

PDV 11 2019/2020

# Vzájemné vyloučení procesů

Michal Jakob

[michal.jakob@fel.cvut.cz](mailto:michal.jakob@fel.cvut.cz)

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



# Příklad (zjednodušený)

Bankovní server v cloudu.

Dva zákazníci současně vloží 10 000 Kč skrze vkladový bankomat na jeden stejný účet.

- oba bankomaty si přečtou původní zůstatek na účtu (1 000 Kč)
- oba bankomaty k zůstatku přičtou lokálně vklad (=11 000 Kč)
- oba bankomaty výsledný zůstatek uloží na server

Právě jste ztratili 10 000 Kč!

Je třeba zaručit, že v jeden okamžik provádí aktualizaci zůstatku **maximálně jeden** bankomat (a ostatní procesy jsou z něj **vyloučeny**).

# Další příklady

## Distribuovaný souborový systém

- uzamykání souborů a adresářů

## Přístup k distribuovaným objektům

- zajistit, že v jednom okamžiku má přístup k objektu maximálně jeden proces

## Koordinace serverů

- výpočet/zpracování je rozdělen přes několik serverů
- servery koordinují pomocí zámků

# Problém vzájemného vyloučení procesů (mutual exclusion)

**Kritická sekce (KS):** část kódu (všech procesů), u které potřebujeme zaručit, že ji vykonává v každém okamžiku **maximálně jeden proces.**

Dvě funkce

- **enter()** pro vstup k KS
- **exit()** pro výstup z KS

# Příklad

## Bankomat 1

```
enter(S);  
// začátek přístup ke zdroji  
přečti zůstatek ze zázamu;  
přičti vklad;  
aktualizuj záznam o zůstatku;  
// konec přístupu ke zdroji  
exit(S);
```

## Bankomat 2

```
enter(S);  
// začátek přístup ke zdroji  
přečti zůstatek ze zázamu;  
přičti vklad;  
aktualizuj záznam o zůstatku;  
// konec přístupu ke zdroji  
exit(S);
```

# Jak řešit vyloučení procesu?

## Jeden OS

(Všechny procesy v jednom OS na jednom počítači nebo VM.)

Můžeme použít semaforey, mutexy, monitor a další abstrakce poskytované OS založené na **sdílené paměti**.

## Distribuovaný systém

(procesy komunikují posíláním zpráv)

Potřebujeme **distribuovaný protokol/algorithmus**.

# Korektnost distribuovaného výpočtu

## Živost (Liveness)

Garance, že v DS **časem** dojde k něčemu **dobrému** (bude dosažen žádoucí stav).

Příklady:

- Distribuovaný výpočet: výpočet skončí.
- Konsensus: všechny procesy se shodnou na výstupní hodnotě.
- Úplnost při detekci selhání: každé selhání je časem detekováno.

## Bezpečnost (Safety)

Garance, že v DS **nikdy** nedojde k něčemu **špatnému** (nebude dosažen nežádoucí stav).

Příklady:

- Nedojde k uváznutí (deadlocku)
- Žádný objekt se nestane sirotkem
- Přesnost při detekci selhání
- Konsensus: Žádné dva procesy nevyprodukují různý výstup.

# Požadavky na algoritmus pro vyloučení procesu

**Bezpečnost:** nejvýše jeden proces v kritické sekci v kterémkoliv okamžiku

**Živost:** každý požadavek na vstup do kritické sekce je časem uspokojen

**Uspořádání (volitelný):** předchází-li žádost jednoho procesu kauzálně žádost druhého procesu, bude vstup nejprve dovolen prvnímu procesu



# Model

Skupina  $N$  procesů.

Procesy **neselhávají**.

**FIFO perfektní komunikační kanál** mezi každým párem procesů, tj. zprávy se neduplikují, nevznikají, neztrácejí a jsou doručovány v pořadí odeslání.

**Asynchronní systém**: neznáma, ale **konečná latence**.



# Centralizovaný algoritmus

# Centralizovaný algoritmu

Zvolíme **koordinátora** (pomocí algoritmu volby lídra/koordinátora)

Koordinátor spravuje:

- speciální **token**, který umožňuje držiteli vstup do KS
- **frontu** požadavků na vstup do kritické sekce (KS)

# Centralizovaný algoritmus

## Akce libovolného procesu

### **enter()**

pošli požadavek  
koordinátorovi

čekej na přijetí TOKEN od  
koordinátora; po přijetí  
TOKENu vstup do KS

### **exit()**

předej TOKEN zpět  
koordinátorovi

## Akce koordinátora

Po přijetí požadavku z procesu  $P_i$   
**if** (koordinátor má TOKEN)  
    předej TOKEN procesu  $P_i$   
**else**  
    přidej  $P_i$  do **fronty**

Po přijetí TOKENu od procesu  $P_i$   
**if** (**fronta** není prázdná)  
    vyzvedni proces z hlavy  
    **fronty** a pošli mu  
    TOKEN  
**else**  
    uchovej TOKEN

# Analýza centralizovaného algoritmu

**Bezpečnost:** Maximálně jeden proces v KS

- splněno: máme pouze jeden token

**Živost:** na každý požadavek časem dojde

- fronta má maximálně  $N$  čekajících procesů
- pokud každý proces časem doběhne a nedochází k selhání, tak je živost garantována

**Uspořádání:** přístup je poskytován v pořadí došlých žádostí

- šlo by uspořádat logickými hodinami

# Analýza výkonnosti

Efektivní vyloučení procesu vyžaduje **méně koordinačních zpráv** a procesy mají **kratší čekací dobu** na vstup.

**Komunikační zátěž:** počet zpráv poslaných při každém vstupu a výstupu do/z KS.

**Zpoždění klienta:** zpoždění klientského procesu při každém vstupu (a výstupu) do/z KS, tj. když žádné jiné procesy nečekají (tj. když je KS volná).

**Synchronizační zpoždění:** časový interval mezi vystoupením jednoho procesu KS a vstupem následujícího procesu do KS (když je pouze jeden čekající proces).

# Analýza výkonosti centralizovaného algoritmu

Komunikační zátěž:

- vstup: **2** zprávy
- výstup: **1** zpráva

Zpoždění klienta:

- **2** komunikační latence (odeslání požadavku a obdržení tokenu)

Synchronizační zpoždění

- **2** komunikační latence (vrácení tokenu a obdržení tokenu)

Ale: Koordinátor je centrální **úzké hrdlo** (a centrální bod selhání).



# Kruhový algoritmus

(Ring-based algorithm)



# Kruhový algoritmus

Velmi jednoduchý algoritmus.

$N$  procesů organizovaných **do kruhu**.

Každý proces může poslat zprávu svému **následníkovi**.

Koluje právě **jeden TOKEN**.

# Kruhový algoritmus

## Akce libovolného procesu

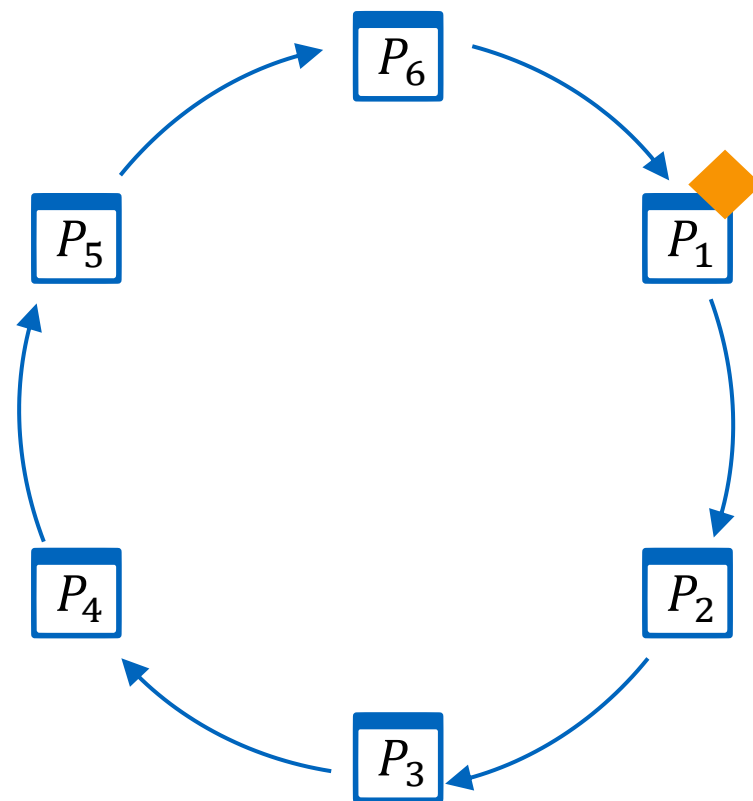
### **enter()**

vyčkej, dokud nedostaneš TOKEN;  
po obdržení vstup do KS;

**exit()** // předpokládá, že proces je v KS  
pošli TOKEN následníkovi;

Jinak pokud obdržíš TOKEN a nejsi aktuálně v KS, tak předej TOKEN následníkovi.

# Kruhový algoritmus



Má aktuálně TOKEN →  
může vstoupit do KS

TOKEN předáván  
následníkovi

Má aktuálně TOKEN →  
může vstoupit do KS

 proces

 TOKEN

# Analýza kruhové algoritmu

**Bezpečnost:** splněná – právě jeden TOKEN.

**Živost:** TOKEN časem oběhnou celý kruh (nepředpokládáme selhání).

**Komunikační zátěž:**

- vstup: implicitně  $N$  zpráv skrze systém (kolují neustále)
- výstup: 1 zpráva

**Zpoždění klienta:** 0 až  $N$  komunikačních latencí po žádosti o vstup

- nejlepší případ: žádající proces už má TOKEN
- nejhorší případ: TOKEN zrovna odeslán následníkovi

**Synchronizační zpoždění:** 1 až  $N$  komunikačních latencí

- nejlepší případ: proces žádající vstup je následníkem procesu opouštějící KS
- nejhorší případ: proces žádající vstup je předchůdce procesu opouštějící KS

# Analýza kruhové algoritmu

Zpoždění klienta a synchronizační zpoždění kruhového algoritmu je  $\mathcal{O}(N)$ .

Můžeme zlepšit?



# Ricart-Agrawalův Algorithmus

# Algoritmus Ricart-Agrawala

Klasický algoritmus z roku 1981.

*Nepoužívá* TOKEN, ale využívá **kauzalitu** (Lamportovy hodiny) a **multicast**.

Má nižší synchronizační zpoždění a zpoždění klienta než kruhový algoritmus a zároveň nepotřebuje centrální proces.

# Logika Ricart-Agrawala

Každý proces si udržuje logickou proměnou **stav** (inicializovanou na RELEASED) a **seznam** požadavků na vstup.

## $P_i$ : enter()

**nastav stav** na WANTED  $\langle T_i, i \rangle$ ;

**pošli** multicast REQUEST  $\langle T_i, i \rangle$  všem procesům, kde  $T_i$  = aktuální Lamportův logický čas v  $P_i$ ;

**čekej** dokud všechny procesy nepošlou zpět OK;

po přijetí OK: **změň stav** na HELD a **vstup** do KS

## $P_i$ : po přijetí REQUEST $\langle T_j, P_j \rangle, i \neq j$

**if** (**stav** = HELD) nebo (**stav** = WANTED  $\langle T_i, i \rangle$  a  $\langle T_i, i \rangle < \langle T_j, j \rangle$ )  
    **přidej** REQUEST do **seznamu** čekajících požadavků;  
**else**  
    **pošli** OK do  $P_j$ ;

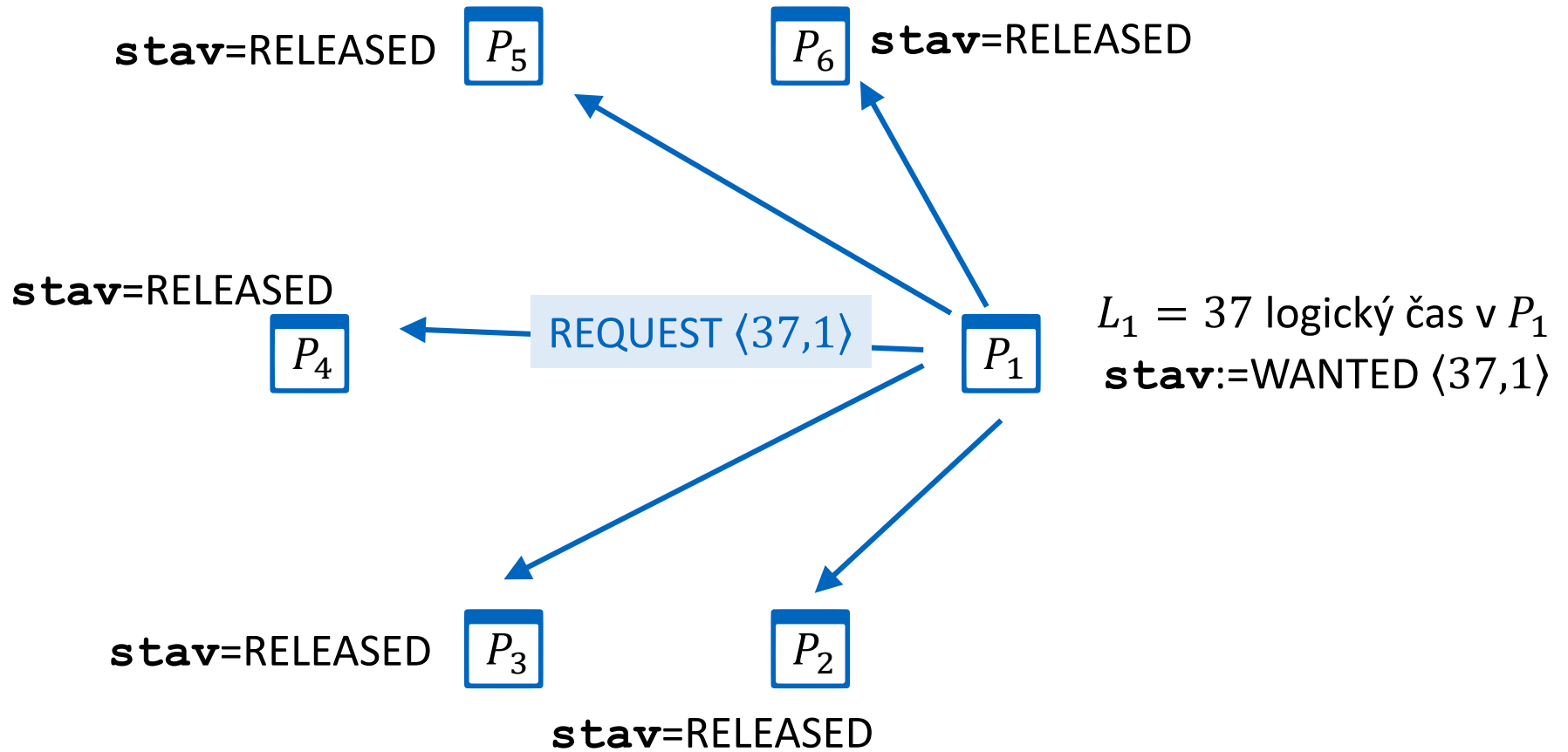
## $P_i$ : exit()

**nastav stav** na RELEASED;

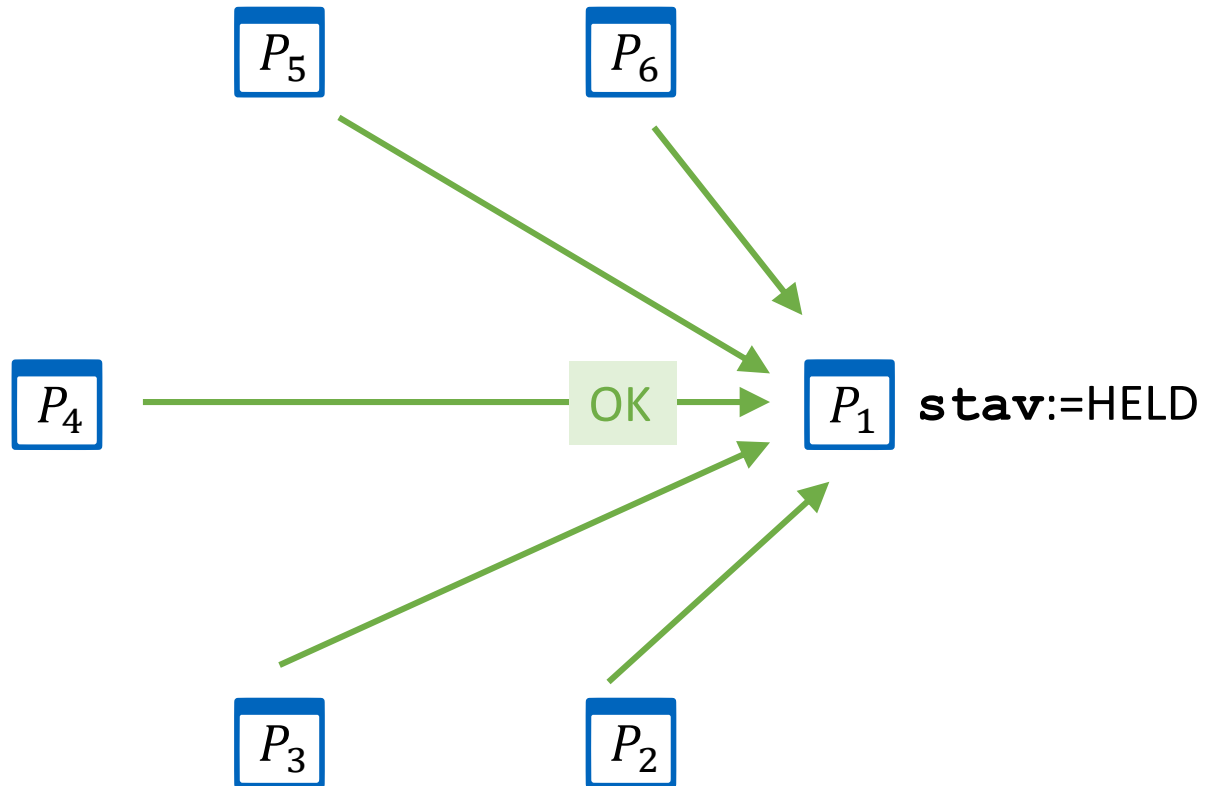
**pošli** OK všem čekajícím procesům ze **seznamu** ;



# Příklad

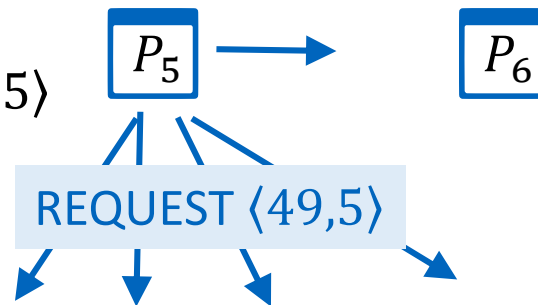


# Příklad



# Příklad

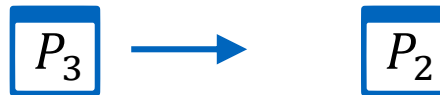
**stav:=**  
WANTED  $\langle 49,5 \rangle$



$P_1$  **stav=HELD**

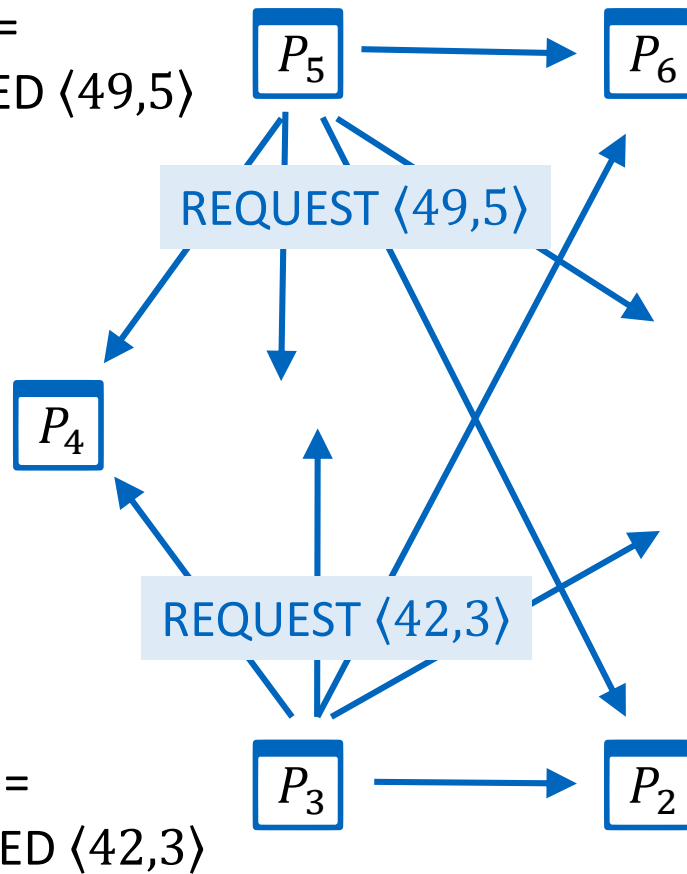


**stav:=**  
WANTED  $\langle 42,3 \rangle$



# Příklad

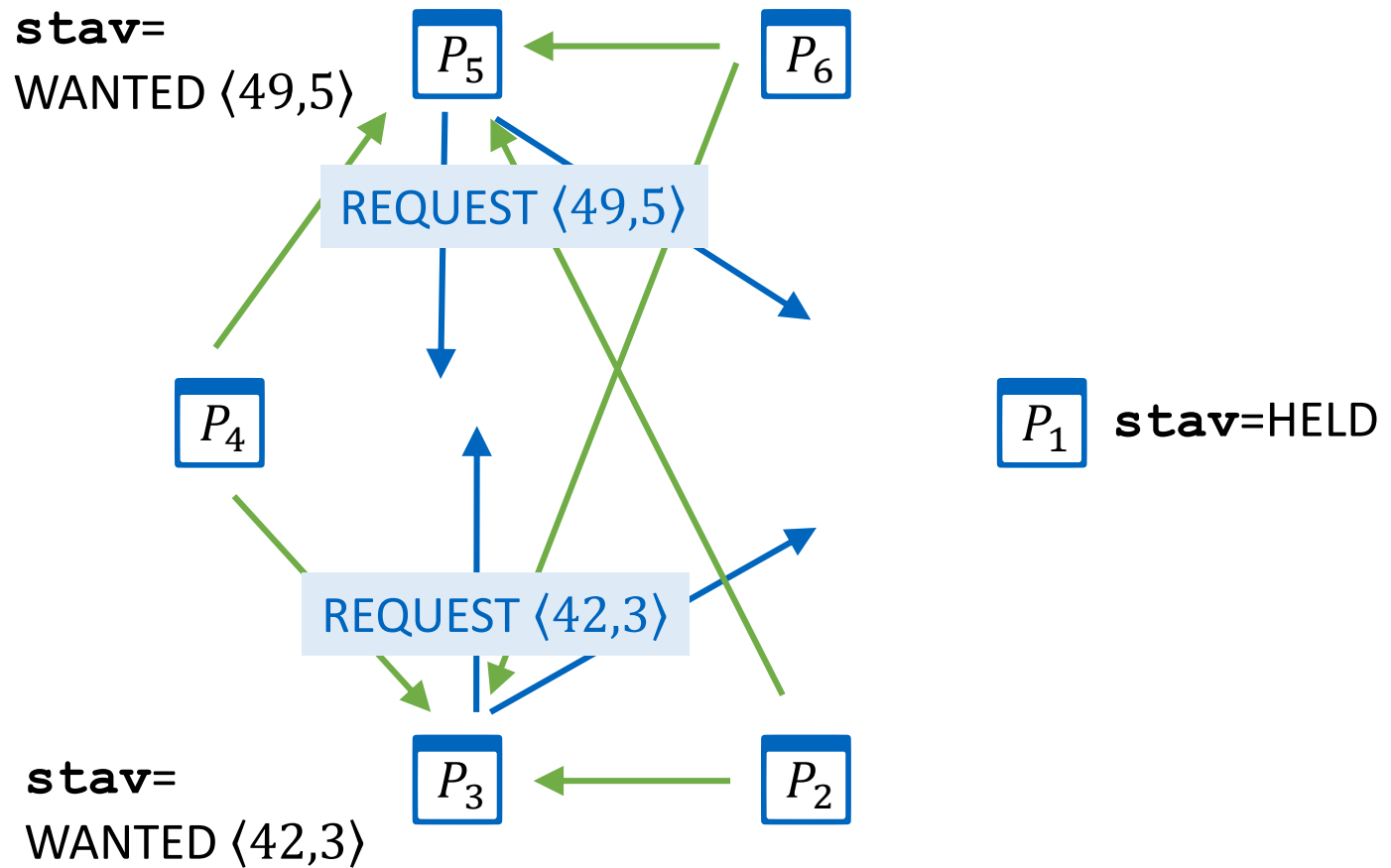
**stav:=**  
WANTED  $\langle 49,5 \rangle$



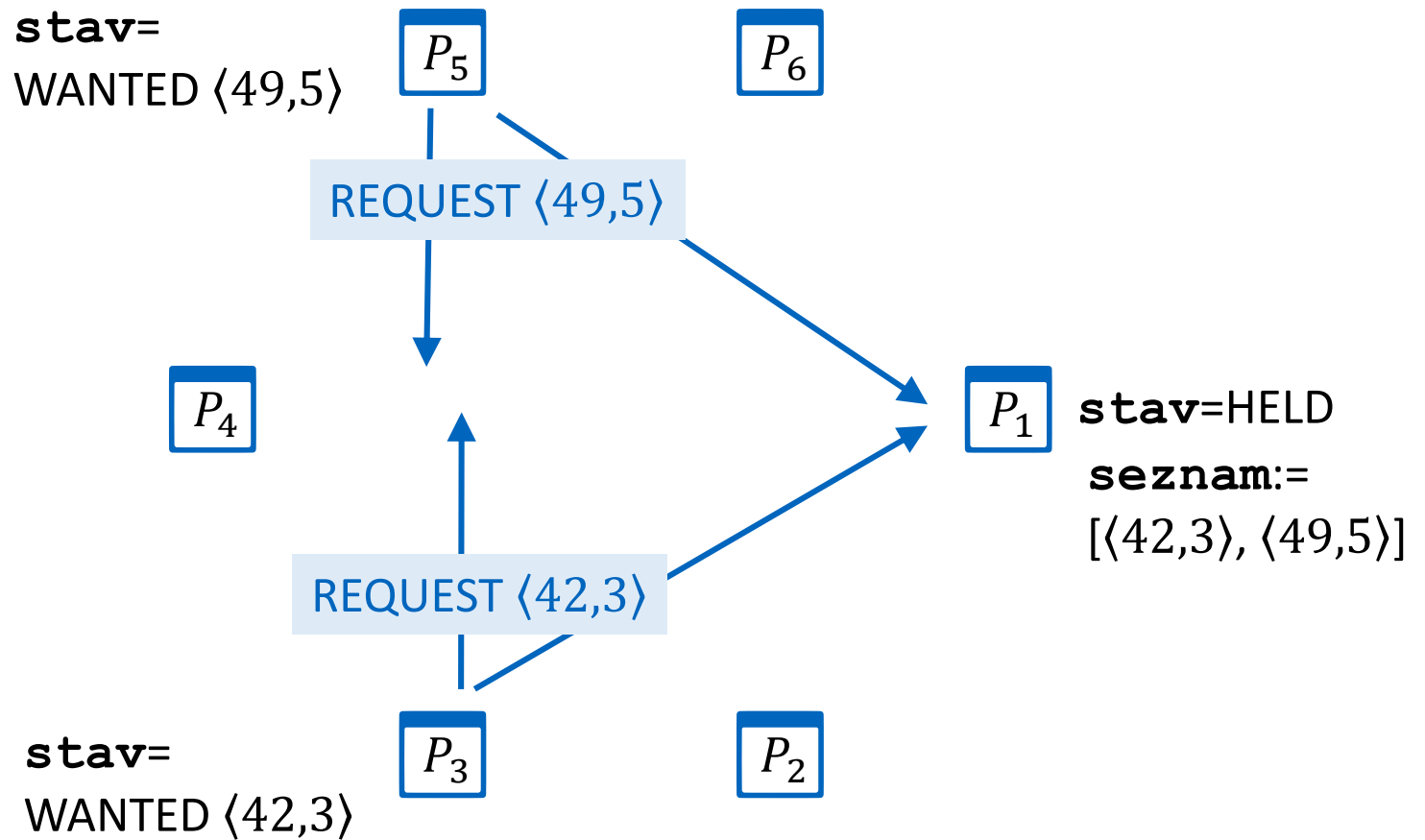
$P_1$  **stav=HELD**

**stav:=**  
WANTED  $\langle 42,3 \rangle$

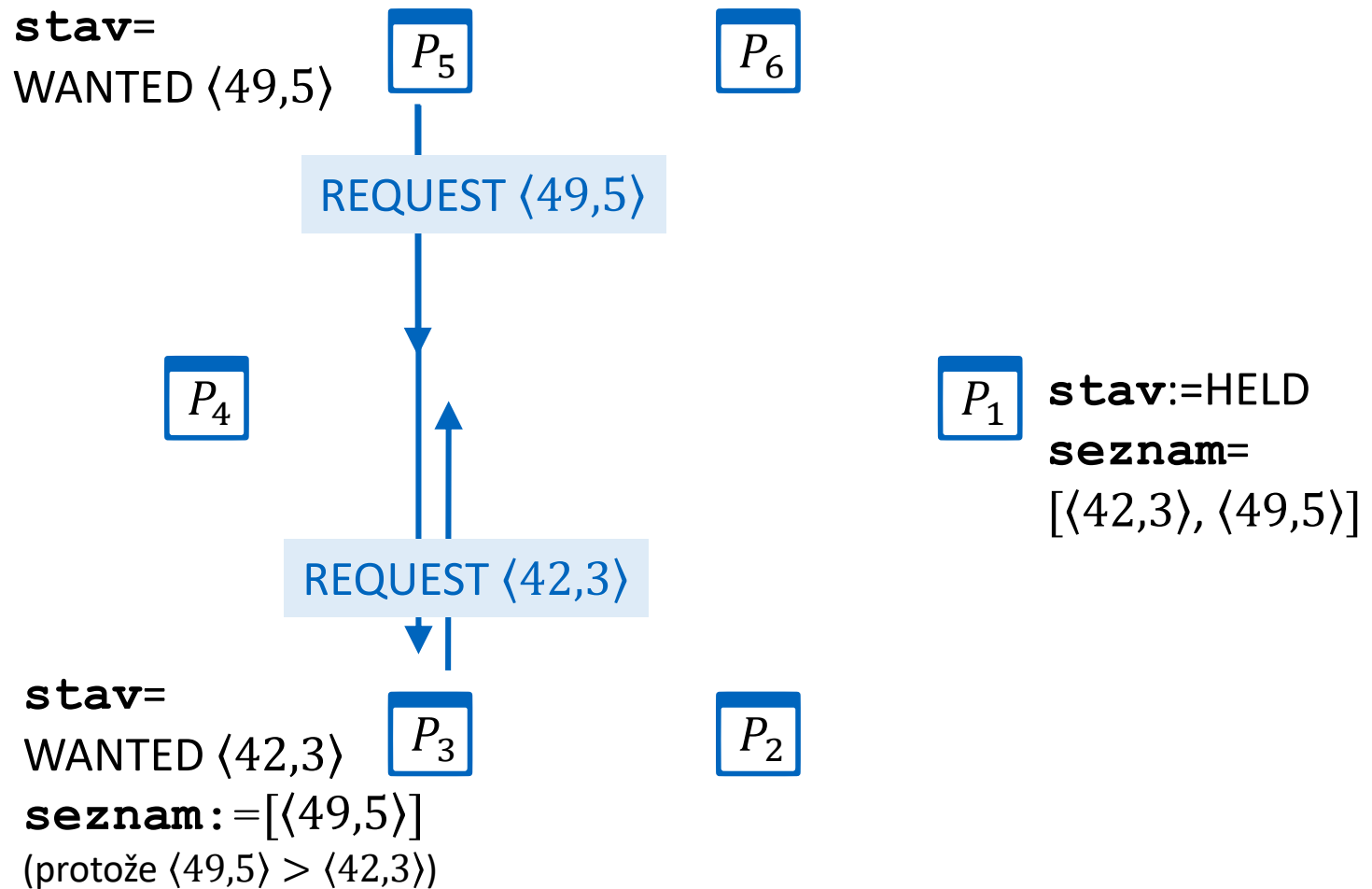
# Příklad



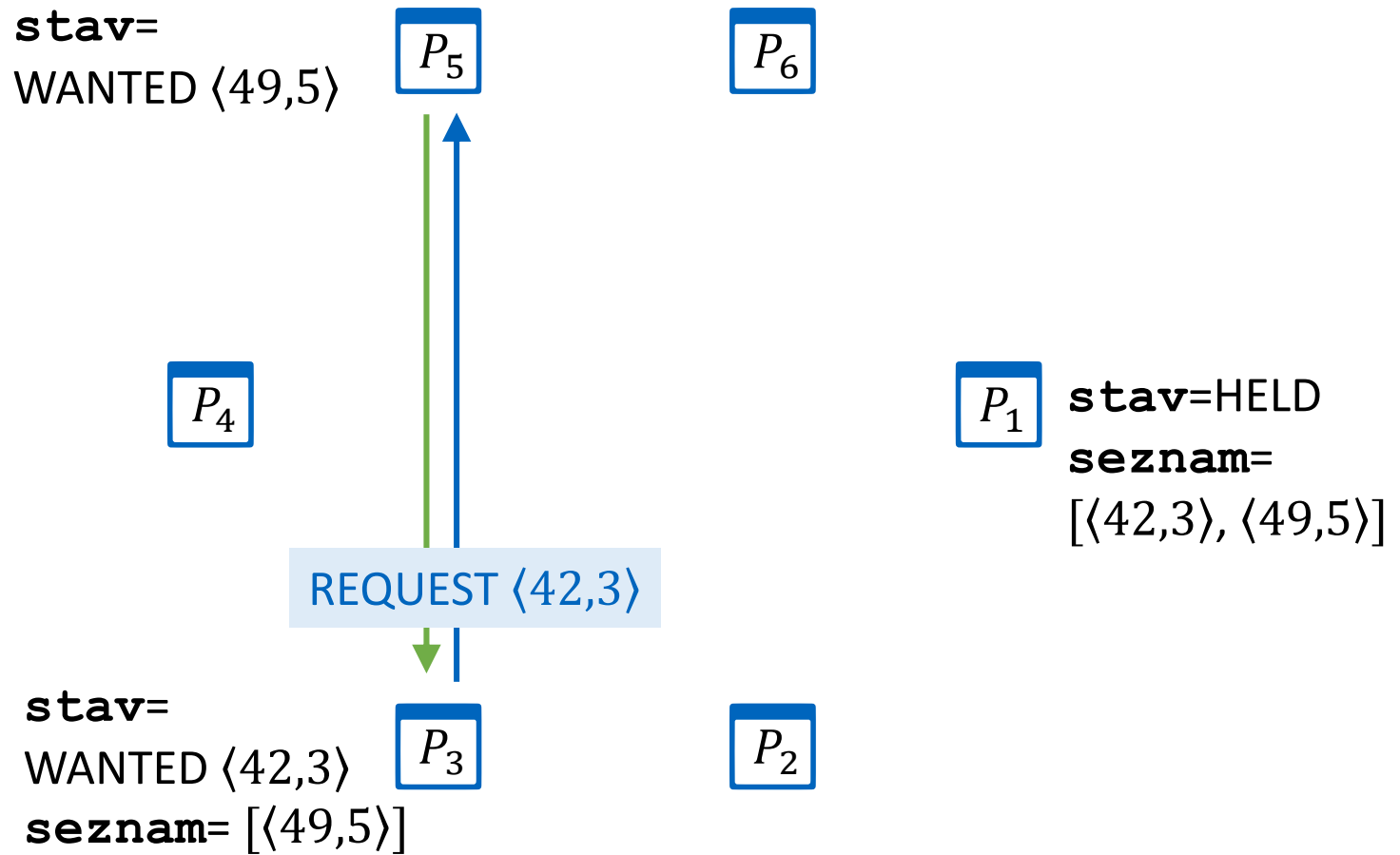
# Příklad



# Příklad



# Příklad





# Příklad

**stav=**

WANTED  $\langle 49,5 \rangle$

(čeká na odpověď  $P_3$ )

$P_4$

$P_5$

$P_6$

$P_1$

**stav:=RELEASED**

Multicast OK  
procesům ze

**seznamu**

$[\langle 42,3 \rangle, \langle 49,5 \rangle]$

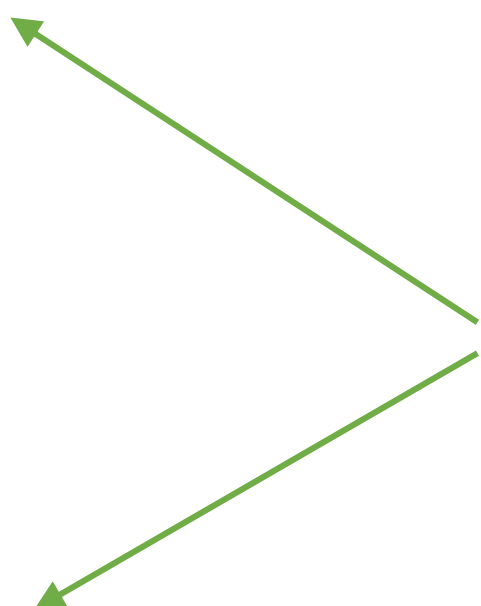
**seznam:=[]**

**stav:=HELD**

**seznam=** $[\langle 49,5 \rangle]$

$P_3$

$P_2$



# Příklad

**stav=**

WANTED  $\langle 49,5 \rangle$

(čeká na odpověď  $P_3$ )

$P_5$

$P_6$

$P_4$

$P_1$

**stav:=RELEASED**

Multicast OK

procesům ze

**seznamu**

$[\langle 42,3 \rangle, \langle 49,5 \rangle]$

**seznam:=[]**

**stav=**

WANTED  $\langle 42,3 \rangle$

**seznam=**  $[\langle 49,5 \rangle]$

$P_3$

$P_2$

# Příklad

**stav=**  
WANTED (49,5)  
(čeká ještě na  
odpověď  $P_3$ )

$P_4$

$P_5$

$P_6$

$P_1$

**stav=RELEASED**

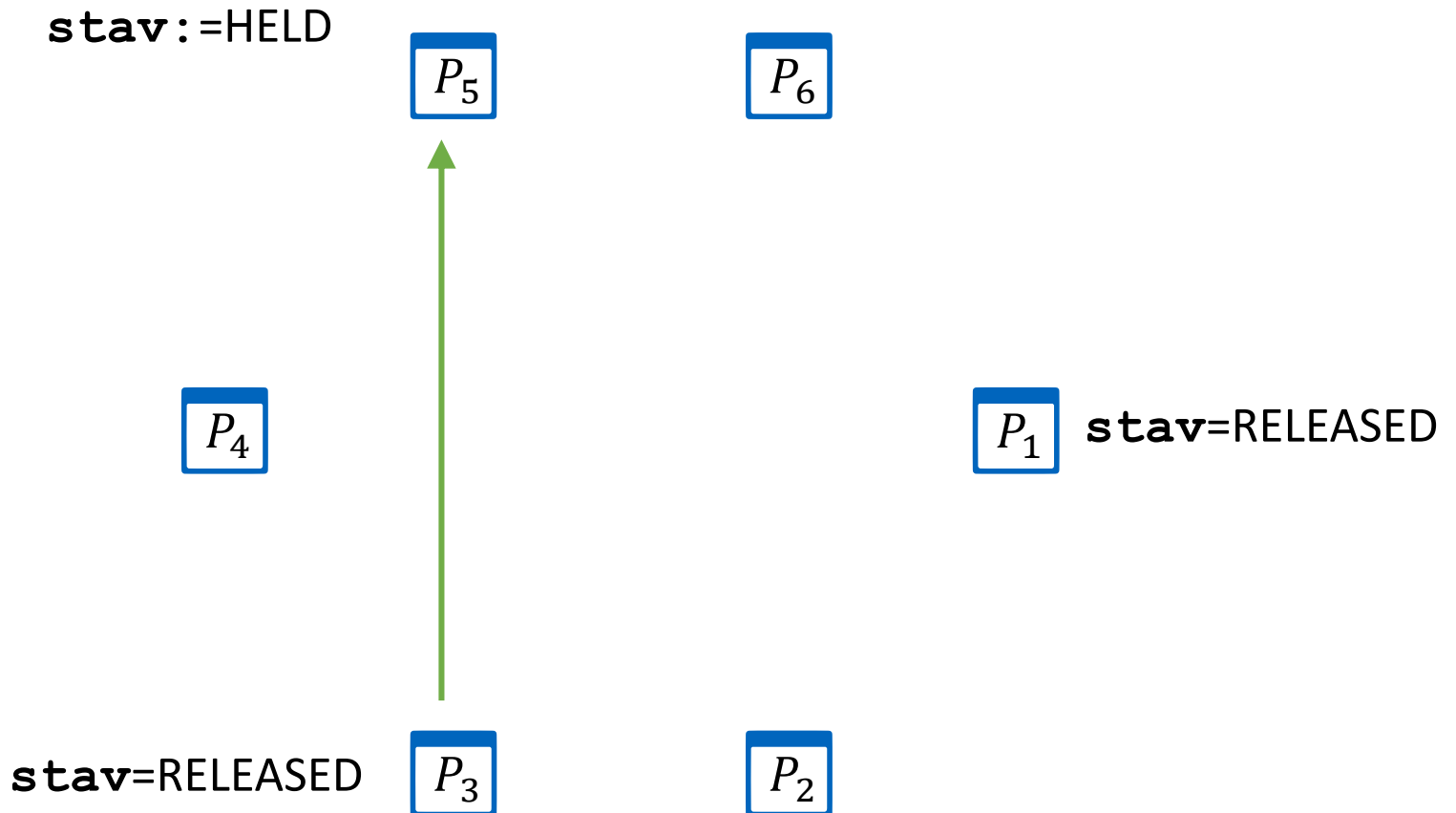
**stav:=RELEASED**  
**seznam:=[]**

$P_3$

$P_2$



# Příklad



# Analýza

**Bezpečnost:** Dva procesy  $P_i$  a  $P_j$  nemohou současně získat přístup do KS

- pokud by získaly, musely by si oba vzájemně poslat OK
- tedy  $\langle T_i, i \rangle < \langle T_j, j \rangle$  a  $\langle T_j, j \rangle < \langle T_i, i \rangle$ , což obojí není možné
- co když  $\langle T_i, i \rangle < \langle T_j, j \rangle$  a  $P_i$  odpověděl na požadavek  $P_j$  předtím, než vytvořil vlastní požadavek?
  - ale: kauzalita a Lamportovy časové značky v  $P_j$  implikují  $T_i > T_j$ , což je spor a tedy tato situace nemůže nastat

# Analýza

**Živost:** nejhorší případ – je potřeba počkat než všech  $(N - 1)$  pošle OK

**Pořadí:** Požadavky s nižší Lamportovou časovou značkou mají přednost

# Analýza

Komunikační zátěž:

- Vstup:  $2 * (N - 1)$   
(resp.  $N$  pokud je k dispozici nativní multicast)
- Výstup:  $N - 1$   
(resp. 1 pokud je k dispozici multicast)

Zpoždění klienta: **1** čas oběhu zprávy

Synchronizační zpoždění: **1** komunikační latence

# Analýza

Ve srovnání s centrálním algoritmem jsme odstranili centrální prvek.

Ale: komunikační zátěž narostla na  $O(N)$

Lze ji dále snížit?

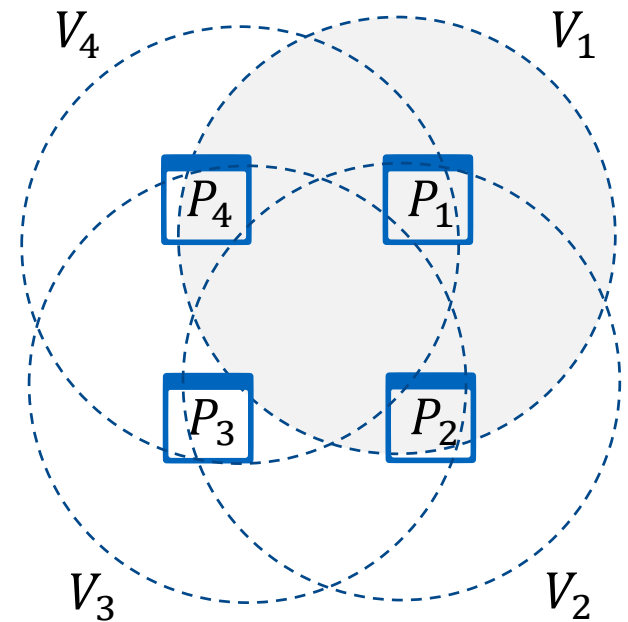


# Maekawův algoritmus

Hlavní myšlenka: pro vstup nepotřebují souhlas všech procesů, ale pouze souhlas všechny z tzv. **volebního okrsku** (voting set).

- Každý proces žádá o vstup pouze procesy ze svého volebního okrsku (tj. ne všechny)
- Každý proces dává svolení nejvýše jednomu procesu (tj. ne všem)

Komunikační složitost Maekawova algoritmu:  $O(\sqrt{N})$



# Souhrn

Distribuované vyloučení procesů je důležitý problém v DS.

Klasické algoritmy: centralizovaný, kruhový, Ricart-Agawala, Maekawa.

Všechny mají zaručenou **bezpečnost, živost a pořadí vstupu**.  
Liší se v komunikační náročnosti a ve zpožděních při vstupu/výstupu a synchronizaci.

Algoritmy se vypořádávají s **asynchronitou**, ale nikoliv se **selháními**.