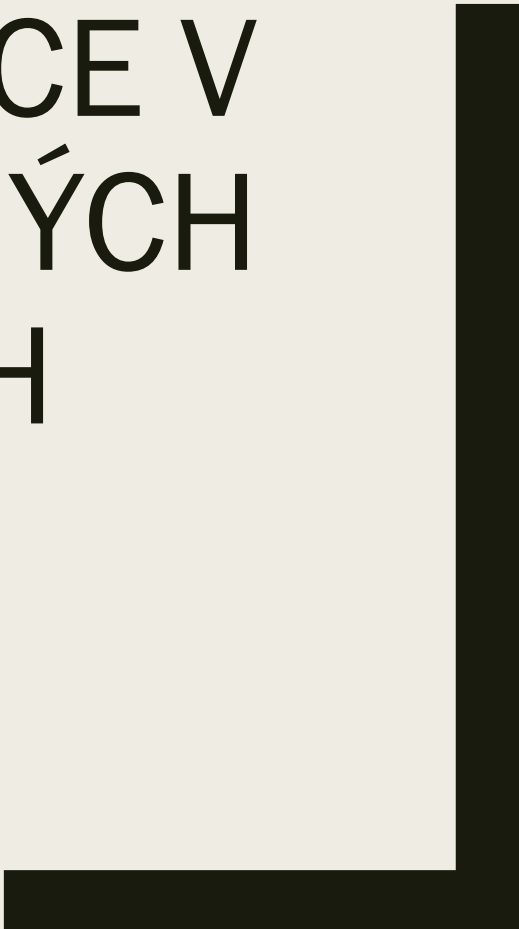


SYNCHRONIZACE V DISTRIBUOVANÝCH SYSTÉMECH

F. Zbořil ml., Petr Hanáček
GTs - 2019



Synchronizace

- **Synchronizace** zaručuje (částečné) uspořádání mezi událostmi
- Pokud máme systémy se sdílenou pamětí, pak můžeme použít nám již známé mechanismy, jako jsou semaforey 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 řídicí uzel*

SYNCHRONIZACE FYZICKÝM A LOGICKÝM ČASEM

FYZICKÝ ČAS
CENTRÁLNÍ PRVEK
LOGICKÉ HODINY

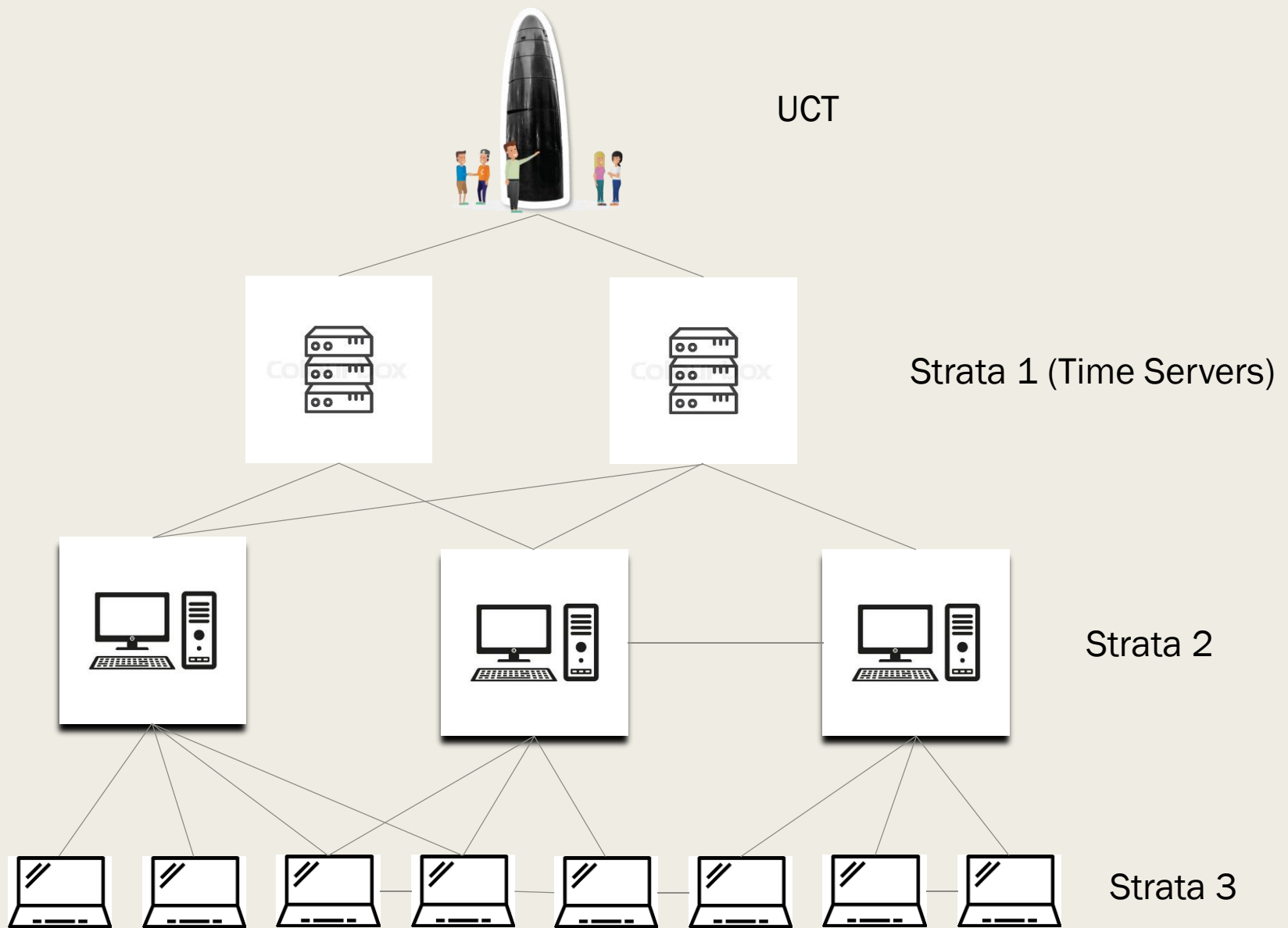
Synchronizace fyzického času, Berkley algoritmus

- 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

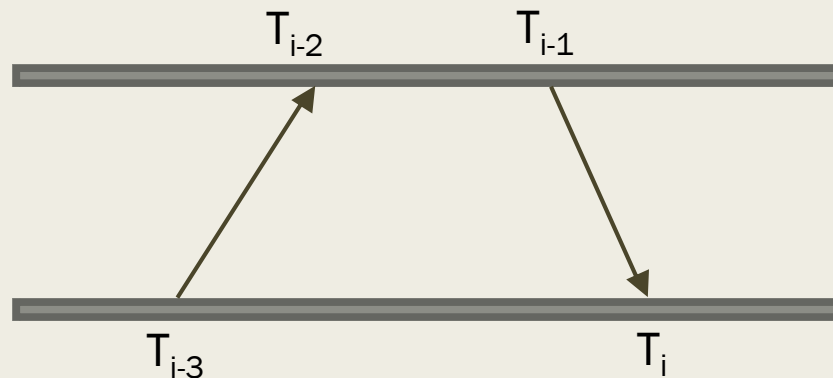


Network Time Protocol (NTP)

- Synchronizace v síti na různých úrovních (strategch)
- 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



NTP – trvání komunikace a zpoždění



- Zpoždění se spočte jako

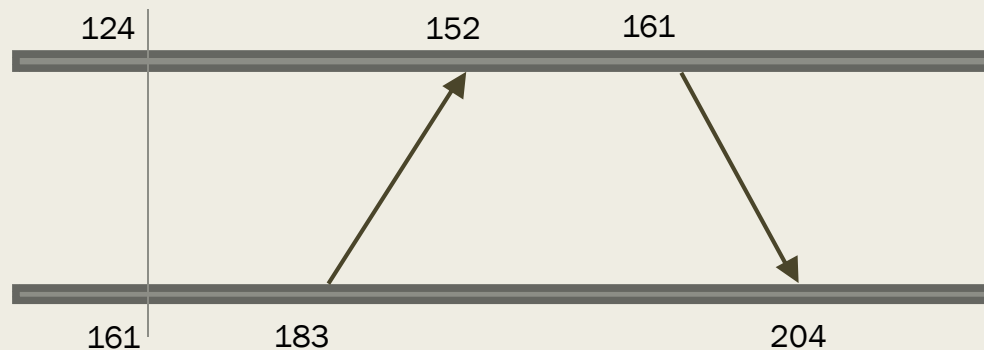
$$d_i = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

- Posunutí (offset) je

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

- 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**

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

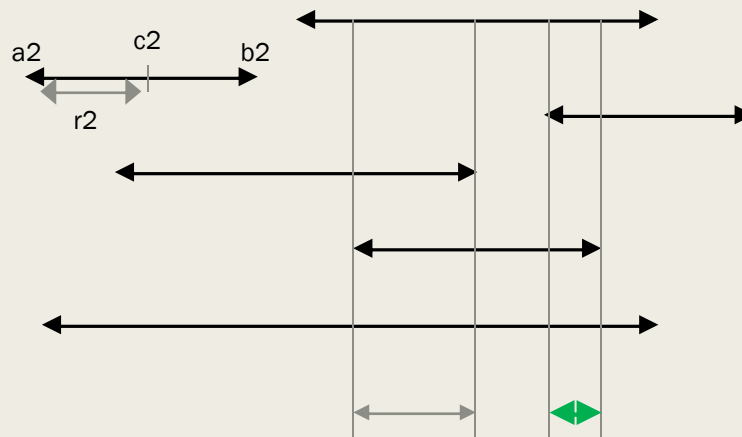
$$d_i = 152 - 183 + 204 - 161 = -31 + 43 = \mathbf{12}$$

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

$$o_i = 1/2(152 - 183 + 161 - 204) = 1/2(-31 - 43) = 1/2(-74) = \mathbf{-37}$$

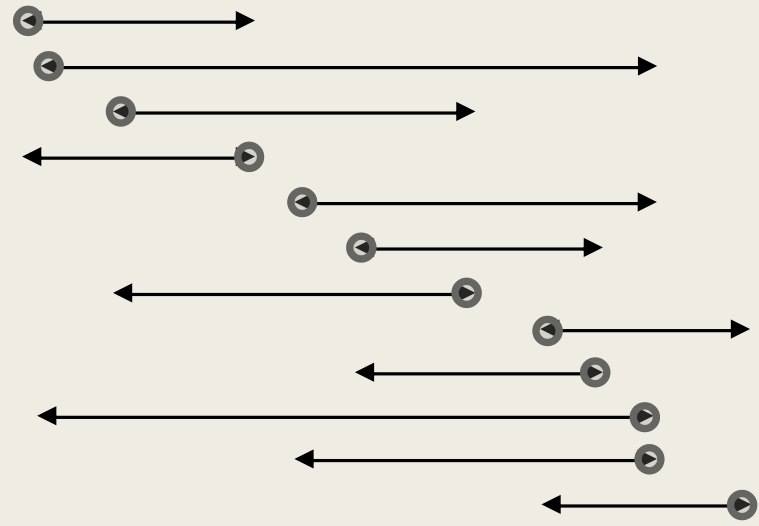
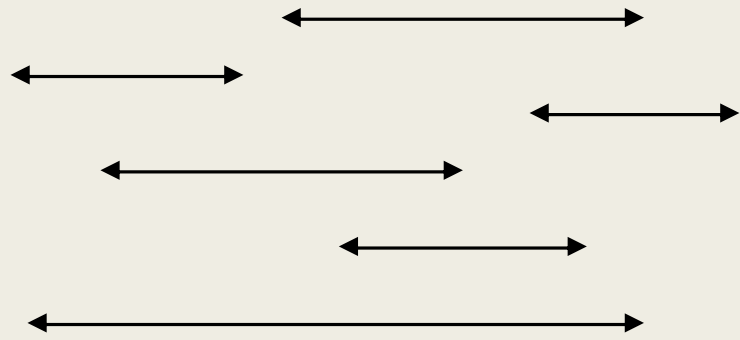
Marzullo-ův algoritmus

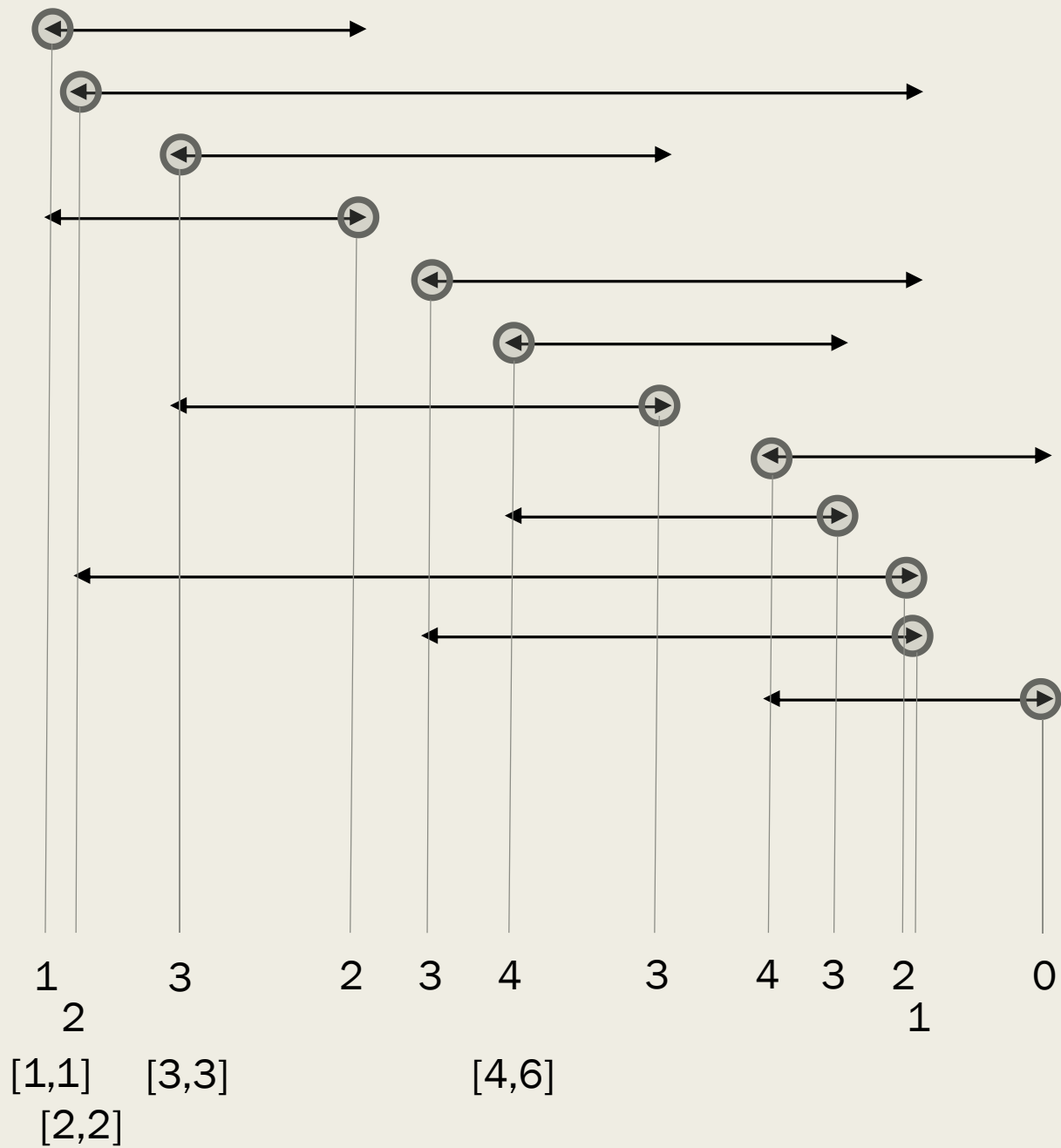
- Pro množinu intervalů hledáme nejmenší interval, který spadá do největšího možného počtu intervalů z této množiny.
- tj. pokud by takové intervaly existovaly pro n intervalů a pro $n+1$ intervalů žádný průnik již nebylo možné nalézt, vezme se ten nejmenší z nich.



Marzullo-ův algoritmus

- Každý interval $\langle a, b \rangle$, resp. $\langle c-r, c+r \rangle$ reprezentují dvě dvojice
 $(a, 1)$, $(b, -1)$ resp. $(c-r, 1)$, $(c+r, -1)$
- 1. Seřad' 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**





VOLBA HLAVNÍHO PROCESU

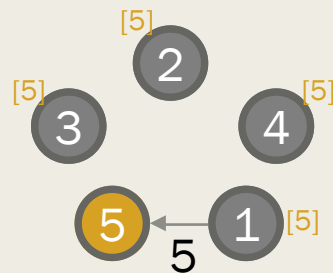
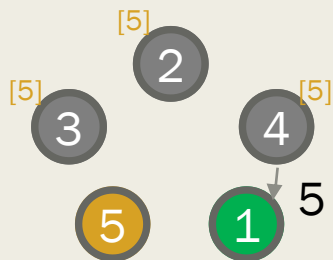
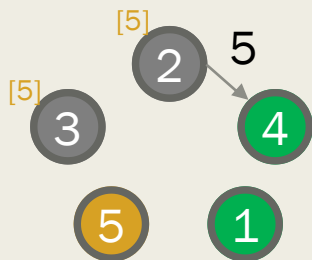
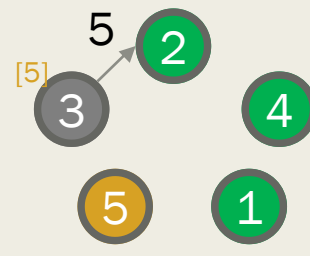
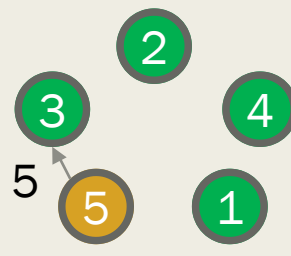
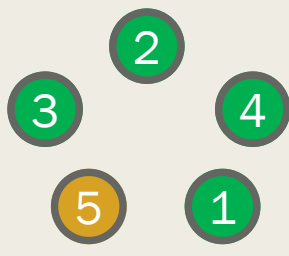
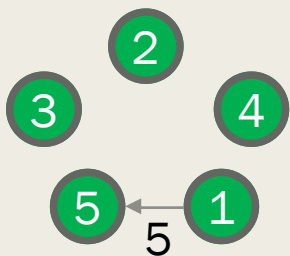
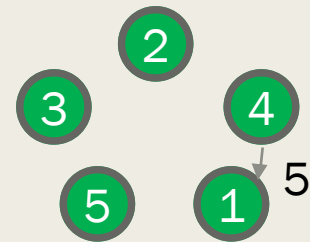
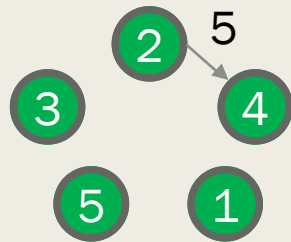
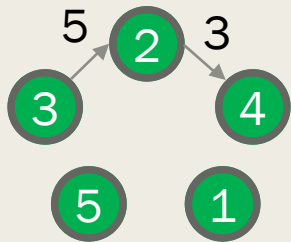
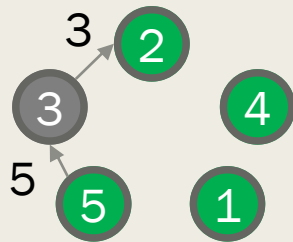
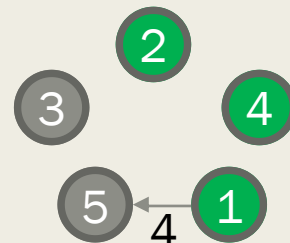
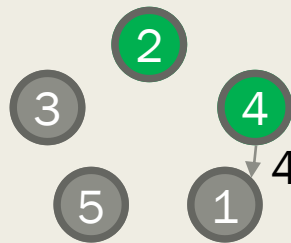
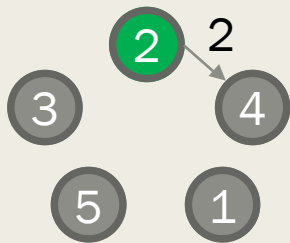
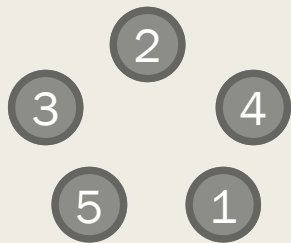


Volba hlavního (master) uzlu, algoritmus Chang a Roberts

- 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 přeposílána 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*



Volba master uzlu, Chang and Roberts, analýza algoritmu

- V nejhorším případě se musí přeposlat $3(n-1)$ zpráv
- Nejhorší případ – uzel s maximálním indexem je první za iniciátorem hlasování

Volba master uzlu pro obecnou topologii

- Ustaví se graf, například metodou **BFS**, od zvoleného uzlu se hledají sousedé jako uzly další úrovně, pokud již nejsou součástí stromu
- Extrakcí největšího UID ze stromu (např dle algoritmu MinExtractionSort)
- Propagace tohoto UID všem uzlům do stromu
- Složitost $O(m+n\log n)$, kde m je počet hran a n je počet uzlů

SYNCHRONIZACE LOGICKÉHO ČASU

LAMPORTOVY HODINY, LAMPORTŮV A RA ALGORITMUS
CHANG A ROBERTSŮV ALGORITMUS VOLBY HLAVNÍHO UZLU
MEAKAWŮV ALGORITMUS
RICCARDSONŮV ALGORITMUS

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 vyloučení*
 - *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*)

$R(e1, e2)$ 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

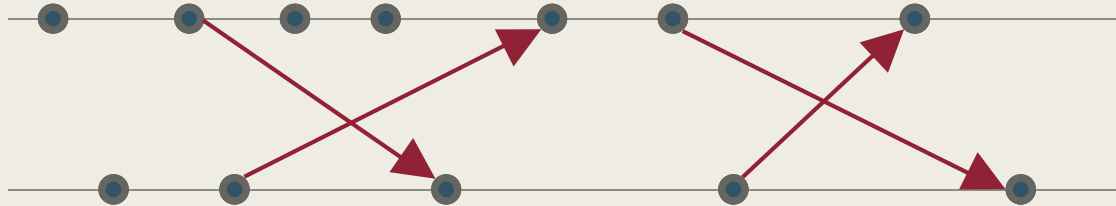
Synchronizace s Lamportovými hodinami

■ Lamportovy logické hodiny

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

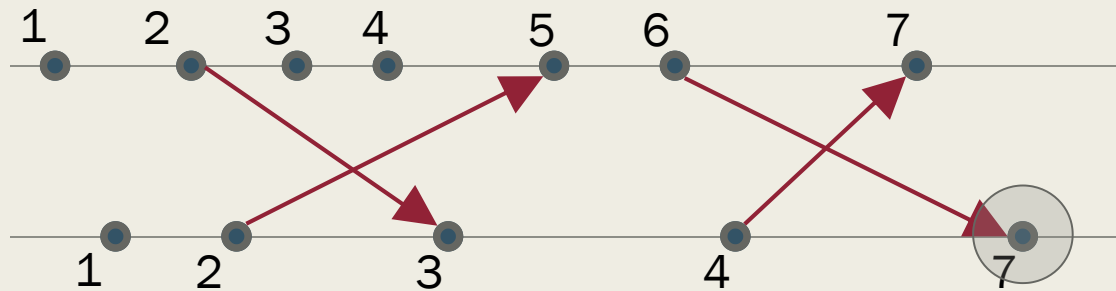
$$a \blacktriangleright b \Rightarrow C(a) < C(b)$$

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 $C_i := \max(C_i + 1, tp + 1)$

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 $C_i := \max(C_i + 1, tp + 1)$

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

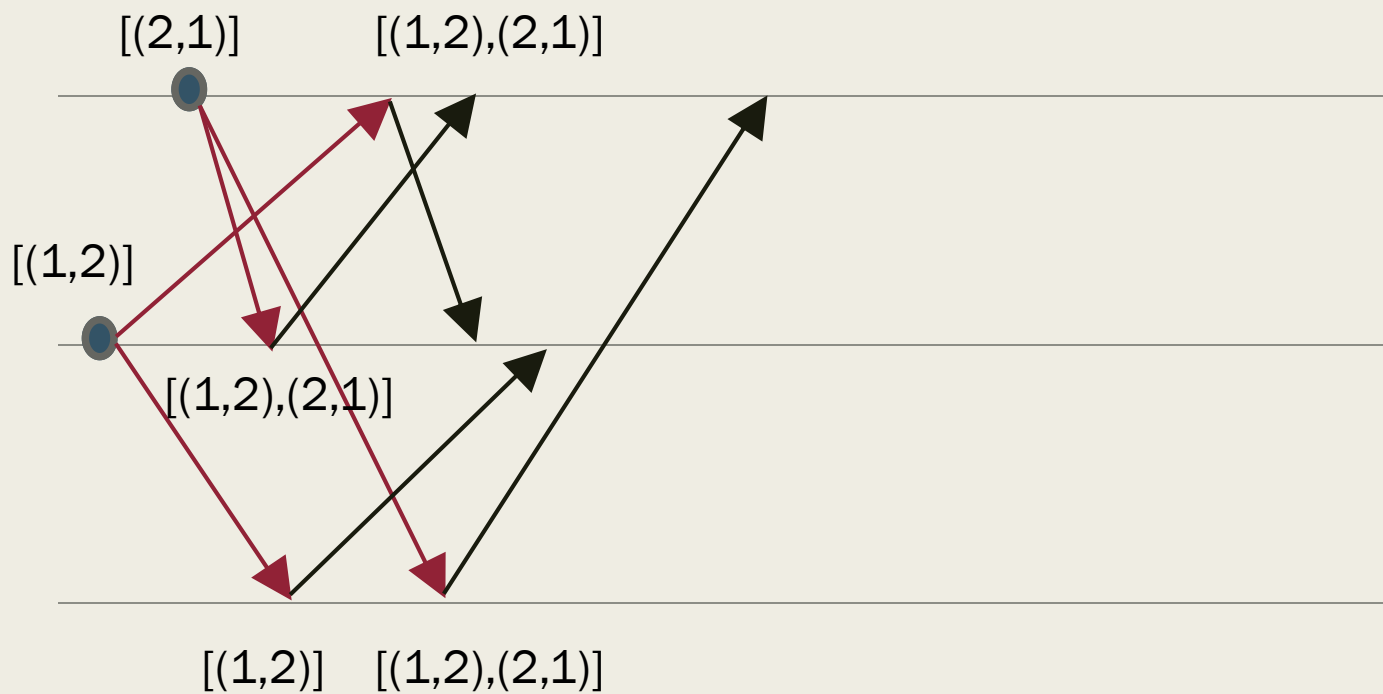
Lamportův algoritmus

- 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

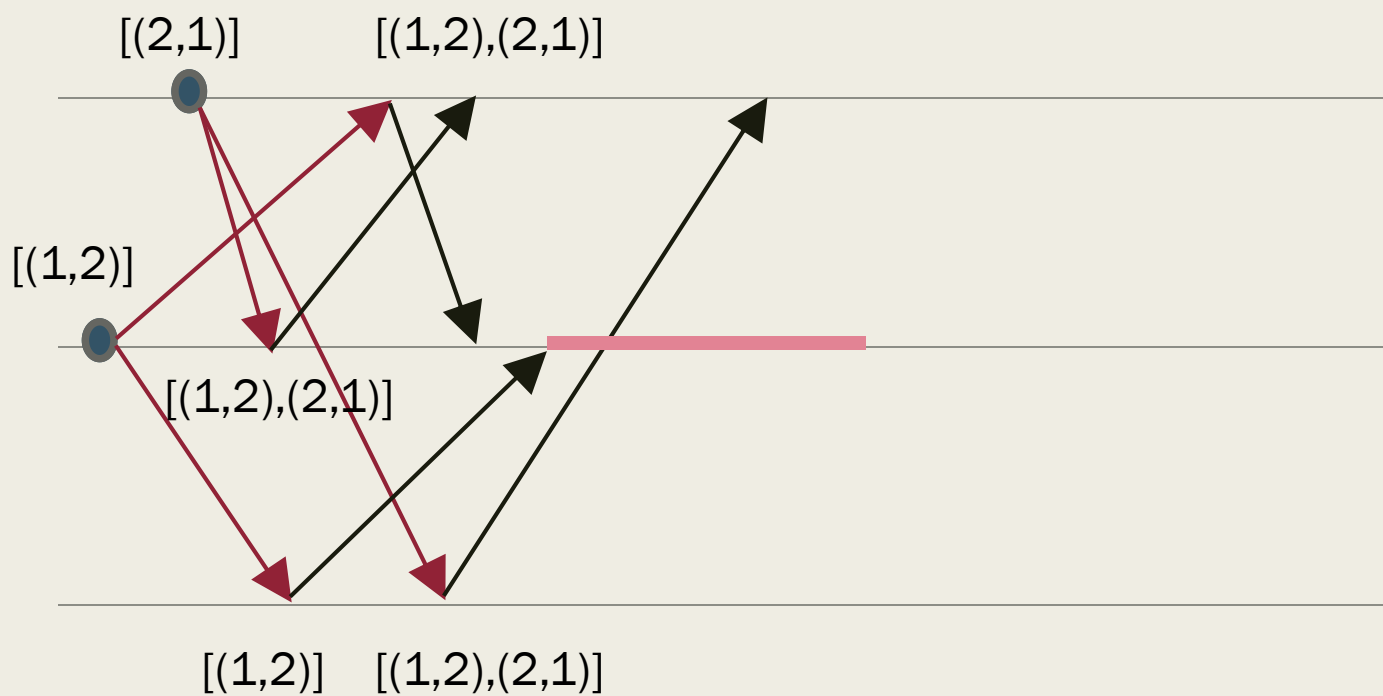
Lamportův algoritmus (požadavek)



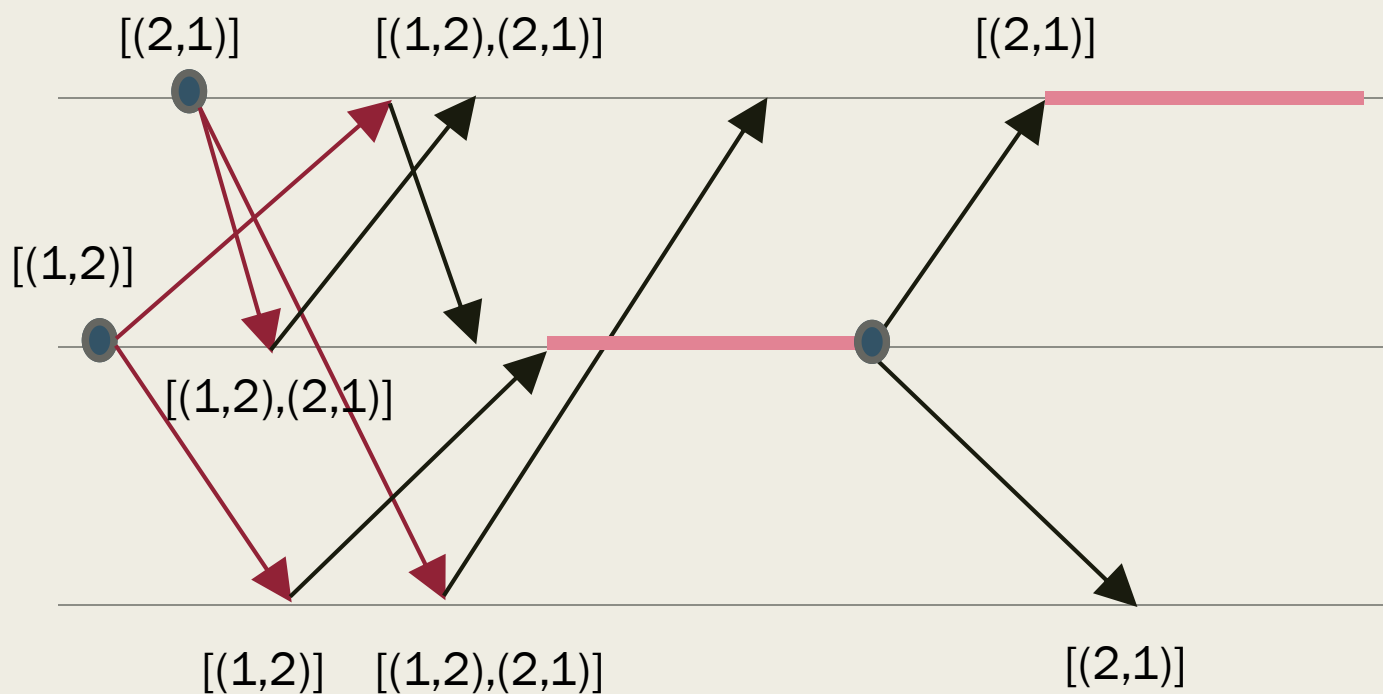
Lamportův algoritmus (odpověď')



Lamportův algoritmus (kritická sekce)



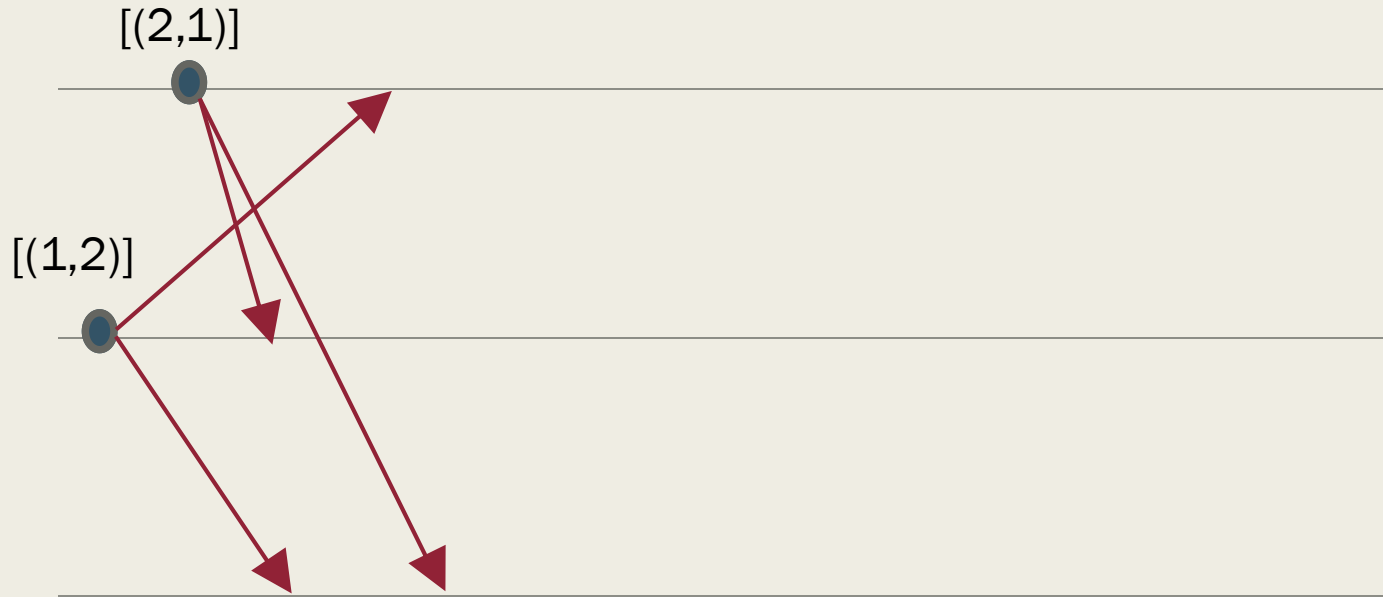
Lamportův algoritmus (uvolnění kritické sekce)



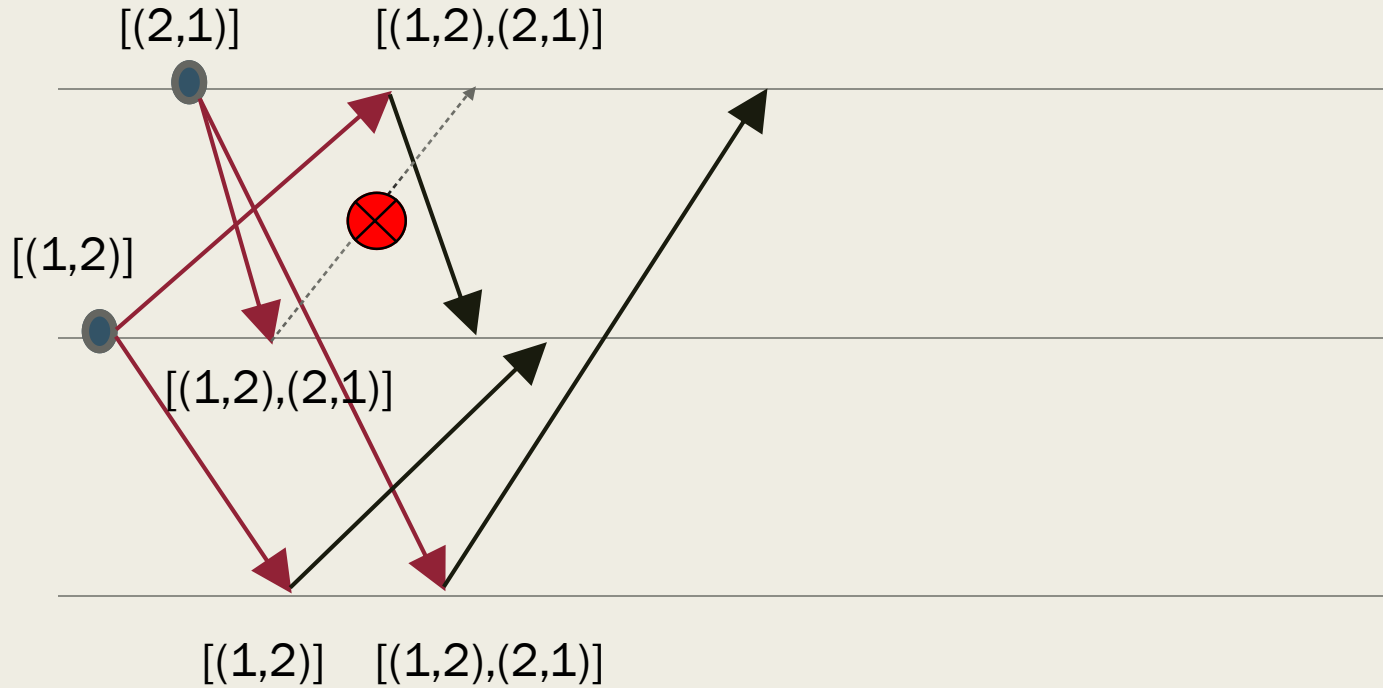
Algoritmus Ricart-Agrawala

- 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 zasláná zpráva*

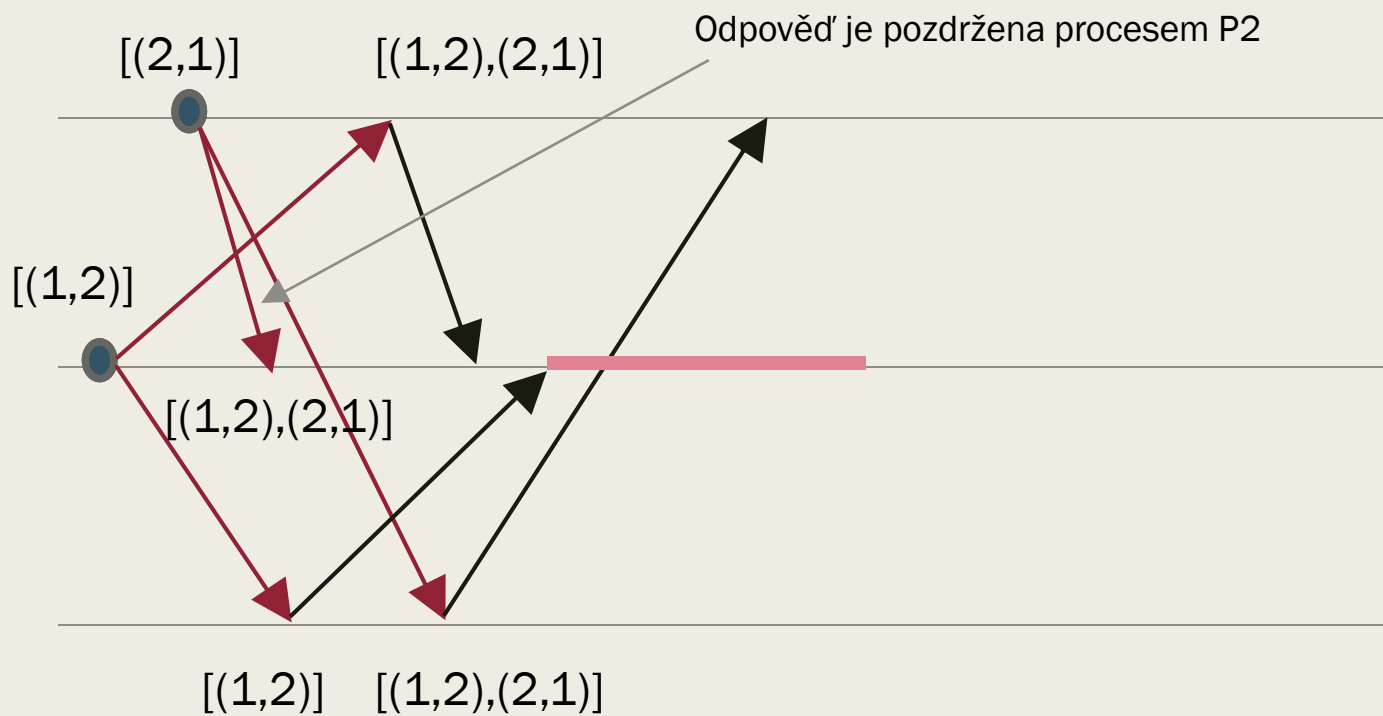
Algoritmus Ricart-Agrawala (požadavek)



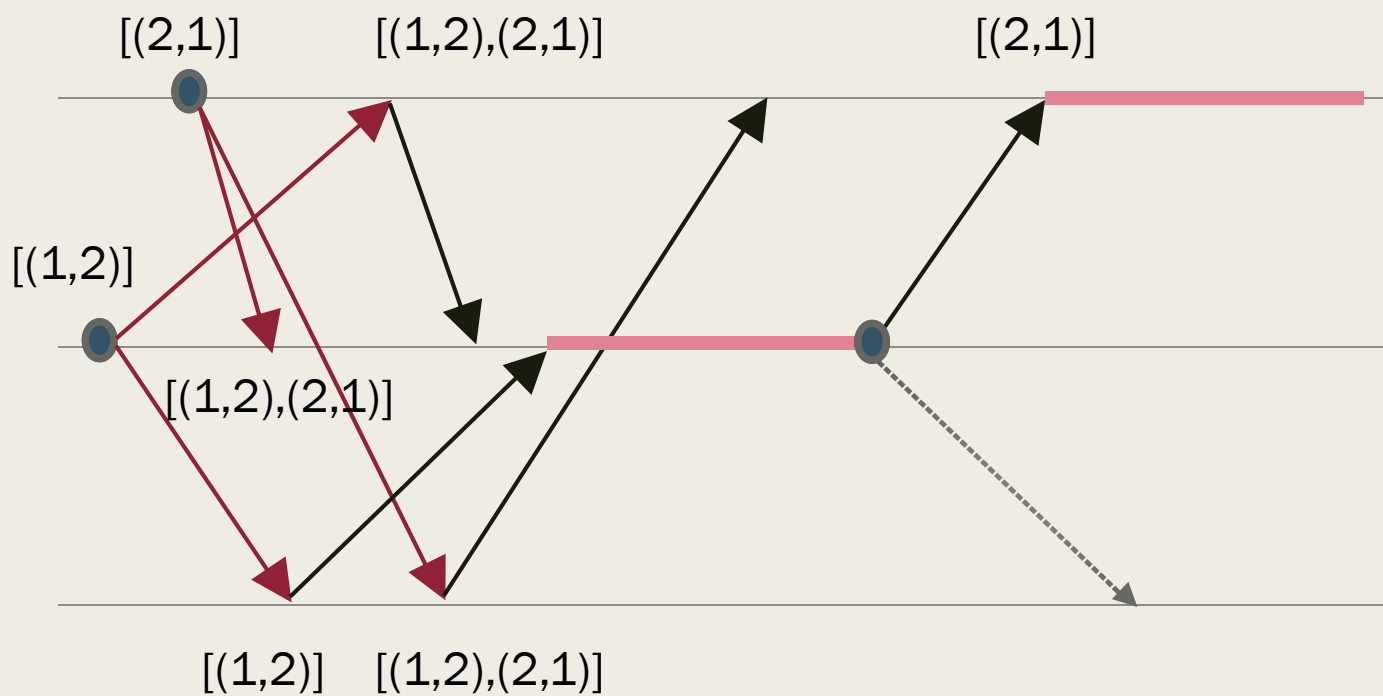
Algoritmus Ricart-Agrawala (odpověď)



Lamportův algoritmus (kritická sekce)



Lamportův algoritmus (uvolnění kritické sekce)

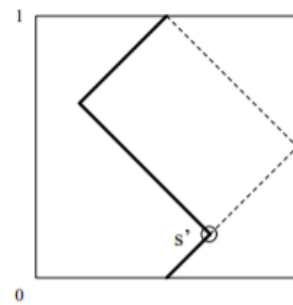
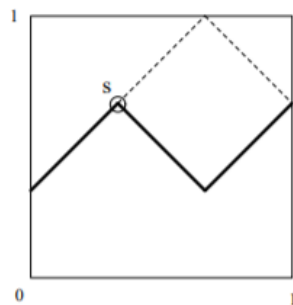
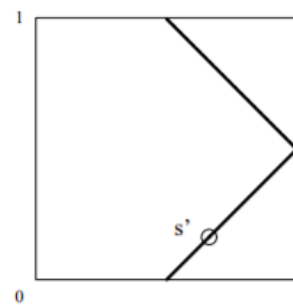
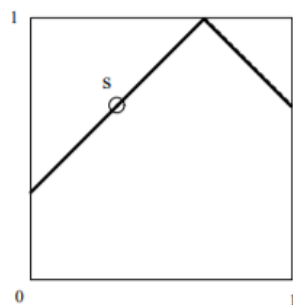


Meakawův algoritmus

- Povolení je třeba jen od podmnožiny procesů
- Každý proces $1 \leq i \leq N$ si udržuje seznam procesů R_i tak, že
 - *Každé dva procesy sdílí nějaký proces, tedy $R_i \cap R_j \neq \{\}$*
 - *Každý proces i je v R_i*
 - *Velikost těchto množin je K , $K = \sqrt{N - 1}$*
 - *Každý proces je přítomen celkem K -krát v seznamech všech procesů*

Meakawův algoritmus, vytvoření quor

- Biliardová metoda pro vytváření quor
- Hrajeme šťouch vždy v tom samém směru, například směrem vpravo 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		<u>16</u>		17	
18		19		20		21
	22		23		24	

Zalomený billiard pro 24 uzlů

- V místě odpovídajícímu číslu procesu, pro který hledáme množinu, 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		<u>16</u>		17	
18		19		20		21
	22		23		24	

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

Zalomený billiard pro 24 uzlů

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

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

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

Meakawův 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 R_i
 - Pokud obdrží **grant(j)** od všech j z R_i
 - Po opuštění kritické sekce pošle **release(i)** všem z R_i
- 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, pošle **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.

Meakawův 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 původnímu)
- Problém - může nastat uvážnutí

Meakawův 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)

Meakawův algoritmus pt.1

- Žádající proces
 - Pokud chce proces i vstoupit do kritické sekce ve stavu svých logických hodin ts , pošle **request(ts, i)** všem procesům z R_i
 - Pokud obdrží **grant(j)** od všech j z R_i
 - Po opuštění kritické sekce pošle **release(i)** všem z R_i
- 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
 - pokud vydal grant procesu s nižší prioritou, pak
 - 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

Meakawův algoritmus pt.2

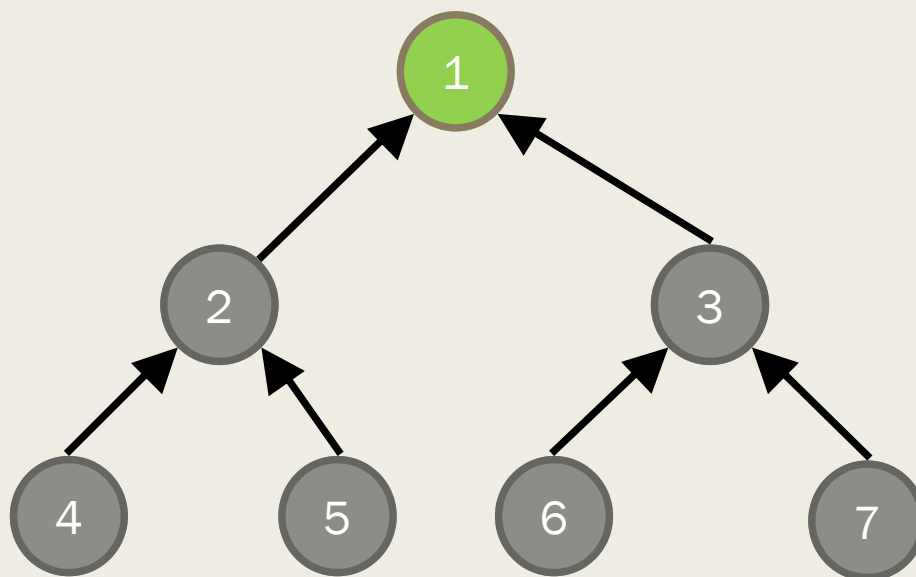
■ Spolupracující proces

- 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 R_i , 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.

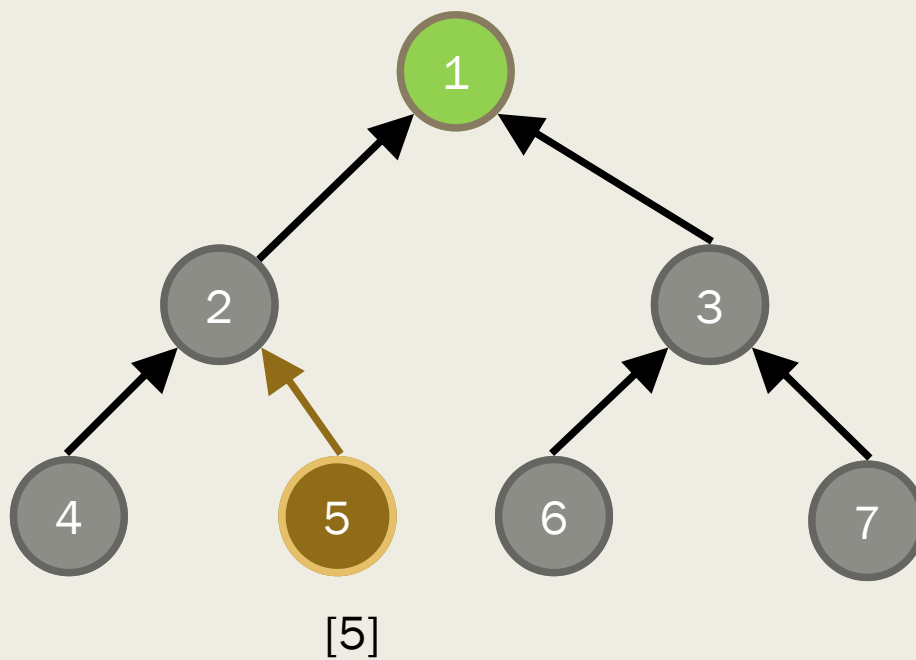
Raymondův algoritmus

- Jeden token reprezentuje v systému jednu 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)
- Vlastnosti
 - *Nezpůsobuje vyhladovění*
 - *Nezpůsobuje uváznutí*
 - *Fault tolerance*

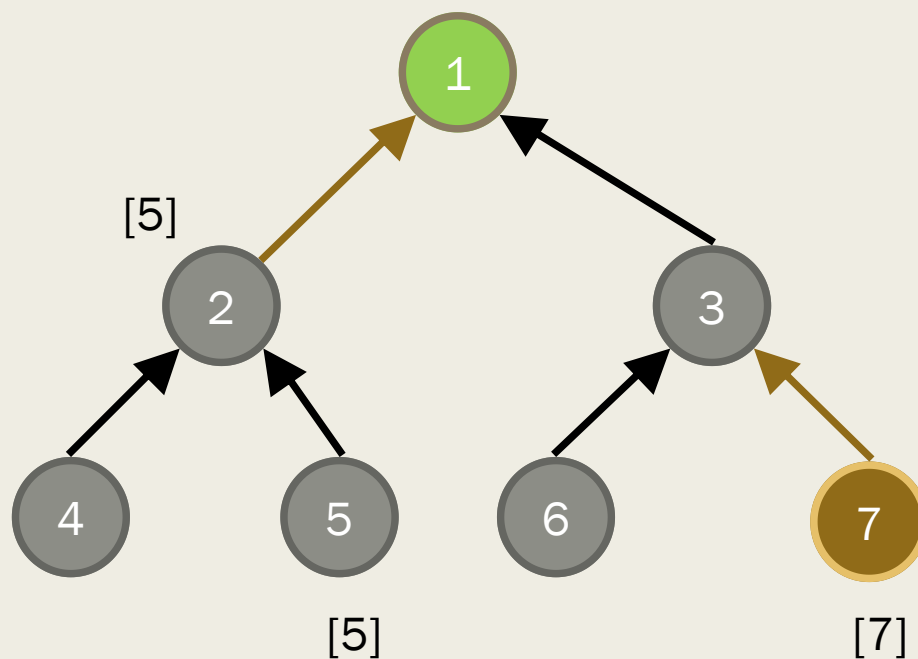
Raymondův algoritmus



Raymondův algoritmus

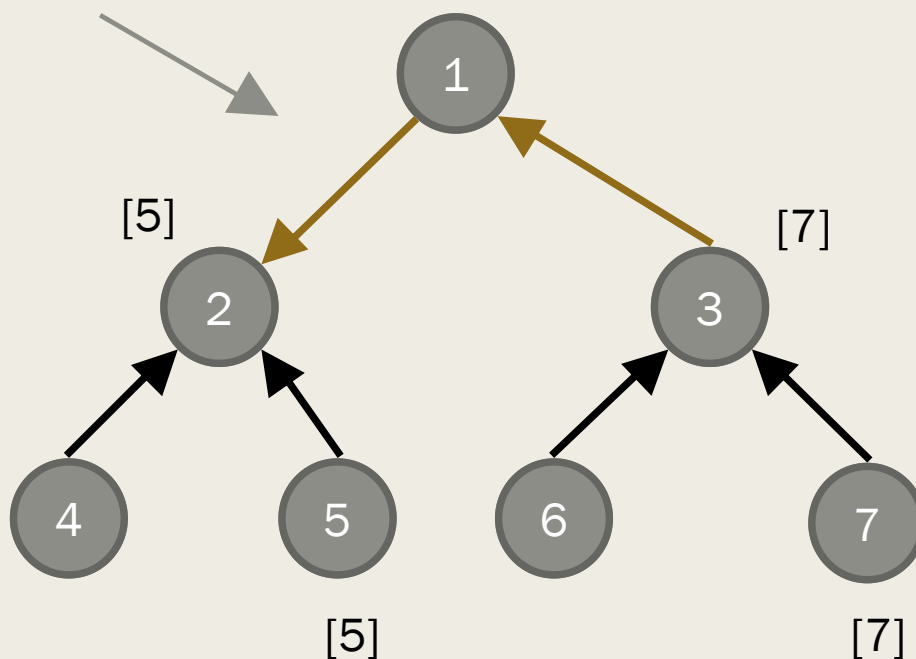


Raymondův algoritmus

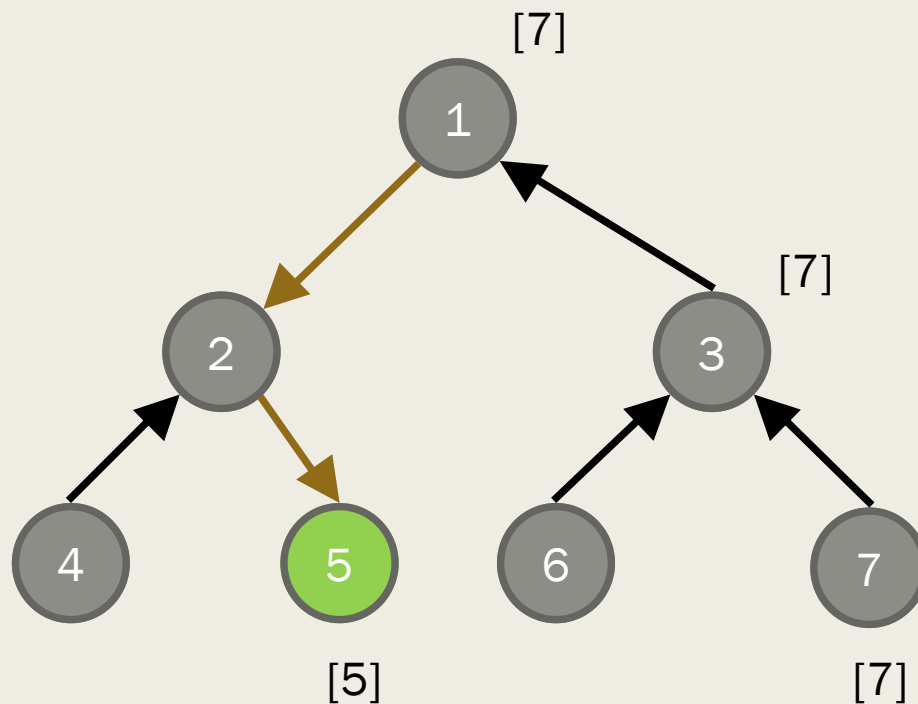


Raymondův algoritmus

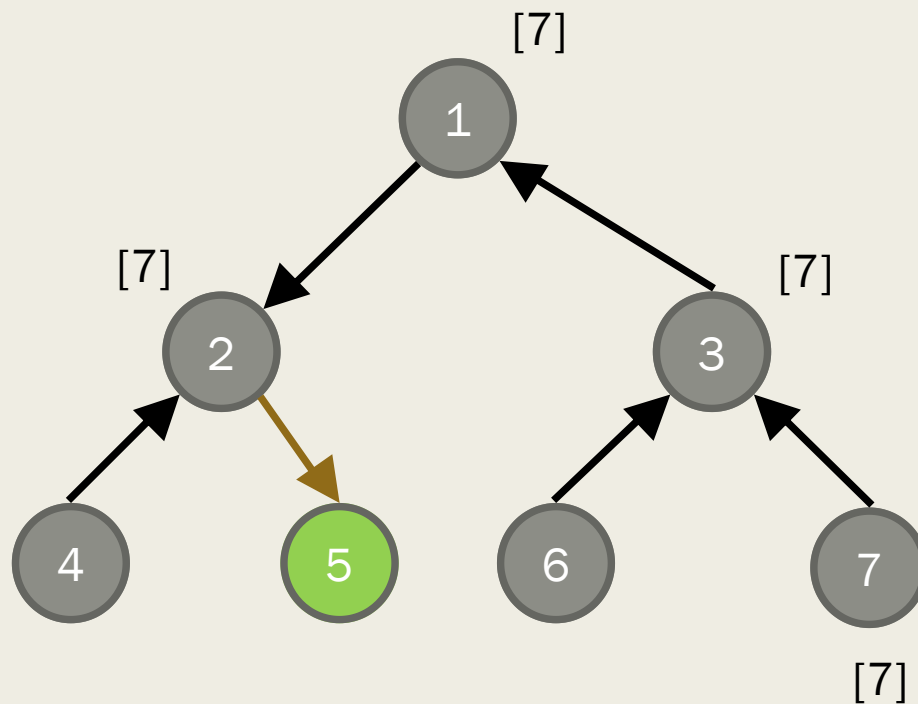
Následování zapamatovaného směru



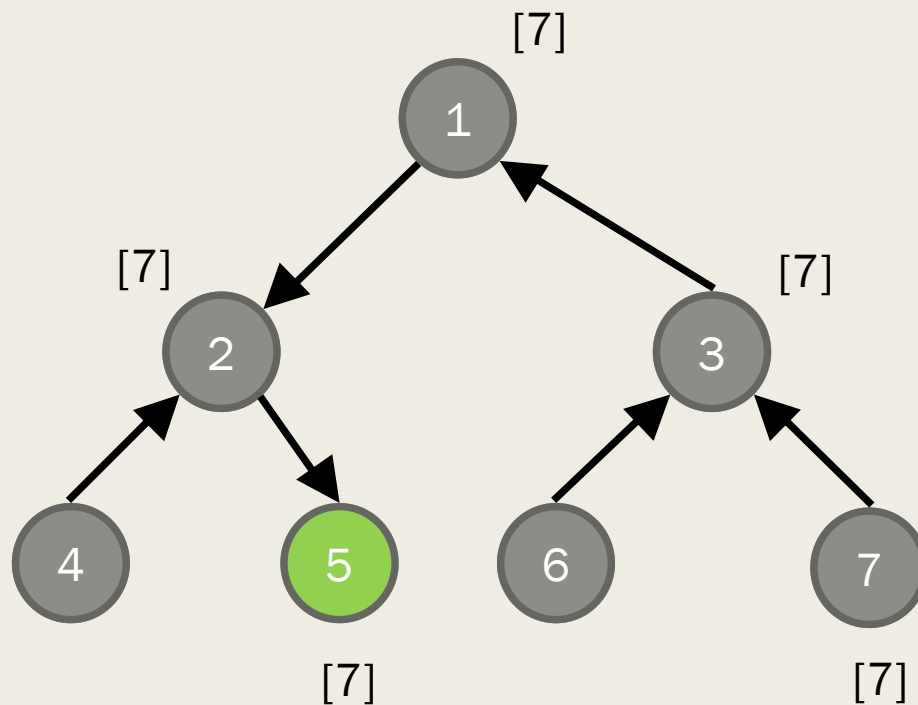
Raymondův algoritmus



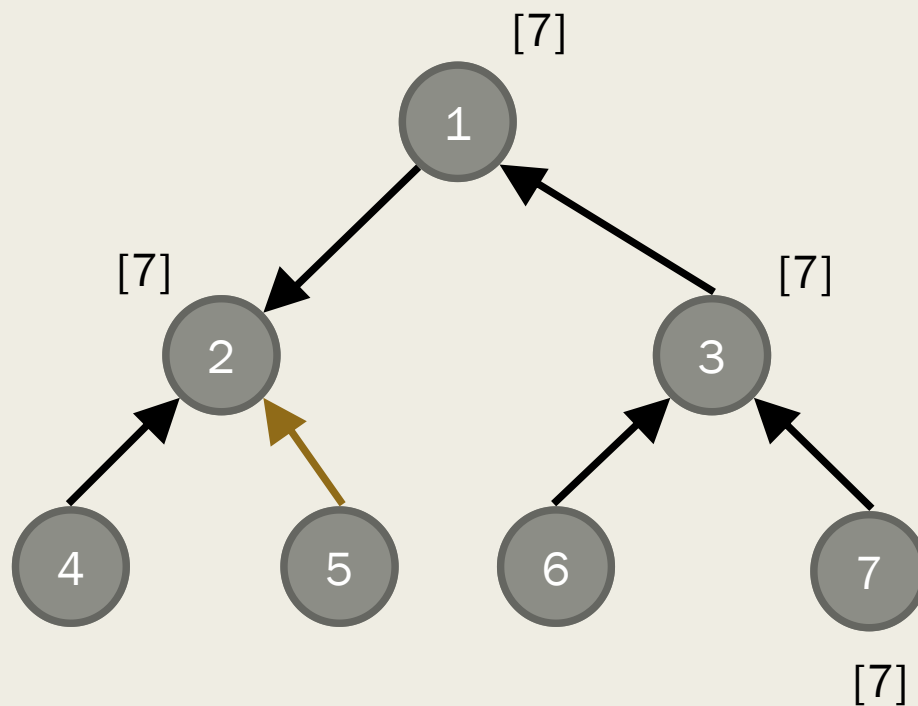
Raymondův algoritmus



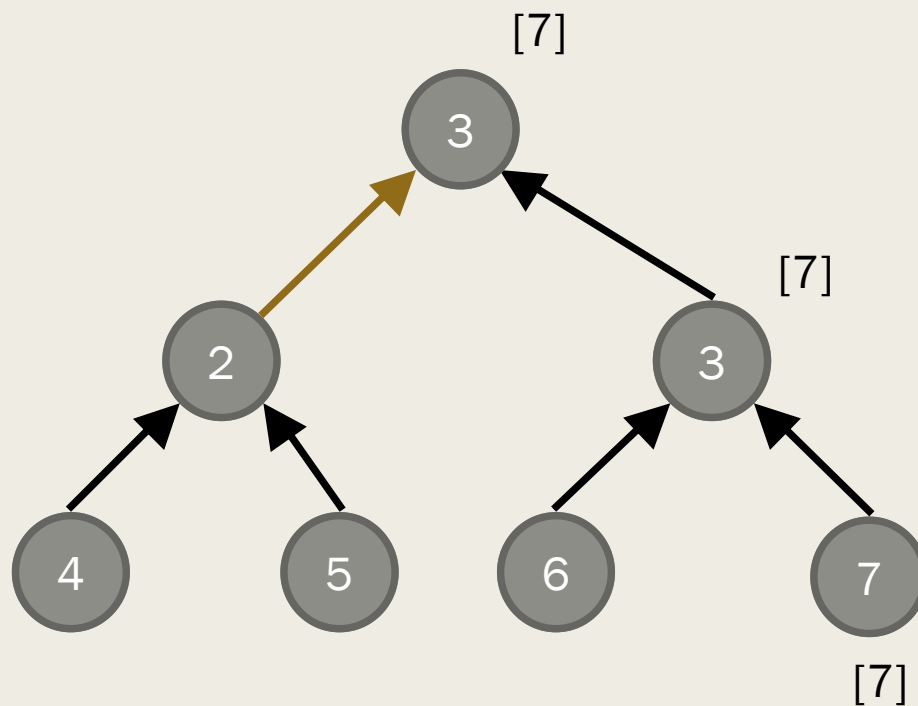
Raymondův algoritmus



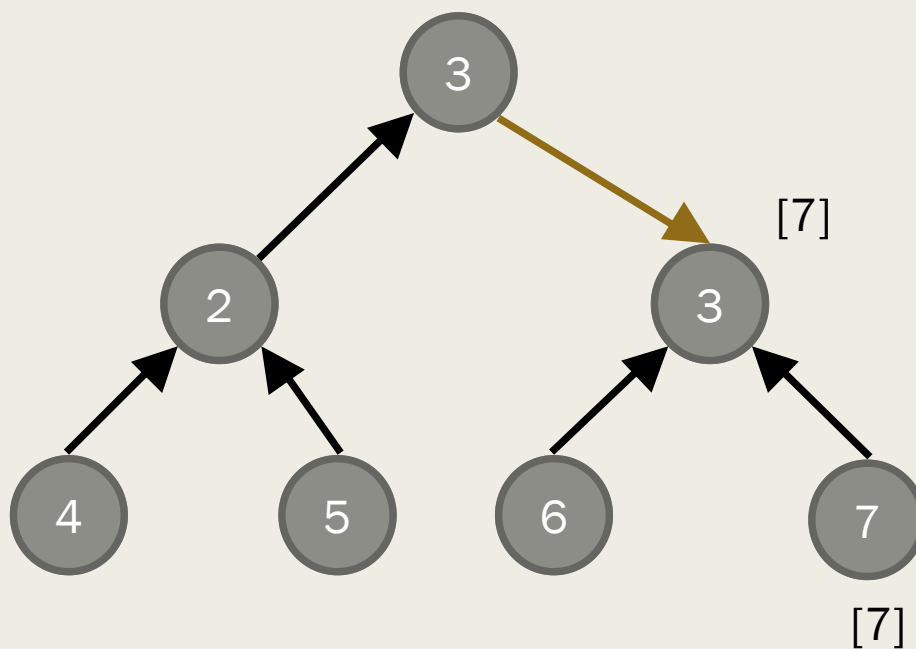
Raymondův algoritmus



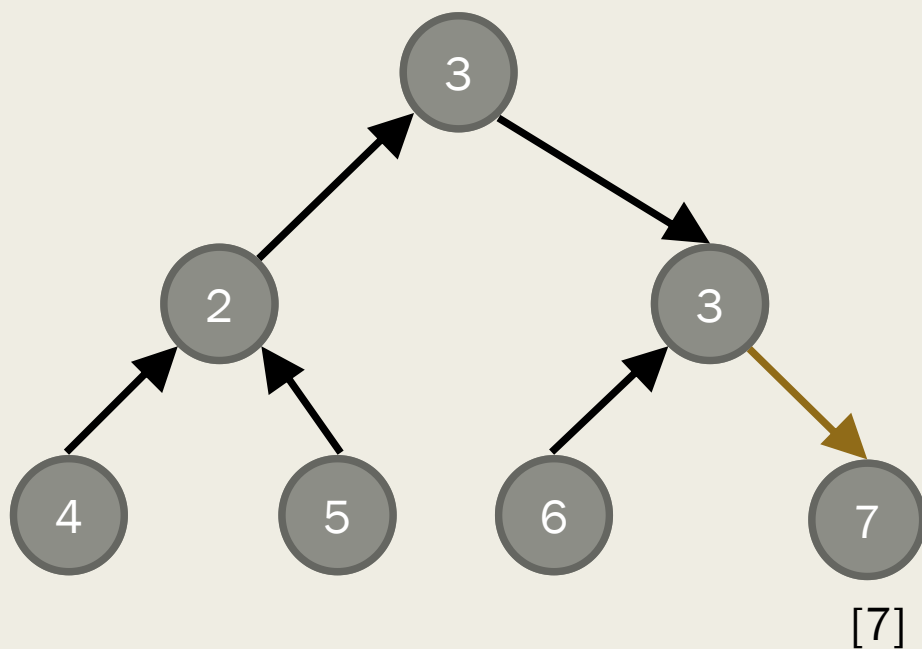
Raymondův algoritmus



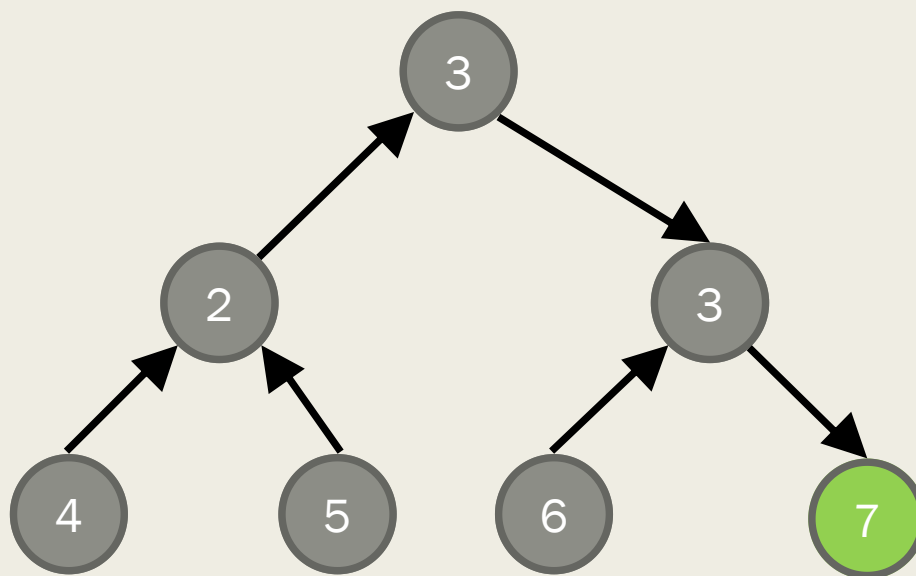
Raymondův algoritmus



Raymondův algoritmus



Raymondův algoritmus



Analýza Raymondova algoritmu

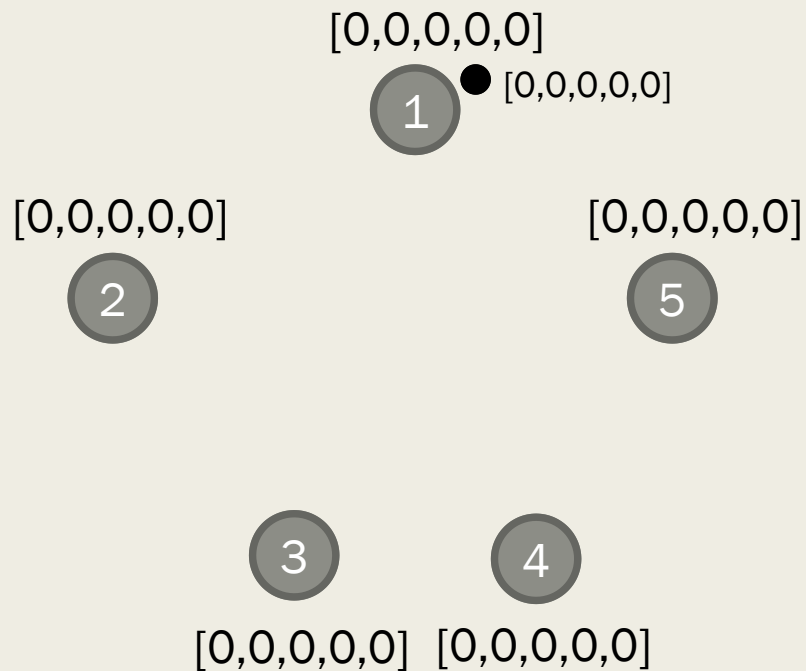
- Počet zaslaných zpráv – třída složitosti $O(\lg N)$
- Průměrné synchronizační zpoždění je $\lg N/2$
- Přenosnost se snižuje při zahlcení sítě zprávami
- Může použít greedy strategii - možnost hladovění

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 vyšší 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

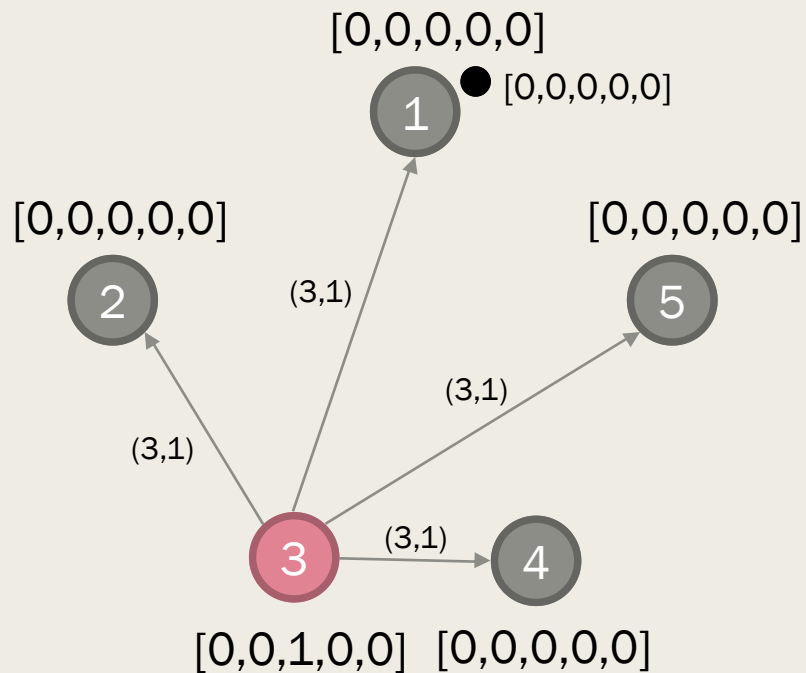
Suzuki – Kasami vysílací algoritmus

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

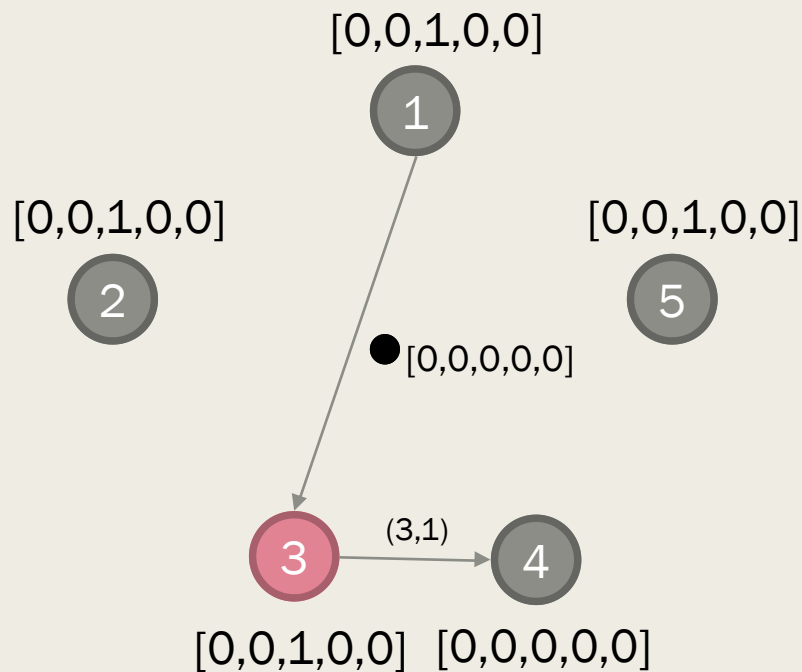


Suzuki – Kasami vysílací algoritmus

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

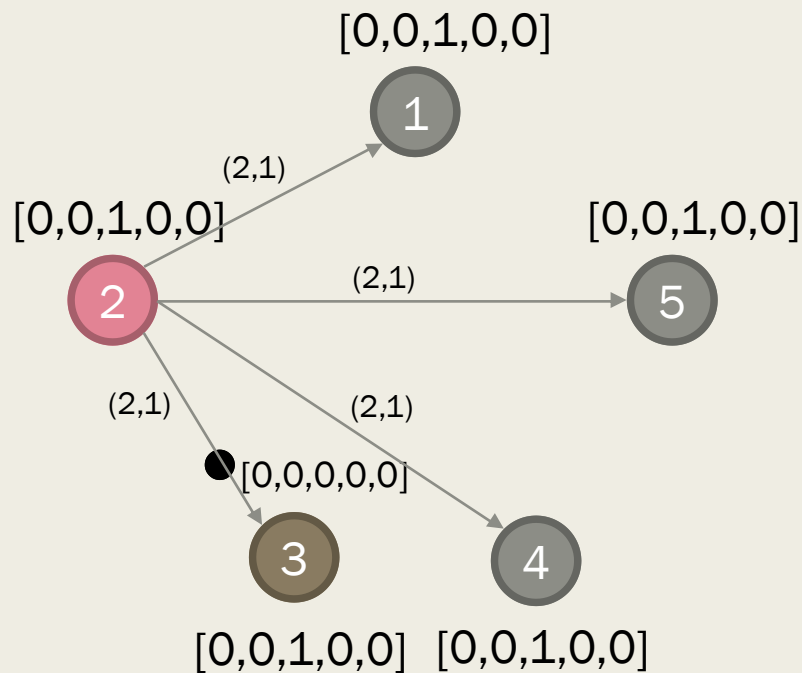


Suzuki – Kasami vysílací algoritmus



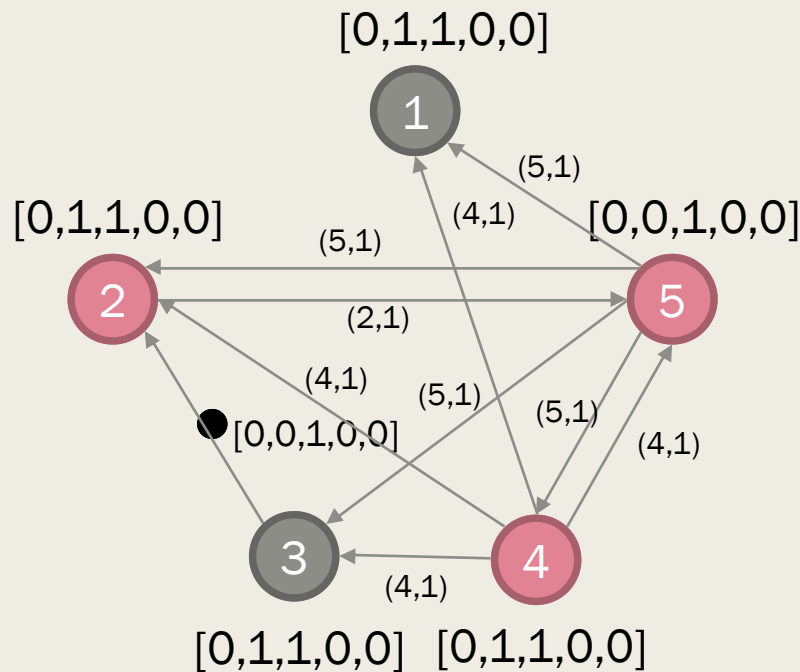
Procesy 1, 2 a 5 zprávu zpracují.
Proces jedna drží token,
nepotřebuje jej a jelikož je třetí
proces jediným, jehož hodnota v
čítači je větší, než hodnota v čítači
tokenu, zašle jej tomuto procesu

Suzuki – Kasami vysílací algoritmus



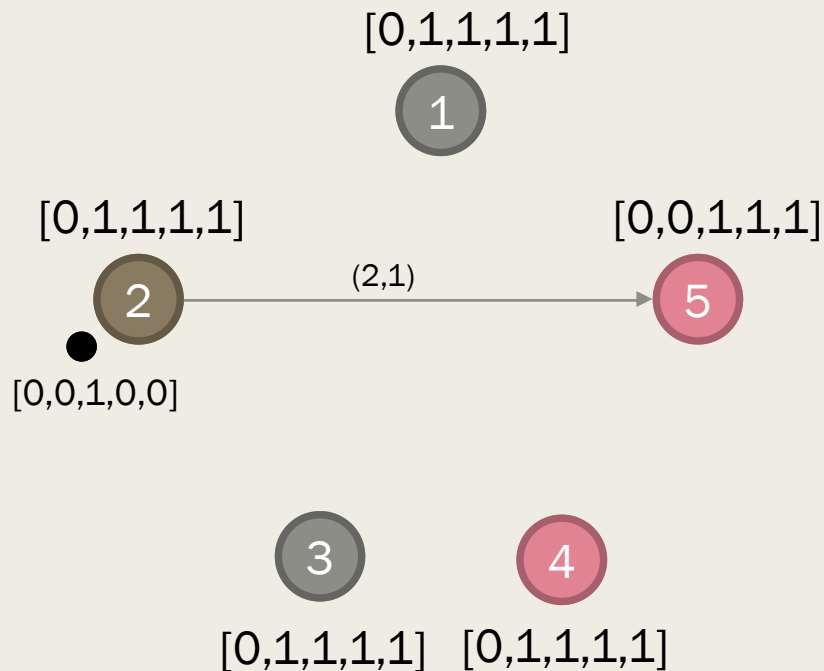
Proces 3 se nachází v kritické sekci. Proces 4 také zpracoval zprávu a upravil si patřičně svůj čítač. V ten samý okamžik žádá o token proces 2. Vyšle všem zprávu, že žádá poprvé a je odložen, čekajíc na token.

Suzuki – Kasami vysílací algoritmus



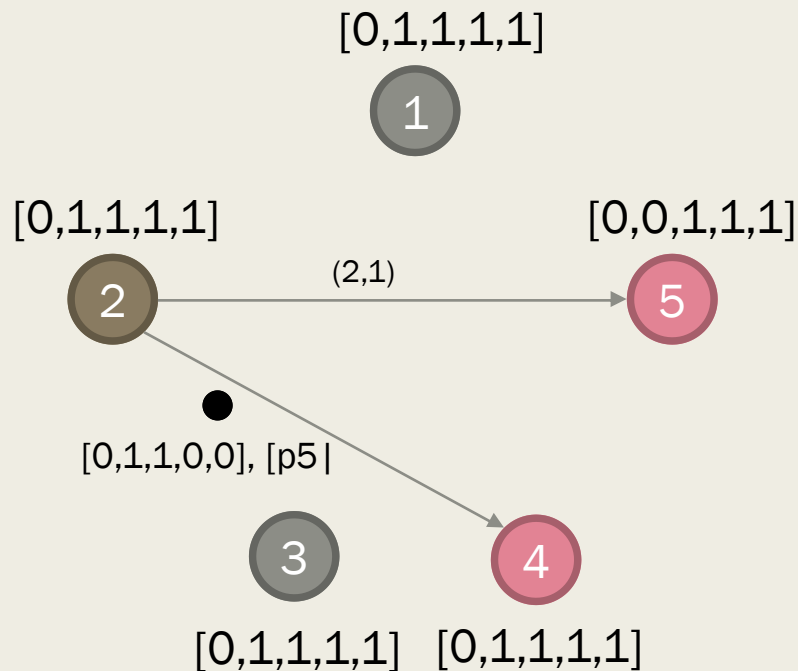
Všechny procesy až na pátý si zpracují žádost od druhého procesu, upraví své čítače a jelikož proces 3 dokončil svoji činnost v kritické sekci, posílá token jedinému procesu, o kterém ví, že jeho počet žádostí je vyšší, než je v tokenu uložený jeho počet přidělení tomuto procesu. Zároveň ale žádají procesy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

Suzuki – Kasami vysílací algoritmus



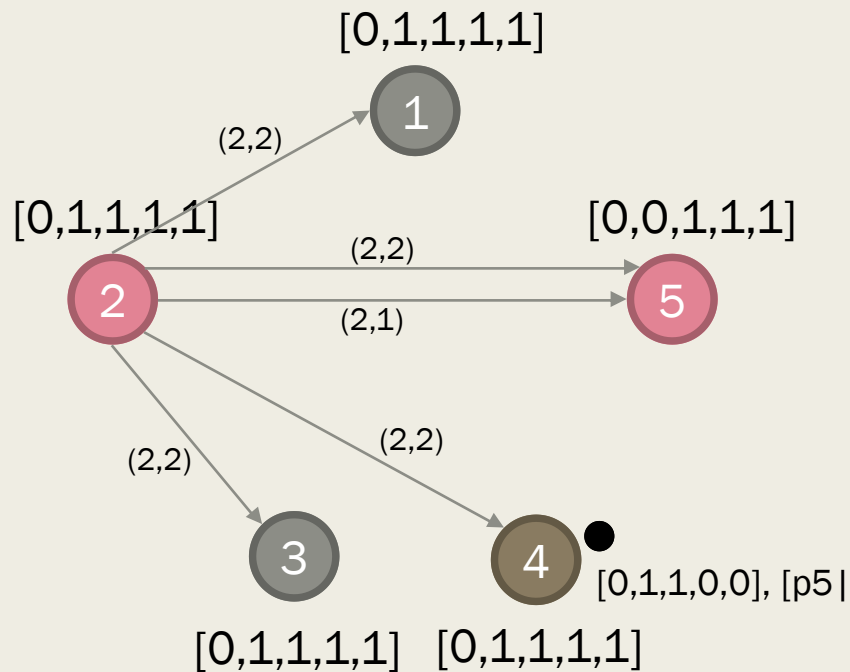
Proces 2 si užívá kritické sekce, procesy 4 a 5 čekají na token a zpráva uzlu pět od uzlu nebyla stále doručena, přestože již žádající uzel token má!

Suzuki – Kasami vysílací algoritmus



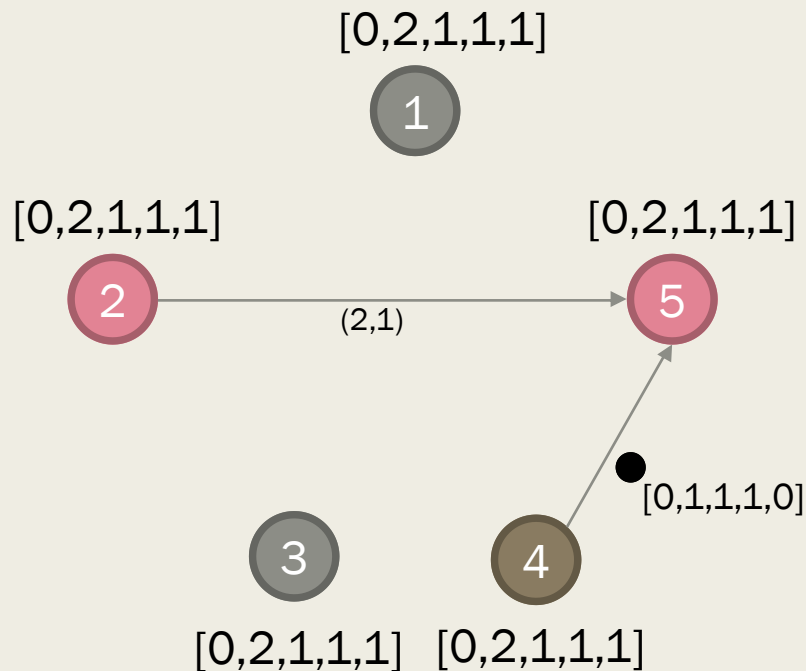
Stále máme jednu starou bloudící zprávu v systému. Proces dva dokončil svoji činnost v kritické sekci a zjistil, že hned dva uzly čekají na token, protože v čítači vidí, že mají vyšší počet žádostí, než je počet přidělení v záznamech tokenu. Vybere jeden, například čtvrtý, tomu pošle token a pátý proces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

Suzuki – Kasami vysílací algoritmus



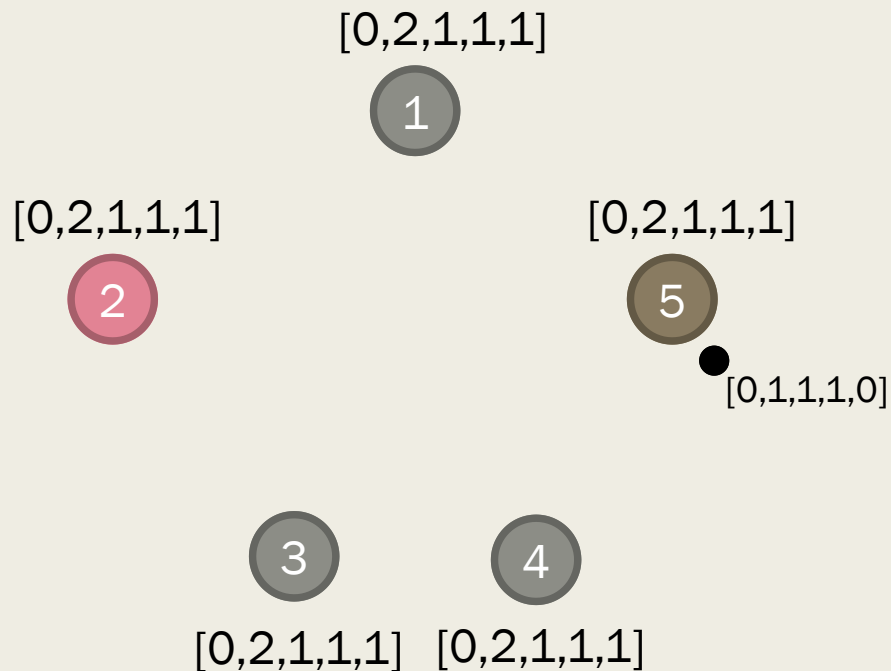
Proces 4 je v kritické sekci a proces 2 znovu žádá o vstup do této sekce. Pošle patřičné zprávy.

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

Nekorektní procesy

- Nekorektní procesy jsou takové, které mohou po spuštění selhat
- Abstrakce nekorektních procesů:
 - ***Crash & stop*** – po selhání procesu neprovádí žádnou činnost (zastaví se)
 - ***Crash & recovery*** – po nějaké době může dojít k obnovení jejich správné činnosti
 - ***Byzantine*** – byzantské procesy, pro které není chování po selhání definováno

Detekce selhání, crash & stop

```
upon event <P,init> do
```

```
    active:= $\Pi$ ;
```

```
    detected:=0;
```

```
    start_timer();
```

```
upon event (Timeout) do
```

```
    forall  $p \in \Pi$ 
```

```
        if( $p \notin alive$ )  $\wedge$  ( $p \notin detected$ ) then
```

```
             $detected = detected \cup \{p\}$ ;
```

```
            trigger<P,Crash | p>;
```

```
            trigger<send | p,[HeartbeatRequest]>;
```

```
            alive:=0;
```

```
            start_timer();
```

```
upon event <Deliver, q,[ HeartbeatRequest]> do
```

```
    trigger <Send, q,[HeartbeatReply]>
```

```
upon event <Deliver, p,[ HeartbeatReply]> do
```

```
     $active := active \cup \{p\}$ 
```

Detekce selhání, procesy crash & recovery

```
upon event < $\Diamond$ P,init> do
    active:= $\Pi$ ; suspected:=0;
    delay:= $\Delta$ ;
    start_timer(delay);

upon event (Timeout) do
    if(alive  $\cap$  suspected  $\neq$  0) then
        delay:=delay+  $\Delta$ ;
    forall  $p \in \Pi$ 
        if( $p \notin$  alive)  $\wedge$  ( $p \notin$  suspected) then
            suspected= suspected  $\cup$  { $p$ };
            trigger<  $\Diamond$ P, Suspect |  $p$ >;
        else if( $p \in$  alive)  $\wedge$  ( $p \in$  suspected) then
            suspected= suspected - { $p$ };
            trigger<  $\Diamond$ 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  $\cup$  { $p$ }
```

VŠESMĚROVÉ VYSÍLÁNÍ



Předpoklady pro komunikaci při všesměrovém vysílání

■ Komunikace

- *Známe největší možné zpoždění při doručování zprávy*
- *Lokální hodiny pro každý z procesů*
- *Známy nejvyšší časový limit pro vykonání interní akce*
- *by a process to execute a step*

■ Topologie

- *Topologie sítě je obecná*

Všesměrové vysílání, n-tity

m : zpráva z množiny zpráv msgs

operace **b'cast**(m), **deliver**(m)

Každá zpráva obsahuje následující položky:

- **sender**(m), identita odesilatele
- **seq**(m), sekvenční číslo
- **sender**(m) = p and **seq**(m) = i značí, že zpráva m je i -tou zprávou zaslanou procesem p

Připomeňme si ...

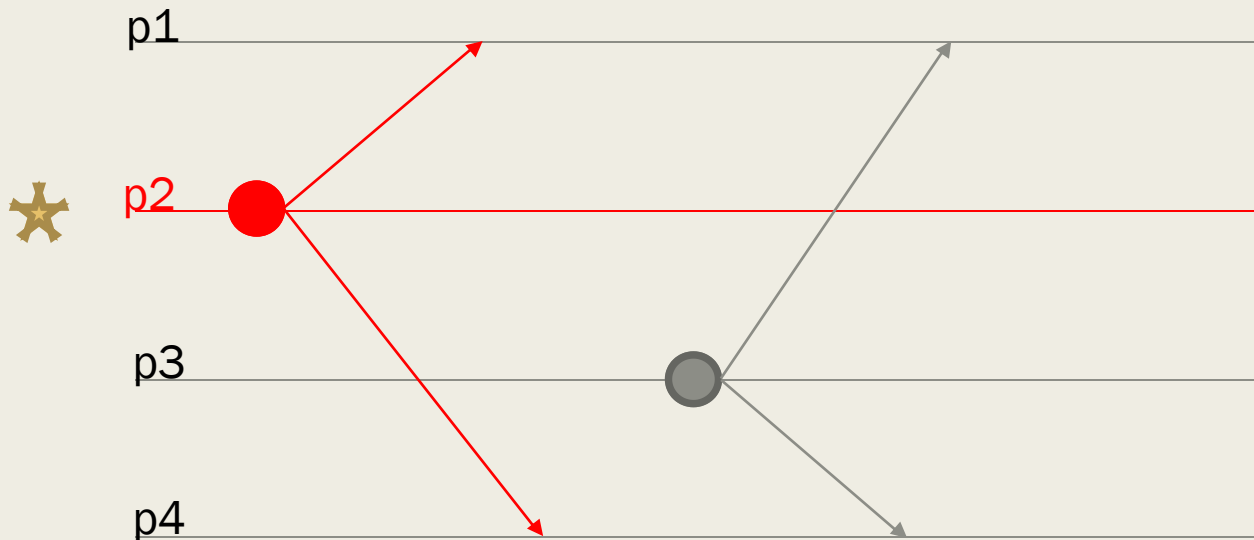
- V paralelních a tedy i distribuovaných systémech požadované vlastnosti mohou patřit mezi **životné (liveness)**, nebo **bezpečné (safety)**
- Požadujeme odpověď na požadavek? Chceme, aby dříve nebo později nastala událost odpovědi → **životná podmínka**
- Chceme, aby byly zprávy doručovány v nějakém pořadí? Nemůže nastat chyba v doručení → **bezpečná podmínka**

Spolehlivý broadcast

- **Platnost (Validity)** – pokud zpráva byla nějakým korektním procesem vysílána, pak dříve nebo později ji každý korektní proces doručí
- **Shoda (Agreement)** – pokud zpráva byla doručena korektním procesem, pak dříve nebo později bude doručena každým korektním procesem
- **Integrita (No duplication nebo Integrity)** – žádná zpráva není doručena více než jednou
- **Opravdovost (No creation)** – pokud process doručil zprávu m od procesu p , potom tento proces zprávu opravdu odeoslal

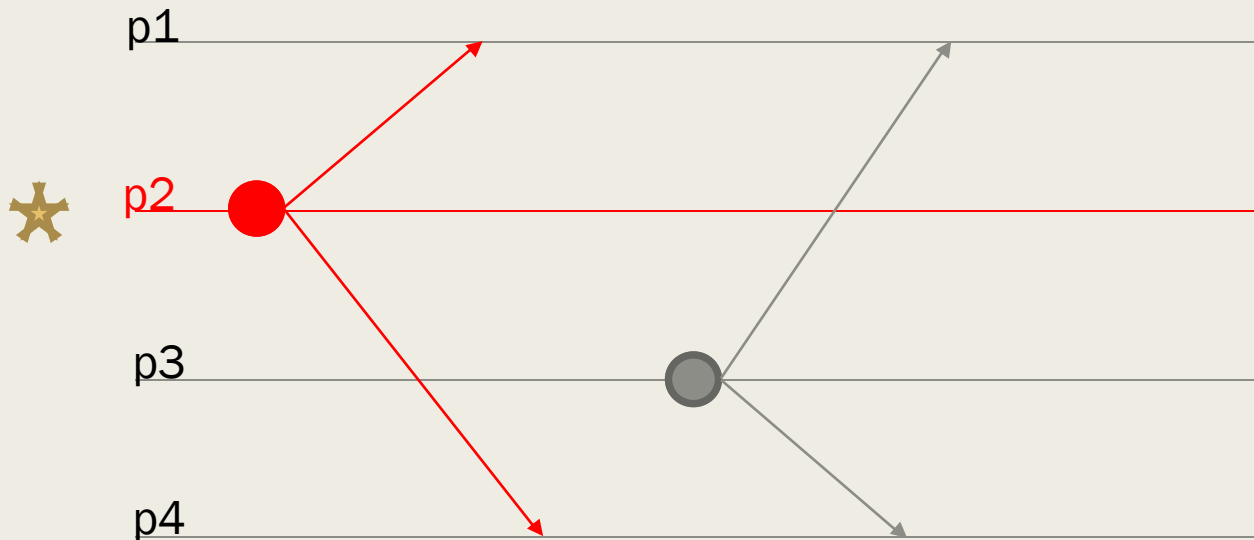
Best Effort BROADCAST

- Je zde garantována **validity (platnost)** ale ne **agreement (shoda)**
- Kdy může nastat situace, že platí validity, ale neplatí agreement?? V systému nemusí existovat pouze korektní procesy. Pak ...



Best Effort BROADCAST

- Vysílá jen jeden korektní proces p3 a od toho doručí zprávu všechny korektní procesy p1 a p4
- Proces p2 není korektní, proto **platnost** nenaruší to, že zpráva není doručena
- Zpráva od p2 byla doručena korektním p1 a p4, pak pro **shodu** musí být doručena i p3, což neplatí! Nedoručení zprávy od p3 procesu p2 ovšem **shodu** nenaruší



Spolehlivé (reliable) všesměrové vysílání

Platí validity + agreement + integrity + no creation

b'cast(R, m):

Tag m with sender(m) and seq(m)

send(m) to all neighbors including self

deliver(R, m):

upon **receive**(m) do

if p has not previously executed deliver(R, m) then

if sender(m) $\neq p$ then **send**(m) to all neighbors

deliver(R, m)

endif

enddo

Další vlastnosti všesměrového vysílání

- **1) FIFO uspořádání:** If a process b casts msg m before it b casts msg n , then no correct process delivers n unless it has previously delivered m – *“FIFO order” from single process*
- **2) Kauzální (Causal) uspořádání:** If b cast of m causally precedes b cast of n , then no correct process delivers n unless it has previously delivered m – *“FIFO order” from all processes*
- **3) Úplné (Total) uspořádání:** If correct processes p and q both deliver messages m and n , then p delivers m before n iff q delivers m before n

Přijímají z pohledu příjemců ve stejném pořadí všechny procesy

Třídy všesměrového vysílání

- $\text{Reliable} = \text{Validity} + \text{Agreement} + \text{Integrity}$
- $\text{FIFO} = \text{Reliable} + \text{FIFO Order}$
- $\text{Causal} = \text{Reliable} + \text{Causal Order}$
- $\text{Atomic} = \text{Reliable} + \text{Total Order}$
- $\text{FIFO Atomic} = \text{Reliable} + \text{FIFO Order} + \text{Total Order}$
- $\text{Causal Atomic} = \text{Reliable} + \text{Causal Order} + \text{Total Order}$

FIFA všesměrové vysílání

Inicializace:

forall q

 msgbag[q] := empty

 next[q] := 1

b'cast(F, m): b'cast(R, m)

deliver(F, m):

 upon deliver(R, m) do

q := sender(m)

 msgbag[q] := msgbag[q] \cup { m }

 while exists message n in msgbag[q] with seq(n) = next[q] do

 deliver(F, n); next[q] := next[q] + 1

 msgbag[q] := msgbag[q] - { n }

Kauzální všesměrové vysílání

- broadcast

- Podobně jako v případě Lamportova algoritmu pro kauzální vysílání musí platit relace 'happens before'
- Relace \rightarrow kauzálně předchází, když
 - $a \rightarrow b$ pokud jeden process vykonal tyto události v tomto pořadí
 - $broadcast(ma) \rightarrow deliver(ma)$
 - *Tranzitivita*
- Pokud jeden proces doručí zprávu ma a pak vyšle zprávu mb , pak všechny procesy musí doručit ma před mb , nezáleží ale v jakém pořadí doručování zpráv před doručením mb probíhá!

Kauzální všesměrové vysílání

Initializace: prevDlvs := empty

b'cast(C, m):

 b'cast(F, <prevDlvs || m>)

 prevDlvs := empty

deliver(C, m):

 upon deliver(F, <m1, ... mk>) do

 for i := 1 to k do

 if p has not previously executed deliver(C, mi)

 then

 deliver(C, mi)

 prevDlvs := prevDlvs || mi

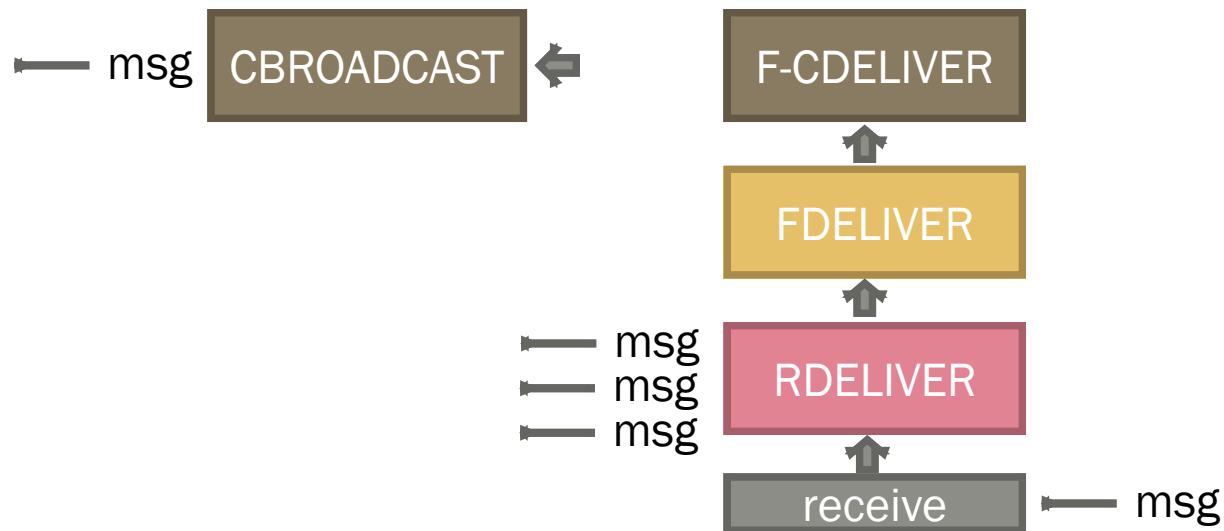
 endif

 enddo

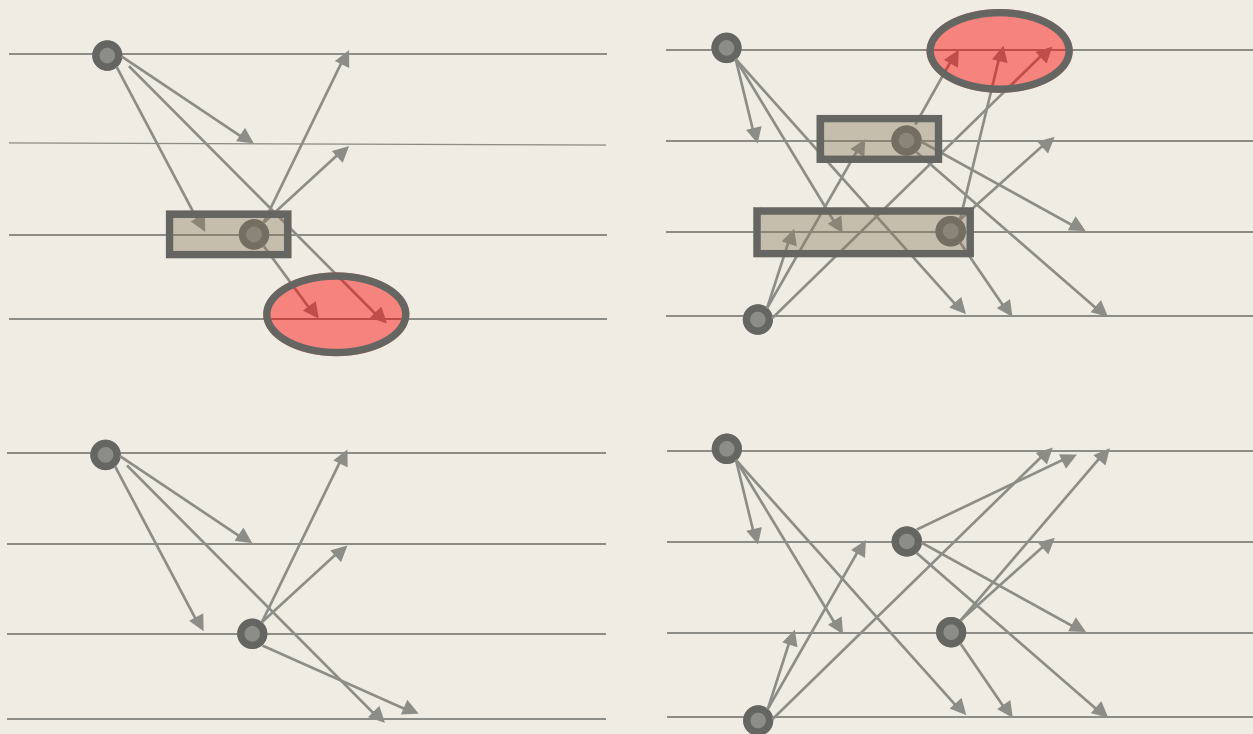
Kauzální vysílání a doručení

broadcast(C,m) // volá se pro C-bcast

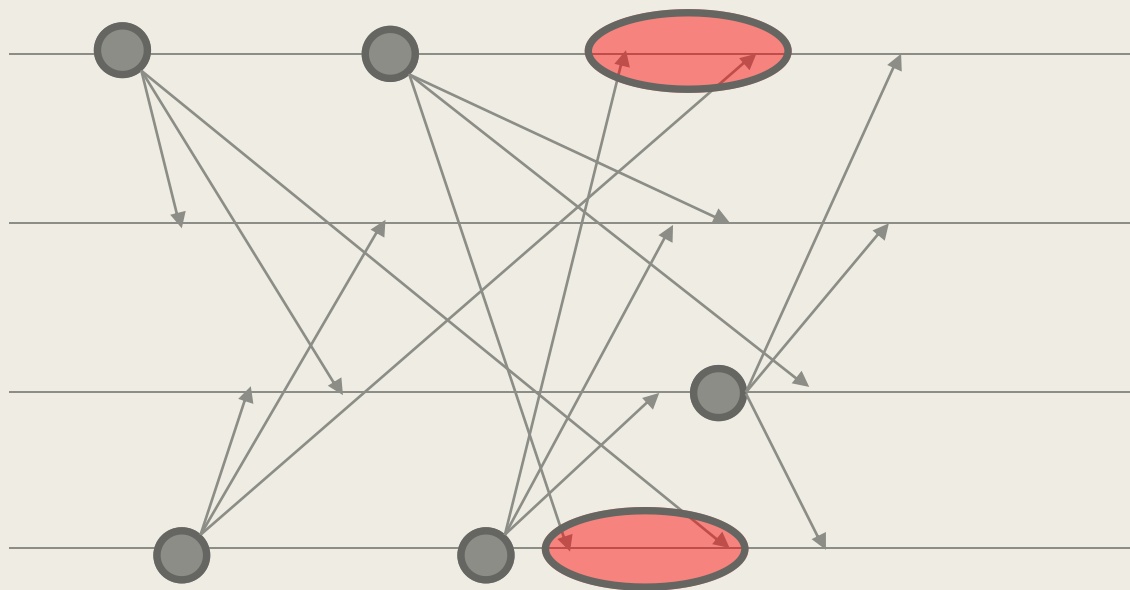
deliver(C,m) // reaguje na přijetí (FIFA-CAUSAL deliver)



KAUZÁLNÍ?



KAUZÁLNÍ, ALE NE FIFA



Atomické všesměrové vysílání (ABCAST)

- ABCAST (phase 1) odesílatel p zasílá $m(p,t)$ všem ostatním procesům
- Každý příjemce q přidá zprávu do $bag[p]$, označí ji “U” a přiřadí ji priority vyšší než doposud zjištěné priority
- Každý příjemce informuje odesílatele o přidělené prioritě zprávě m , “U” – priorita odeslaná odesílateli – “D” – priorita upravená odesílatelem
- Odesílatel sesbírá všechny odpovědi na svoji zprávu, zjistí maximální přiřazenou prioritu a o té informuje ostatní procesy
- Tyto si prioritu k procesu poznačí jako „D“
- Doručí všechny zprávy podle priorit dokud je nějaká zpráva v bagu, nebo nějaká zpráva s aktuálně nejnižší prioritou je označena jako “U”

ABCAST – stejná hodnota, rozhodnutí ?

- $P1[1]: \text{bcast}(m1, (p1, t1))$; $P2[1]: \text{bcast}(m2, (p2, t1))$
- $P1[2]: (\text{send}, U(m1, 2))$; $P2[2]: (\text{send}(p1, U(m2, 2))$;
- $P1[3]: \text{send}(P2, D(m1, 2))$; $P2[3]: \text{send}(P1, D(m2, 2))$
- $M1$ a $m2$ mají stejnou hodnotu konsensu =>
- V tom případě se doručí podle indexu procesu a časového razítka, které přidělil vysílající proces (tj. sekundární klíče jsou nejprve číslo procesu a pak časové razítko, které dal vysílající proces zprávě).

ABCAST – náznak důkazu

- Zpráva m s nejnižším rozhodnutím je nějakým procesem P doručena, a přitom přijde rozhodnutí o zprávě m' , že má vyšší nebo stejnou prioritou doručení, než má m (???)
- *A, tato zpráva ještě nebyla procesu P předána spolehlivým broadcastem, potom ale dostane vyšší číslo U od tohoto procesu (výsledná priorita musí být tím pádem nižší).*
- *B, proces již poslal své U pro tuto zprávu a to je nižší nebo stejné než je D pro zprávu m -> potom musí být ale ve frontě doručovaných zpráv záznam pro m' před m , což není*
- *Pozn: fronta zpráv k doručení řadí U před L pro stejné časové razítko, nebo alespoň podle sekundárních klíčů!*

BYZANTSKÉ PROCESY

Problém Byzantských generálů
Volby hlavního uzlu, Broadcast

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

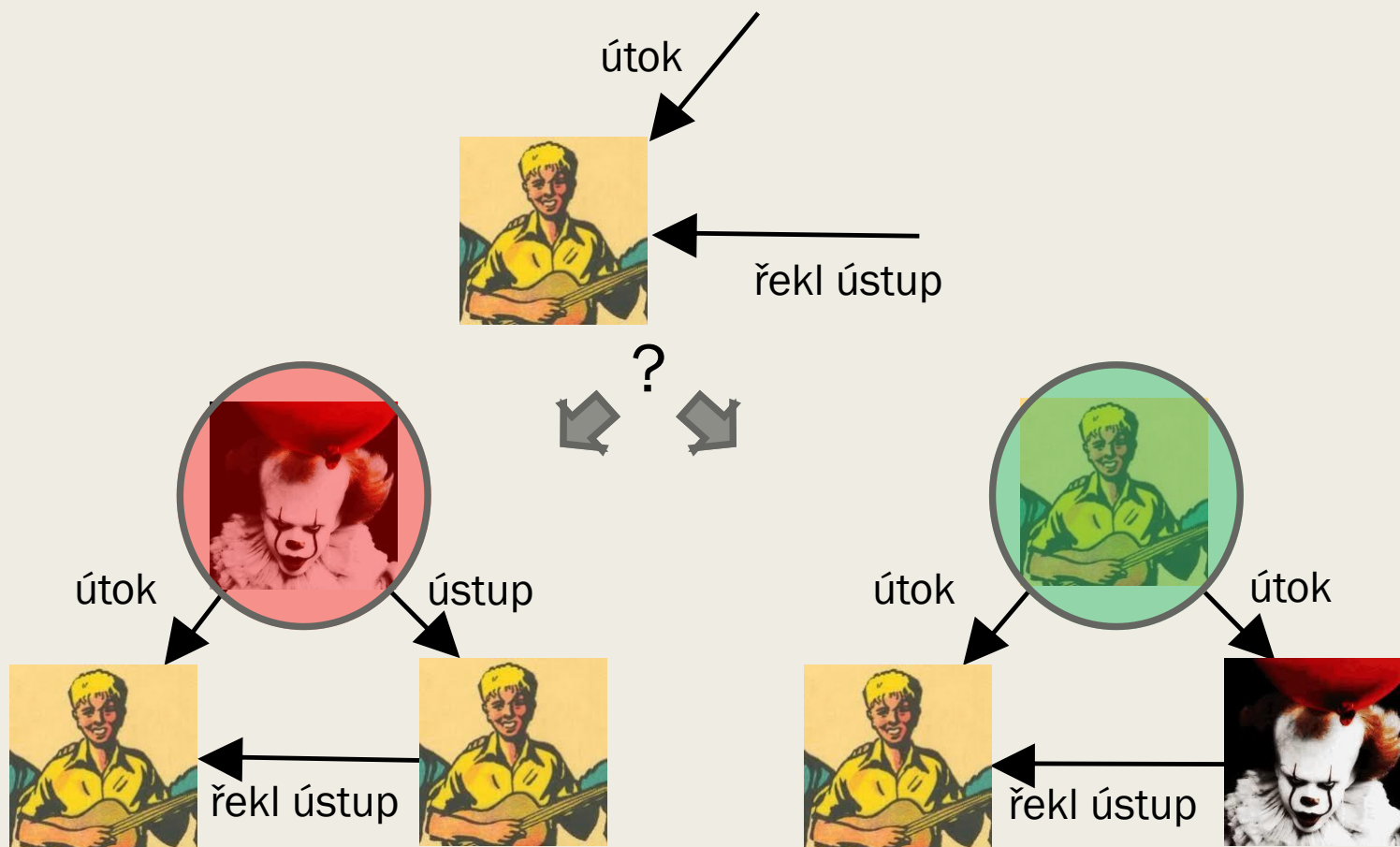
Problém Byzantských generálů – příběh

- 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 generál není loyální, pak pro způsobení ztrát vydá odlišné 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

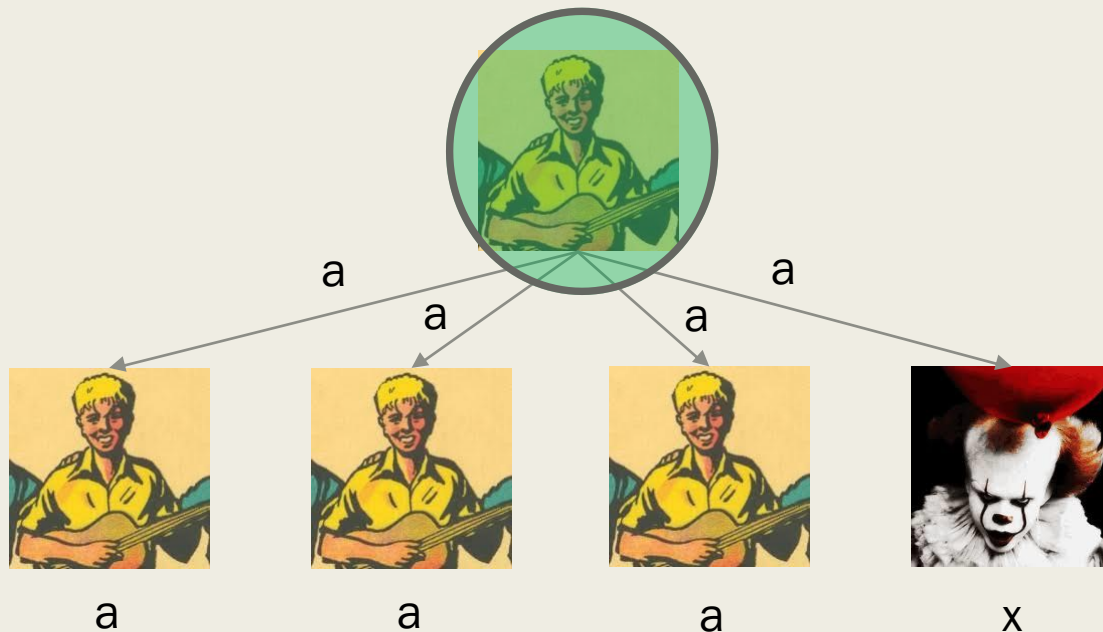
- n Albánských generálů a z toho m zrádců
- Pokud $n \leq 3*m$ problém nemá řešení
- Spor
 - *Dobrá, umíme vyřešit problém pro $n=3*m$ Albánských generálů ...*
 - ... pak bychom ale měli být schopni řešit i problém tří Byzantských generálů s jedním zrádcem!
 - Protože bychom redukovali Albánce po skupinách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

Algoritmus pro řešení PBG

- 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 v_i , 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 v_j po ukončení kroku 2 (výsledek OM($m-1$) pro tohoto seržanta), nebo Defaultní hodnotu, Pak je jím zvolenou hodnotou (na této úrovni rekurze) hodnota majority($v_1 \dots v_{n-1}$)

Generál věrný, OM(0)

- Pokud jsou všichni věrní, není co řešit a OM(0) funguje pro IC2 a tím pádem i pro IC1
- 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, $OM(m)$

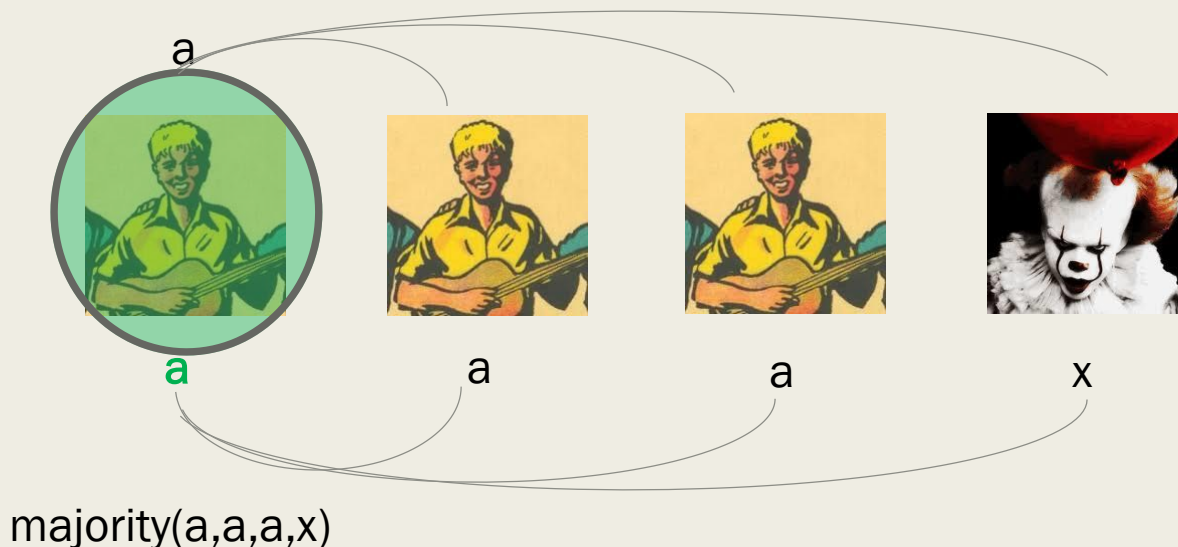
- Budeme potřebovat, aby fungoval i algoritmus $OM(m)$, $m > 0$
- Pro případ věrného generála je větší m komplikací, ale bez toho to pro obecný případ i možných něvěrných hlavních generálů nepůjde.

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 přijme jako své rozhodnutí.

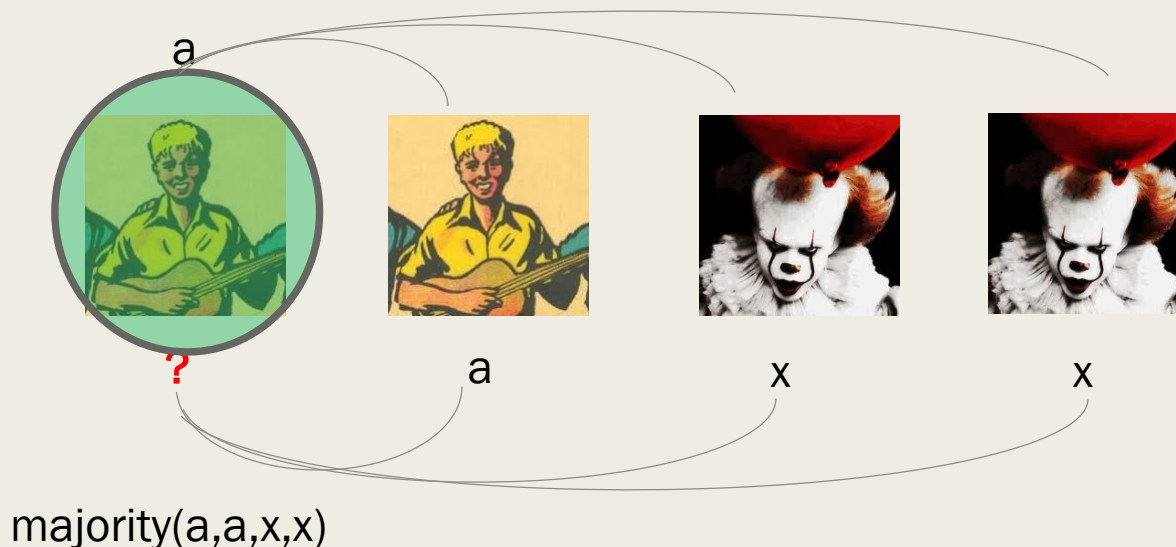
OM(1), generál věrný

- 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, přijme ji také (dle principu majority)
- Všichni ostatní věrní seržanti si stejným způsobem ověří svoji hodnotu



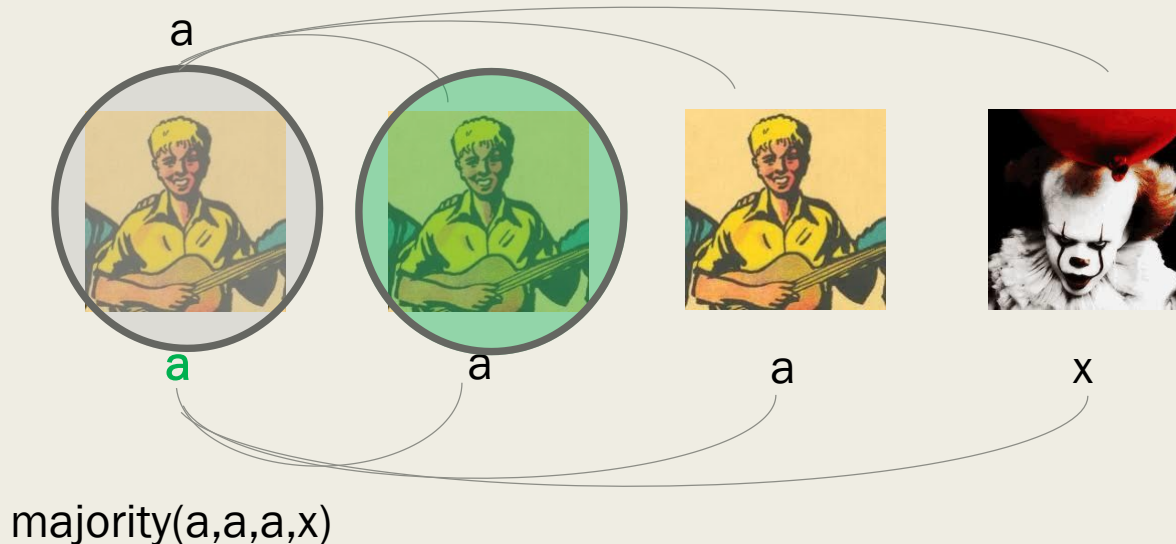
OM(1), generál věrný

- 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.



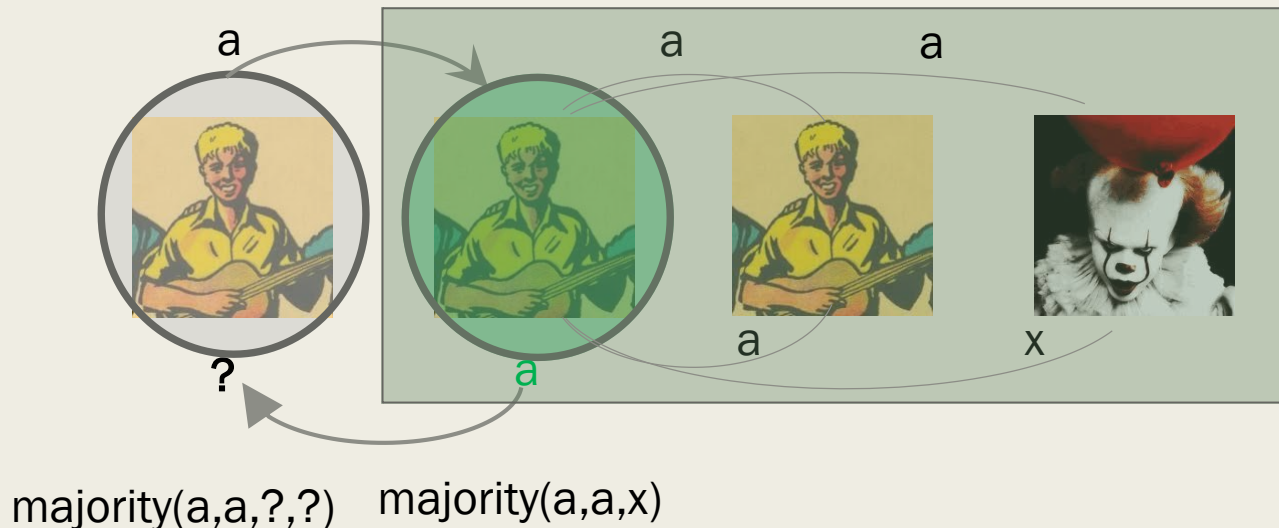
OM(2), generál věrný

- 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!



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 ještě nedojde ke zmatení, protože v o-jednoho věrného menších podskupinkách získá zrádce trochu navrch, ale stále dva loajální určí správce



Zjistí stejným způsobem pro L3 a zrádného L4

OM(m), generál věrný

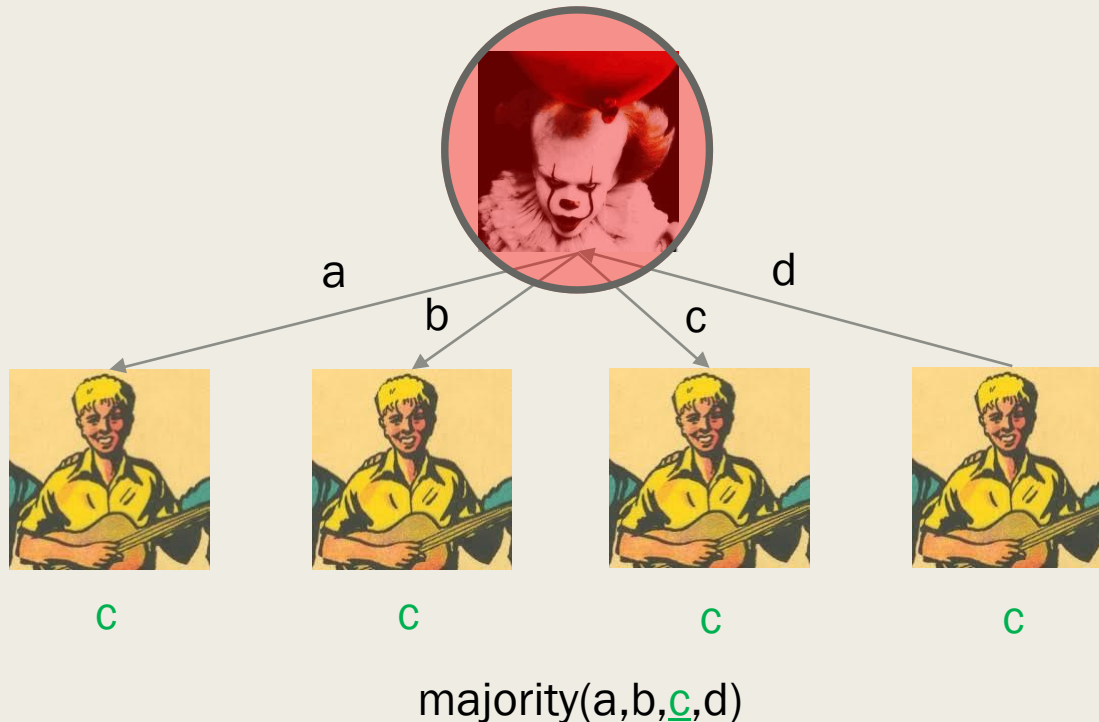
- Algoritmus OM(m) splňuje podmínku IC2, a tím i podmínku IC1, v případě, když hlavní, původní rozkaz vydávající generál je věrný, pokud platí

$$\text{CELKOVÝ_POČET_VOJÁKŮ} > 2 * \text{POČET_ZRÁDCŮ} + m$$

- 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 $5 > 2 * 2 + 1$

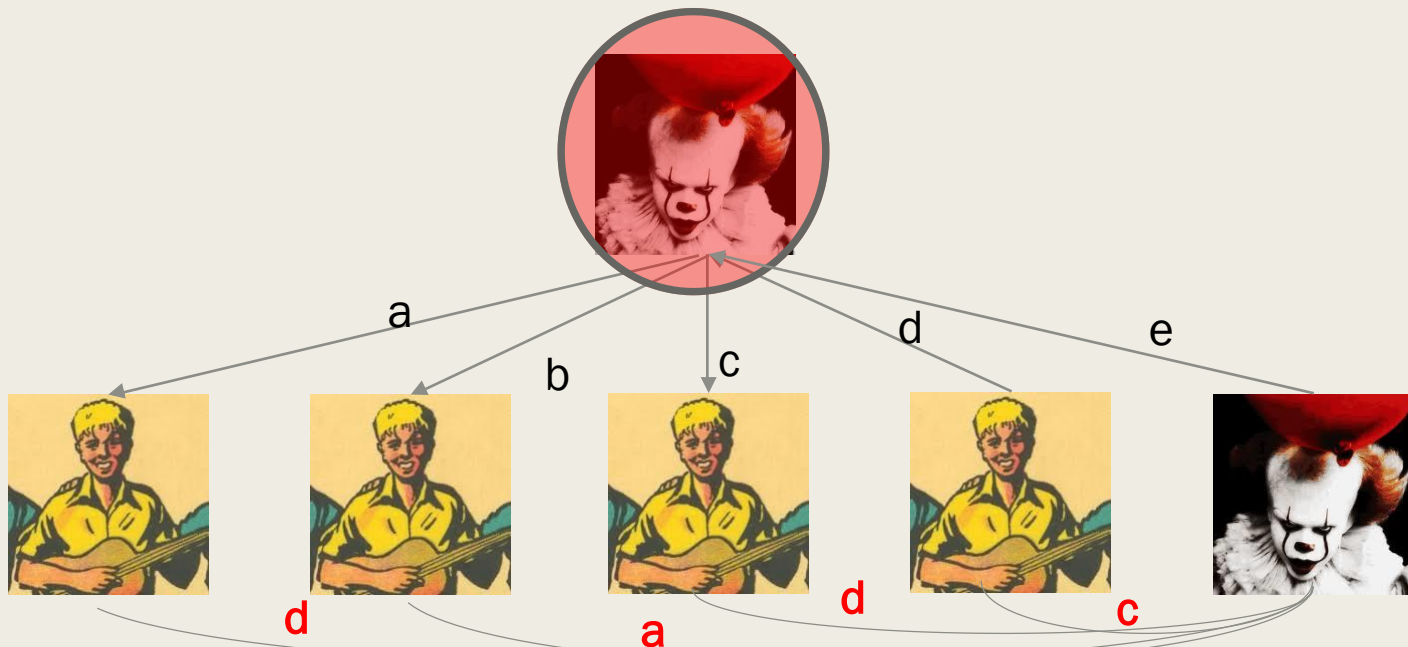
OM(1), generál zrádný

- 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



OM(1), generál zrádný (+1)

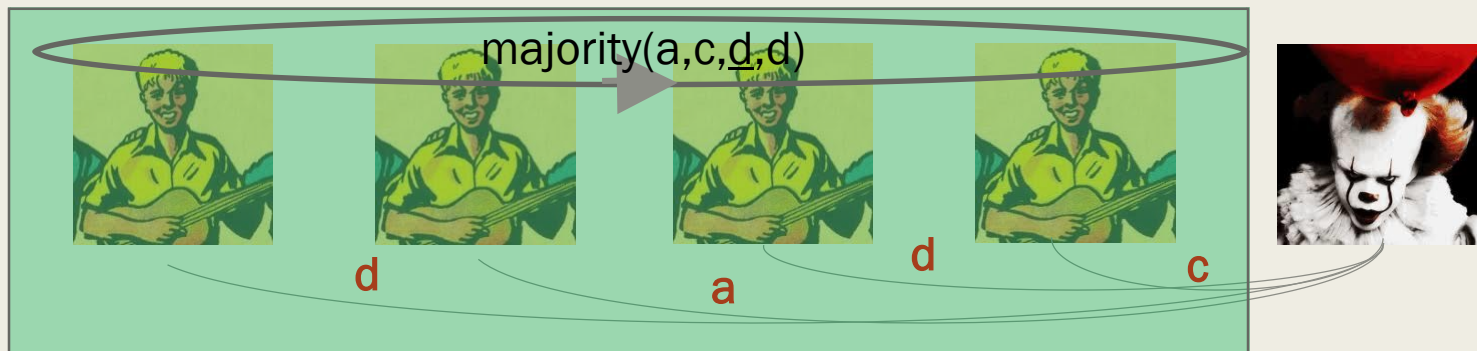
- V případě dvou zrádných vojáků z pěti by algoritmus nefungoval ani v případě věrného generála. Pro šest vojáků a dva zrádce ano.
- Fungoval by algoritmus OM(1) pro zrádného generála, jednoho ze seržantů a čtyř věrných?
- **Nefungoval**: při vyjednávání o generálově rozkazu zrádný seržant zmaří shodu 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 (viz slide „“)
- 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 obdrženu zprávu od zrádce a každý vektor zpráv v této skupině vyhodnotí stejně. Každý iniciuje jednání o zprávě, kterou dostali od zrádce a vyhodnotí ($4 \times \text{majority}(a, c, d, d)$)



- Každý iniciuje jednání o zprávě, co dostali od zrádce ($4 \times \text{majority}(a, c, \mathbf{d}, d)$)
Címž 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

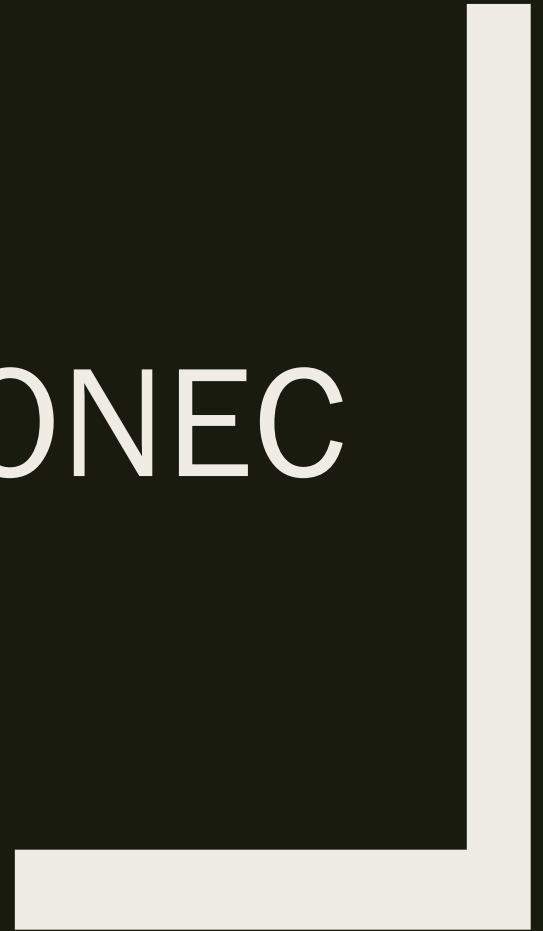
- Algoritmus $OM(m)$ řeší problém Byzantských generálů, pokud je maximálně m zrádců a celkem je více než $3*m$ vojáků
- Víme, že pro případ věrného generála dokáže algoritmus $OM(m)$ zajistit obě podmínky **IC1** a **IC2**, pokud počet vojáků je větší než dvojnásobek zrádců plus m , což platí, protože 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 $> 3* \text{zrádců}$, tedy $3*m-1 > 3*(m-1)$, což platí
- Provedením OM($m-1$) každý dva věrní seržanti se shodnou na stejné hodnotě v_j .

* 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.

KONEC

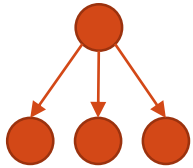


PROBLÉM BYZANTSKÝCH GENERÁLŮ, PŘÍKLAD

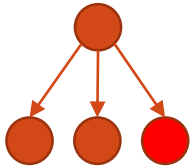
FZjr 2019



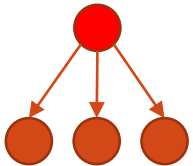
1 ZRÁDCE PRO OM(0)



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



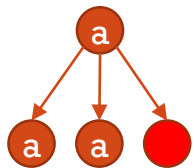
OM(0) funguje (!)



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

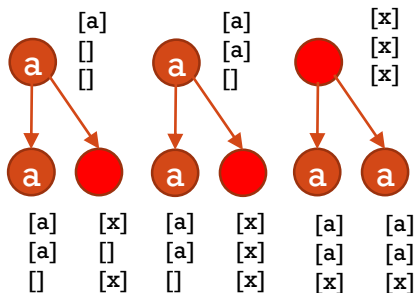


1 ZRÁDCE PRO OM(1)



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

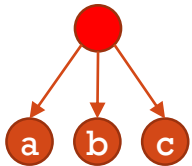
Všichni sice od poctivce obdrží tu samou informaci, OM(0) by problém řešila, ale zkusíme, jestli je toto robustní pro jedno kolo porady



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

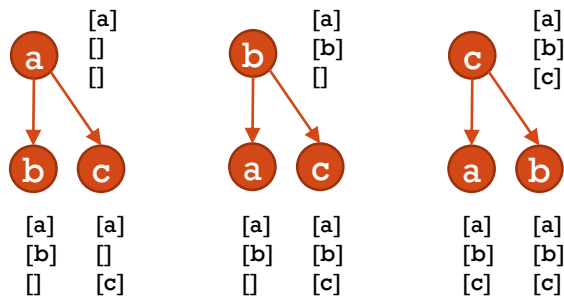


1 ZRÁDCE PRO OM(1)

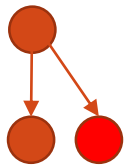


Pokud oním zrádcem je maršál, MP(0) nefunguje, ale co porada?

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



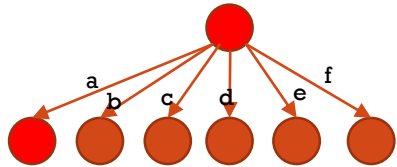
2 ZRÁDCI, OM(2), MARŠÁL VĚRNÝ



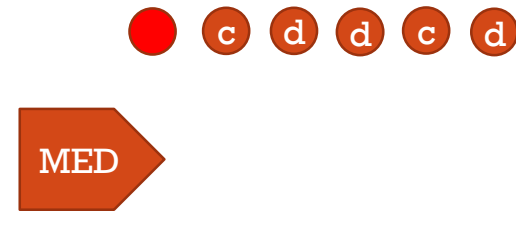
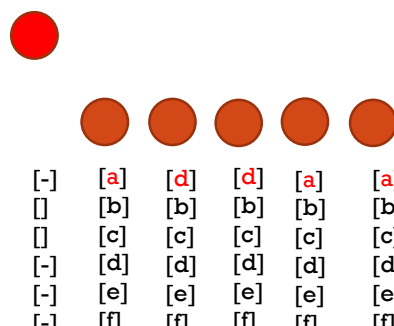
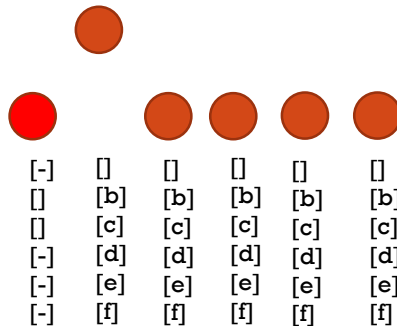
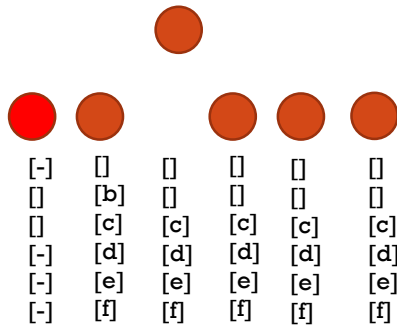
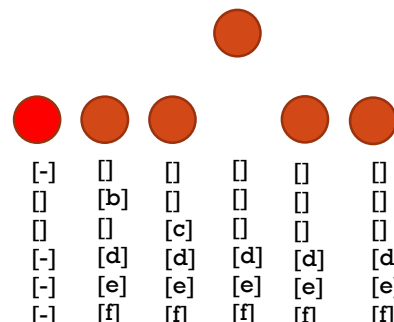
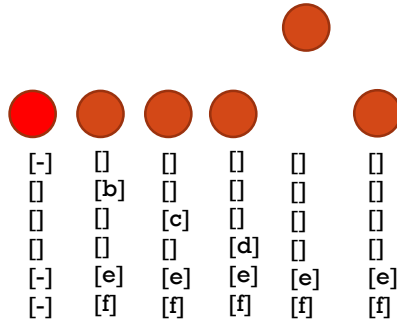
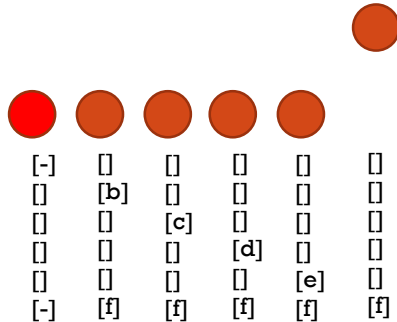
Při první poradě má každý poctivý generál (zahajovatel první rady) správnou hodnotu od věrného maršála, jeden ze seržantů také, a druhý seržant je zrádce
Druhá porada probíhá mezi oběma seržanty, u zrádce by to byla jen předstíraná snaha, ale poctivý seržant je manipulován zrádcem a ... může zvolit jinou 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 víc 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













					
[-]	[a]	[d]	[d]	[a]	[d]
[-]	[b]	[b]	[b]	[b]	[b]
[-]	[c]	[c]	[c]	[c]	[c]
[-]	[d]	[d]	[d]	[d]	[d]
[-]	[e]	[e]	[e]	[e]	[e]
[-]	[f]	[f]	[f]	[f]	[f]






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 ...

				
[a]	[]	[]	[]	[]
[]	[d]	[]	[]	[]
[]	[]	[d]	[]	[]
[]	[]	[]	[a]	[]
[d]	[d]	[d]	[d]	[d]

				
[a]	[]	[]	[]	[]
[]	[d]	[]	[]	[]
[]	[]	[d]	[]	[]
[a]	[a]	[a]	[a]	[a]
[d]	[d]	[d]	[d]	[d]

				
[a]	[]	[]	[]	[]
[]	[d]	[]	[]	[]
[d]	[d]	[d]	[d]	[d]
[a]	[a]	[a]	[a]	[a]
[d]	[d]	[d]	[d]	[d]

				
[a]	[]	[]	[]	[]
[d]	[d]	[d]	[d]	[d]
[d]	[d]	[d]	[d]	[d]
[a]	[a]	[a]	[a]	[a]
[d]	[d]	[d]	[d]	[d]

				
[a]	[a]	[a]	[a]	[a]
[d]	[d]	[d]	[d]	[d]
[d]	[d]	[d]	[d]	[d]
[a]	[a]	[a]	[a]	[a]
[d]	[d]	[d]	[d]	[d]





[-]	[d]	[d]	[d]	[d]	[d]
[-]	[b]	[b]	[b]	[b]	[b]
[-]	[c]	[c]	[c]	[c]	[c]
[-]	[d]	[d]	[d]	[d]	[d]
[-]	[e]	[e]	[e]	[e]	[e]
[-]	[f]	[f]	[f]	[f]	[f]

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

[x]	[]	[]	[]	[]
[]	[b]	[]	[]	[]
[]	[]	[b]	[]	[]
[]	[]	[]	[b]	[]
[x]	[b]	[b]	[b]	[b]

[x]	[]	[]	[]	[]
[]	[b]	[]	[]	[]
[]	[]	[b]	[]	[]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]

[x]	[]	[]	[]	[]
[]	[b]	[]	[]	[]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]







[x]	[]	[]	[]	[]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]

[x]	[a]	[a]	[d]	[d]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]
[x]	[b]	[b]	[b]	[b]

MED

b b b b b













					
[-]	[a]	[d]	[d]	[a]	[d]
[-]	[b]	[b]	[b]	[b]	[b]
[-]	[c]	[c]	[c]	[c]	[c]
[-]	[d]	[d]	[d]	[d]	[d]
[-]	[e]	[e]	[e]	[e]	[e]
[-]	[f]	[f]	[f]	[f]	[f]






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
 $\Rightarrow 6 \cdot 5 \cdot 4$ jednání o shodách, každé jednání mezi čtyřmi agenty






Navíc tento systém (tři zrádci ze sedmi) nebude fungovat v případě věrného maršála! Pro tři zrádce musí být minimálně sedm poctivců.

				
[a]	[]	[]	[]	[]
[]	[x]	[]	[]	[]
[]	[]	[d]	[]	[]
[]	[]	[]	[a]	[]
[d]	[x]	[d]	[d]	[d]

				
[a]	[]	[]	[]	[]
[]	[x]	[]	[]	[]
[]	[]	[d]	[]	[]
[a]	[x]	[a]	[a]	[a]
[d]	[x]	[d]	[d]	[d]

				
[a]	[]	[]	[]	[]
[]	[x]	[]	[]	[]
[d]	[x]	[d]	[d]	[d]
[a]	[x]	[a]	[a]	[a]
[d]	[x]	[d]	[d]	[d]

				
[a]	[]	[]	[]	[]
[a]	[x]	[a]	[d]	[d]
[d]	[x]	[d]	[d]	[d]
[a]	[x]	[a]	[a]	[a]
[d]	[x]	[d]	[d]	[d]

				
[a]	[x]	[a]	[a]	[a]
[a]	[x]	[a]	[d]	[d]
[d]	[x]	[d]	[d]	[d]
[a]	[x]	[a]	[a]	[a]
[d]	[x]	[d]	[d]	[d]

MED

				
--	--	--	--	--

