obecný případ i možných nevěrných hlavních generálů nepůjde

tedy …

- Každý seržant je generálem pro skupinu bez původního generála a nechá řešit ostatní problém s tím, že  mu tito sdělí své rozhodnutí
- Z těchto rozhodnutí zvolí majoritu a tuto příjme jako své rozhodnutí.

# OM(1), generál věrný

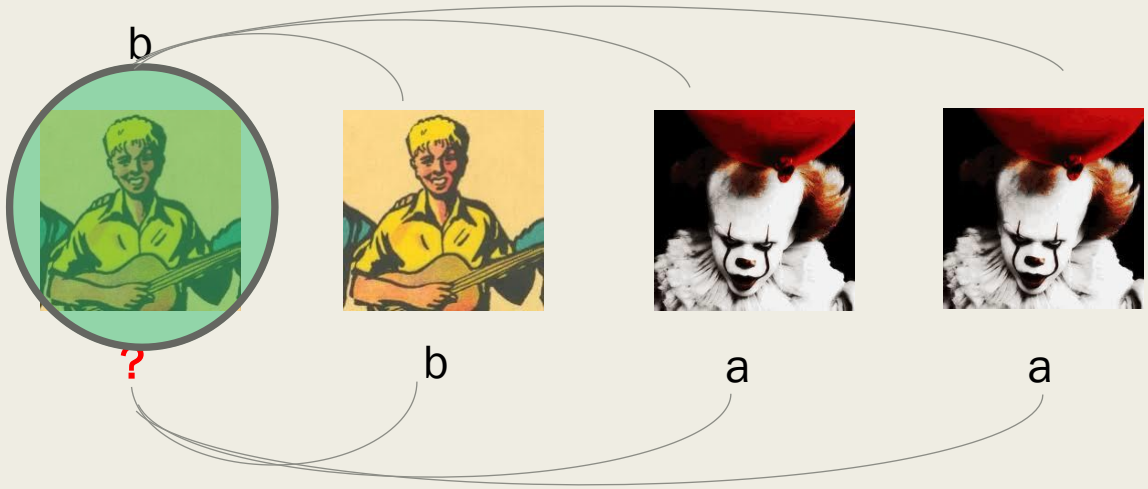
- Pokud byl původní generál věrný, obdrží všichni seržanti stejnou hodnotu, a proto v případě vyjednávání ve skupině bez původního generála dojde ke shodě mezi věrnými seržanty, pokud jich je více než zrádců.
- První seržant navrhne hodnotu, ostatní věrní hodnotu přijmou a první seržant jakmile zjistí, že nadpoloviční většina ostatních seržantů hodnotu přijala, příjme ji také (dle principu majority)
- Všichni ostatní věrní seržanti si stejným způsobem ověří svoji hodnotu




majority(a,a,a,x)

# OM(1), generál věrný

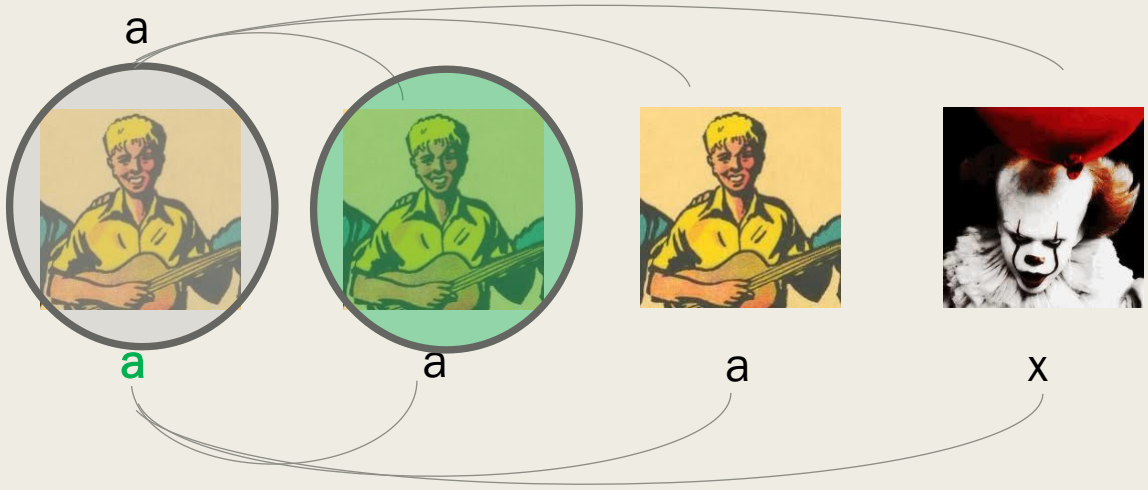
- V případě dvou zrádců z celkového počtu pěti vojáků (i s původním generálem) již algoritmus OM(1) nefunguje správně
- Zrádci mou poslat jinou hodnotu rozhodnutí pokud hledání shody vyhlásí věrný první seržant a jinou pokud druhý seržant hledá shodu na rozkazu.




majority(a,**a**,b,b)

# OM(2), generál věrný

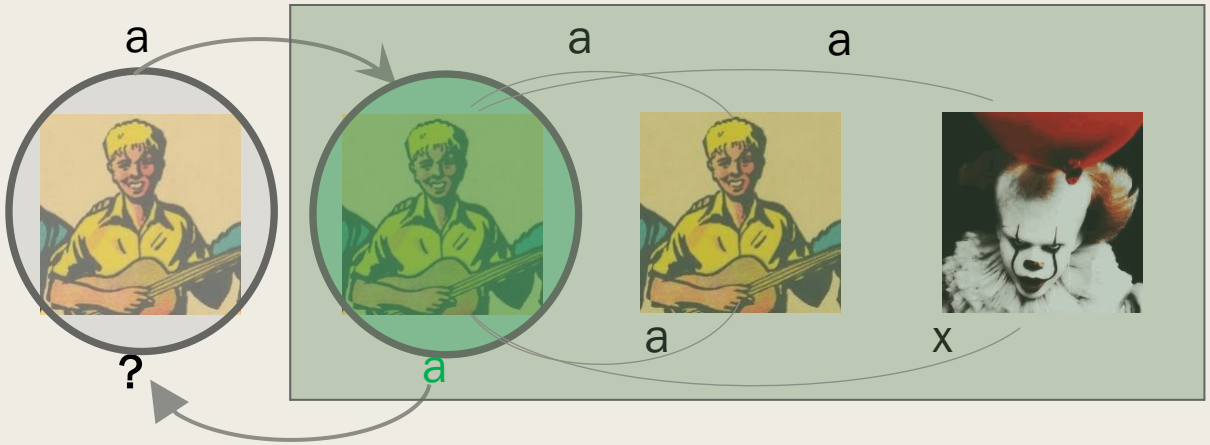
- V případě dalšího kola vyjednávání o hodnotách, kdy zbylí tři seržanti neodpoví ihned, ale utvoří další podskupinky bez původních generálů a radí se navzájem.
- Tedy věrní hodnotu nepotvrdí ihned, ale v tomto případě dojde ke zmatení, protože v o jednoho věrného menších podskupinkách získá zrádce navrch a zmate zbývající věrné!




majority(a,a,a,x)

# OM(3), generál věrný

- V případě dalšího kola vyjednávání o hodnotách, kdy zbylí tři seržanti neodpoví ihned, ale utvoří další podskupinky bez původních generálů a radí se navzájem.
- Tedy věrní hodnotu nepotvrdí ihned, ale v tomto případě ještě nedojde ke zmatení, protože v o-jednoho věrného menších podskupinkách získá zrádce trochu navrch, ale stále dva loajální určí správce




majority(a,a,?,?) majority(a,a,x)

Zjistí stejným způsobem pro L3 a zrádného L4

# OM(m), generál věrný

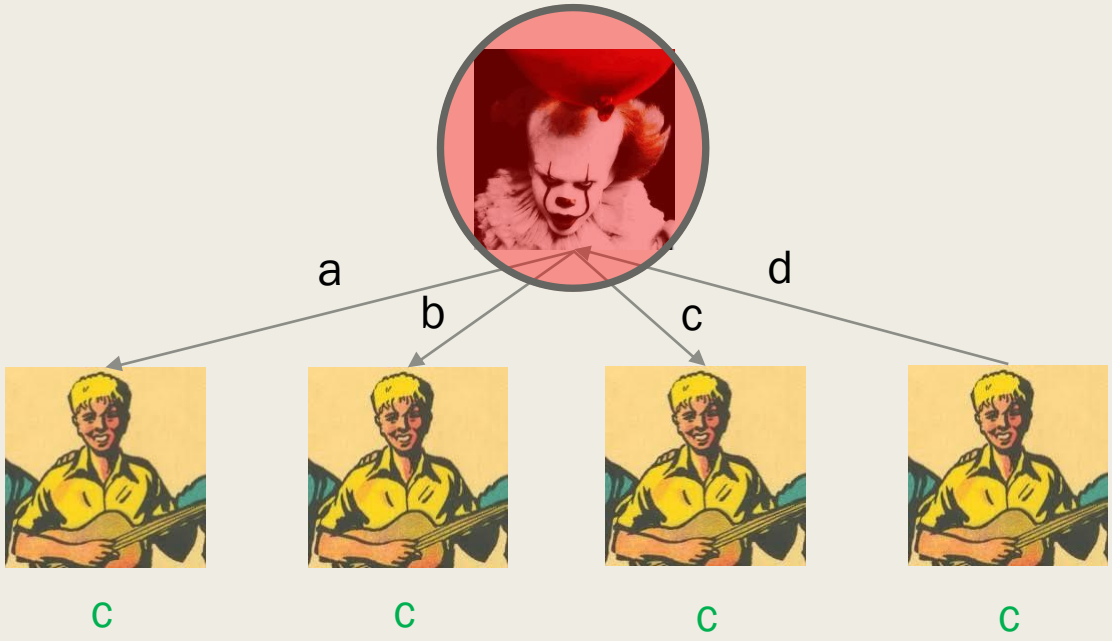
- Algoritmus OM(m) splňuje podmínku IC2, a tím i podmínku IC1, v případě, když hlavní, původní rozkaz vydávající generál je věrný, pokud platí

$$\text{CELKOVÝ\_POČET\_VOJÁKů} > 2*\text{POČET\_ZRÁDCů} + m$$

- V našem případě bylo pět vojáků, jeden zrádce, a fungovalo nám i OM(2), protože 5>2*1+2
- OM(3) by již nefungovalo, neplatí 5>2*1+3,
- Stejně tak nefungovalo již OM(1) pro dva zrádce. Neplatí totiž 5>2*2+1

# OM(1), generál zrádný

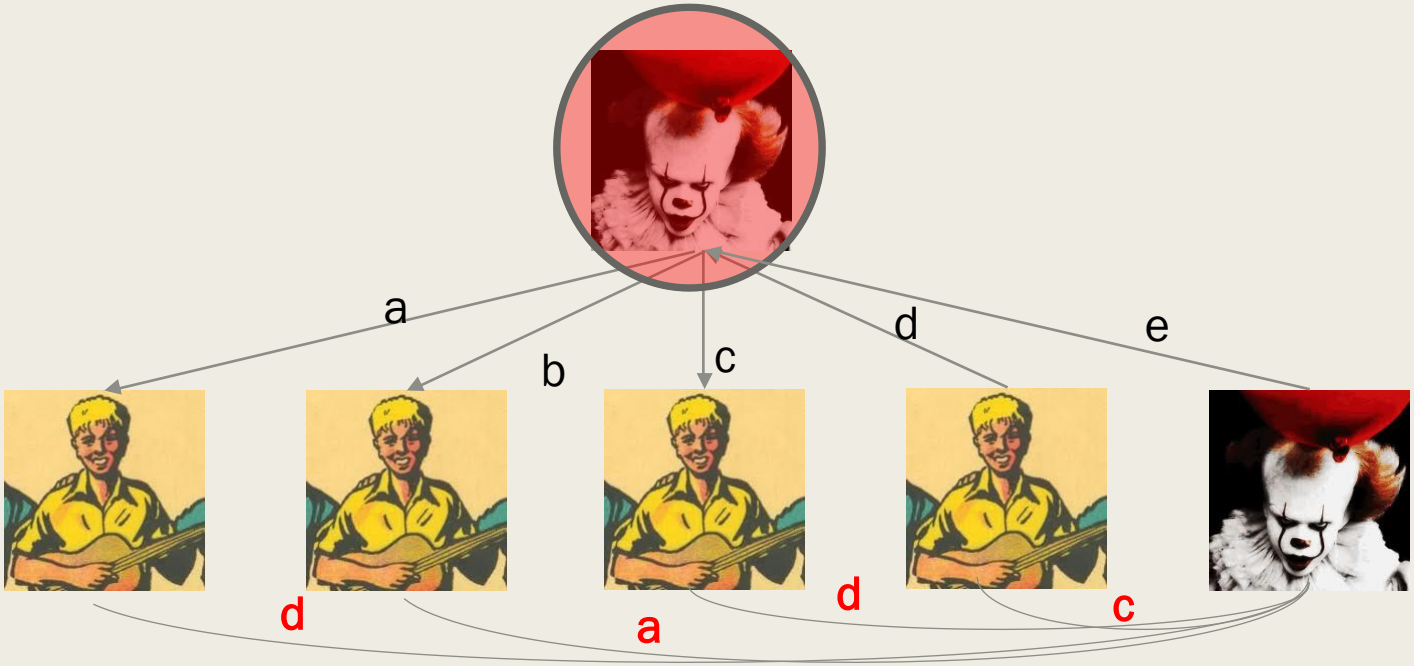
- Skupina seržantů si dokáže poradit se zrádným generálem, pokud jdou alespoň tři a všichni jsou věrní a to algoritmem OM(1).
- Poradí se, předají si hodnoty, které jim generál zaslal a shodnou se na majoritě, protože jako věrní nelžou a každý řekne po pravdě obdrženou hodnotu
- Je splněna podmínka IC1 a IC2 neřešíme, ta je jen pro případ věrného generála




majority(a,b,c,d)

# OM(1), generál zrádný (+1)

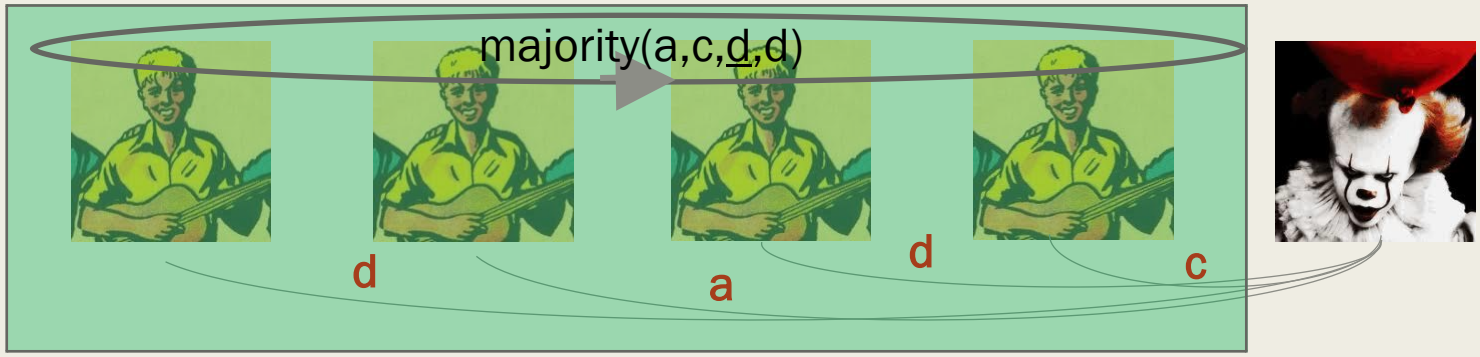
- V případě dvou zrádných vojáků z pěti by algoritmus nefungoval ani v případě věrného generála. Pro šest vojáků a dva zrádce ano.
- Fungoval by algoritmus OM(1) pro zrádného generála, jednoho ze seržantů a čtyř věrných?
- Nefungoval: při vyjednávání o generálově rozkazu zrádný seržant zmaří shodu věrných podstrčením různých údajně obdržených hodnot.




majority(a,b,c,d,?): např majority(a,a,b,c,d)≠ majority(a,b,c,c,d)

# OM(2), generál zrádný (+1)

- Pokud by se ale poradili o jedna menší skupiny o každé zprávě. Tyto skupiny vyvolá každý věrný seržant
- Pokud se bude diskutovat o zprávě od věrného seržanta, bude v podskupině přítomen i zrádce, ale převaha věrných se shodne na zprávě věrného tak, jak ji poslal.
- Pokud se bude diskutovat o zprávě zrádce, budou jednat jen věrní seržanti a ti pokaždé předloží jimi obdrženou zprávu od zrádce a každý vektor zpráv v této skupině vyhodnotí stejně. Každý iniciuje jednání o zprávě, kterou dostali od zrádce a vyhodnotí (4 x majority(a,c,d,d))




- Každý iniciuje jednání o zprávě, co dostali od zrádce (4 x majority(a,c,d,d)) Čímž si věc ujasní a shodnou se na zprávě od zrádce, v tomto případě na d
- Nyní již mohou tuto hodnotu každý použít v původním OM(1) z minulého slajdu a loajální se shodnou na stejné akci -> Platí IC1

# Problém Byzantských generálů. Řešení pro *m* zrádců

- **Algoritmus OM(m) řeší problém Byzantských generálů, pokud je maximálně m zrádců a celkem je více než 3*m vojáků**
- Víme, že pro případ věrného generála dokáže algoritmus OM(m) zajistit obě podmínky IC1 a IC2, pokud počet vojáků je větší než dvojnásobek zrádců plus *m,* což platí, protože zrádců je *m*, stejné číslo je argumentem tohoto algoritmu, a vojáků je předpokládáno více než 3*m, což je ‘přesně potřeba’

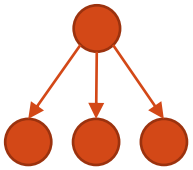
# Problém Byzantských generálů. Řešení pro *m* zrádců

- **Algoritmus OM(m) řeší problém Byzantských generálů, pokud je maximálně m zrádců a celkem je více než 3*m vojáků**
- Důkaz indukcí funkčnosti OM(m) pro případ zrádného generála a 3*m-1 seržantů
- OM(0) platí pro situaci bez zrádce triviálně
- Indukcí, pokud OM(m-1) splňuje IC1 a IC2, pak lze dokázat splnění těchto podmínek i OM(m)
- Pokud je generál zrádný, pak v diskusi o jeho rozkazech OM(m-1) z druhého kroku algoritmu je více než (3*m-1) vojáků a zrádců je m-1. Musí platit, že je vojáků > 3* zrádců, tedy 3*m-1 > 3*(m-1), což platí
- Provedením OM(m-1) každí dva věrní seržanti se shodnou na stejné hodnotě vj.

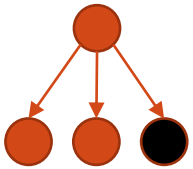
* pozn.: jelikož generál je zrádce, je v OM(m-1) o jednoho méně zrádců, a tedy buď již generál zrádce není, a OM(m-1) funguje, nebo je opět generál zrádce a v následně použitém OM(m-2) je o zrádce méně. Pokud jsou pokaždé generálové zrádci, dostaneme se po m ‘zanořeních’ do OM(0), kde již žádný zrádce nezbývá.

# PROBLÉM BYZANTSKÝCH GENERÁLŮ, PŘÍKLAD

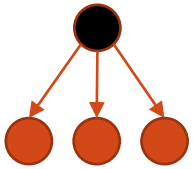
# 1 ZRÁDCE PRO OM(0)



OM(0) funguje – tj. všichni poctivci si uloží došlou hodnotu, pak IC1 i IC2

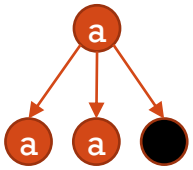


OM(0) funguje (!)



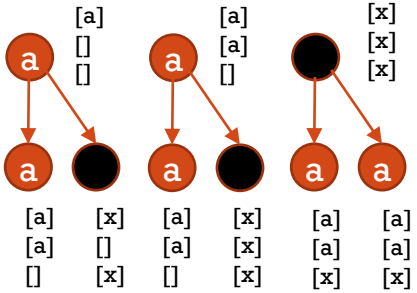
OM(0) nefunguje – poctivci si uloží hodnotu od zrádce a každý může mít jinou

# 1 ZRÁDCE PRO OM(1)



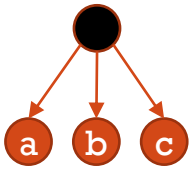

OM(1) funguje taky, i když se nám to trochu zkomplikuje

Všichni sice od poctivce obdrží tu samou informaci, OM(0) by problém řešila, ale zkusíme, jestli je toto robustní pro jedno kolo porady



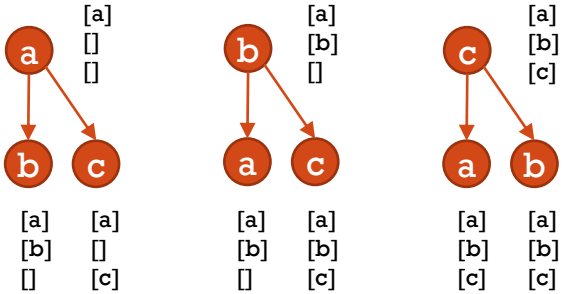

Co nakonec tvrdí zrádce je jedno, důležité je, aby oba poctivci zvolili stejnou hodnotu, a tu zvolí, protože nakonec u obou ve vektoru hodnot bude původní hodnota od poctivého generála dvakrát, od zrádce nějaká hodnota jednou, a mediánem bude ona původní hodnota od poctivého maršála

# 1 ZRÁDCE PRO OM(1)

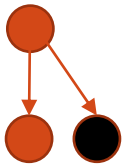



Pokud oním zrádcem je maršál, MP(0) nefunguje, ale co porada?

Radí se vždy ti samí poctivci, akorát pokaždé hledá ‘pravdu’ jiný z nich. Ale vektor výsledků
Bude vždy stejný a každý zvolí tu samou střední hodnotu

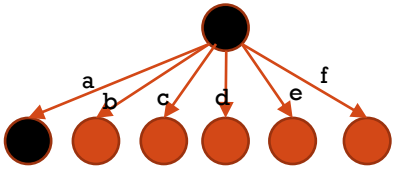


# 2 ZRÁDCI, OM(2), MARŠÁL VĚRNÝ

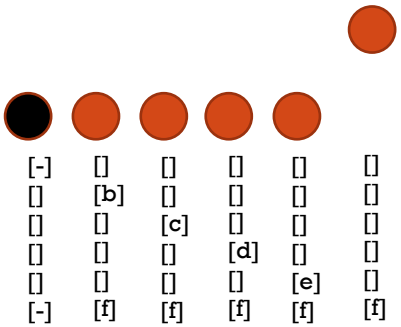


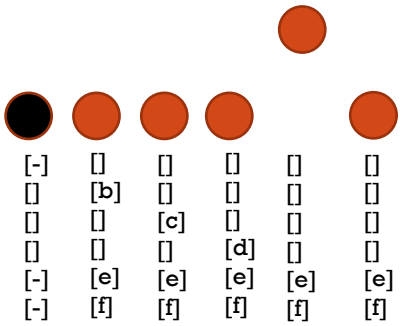
Při první poradě má každý poctivý generál (zahajovatel první porady) správnou hodnotu od věrného maršála, jeden ze seržantů také, a druhý seržant je zrádce
Druhá porada probíhá mezi oběma seržanty, u zrádce by to byla jen předstíraná snaha, ale poctivý seržant je manipulován zrádcem a … může zvolit jinou hodnotu než poctivý generál zadal -> viz úvodní příklad
TEDY MP(2) nefunguje pro tři poctivce a jednoho zrádce!!!
Protože pro systém s *n* zrádci funguje max. OP(n), pokud je poctivců vic než *n*

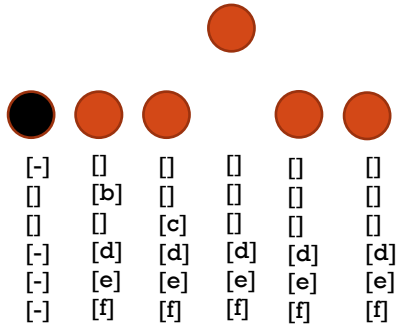
# 2 ZRÁDCI, OM(2), MARŠÁL ZRÁDNÝ

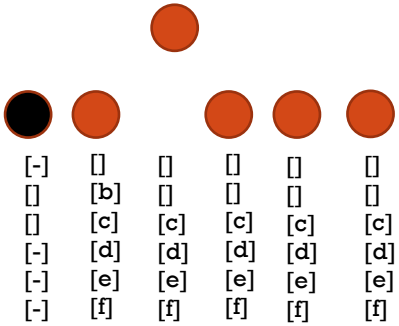


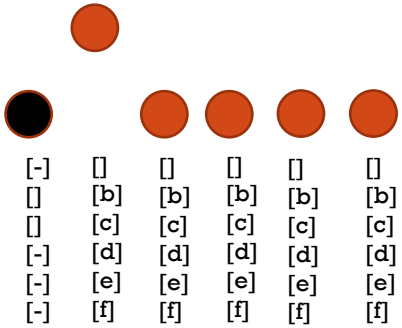

Každý provede OM(1) pro data od maršála. Tedy vyloučí jej z rozpravy a zeptá se ostatních, jakou informaci dostali
Jelikož je to OM(1), tak pro každou odpověď vyvolají diskuzi opět

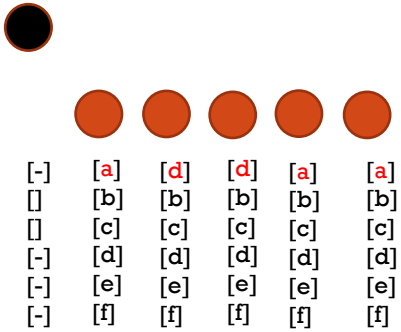


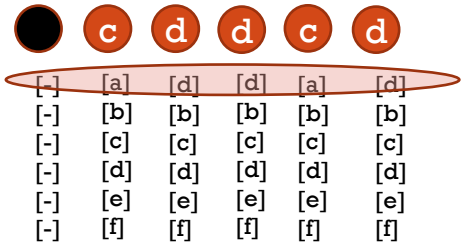



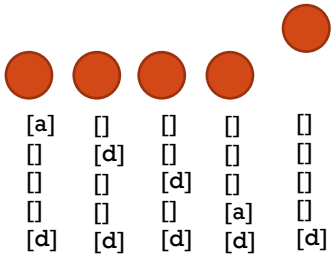



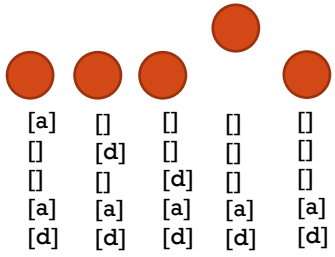

MED

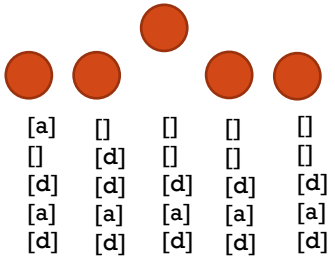


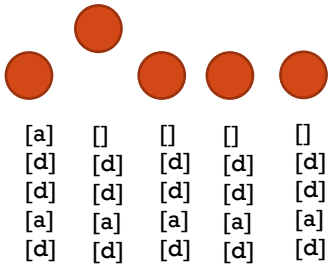

Nyní budou šestkrát debatovat o každé z šesti hodnot. Každá debata obsahuje jedno rozeslání od generála a vyhodnocení. Zkusme debatu u prvních dvou hodnotách, té od zrádce, a potom od prvního věrného zleva

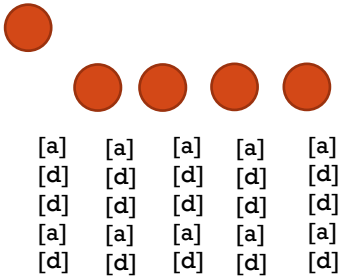
V debatě o hodnotě zrádce tohoto vynechají a provedou …



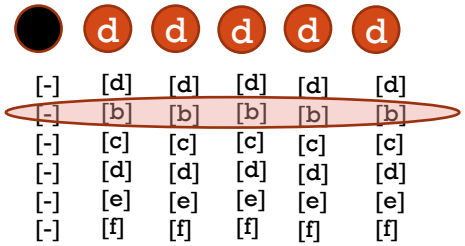



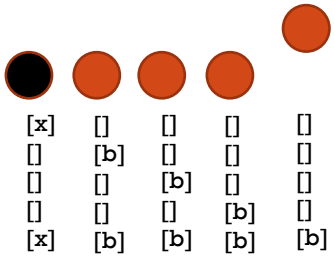



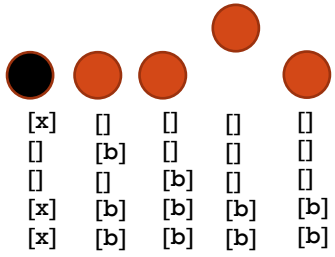



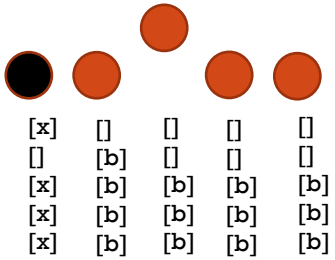


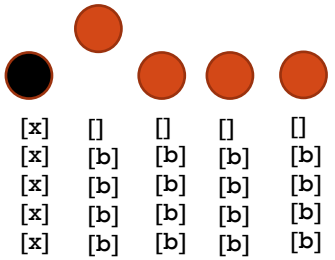


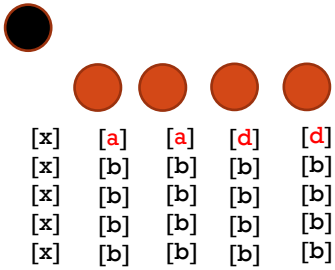

O hodnotě prvního (a zrádného) mají jisto, ale musí se shodnout pro jistotu i na hodnotách ostatních. Uvedeme shodu pro druhého, nyní loajálního, seržanta a obdobně pak bude probíhat shoda na ostatních hodnotách



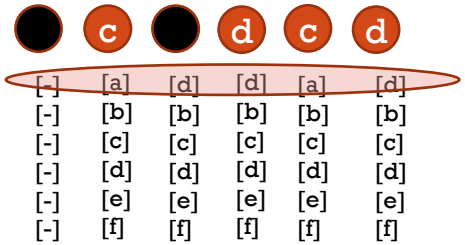



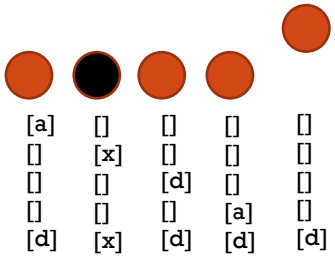



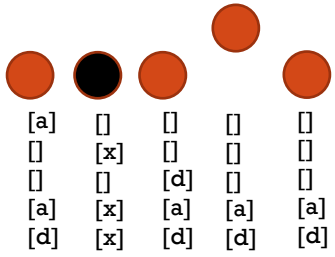



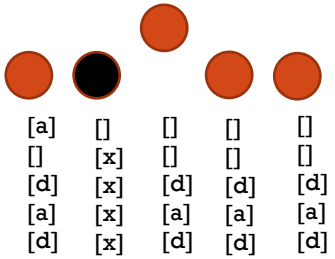


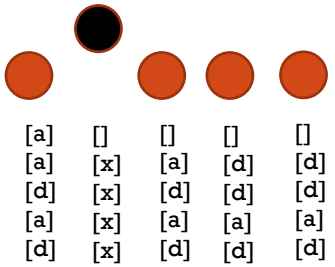


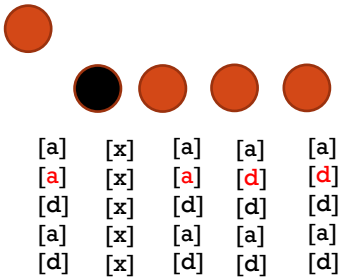

Zpět ale k příkladu, kde se jedná o shodě na informaci od prvního seržanta – zrádce. Pokud by zrádní měli být vedle maršála i dva seržanti, pak …
Musí se šestkrát dosáhnout shody na 25ti hodnotách ve skupinách po čtyřech
⇒ 6*5*4 jednání o shodách, každé jednání mezi čtyřmi agenty
Navíc tento systém (tři zrádci ze sedmi) nebude fungovat v případ věrného maršála! Pro tři zrádce musí být minimálně sedm poctivců.



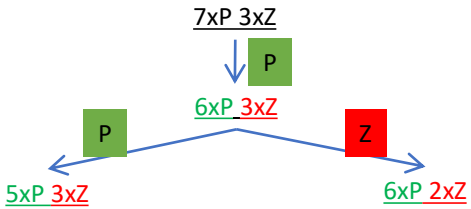





# OM(2) PRO GENERÁLA P, CELKEM 7P A 3Z




O(1)

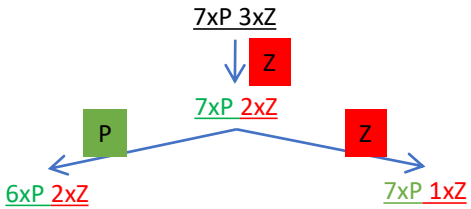
O(2)

Shoda korektních na zprávě?

Pro P↓ potřebujeme většinu

Pro Z↓ potřebujeme všechny

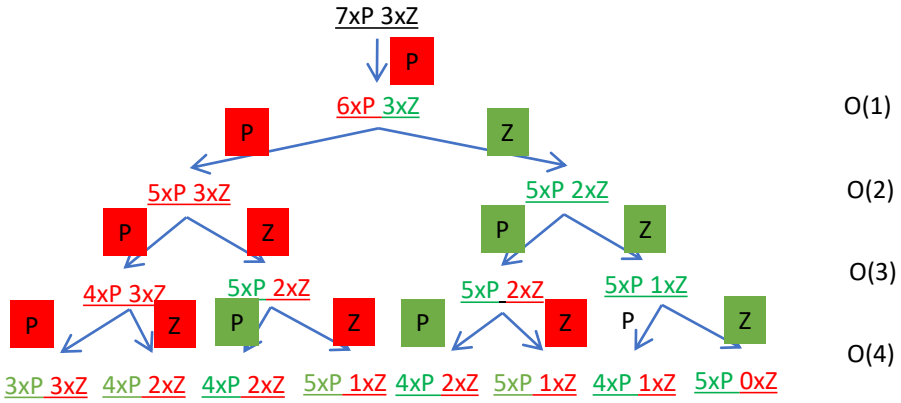
# OM(2) PRO GENERÁLA Z, CELKEM 7P A 3Z



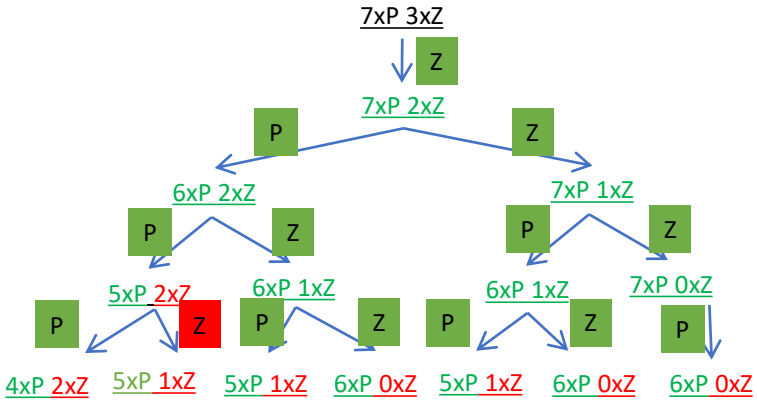

O(1)

O(2)

# OM(4) PRO GENERÁLA P, CELKEM 7P A 3Z



# OM(4) PRO GENERÁLA Z, CELKEM 7P A 3Z



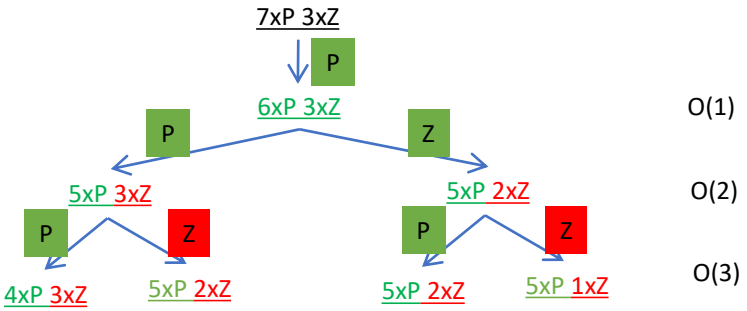

O(1)

O(2)

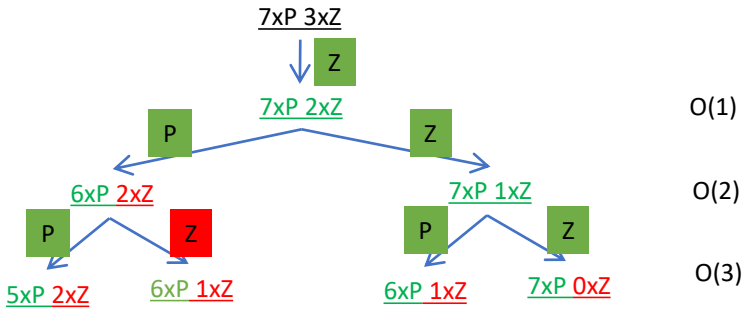
O(3)

O(4)

# OM(3) PRO GENERÁLA P, CELKEM 7P A 3Z



# OM(3) PRO GENERÁLA Z, CELKEM 7P A 3Z



# Rotating Byzantine Leader Detection

- Požadavky:
  - *Nakonec úspěch (Eventual succession): pokud více než f korektních procesů ztratí důvěru v leadera, nový leader dostane důvěru korektních procesů*
  - *Odolnost proti puči (Putsch resistence): novému / jinému než stávajícímu leaderu korektní proces uvěří jen když si aspoň jeden proces stěžuje na stávajícího lídra*
  - *Nakonec shoda (Eventual agreement): procesy budou po nějakém čase (pro koordinaci) věřit stejnému leaderu*

***Princip:***

- Procesy si mohou ‚stěžovat' na chování leadera z vlastní zkušenosti
- Pokud je korektnímu zřejmé , že si stěžuje aspoň jeden korektní, ke stížnosti se připojí
- Pokud si stěžuje více než polovina korektních procesů, vymění leadera

# Rotating Byzantine Leader Detection, algoritmus

- Algoritmus pracuje s koly kdy číslo kola určuje rank potenciálního lídra. Udržuje seznam procesů. které si na aktuálního lídra stěžovali a příznak, zdali si tento proces také na lídra stěžoval
- Pokud sezná, že lídr se chová nekorektně, zašle stížnost všem procesům

```
upon event <bld, init> do
     round:=1; complainist:=[]; complained:=False;
     trigger <bld, Trust | LEADER(round)>

upon event <bld, complain | p> such that p=leader(round)
                                    & complained=False
     complaint:=True;
     trigger <al, Send | q, COMPLAINT ,round] to all processes
```

# Rotating Byzantine Leader Detection, algoritmus

Zpracovává se událost, že přišla informace o stížnosti na aktuálního lídra a od tohoto procesu stížnost ještě nemáme poznačenou → poznačíme si

```
upon event <al, Deliver | p, COMPLAINT, round>
      such that r=round and complainist[p]=nill do
            complainist[p]=COMPLAINT
            if (#complainist)>f &
                        complained=FALSE then
                  complained=TRUE
                trigger <al, Send | q, COMPLAINT, round]> to all processes
            else (#complainist)>2f then
                  round:=round+1; complainist:=[]
                  complained:=false;
                  trigger <bld, Trust | LEADER(round)>
```

Pokud je stěžovatelů více než nekorektních a proces si ještě nestěžoval, připojí se (aspoň jeden korektní si stěžuje → s lídrem něco je)

Důvěru v nového lídra daného rankem round+1 získáme, pokud si na původního stěžovala napolovičí většína korektních

# Princip ‚lavinového' šíření stížností

Stežuje si korektní proces 3

Nekorektní proces 5 posílá stížnost jen dvěma korektním 4 a 7

Stežuje si korektní proces 10

Stežuje si korektní proces 1

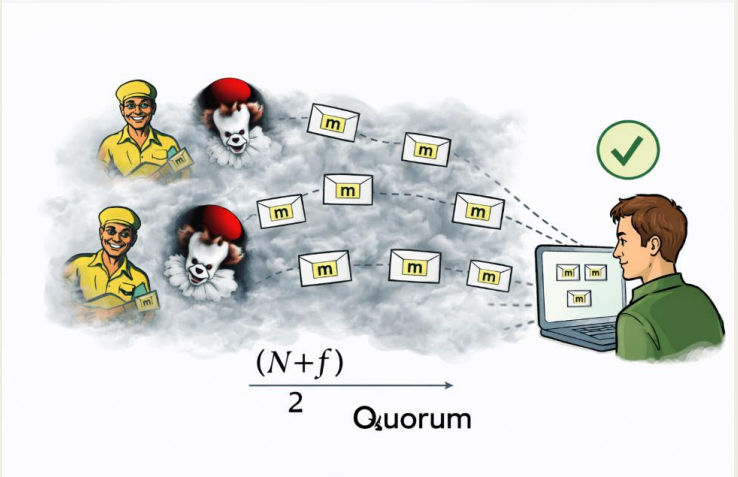
Korektní 4 a 7 mají >f stížností, připojují se

Stěžuje si nadpoloviční většina korektních, všechny korektní se připojují ke stížnosti → nakonec dojde k >2f stížnostem → změna leadera

# Rotating Byzantine Leader Detection, princip

- Protože se vyžaduje, aby přišla stížnost alespoň od f+1 procesů, nekorektní procesy samy nemohou vyprovokovat změnu lídra
- Pokud obdržel 2f+1 stížností, je jisté, že si stěžuje alespoň f+1 korektních
  - *Tedy každý korektní obdržel jistě f+1 stížností*
  - *Zamezím případu, kdy všichni Byzanští pošlou stížnost procesu A, ostatním ne*
  - *Jeden korektní detekuje selhání a také posílá stížnost, pak proces A má f+1 stížností a ostatní 1*
  - *Jen pokud se podaří lavinově stížnost propagovat pro nadpoloviční většinu, je možné lídra změnit – proces A bude mít 3f+1 stížností, ostatní 2f+1 → změní lídra všichni*

# Byzantské kvorum




- Pokud z *N* procesů je *f* nekorektních, byzantských

  byzantské kvorum je skupina více než $\frac{N+f}{2}$ procesů
- Pak dvě quora obsahují alespoň jeden společný korektní proces

  Korektních procesů je $N - f$

  v kvoru je korektních $> \frac{N+f}{2} - f = \frac{N-f}{2}$ (více než polovina)

... a tedy dvě kvora musí mít alespoň jeden společný korektní proces

- Princip: Protože quora o velikosti větší než $\frac{N+f}{2}$ se vždy překrývají v korektním procesu, pokud jedno quorum potvrzuje hodnotu ***v***, nemůže existovat jiné quorum potvrzující odlišnou hodnotu ***v′***.
- **Důvod:**
  Korektní procesy neposílají konfliktní informace.

# Byzantský konzistentní broadcast

- Požadavky:
  - ***Platnost*** *(Validity)*
  - ***Bez zdvojení*** *(No Duplication)*
  - ***Integrita*** *(Integrity)*
  - ***Konzistence*** *(Consistency): Pokud dva korektní procesy doručí zprávy* $m$ *a* $m'$ *pak* $m = m'$

  (na rozdíl od ***shody (Agreement)***, podmínkou není, aby zprávu přijaly všechny korektní procesy)

***Princip***:

1. Vysilatel vysílá zprávu a procesy avizují (zasílají echo) její obdržení ostatním
2. Pokud proces obdrží byzantské quorum se stejnou zprávou, uloží tuto zprávu

# Authenticated Echo Broadcast, algoritmus

Udržuje příznak, zda již na obdrženou zprávu odpověděl, jaké echa obdržel a zdali již na základě ech doručil hodnotu

Broadcast rozesílá zprávu všem ostatním

Při doručení zprávy rozesílá **echo** všem ostatním

Po obdržení echa si toto zaznamená

Pokud pro nějakou zprávu m dostal byzantské quorum ech, zprávu di doručí

**upon event** <bcb, init,bcb> **do**
sentecho:=False; delivered:=False; echos:=[ _ ]

**upon event** <bcb, b'cast | *m*> **do**
**send**([SEND, *m*]) to all processes

**upon event** <al, deliver | [SEND, *m*]> **do**
from sender **s** and sendecho=False
sentecho=True
**trigger** <Send | m> to all processes

**upon event** <al, deliver | [p, Echo, m]> **do**
**if** echos[p]=nill **then** echos[p]:=m

**when** #({echos[p]=m})>$\frac{N+f}{2}$ **and** delivered=False
delivered:=True; deliver(bcb, m)

# Authenticated Echo Broadcast, příklad

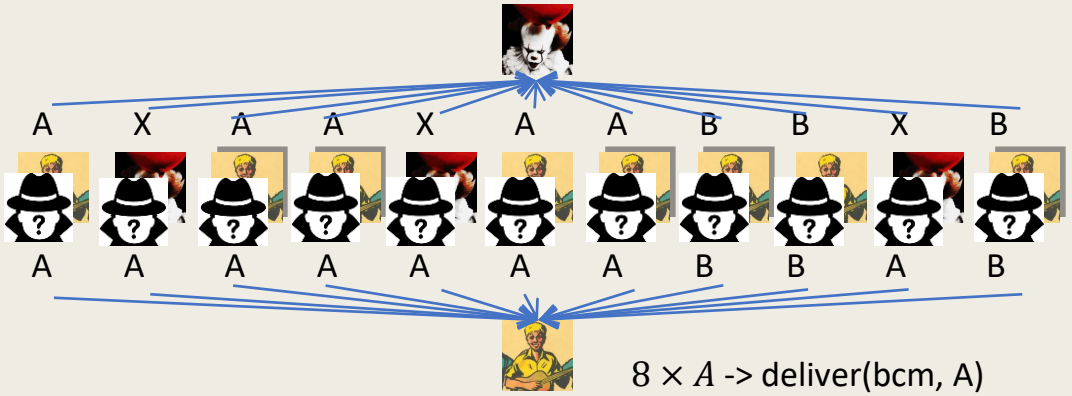
Pro 8 procesů, z nichž tři jsou nekorektní / Byzantské, je byzantské kvórum $> \frac{8+3}{2}$ tedy 8

Byzanstský proces rozeslal zprávu a snaží se zmást korektní procesy (pěti poslal A, třem B).

Proces může uložit pouze A, a to jen pokud Všechny Byzanstké pošlou echo(A). Jinou hodnotu (třeba B) není možné korektnímu procesu k uložení vnutit

Všimněte si, že …

Pokud více než polovina korektních procesů neobdrží stejnou hodnotu, nelze kvorum vytvořit




# Authenticated Echo Broadcast, korektnost

- Korektnost pro $N > 3f$
- **Konzistence** vyplývá z toho, že každé dvě quora mají alespoň jeden společný korektní proces
  - *Pro korektnost může být libovolný poměr korektních a nekorektních procesů*
- **Platnost** ovšem vyžaduje více než dvoutřetinový počet korektních
  - *Pokud by korektní proces vysílal a třetina nebo více nekorektních by neposlala echo, korektní procesy, kterých je tak méně než $\frac{N+f}{2}$ by neuložily*
- Analýza (počet zpráv) $O(N^2)$