

PDV 08 2019/2020

Úvod do distribuovaných systémů

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



Paralelní
výpočty

Distribuované
výpočty

Studijní materiály

Slidy

Hlavní kniha: Maarten van Steen, Andrew S. Tanenbaum:
Distributed Systems (3.01 Edition), 2017, k dispozici online:
<https://www.distributed-systems.net/index.php/books/distributed-systems-3rd-edition-2017/>

Sekundární kniha: George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair: **Distributed Systems: Concepts and Design (5th Edition)**, 2011

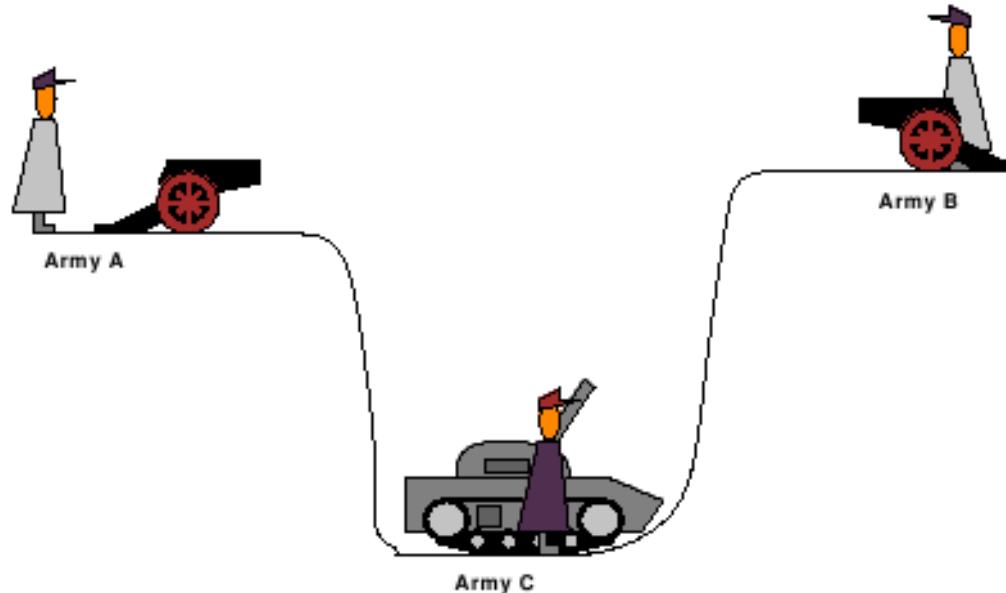


Úvod do Distribuovaných Systémů





Dva generálové



Pouze **synchronizovaný útok uspěje** → generálové se potřebují dopředu **shodnout na čase**, kdy oba zaútočí.

Komunikují skrze **posílání zpráv**. Poslané zprávy se mohou **ztratit**...

Jak zaručit, že zaútočí ve stejný čas?

Dva generálové

Možné řešení:

- Generál A: zaútočit za úsvitu!
- Generál B: potvrzuji, zaútočit za úsvitu!
- Generál A: potvrzuji, potvrzuji, zaútočit za úsvitu!
- Generál B: potvrzuji, potvrzuji, potvrzuji, zaútočit za úsvitu!
- ...

Lze ukázat:

Řešení problému Dvou generálů za těchto předpokladů **neexistuje**!

Reality Check

Příklad: Výběr z bankomatu

- **vyberete** si z bankomatu v Praze 1000 Kč
- ze zůstatku z vašeho účtu se **odečte** 1000 Kč

Problém **atomického commitu**: atomický commit je sada více operací, které jsou provedeny jako jedná operace.

- zúčastněné systémy musí zkoordinovat zda a kdy tyto operace budou provedeny

Pragmatické řešení problému Dvou Generálů



Předpokládejme, že pravděpodobnost chycení kurýra je p a že opakované zachycení kurýra jsou nezávislé jevy.

Možné řešení:

- Generál A pošle **n kurýrů** a **každopádně** zaútočí za úsvitu
- Generál B zaútočí za úsvitu pokud k němu dorazí **aspoň jeden** kurýr

Pravděpodobnost **nekoordinovaného útoku** je p^n .

- snížit pravděpodobnost nekoordinovaného útoku můžeme posláním více kurýru ...
- ... za cenu nutnosti poslat více kurýrů

Jistoty koordinovaného útoku ale dosáhnout nemůžeme
(pokud $p > 0$).

Obecně

Řešitelhost problémů v distribuovaných systémech

- Řada zdánlivě jednoduchých problémů nemá v distribuovaných systémech 100% řešení...
- ...ale mají pragmatická řešení.
- Některé problémy 100% řešení mají...
- ... ale jen za určitých předpokladů.

Pochopit, které problém jsou které, a jak se dají řešit je cílem tohoto předmětu.



Co to je distribuovaný
systém?

Distribuované systémy jsou všude

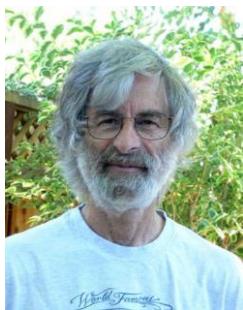
Od té doby, co existují počítačové sítě (natož Internet!) už prakticky **neexistují systémy**, které by **nebyly** aspoň částečně **distribuované**.

Co to je DS?



Definice (optimistická)

Soubor (*collection*) autonomních výpočetních elementů, které se uživateli jeví jako jeden koherentní systém. [Andrew Tanenbaum]



Definice (pesimistická)

Systém, ve kterém selhání počítače, o které jste vůbec nevěděli tušení, že existuje, učiní váš vlastní počítač nepoužitelný. [Leslie Lamport]

Definice (pragmatická)

Soubor (*collection*) nezávislých, autonomních výpočetních elementů propojených komunikační sítí. Výpočetní elementy komunikují formou posílaní zpráv za účelem určité formy spolupráce.

Příklady distribuovaných systémů



Mezibankovní
platby

MMOG

Uber/Lyft/
Liftago

Sensorové
sítě

P2P sítě

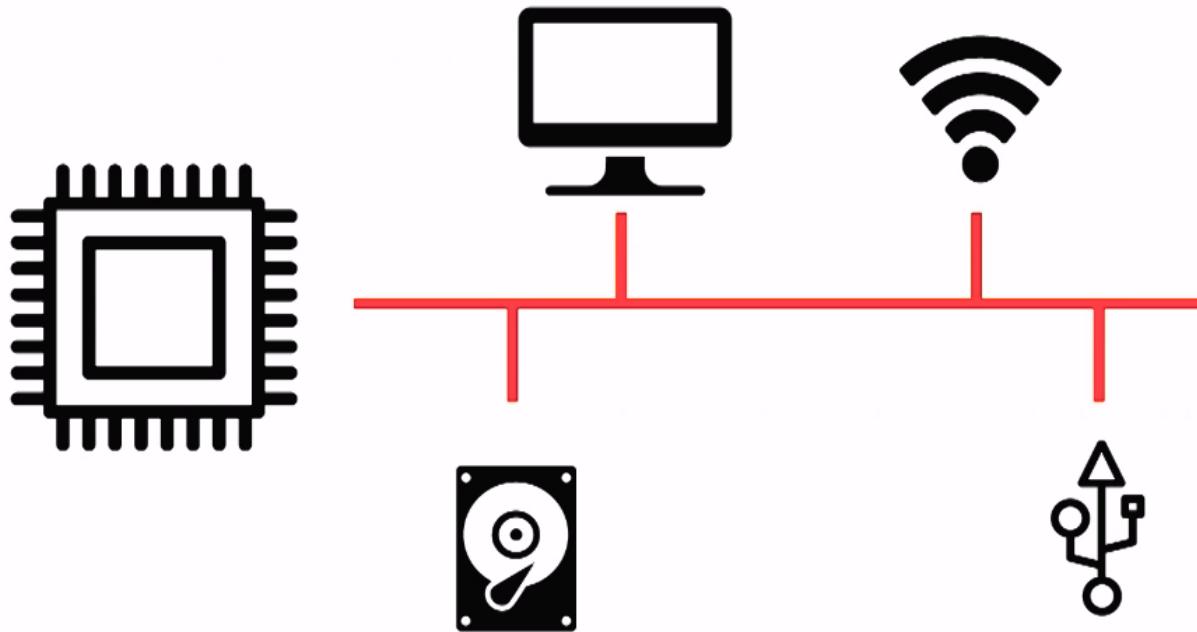
Blockchain

Řízení výrobní
linky

Datové
centrum

Internet

Je počítač distribuovaný systém?



Není* → Proč?

(moderní počítače se jim v lecčems blíží)

Charakteristiky DS

V distribuovaném systému:

1. Počítače pracují **souběžně** (concurrently)
2. Počítače **nesdílejí** globální **hodiny**
3. Počítače **selhávají** **nezávisle**

Naopak:

Paralelní systémy sdílejí

1. **Paměť**: předávání stavu používá sdílenou paměť a synchronizační mechanismy
2. **Hodiny**: přístup ke společným hodinám
3. **Osud**: selže buď nic nebo vše

Cíle při vývoji DS

Výkon/Škálovatelnost

Schopnost řešit více úloh nebo větší instance úloh, než je možné s jedním počítačem

Podobné jako u paralelních výpočtů, ale řeší i případy, když už jeden počítač nestačí.

Spolehlivost/Dostupnost

Schopnost zajistit (téměř) trvalou dostupnost požadovaných služeb

U paralelních výpočtů tento cíl typicky nemáme

zaměření distribuované části PDV

Další cíle: otevřenost, bezpečnost, ...

Nezávislá (a částečná) selhání

Nezávislá selhání **komplikují život vývojářům**

Algoritmy je třeba navrhnut tak, aby byly **robustní vůči selhání**.

n závislých procesů,
pravděpodobnost selhání $p \rightarrow$
dostupnost $(1 - p)^n$



Nezávislá selhání **usnadňují život uživatelům**

Replikace pomocí více nezávislých počítačů **zvyšuje odolnost** vůči selhání

n nezávislých procesů,
pravděpodobnost selhání $p \rightarrow$
dostupnost $1 - p^n$

Nejen odolnost, ale taky např.
výkonnost

Proč existují DS?

Distribuce jako nástroj

Distribuce může být cílenou **volbou v softwarovém návrhu** směřující k splnění specifických **požadavků**: odolnost vůči selhání, zvýšený výkon nebo poměr cena / výkon, nebo minimální QoS.

Např. replikované servery, cloud, ...



Inherentní distribuce

Aplikace vyžadující sdílení zdrojů nebo šíření informace mezi geograficky nebo organizačně **vzdálenými entitami** jsou „**přirozené**“ distribuované systémy.

Např. banka se několika pobočkami, senzorová síť, ...

Hlavní typy DS

DS pro vysoko
výkonné výpočty
(high-performance
computing)

Distribuované
informační systémy

DS pro pervazivní
výpočty/IoT

DS pro spolehlivé a vysoko výkonné výpočty

DS jako nástroj pro zvýšení výkonů, spolehlivosti a ekonomické efektivity.

Model sdílené paměti se na multi-počítačové architektury **nepodařilo rozšířit** → Další škálování možné jen v paradigmatu distribuovaných výpočtů založených na posílání zpráv.



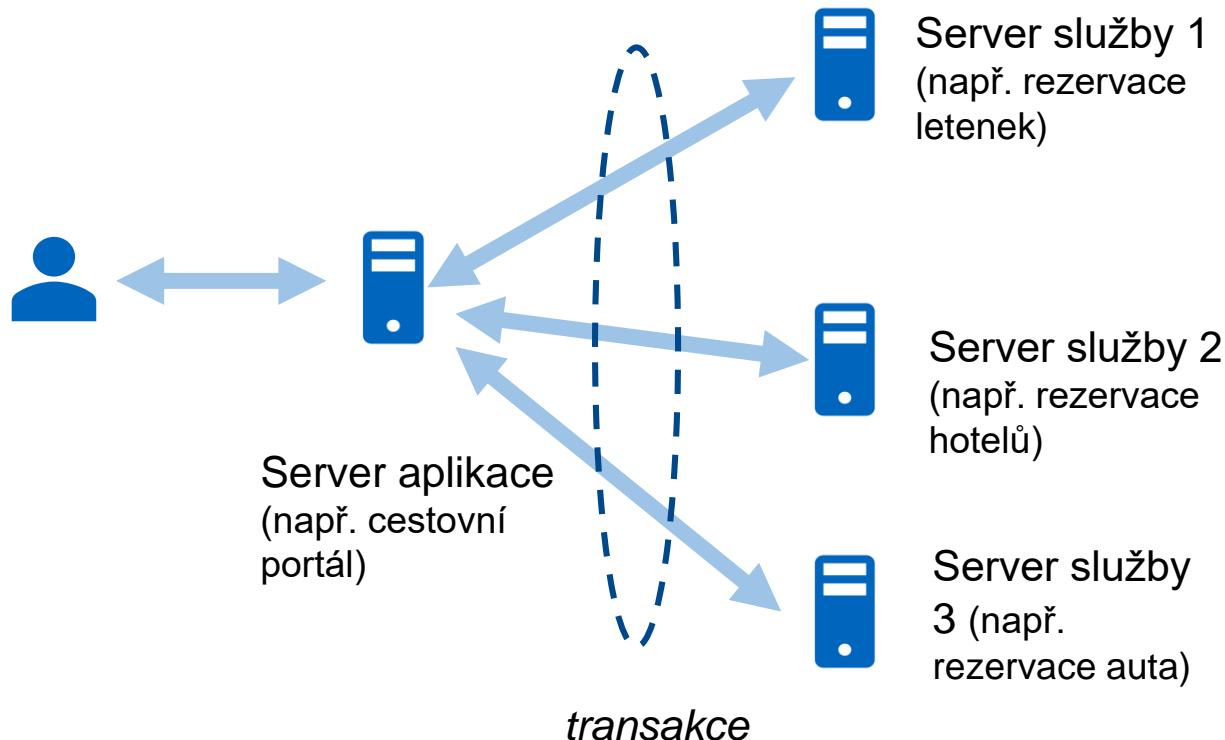
Výkonnost

Výkonnost,
sdílení zdrojů

Sdílení zdrojů,
výkonnost, spolehlivost

Distribuované informační systémy

Distribuce vynucena příslušností jednotlivých výpočetních uzelů do **různých organizací**.

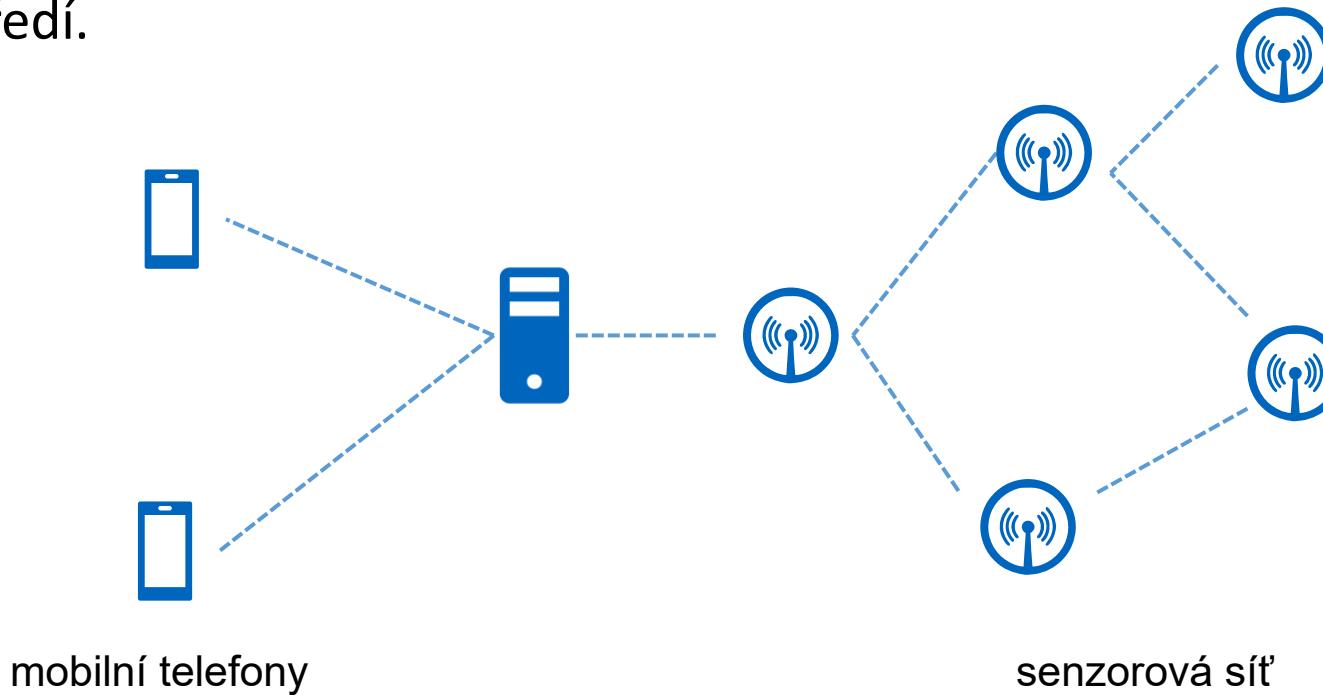


Hlavní výzva: **spolehlivost a konzistence**.

Pervazivní DS/IoT

Malé, mobilní výpočetní uzly, často **vtělené do fyzického prostředí**. Omezení na dosah a rychlosť komunikace, a na spotřebu energie.

Distribuce vynucena nutnosti **fyzické blízkosti** uzlu k uživateli nebo prostředí.



Hlavní výzva: **efektivita, dostupnost,**



Modelování DS

Klíčové elementy DS

Výpočet

Interakce

Čas

Selhání

Výpočet

Proces: jednotka výpočtu v distribuovaném systému (někdy nazýván uzel, hostitel, element, ...)

Množina (skupina) procesů: označována Π – je složena ze souboru N jednoznačně identifikovaných procesů p_1, p_2, \dots, p_N .

Klasické předpoklady DS:

- množina procesů je **konstantní** (N je dobře definováno)
- procesy se navzájem **znají**
- BÚNO: všechny procesy provádějí kopii stejného **algoritmu** (souhrn těchto kopií vytváří distribuovaný algoritmus)

V reálných rozsáhlých (někdy tzv. **extrémních**) distribuovaných systémech tyto předpoklady nemusí být splněny.

Interakce

Procesy komunikují **zasíláním zpráv**

- $send(m, p)$ pošle zprávu m procesu p
- $receive(m)$ přijme zprávu m

Zprávy mohou být v některých případech jednoznačně **identifikovány**

- odesílatelem zprávy
- sekvenčním číslem, které je lokální odesílateli

Klasickým předpoklad: každý pár procesů je propojen **obousměrným** komunikačním **kanálem** (point-point messaging)

- plně propojená topologie může být realizována pomocí směrování (routingu)

Čas

Přesné globální hodiny by umožnily globální uspořádání výpočetních kroků v DS. Bohužel **neexistují**.

Každý proces má své **lokální hodiny**.

Lokální hodiny **nemusí** ukazovat **přesný čas**.

Synchronizce lokálních hodin je možná **jen** s určitou **přesností**.

DS a Čas

V distribuovaných systémech je obtížné uvažovat o čase nejen kvůli absenci globálních hodin, ale také proto, že obecně nelze dát časové **limity** na **komunikaci** a **délku výpočtů**.

Různé možné modely:

- **Asynchronní DS**
- **Synchronní DS**
- **Částečně synchronní DS**

Synchronní vs. Asynchronní

Asynchronní systém

- Žádné časové limity na **relativní rychlosť** vykonávání procesů.
- Žádné časové limity na **trvání přenosu** zpráv.
- Žádné časové limity na **časový drift** lokálních hodin



Synchronní systém

- **Synchronní výpočty:** známe horní limit na relativní rychlosť vykonávání procesů.
- **Synchronní komunikace:** známé horní limit na dobu přenosu zpráv.
- **Synchronní hodiny:** procesy mají lokální hodiny a je znám horní limit na rychlosť driftu lokálních hodin vzhledem k globálním hodinám.

Částečně synchronní DS

Částečná synchronicita: pro většinu systému je relativně snadné definovat časové limity, které platí **většinu času**. Občas se ale mohou vyskytnout období, během kterých tyto časové limity neplatí.

- zpoždění procesů: např. swappování, garbage collection
- zpoždění komunikace: přetížení sítě, ztráta zpráv (vyžadující jejich opakovaný přenos)

Prakticky užitečné systémy **jsou částečně synchronní**

→ Umožňuje **v praxi vyřešit** problémy, které jsou za předpokladu plně asynchronních DS **neřešitelné**.

- některé z časových úseků synchronního běhu DS jsou dostatečně dlouhá na to, aby distribuovaný výpočet skončil

Selhání

Jak procesy, tak komunikační kanály mohou v DS selhat.

Selhání procesu

- **havárie (crash/fail-stop):** proces přestane vykonávat algoritmus (a reagovat na zprávy)
- **libovolné (byzantské) selhání:** proces může pracovat dále (a reagovat na zprávy), ale vykonává chybný algoritmus (z důvodu softwarový chyby nebo úmyslu)

Selhání kanálu

- **ztráta zprávy (message drop):** zpráva není doručena cílovému procesu (např. kvůli přetížení sítě nebo přetečení zásobníku v OS u přijímacího procesu)
- **rozdělení (partitioning):** procesy jsou rozdělené do disjunktních množin (oddílů - partitions) tak, že v rámci oddílu je komunikace možná, ale mezi oddíly nikoliv

V případě synchronních DS definujeme ještě **selhání časování**, pokud doba odezvy procesu nebo přenosu zprávy po síti vybočila z dohodnutého **časového rozmezí**.

Předpoklad na komunikační kanál

Dokonalý
(perfect)
kanál

Spolehlivé doručování: Pokud proces p pošle zprávu procesu q ani p a ani q nehavaruje, pak q nakonec zprávu obdrží.

Žádná **duplicace**: Žádná zpráva není doručena vícekrát než jednou.

Žádné **vytváření**: Je-li zpráva m doručena procesu p , tak zpráva m byla dříve poslána nějakým procesem q procesu p .

Garantované pořadí doručování: Odešle-li proces p procesu q zprávy m_1 a m_2 , tak pokud byla m_1 odeslána dříve než m_2 , tak m_2 nemůže být doručena aniž by předtím byla doručena m_1 .

Spolehlivý
kanál
FIFO kanál

Chybné předpoklady při vývoji DS

Řada DS je **zbytečně komplexních**. Komplexita je způsobena chybami, které je třeba později záplatovat. Chyby vycházejí z **mylných předpokladů**.

Typické mylné předpoklady

- Sítě je spolehlivá
- Sítě je zabezpečená
- Sítě je homogenní
- Topologie sítě se nemění
- Sítě ma nulovou latenci
- Neomezená kapacita sítě
- Systém má jednoho administrátora

Shrnutí

Distribuované systémy jsou všude kolem nás a jejich **význam a složitost** dále roste.

Základním rozdílem mezi paralelními a distribuovanými výpočty jsou: absence **sdílené paměti**, absence **globálních hodin** a nezávislá **selhání**.

PDV 08 2019/2020

Detekce selhání

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



Detekce selhání

Systémy založeny na **skupinách procesů**

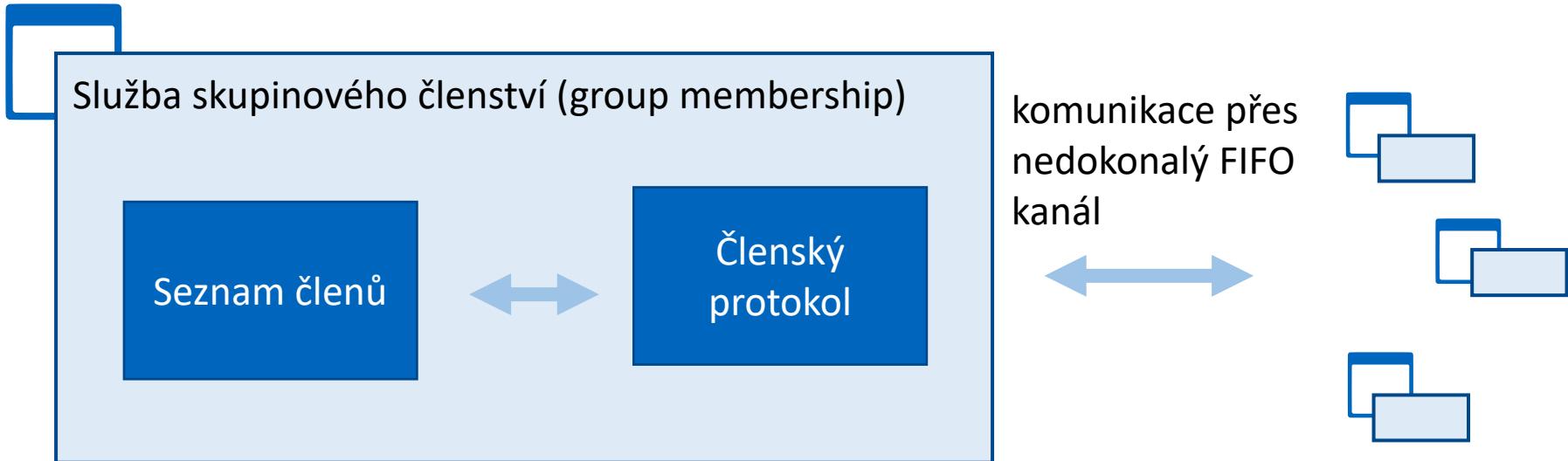
- cloudy / datová centra
- replikované servery
- distribuované databáze

Frekvence selhání roste lineárně s počtem procesů ve skupině
=> selhání jsou *běžná*.

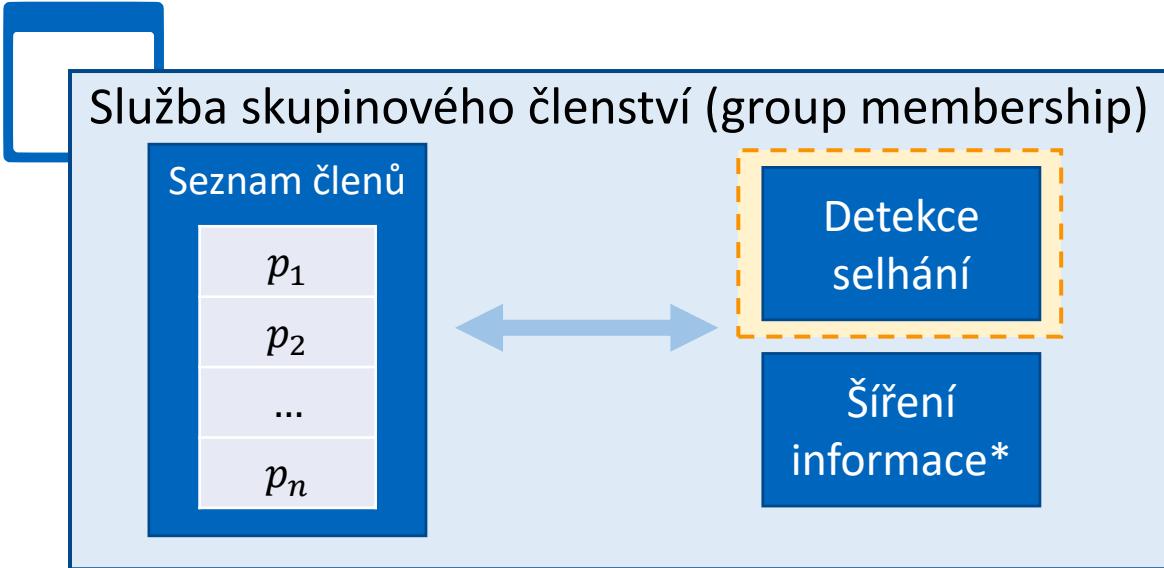
V následujícím předpokládáme **havárii procesu (crash-stop)** jako model selhání a komunikaci přes ztrátový/nedokonalý FIFO kanál.

Obecněji: Správa skupin (group membership)

proces ve skupině

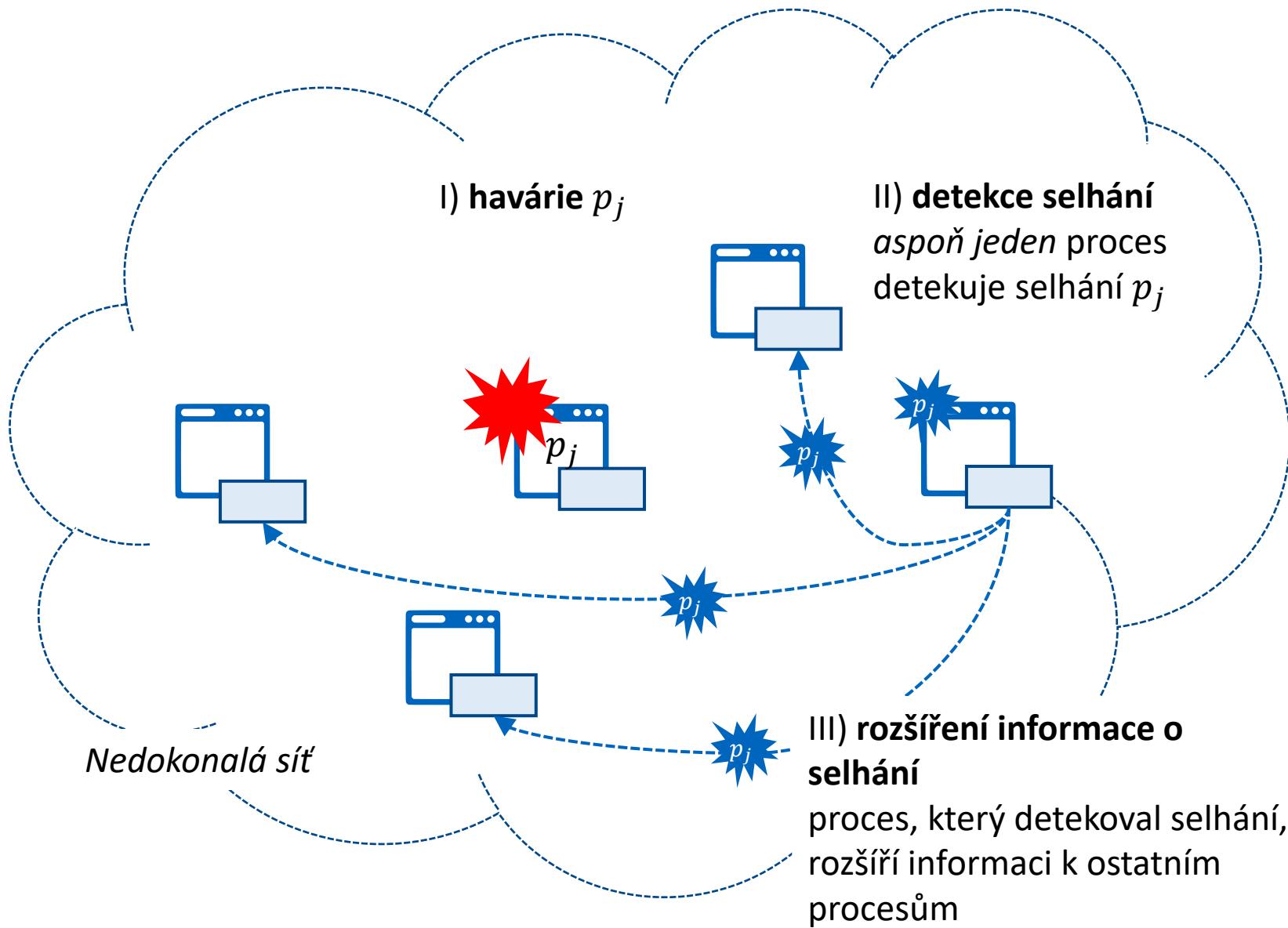


Dva podprotokoly



*Informace o selhání
(a také o vstupu / výstupu procesu do skupiny)

Průběh detekce





Vlastnosti detektorů selhání

Požadované vlastnosti detektorů selhání



Úplnost

= každé selhání je časem **detekováno** aspoň jedním **bezvadným** procesem

Přesnost

= nedochází k mylné **detekci**

Pokud je proces detekován jako havarovaný, tak skutečně havaroval (žádná false positives)



Nelze dosáhnout obou vlastností současně při nedokonalé komunikaci

(detekce selhání je ekvivalentní problému konsensu—procesy se musí shodnout na tom, které procesy jsou bezvadné a které havarované)

Požadované vlastnosti detektorů selhání

100% Úplnost

Úplnost nelze obětovat <= potřebujeme vědět, že proces havaroval

Vždy splněna v prakticky používaných detektorech

<100% Přesnost

Chybně detekované selhání vede ke zbytečným operacím, ale lepší než přehlédnutí havárie

Pouze částečně (pravděpodobnostní garance)

Další vlastnosti detektorů selhání

Rychlosť detekce

= čas do okamžiku, kdy **první** proces detekuje selhání

Škálovatelnost

= **počet posílaných zpráv a rovnoměrné rozložení komunikační zátěže**

Nechceme úzká hrdla a centrální body selhání

Vlastnosti detektorů selhání

Úplnost

Přesnost

Rychlosť

Škálovatelnost



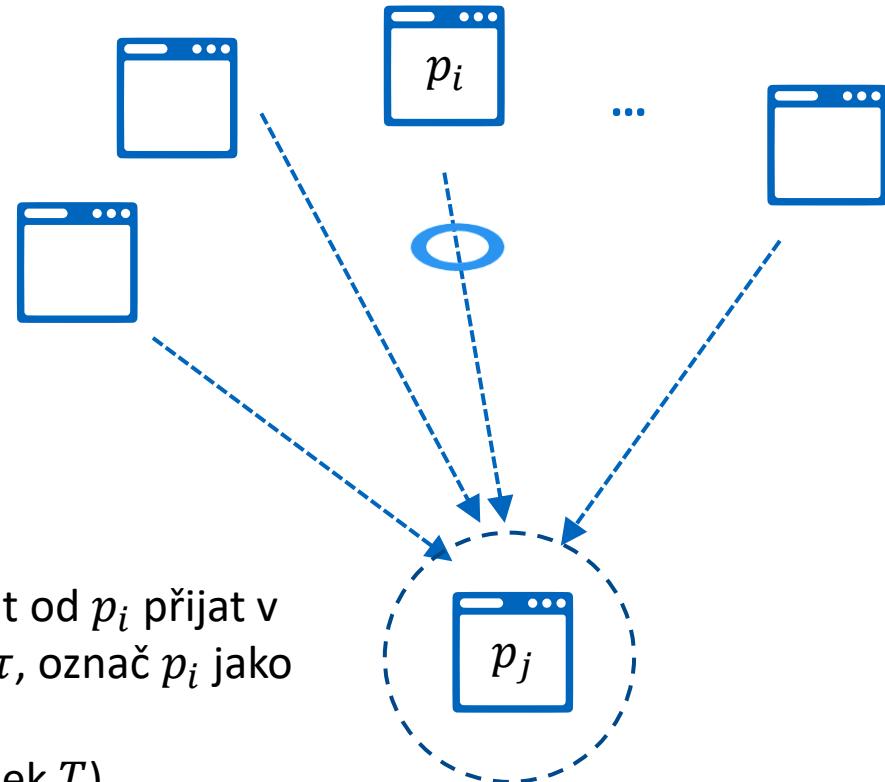
Chceme zajistit i v případě
současného selhání více
procesů



Základní protokoly pro detekci selhání

Centralizovaný heartbeat

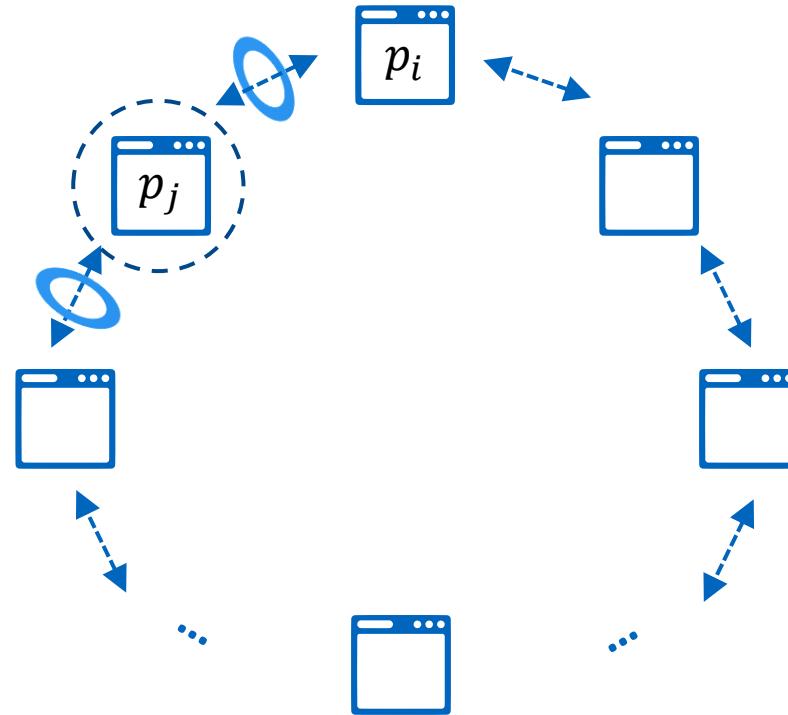
- Hearbeats jsou **odesílány periodicky** (každých T časových jednotek) jednomu vybranému procesu.
- Heartbeat má **pořadové číslo**. (Po odeslání heartbeatů je inkrementován lokální **čítač heartbeatů** každého procesu.)



- 😊 **Úplný** pro všechny procesy s výjimkou p_j
- 😢 Selhání p_j **není detekováno**.
- 😢 Při vysokém počtu procesů může být p_j **přetížený**.

Kruhový heartbeat

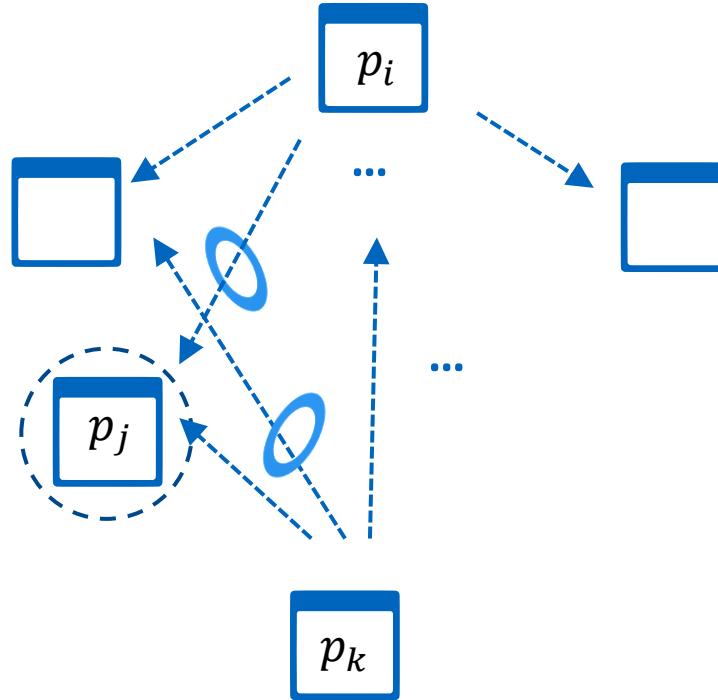
- Heartbeats jsou odesílány periodicky **sousedům každého** procesu (jedno- nebo oboustranně)



- 😊 Není **centrální bod**
- 😢 **Neúplné** při současném selhání více procesů
- 😢 Je třeba udržovat **kruh**

Všichni-všem (all-to-all) heartbeating

- **Každý** proces posílá periodicky heartbeats **všem** ostatním procesům.



- 😊 **Rovnoměrná zátěž** všech procesů (ale poměrně **vysoká**).
- 😊 **Úplný**: pokud zůstane aspoň jede nehavarovaný proces, detekuje selhání libovolného jiného procesu.
- 😢 **Nízká přesnost**: stačí, když jeden proces nedostane včas heartbeats a může označit všechny procesy jako havarované.



Porovnání a analýza detektorů

Porovnání/analýza detektorů

Úplnost

Vždy garantována

(Ne)Přesnost

p_m ... pravděpodobnost, že
dojde k **chybné detekci** během
 T časových jednotek

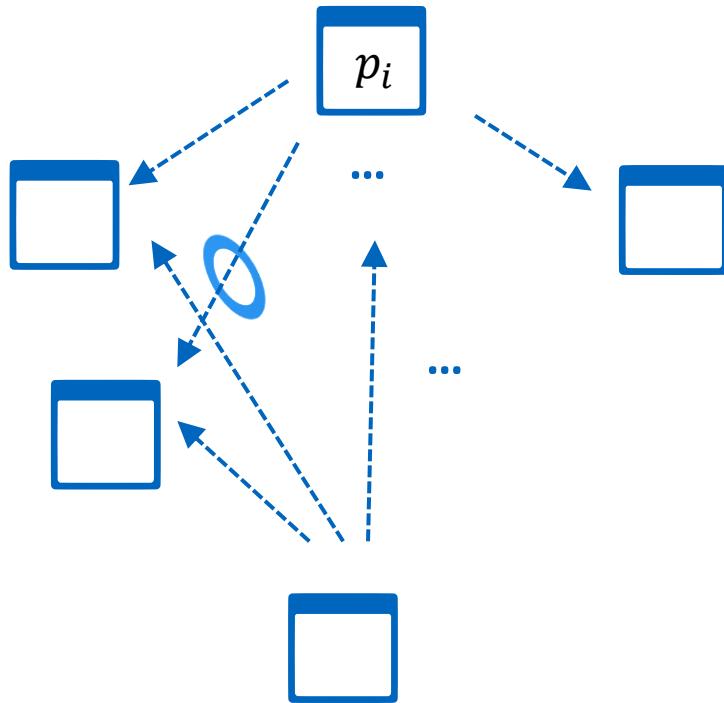
Rychlosť

T_d ... průměrný počet časových
jednotek do **první detekce** selhání

Škálovatelnost

L ... **komunikační zátěž**, tj. počet
zpráv na proces a jednotku času

Analýza all-to-all heartbeating



Počet procesů: N

Perioda heartbeatu: T

Timeout interval: τ

Přesnost: závisí na
spolehlivosti komunikace

**Maximální čas do první
detekce:** $T_d = \tau + T$

Komunikační zátěž: $L = \frac{N}{T}$

Komunikační zátěž je
lineární v počtu procesů.

(celkový počet zpráv v systému roste
kvadraticky s počtem procesů)

Optimální detektor

All-to-all heartbeat má **lineární zátěž** $L = O(N/T)$

- celkový počet posílaných zpráv ve skupině procesů pak roste **kvadraticky**

Proč?

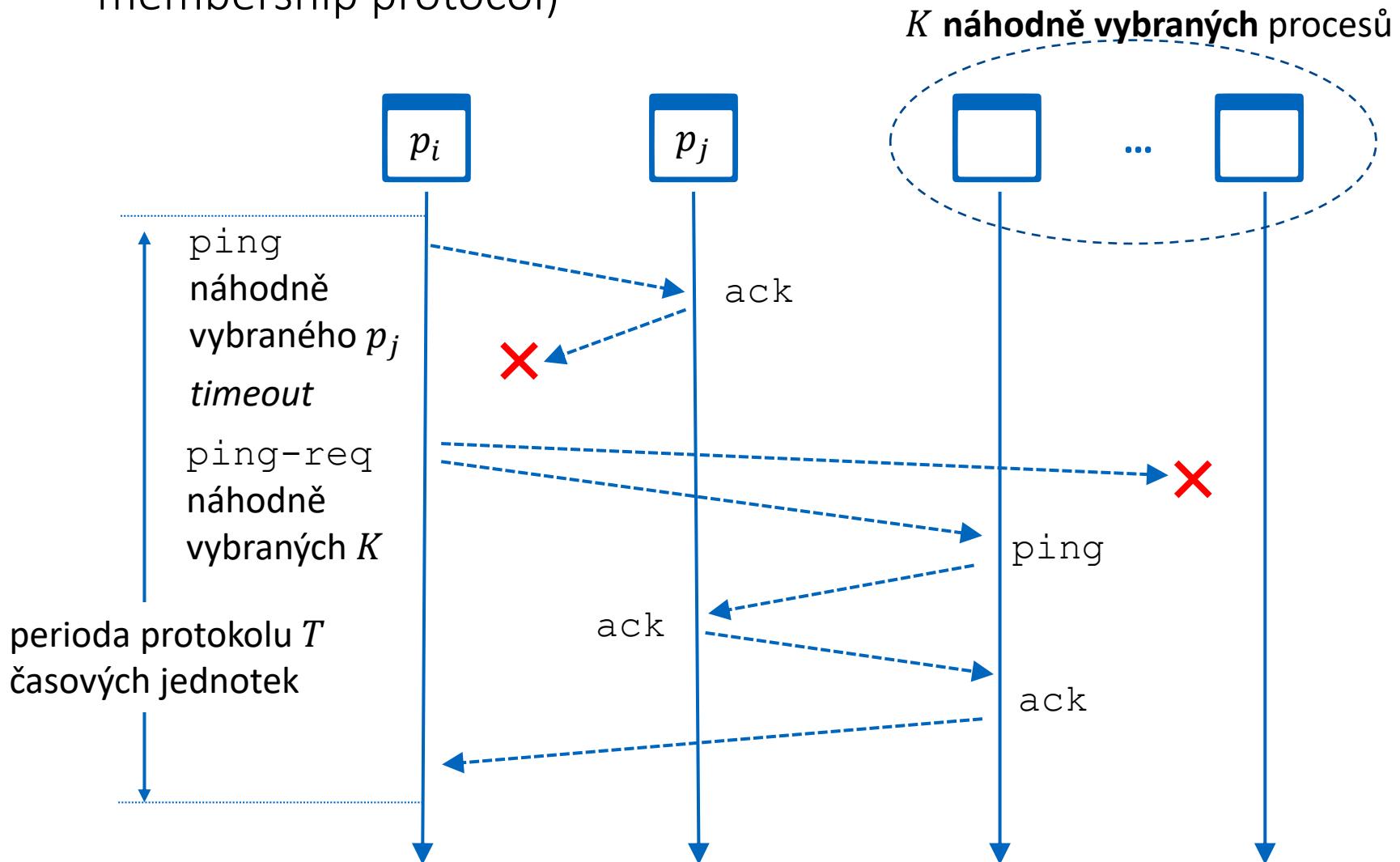
Všechny procesy se snaží detektovat selhání samy, informaci o detekovaném selhání si nevyměňují!

Pro **efektivnější detekci** selhání je potřeba

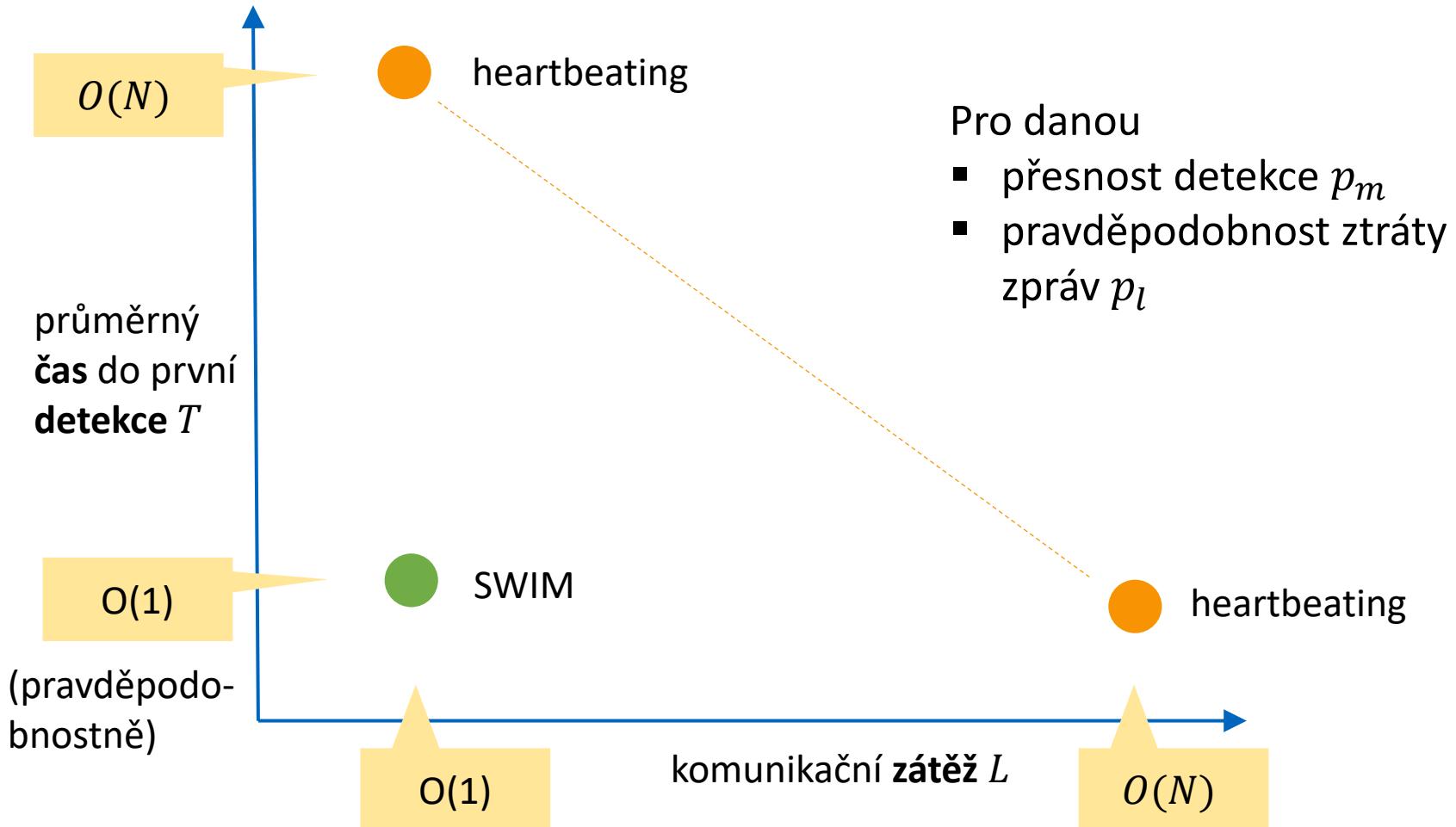
- oddělit detekci selhání a šíření informace o selhání
- použít jinou detekci než založenou na heartbeat (ale na opačném mechanismu)

SWIM Detektor Selhání

(Scalable weakly consistent infection-style proces group membership protocol)



SWIM versus heartbeating



Další vlastnosti SWIM

Průměrný čas do první detekce T_d	<ul style="list-style-type: none">■ nezávisí na počtu procesů■ Lze ukázat, že $E[T_d] = \frac{e}{e-1} T \sim 1.58T$ (pro velká N)
Komunikační zátěž L	<ul style="list-style-type: none">■ nezávisí na počtu procesů
(Ne)přesnost p_m	<ul style="list-style-type: none">■ lze nastavit nastavením K■ klesá exponenciálně s rostoucím K
Úplnost	<ul style="list-style-type: none">■ Selhání bude <i>časem</i> detekováno – průměrný čas $\frac{e}{e-1} T$■ Čas do detekce lze omezit i deterministicky na $O(N)$ (viz dále).

Časově garantovaná úplnost

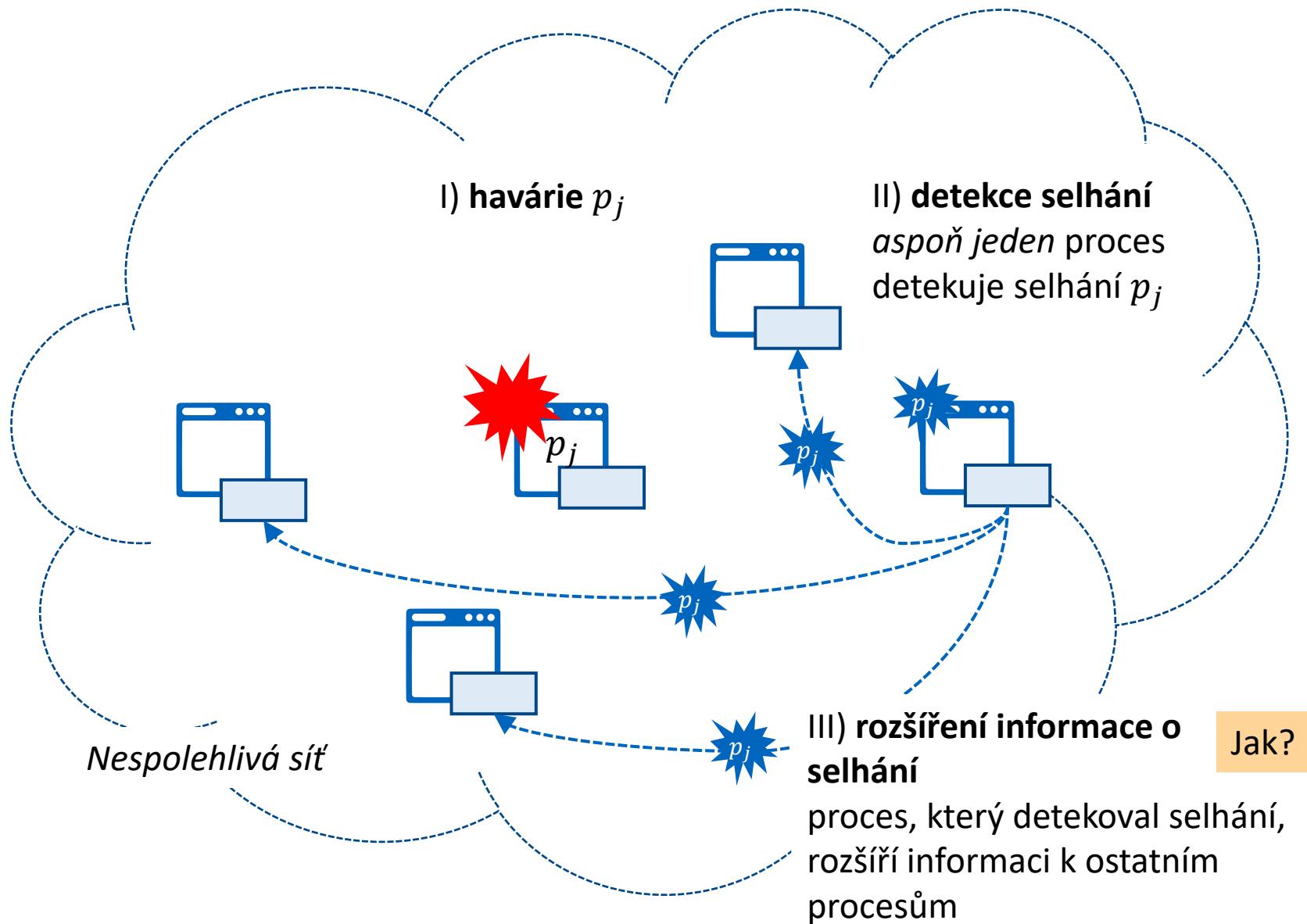
Hlavní myšlenka: Zvolit každý proces ze seznamu **právě jednou** při posílání ping zprávy.

- postupné procházení seznamu
- premutace seznamu po obeslání všech procesů

Každé selhání je detekováno v maximálně $2N - 1$ periodách protokolu

Ostatní vlastnosti SWIM jsou **zachovány**.

Šíření informace



Jak šířit informaci?

Multicast (hardware / IP)

- ne vždy k dispozici

Point-to-point (TCP / UDP)

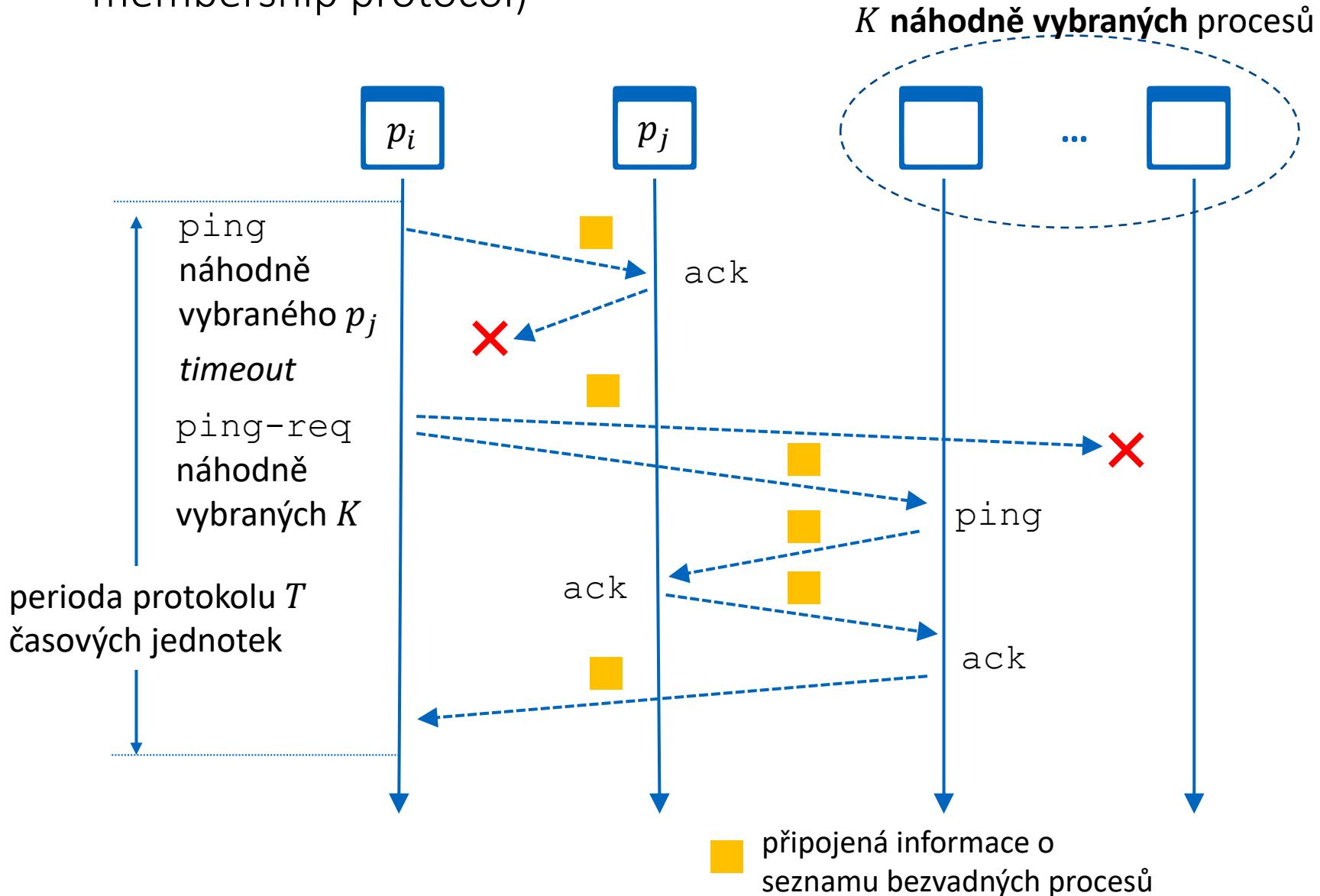
- nákladné

Bez dodatečných zpráv: připojit ke zprávám protokolu detekce selhání!

- tzv. infekční styl šíření informace

SWIM Failure Detector

(Scalable weakly consistent infection-style process group membership protocol)



Vlastnosti šíření informace

Epidemický styl šíření informací: Po $\lambda \log N$ periodách protokolu lze očekávat, že $N^{-2(\lambda-1)}$ procesů nedostalo aktualizaci (update).

Implementace

- každý proces udržuje lokální buffer obdržených aktualizací
- může mít fixní délku nebo fixní dobu expirace
- doba expirace ovlivňuje míru konzistence

Souhrn

Detekce selhání **klíčový stavební prvek** pro spolehlivé DS.

Jednoduchá detekce pomocí **heartbeat** vyžaduje lineární komunikační zátěž každého uzlu.

Pravděpodobnostní SWIM algoritmus je škálovatelný s konstantní zátěží.

PDV 09 2019/2020

Čas a kauzalita v DS

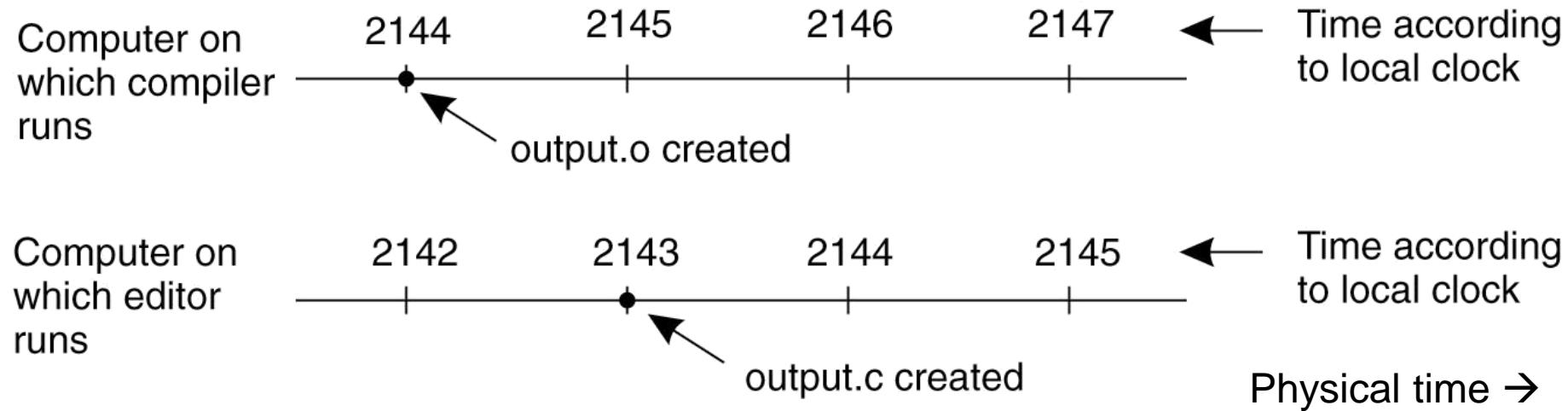
Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



A distributed edit-compile workflow



$2143 < 2144 \rightarrow$ make doesn't call compiler

Lack of time synchronization result – a possible object file mismatch



Synchronizace fyzických hodin



Mimoběžnost vs. Drift

Mimoběžnost hodin (clock skew)

Rozdíl v času hodin dvou procesů.

(jako vzdálenost dvou jedoucích automobilů)

Mají-li dvoje hodiny nenulovou mimoběžnost, jsou **nesynchronizované**.

Drift hodin (clock drift)

Rozdíl v rychlosti (frekvenci) hodin dvou procesů.

(jako rozdíl v rychlosti jedoucích automobilů)

Mají-li dvoje hodiny **nenulový drift**, tak se jejich mimoběžnost v čase bude (nakonec) **zvyšovat**

- jsou-li popředu pomalejší hodiny, tak nejdříve dojde k vyrovnání času



Synchronizace

Uvažujeme skupinu procesů

Externí synchronizace

- Čas C_i hodin každého procesu p_i je udržován v rozmezí δ od času S externích **referenčních hodin**, tj. v každém okamžiku $|C_i - S| \leq \delta$
- Externí hodiny mohou být napojeny na UTC nebo na atomové hodiny
- Algoritmy: např. Cristianův, NTP

Externí synchronizace s rozmezím δ **implikuje** interní synchronizaci s rozmezím $2 * \delta$.

Interní synchronizace

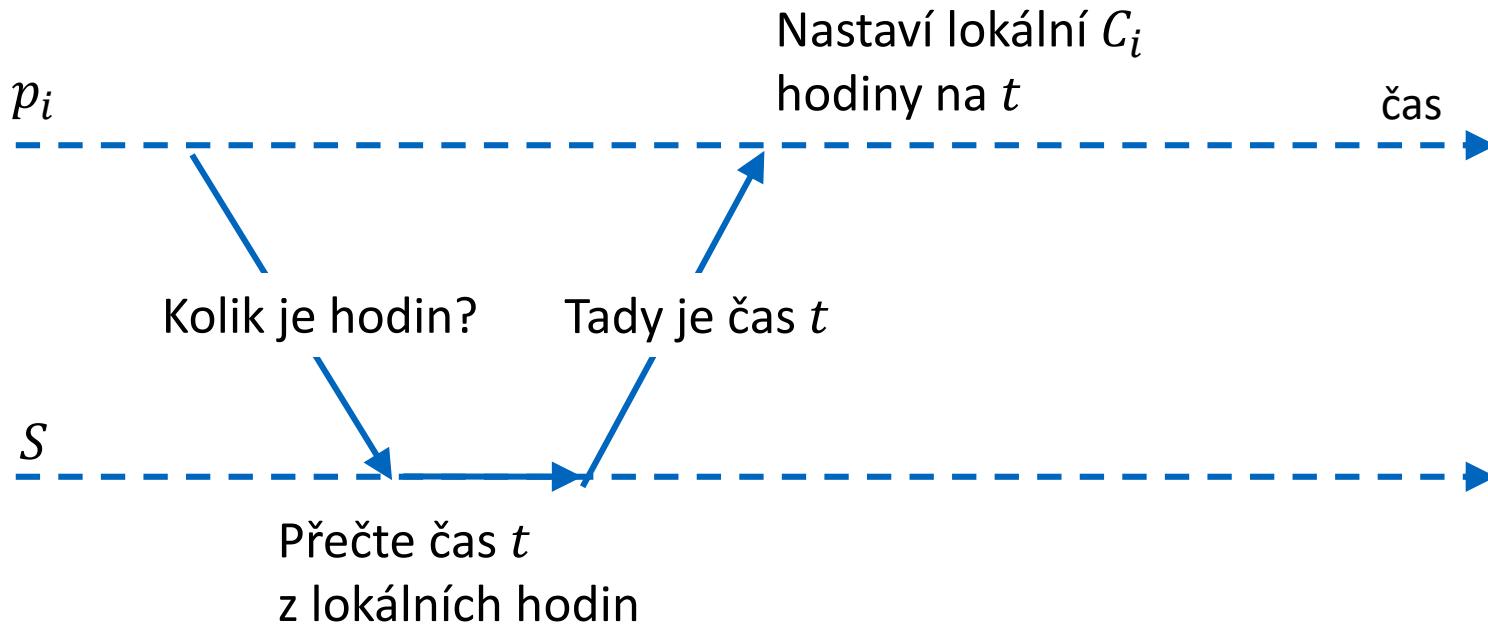
- Každý pár procesů (p_i, p_j) má hodnoty času svých hodin v rozmezí δ , v každém okamžiku tj. $|C_i - C_j| \leq \delta$
- Algoritmy: např. Berkeley

Interní synchronizace **neimplikuje** externí synchronizaci.



Synchronizace fyzických hodin

Externí synchronizace: všechny procesy p_i se synchronizují s externím časovým serverem S .

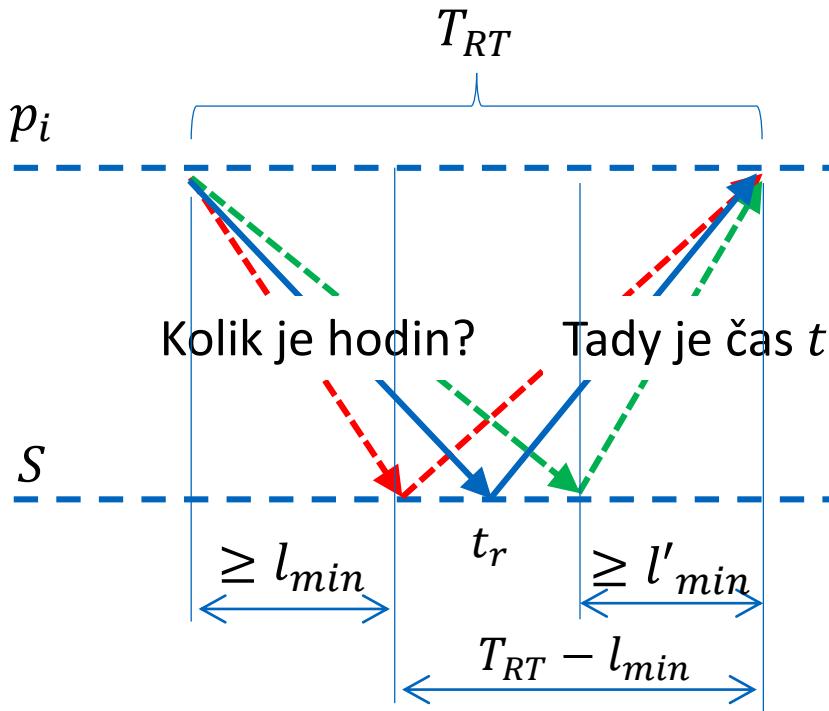


Co je špatně?

- V okamžiku, kdy p_i obdrží odpověď, se už **čas posunul**.
- Míra nepřesnosti C_i závisí na komunikační **latenci**.
- V asynchronním systému je latence **konečná**, ale **neomezená (a neznámá!)** ⇒ neomezená je i **nepřesnost hodin**.



Cristianův Algoritmus



Skutečný čas v okamžiku, kdy
 p_i přijme odpověď je

$$[t_r + l'_{min}, t_r + T_{RT} - l_{min}]$$

t_r ... skutečný čas, kdy S
odpověděl na dotaz

T_{RT} ... změřený čas oběhu (round trip time) synchronizační zprávy
 l_{min}, l'_{min} ... minimální latence při komunikaci směrem k externím hodinám, resp. od externích hodin

Cristianův algoritmus:

$$C_i := t + \frac{T_{RT} - l_{min} + l'_{min}}{2}$$

Chyba je omezena, tj. maximálně
 $(T_{RT} - l_{min} - l'_{min})/2$

(Je-li chyba měření příliš vysoká, je možné poslat více zpráv a výsledek průměrovat.)



Pozor

Lokální čas je možné posunout libovolně **dopředu**...

...ale **nikoliv dozadu!**

- posun dozadu by mohl narušit lokální uspořádání události v procesu

Je možné zvýšit nebo snížit **rychlosť** hodin



Jak často synchronizovat?

Maximální rychlosť driftu (maximum drift rate *MDR*) hodin

Absolutný ***MDR*** je definován relativne vůči UTC (universal coordinated time). UTC je **správny (presný)** čas v každém okamžiku.

Vzájemná maximální rychlosť driftu mezi dvěma procesy se stejnou absolutní *MDR* je $2 * MDR$.

Je-li maximální tolerovaná mimoběžnost mezi jakýkoliv párem hodin je M , pak je třeba hodiny synchronizovat aspoň každých $M/(2 * MDR)$ časových jednotek.



Network Time Protocol (NTP)

Využíván od roku 1985 pro synchronizaci času v počítačových sítích s proměnlivou latencí.

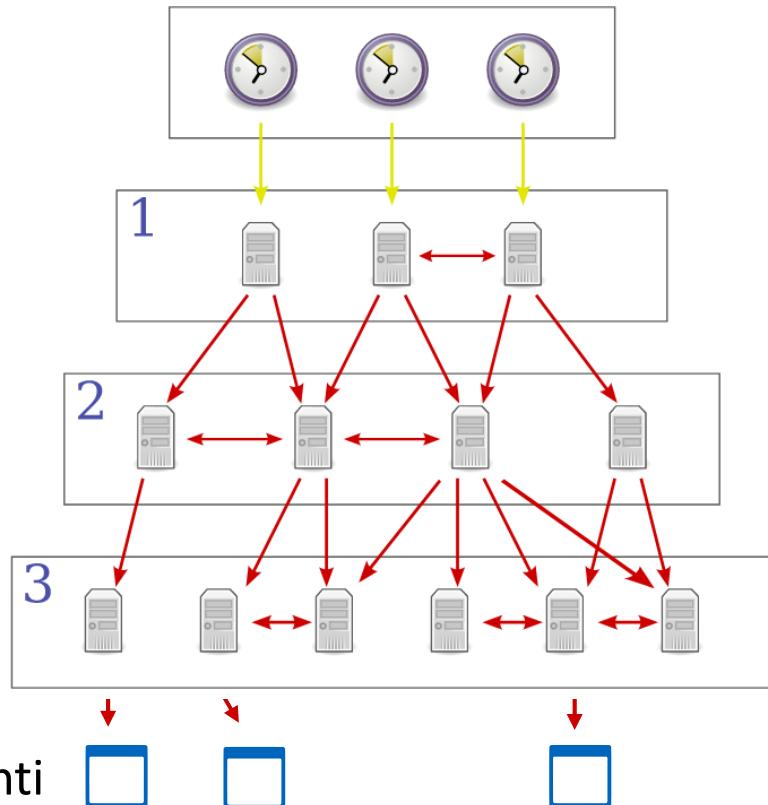
NTP servery jsou uspořádány do **stromu**

- uzly synchronizují se svými rodiči a někdy i dalšími servery na stejné úrovni
- klienti tvoří listy stromu

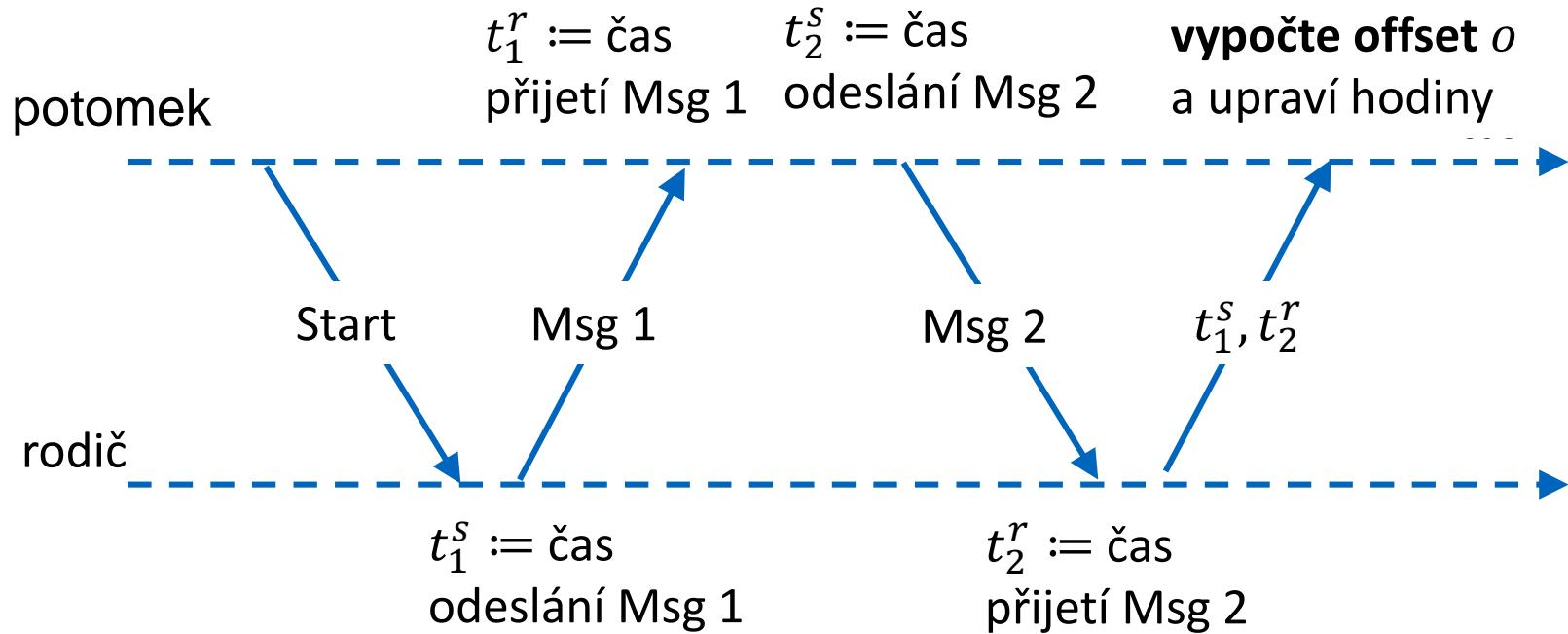
Dosažitelná přesnost:

- internet: **desítky ms**
- LAN: **1 ms**

→ přímé spojení
→ síťové spojení



Základ protokolu



Potomek spočítá **offset** mezi svým časem a časem rodiče:

$$o = \frac{(t_1^r - t_2^r + t_2^s - t_1^s)}{2}$$



Odvození

Proč $o = \frac{(t_1^r - t_2^r + t_2^s - t_1^s)}{2}$?

Předpokládejme skutečný offset je o_{real} , tj. čas potomka je popředu o o_{real} a čas rodiče je pozadu o o_{real} .

Předpokládejme, že latence pro zprávu 1 je l_1 a pro zprávu 2 je l_2 .

- hodnoty l_1 a l_2 **nejsou známé**

Pak platí

- $t_1^r = t_1^s + l_1 + o_{real}$
- $t_2^r = t_2^s + l_2 - o_{real}$

$$o_{real} = \frac{(t_1^r - t_2^r + t_2^s - t_1^s)}{2} + \frac{l_2 - l_1}{2}$$

$$o_{real} = o + \frac{(l_2 - l_1)}{2}$$

l_1 a l_2 jsou
nezáporné

$$|o - o_{real}| \leq \frac{l_2 + l_1}{2}$$

tj. chyba je omezená časem oběhu zprávy



Nicméně

Stále **nenulová** chyba synchronizace.

Dokud bude **latence nenulová a neznámá**, chyby se nezbavíme!

Pomocí fyzických hodin lze uspořádávat jen události v „**pomalých**“ distribuovaných výpočtech

- tj. když intervaly mezi kroky výpočtu trvají výrazně déle než je mimoběžnost hodin
- pro standardně rychlé výpočty bychom potřebovali synchronizaci hodin řádově v nanosekundách

V praxi jsou ale výpočty řádově rychlejší → lze problém uspořádání události vyřešit **bez potřeby synchronizovat** fyzické hodiny?



Motivation: Multi-site database replication

A New York-based bank wants to make its transaction ledger database **resilient** to **whole-site failures**

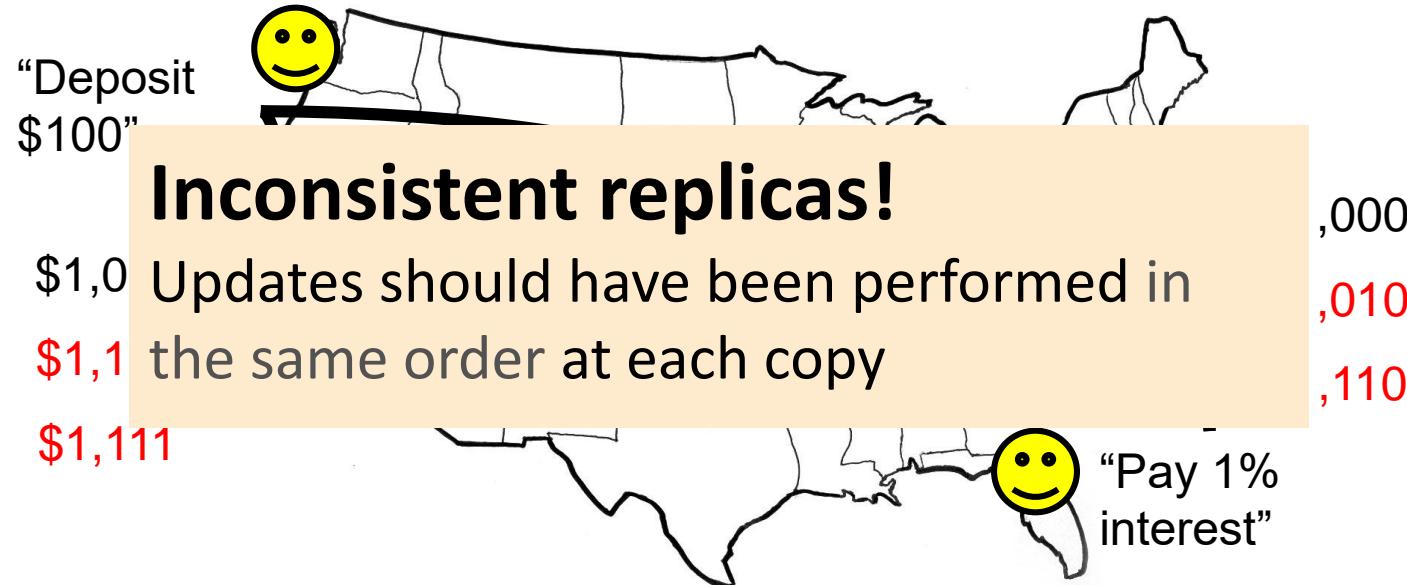
Replicate the database, keep one copy in sf, one in NYC



The consequences of concurrent updates

Replicate the database, keep one copy in sf, one in nyc

- Client sends **query requests to the nearest copy**
- Client sends **update requests to both copies**



Logické hodiny



Uspořádání událostí v DS

Synchronizace fyzických hodin je jeden z přístupů k uspořádání událostí v DS. Vzhledem k omezené přesnosti synchronizace lze ale použít jen uspořádání událostí, mezi kterými uplyne **dostatečné množství času**.

Alternativní přístup: Co kdybychom místo absolutního/fyzického času přiřazovali událostem **logické časové značky**?

- Pokud by přiřazení logických značek respektovalo **kauzální vztah** mezi událostmi, tak by fungovalo.

Kauzální vztah mezi událostmi: první událost může ovlivnit druhou.



Model

Uvažujeme **asynchronní DS** sestávající z **procesů**.

Každý proces má **stav** (hodnoty **proměnných**).

Každý proces vykonává **akce**, aby změnil svůj stav. Akce může být **instrukce** nebo **poslání zprávy (send, receive)**.

Událost je výskyt akce. Poslání zprávy generuje dvě událost: **odeslání** a **přijetí**.

Každý proces má **lokální hodiny**.

Události v rámci procesu mohou mít přiřazeny **časové značky** (timestamps), a mohou tak být linerárně **seřazeny**.

- Ale my potřebujeme řadit globálně v celém DS.

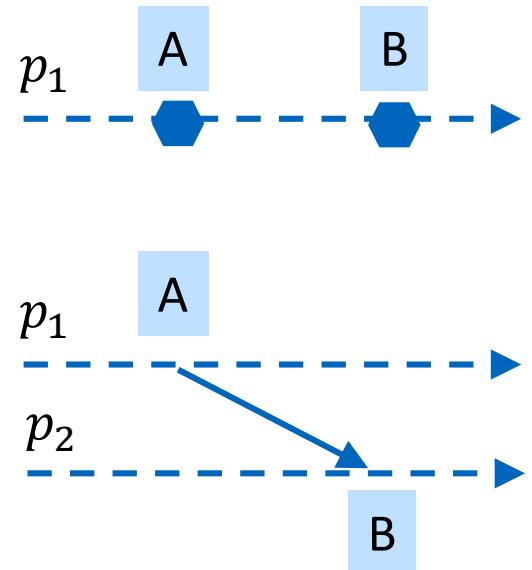


Relace „stalo se před“

Potenciální kauzalitu zachycujeme pomocí relace „stalo se před“

Definice (Relace \rightarrow *stalo se před*)

- Jsou-li A a B události ve stejném procesu p a pokud A předchází B , pak $A \rightarrow B$.
- Je-li A odeslání zprávy a B je přijetí této zprávy, pak $A \rightarrow B$.
- Pokud $A \rightarrow B$ a $B \rightarrow C$, pak $A \rightarrow C$



Kauzální závislost/nezávisost

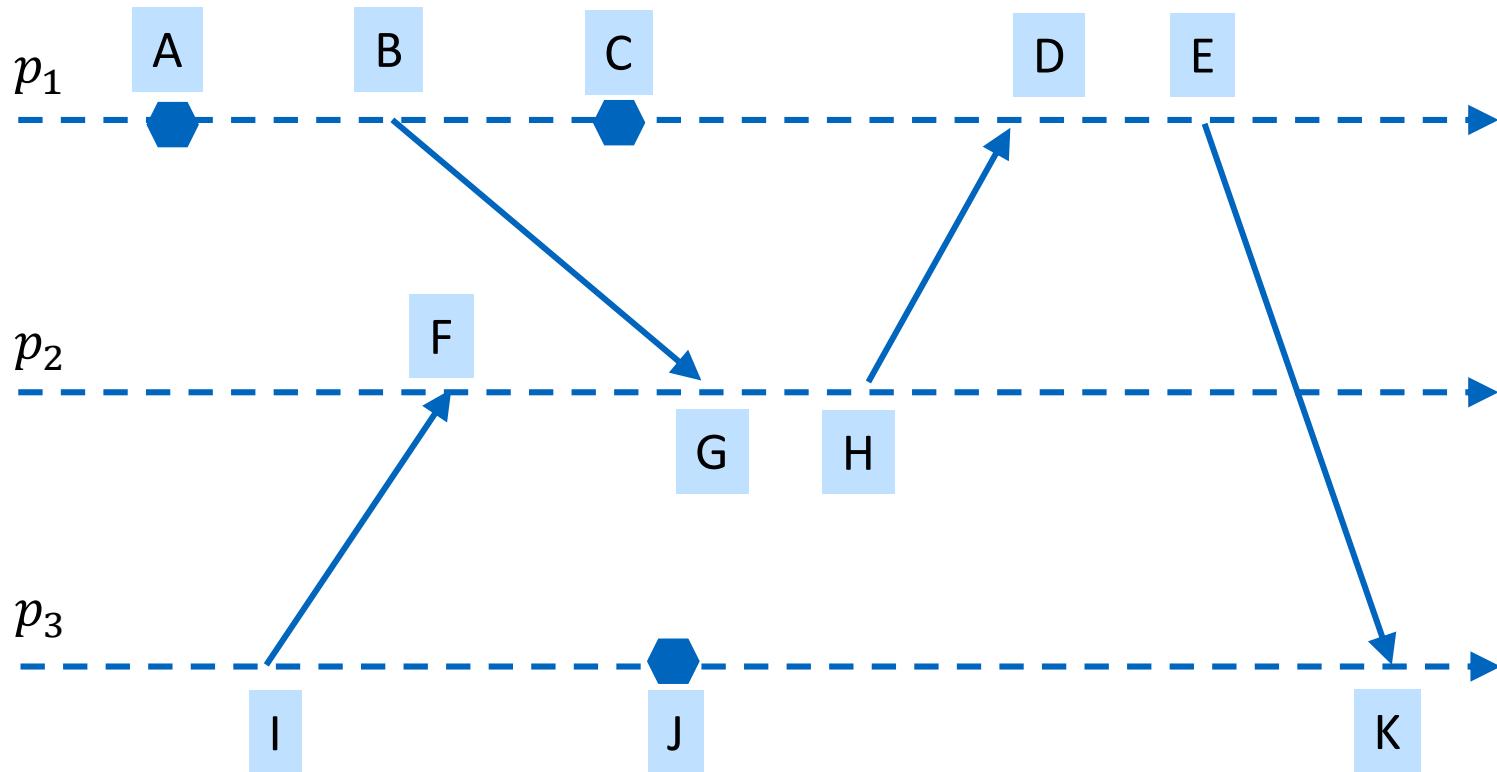
Relace *stalo se před* zavádí **částečné uspořádání událostí** → potenciální **kauzální závislost**

$e_1 \rightarrow e_2$: potenciálně **kauzálně závislé** události
(může být kauzální vztah, tj. e_1 *mohla* ovlivnit e_2 – ale *nemusela*)

$e_1 \parallel e_2$: **současné** události
(kauzální vztah *určitě* není, tj. e_1 *nemohla* ovlivnit e_2 a e_2 *nemohla* ovlivnit e_1)



Příklad: Stalo se před →



$A \rightarrow B$	$I \rightarrow F$	$C \parallel G$
$B \rightarrow G$	$F \rightarrow K$	$A \parallel J$
$A \rightarrow G$	$A \rightarrow K$	$C \parallel H$
	$C \rightarrow K$	

◆ instrukce
→ zpráva

Lamportovy logické hodiny

Jak přiřadit události e časovou značku $C(e)$ tak, aby **respektovaly kauzalitu**, tj. jestliže $e_1 \rightarrow e_2$, pak $C(e_1) < C(e_2)$?

→ **Lamportovy logické hodiny:** Každý proces má své logické hodiny, které se **synchronizují podle přijímaní zpráv**.

- navržené Leslie Lamportem v 70. letech
- používané prakticky ve všech distribuovaných systémech (a všech cloudových platformách)

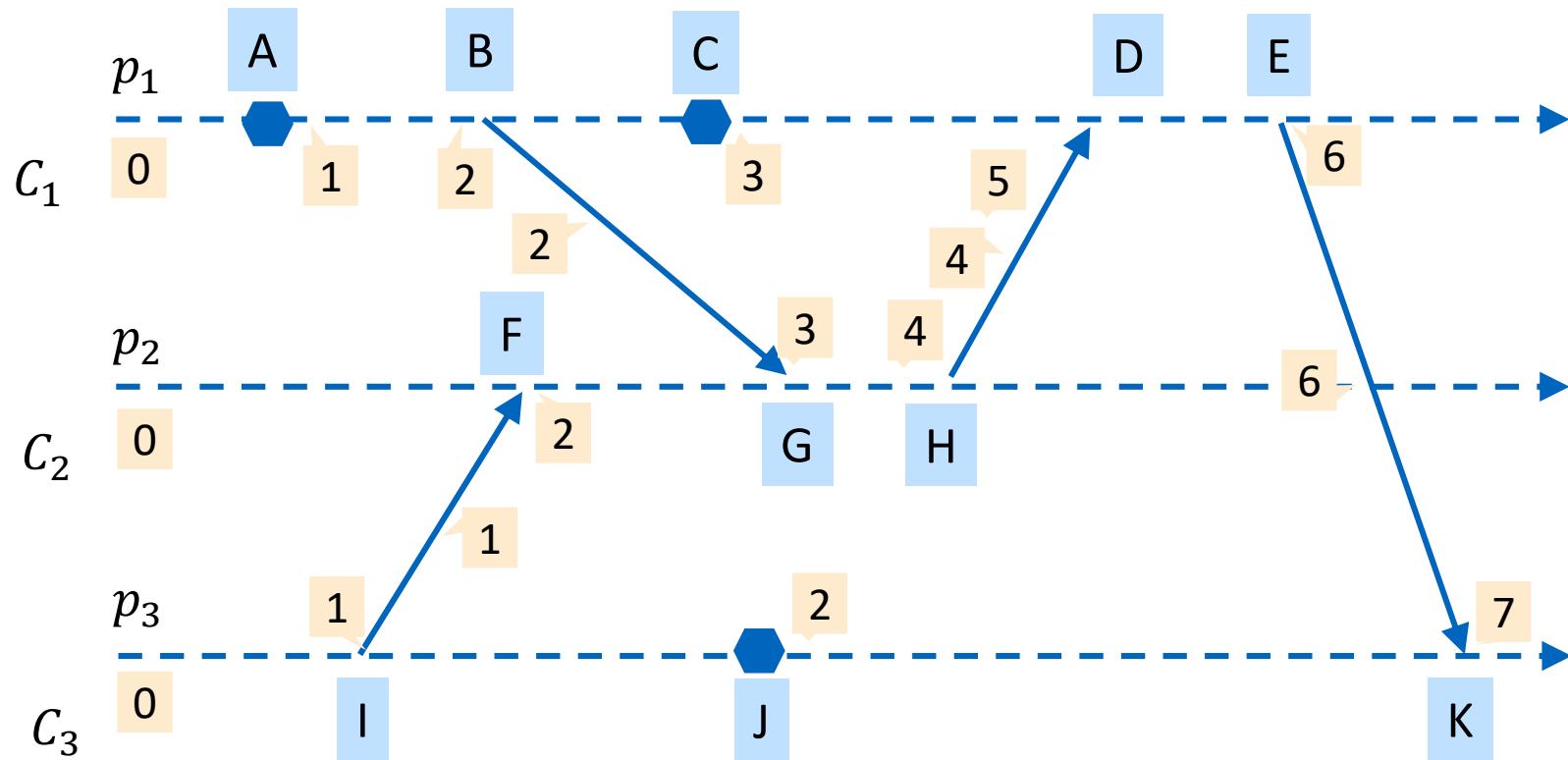
Synchronizace logických hodin

Synchronizace logických hodin

Každý proces p_i si udržuje lokální logické hodiny C_i a nastavuje

1. Po každé události, která se odehraje v p_i , se C_i **inkrementuje** o 1.
2. Každé zprávě m **odeslané** procesem p_i je přiřazena časová značka $ts(m) = C_i$.
3. Kdykoliv proces p_j přijme zprávu m , tak
 - I. upraví své lokální hodiny C_j na $\max\{C_j, ts(m)\}$; a poté
 - II. provede krok 1 předtím, než předá m aplikaci
(\Leftarrow přijetí zprávy je událost)

Příklad: Lamportovy hodiny



$A \rightarrow B \quad 1 < 2 \quad G \rightarrow H \quad 3 < 4$

$B \rightarrow G \quad 2 < 3 \quad F \rightarrow K \quad 2 < 7$

$A \rightarrow G \quad 1 < 3 \quad H \rightarrow K \quad 4 < 7$

$C \rightarrow K \quad 3 < 7$

$C \parallel G? \quad 3 = 3$

$A \parallel J? \quad 1 < 2$

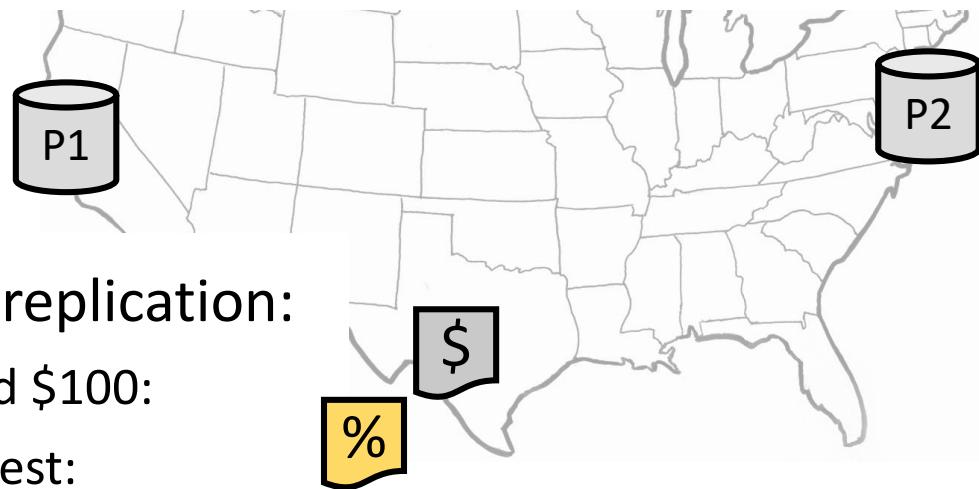
$C \parallel H? \quad 3 < 4$

Lamportovy hodiny
neimplikují kauzalitu!

Making concurrent updates consistent

Recall multi-site database replication:

- San Francisco (**P1**) deposited \$100:
- New York (**P2**) paid 1% interest:



We reached an inconsistent state

Could we design a system that uses Lamport Clock total order to make multi-site updates consistent?

Totally-Ordered Multicast

Goal: All sites apply updates in (same) Lamport clock order

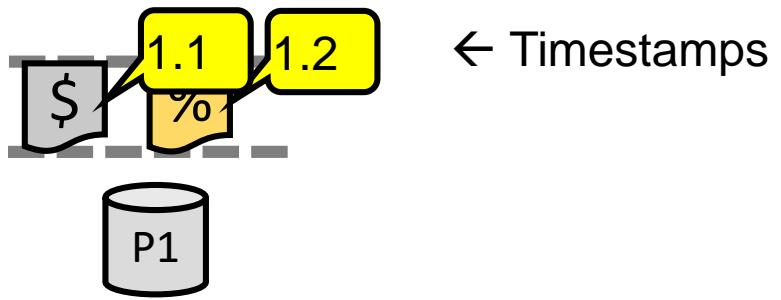
Client sends requests to **one** replica site j

- Replica **assigns** it Lamport timestamp $C_j.j$

Key idea: Place events into a sorted **local queue**

- **Sorted** by increasing Lamport timestamps

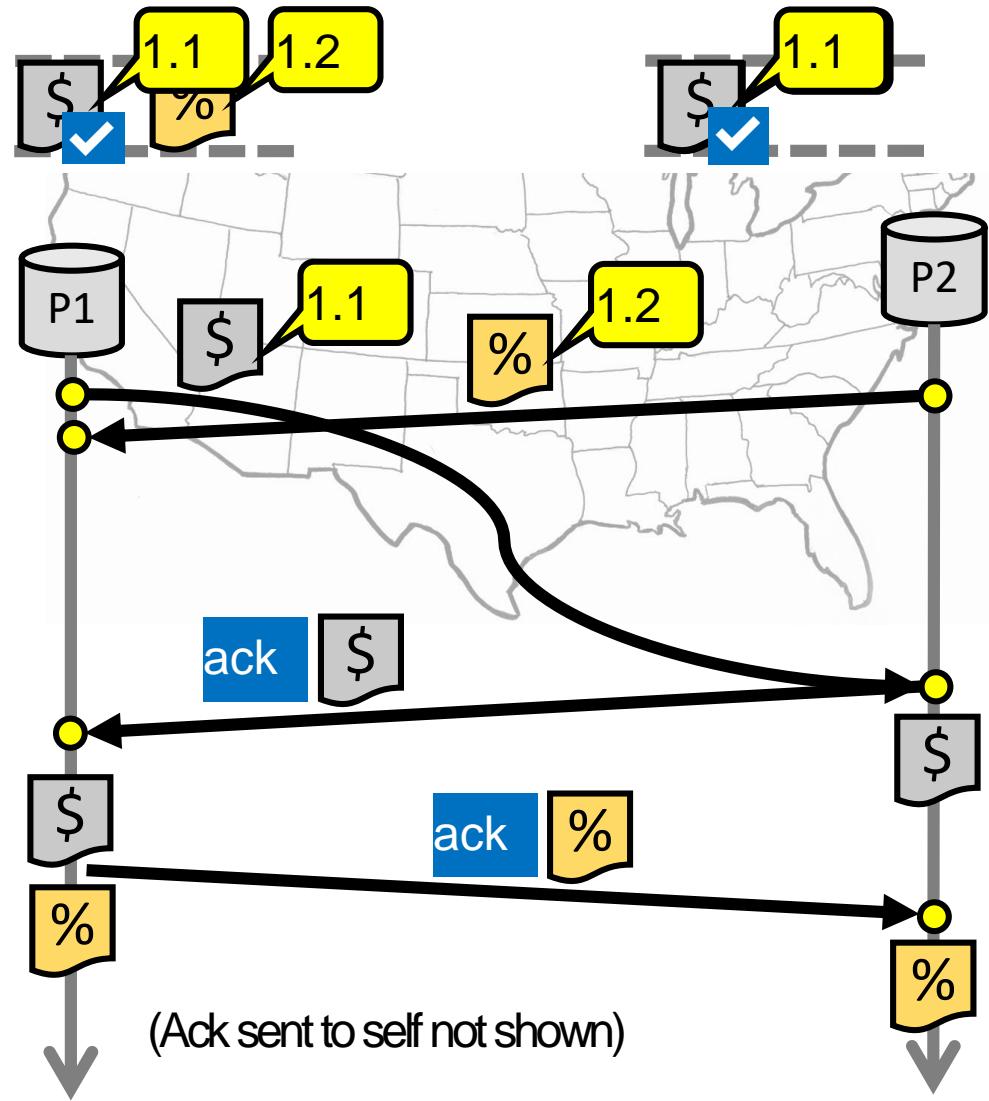
Example: P1's
local queue:



Totally-Ordered Multicast (Correct version)

1. On **receiving** an update from **client**, broadcast to others (including yourself)
2. On **receiving** or **processing** an update from a **replica**:
 - a) Add it to your local queue, if **received** update
 - b) Broadcast an **acknowledgement message** to every replica (including yourself) only from head of queue
3. On **receiving** an **acknowledgement**:
 - Mark corresponding update **acknowledged** in your queue
4. **Remove and process** updates *everyone* has ack'ed from *head* of queue

Totally Order Multicast



Logické hodiny a kauzalita

Pár současných událostí nemá kauzální cestu od jedné události ke druhé (ani jedním směrem).

Lamportovy časové značky pro současné události mohou, ale nemusí mít stejnou hodnotu!

- Platí: jestliže $e_1 \rightarrow e_2$, pak $C(e_1) < C(e_2)$
- Neplatí: jestliže $C(e_1) < C(e_2)$, pak $e_1 \rightarrow e_2$

Jak zajistit, že z uspořádání časových značek jednoznačně poznáme potenciální kauzalitu?



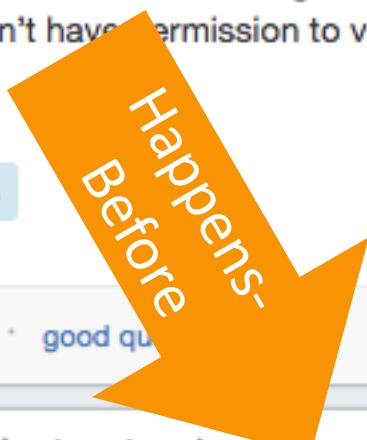
Vektorové hodiny

Motivation: Distributed discussion board



Can't access paper review

I get this error when clicking on the link on 1
"You don't have permission to view submis:



logistics

edit

good qu



the instructors' answer, where instruc

Thanks for letting me know - it was several

edit

good answer | 0

▼ WEEK 2/12 - 2/18

■ Instr First office hours coming up ...

2/13/17

Hi all -- as you reading SampleRate and Roofnet
for tomorrow's class, please begin to think
about your projects and

▼ WEEK 2/5 - 2/11

■ Instr Class is on today

2/9/17

I'm planning on holding class today, but of
course understand if you have travel difficulties
for those of you comin

O K

O K

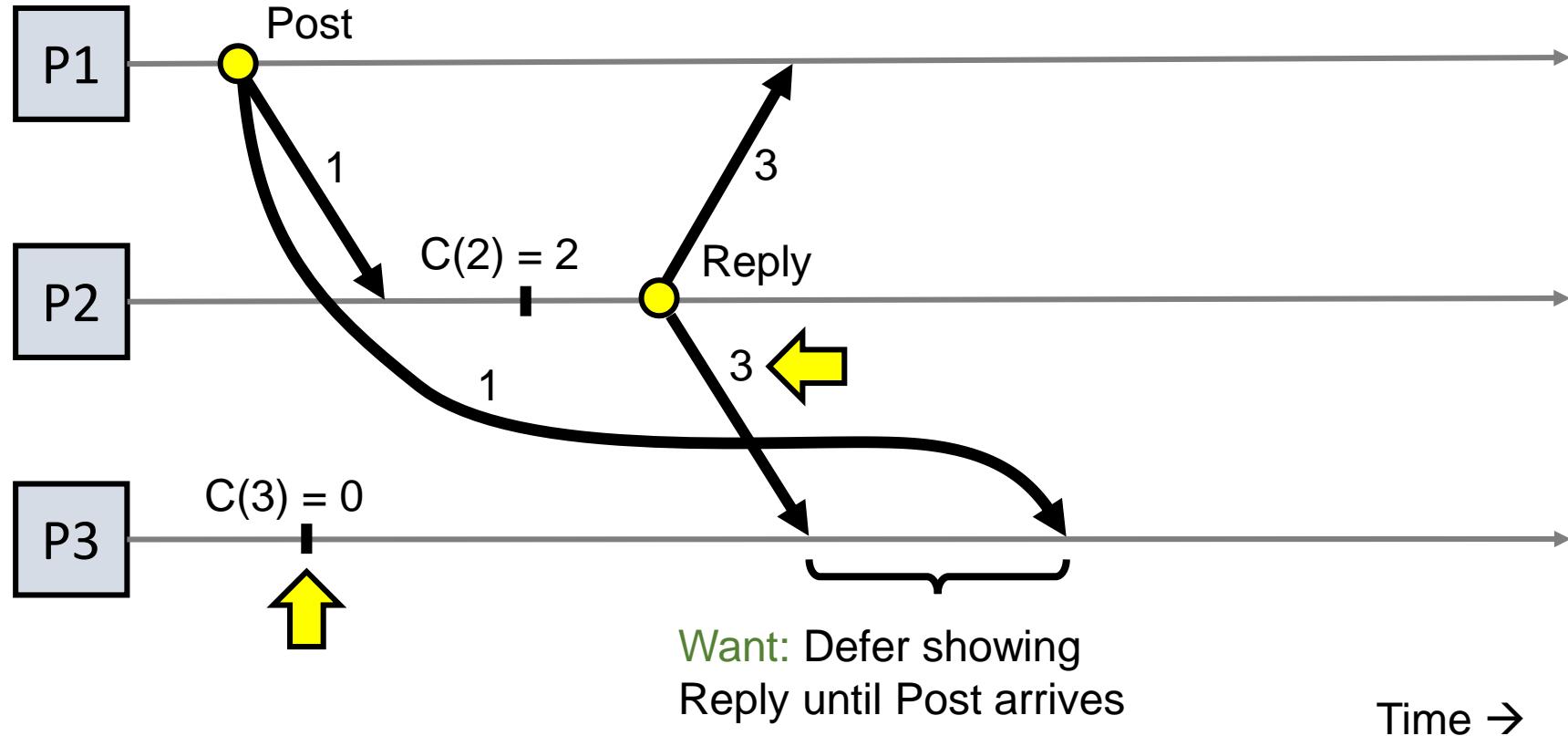
Can't access paper review

I get this error when clicking on the link on the
syllabus: "You don't have permission to view
submission #1. Enter

2/8/17

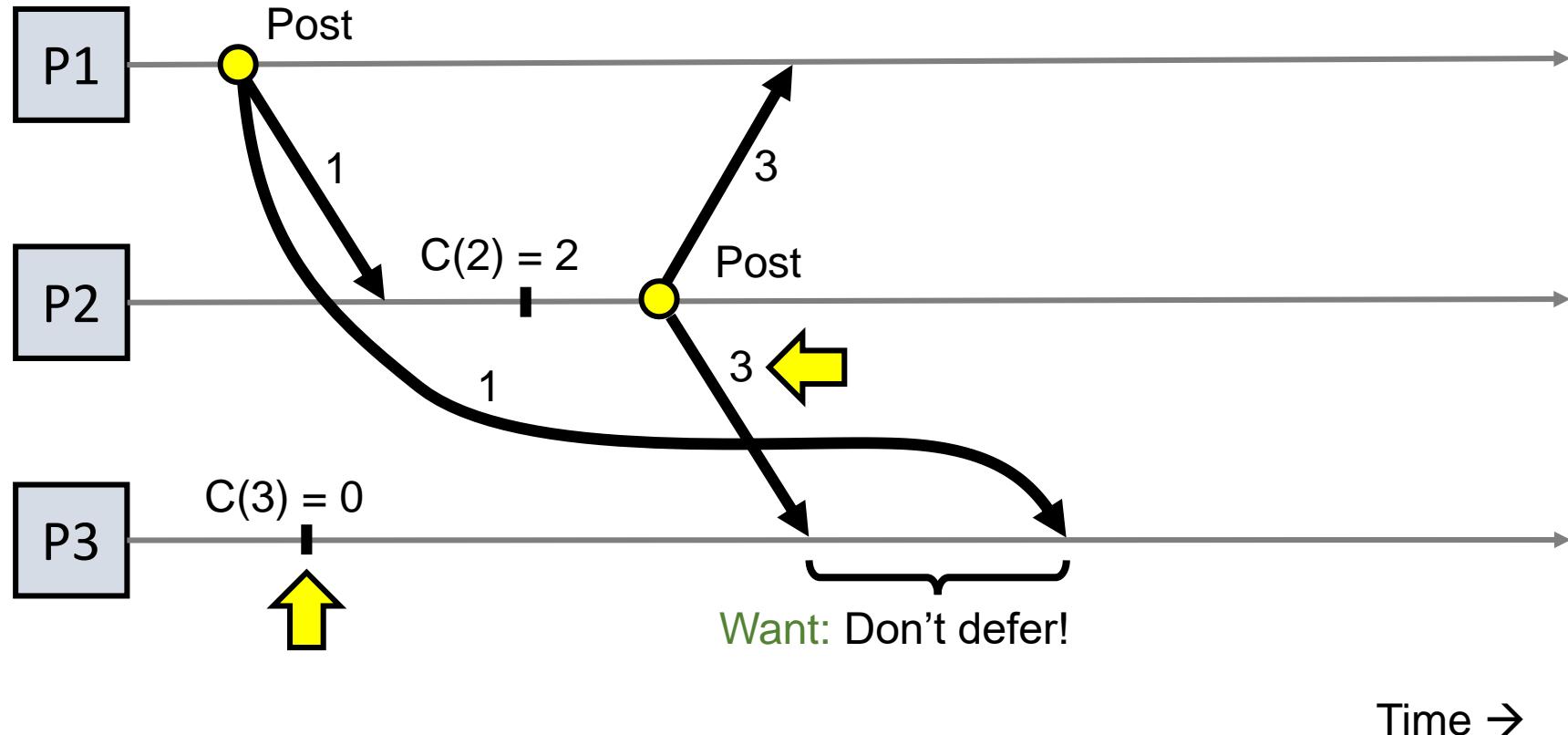
i

Lamport Clock-based discussion board



- Defer showing message if $C(\text{message}) > \text{local clock} + 1$?

Lamport Clock-based discussion board



- No! Gap could be due to other **independent** posts

Vektorové hodiny

Předpokládejme skupinu procesů $\{p_1, \dots p_N\}$

Každý proces si udržuje **vektor** celočíselných hodin $V_i[1 \dots N]$

- j -tý element vektorových hodin procesu $V_i[j]$ je znalost procesu i o událostech v procesu j

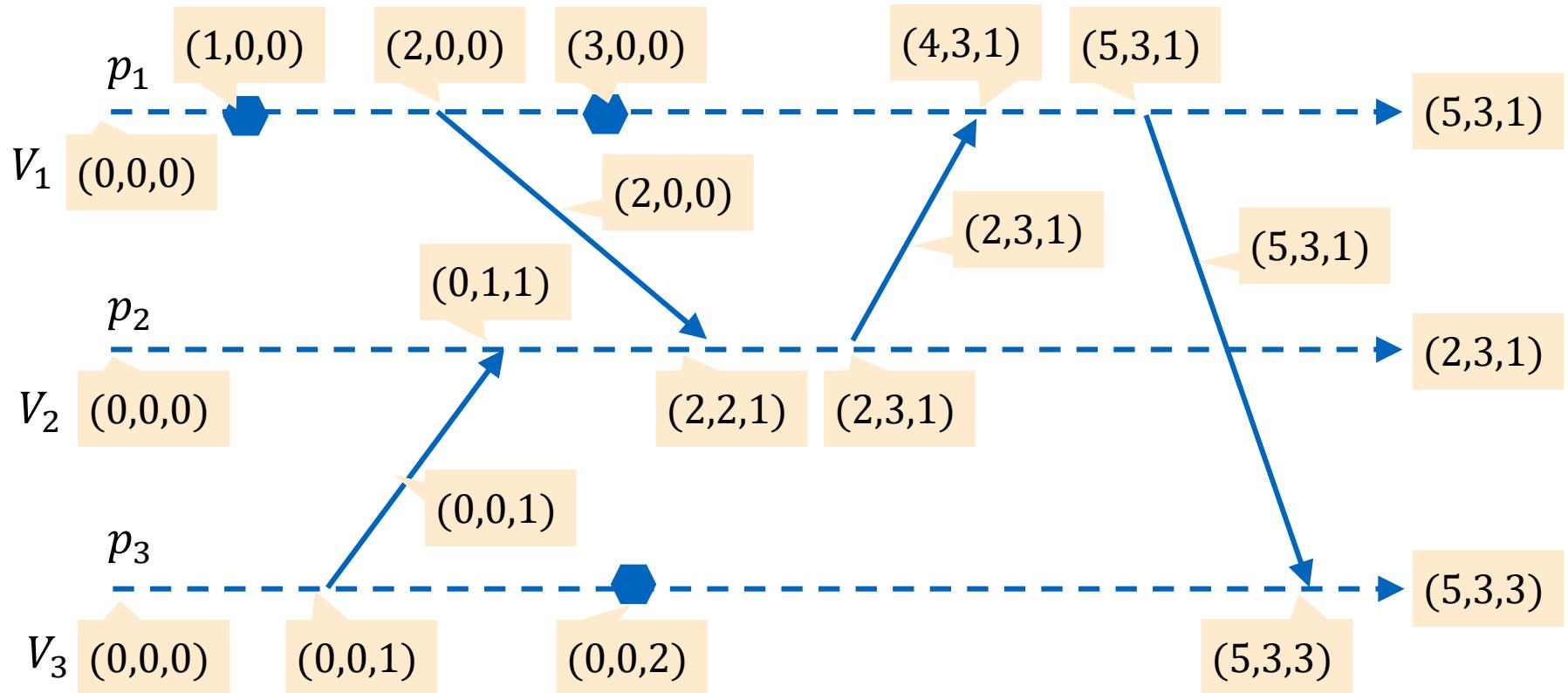
Synchronizace vektorových hodin

Synchronizace vektorových hodin

Každý proces p_i si udržuje **lokální vektorové hodiny** V_i a nastavuje

1. Před **provedením akce** v procesu p_i se V_i **inkrementuje** o 1, tj. $V_i[i] := V_i[i] + 1$
2. **Pošle-li** proces p_i zprávu m procesu p_j , **nastaví vektorovou časovou značku** $ts(m)$ zprávy m na V_i (poté, co provedl krok 1)
3. Proces p_j po **přijetí** zprávy m
 - nastaví své hodiny $V_j[k] := \max(V_j[k], ts(m)[k])$ pro všechna $k = 1, \dots, N$ (tzv. **sloučení**)
 - poté **inkrementuje** $V_j[j]$ a předá zprávu m aplikaci.

Vektorové hodiny: příklad



Vektorové hodiny a kauzalita

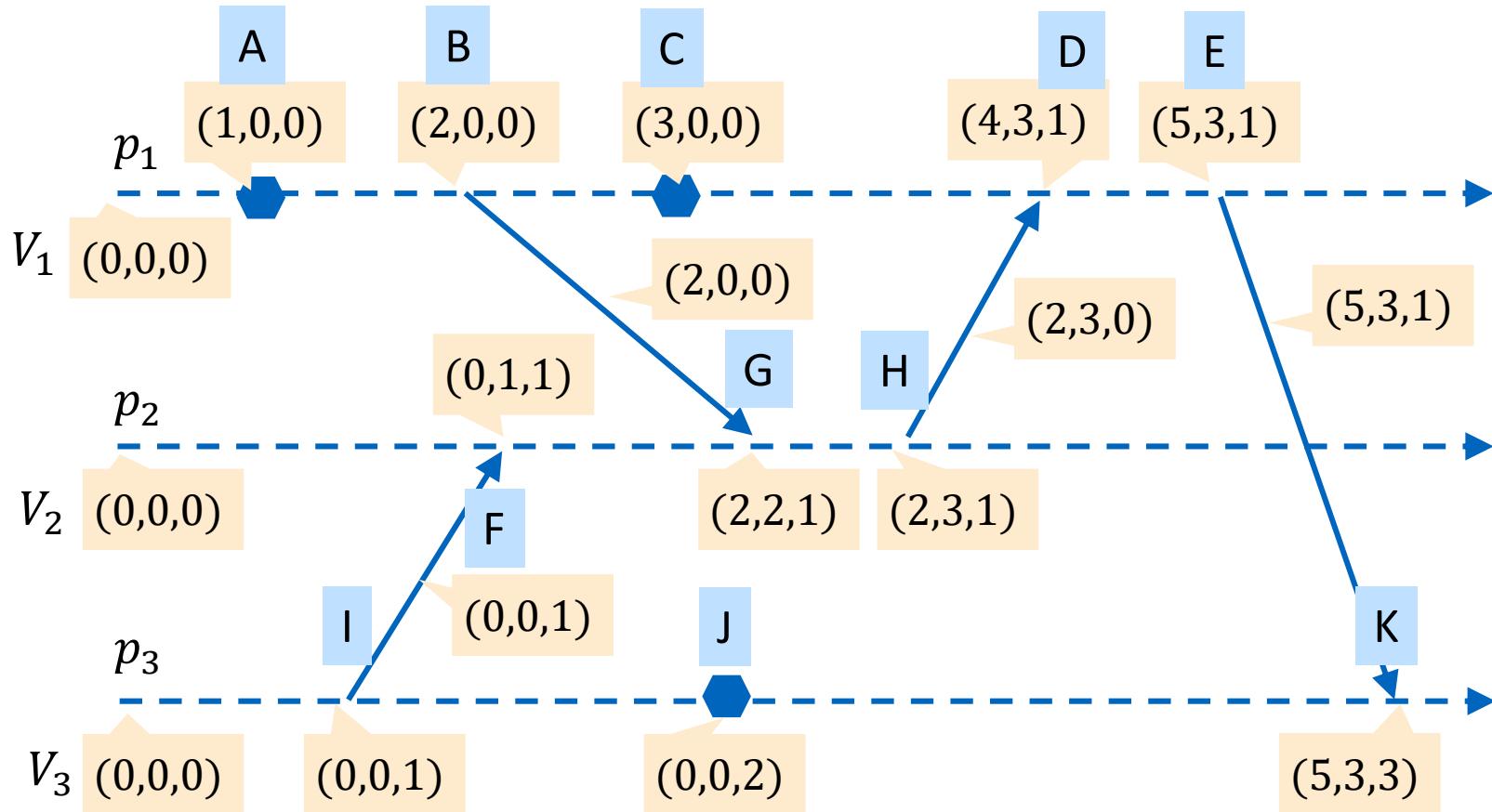
Upořádání vektorových časových značek:

- $V_1 = V_2$ iff $V_1[i] = V_2[i]$ pro všechna $i = 1, \dots, N$
- $V_1 < V_2$ iff $V_1[i] \leq V_2[i]$ pro všechna $i = 1, \dots, N$ a existuje j , že $V_1[j] < V_2[j]$
- Pokud: $\neg(V_1 < V_2) \wedge \neg(V_2 < V_1)$, pak píšeme $V_1 \parallel V_2$

Pro vektorové hodiny platí:

$e_1 \rightarrow e_2$ iff pro časové událostí platí: $V_{e_1} < V_{e_2}$

Vektorové hodiny: kauzalita



$$A \rightarrow B \quad (1, 0, 0) < (2, 0, 0)$$

$$B \rightarrow G \quad (2, 0, 0) < (2, 2, 1)$$

$$A \rightarrow G \quad (1, 0, 0) < (2, 2, 1)$$

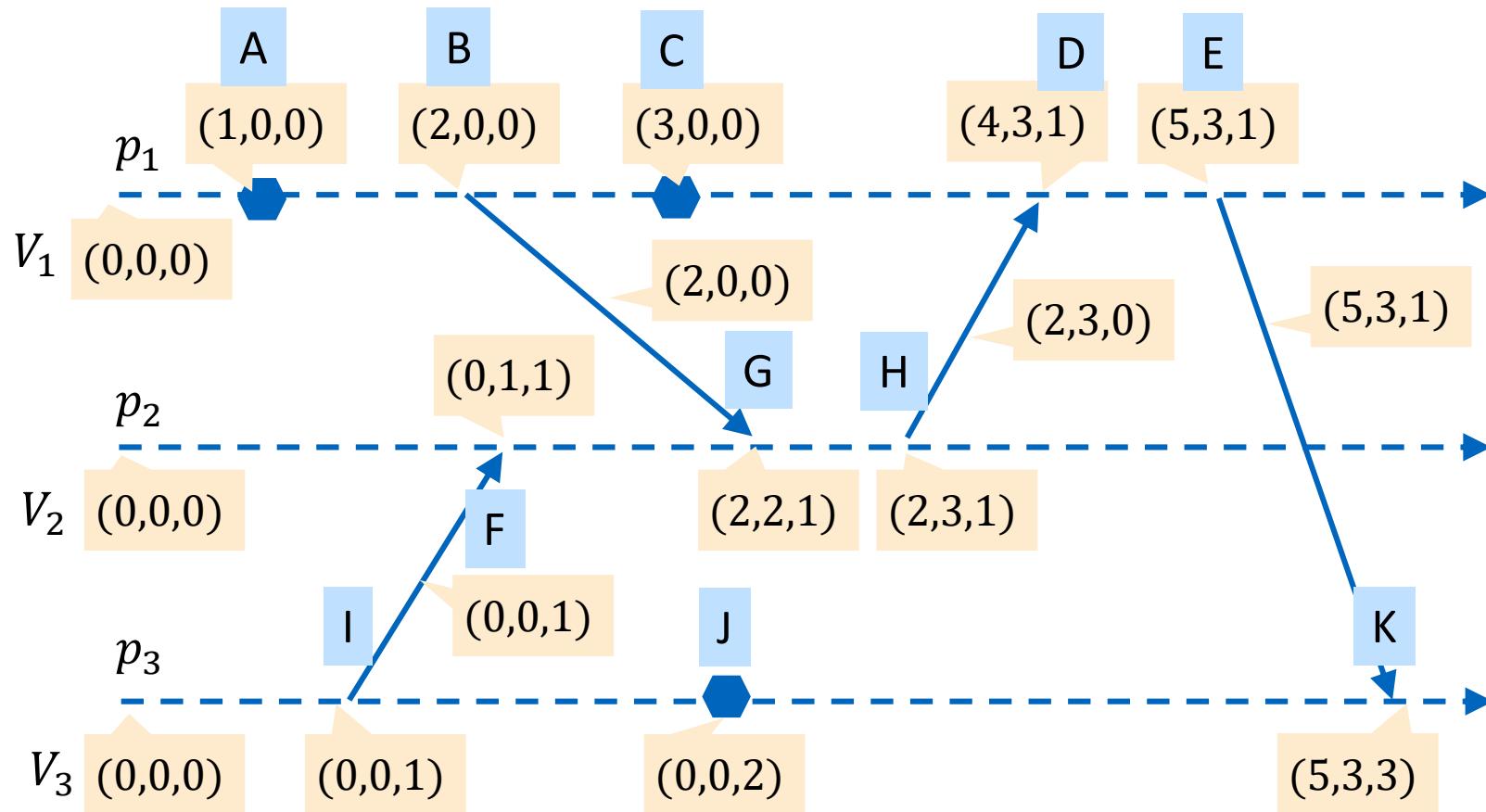
$$I \rightarrow H \quad (0, 0, 1) < (2, 3, 1)$$

$$F \rightarrow K \quad (0, 1, 1) < (5, 3, 3)$$

$$I \rightarrow K \quad (0, 0, 1) < (5, 3, 3)$$

$$C \rightarrow K \quad (3, 0, 0) < (5, 3, 3)$$

Souběžné události



$$C \parallel G \quad (3,0,0) \parallel (2,2,1)$$

$$A \parallel J \quad (1,0,0) \parallel (0,0,2)$$

$$C \parallel H \quad (3,0,0) \parallel (2,3,1)$$

vektorové hodiny rozliší
souběžné události

Logické hodiny: Souhrn

Lamportovy (skálární) logické hodiny

- respektují **kauzalitu**
- ale nerozliší současné události

Vektorové hodiny

- respektují kauzalitu
- potřebují více místa, ale rozliší **současné události**

Lze dokázat, že pro zachycení kauzality ve skupině N procesů je potřeba **vektorové hodiny délky N** pro každý proces

- Existují algoritmy, které redukují množství dat potřebných pro udržbu vektorových hodin (např. Raynal a Singhal, 1996)

Čas v DS: Shrnutí

Přesně **synchronizované globální hodiny** v DS (s nenulovou latencí přenosu zpráv) **neexistují**.

Lze synchronizovat s určitou přesností:

- Cristianův algoritmus
- NTP
- Berkely algoritmus

... ale chyba je **nenulová** a je funkci **doby oběhu zprávy (RTT)**

Nutnosti synchronizovat hodiny se můžeme vyhnout využitím **logických hodin**.

PDV 10 2019/2020

Výpočet globálního stavu

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT





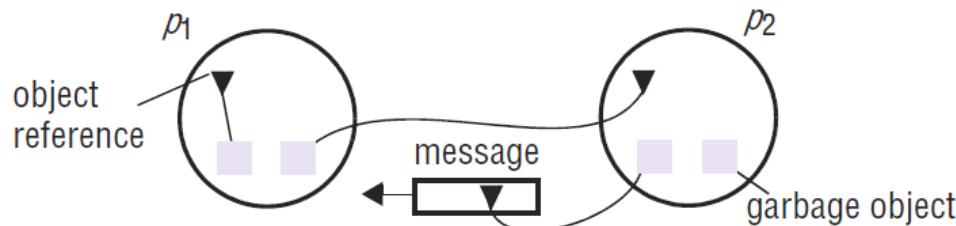
Globální Stav

Globální stav: množina lokální stavů procesů v DS a stavů všech komunikačních kanálů v jednom okamžiku*.

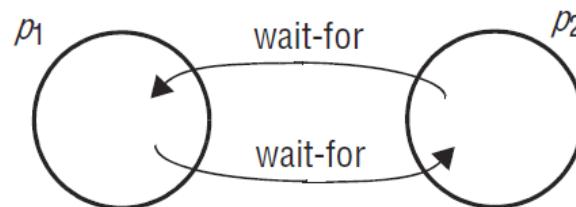
Globální snapshot: záznam globální stavu.

Příklady

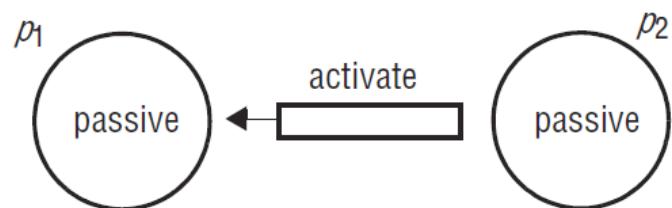
Garbage collection: nutnost identifikovat objekty, na které není *globálně* žádná reference.



Detekce uváznutí (deadlock): nutnost identifikovat cykly *globálním* wait-for grafu.



Detekce ukončení výpočtu: nutnost zajistit, že všechny procesy jsou pasivní a v žádném kanálu přenosu není žádná potenciálně aktivační zpráva.



Checkpointing za účelem obnovení globálního stavu systému.

...

Globální stav

Pokud bychom měli **globální hodiny**, tak zaznamenat globální stav jednoduché: všechny procesy by zaznamenaly stav v dohodnutý čas, tj. v jednom **fyzickém okamžiku**.

Jak zaznamenat globální stav **bez** globálních hodin?

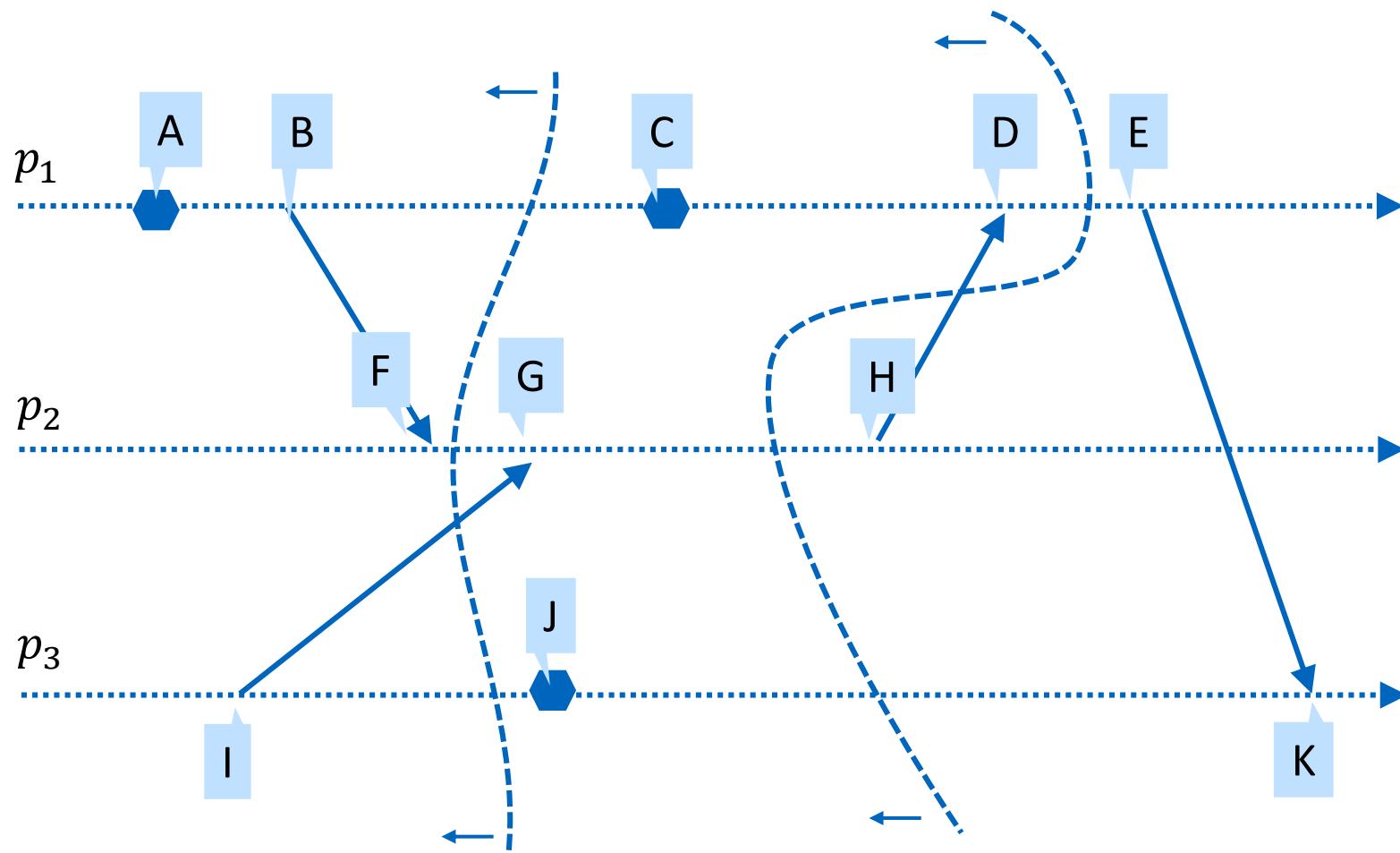
→ **Logický okamžik**

Řez distribuovaného výpočtu

Řez: časová hranice v každém procesu a v každém komunikačním kanále.

- události, které nastanou před řezem, jsou **v řezu**
- události, které nastanou po něm, jsou **mimo řez**.

Řez: Příklad



Konzistentní řez

Definice (optimistická)

Řez R je konzistentní pokud splňuje kauzalitu, tj. pokud pro každý pár událostí e, f v systému platí:

$$f \in R \wedge e \rightarrow f \Rightarrow e \in R$$

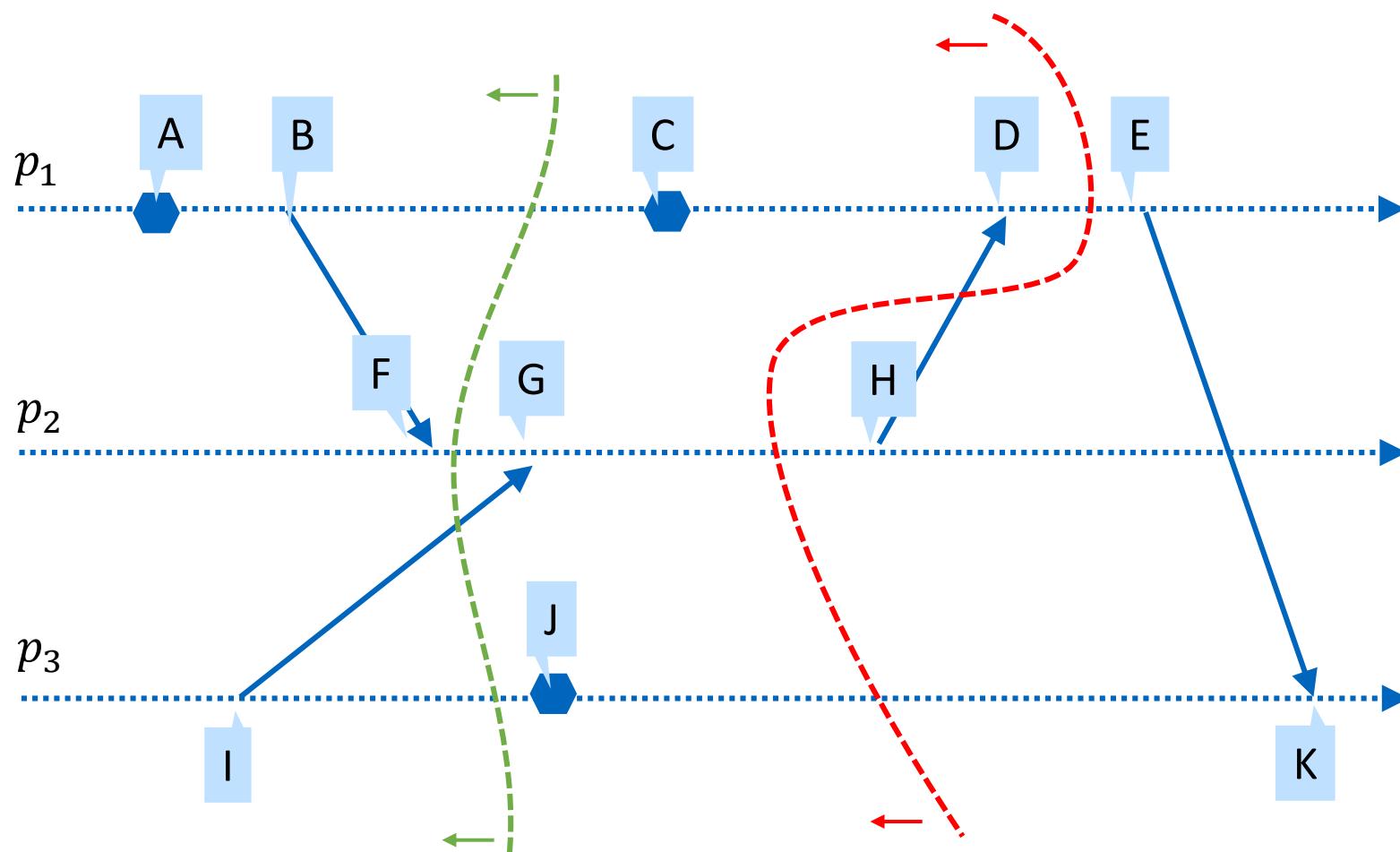
tj. pokud řez obsahuje nějakou událost, obsahuje i všechny, které ji předcházejí dle relace **stalo se před** (tj. nelze, aby v řezu byl „důsledek“ a nebyla tam „příčina“.)

Konzistentní řez = **logický okamžik**

Konzistentní globální stav odpovídá konzistentnímu řezu.

- Globální stav je konzistentní, pokud by mohl byt pozorován externím pozorovatelem.

Řez: Příklad



konzistentní řez
(mohl, ale *nemusel*
být pozorován)

nekonzistentní řez
(nikdy nemohl být pozorován)



Výpočet konzistentního globálního stavu

Problém výpočtu globálního stavu

Cíl: Zaznamenat **globální snapshot**, tj. stav pro každý proces a každý komunikační kanál.

Požadavek: Zaznamenání snapshotu by neměla **interferovat** s během distribuované aplikace a neměla by vyžadovat po aplikaci zastavení posílání aplikačních zpráv.

Model

(Existují i algoritmy se slabšími předpoklady)

- skupina N procesů p_1, \dots, p_N
- **FIFO perfektní komunikační kanál** mezi každým párem procesů, tj. zprávy se neduplikují, nevznikají, neztrácejí a jsou doručovány v pořadí odeslání (značení: $C_{i,j}$ kanál přenáší zprávy z procesu p_i do procesu p_j)
- **asynchronní** systém: neznáma, ale **konečná latence**

Předpoklad: Každý proces je schopen zaznamenat svůj vlastní **aplikační stav** (případně low-level systémový stav).

Chandy-Lamport algoritmus pro distribuovaný globální snapshot

(Vytváření snapshotu je distribuované.)

Speciální zpráva: **ZNAČKA** ■

Jeden (libovolný) z procesů iniciuje vytvoření snapshotů.

Procesy reagují na příjem zprávy **ZNAČKA** ■ .

Chandy-Lamport algoritmus pro globální snapshot

Zahájení tvorby snapshotu

Iniciující **proces** p_i odešle ZNAČKU ▪ všem ostatním procesům (i sobě)

Příjem ZNAČKY ▪ procesem p_i kanálem $C_{m,i}$

if (p_i dosud nezaznamenal svůj stav) **then**

p_i **zaznamená** svůj stav (a vykoná pravidlo pro odeslání ZNAČKY ▪);

p_i **zaznamená** stav kanálu $C_{m,i}$ jako prázdnou množinu;

p_i každým odchozím **kanálem** $C_{i,j}$ **odešle** jednu ZNAČKU ▪ (předtím než skrze $C_{i,j}$ pošle jakoukoliv jinou zprávu);

p_i **zapne zaznamenávání** zpráv doručených skrze všechny ostatní příchozí kanály $C_{j,i}$ kromě $C_{m,i}$

else

p_i **zaznamená** stav kanálu $C_{m,i}$ jako množinu všech zpráv, které p_i obdržel skrze $C_{m,i}$ od doby, kdy zahájil záznam $C_{m,i}$;

p_i záznam **ukončí**;

end if

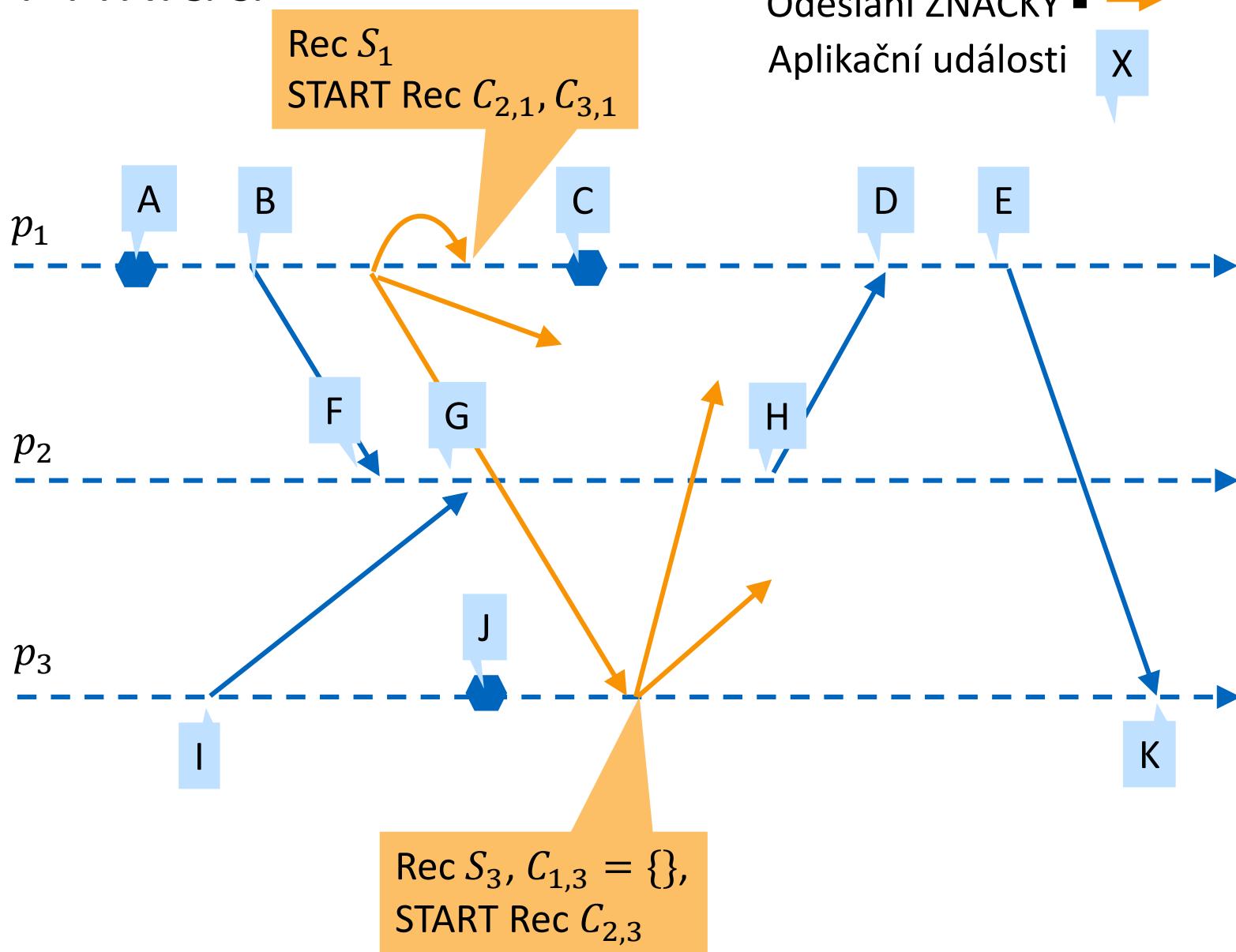
Ukončení

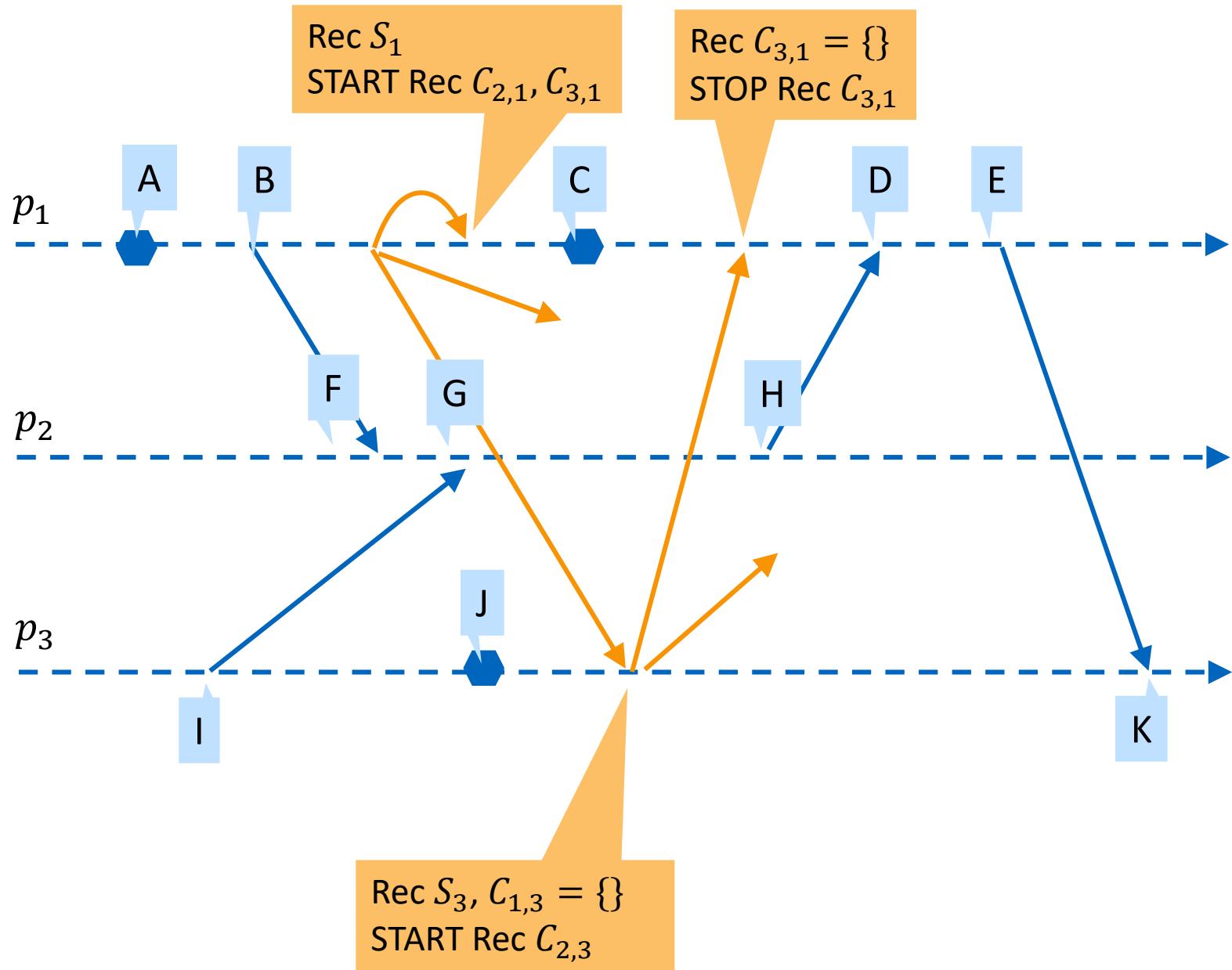
Algoritmus končí jakmile:

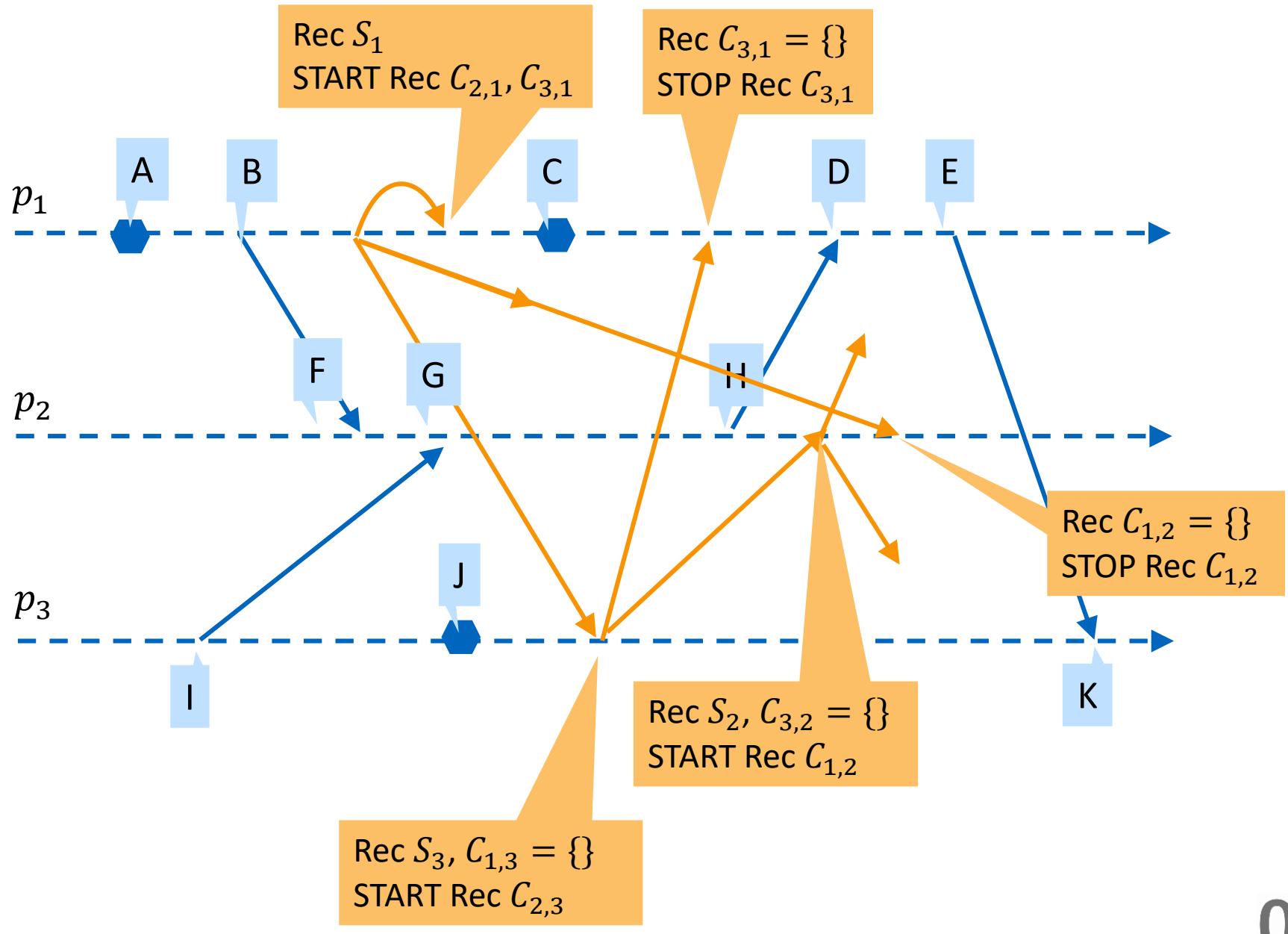
- všechny procesy přijaly ZNAČKU ■ (aby zaznamenaly svůj stav) a
- všechny procesy přijaly Značku ■ na všech $N - 1$ svých příchozích kanálech (aby zaznamenaly stav na všech kanálech)

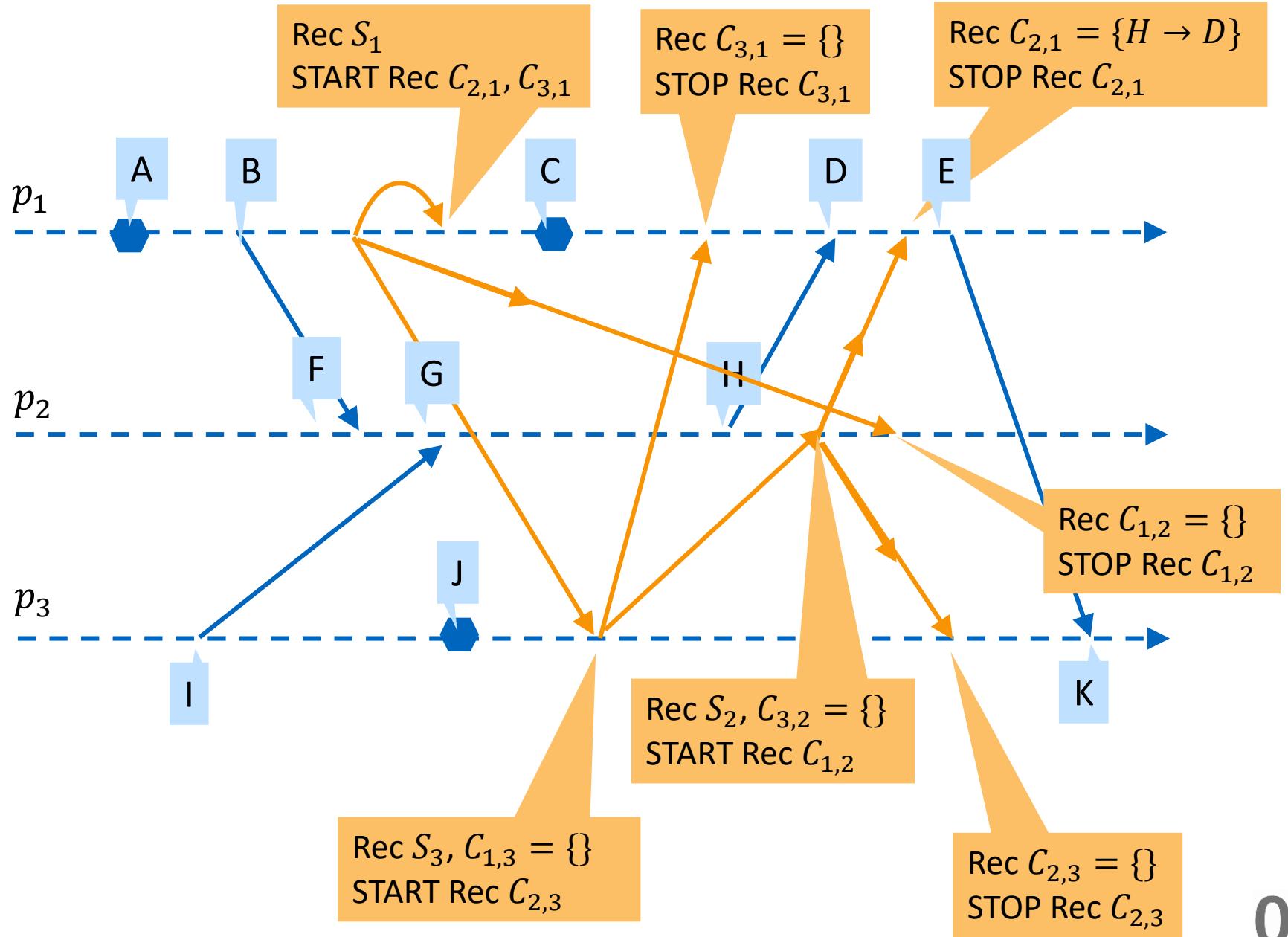
Následně (je-li potřeba) mohou být jednotlivé fragmenty globální stavu posbírány centrálním servery a poskládán **plný globální snapshot**.

Příklad

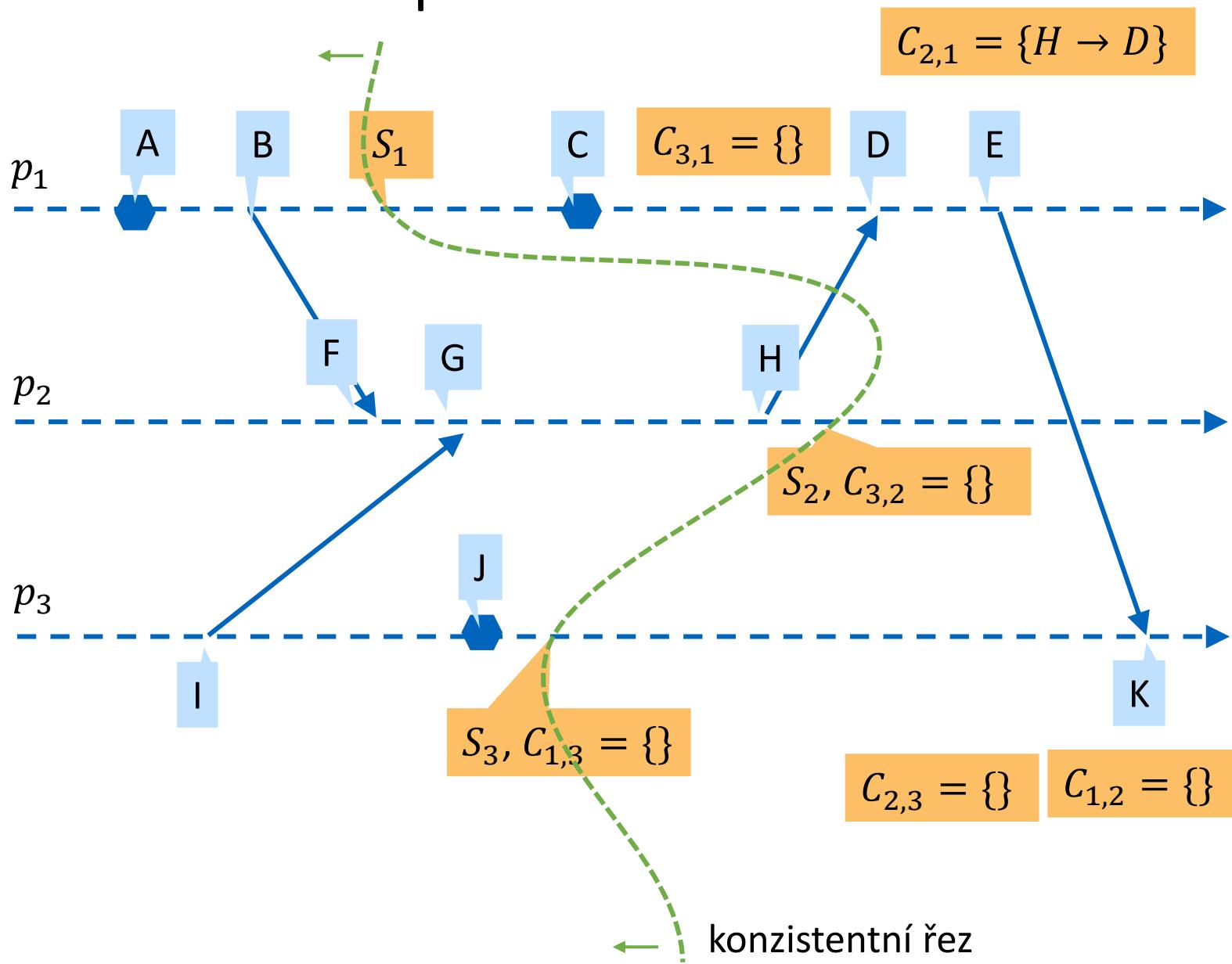








Finální Snapshot





Vlastnosti

Vlastnosti Chandy-Lamport

Výsledkem Chandy-Lamport algoritmu pro výpočet globální snapshotu je **konzistentní řez**.

Důkaz:

- nechť R je výsledný řez a e_i a e_j jsou události v procesech P_i a P_j tak, že $e_i \rightarrow e_j$
- pro konzistenci třeba dokázat implikaci: $e_j \in R \Rightarrow e_i \in R$
- tj. pokud $e_j \rightarrow < P_j$ zaznamená svůj stav $>$ pak $e_i \rightarrow < P_i$ zaznamená svůj stav $>$
- Přepokládejme, že tato implikace neplatí, tj. $e_j \rightarrow < P_j$ zaznamená svůj stav $>$ a $< P_i$ zaznamená svůj stav $>> e_i$
- Uvažujme cestu aplikačních zpráv z e_i do e_j (přes další procesy)
- Vzhledem k FIFO kanálu musí ZNAČKY na všech spojích výše uvedené cesty předcházet aplikační zprávy
- Tedy, protože $< P_j$ zaznamená svůj stav $>> e_i$, tak musí platit, že P_j obdržel svoji ZNAČKU před e_j
- A tedy $e_j \notin R \rightarrow$ spor

Vyhodnocení stabilních vlastností

Uvažujme **vlastnost distribuovaného výpočtu** jako logickou formulí definovanou nad globálním stavem distribuovaného výpočtu.

Stabilní vlastnost je taková vlastnost, že jakmile je ve výpočtu **jednou** splněna, zůstává splněna **navždy**.

- příklad stabilní vlastnosti živosti: výpočet skončil
- příklad stabilní vlastnosti porušující bezpečnost: nastalo uváznutí, objekt je sirotek (neukazuje na něj žádná reference)

Chandy-Lamport algoritmus lze použít pro detekci **stabilních** globálních vlastností.

(Lze ukázat, že pokud nějaká stabilní vlastnost splněna v globálním snapshotu zachyceným snapshot algoritmem, bude splněna i výpočtu splněna i ve fyzickém okamžiku doběhnutí snapshotů algoritmu. Naopak, pokud ve vypočteném globálním snapshotu nějaká stabilní vlastnost splněna není, nemohla být ve výpočtu splněna v okamžiku zahájení snapshotů algoritmu).

Shrnutí

Schopnost **zachytit globální stav** distribuovaného výpočtu je důležitá.

Vytvoření snapshotů by nemělo nijak omezovat probíhající výpočet.

Chandy-Lamportův algoritmus vypočte globální snapshot.

Vypočtený globální snapshot odpovídá **konzistentnímu řezu**.

Globální snapshot může být využit k **detekci stabilních vlastností** výpočtu.

PDV 11 2019/2020

Konsensus

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



Co mají tyto příklady společného?

Skupina procesů usilujících o :

- udržování jejich lokálních seznamů aktivních procesů aktuálních [detekce selhání]
- zvolení lídra a zajištění, že každý ví, kdo je lídrem [volba lídra]
- zajistit vzájemně exkluzivního přístupu ke sdílenému prostředku (např. souboru) [vyloučení procesů]
- usilujících o doručení stejných aktualizací ve stejném pořadí [uspořádaný multicast]

Souvisí s konsensem

Ve všech těchto případech se procesy snaží vzájemně koordinovat, aby se shodly na nějaké hodnotě

- na stavu každého procesu (aktivní/neaktivní)
- kdo je lídrem
- kdo má přístup k sdílenému prostředku
- pořadí zpráv

Všechny tyto problémy souvisejí s problémem **konsensu**

Problém konsensu

N procesů

Každý proces P má

- vstupní proměnou x_P (výchozí návrh): zpočátku buď 0 nebo 1
- výstupní proměnou y_p : může být změněna pouze jednou

Problém konsensu: navrhnout takový protokol, že bud:

- všechny procesy nastaví svou výstupní proměnou na 0
- všechny proces nastaví svou výstupní proměnnou na 1

Cílem je **shodnout na hodnotě** výstupní proměnné.

- Procesy nemohou mít hodnotu výstupní proměnné pevně předprogramovanou – výstupní proměnné musí záviset na vstupních proměnných

Proč je konsensus důležitý?

Mnoho problémů v DS je **ekvivalentních** konsensu

- perfektní detekce selhání
- volba lídra
- vyloučení procesů
- spolehlivý nebo totálně uspořádaný multicast
- ...

Vyřešení konsensu by tedy bylo velmi užitečné.

Algoritmus Paxos

Nejstarší a nejznámější algoritmus pro distribuovaný konsensus (Leslie Lamport)

Pracuje v kolech; každé kolo má unikátní číslo volebních zpráv

Kola jsou asynchronní

- Je-li proces v kole j a dorazí-li mu zpráva z kola $j + 1$: přeruší činnost v rámci kola j a posuň se do kola $j + 1$
- využívá časová time-outy (může být pesimistický)

Každé kolo rozděleno do třech fázi (které jsou taky asynchronní)

1. Fáze ELECTION: Je zvolen lídr
2. Fáze BILL: zvolený lídr navrhne hodnotu, ostatní procesy potvrzují
3. Fáze LAW: Lídr rozešle všem ostatním procesům finální (potvrzenou) hodnotu

Řešitelnost konsensu

V synchronním DS je konsensus řešitelný.

- můžeme využívat time-outů k rozlišení mezi selháním a zpožděním

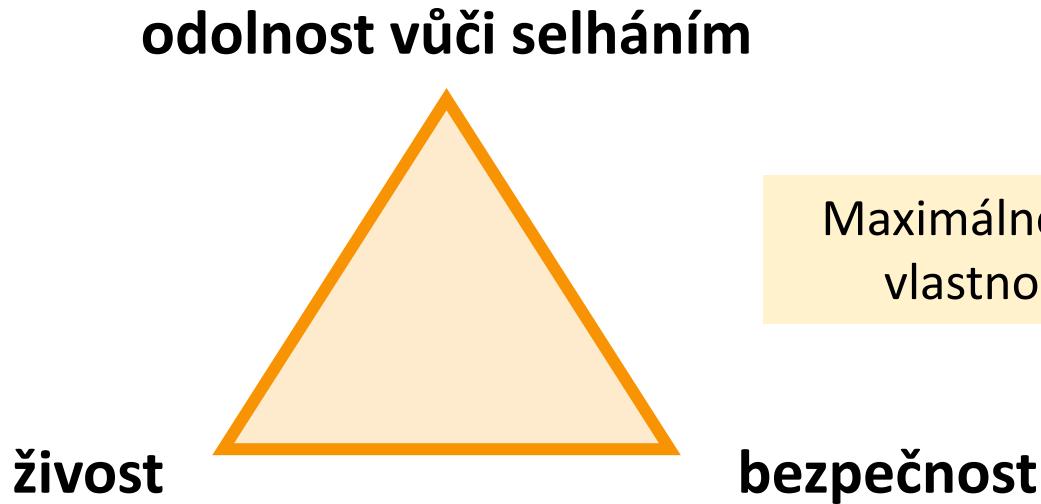
V asynchronních DS není konsensus řešitelný.

- pro jakýkoliv algoritmus existuje nejhorší možný průběh (se selháními procesů nebo kanálů), který zabrání dosažení konsensu
- vyplývá v tzv. **FLP teorému**

FLP teorém

FLP teorém

V asynchronním distribuovaném systému **nelze** dosáhnout současně **bezpečnosti** a **živosti** distribuovaného výpočtu, pokud v něm může docházet k **selháním** (byť i jediného procesu).



Řešitelnost konsensu

Ale: V praxi vždy vyžadujeme bezpečnost a díky částečné synchronicitě ve velkém množství běhů DS konsensu dosáhneme v **konečném čase (tzv. konečná živost – eventual liveness)**.

- existují i pravděpodobnostní algoritmy mající **konečnou střední hodnotu běhu**

Souhrn

Problém konsensu je v jádru mnoha problémů v DS.

V asynchronním DS **nelze** při přítomnosti selhání **konsensus vyřešit** ve smyslu bezpečnosti a živosti.

Praktická řešení většinou garantují **bezpečnost**.

PDV 11 2019/2020

Vzájemné vyloučení procesů

Michal Jakob

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 semafory, mutexy, monitor a další abstrakce poskytované OS založené na **sdílené paměti**.

Distribuovaný systém

(procesy komunikují posíláním zpráv)

Potřebujeme **distribuovaný protokol/algoritmus**.

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 proces 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:

- Nedoje 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 umožnuje 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

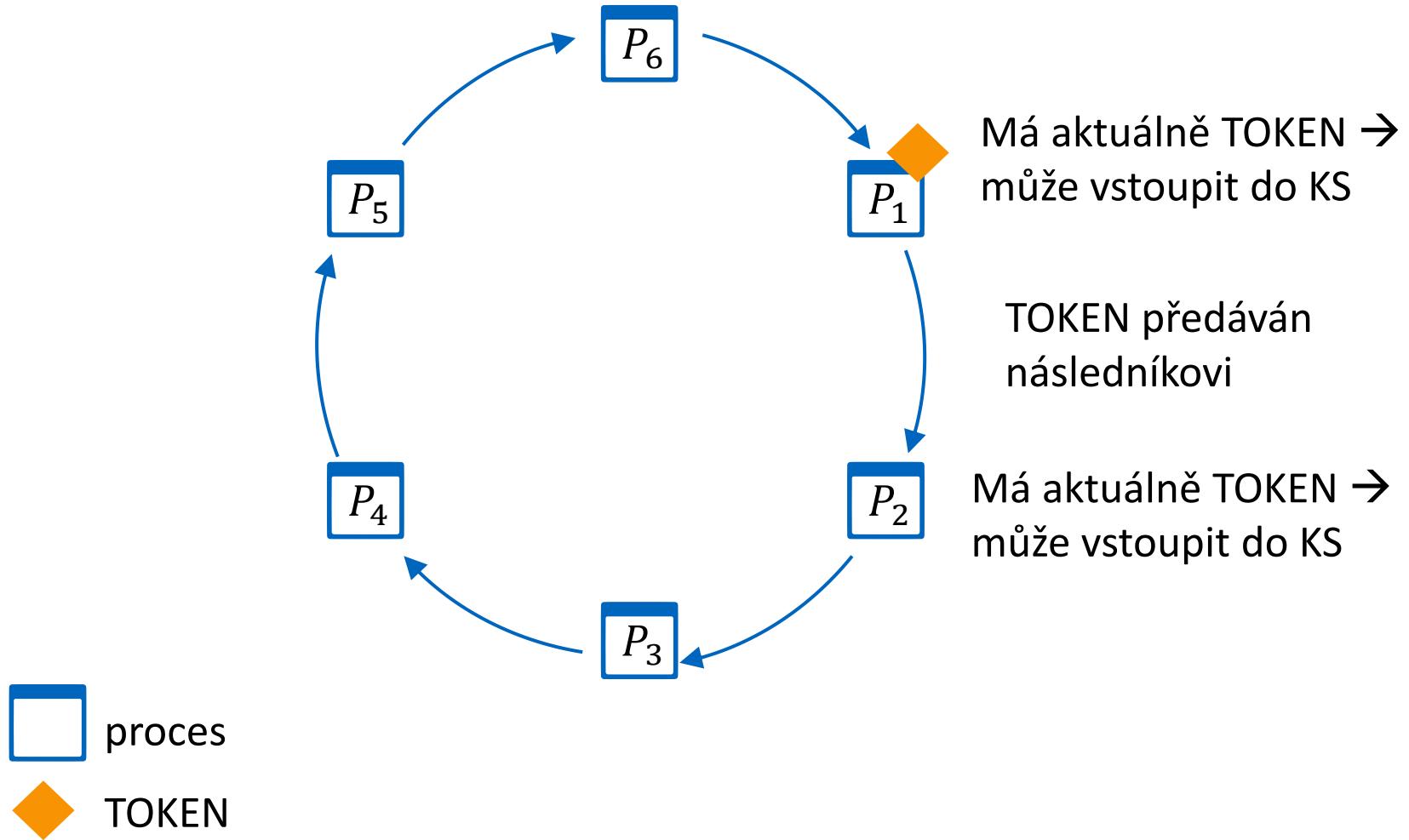
enter()

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



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 $O(N)$.

Můžeme zlepšit?



Ricart-Agrawal 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** (initializovanou 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 $\langle T_j, P_j \rangle$, $i \neq j$

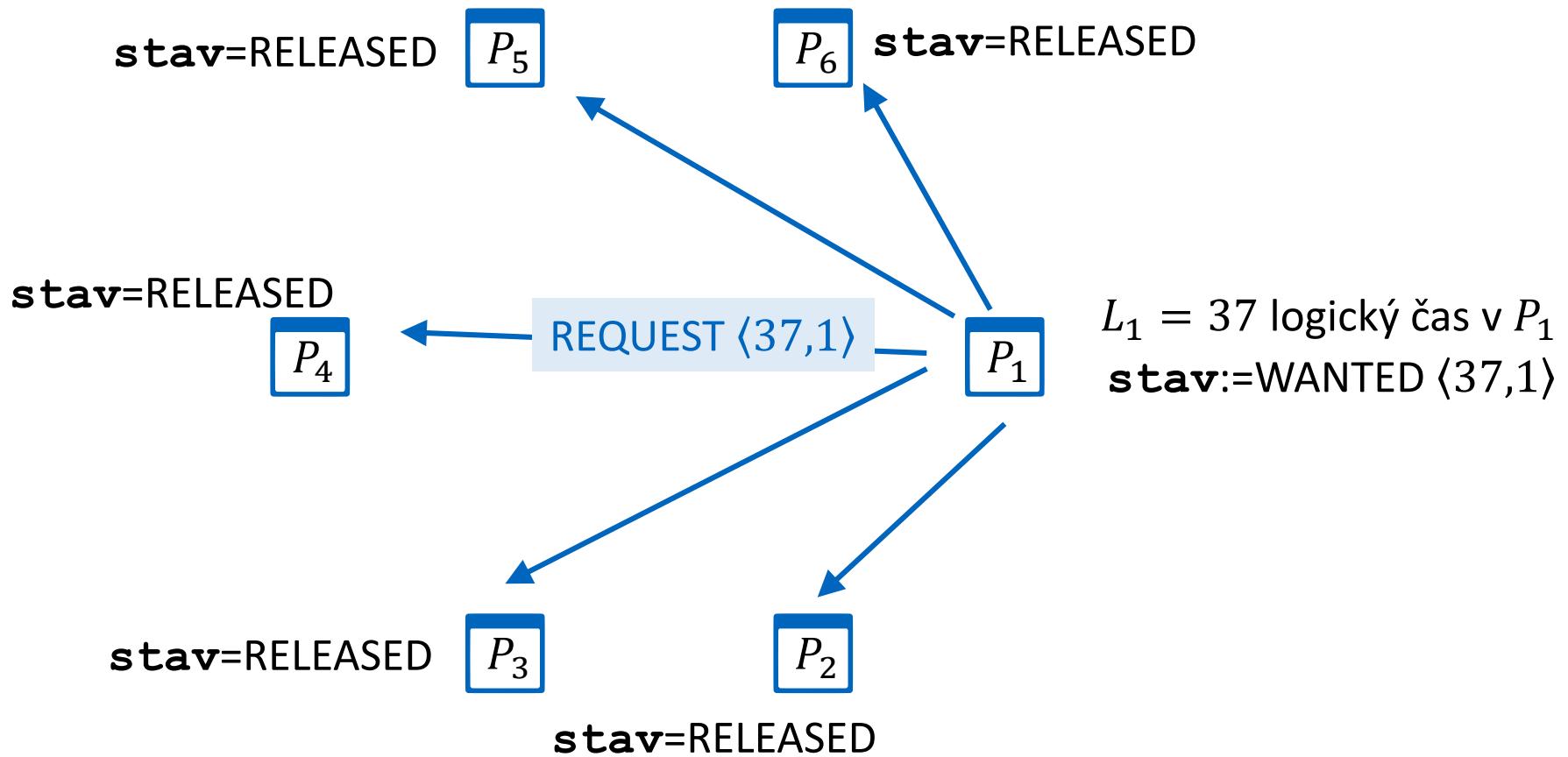
if (**stav** = HELD) nebo (**stav** = WANTED $\langle T_i, i \rangle$ a $\langle T_i, i \rangle < \langle T_j, j \rangle$)
 přidej REQUEST do **seznamu** čekajících požadavků;
else
 pošli OK do P_j ;

P_i : exit()

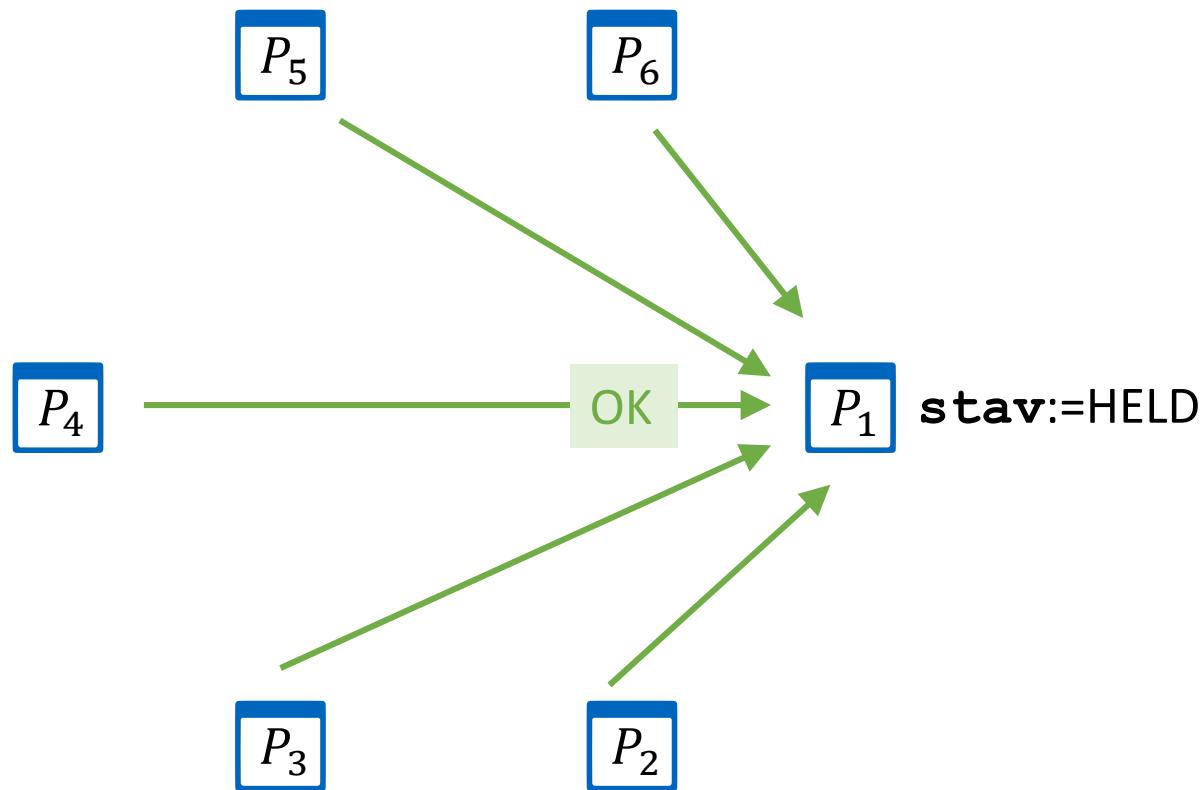
nastav stav na RELEASED;
pošli OK všem čekajícím procesům ze **seznamu** ;



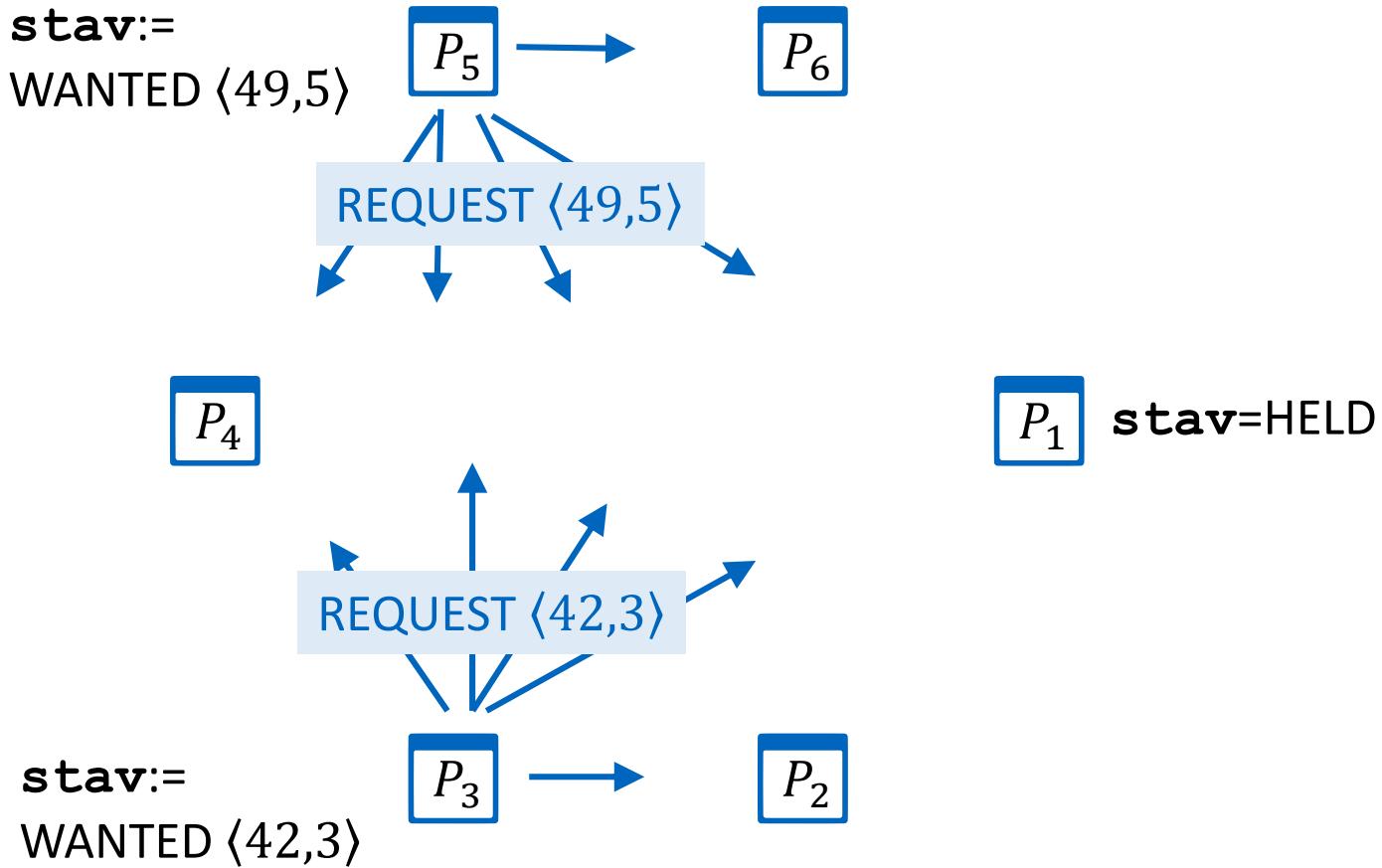
Příklad



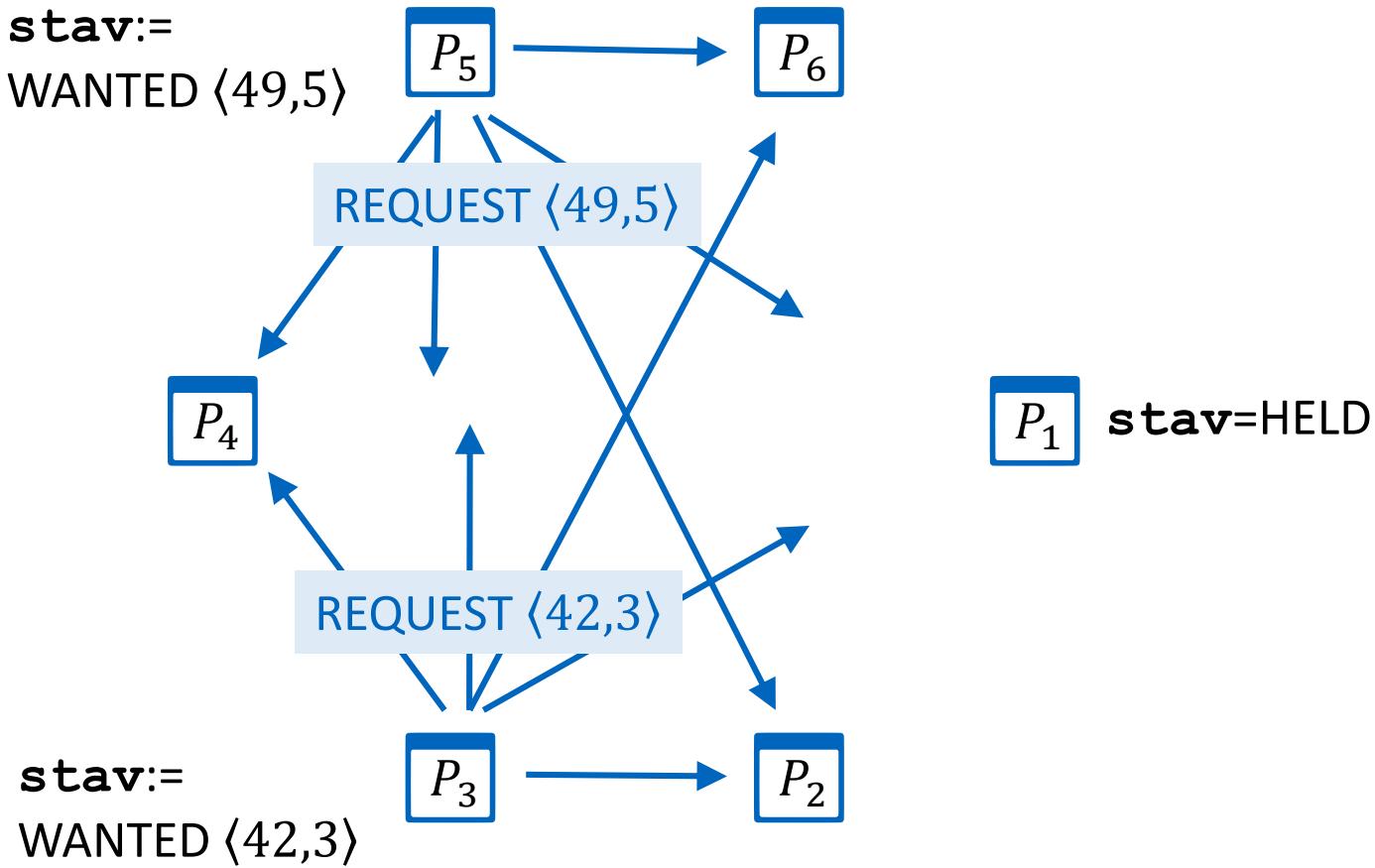
Příklad



Příklad

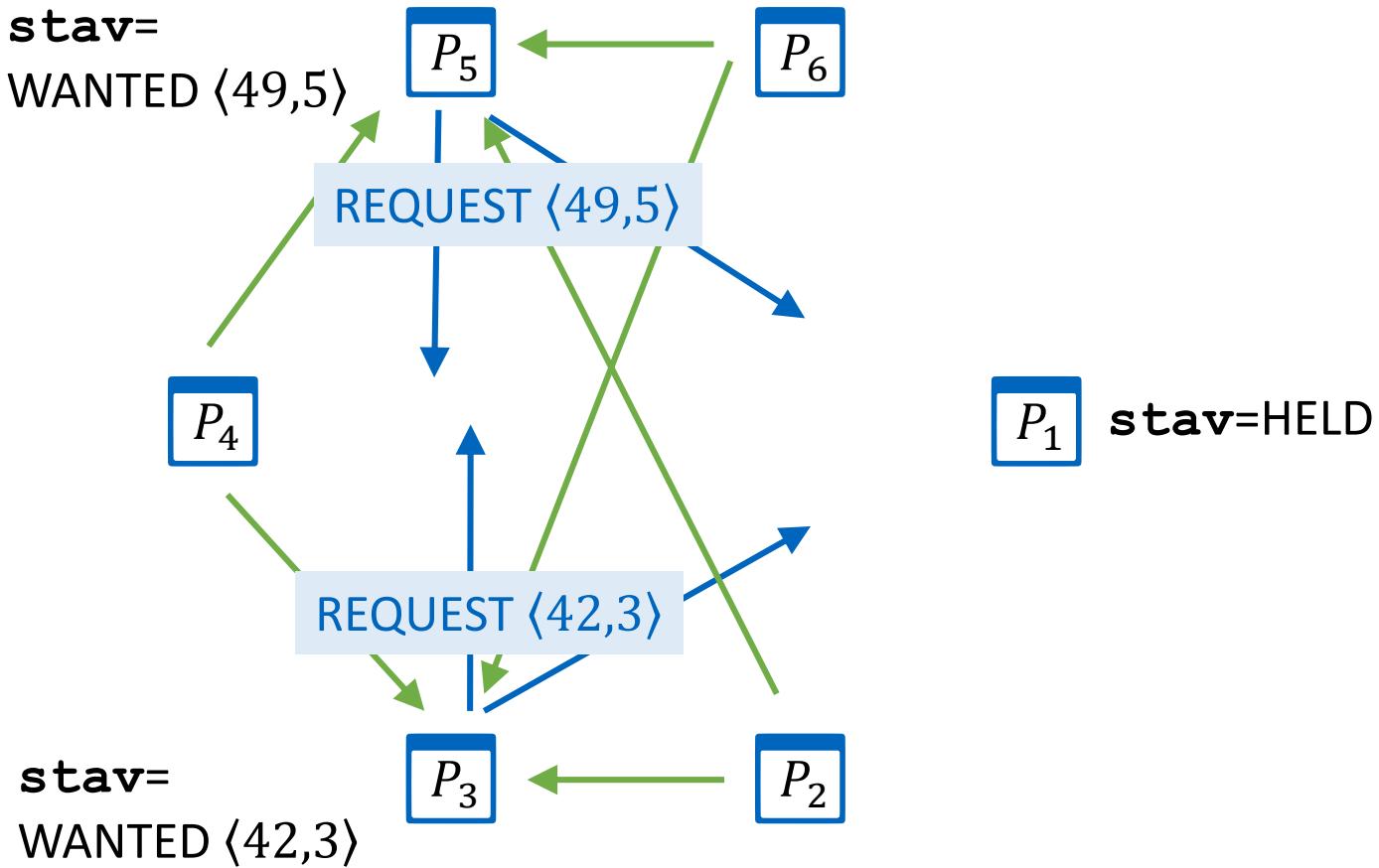


Příklad

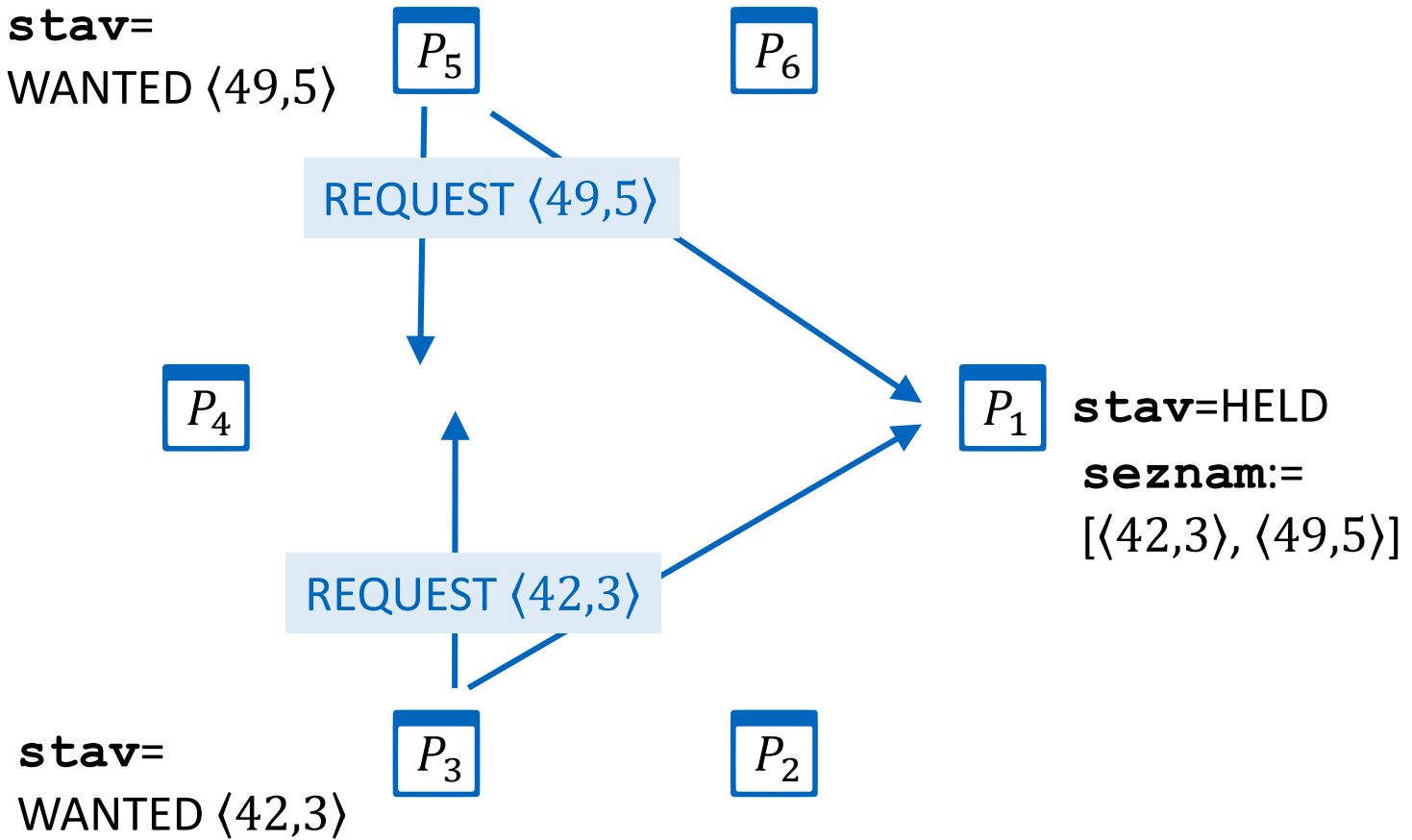


Příklad

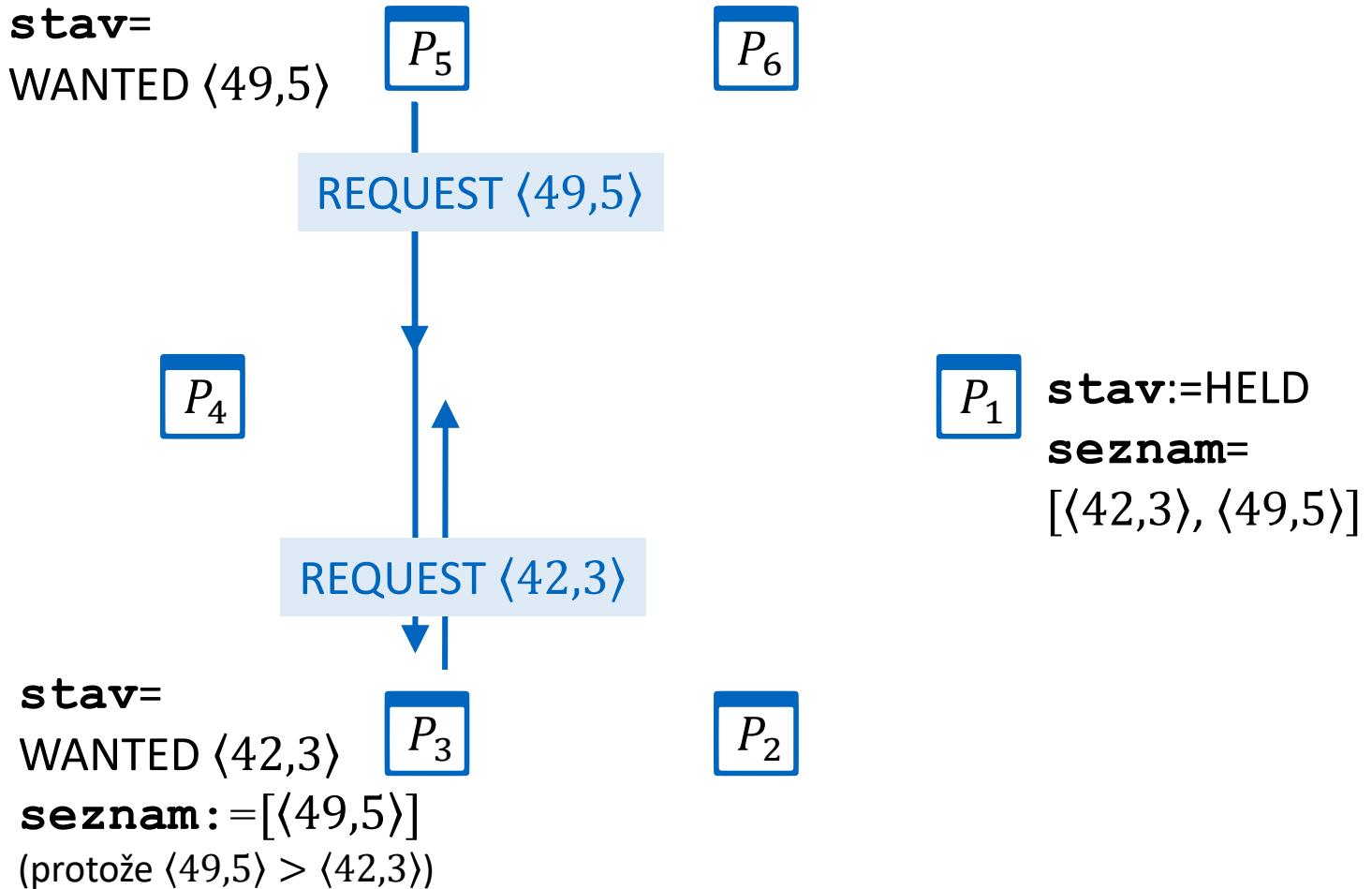
→ OK



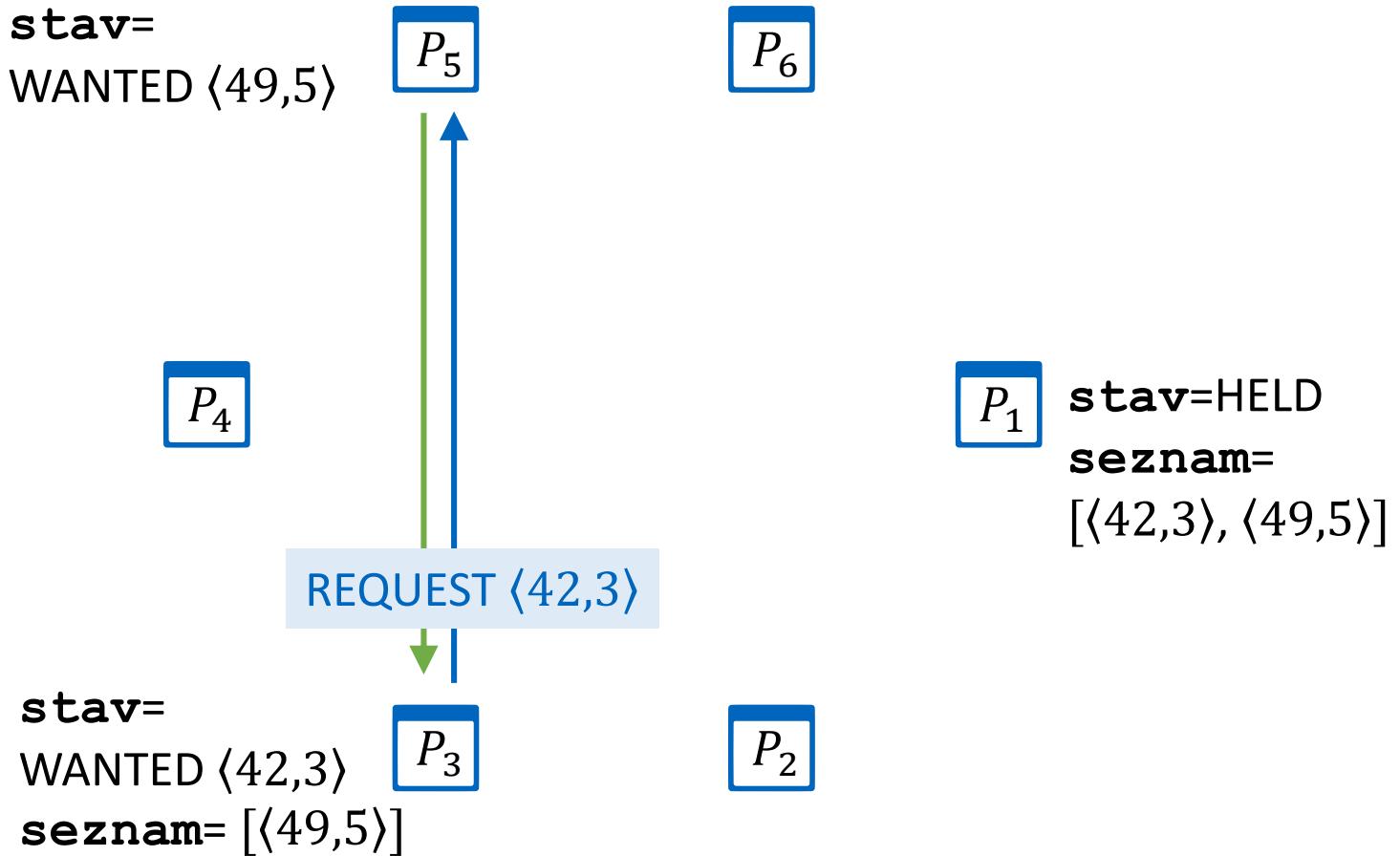
Příklad



Příklad



Příklad



Příklad

stav=

WANTED $\langle 49,5 \rangle$

(čeká na odpověď P_3)

P_5

P_6

P_4

P_1

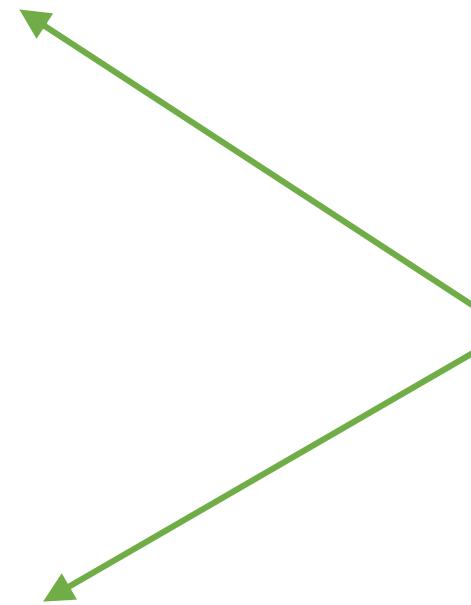
stav:=HELD

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

P_3

P_2

stav:=RELEASED
Multicast OK
procesům ze
seznamu
 $[\langle 42,3 \rangle, \langle 49,5 \rangle]$
seznam:=[]



Příklad

stav=

WANTED $\langle 49,5 \rangle$

(čeká na odpověď P_3)

P_5

P_6

P_4

P_1

stav=

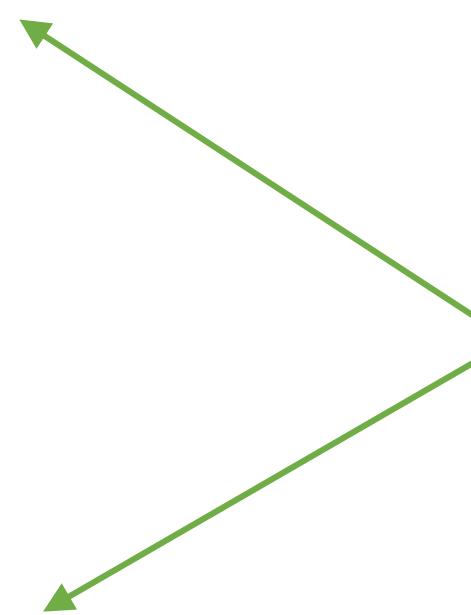
WANTED $\langle 42,3 \rangle$

seznam= $\langle 49,5 \rangle$

P_3

P_2

stav:=RELEASED
Multicast OK
procesům ze
seznamu
 $\langle \langle 42,3 \rangle, \langle 49,5 \rangle \rangle$
seznam:=[]



Příklad

stav=

WANTED {49,5}

(čeká ještě na
odpověď P_3)

stav:=RELEASED
seznam:=[]

P_4

P_5

P_3

P_6

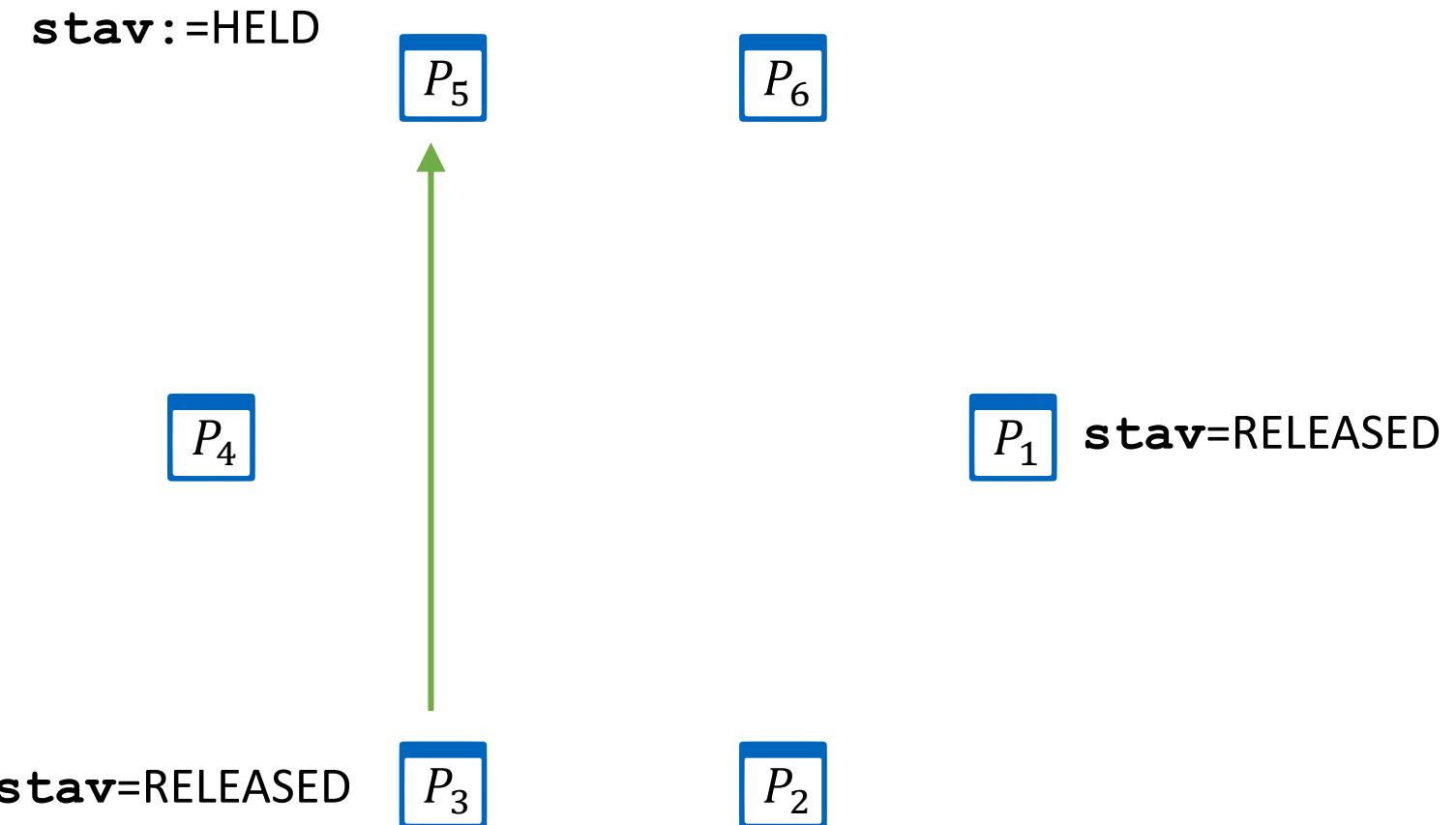
P_2

P_1

stav=RELEASED



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 $O(N)$

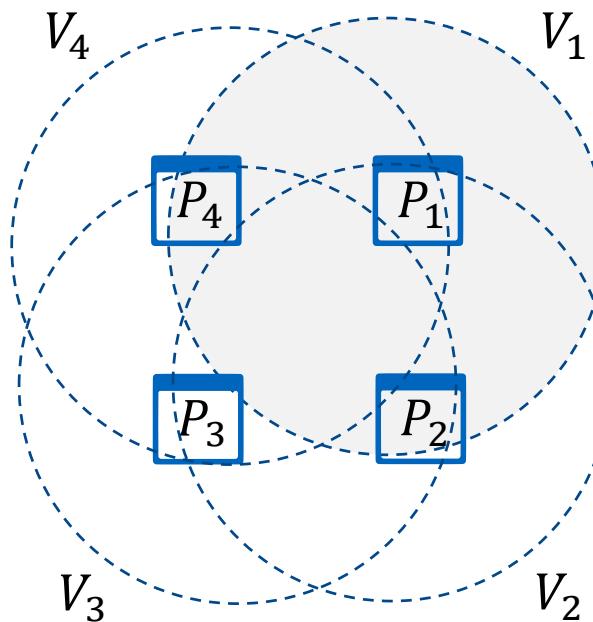
Lze ji dále snížit?

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 **asynchronicitou**, ale nikoliv se selháními.

PDV 12 2019/2020

Algoritmus Raft

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT





Algorithmus RAFT

Model a požadavky

Asynchronní systém se selháními

- procesy mohou **havarovat** (fail-stop, tj. nikoliv byzantsky)
- zprávy se mohou **ztrácet** (ale dodržují pořadí → nedokonalý FIFO kanál)

Algoritmus pro konsensus musí garantovat **bezpečnost** a měl by maximalizovat **živost** (dostupnost)

- obojí garantovat nelze ← **FLP teorém**

Přístupu k problému konsensu

Symetrický/bez lídra

- všechny servery mají stejnou roli
- klienti mohou kontaktovat kterýkoliv server

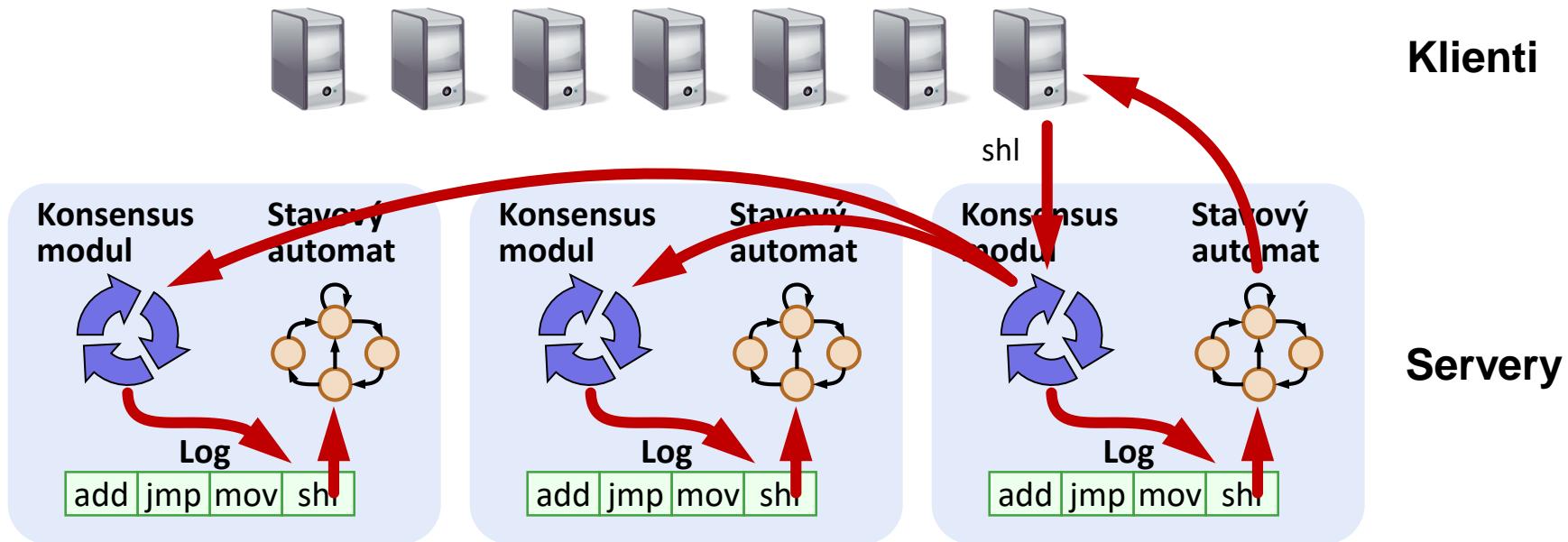
Asymetrický/s lídrem

- v každém okamžiku je jeden server lídrem a ostatní přijímají jeho rozhodnutí
- klienti komunikují s lídrem

Raft využívá lídra – výhody:

- **dekomponuje** problém na 1) **běžný chod** a 2) změny lídra
- **zjednodušuje** běžný chod (nedochází ke konfliktům)
- **efektivnější** než symetrické přístupy bez lídra (selhání lídra jsou v praxi vzácná)

Cíl: Replikovaný log



Replikovaný log → replikovaný stavový automat

- Všechny procesy vykonávají příkazy ve stejném pořadí

Konsensus modul zajišťuje správnou **replikaci logu** a rozhoduje, kdy mohou být příkazy vykonány.

Zpracování požadavků klientů postupuje pokud je **nadpoloviční většina** serverů aktivních.

Přehled Raftu

1. Volba lídra
 - volba jednoho ze serveru jako lídra
 - detekce selhání a vyvolání volby nového lídra
2. Běžný chod (základní replikace logu)
3. Bezpečnost a konzistence po změně lídra
4. Neutralizace starých lídrů
5. Interakce s klienty
6. (Rekonfigurace)



Volba lídra

Stavy serveru¹

V každém okamžiku je každý server v právě **jednom stavu**:

Lídr

obsluhuje
požadavky klientů
a replikuje log

Následovník

pasivní – pouze
reagují na zprávy
od jiných serverů

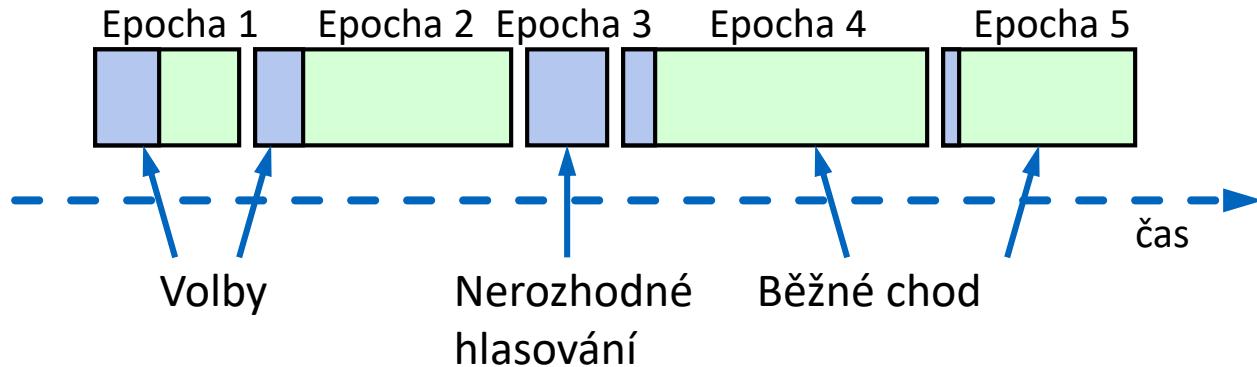
Kandidát

přechodná role
v průběhu volby
lídra

Běžný chod: 1 lídr , $N - 1$ následovníků

¹Skupinu procesů, které se účastní konsensu, budeme označovat jako servery – pro lepší odlišení od procesů, které běží na klientských počítačích a které se konsensu neúčastní.

Epochy (volební období)



Čas je rozdělen do **epoch** (~logický čas): každá epocha má své **číslo**, čísla jsou **inkrementována** a nikdy nejsou znova použitá. Každý server si (*persistentně!*) udržuje číslo **aktuální epochy**.

Epochy mohou mít dvě části

- **Volby** (buď nedopadnou nebo vyústí ve zvolení právě jednoho lídra)
- **Běžný chod** pod jedním zvoleným lídrem

Maximálně jeden lídr v každé epoše; některé epochy ale nemají lídra (neúspěšné volby).

Epochy slouží k identifikování **zastaralých informací**.

Raft Protocol Summary

Followers	RequestVote RPC
<ul style="list-style-type: none">Respond to RPCs from candidates and leaders.Convert to candidate if election timeout elapses without either:<ul style="list-style-type: none">Receiving valid AppendEntries RPC, orGranting vote to candidate	Invoked by candidates to gather votes.
Candidates	
<ul style="list-style-type: none">Increment currentTerm, vote for selfReset election timeoutSend RequestVote RPCs to all other servers, wait for either:<ul style="list-style-type: none">Votes received from majority of servers: become leaderAppendEntries RPC received from new leader: step downElection timeout elapses without election resolution: increment term, start new electionDiscover higher term: step down	Arguments: candidateId candidate requesting vote term candidate's term lastLogIndex index of candidate's last log entry lastLogTerm term of candidate's last log entry Results: term currentTerm, for candidate to update itself voteGranted true means candidate received vote Implementation: <ol style="list-style-type: none">If term > currentTerm, currentTerm \leftarrow term (step down if leader or candidate)If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout
Leaders	AppendEntries RPC
<ul style="list-style-type: none">Initialize nextIndex for each to last log index + 1Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeoutsAccept commands from clients, append new entries to local logWhenever last log index \geq nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successfulIf AppendEntries fails because of log inconsistency, decrement nextIndex and retryMark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of serversStep down if currentTerm changes	Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .
Persistent State	
Each server persists the following to stable storage synchronously before responding to RPCs: currentTerm latest term server has seen (initialized to 0 on first boot) votedFor candidateId that received vote in current term (or null if none) log[] log entries	Arguments: term leader's term leaderId so follower can redirect clients prevLogIndex index of log entry immediately preceding new ones prevLogTerm term of prevLogIndex entry entries[] log entries to store (empty for heartbeat) commitIndex last entry known to be committed Results: term currentTerm, for leader to update itself success true if follower contained entry matching prevLogIndex and prevLogTerm Implementation: <ol style="list-style-type: none">Return if term < currentTermIf term > currentTerm, currentTerm \leftarrow termIf candidate or leader, step downReset election timeoutReturn failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTermIf existing entries conflict with new entries, delete all existing entries starting with first conflicting entryAppend any new entries not already in the logAdvance state machine with newly committed entries
Log Entry	
term term when entry was received by leader index position of entry in the log command command for state machine	

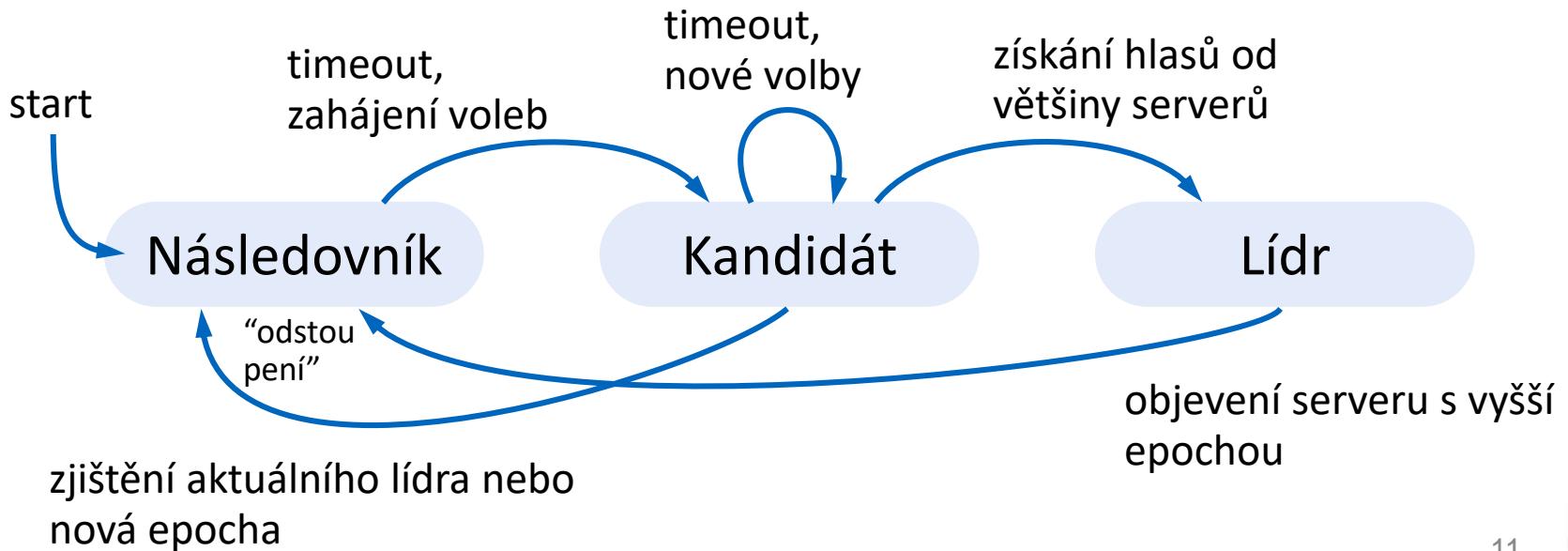
Stavy serveru

Servery začínají jako **Následovníci**.

- Následovníci očekávají zprávy od Lídra nebo Kandidátů

Lídři posílají **heartbeats** (prázdné zprávy **AppendEntries**), aby si udrželi autoritu.

Jakmile Následovník neobdrží zprávu do **volebního timeoutu** (typicky 100-500ms), předpokládá, že Lídr havaroval a **iniciuje volbu** nového lídra.



Spuštění voleb

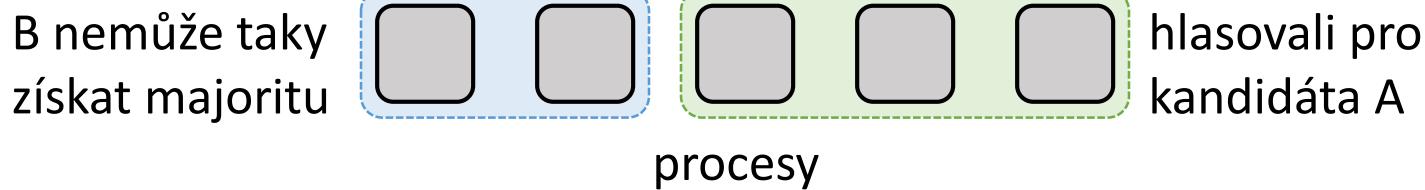
Server, který vyvolá volby, provede následující:

1. Zvýší číslo epochy.
2. Změní svůj stav na KANDIDÁT
3. Zahlasuje pro sebe
4. Pošle **RequestVote** všem ostatním serverům a čeká dokud nenastane jedno z následujících:
 1. Obdrží hlasy od většiny serverů (hladký průběh):
 - Změní stav na LÍDR
 - Pošle **AppendEntries** heartbeats všem ostatním procesům
 2. Přijme zprávu od validního LÍDRA (byl zvolen dříve):
 - Vrátí se do stavu NÁSLEDOVNÍK
 3. Nikdo nevyhraje volby (vyprší volební timeout):
 - Zvýší epochu a začne nové volby

Klíčové vlastnosti voleb

Bezpečnost: maximálně jeden vítěz v každé epoše

- každý proces hlasuje pouze jednou v jedné epoše (a hlas persistuje)
- dva kandidáti nemohou získat většinu v jedné epoše



Živost: jeden z kandidátů musím časem vyhrát

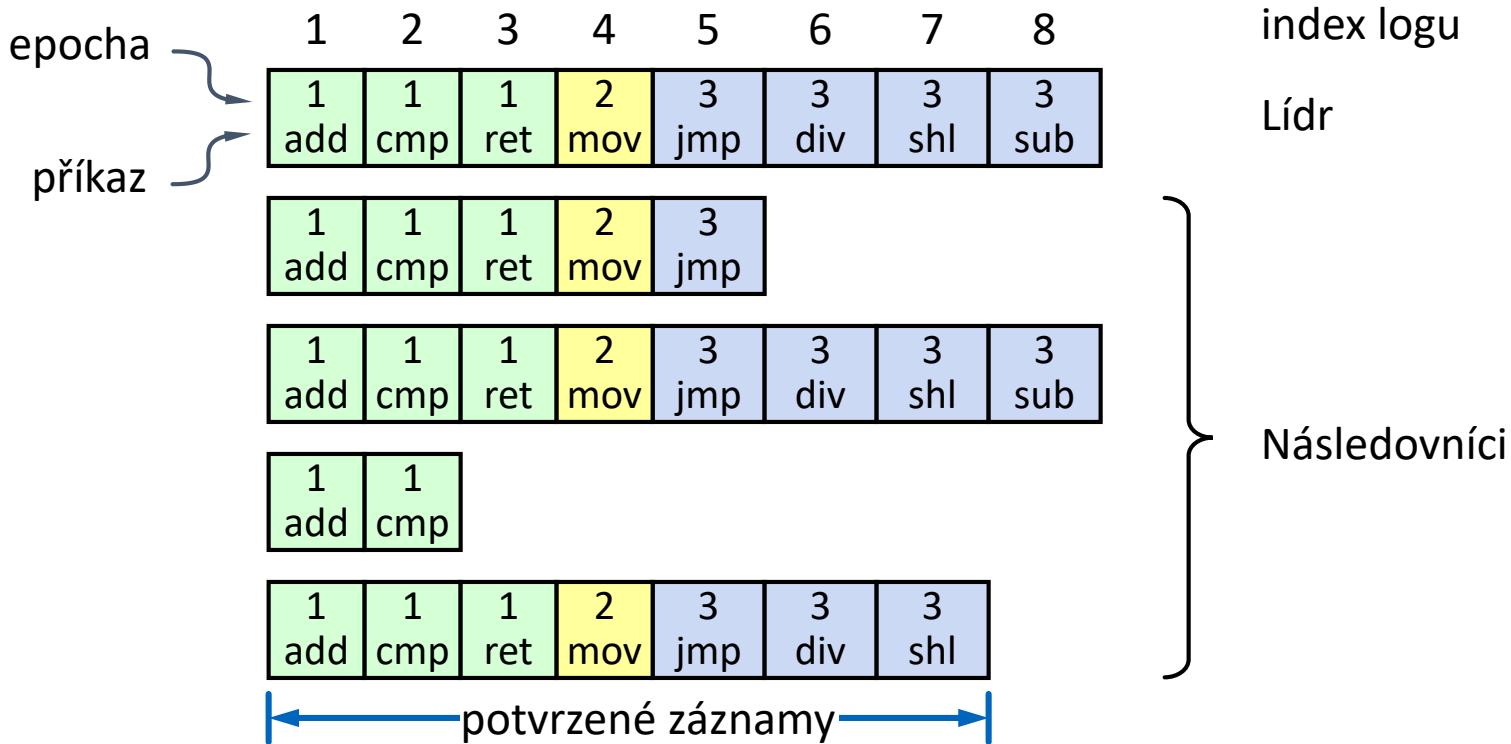
- Každý proces volí volební timeout **náhodně** v intervalu $[T, 2T]$
- Jeden proces typicky iniciuje volby a zvítězí dříve, než ostatní začnou (volby skončí typicky za ms až desítky ms)
- funguje dobře pokud $T \gg RTT$ (čas oběhu zpráv) a pokud $T \ll MTBF$ (střední doba mezi selháními)

<https://raft.github.io/>



Běžný chod

Struktura logů



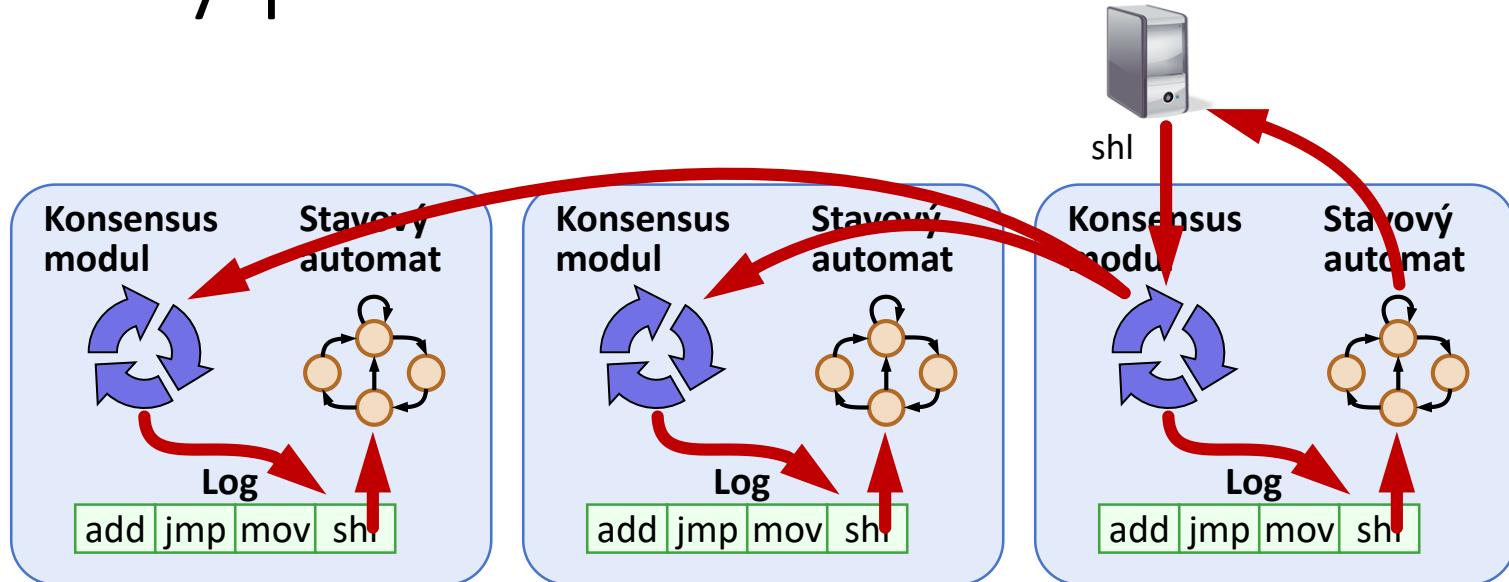
Zážnam v logu = < index, epocha vzniku, příkaz >

Logy jsou uloženy v **perzistentním** uložišti (disk); tj. přežijí havárie

Záznam je **potvrzený (committed)**, je-li známo, že je uložen ve většině procesů

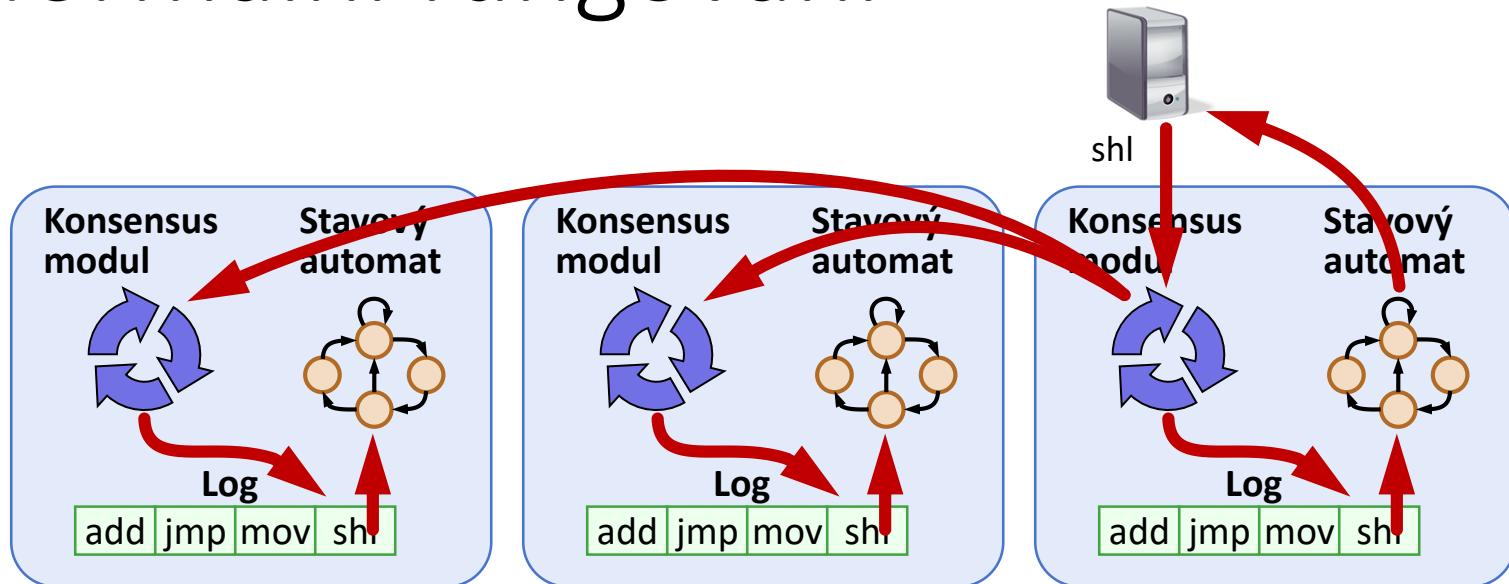
- trvalý: nebude změněn a bude nakonec vykonán stavovým automatem

Běžný provoz



1. Klient pošle příkaz lídrovi.
2. Lídr přidá příkaz na konec svého logu.
3. Lídr pošle zprávu **AppendEntries** následovníkům, typicky paralelně, a čeká na odpovědi.
4. Jakmile je nový záznam potvrzený (committed)
 - Lídr předá příkaz k vykonání svému stavovému automatu a výsledek pošle klientovi.
 - Lídr přidá informaci o potvrzení (commit) do následující zprávy **AppendEntries** pro následovníky
 - Následovníci předají příkaz svým stavovým automatům.

Normální fungování



Havarovaní / pomalí následovníci?

- Lídr opakovaně posílá zprávu **AppendEntries**, dokud doručení neuspěje

V běžném provozu velmi efektivní:

- Stačí úspěšné doručení **AppendEntries** většině procesů a lze odpovědět klientovi

Konzistence logů

	1	2	3	4	5	6
server 1	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
server 2	1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

Raft je navržen tak, aby vynucoval/garantoval následující **invarianty**¹:

1. Mají-li záznamy logů uložené na různých serverech stejný index a epochu, pak
 - obsahují **stejný příkaz**
 - logy jsou **identické** ve všech **předcházejících** záznamech
2. Je-li daný záznam potvrzený, jsou **potvrzené** i všechny **předcházející** záznamy

¹Invariant = vlastnost splněna po celou dobu běhu algoritmu

Kontrola konzistence

	1	2	3	4	5
lídr	1 add	1 cmp	1 ret	2 mov	3 jmp
následník	1 add	1 cmp	1 ret	2 mov	3 jmp

AppendEntries uspěje:
shodný záznam

	1	2	3	4	5
lídr	1 add	1 cmp	1 ret	2 mov	3 jmp
následník	1 add	1 cmp	1 ret	1 shl	

AppendEntries neuspěje:
neshodný záznam

AppendEntries obsahuje $\langle \text{index}, \text{term} \rangle$ záznamu předcházejícího nově přidávané záznamy.

Následovník musí obsahovat **shodný záznam**; jinak je zápis odmítnut.

Kontrola shodnosti předcházejícího záznamu implementuje **indukční krok** a zajišťuje konzistenci logu.



Změna lídra

Konsistence logů

Během **normální** fungování zůstávají logy lídra a následovníků **konzistentní**.

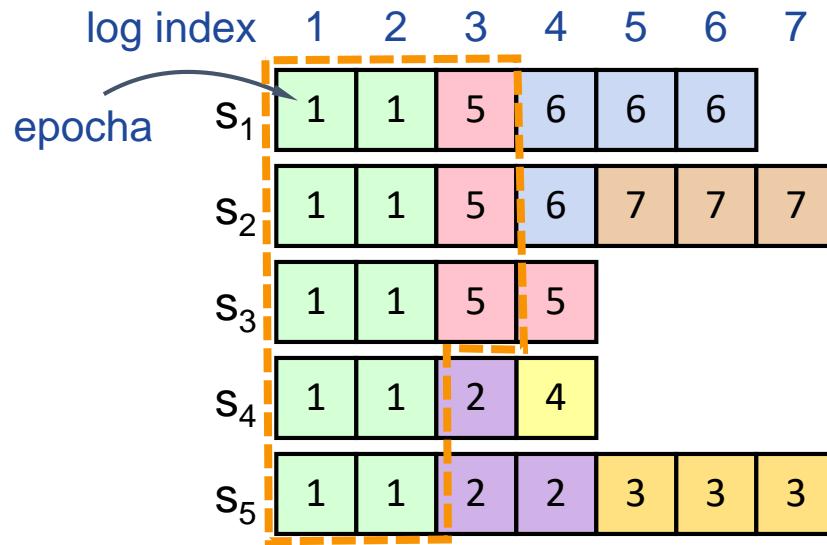
Pokud ovšem lídr **havaruje** a je zvolen nový lídr, mohou být logy lídra a následovníků **nekonzistentní**.

Změny lídra

Log nového lídra vždy reprezentuje „pravdu“ (**správné záznamy**) – po jeho zvolení není potřeba žádné speciální kroky, vykoná logiku běžného chodu.

- logika běžného chodu *časem* udělá logy následovníků identické s logem nového lídra
- záznamy logu předchozího lídra mohou být částečně replikovány (ale nepotvrzeny) – budou postupně eliminovány

Dojde-li k několika haváriím lídrů po sobě, může být v lozích jednotlivých procesů řada přebytečných záznamů, které budou postupně eliminovány.



Bezpečnost

Obecně nutná bezpečnostní garance pro replikaci

Jakmile je příkaz ze záznamu logu vykonán některým stavovým automatem, nesmí žádný jiný stavový automat vykonat *jiný* příkaz pro stejný záznam.

Bezpečnostní invariant Raftu: Jakmile lídr prohlásí záznam v logu za potvrzený, jakýkoliv budoucí lídr bude mít tento záznam ve svém logu.

Invariant Raftu implikuje bezpečnostní garanci:

- lídři **nikdy nepřepisují** záznamy ve svých lozích (pouze přidává)
- pouze záznamy **v logu lídra** mohou být **potvrzeny**
- záznamy (příkazy) musí být v logu **potvrzeny předtím**, než jsou **vykonány** stavovým automatem

Zpřísňení Raftu

Dosavadní logika fungování Raftu bezpečnostní invariant negarantuje!

→ nutno zpřísnit pravidla:

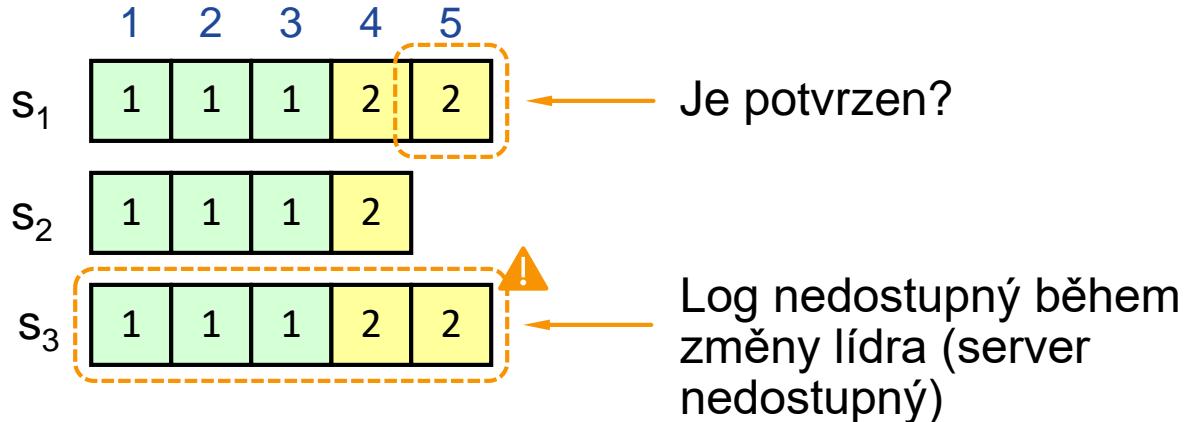
Potvrzený → Přítomný v logu jakéhokoliv budoucího lídra

Zpřísňení definice
potvrzení

Zpřísňení pravidla
volby lídra

Zpřísnění výběru lídra

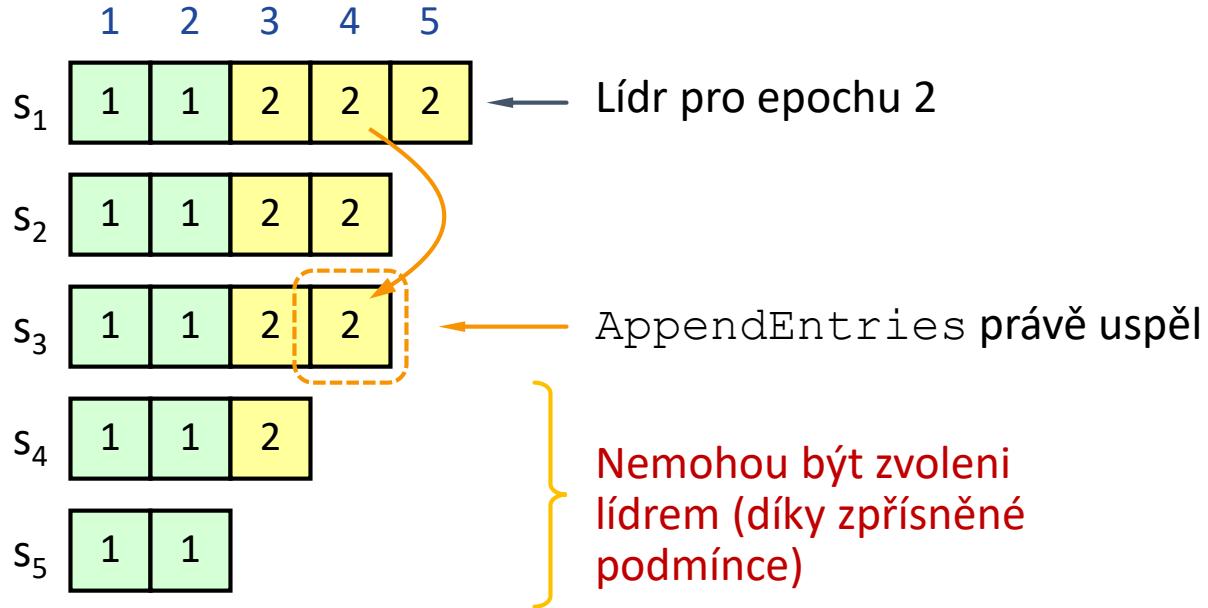
Nelze říct, které záznamy byly potvrzeny



Raft volí kandidáta, který má nejúplnější log. Kandidáti do zprávy **RequestVote** vloží index a epochu posledního záznamu svého logu

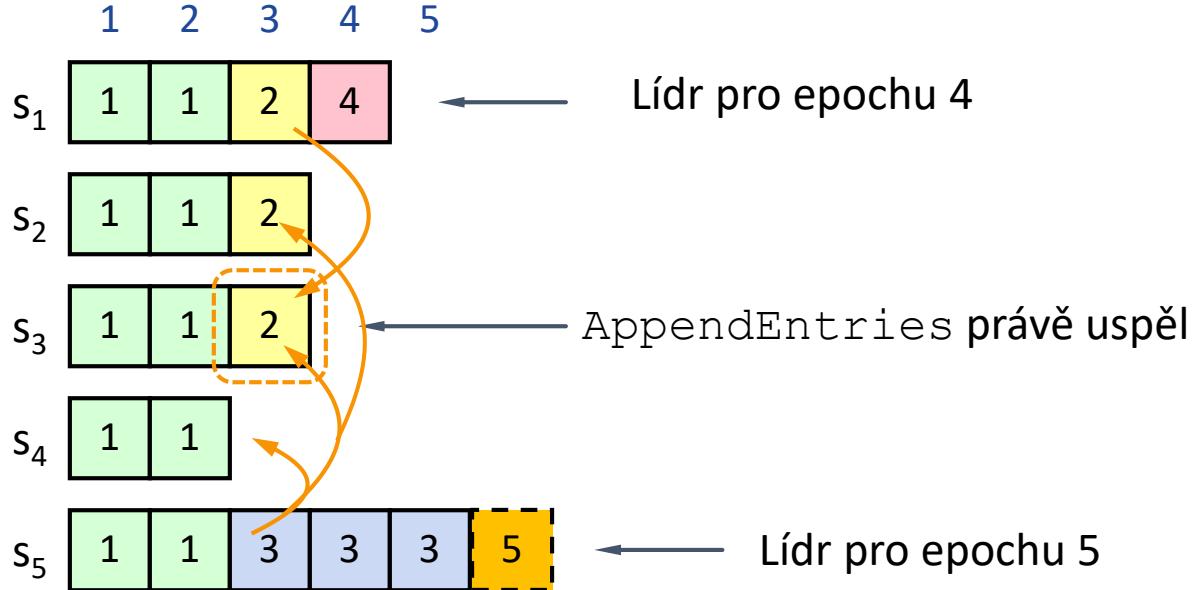
- Volící server hlas pro kandidáta odmítne, pokud jeho vlastní log je **úplnější**, tj. pokud má na konci záznam s vyšší epochou nebo stejnou epochou, ale vyšším indexem.
- Lídr tedy bude mít **nejúplnější log** mezi většinou procesů, kterou byl zvolen.

Potvrzování záznamu z aktuální epochy



Záznam 4 je bezpečně potvrzen: jakýkoliv lídr pro epochu tři musí obsahovat v logu záznam 4.

Potvrzování záznamu z dřívější epochy



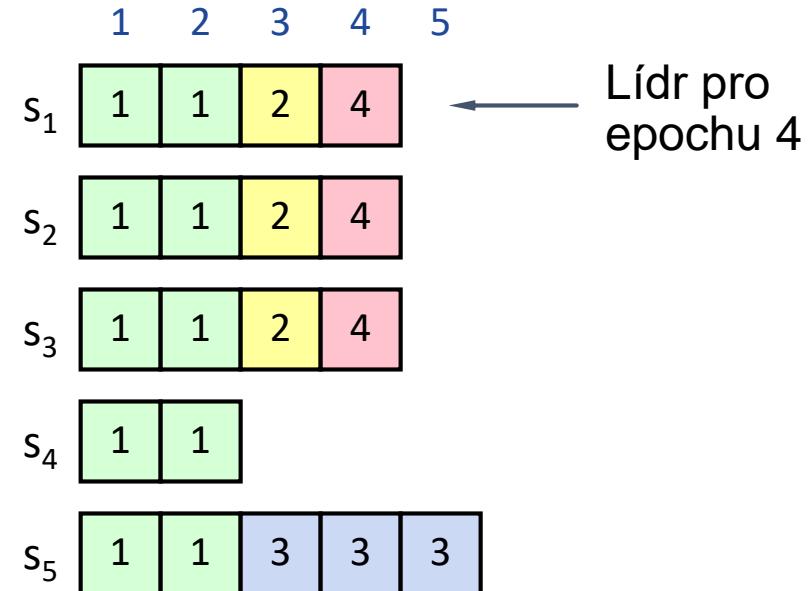
Záznam 3 není bezpečně potvrzený

- S_5 může být zvolen jako lídr pro epochu 5
- Byl by-li zvolen, přepíše záznam 3 v S_1, S_2, S_3

Nová pravidla pro potvrzování

Aby lídr považoval záznam za potvrzený:

1. záznam musí být uložený na většině serverů
2. **aspoň jeden nový záznam** z lídrovy aktuální epochy musí být taky na většině serverů



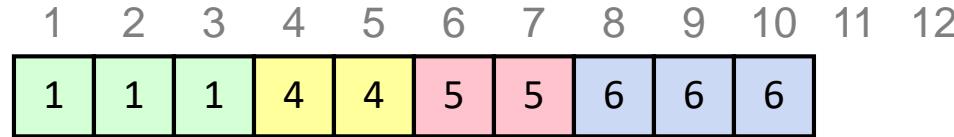
Příklad: Jakmile je záznam 4 potvrzen, S_5 nemůže být zvolen lídrem pro epochu 5 a záznamy 3 a 4 jsou bezpečně potvrzeny.

Kombinace nové pravidla pro výběr lídra a zpřísňené definice potvrzování **garantuje bezpečnostní invariant Raftu**

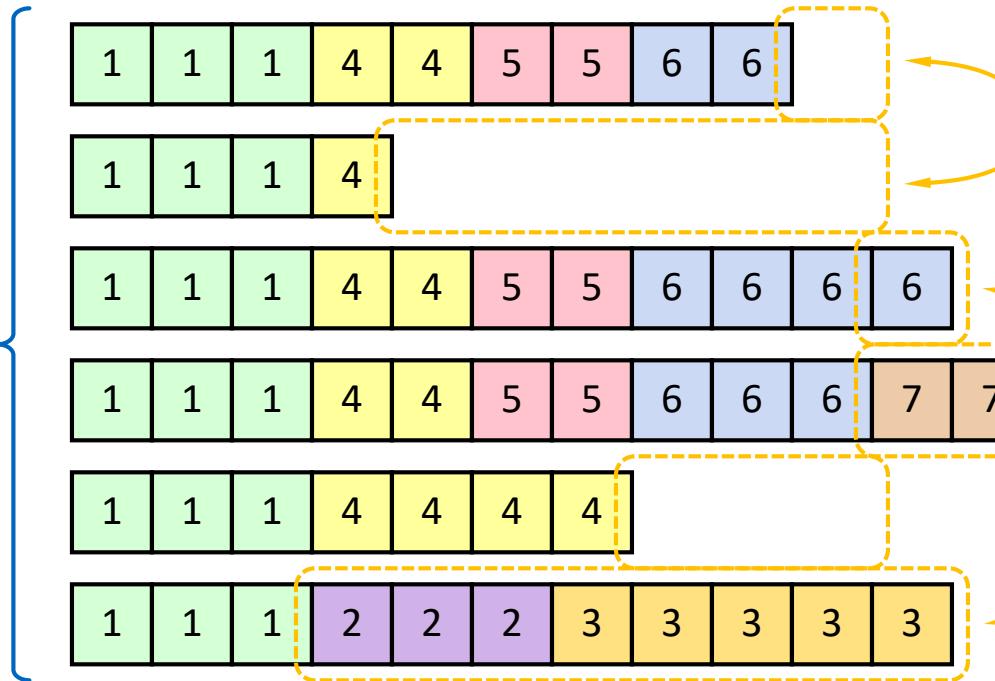
Komplikace: nekonzistence logu

Změna lídra mohou vést k nekonzistencím logu

Lídr pro epochu 8



Možní
následovníci



Chybějící
záznamy

Přebytečné
záznamy

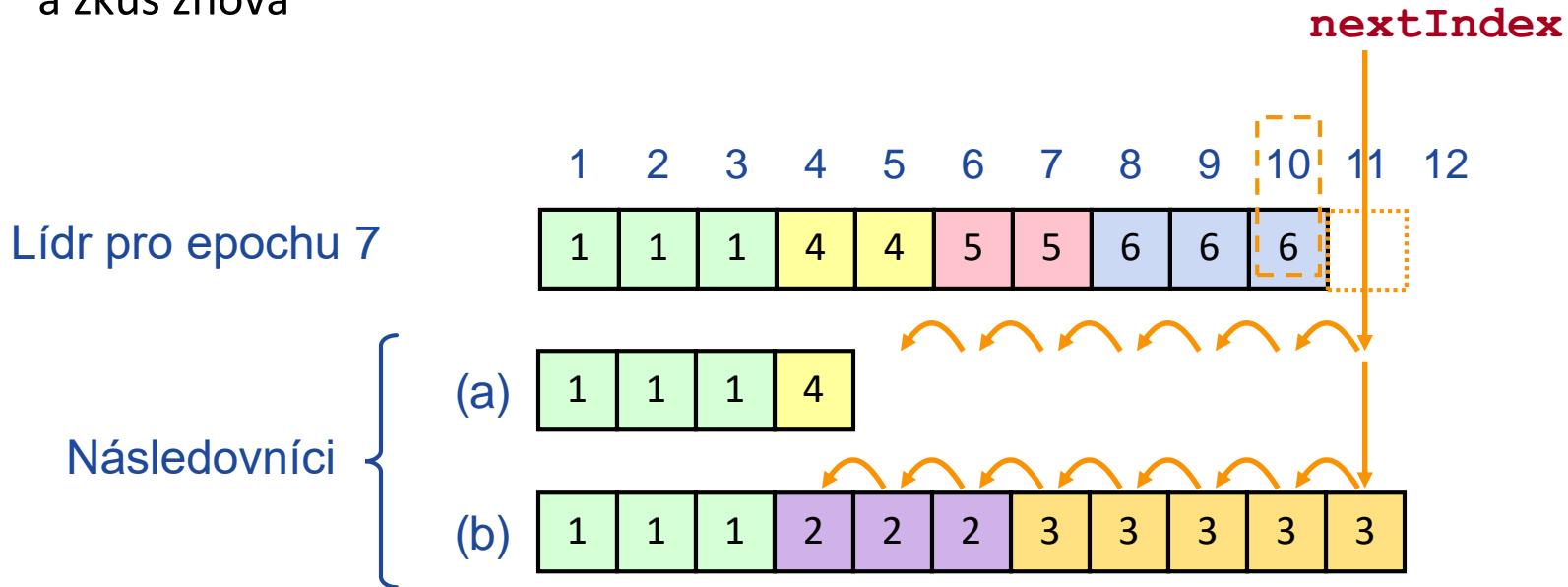
Oprava logů následovníků

Nový lídr musí udělat logy následovníků konzistentní se svým logem, tj. smazat přebytečné záznamy a doplnit chybějící záznam.

Lídr udržuje proměnou **nextIndex** pro každého následovníka:

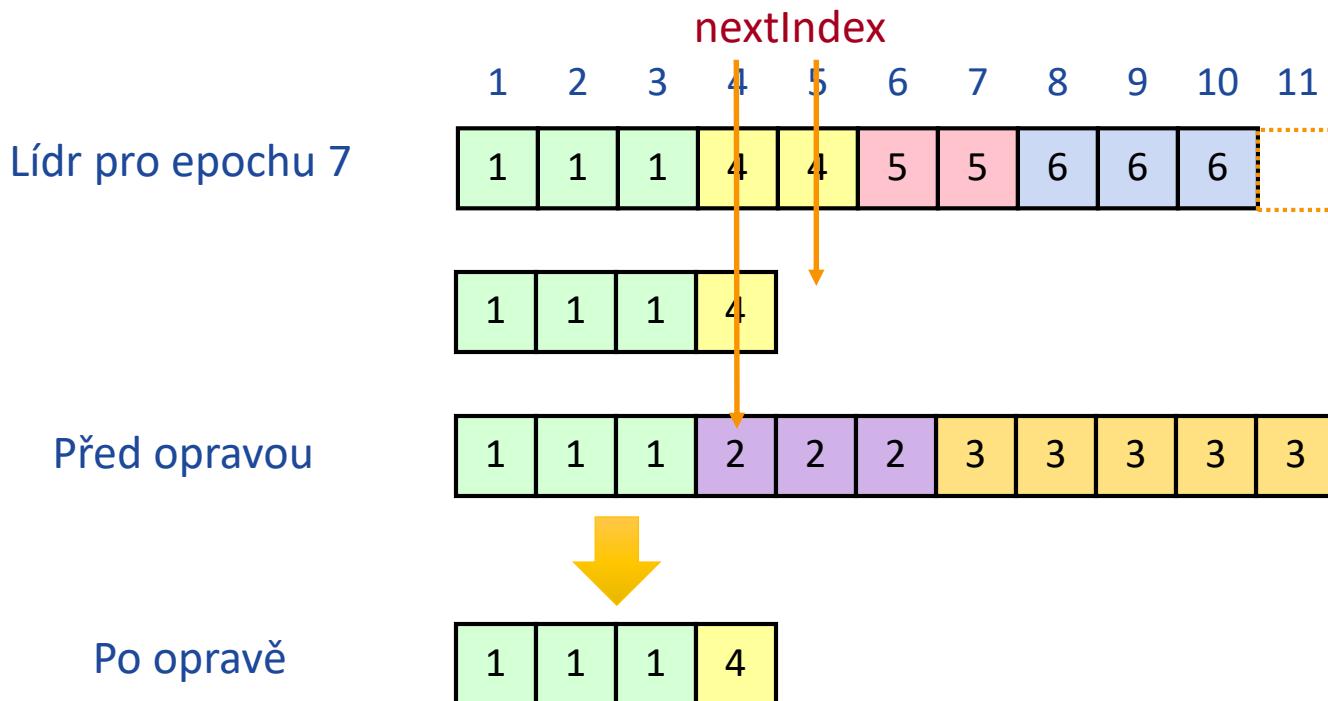
- index další záznamu logu, který by měl být odeslán následnikovi
- inicializován na $1 + \text{index poslední záznamu lídra}$

Pokud kontrola konzistence **AppendEntries** selže, s niž nextIndex o jednu a zkuste znova



Oprava logů následovníků

Pokud následovník přepíše nekonzistentní záznam, odstraní i **všechny následující záznamy**.





Neutralizace starého lídra

Neutralizace starých lídrů

Sesazený lídr **nemusí** být **trvale** havarovaný

- přechodné odpojení od sítě
- jiné procesy zvolí nového lídra
- starý lídr se znovu připojí a pokusí se potvrzovat svoje záznamy

Epochy slouží k **detekci neaktuálního** lídra

- každá zpráva obsahuje epochu odesílatele
- je-li epocha odesílatele starší, zpráva je odmítnuta, odesílatel se změní na Následovníka a aktualizuje si epochu
- je-li epocha příjemce starší, tak příjemce se změní na Následovníka, aktualizuje si epochu a následně zprávu normálně zpracuje

Volby aktualizují epochy většiny serverů → sesazení lídři nemohou potvrdit nové záznamy



Klientský protokol

Protokol klienta

Klienti posílají příkazy lídrovi

- Není-li lídr známý, kontaktují libovolný server a ten je případně přesměruje na lídra

Lídr pouze vrací odezvu na příkaz poté, co je příkaz **zalogován, potvrzen** a následně vykonán **lídrem**.

Pokud **nepřijde** v časovém limitu **odezva** na požadavek (např. lídr havaroval):

- klient vybere (náhodně) jiný server
- a po případném přesměrování nakonec odešle příkaz novému lídrovi

Protokol klienta: jediné vykonání

Lídr může havarovat poté, co vykonal příkaz, ale před odesláním odpovědi

→ Riziko **opakovaného vykonání** příkazu.

Řešení: Pro zajištění právě **jednoho vykonání** příkazu klient vloží unikátní ID příkazu do každého požadavku

- Toto ID je uložené v záznamech v logu

Před přijetím požadavku lídr zkонтroluje, zda-li už nemá záznam se stejným ID ve svém logu

- Pokud ne → příkaz vykoná;
- Pokud ano → příkaz odmítne.

Souhrn

Problém konsensu je v jádru mnoha problémů v DS.

V asynchronním DS **nelze** při přítomnosti selhání **konsensus vyřešit** ve smyslu bezpečnosti a živosti.

Praktická řešení garantují **bezpečnost**.

Raft je moderní algoritmus pro replikaci logů / výpočtů.

Je využíván v řadě reálných DS.

Literatura:

- [Raft] Ongaro, D. and Ousterhout, J.K., 2014, June. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*. [\[link\]](#)

PDV 13 2019/2020

Volba lídra (koordinátora)

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



Motivace

Design distribuovaných algoritmů pro řadu problémů distribuovaných výpočtů se **zjednoduší**, když jsou **asymetrické** a předpokládají, že jeden z procesů má roli **lídra** (a s ní spojenou logiku).

- konsensus, replikace, vyloučení procesů, ...

V situacích, kdy uvažujeme **selhání procesů**, musím být schopni lídra dynamicky nahradit.

Problém volby lídra

Ze skupiny procesů **vybrat lídra** (který bude řešit specifické úkoly) a **dát vědět všem procesům** ve skupině, kdo je lídrem.

Co se stane, když lídr selže?

- nějaký proces detekuje pomocí detektoru selhání a spustí nové volby

Algoritmus pro volbu lídra musí zajistit:

1. zvolí právě jednoho lídra z bezvadných procesů
2. všechny bezvadné procesy ve skupině se shodnou na tom, kdo je lídr

Systémový model

Skupina N procesů s unikátními identifikátory.

- známe všechny procesy, ale nevíme, které jsou aktivní (bezvadné)

Procesy **mohou havarovat**.

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**.

Další požadavky

Každý proces může vyvolat volby.

Jeden proces může vyvolat v jeden okamžik pouze **jedny volby**.

Více procesů může vyvolat volby **současně** - pak požadujeme, aby se nakonec shodly na jednom lídrovi.

Výsledek volby lídra by **neměl záviset** na tom, který proces volby **vyvolal**.

Po skončení běhu algoritmu volby lídra má každý proces ve své proměnné *ELECTED* identifikátor lídra s nejvyšší hodnotou volebního kritéria (a nebo nikdo, tj. volby skončily neúspěšně).

Volební kritérium:

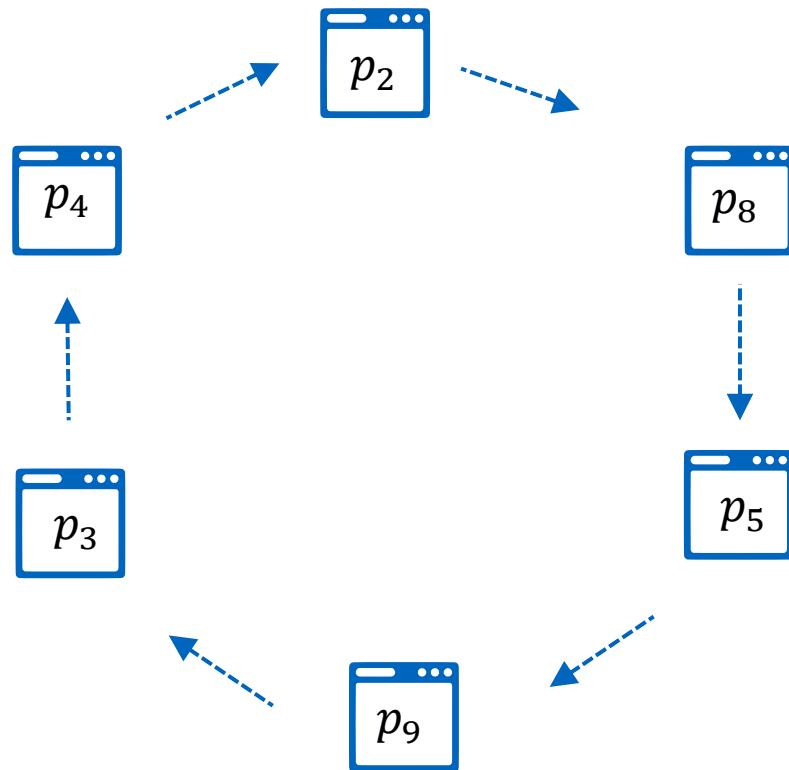
- typicky nejvyšší identifikátor, tj. IP adresa
- ale taky např: nejvíce RAM, diskového prostoru, nejvíce souborů v případě P2P sítí nebo nejúplnější log v případě RAFTu
- musí být fixní a známe všem procesům při zahájení volby

Kruhový algoritmus

Procesy jsou uspořádané do logického kruhu.

Zprávy jsou posílány dokola kruhu **v jednom směru**.

Procesy jsou schopné **detektovat selhání** ostatních procesů



Kruhový algoritmus

P_i : zahájení voleb (po detekci selhání)

P_i pošle po kruhu zprávu ELECTION(i) obsahující jeho identifikátor

P_i : zpracování zprávy ELECTION(j)

if $i < j$ **then** přepošle zprávu ELECTION(j)

if $i > j$ **then** nahradí zprávu za ELECTION(i) a pošle dál

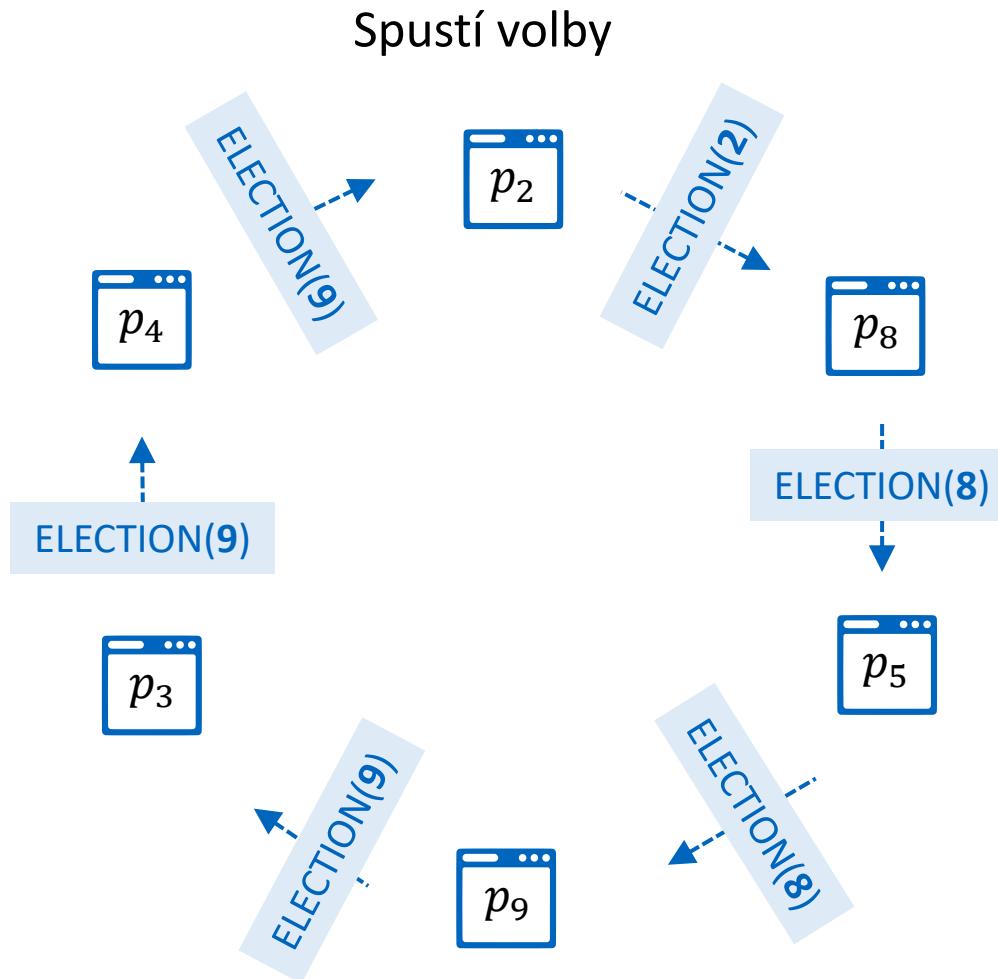
if $i = j$ **then** P_i je nový koordinátor; odešle zprávu ELECTED(i)

P_i : přijetí zprávy ELECTED(j)

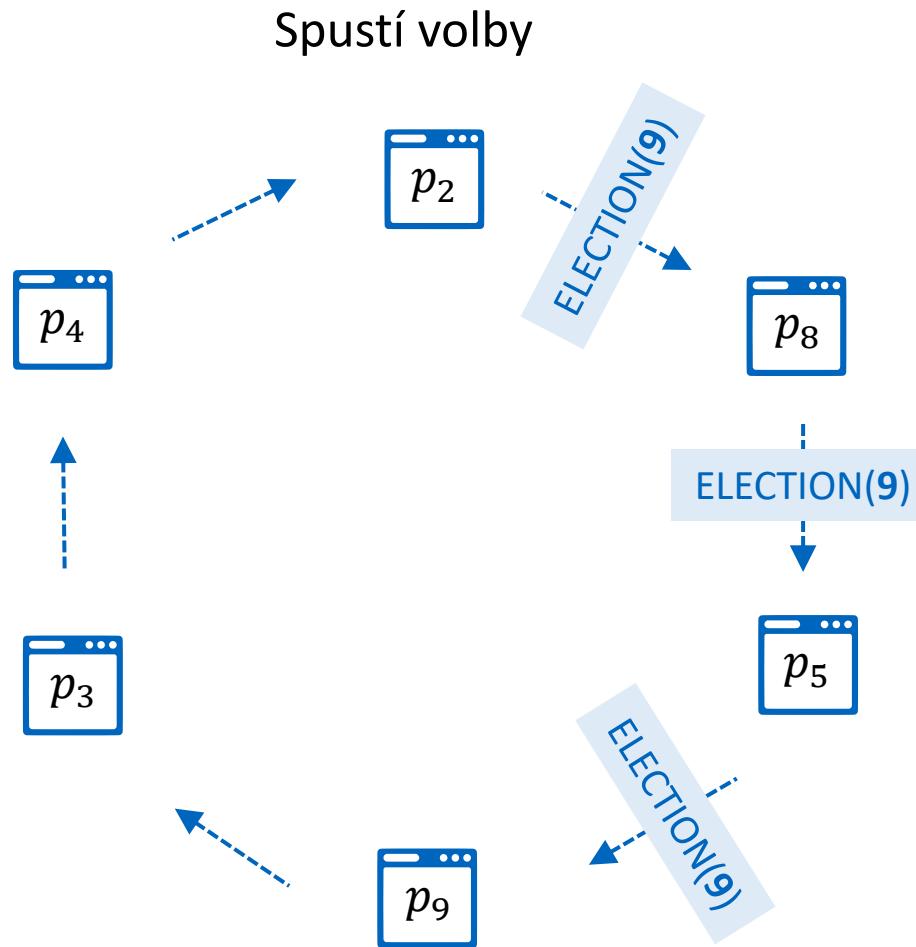
Nastaví svou proměnou $\text{elected}_i := j$

if $i \neq j$ **then** přepošle zprávu ELECTED(j)

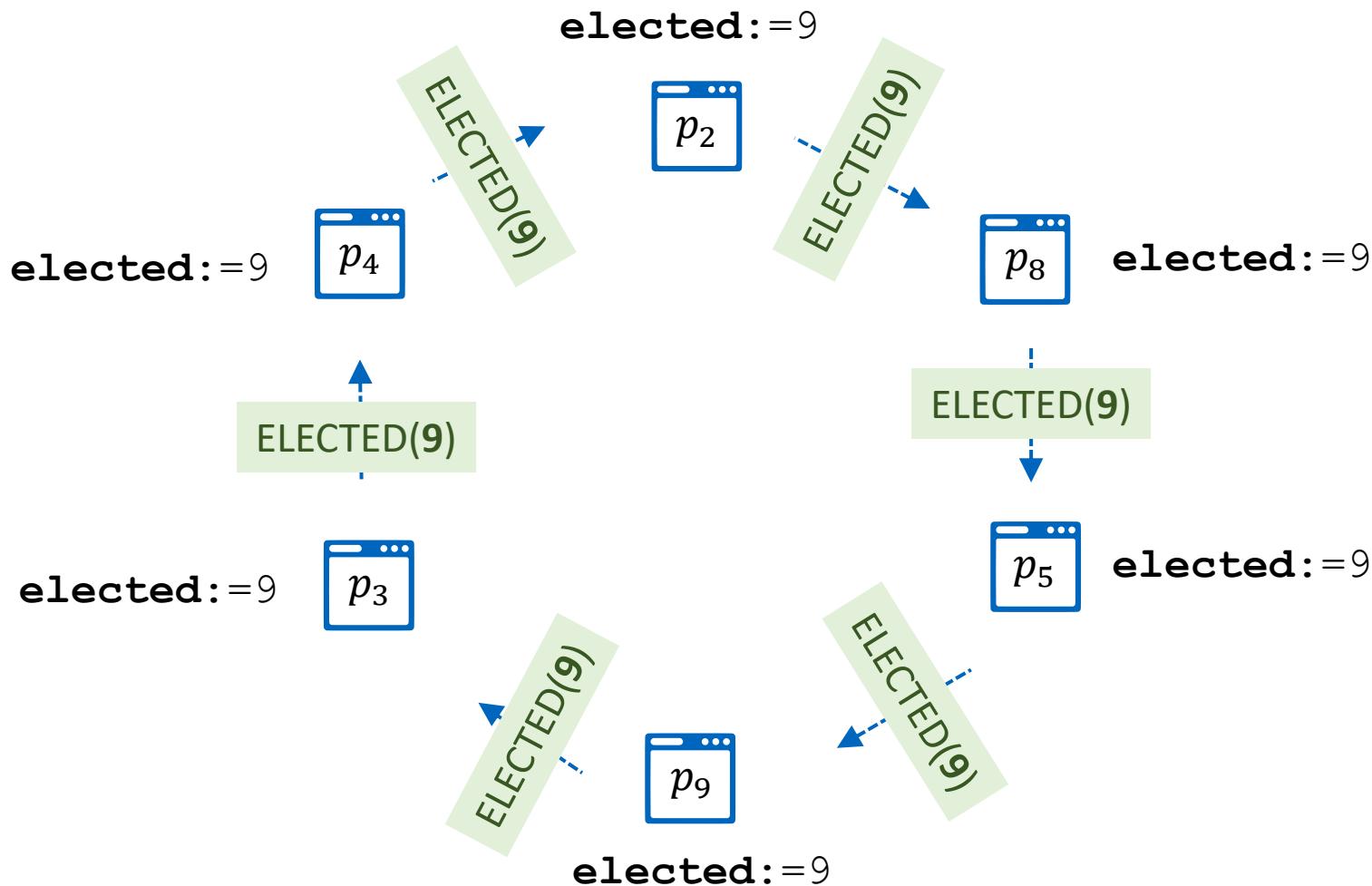
Příklad



Příklad



Příklad



Analýza

N procesů; předpokládáme, že v průběhu volby nedochází k selháním

Nejhorší případ

- iniciátor je následník budoucího lídra
- $N - 1$ zpráv pro zprávu ELECTION než se dostane od iniciátora P_i k budoucímu lídrovi P_j
- N přeposlání nezměněné zprávy ELECTION(j) okolo kruhu
- N přeposlání zprávy ELECTED(j) okolo kruhu
- Celkem $3N - 1$ zpráv
- Čas běhu: $3N - 1$ komunikačních latencí

Nejlepší případ

- iniciátor je budoucí lídr
- Celkem $2N$ zpráv
- Čas běhu: $2N$ komunikačních latencí

Pokud nejsou selhání, tak volba skončí **v konečném čase** (živost). Pokud volby skončí úspěšně, tak všechny procesy znají **stejného lídra** (bezpečnost).

Více iniciátorů

Proces si pamatují nejvyšší $\max ID$ ze všech dosud obdržených ELECTION/ELECTED zpráv (od poslední úspěšné volby lídra).

Procesy ignorují ELECTION/ELECTED zpráv s nižším ID než $\max ID$.

Dojde k rychlejší eliminaci zpráv s ELECTION zpráv s ID procesů, které nemohou být zvoleny.



Bully Algorithmus

Bully algoritmus

Klasický algoritmus pro volbu lídra.

Základní princip: proces, který **detekoval selhání** dosavadního lídra, **vyzve** procesy s **vyšším ID** ve volbách.

Bully algoritmus

P_i : zahájení voleb (po detekci selhání nebo jako reakce na volby)

if P_i má nejvyšší ID
Pošli COORDINATOR zprávu všem procesům s nižšími identifikátory (volby končí).
else // zahájí volby
Pošli ELECTION zprávu všem procesům s vyšším ID
// následně čekání na odpověď

P_i : reakce na ELECTION

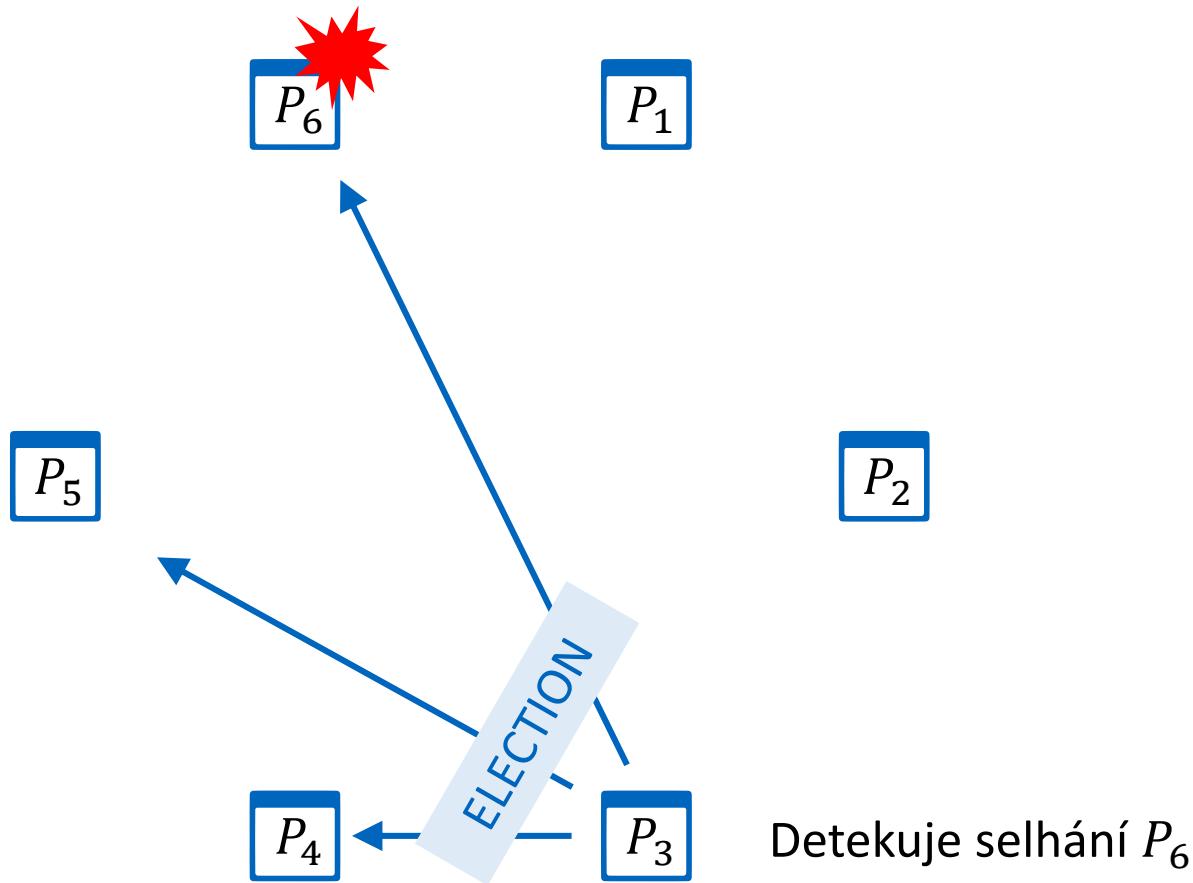
Odpověz OK
If pokud P_i dosud nezahájil volby
zahaj volby

P_i : čekání na odpovědi
(po vyvolání voleb)

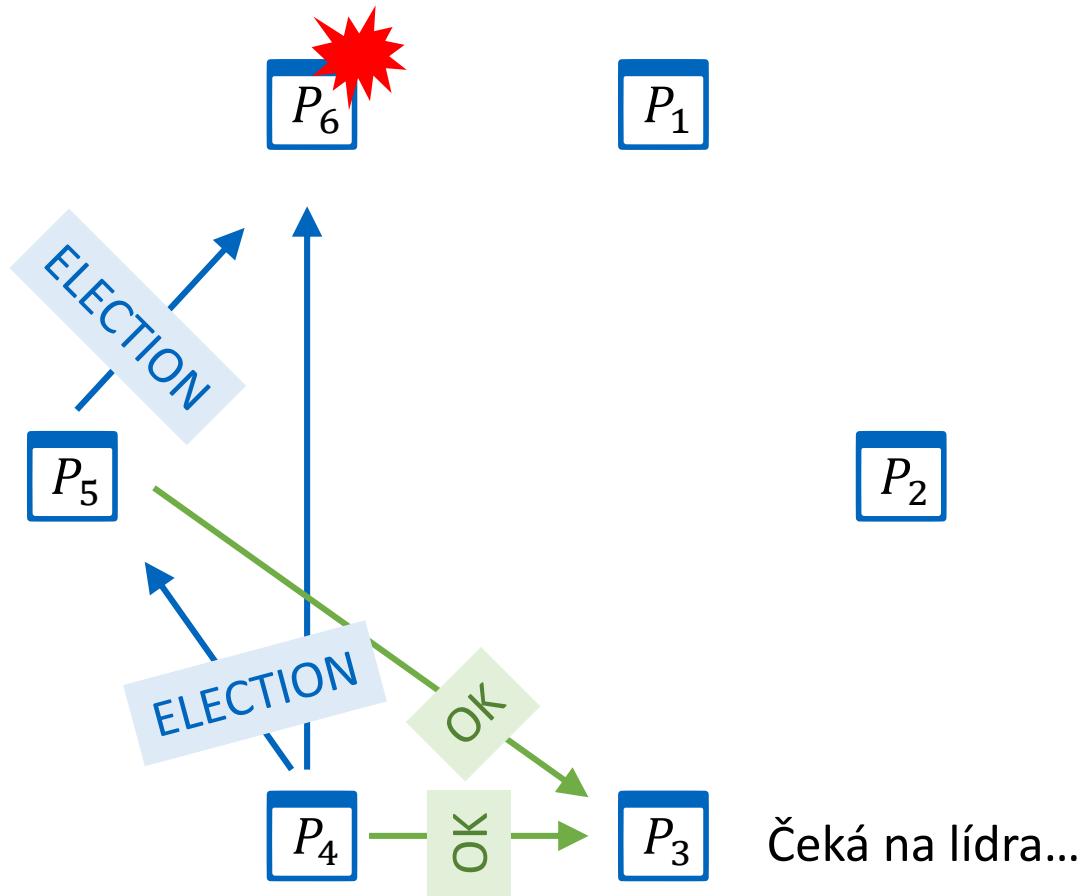
if nedorazí žádná odpověď v časovém limitu
prohlaš se jako P_i za lídra;
pošli COORDINATOR zprávu všem procesům s nižším ID;
// volby skončily

else // existuje aktivní proces s vyšším ID než P_i
čekej na zprávu COORDINATOR;
pokud nedorazí v časovém intervalu, iniciuj nové volby

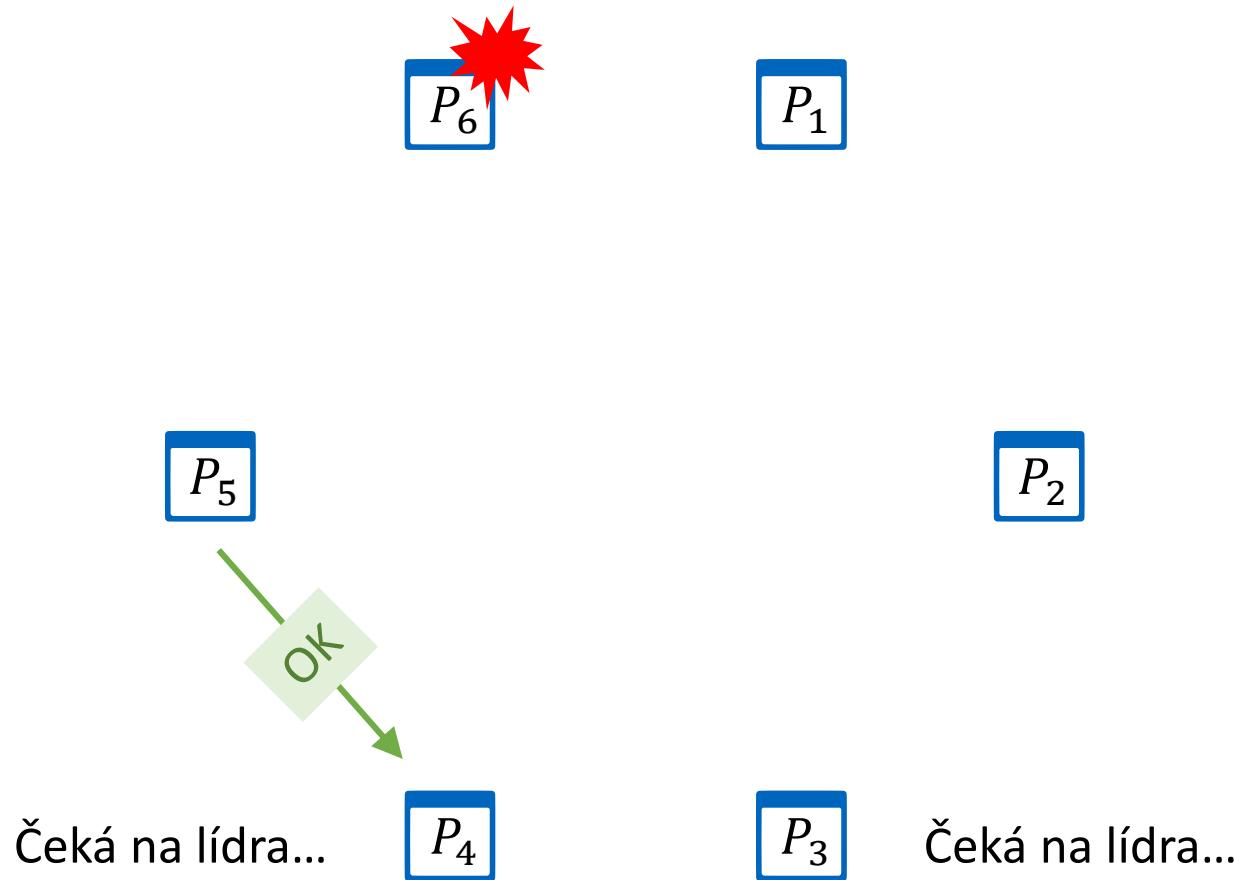
Bully algoritmus



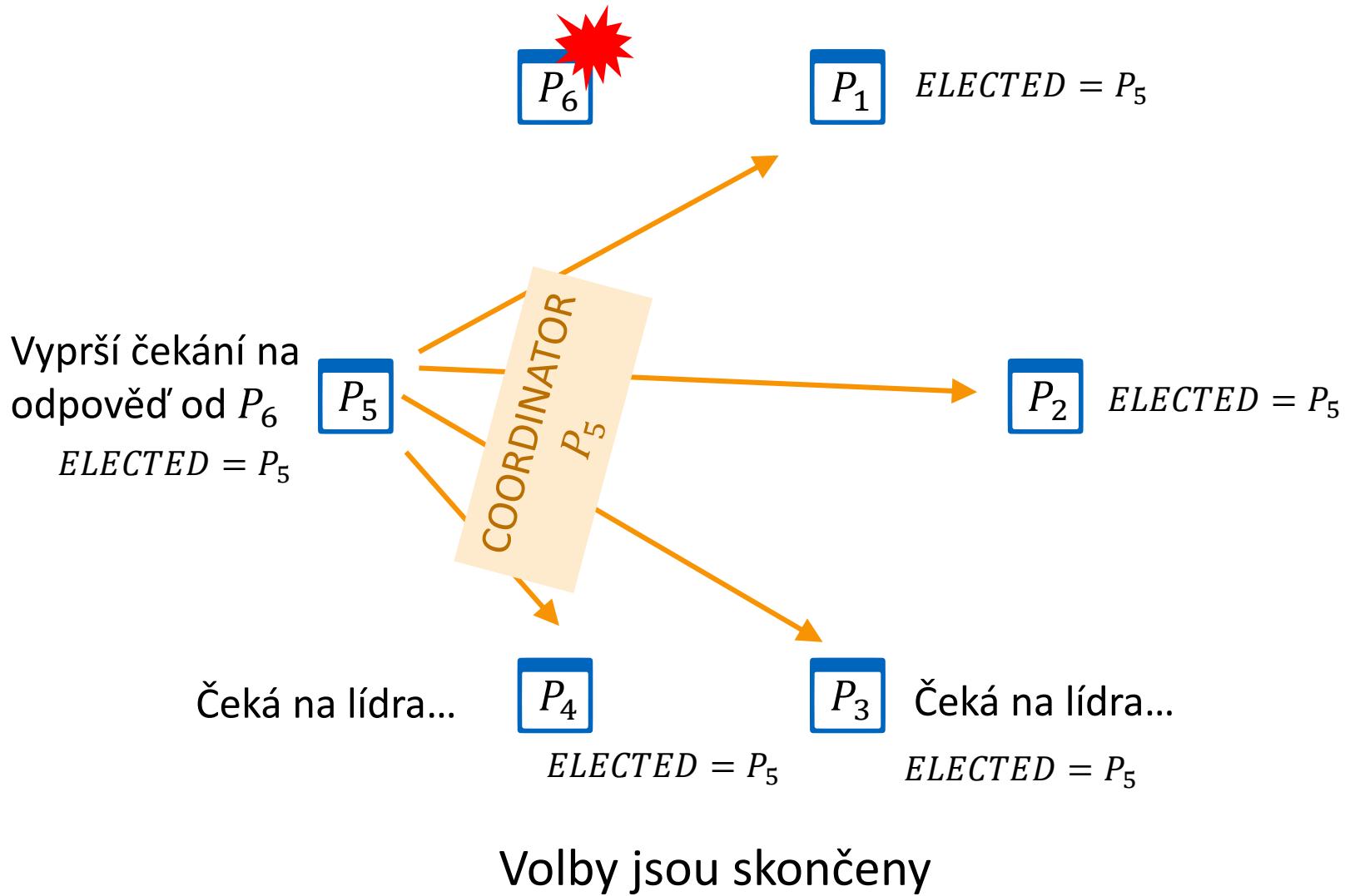
Bully algoritmus



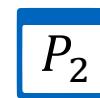
Bully algoritmus



Bully algoritmus



Selhání během volby

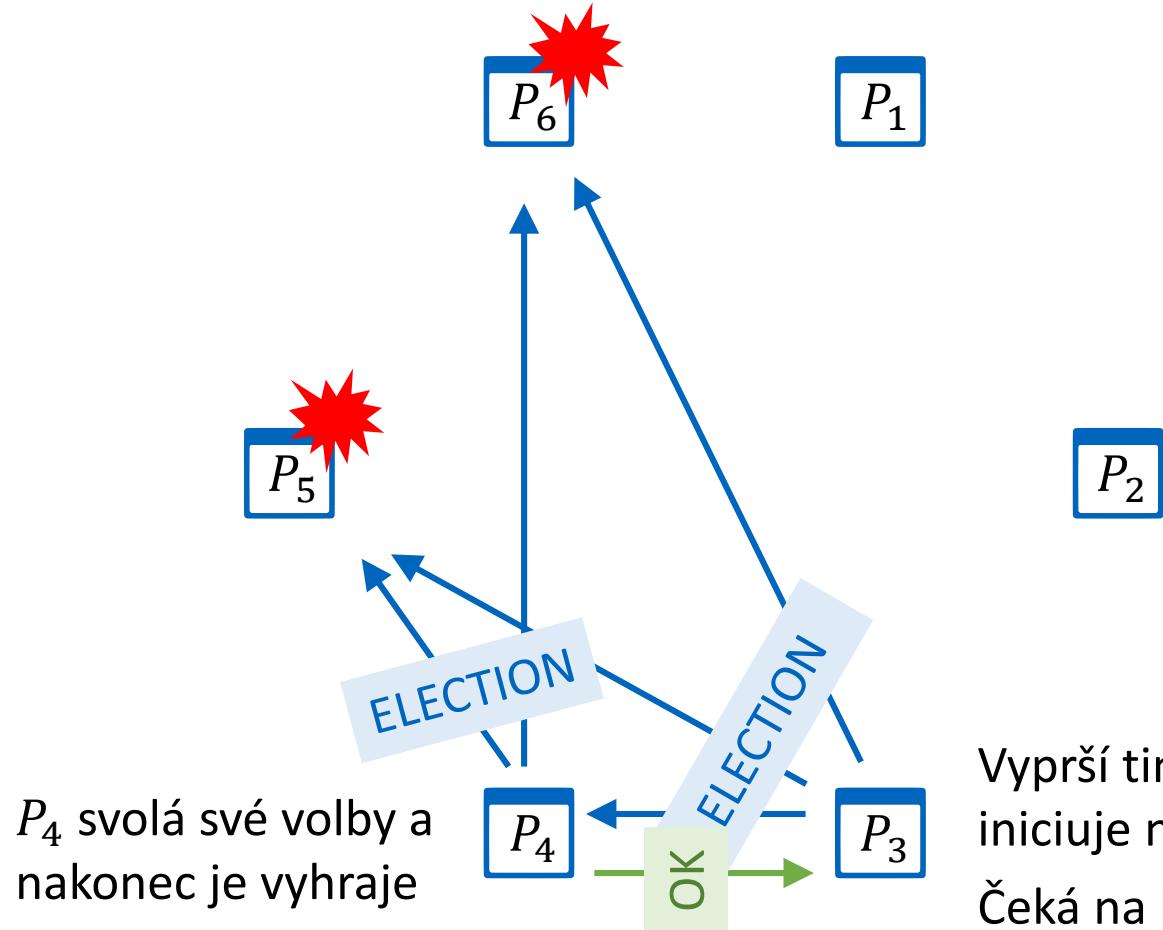


Čeká na lídra...

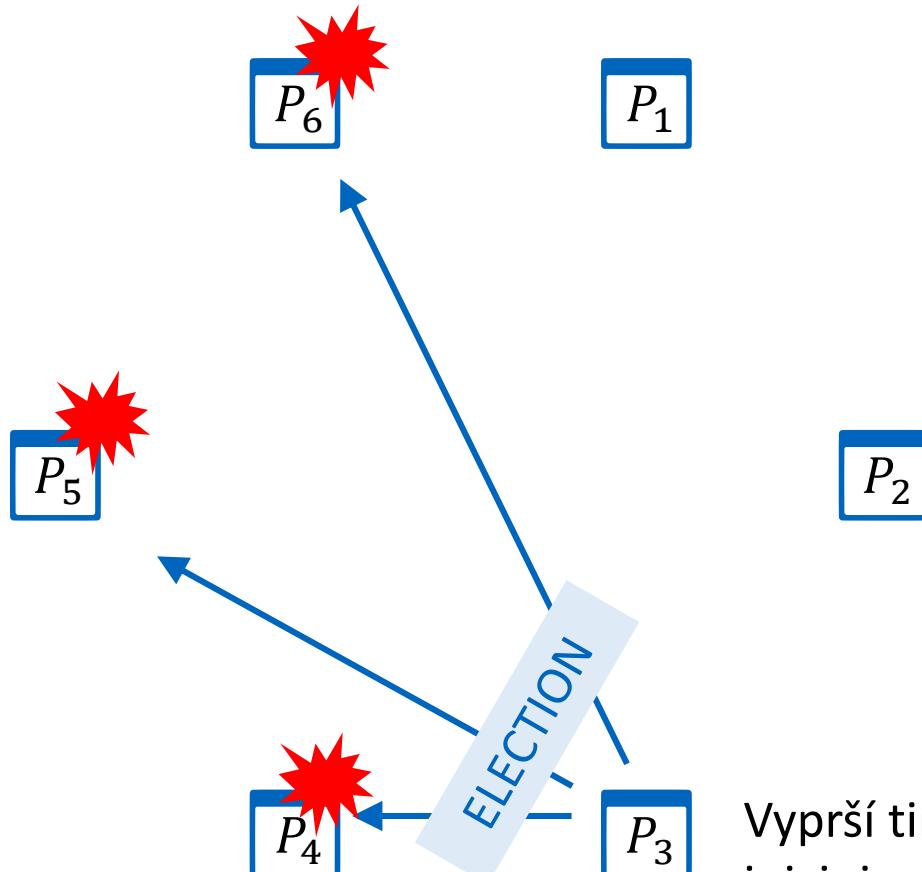


Čeká na lídra ...

Selhání během volby



Selhání během volby



Vyprší timeout →
iniciuje nové volby a
nakonec je vyhraje

Analýza

Nejhorší případ – selhání lídra detekováno procesem s nejnižším ID.

Pro dokončení volby lídra je v nejhorším případě potřeba **čtyři komunikační latence**:

1. Proces s nejnižším ID pošle ELECTION zprávu ostatním procesům
2. Proces s druhým nejvyšším ID:
 - pošle OK procesům s nižšími ID
 - pošle ELECTION procesu s nejvyšším ID
3. Procesu s druhým nejvyšším ID vyprší timeout při čekání na odpověď procesu s nejvyšším ID
4. Proces s druhým nejvyšším ID rozešle zprávu COORDINATOR

Nejhorší případ – komunikační zátěž

- celkem $N - 1$ procesů pošle zprávu ELECTION procesům s vyšším ID
- tj. celkem: $N - 1 + N - 2 + \dots + 1 = \frac{N(N-1)}{2} = O(N^2)$ zpráv

Analýza

Nejlepší případ: proces s druhým nejvyšším ID detekuje selhání lídra

Nejlepší případ - komunikační zátěž:

- $(N - 2)$ zpráv COORDINATOR
- čas do ukončení: **1 komunikační latence**

Bezpečnost

Bezpečnost volby lídra: Každý proces buď zvolí stejného lídra L nebo nezvolí žádného lídra.

Bully algoritmus bezpečnost **garantuje** pokud se **nerestartují** havarované procesy se stejným ID.

Živost

Bully algoritmus pracuje s timeouts → v **asynchronním systému** jeho běh nemusí nikdy skončit → **živost není garantována**.

V synchronním systému:

- lze spočítat nejhorší jednosměrnou latenci = doba přenosu zprávy + doba reakce na zprávu.
- pokud timeout nastavíme na násobek nejhorší jednosměrné latence, je **živost garantována**.

Proč je volba lídra tak těžké?

Protože souvisí s problémem **konsensu**!

Pokud bychom byli schopni vyřešit volbu lídra, tak umíme řešit konsensus.

Stačí zvolit lídra a poslední bit jeho ID interpretovat jako výsledek konsensu.

Ale protože konsensus není řešitelný v asynchronních DS se selháními, není řešitelná ani volba lídra.

(řešitelnost = existence algoritmu garantujícího bezpečnost i živost současně)

Souhrn

Volba lídra důležitý problém v DS.

Bully algoritmus klasický algoritmus předpokládající selhání procesů, ale perfektní FIFO kanál.

Bully algoritmus garantuje **bezpečnost** (za předpokladu, že se havarované procesy nerestartují se stejným ID).

V asynchronním systému **negarantuje živost**; v synchronním nebo částečně synchronním systému lze (konečnou) živost (tzv. eventual liveness) dosáhnout vhodným **nastavením timeout**.

PDV 13 2019/2020

Závěr a shrnutí

Michal Jakob

michal.jakob@fel.cvut.cz

Centrum umělé inteligence, katedra počítačů, FEL ČVUT



Hlavní výzvy DS

Asynchronicita

Selhání

Selhání

Jak procesy, tak komunikační kanály mohou v DS selhat.

Selhání procesu

- **havárie (crash/fail-stop):** proces přestane vykonávat algoritmus (a reagovat na zprávy)
- **libovolné (byzantské) selhání:** proces může pracovat dále (a reagovat na zprávy), ale vykonává chybný algoritmus (z důvodu softwarový chyby nebo úmyslu)

Selhání kanálu

- **ztráta zprávy (message drop):** zpráva není doručena cílovému procesu (např. kvůli přetížení sítě nebo přetečení zásobníku v OS u přijímacího procesu)
- **rozdělení (partitioning):** procesy jsou rozdělené do disjunktních množin (oddílů - partitions) tak, že v rámci oddílu je komunikace možná, ale mezi oddíly nikoliv

V případě synchronních DS definujeme ještě **selhání časování**, pokud doba odezvy procesu nebo přenosu zprávy po síti vybočila z dohodnutého **časového rozmezí**.

Synchronní vs. Asynchronní

Asynchronní systém

- Žádné časové limity na **relativní rychlosť** vykonávání procesů.
- Žádné časové limity na **trvání přenosu** zpráv.
- Žádné časové limity na **časový drift** lokálních hodin



Synchronní systém

- **Synchronní výpočty:** známe horní limit na relativní rychlosť vykonávání procesů.
- **Synchronní komunikace:** známé horní limit na dobu přenosu zpráv.
- **Synchronní hodiny:** procesy mají lokální hodiny a je znám horní limit na rychlosť driftu lokálních hodin vzhledem k globálním hodinám.

Dále: Částečně synchronní systém

Korektnost v DS

Živost (Liveness)

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

(živost prakticky souvisí s **dostupností** systému)

Bezpečnost (Safety)

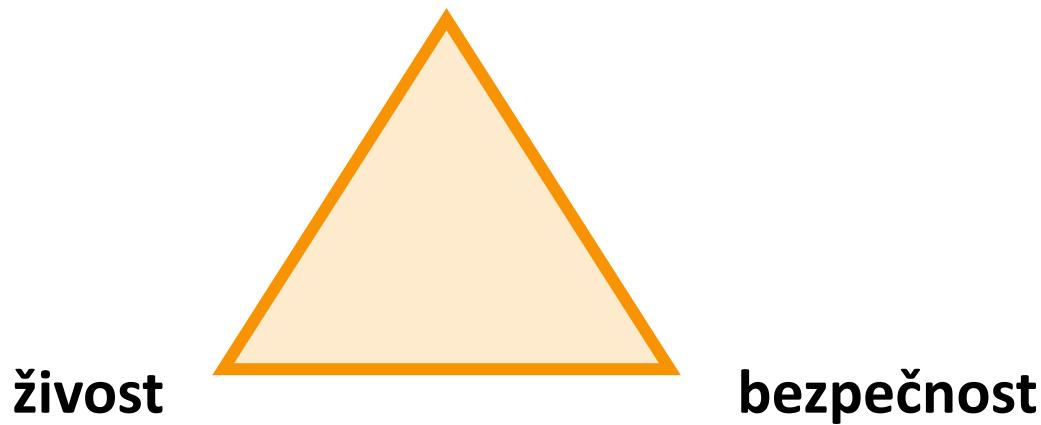
Garance, že v DS *nikdy* nedojde k něčemu **špatnému** (nebude dosažen nežádoucí stav).

FLP teorém

FLP teorém

V asynchronním distribuovaném systému **nelze dosáhnout současně bezpečnosti a živosti** distribuovaného výpočtu, pokud v něm může docházet k selháním (byť i jediného procesu).

odolnost vůči selháním



Řešitelnost problémů

V praxi vždy vyžadujeme bezpečnost a díky částečné synchronicitě ve velkém množství běhu distribuovaných algoritmů dosáhneme výsledků v **konečném čase (tzv. konečná živost – eventual liveness)**.

- existují i pravděpodobnostní algoritmy mající **konečnou střední hodnotu běhu**

Problémy

Problém (algoritmy)	Model (zjednodušeně)	Garance
Detekce selhání (centrální, kruhový, all-to-all, SWIM)	asynchronost + selhání	živost
Kauzalita a čas (fyzikální, Lamportovy, vektorové hodiny)	asynchronost	bezpečnost + živost
Globální snapshot (Chandy-Lamport)	asynchronost	bezpečnost + živost
Vyloučení procesů (kruhový, Ricart-Agrawala)	asynchronost	bezpečnost + živost
Volba lídra (Raft, kruhový, Bully)	asynchronost + selhání	bezpečnost
Konsensus (Raft)	asynchronost + selhání	bezpečnost

Materiály

Úvod a modely	[Steen] 1.1,1.3; [Coulouris] 2.4.1, 2.4.2
Detekce selhání	[SWIM]
Kauzalita a čas	[Steen] 6.1-6.2; [Coulouris] 14.1-14.4
Globální snapshot	[Coulouris] 14.5
Vyloučení procesů	[Steen] 6.3; [Coulouris] 15.2
Volba lídra	[Steen] 6.4; [Coulouris] 15.3
Konsensus	[Steen] 8.2; [Coulouris] 15.5 [Raft]

[Steen] Van Steen, M. And Tanenbaum, A.S., 2017. *Distributed systems: principles and paradigms (3.01 Edition)*.[\[link\]](#)

[Coulouris] Coulouris, G.F., Dollimore, J. and Kindberg, T., 2005. *Distributed systems: concepts and design*.

[SWIM] Das, A., Gupta, I. and Motivala, A., 2002. Swim: Scalable weakly-consistent infection-style process group membership protocol. In Dependable Systems and Networks, 2002. [\[link\]](#)

[Raft] Ongaro, D. and Ousterhout, J.K., 2014, June. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*. [\[link\]](#)

Ukončení předmětu

Teoretická zkouška (max 40 bodů) – přihlašování přes KOS:

- Temíny TBD, ale měly by být (z velké části) v červnu.

Programovací zkouška (max 20 bodů) – přihlašování skrze formulář:

- Termíny TBC, aktuálně navrženy **pátky** dopoledne, začíná se 29.5.

Další termín bude v září (programovací i teoretická).

Zápočet, programovací a teoretická zkouška jsou **nezávislé**, lze udělat v libovolném pořadí.