Principy distribuovaných systémů

NSWI035
ZS 2012/13

RNDr. Filip Zavoral, Ph.D.
Katedra softwarového inženýrství

zavoral@ksi.mff.cuni.cz
http://ulita.mff.cuni.cz -> Výuka

29.1.2013 11:15

Obsah přednášky

1. Úvod
- motivace, funkce, architektury

2. Meziprocesová komunikace
- zprávy, spolehlivost, RPC, skupinová komunikace

3. Synchronizační algoritmy
- fyzické a logické hodiny, vyloučení procesů, volba koordinátora
- kauzální závislost, doručovací protokoly, virtuální synchronie
- distribuovaný konsensus, detekce globálního stavu

4. Distribuovaná sdílená paměť (DSM)
- konzistenční modely, distribuované stránkování

5. Správa prostředků
- zablokování, distribuované algoritmy detekce
- správa prostoru jmen, kapability

6. Procesy
- vzdálené spouštění procesů, migrace, vyvažování zátěže

7. Replikace
- replikace, aktualizační protokoly
- klientocentrické konzistenční modely, epidemické protokoly

Literatura

- Pokrývá přednášenou (a zkoušenou) látku
  - Tanenbaum, van Steen: Distributed Systems - Principles and Paradigms, *2nd ed*
  - Chow, Johnson: Distributed Operating Systems & Algorithms

- Další zdoje informací o DS
  - Kshemkalyani, Singhal: Distributed Computing - Principles, Algorithms and Systems
  - Coulouris, Dollimore: Distributed Systems - Concepts and Design, *4th ed*
  - Singhal, Shivaratri: Advanced Concepts in Operating Systems
  - Sinha: Distributed Operating Systems - Concepts and Design

- Formální metody a distribuované algoritmy
  - Santoro: Design and Analysis of Distributed Algorithms
  - Mullender: Distributred Systems, *2nd ed*

- slajdy nejsou skripta
- tzv. 'učební texty' D.O.S. nepoužívejte
  - na různých fórech a skladištích vědomostí
  - **velmi** zastaralé, nepřesné, neúplné, chybové

"Definice"

Distribuovaný systém
je systém propojení množiny nezávislých uzlů,
který poskytuje uživateli dojem jednotného systému

Hardwarová nezávislost

- uzly jsou tvořeny nezávislými "počítači" s vlastním procesorem a pamětí
- jedinou možností komunikace přes komunikační (síťové) rozhraní

Dojem jednotného systému

- uživatel (člověk i software) komunikuje se systémem jako s celkem
- nemusí se starat o počet uzlů, topologii, komunikaci apod.

distribuovaný systém - celá škála různých řešení

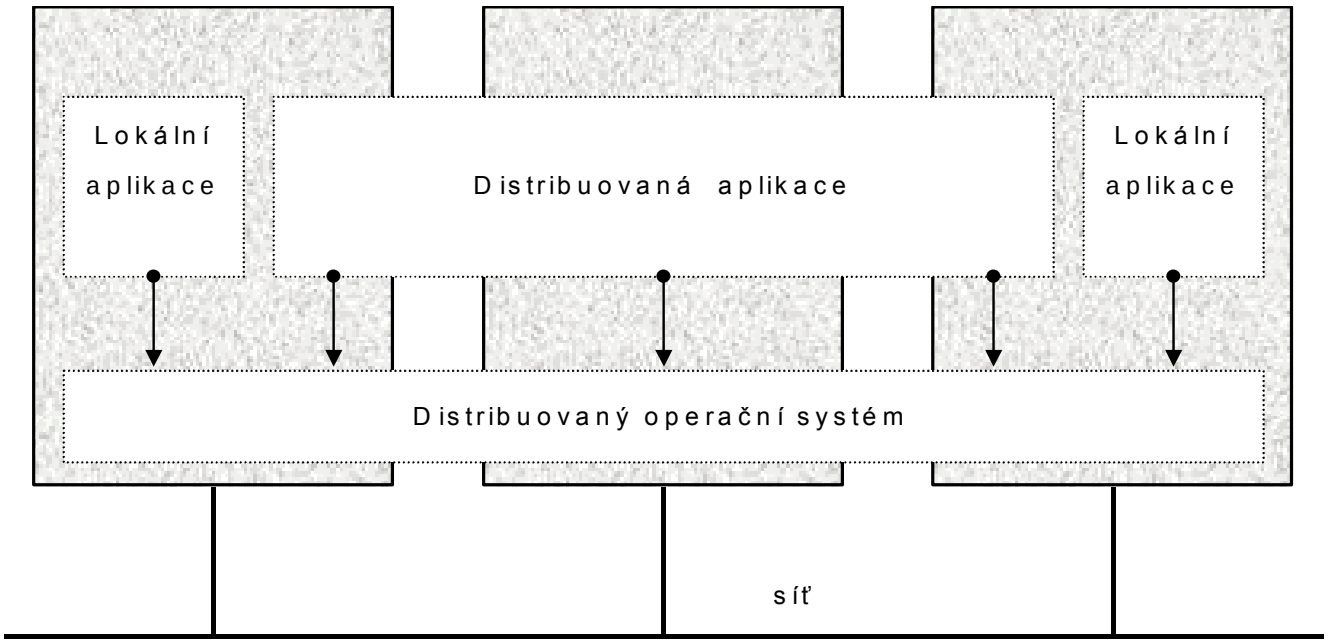
- multiprocesory na jedné základové desce
- celosvětový systém pro miliony uživatelů
- moderní technologie: cluster, cloud, grid, ...

Zaměření přednášky: uzel = ('běžný') počítač, běžná síť, softwarový systém

- principy, algoritmy

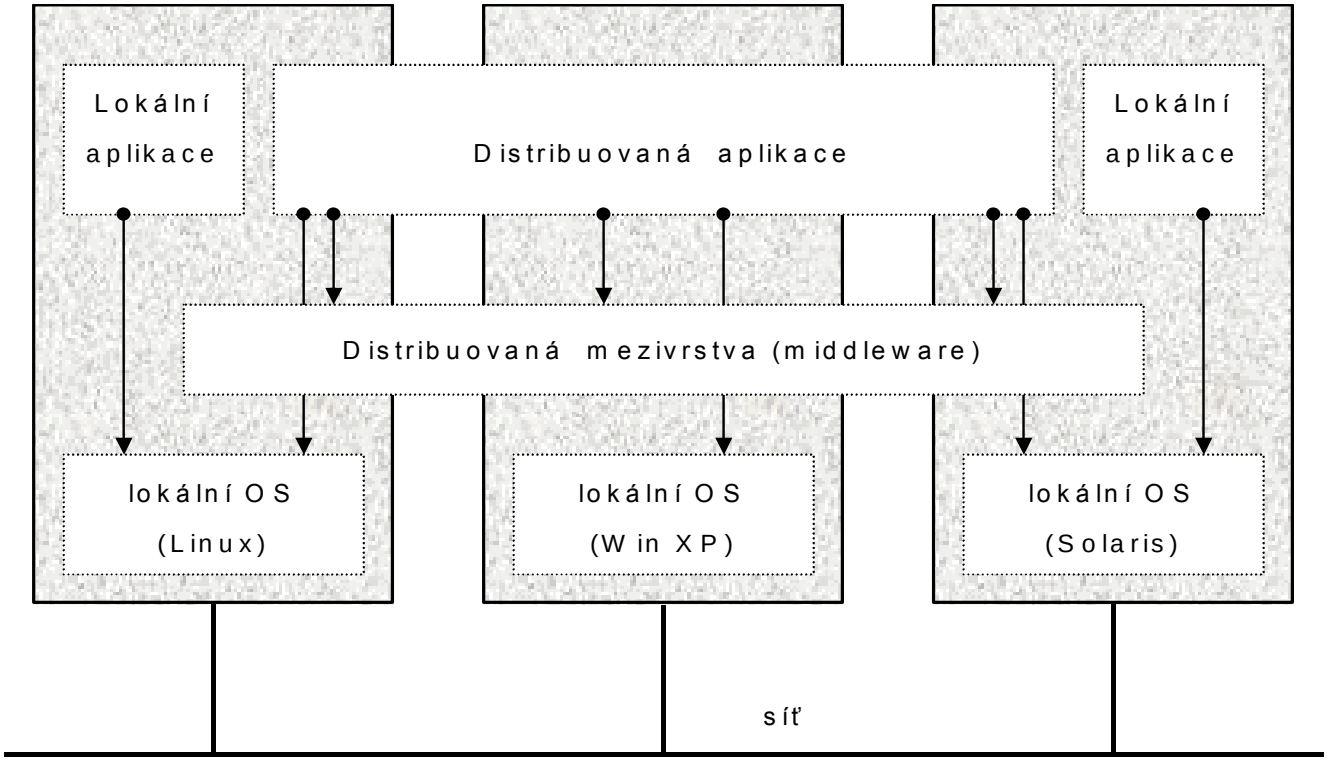
199x - Distribuované operační systémy

- operační systém vyvinutý speciálně pro potřeby DS
- "zlatý věk": 90. léta - Amoeba, Sprite, V, Chorus, Mosix ... T4
- součástí jádra OS podpora distribuovanosti, komunikace, synchronizace
- aplikace: transparentní využití distribuovaných služeb
- neexistuje rozdíl mezi lokální a distribuovanou aplikací
- nesplněná očekávání, žádný systém nedotažen do masově použitelného stavu



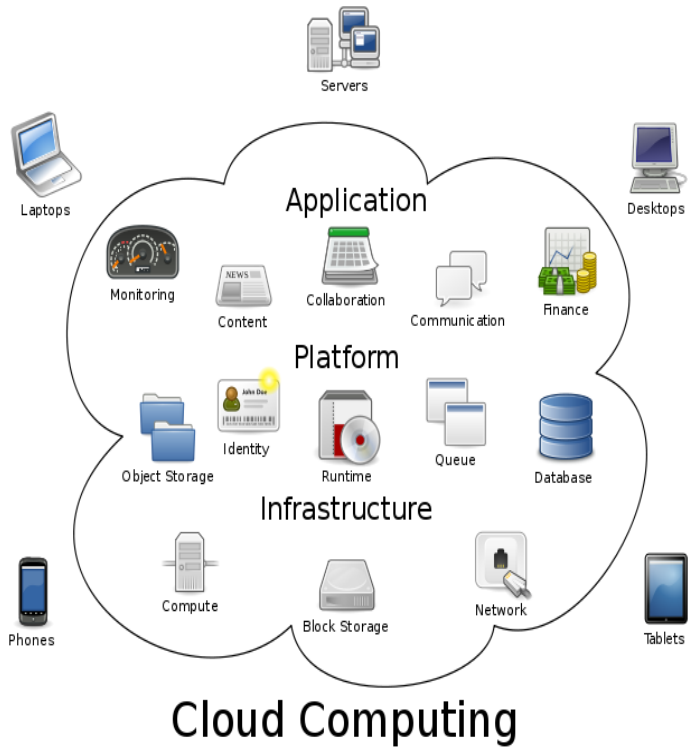
200x - Distribuované frameworky a aplikace

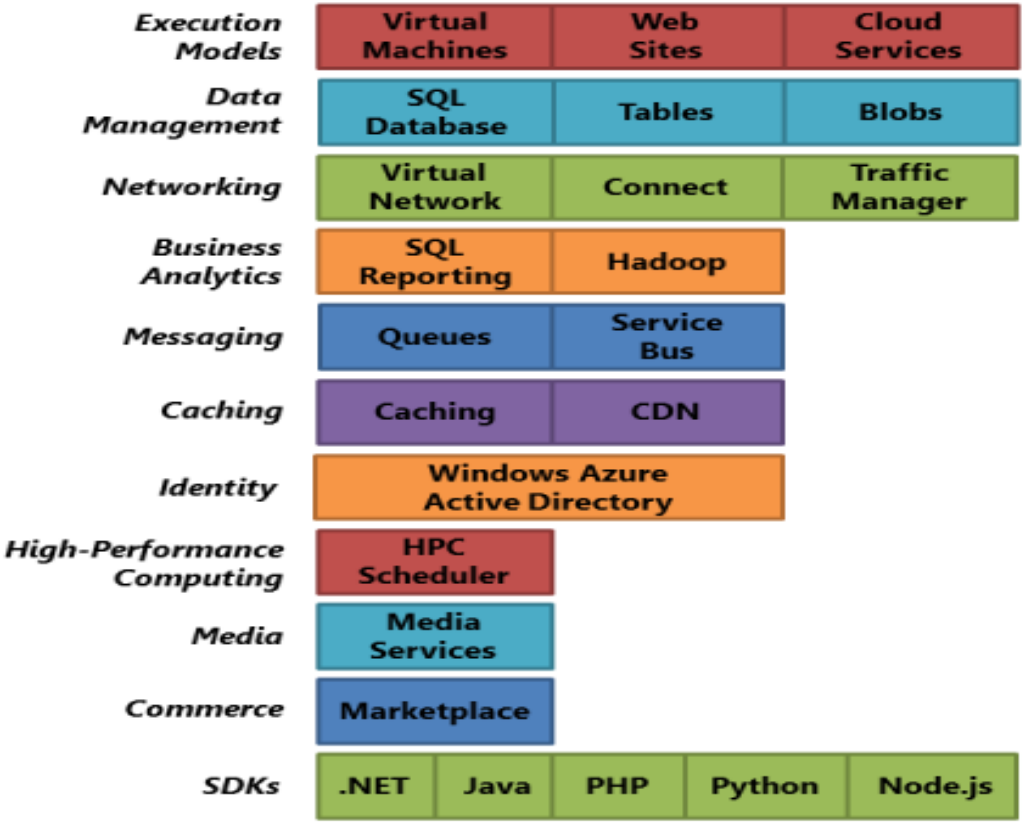
- softwarový průmysl - jiný směr
- dominance MS Windows, UNIX / Linux
- použití některých principů a mechanismů distribuovaných systémů
- libovolný lokální OS, rozšiřující vrstva - distribuované prostředí + další služby
- prostředí pro distribuované aplikace – middleware (OSF DCE, CORBA, DCOM, Globe, ..)



201x - Cloud computing

- service-oriented computing
  - SaaS, PaaS, IaaS
- nutná podpora virtualizace
- kompletní infrastruktura připravená k použití
- Windows Azure, Amazon Elastic Compute Cloud, Google Cloud Platform, ...





Distributed Computing

Obecné principy distribuovaných systémů aplikovatelné na různá prostředí

- mobilní systémy
  - bezdrátové sítě, routing, lokalizace, replikace
- ubiquitní a pervasivní systémy
  - mikrouzly, každodenní sci-fi, samoorganizace, kolektivní znalost a rozhodování
- peer-to-peer sítě
  - sdílení MM obsahu, P2P výpočty, distribuované ukládání, replikace, důvěryhodnost
- content management *(správa obsahu)*
  - publish-subscribe, media distribution, streaming
- distribuovaní agenti
  - mobilita, migrace, task-oriented entities, kooperace
- cluster / grid *(LHC-CERN)* / cloud / sky computing
- distribuované databáze, NoSQL databáze, Big Data
- distributed data mining *(dolování dat)*
- ...

Motivace a cíle návrhu distribuovaného systému

Motivace

- **Ekonomika**
  - síť běžných PC × supercomputer
- **Výkon**
  - technologické limity
- **Distribuovanost**
  - 'inherentní distribuovanost'
- **Spolehlivost**
  - výpadek jednoho uzlu × výpadek CPU
- **Rozšiřitelnost**
  - přidání uzlu × upgrade centrálního serveru

Cíle návrhu

- Transparentnost – přístupová, lokační, migrační, replikační, ...
- Přizpůsobivost – autonomie, decentralizované rozhodování, ...
- Spolehlivost
- Výkonnost
- Rozšiřitelnost – rozdílné metody pro 100 a 100.000.000 uzlů
- . . .

Transparentnost

na úrovni komunikace s uživatelem, komunikace procesů s jádrem
stupeň transparentnosti × výkonnost systému

- **Přístupová**
  - proces má stejný přístup k lokálním i vzdáleným prostředkům
  - jednotné služby, reprezentace dat, ...
- **Lokační**
  - uživatel (proces) nemůže říci, kde jsou prostředky umístěny
  - *nevyhovuje: počítač:cesta/soubor*
- **Exekuční / Migrační**
  - procesy mohou běžet na libovolném uzlu / přemisťovat se mezi uzly
- **Replikační**
  - uživatel se nemusí starat počet a aktualizaci kopií objektů
- **Perzistentní**
  - uživatel se nemusí starat o stav objektů, ke kterým přistupuje
- **Konkurenční**
  - prostředky mohou být automaticky využívány více uživateli
- **Paralelismová** 😐
  - různé akce mohou být prováděny paralelně bez vědomí uživatele

Přizpůsobivost, spolehlivost

Přizpůsobivost

- Autonomie
  - každý uzel je schopný samostatné funkčnosti
- Decentralizované rozhodování
  - každý uzel vykonává rozhodnutí nezávisle na ostatních
- Otevřenost
  - lze zapojit různé komponenty vyhovující rozhraní
  - oddělení interface × implementace
- Migrace procesů a prostředků
  - procesy i prostředky mohou být přemístěny na jiný uzel

Spolehlivost

- Teorie: nespolehlivost jednoho serveru 1%

  $\Rightarrow$ nespolehlivost čtyř serverů $0.01^4 = 0.00000001$

- Leslie Lamport: distribuovaný systém je ...

  "systém, který pro mě odmítá cokoliv vykonat,
  protože počítač, o kterém jsem nikdy neslyšel, byl náhodně vypnut"

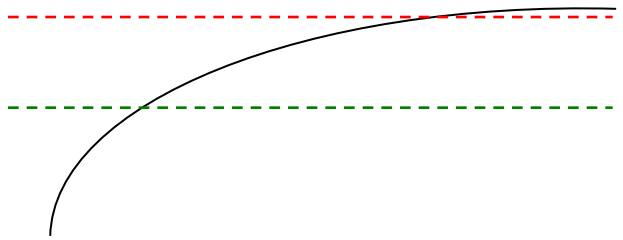
Rozšiřitelnost

Rozšiřitelnost - scalability

- Rozdílné metody pro 100 uzlů × 100 milionů uzlů
- Princip: vyhnout se čemukoliv centralizovanému
  - žádný uzel nemá úplnou informaci o celkovém stavu systému
  - uzly se rozhodují pouze na základě lokálně dostupných informací
  - výpadek jednoho uzlu nesmí způsobit nefunkčnost algoritmu
  - nelze spoléhat na existenci přesných globálních hodin
    - *'každou celou hodinu si všechny uzly zaznamenají poslední zprávu'*
- Další problémy
  - koordinace a synchronizace
  - geografická odlehlost, administrace
- Techniky škálovatelnosti
  - asynchronní komunikace
  - distribuce
  - caching a replikace

serverům, službám, tabulkám, algoritmům, ...

Výkonnost, chyby návrhu



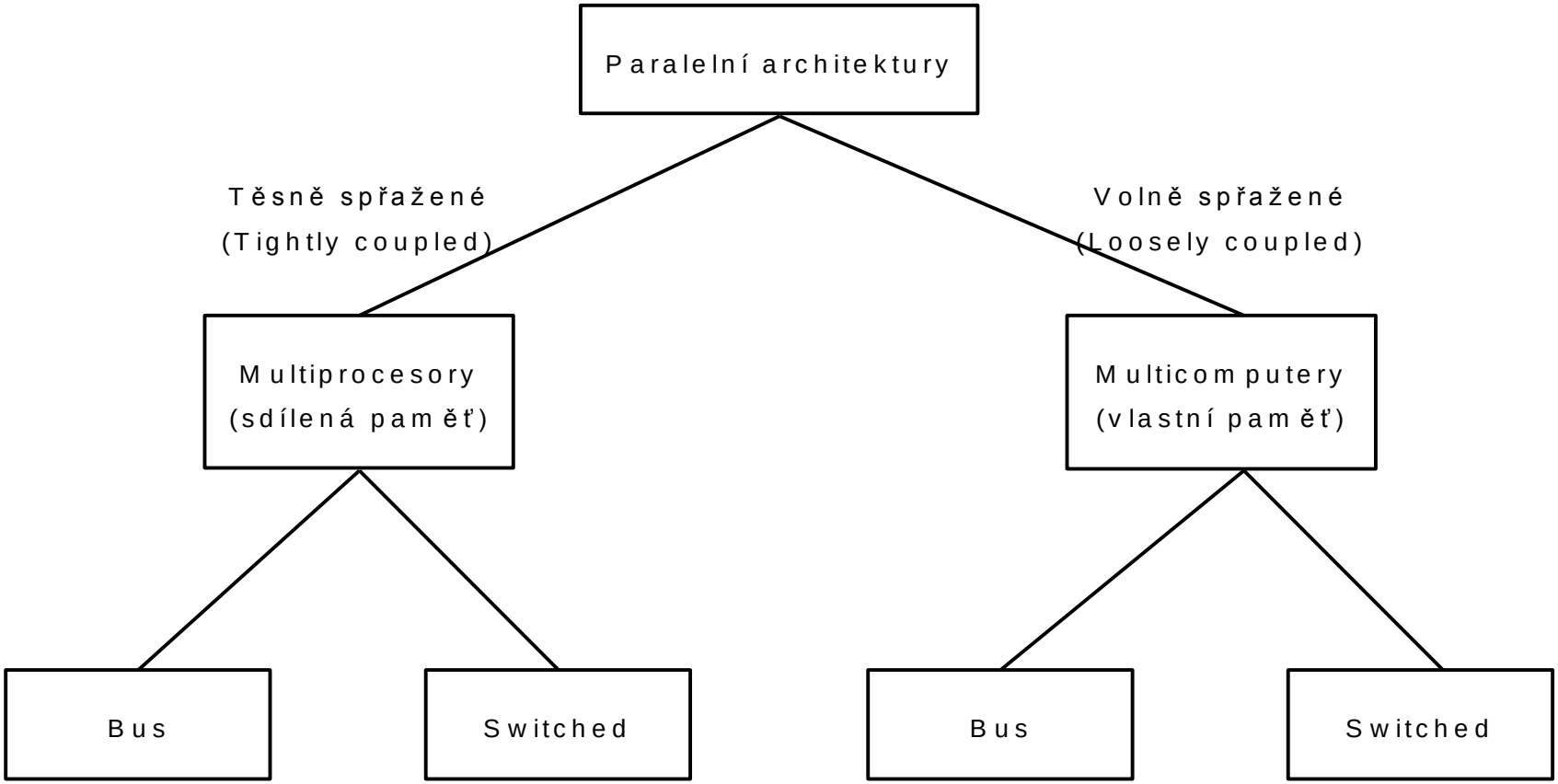
Výkonnost

- Teorie: více uzlů vyšší výkon
- Praxe: výrazně nižší než lineární nárůst výkonu
- Příčiny: komunikace, synchronizace, komplikovaný software
- Granularita výpočetní jednotky

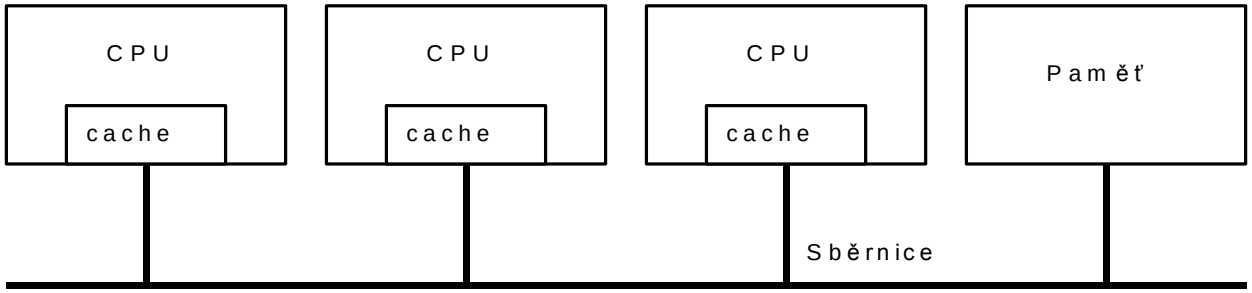
Chyby návrhu distribuovanách systémů

- Síť je spolehlivá
- Síť je zabezpečená
- Síť je homogenní
- Topologie se nemění
- Nulová latence
- Neomezená kapacita sítě
- Jeden administrátor

Paralelní architektury



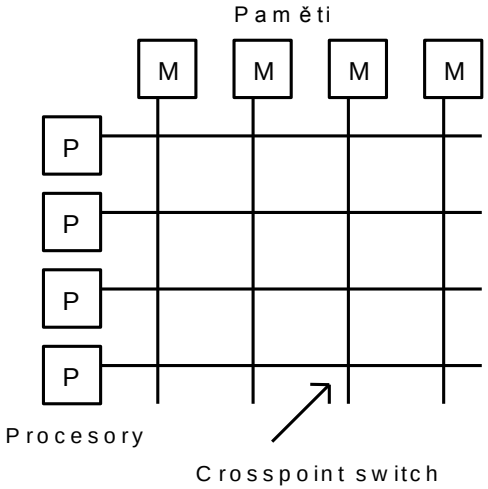
Multiprocesory – sběrnicová architektura

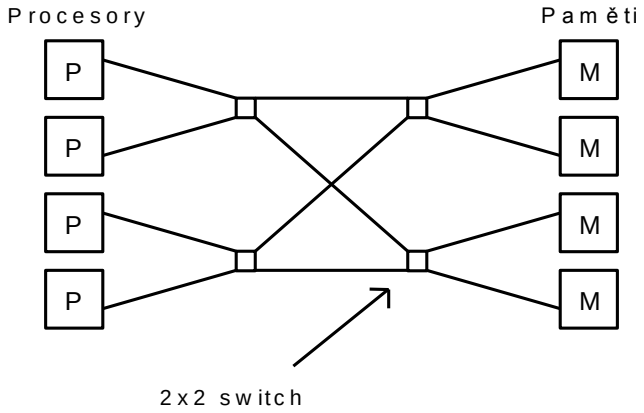


Sběrnicová architektura

- sdílená sběrnice mezi procesory a pamětí
- levná, běžně dostupná
- max. 32 (64) uzlů
- cache – synchronizace

Multiprocesory – přepínačová architektura

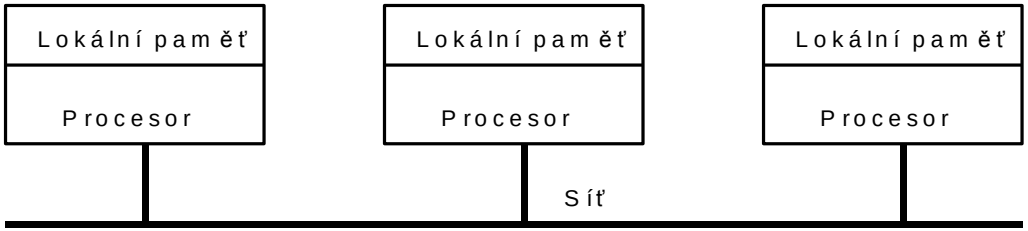




Přepínačová (switched) architektura

- Možné zapojení i většího počtu uzlů
- Mřížka - crosspoint switch
  - sběrnicová mřížka - procesory × paměťové bloky
  - nákladné – kvadratický počet přepínačů
- Omega network
  - postupně přepínaná cesta mezi procesorem a pamětí
  - při větším počtu úrovní pomalé
  - počet přepínačů n * log n

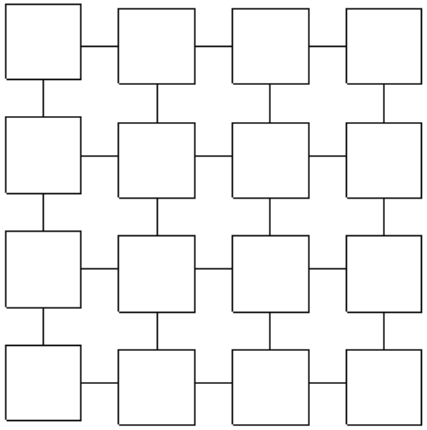
Multicomputery – sběrnicová architektura



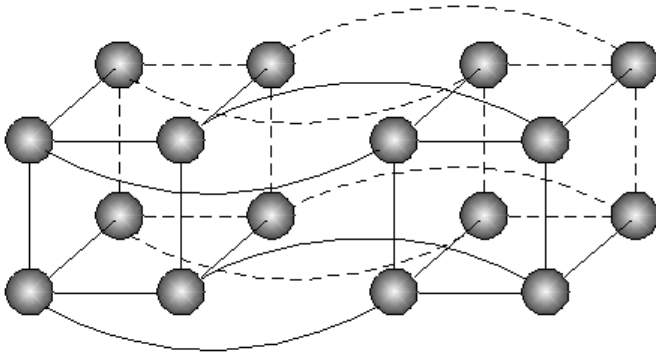
Sběrnicová architektura

- vlastní paměť každého uzlu → výrazně nižší komunikace
- možnost teoreticky neomezeného počtu uzlů
- levná, běžně dostupná
- '*normální*' počítače propojené '*normální*' sítí
- lze i jiné provedení: processor pool

Multicomputery – přepínačová architektura



(a)



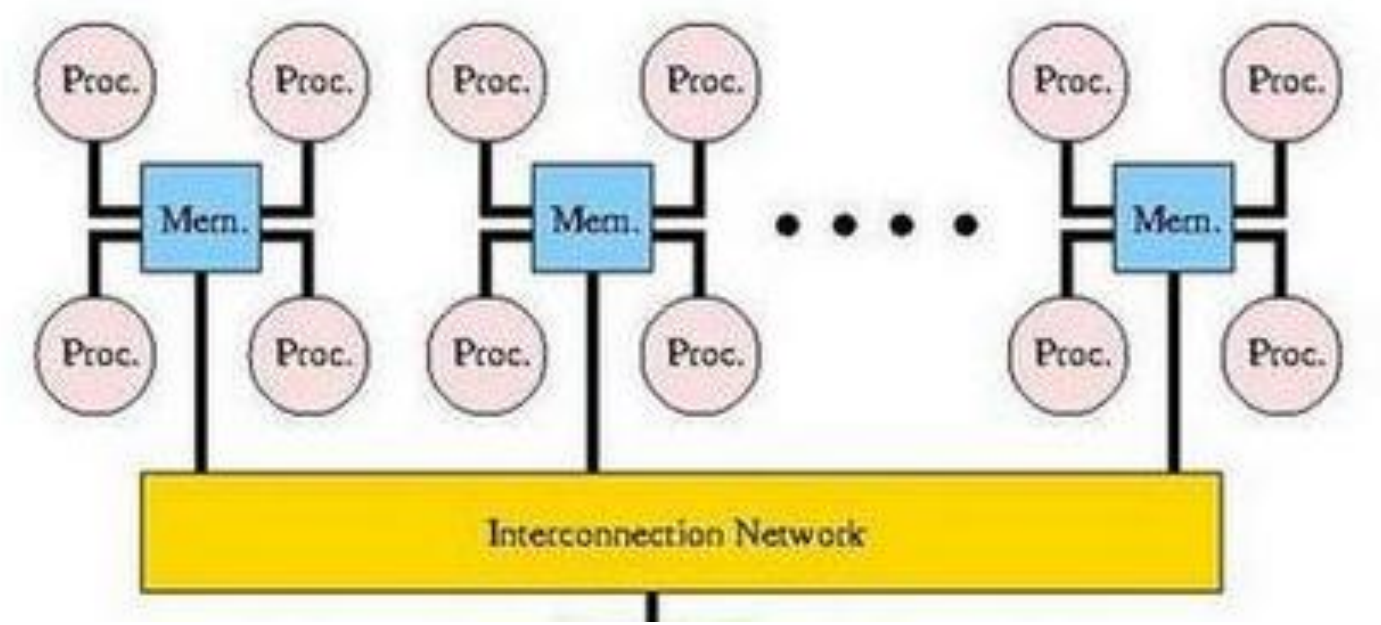
(b)

Přepínačová (switched) architektura

- Mřížka
  - relativně jednoduchá (dvourozměrná) implementace
  - vhodné pro řešení 'dvourozměrných' problémů – grafy, analýza obrazu, ...
- Hyperkrychle
  - n-rozměrná krychle, častý rozměr 4
  - každý uzel přímo propojen se sousedy ve všech rozměrech
  - supercomputers - tisíce až miliony uzlů
    - 2012 IBM Sequoia Blue Gene - 1.5 mil cores, 1.5 PB RAM, 8 MW ..... Linux ☺

NUMA

Non Uniform Memory Access

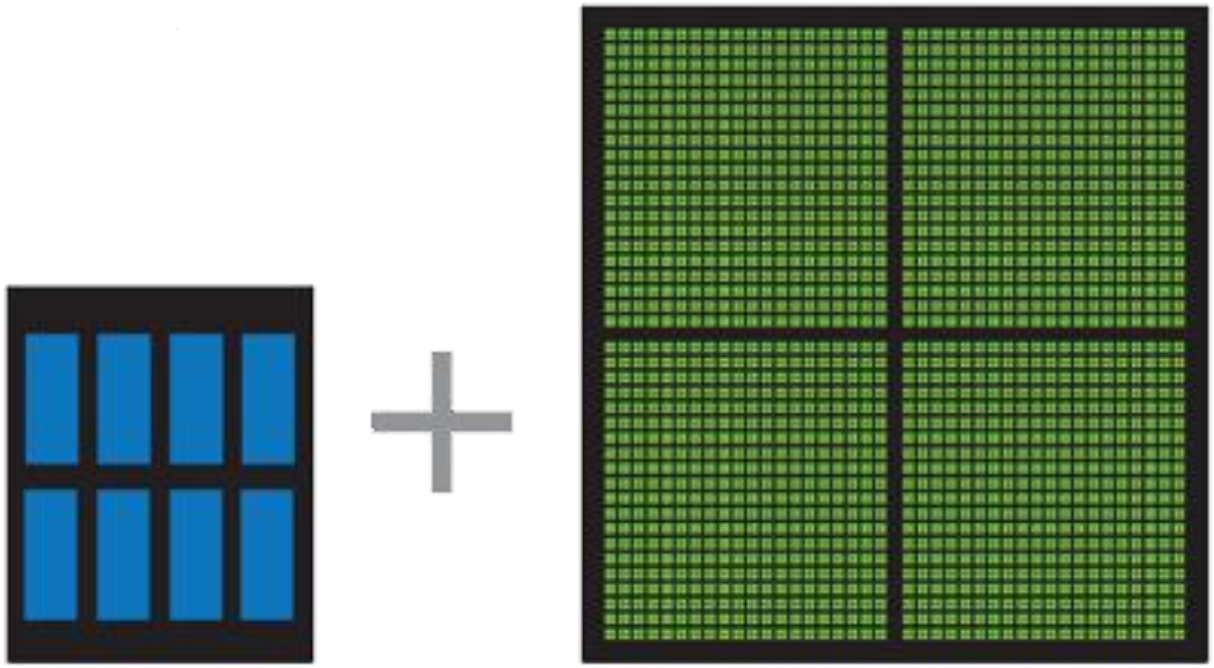


- S-COMA - Simple Cache-Only Memory Architecture
  - požadavek na nelokální data je propagován vyšší vrstvě
- ccNUMA - Cache Coherent NUMA
  - globální adresace, konzistence cache
- NUMA-faktor - rozdíl latence lokálních a vzdálených paměťových přístupů

GPGPU

General-purpose computing on graphics processing units

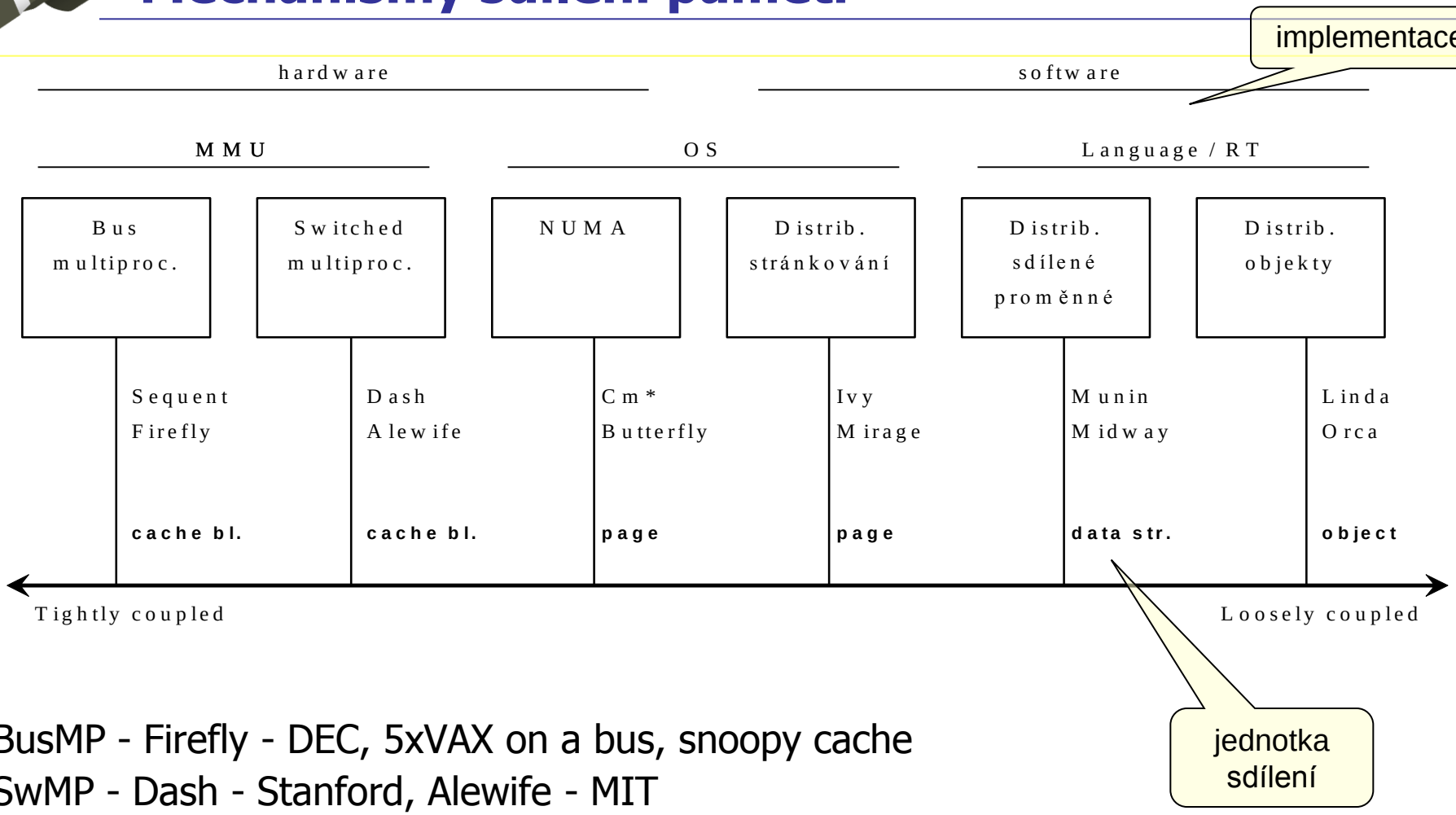
- Frameworks
  - OpenCL, CUDA, C++Amp
- Tesla GPU, Kepler, ...



CPU
MULTIPLE CORES

GPU
THOUSANDS OF CORES

Mechanismy sdílení paměti



BusMP - Firefly - DEC, 5xVAX on a bus, snoopy cache
SwMP - Dash - Stanford, Alewife - MIT
Cm* - hw přístup do maměti, sw caching
Distributed paging - Ivy, Li; Mirage - Bershad 1990
Munin Benett 1990

Komunikace

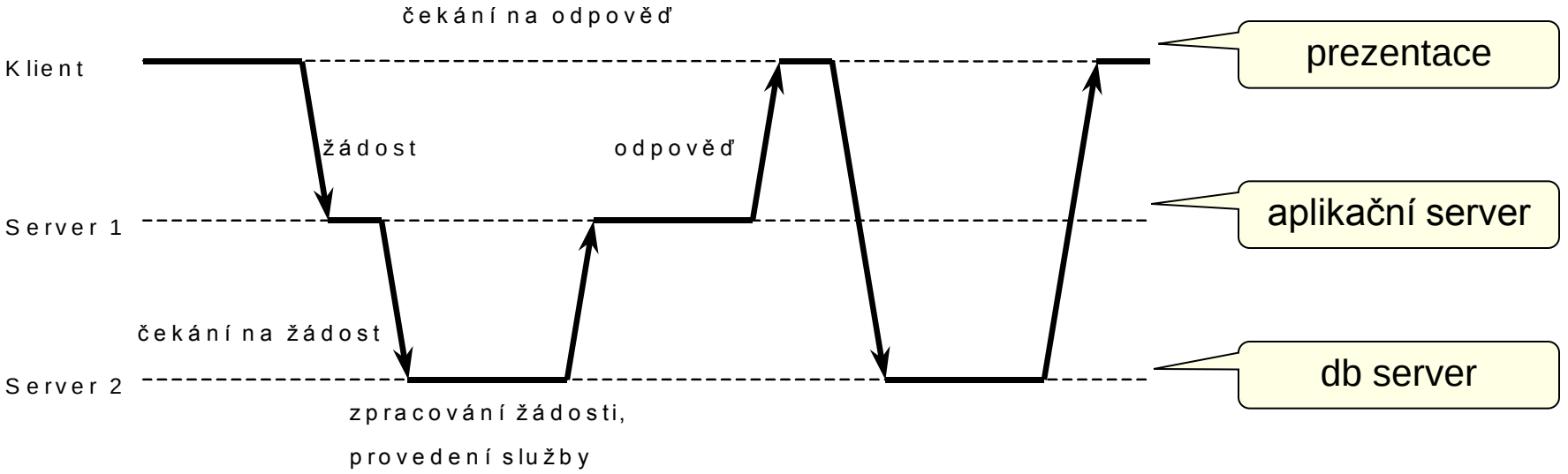
absence sdílené paměti → zasílání zpráv

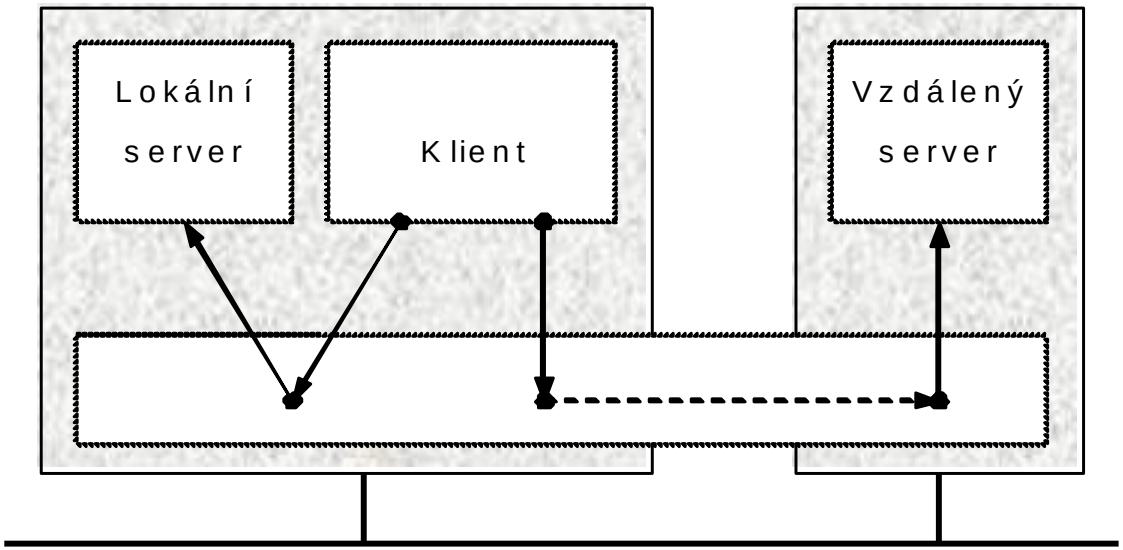
RPC, doors, RMI, M-queues, ...

Jednotný komunikační mechanismus

- jednotný obecný mechanismus pro rozdílné podmínky
  - lokální komunikace, rychlá síť, vzdálené sítě s pomalým přístupem
- nízká režie
  - lokální služby
- síla služeb
  - mapování, sdílení, ochrana, ...
- flexibilita
  - nespolehlivost dálkové komunikace, zpoždění, zabezpečení
- distribuované aplikace
  - typicky ignorují problém síťové režie

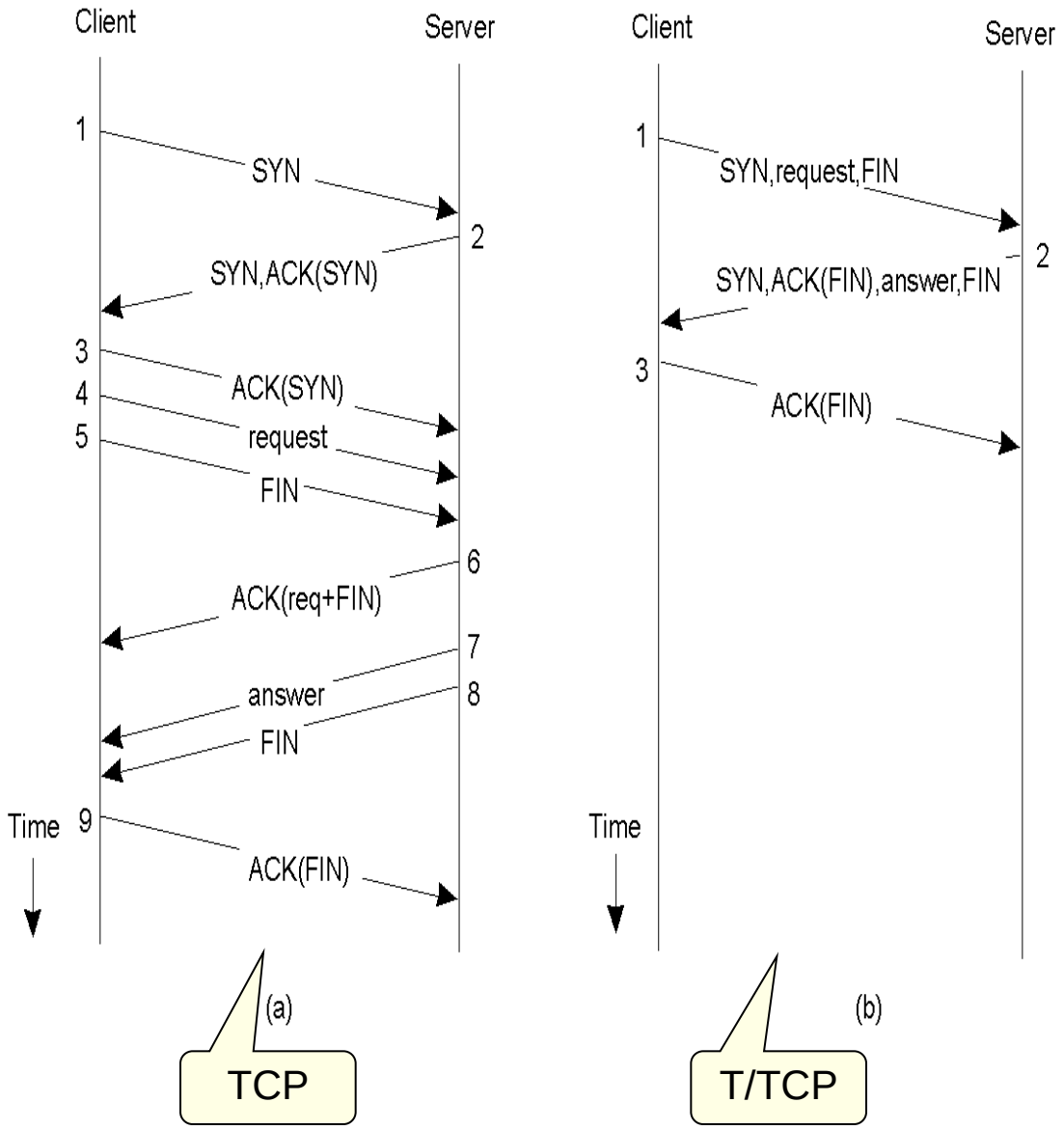
Klient – server model





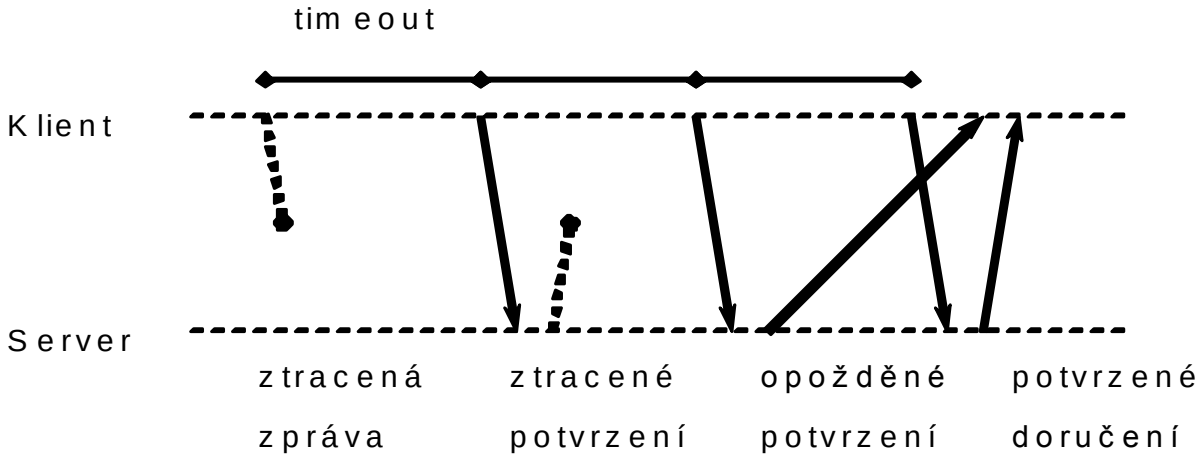
Síťová komunikace

- TCP - standard pro spolehlivý přenos vzdálených dat
  - ♦ ale: většina komunikace v rámci rychlé a relativně spolehlivé LAN
  - ♦ nevýhoda TCP - složitost protokolu, menší výtěžnost sítě
- specializované protokoly pro (spolehlivé) lokální sítě
  - ♦ optimistické
  - ♦ vysoký výkon za cenu složitějšího zotavení z chyb
  - ♦T/TCP, FLIP, ...



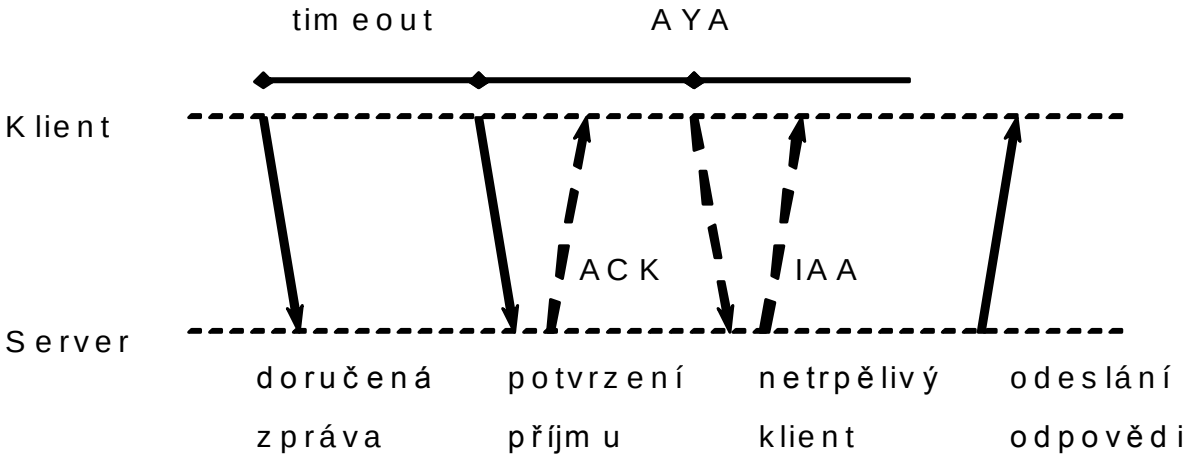
Spolehlivost komunikace

- duplicita zpráv - číslování, zahození - poslední zpráva / okno
- předbíhání zpráv - číslování, zpráva o nedoručení - režie, okno
- ztráta zprávy - komplikovanější
  - klient vyslal žádost a neví .....
  - ztráta žádosti / ztráta odpovědi / příliš pomalá komunikace
  - typické řešení: potvrzování, timeout, opakování



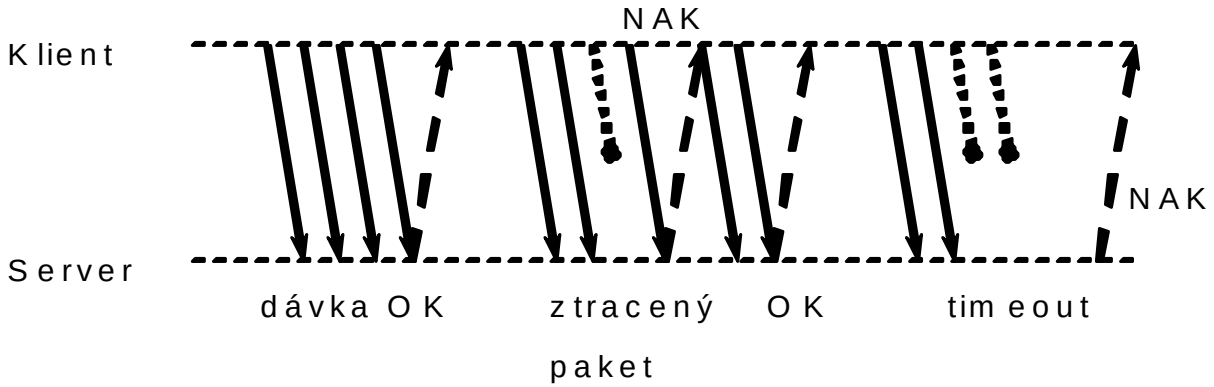
Spolehlivost komunikace

- potvrzování → zvýšení režie → odpověď = potvrzení příjmu
- při delším zpracování - timeout serveru nebo klienta
- další služební zprávy - ACK, NAK, AYA, IAA, CON, ...



Přenos dlouhých zpráv

- jednotka přenosu = zpráva
  - příliš dlouhá → rozdělení na pakety
- co potvrzovat?
  - každý paket? — režie při správném přenosu !
  - celá zpráva? — režie při špatném přenosu !
- možné řešení - ***buřty*** (dávka, blast, burst)
  - potvrzování dávky (definovaný počet paketů)
  - v pořádku celá dávka → ACK
  - předbíhání, výpadek → NAK
    - přenos celé dávky / od místa výpadku
  - ztráta posledního (-ích) paketů → timeout



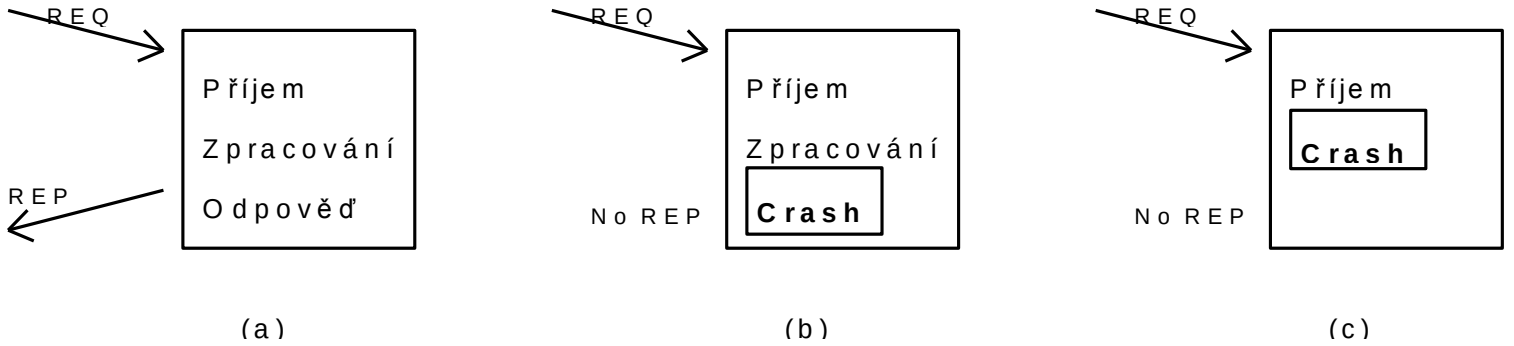
Pevné a dynamické dávky

- **pevné dávky** - jednodušší
  - při vyšší zátěži sítě velká chybovost
  - při volné síti nízká výtěžnost
- **dynamické dávky** - úprava velikosti podle aktuální situace
  - n-krát správný přenos dávky → zvětšení
  - špatný přenos → zmenšení
  - vysoká výtěžnost aniž by jeden přenos omezoval ostatní
  - konsensus o velikosti dávky!

Spolehlivost serveru

Nepřichází odpověď klientovi:

- ztratila se zpráva se žádostí
- ztratila se zpráva s odpovědí
- server je příliš pomalý nebo zahlcen
- server nefunguje



- Havárie serveru
  - obecně nelze zjistit, zda se operace provedla
  - i když lze, pro některé služby náročné (transakce)

Sémanitka zpracování

idempotentní služby

- nevadí opakované provedení služby
- 👍 sečti 1 + 1
- 👎 vyber 1000000 $ z mého účtu ☺

- **exactly once** sémantika
  - pro některé služby nelze (tisk na tiskárně)
- **at-least-once** sémantika
  - služba se určitě alespoň jednou provede
  - nelze zajistit, aby se neprovedla vícekrát
- **at-most-once** sémantika
  - služba se určitě neprovede vícekrát
  - nelze zajistit, že se provede

Havárie klienta

sirotci (orphans)

typicky se neřeší – specializované dlouhotrvající výpočty

Havárie klienta

sirotci (orphans)

typicky se neřeší – specializované dlouhotrvající výpočty

- exterminace
  - klient po zrození sám zruší službu
- reinkarnace
  - klient po zrození nastartuje novou epochu
  - server zruší všechny úlohy patřící do prošlých epoch
- expirace
  - úloha má přiděleno kvantum času
  - při překročení si server vyžádá nové kvantum

Implementace vzdáleného volání

```
main()
{
   struct message m1, m2;
   for(;;)
   {
      receive( &m1);
      switch( m1.opcode)
      {
        case SERVICE1: fnc1( &m1, &m2); break;
        case SERVICE2: fnc2( &m1, &m2); break;
        case SERVICE3: fnc3( &m1, &m2); break;
        default:  m2.retval = ERR_BAD_OPCODE;
      }
      send( m1.client, &m2);
   }
}
```

server

klient

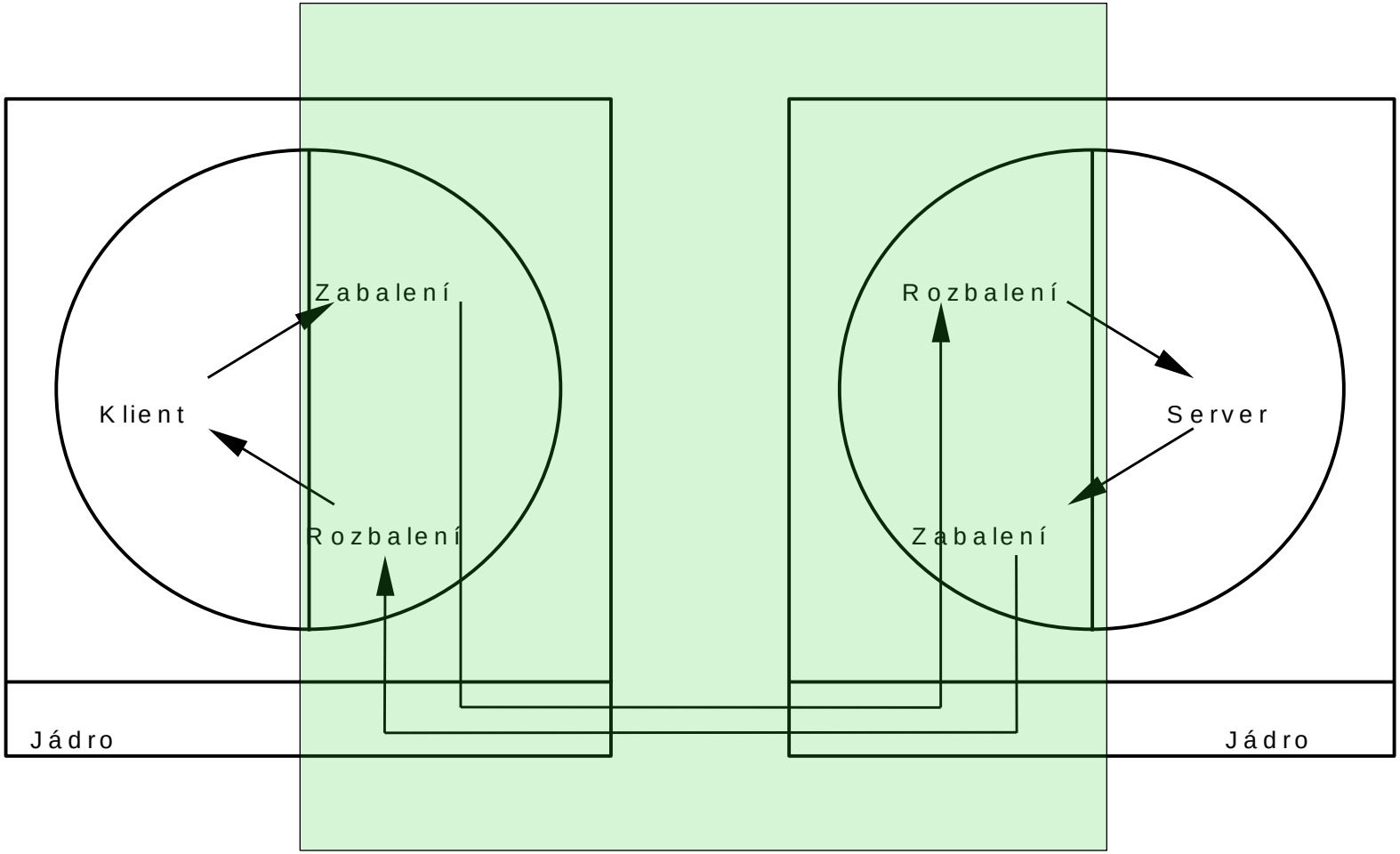
```
fnc()
{
     struct message m;
     m.opcode = SERVICE1;
     m.arg1 = ...;
     m.arg2 = ...;
     send( MY_SERVER, &m);
     receive( &m);
     if( m.retval != OK)
        error_handler( m.retval);
     process_data( &m.data);
}
```

RPC – vzdálené volání procedur

Meziprocesová komunikace většinou příliš nízkoúrovňová

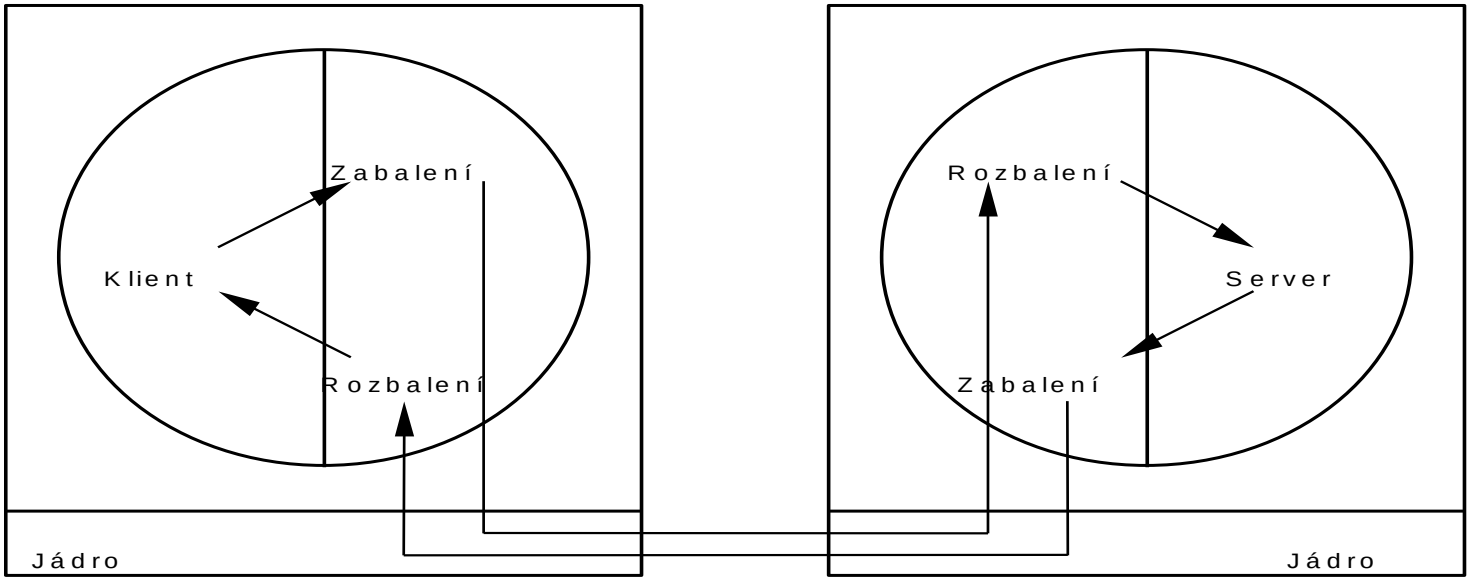
Idea: přiblížit zažitým mechanismům volání procedur

RPC - Remote Procedure Call



RPC – mechanismus volání

1. Klientova funkce normálním způsobem zavolá klientský stub
2. Stub vytvoří zprávu a zavolá jádro
3. Jádro pošle zprávu jádru počítače, kde běží server
4. Vzdálené jádro předá zprávu stubu serveru / skeletonu
5. Stub rozbalí parametry a zavolá server
6. Server zpracuje požadavky a normálním způsobem se vrátí do stubu
7. Stub zabalí výstupní parametry do zprávy a zavolá jádro
8. Jádro pošle zprávu klientovu jádru
9. Klientovo jádro předá zprávu stubu
10. Stub rozbalí výstupní parametry a vrátí se ke klientovi



RPC - spojení

Kompilace

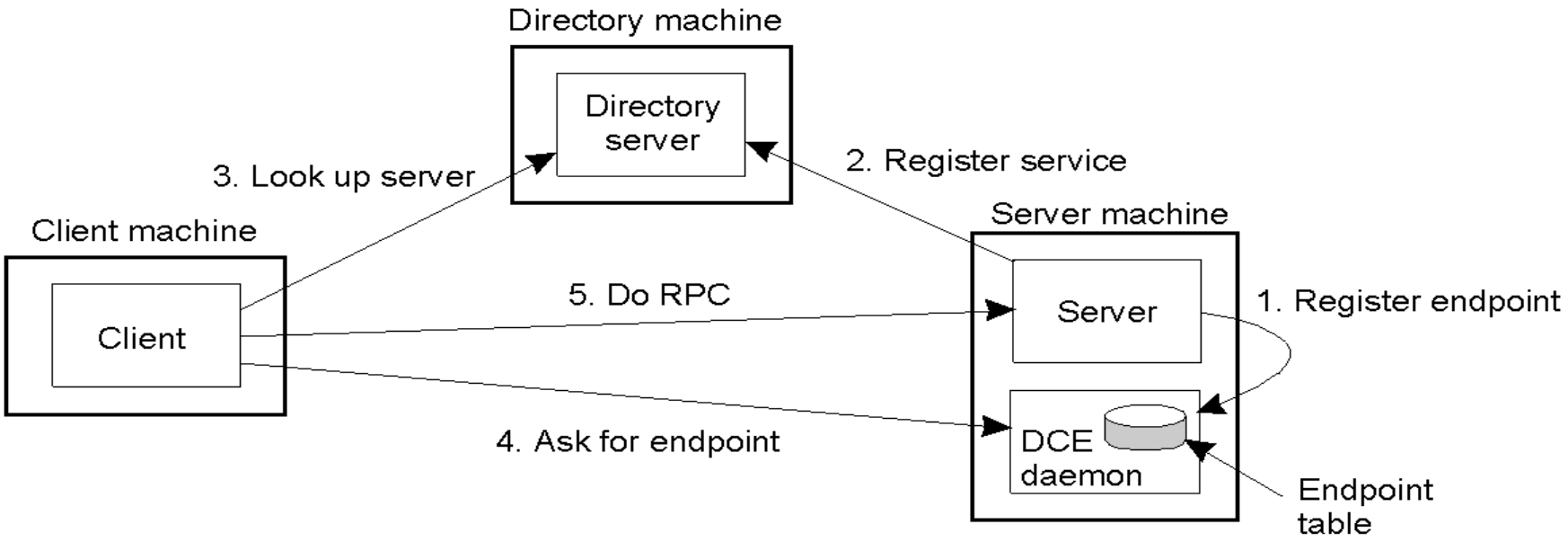
- Klient a server jsou programovány "lokálně"
- Vytvoří se stub (klient) a skeleton (server)

Spojení

- Server se "zviditelní" (exporting, registering) - *binder, trader, broker*
- Klient se "spojí" (binding) se serverem

Vyvolání

- Vytvoří (marshalling) a přenese se zpráva
- Potvrzování, exception handling



Příklad IDL (DCE)

```
[ uuid(a01d0280-2d27-11c9-9fd3-08002b0ecef1), version(1.0) ]

interface math{
        const long ARRAY_SIZE = 10;
        typedef long array_type[ARRAY_SIZE];
        long get_sum([in] long first, [in] long second);
        void get_sums([in] array_type a,
                [in] array_type b,
                [out] array_type c);
        }
```

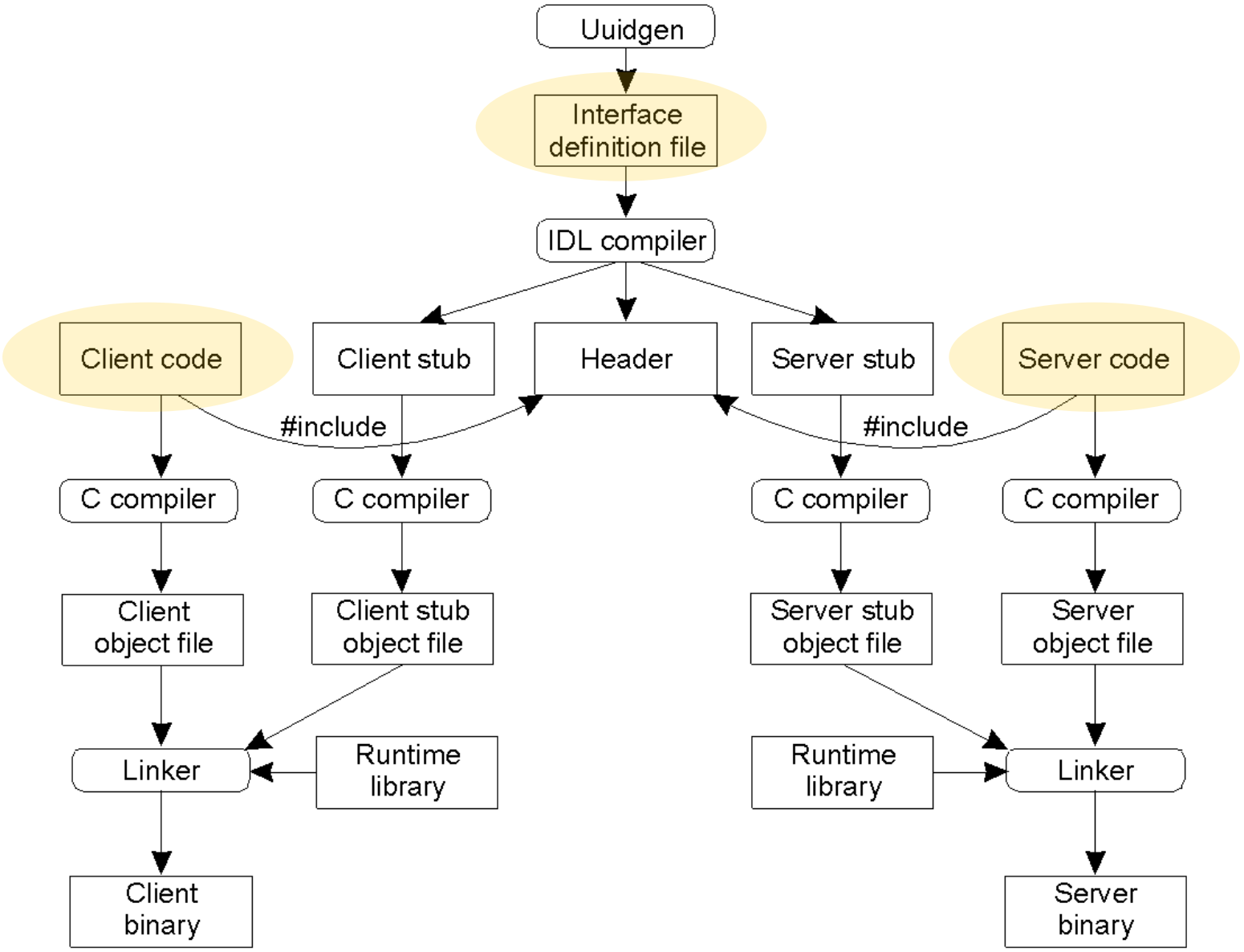
jednoznačný id rozhraní

definice rozhraní

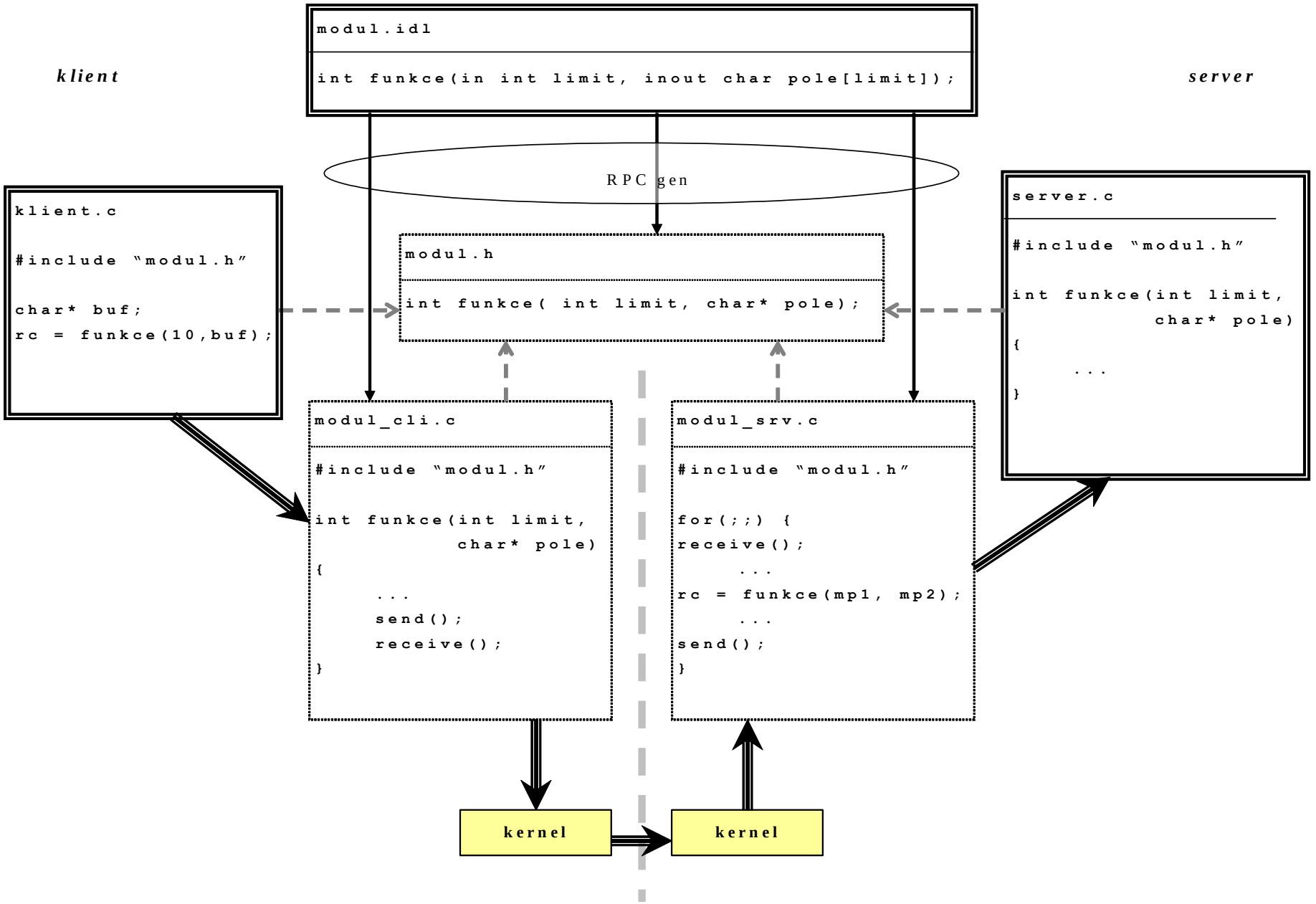
omezení velikosti polí

vstupní / výstupní parametry

RPC – kompilace modulů



RPC – kompilace modulů



RPC - problémy

Problémy

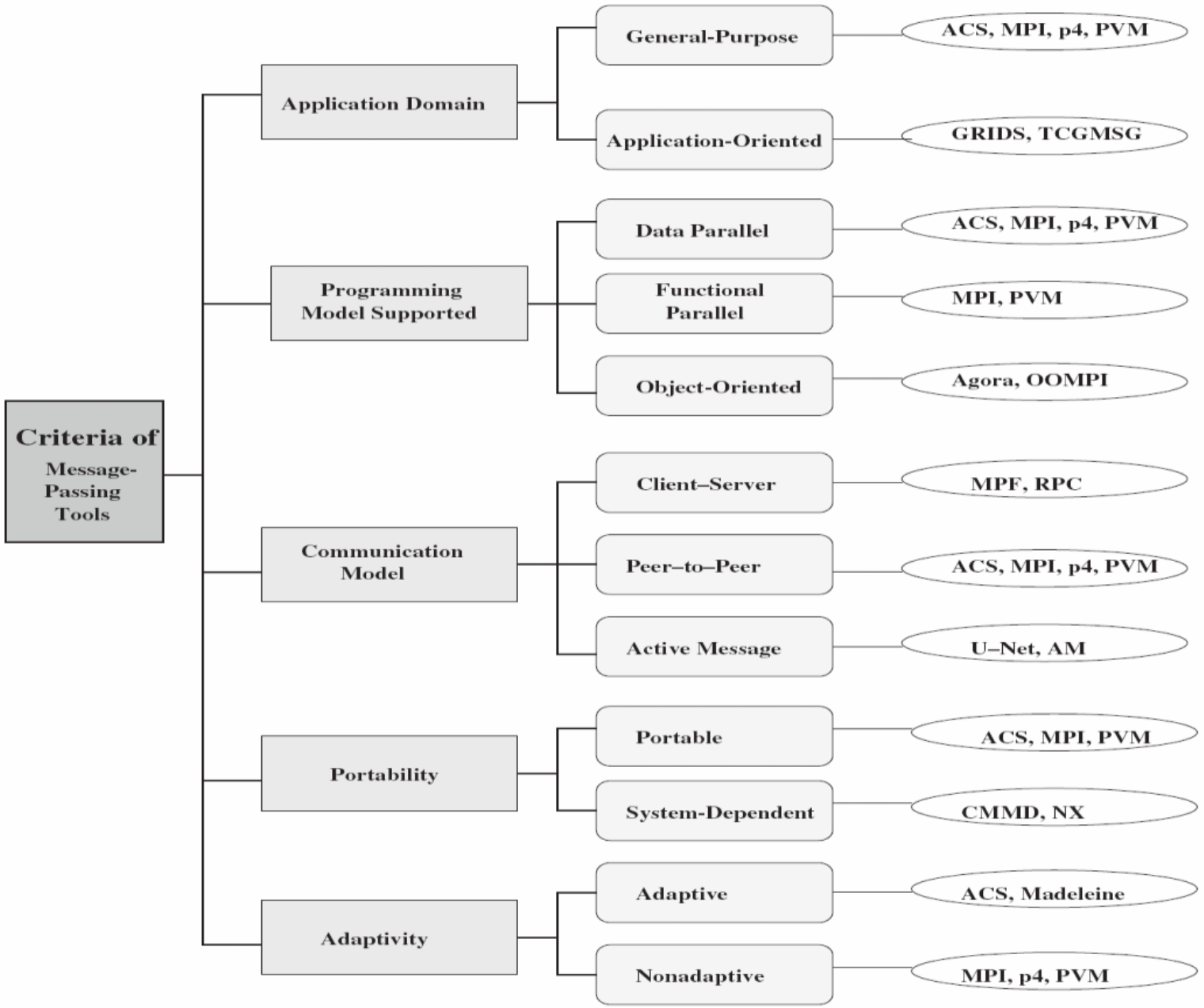
- reprezentace dat
  - endians, float, kódování řetězců
- přístup ke globálním proměnným
  - neexistence sdílené paměti
  - možné řešení DSM – příliš těžkopádné
- předávání ukazatelů, dynamické struktury
- předávání polí
  - copy/restore, aktuální velikost
- variabilní parametry - printf
- skupinová komunikace
- komunikační chyby - odlišná sémantika
- bezpečnost

RPC obecně

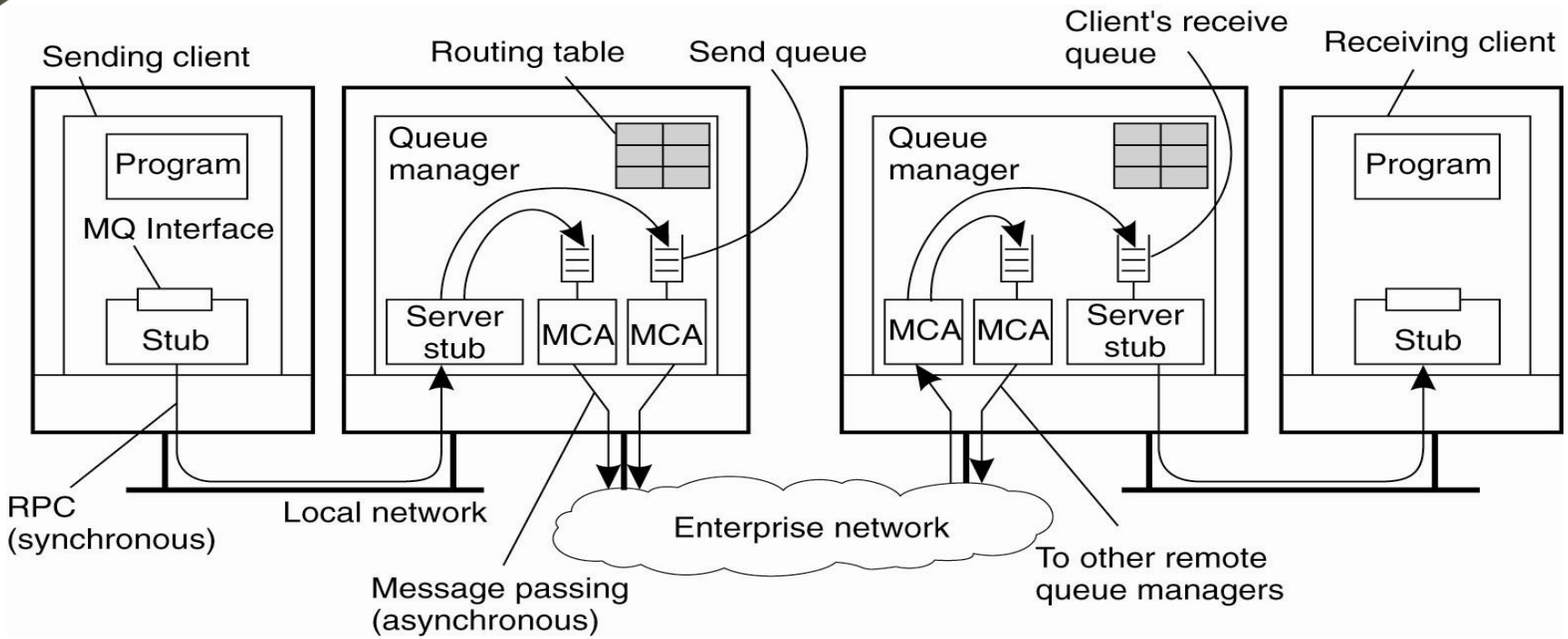
- užitečné, prakticky použitelné (a používané)
- není zcela transparentní, nutno dávat pozor na omezení

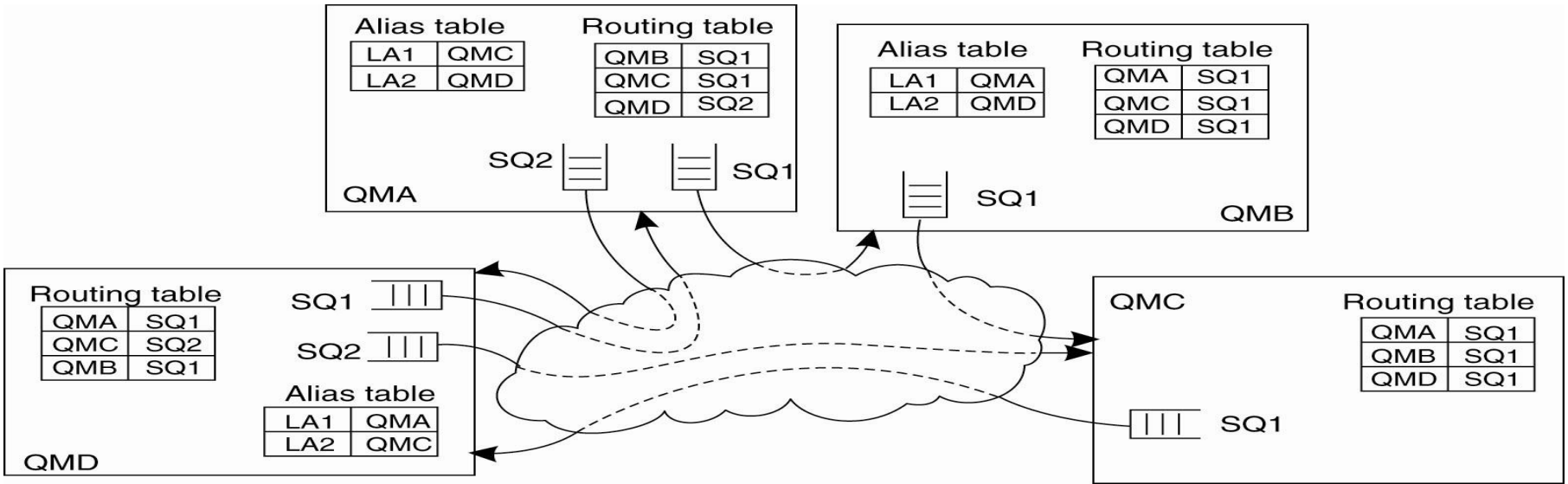
Další mechanismy:
SRPC
Asynchronous RPC
Doors, RMI
MPI, MQ, MOM, ...

Message passing interfaces



Message Queues





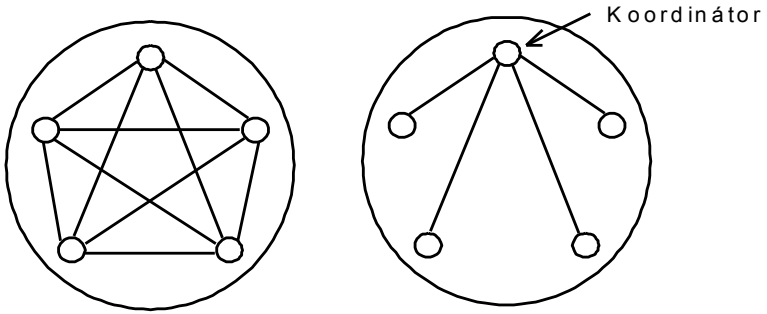
Skupinová komunikace

Jeden odesílatel – více příjemců

- Atomicita
  - doručení *všem* členům nebo nikomu
  - evidence a změna členství
- Synchronizace
  - doručování od různých odesílatelů

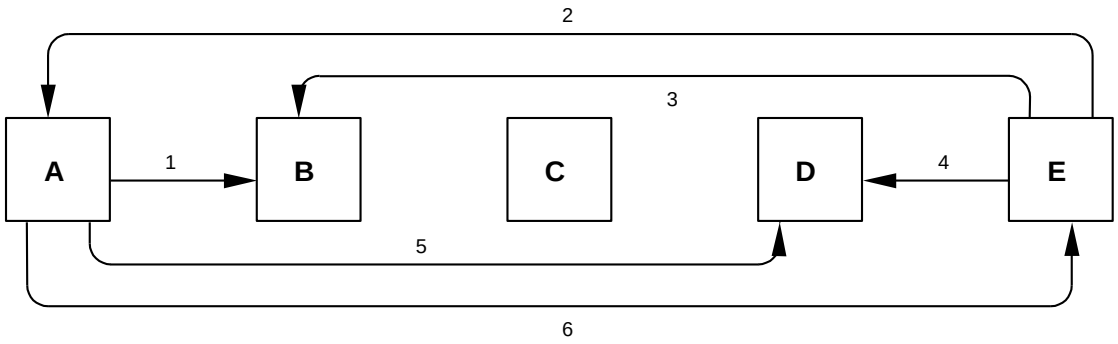
- Uzavřená skupina
  - pouze členové skupiny
  - vhodné pro kooperativní algoritmy
- Otevřená skupina
  - zasílat zprávy může kdokoliv
  - vhodné pro distribuované služby, replikované servery

- Vnitřní uspořádání skupin
  - všechny procesy rovnocenné
  - hierarchické uspořádání
  - koordinátor skupiny



Konkurenční zasílání zpráv

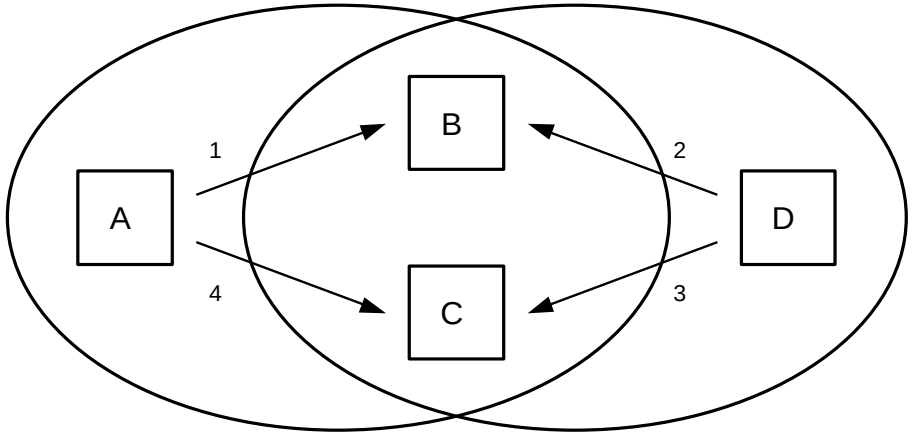
Problém:
různé pořadí doručení zpráv



Uzel B:
A: **x += 1;**
E: **x *= 2;**
Uzel D:
E: **x *= 2;**
A: **x += 1;**

Doručovací protokoly

- příjem
  - kdy přišla síťová zpráva
- doručení
  - předání příjemci

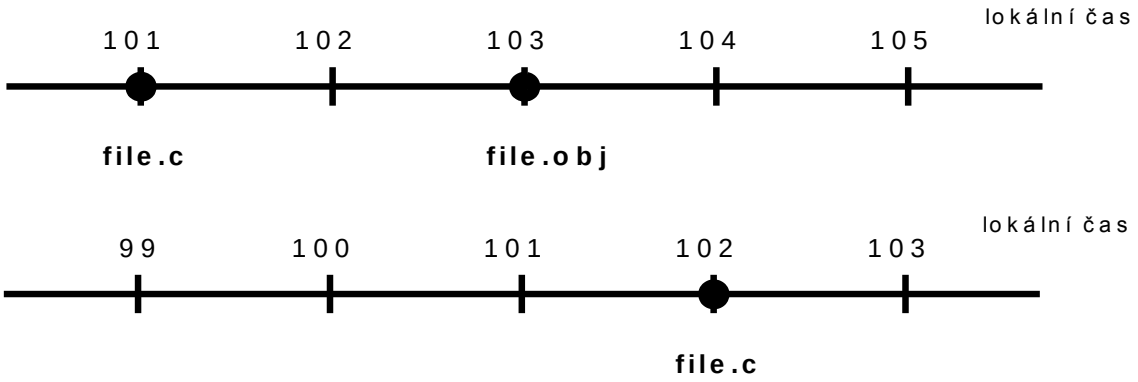


Překrývající se skupiny

Synchronizace

- logické a fyzické hodiny a jejich synchronizace
- distribuované vyloučení procesů
- volba koordinátora
- doručovací protokoly, virtuální synchronie, změny členství ve skupinách
- detekce globálního stavu
- distribuovaný konsensus

- neexistence sdílené paměti - zprávy
- rozprostřená informace mezi několika uzly
- rozhodování na základě lokálních informací
- vyloučení havarijních komponent
- neexistence společných hodin



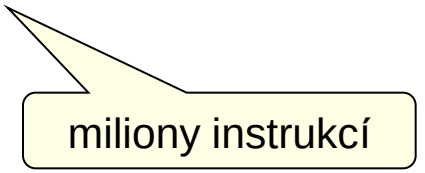
Fyzické hodiny

Synchronizace s fyzickým časem

- jak sesynchronizovat lokální hodiny jednotlivých komponent systému
- jak sesynchronizovat jednotlivé hodiny mezi sebou

Fyzický čas

- astronomické měření – solární sec. = 1/86400 solárního dne
- 1948 - atomové hodiny - 1s ≈ 9 mld přechodů atomu cesia 133
- 1950 - TAI - Bureau International de l'Heure – průměr asi 50 laboratoří
- 1 TAI den je o 3 ms kratší než solární den
- rozdíl >800 ms – vložená sekunda – Universal Coordinated Time UTC
- vysílání UTC (krátké vlny, satelit) – nepřesnost cca 0.1 - 10 ms



Fyzické hodiny

Synchronizace s fyzickým časem

- jak sesynchronizovat lokální hodiny jednotlivých komponent systému
- jak sesynchronizovat jednotlivé hodiny mezi sebou

Fyzický čas

- astronomické měření – solární sec. = 1/86400 solárního dne
- 1948 - atomové hodiny - 1s ≈ 9 mld přechodů atomu cesia 133
- 1950 - TAI - Bureau International de l'Heure – průměr asi 50 laboratoří
- 1 TAI den je o 3 ms kratší než solární den
- rozdíl >800 ms – vložená sekunda – Universal Coordinated Time UTC
- vysílání UTC (krátké vlny, satelit) – nepřesnost cca 0.1 - 10 ms

Poslední změna UTC 30.6.2012 neproběhla zrovna podle představ MTU. Problémy hlásily hlavně servery s OS Debian Linux, některé z nich postihl krátký výpadek. Poruchy hlásil také Reddit, který měl potíže kvůli databázím Cassandra a programům napsaným v Javě. V jeho případě však určitou roli zřejmě hrály také bouřky v Severní Virginii.

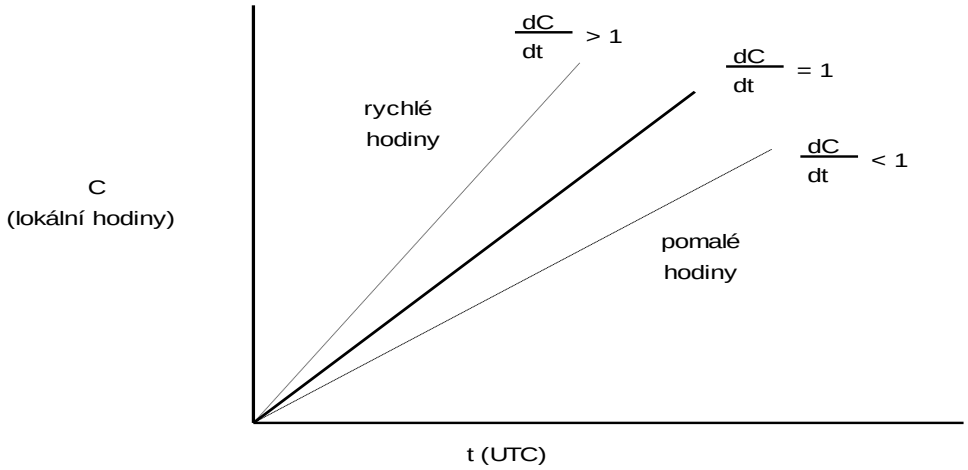
Synchronizace fyzických hodin

vnitřní hw hodiny C, fyzický čas t

hodiny na počítači p v čase t je Cp(t)

přesné hodiny: Cp(t) = t ∀t, tedy dC/dt = 1

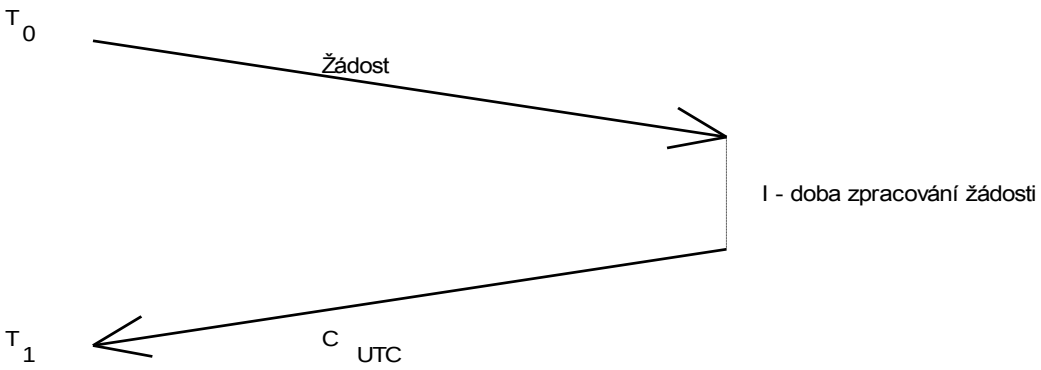
míra přesnosti ρ: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$



maximální odchylka dvojice hodin 2ρΔt

požadovaná odchylka δ ⇒ intervaly max. δ/2ρ

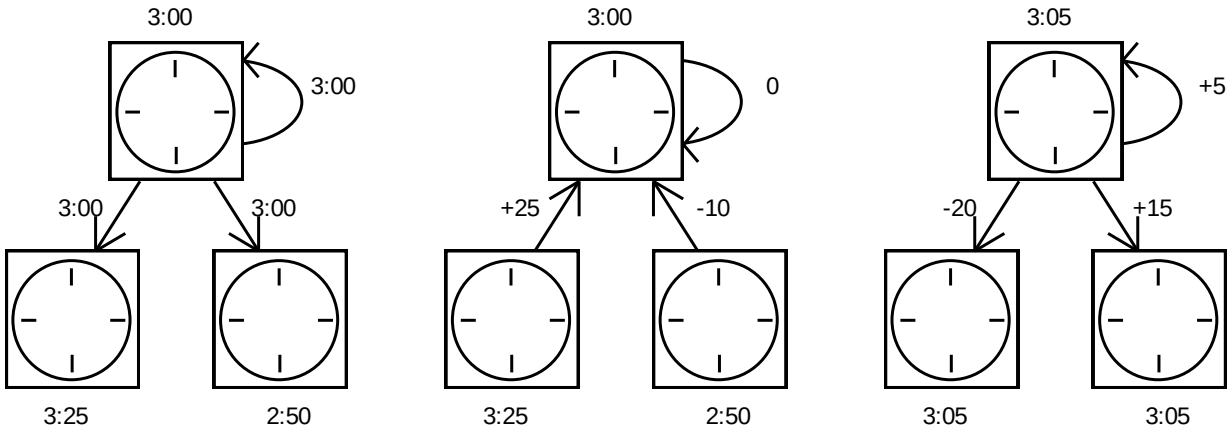
Cristianův algoritmus



- jeden pasivní time server (UTC)
- periodické dotazy v intervalech δ/2ρ
- T = T_UTC + (T1 - T0 - I)/2
- nikdy nepřeřizovat najednou !
  - (dis)kontinuita
  - postupná změna
  - zrychlování nebo zpomalování

Berkeley algoritmus

- aktivní time server
- periodicky se ptá ostatních na rozdíl času
- počítá průměr, vrátí rozdíl
- postupná synchronizace



Další algoritmy

Distribuovaný algoritmus

- žádný server ani koordinátor
- resynchronizační intervaly pevné délky
- i-tý interval: T0+iR .. T0+(i+1)R
- broadcast (ve fyzicky nestejný čas), spočítání průměru
- případné zahození extrémů

Intervalový čas

- čas není okamžik ale interval
- porovnání časů - některé časové údaje jsou neporovnatelné
- time clerk (klient) – určování času, započítání odchylky
- např. DCE – 33 knihovních funkcí

Logické hodiny

Problém: fyzické hodiny nelze dostatečně přesně sesynchronizovat

Lamport:

- důležité pořadí, nikoliv přesný čas
- nekomunikující procesy nemusí být sesynchronizovány

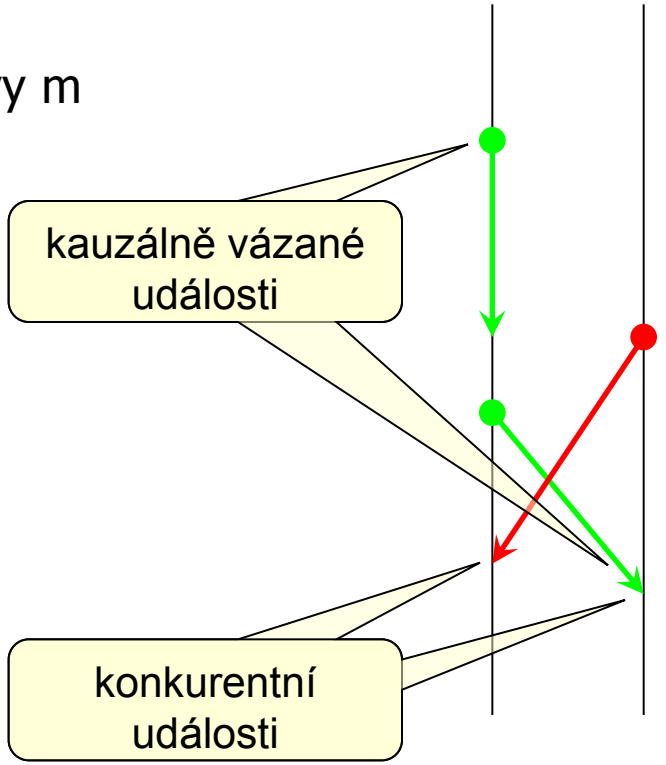
$e1 \rightarrow^p e2$ uspořádání v rámci procesu p

send(m) resp. rcv(m) je odeslání resp. příjem zprávy m

Kauzální závislost

- jestliže $\exists p: e1 \rightarrow^p e2$ potom $e1 \rightarrow e2$
- $\forall$ m: send(m) $\rightarrow$ rcv(m)
- jestliže $e1 \rightarrow e2$ & $e2 \rightarrow e3$ potom $e1 \rightarrow e3$

Události konkurentní: $e1 \not\rightarrow e2$ & $e2 \not\rightarrow e1$



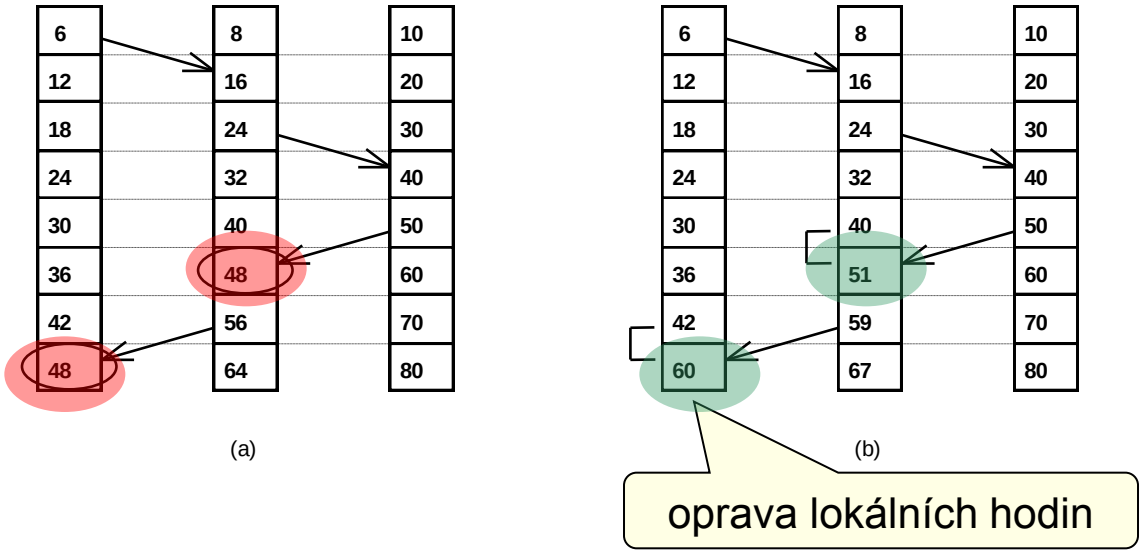
Logické hodiny

- událost a, čas C(a)
- jestliže $a \rightarrow b$ pak $C(a) < C(b)$

Synchronizace logických hodin

Synchronizace podle přijímání zpráv - časová značka zprávy Tm

- Proces *i* vysílá v čase $C_i(a)$ zprávu *m*; Tm = C_i (a)
- Proces *j* přijme zprávu *m* v čase $C_j(b)$. Pak C_j = max($C_j(b)$, Tm*+1*)



Zúplnění

- událost a v procesu i, událost b v procesu j
- C(a)=C(b) & Pi<Pj ⇒ C'(a)<C'(b)
- *'byrokratické'* uspořádání

Vektorové / maticové hodiny
- kauzalita
- doručovací protokoly
- virtuální synchronie

Logické hodiny a kauzalita

Kauzální závislost

- jestliže $\exists p$: e1 $\rightarrow^p$ e2 potom e1 $\rightarrow$ e2
- $\forall$ m: send(m) $\rightarrow$ rcv(m)
- jestliže e1 $\rightarrow$ e2 & e2 $\rightarrow$ e3 potom e1 $\rightarrow$ e3

Platí ☺

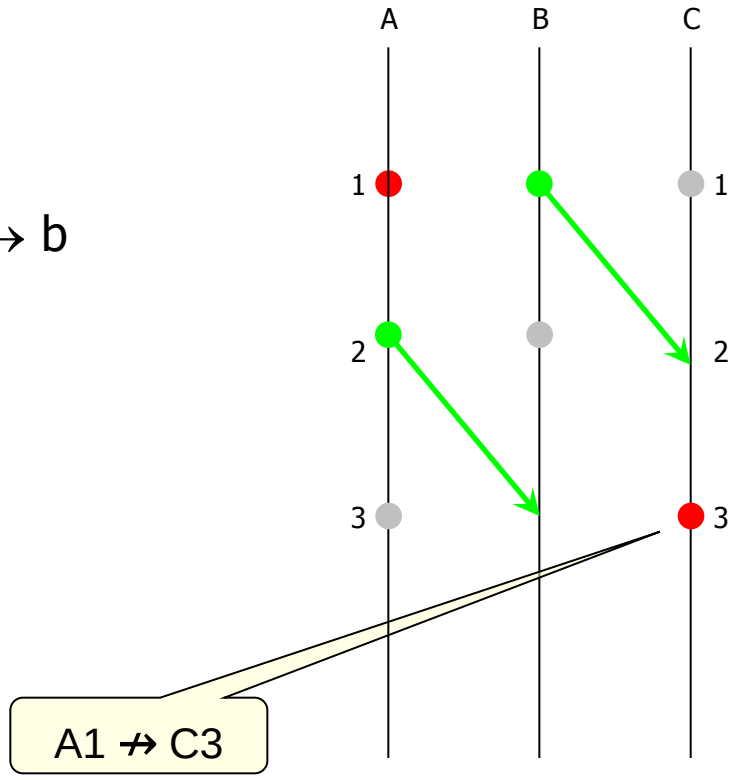
- jestliže a $\rightarrow$ b pak C(a) < C(b)

Neplatí ☹

- jestliže C(a) < C(b) pak by bylo hezké aby a $\rightarrow$ b

Jak tedy zjistím kauzalitu událostí?

- později - vektorové hodiny



Vzájemné vyloučení procesů

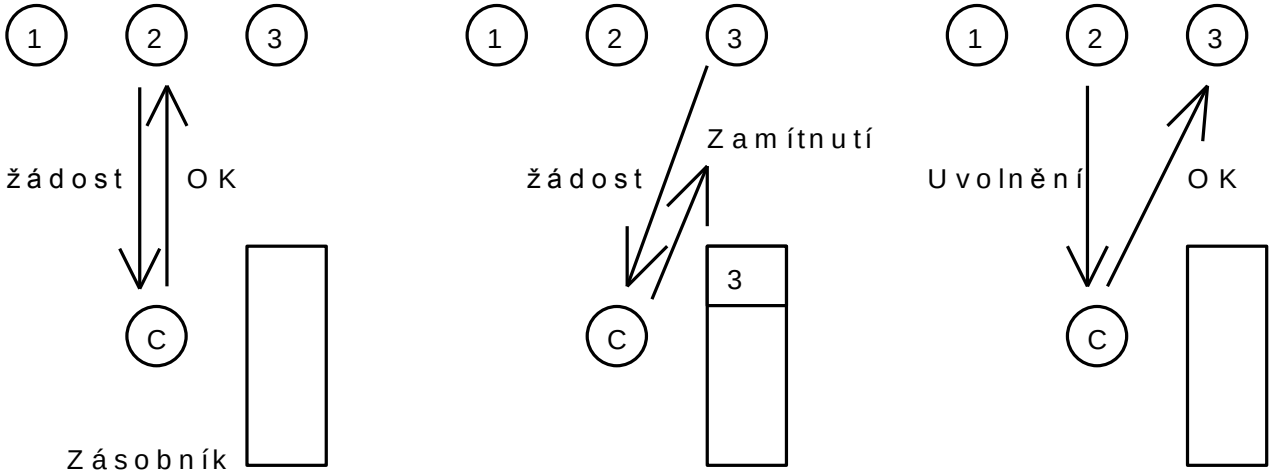
- neexistence společné paměti - semafor!
- pouze zprávy
- korektnost

- Způsoby řešení
  - ♦ centralizovaná ochrana přístupu
  - ♦ permission-based
    - časové značky — *Lamport* (3n), *Ricart-Agrawala* (2n)
    - volby — *Maekawa* (sqrt n), *Agrawal-ElAbadi* (log n)
  - ♦ token-based
    - token-passing — *Suzuki-Kasami*
    - strom — *Raymond*
    - token ring

Centralizovaný algoritmus

Centralizovaný algoritmus

- jeden server s frontou - sekvencér
- žádost / potvrzení / zamítnutí / uvolnění
- centralizovaná komponenta
  - ideově nevhodné
  - výpadek serveru - ztráta informace, nutnost volby
  - výpadek klienta - problém vyhladovění (starvace)



Lamportův algoritmus

Idea: Proces vyšle žádost a čeká až

- dorazí odpovědi od všech procesů a
- všechny žádosti v jeho frontě mají větší časovou značku

proces *p*, časová značka odeslání jeho zprávy *Tp*, fronta žádostí a potvrzení

zprávy typu *req*, *ack*, *rel*

akce se zprávami `send`, `add`, `del`

Událost	Akce procesu p
Žádost Mp	`send` Mp={req,p,Tp}
Přijetí žádosti Mi	`add` Mi; `send` {ack,p,Tp}
Přijetí potvrzení Ai	`add` Ai
Podmínka vstupu Mp	**∀ i≠p ∃ {ack,i,Ti} : Tp<Ti & ∀ {req,i≠p,Ti } : Tp<Ti**
Uvolnění Rp	`send` {rel,p,Tp}
Přijetí uvolnění Ri	`del` {req,i,Tk } : Tk < Ti

uložení žádosti a odpověď s vlastním časem

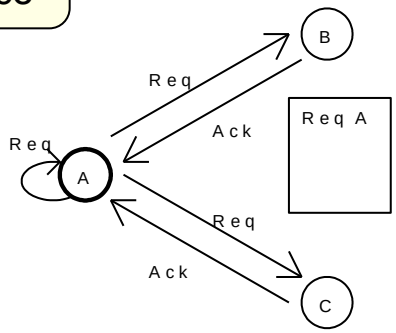
od všech přišlo novější potvrzení

neeviduju starší žádost

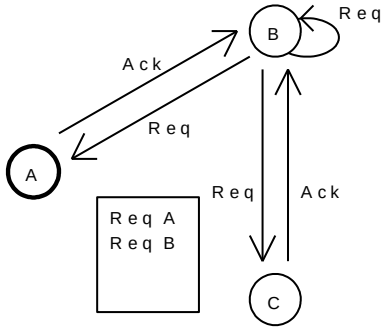
po přijetí uvolnění smažu starší žádosti tohoto procesu

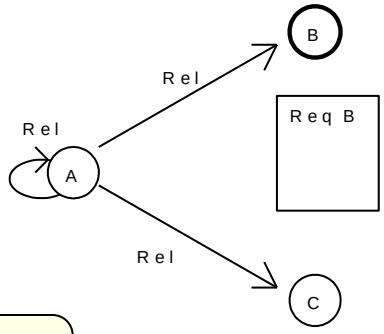
Lamportův algoritmus

Vstup A
do kritické sekce

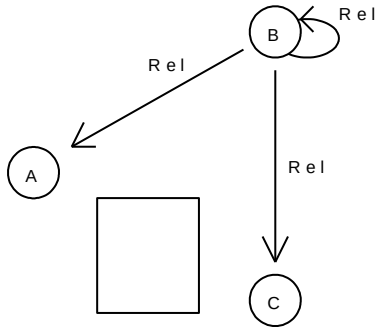


Žádost B o vstup





Uvolnění A
a vstup B



Uvolnění kritické
sekce

Lamportův algoritmus

initially: $(\forall q :: \neg x_q)$

program p:

```
do forever →
        { ¬x_p }
        x_p, F_p := true, c_p
    ; { x_p }
        ( || q : p≠q :  send F_p to co p,q
                    ;  receive-acknowledgement from co p,q
                       { ¬x_q ∨ F_p < F_q } { F_p < c_q }
        )
    ; { x_p } { (∀q : p≠q : (¬x_q ∨ F_p < F_q) ∧ F_p < c_q) }
        Critical Section p
    ;  x_p := false
od
```

Vyloučení procesů - Ricart & Agrawala

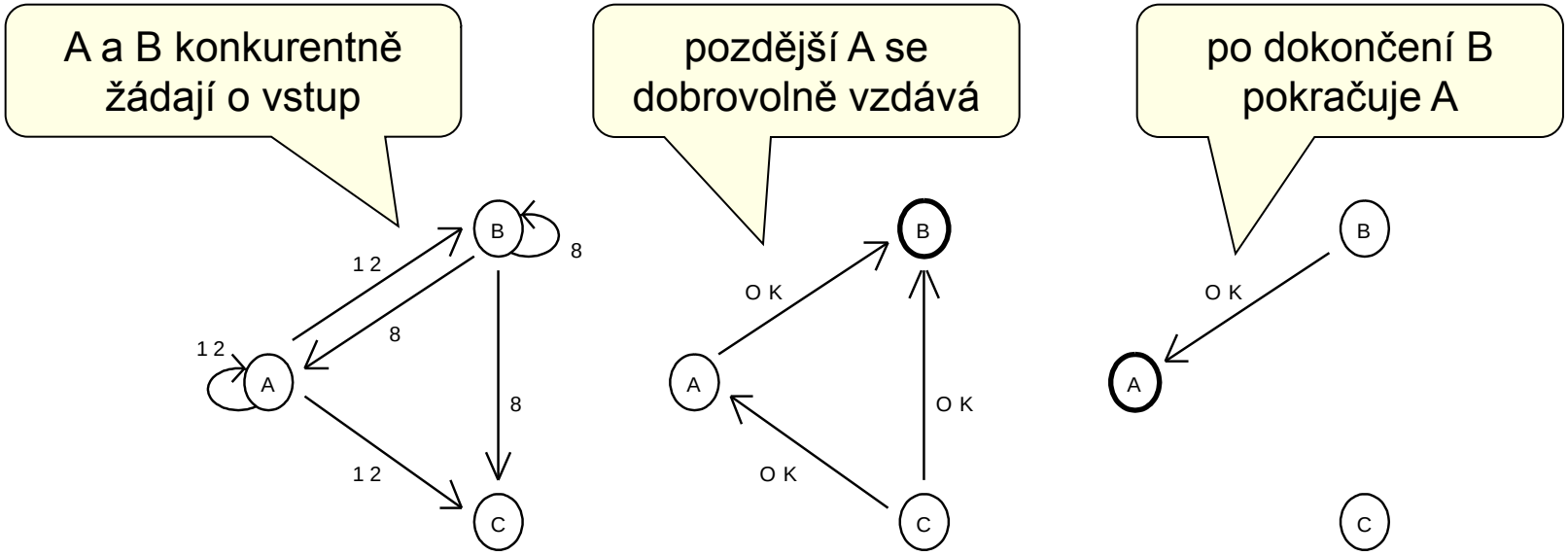
Proces chce vstoupit do kritické sekce:

- zašle žádost s TM ostatním procesům a čeká na všechny došlé odpovědi s potvrzením

Proces přijme zprávu se žádostí:

- Jestliže není v kritické sekci a ani do ní nechce vstoupit, pošle zprávu s potvrzením
- Jestliže je v kritické sekci, neodpovídá, požadavek si zařadí do fronty
- Jestliže není v kritické sekci, avšak chce do ní vstoupit, porovná TM žádosti s vlastní žádostí:
  - Pokud má vlastní žádost nižší časovou značku (byla vyslána dříve), neodpovídá a zařadí žádost odesilatele do fronty
  - V opačném případě (žádost odesilatele byla odeslána dříve) pošle zpět potvrzení

Po opuštění kritické sekce pošle proces potvrzení všem procesům, které má ve frontě



Vyloučení procesů - princip voleb

- Základní princip
  - procesy se snaží získat hlasy ostatních procesů
  - každý proces má v jeden okamžik právě jeden hlas
    - může ho dát sobě nebo jinému procesu
  - před vstupem do kritické sekce žádost o hlasy
    - pokud proces dostane víc hlasů než jakýkoliv jiný, může vstoupit
    - pokud dostane méně, čeká dokud nedostane dostatečný počet
  - problém: jak hlasovat a počítat výsledky, kdy už proces ví že vyhrál
- Naivní volby
  - každý proces zná všechny ostatní
  - ReceivedVotes / MyVote *(Available/Unavailable)*
  - při žádosti jiného procesu dá proces hlas pokud je Available
  - relativně odolný proti výpadkům
    - vydrží výpadek až poloviny procesů
  - složitost O(n)
    - není lepší než TS
  - deadlock!
    - více procesů může dostat stejný počet hlasů

Vyloučení procesů - pseudokód naivních voleb

```
Naive_Voting_Enter_CS()
  if (MyVote == Available)
  {
    for every( q != self)
      send(q, VoteRequest);
    ReceivedVotes = self;     // my vote
    wait while( ReceivedVotes < (M+1)/2);
    MyVote = Unavailable;
    enter_CS();
  }
```

čeká na nadpoloviční většinu hlasů

```
Naive_Voting_Monitor_CS()
  wait for( msg from p);
  switch( msg.type)
  { case VoteRequest:
      if (MyVote == Available){
        Send(p, VoteOk);
        MyVote = Unavailable;
      }
    case VoteRelease:
      MyVote = Available;
    case VoteOk:
      VotesReceived.Add(p);
  }
```

```
Naive_Voting_Exit_CS()
  for every( p in VotesReceived)
    if( p != self)
      send(p, VoteRelease);
  MyVote = Available;
```

Vyloučení procesů - Maekawa

- Mamoru Maekawa (1985)
  - organizace procesů pro optimalizaci komunikační složitosti
  - každému procesu p je přiřazen **volební okrsek S_p** (voting district)
  - pro vstup do kritické sekce je nutné získat všechny hlasy z vlastního okrsku

- Podmínky pro volební okrsky
  - $\forall p, q: S_p \cap S_q \neq \varnothing$ — korektnost
    - každá dvojice kandidátů má alespoň jednoho společného voliče
    - každý proces může volit pouze jednou $\Rightarrow$ nemohou být současně zvoleny dva procesy
  - $\forall p, q: |S_p| = |S_q| = K$
    - velikost volebních okrsků je konstantní, procesy potřebují stejný počet hlasů
  - $\forall p, q: |S_i: p \in S_i| = |S_j: p \in S_j| = D$ — spravedlnost
    - p je obsažen ve stejném počtu D volebních okrsků
    - každý proces má stejnou zodpovědnost

- Důsledek - komunikační složitost $O(|S_p|)$
  - cíl: minimalizovat velikost volebních okrsků

Maekawa - volební okrsky

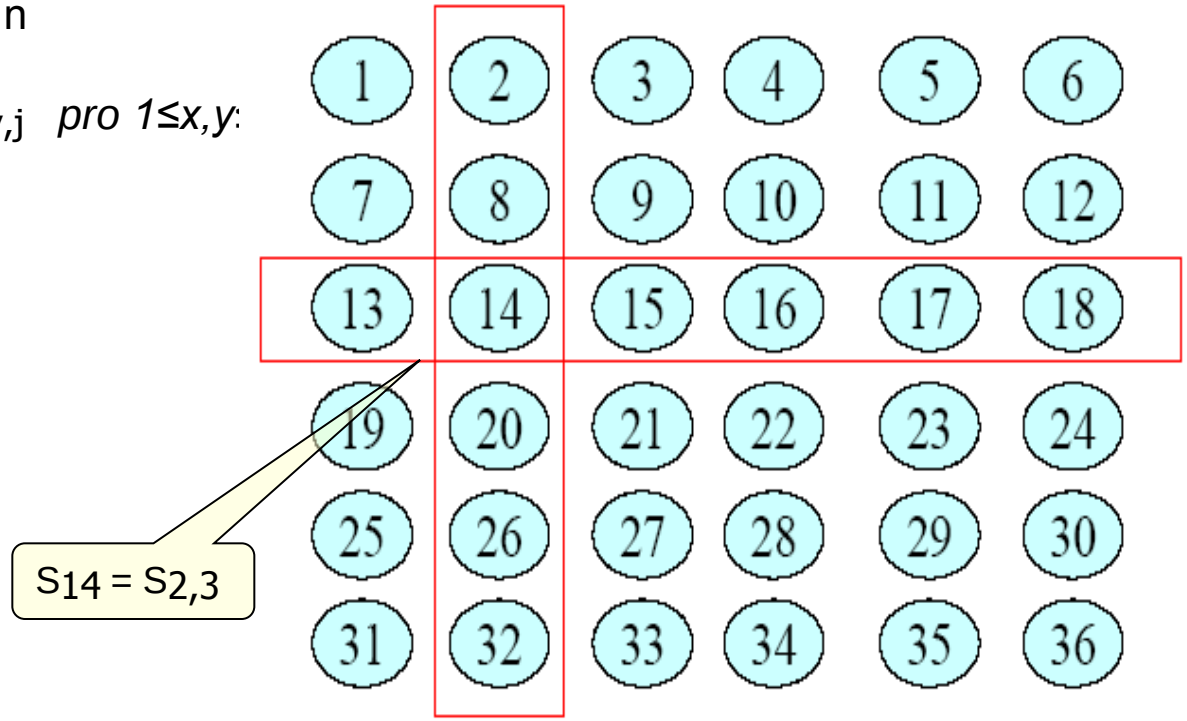
- Rozdělení do okrsků
  - jak velké okrsky?
  - v kolika okrscích má být každý proces?
- počet okrsků: $N \times K = D \times M$
- proces $\approx$ okrsek
  - musí získat všechny hlasy svého okrsku
  - $D \times M / K = M \rightarrow K = D$
- každý $q \in S_p$ je obsažen v D-1 jiných okrscích
  - max. počet okrsků $(D-1)K+1$
- $M = K(K-1)+1 \rightarrow K = O(\sqrt{M})$

K - velikost okrsku
N - celkový počet okrsků
M - počet procesů
D - počet okrsků ve kterých je každý proces

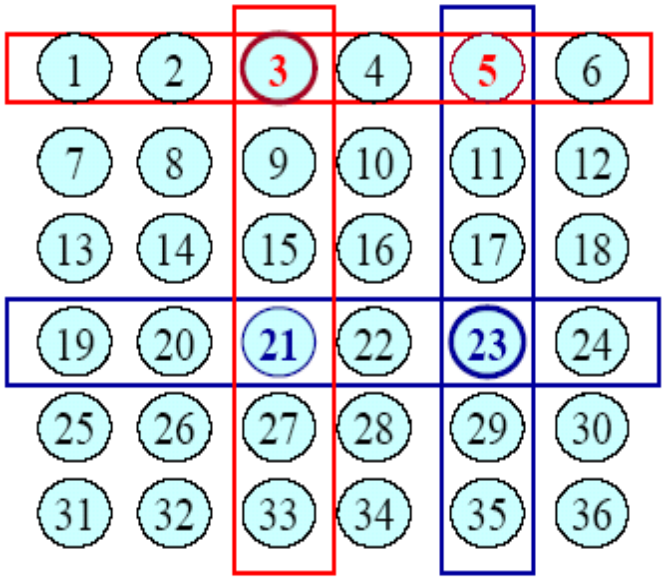
Maekawa - volební okrsky

K - velikost okrsku
M - počet procesů

- Algoritmus rozdělení procesů do okrsků
  - optimální pro M=K(K-1)+1 složitý
    - vyžaduje restrukturalizaci při změně členství
  - suboptimální pro K = O(√M) jednoduchý
    - prakticky použitelný
    - $M = n^2$, $P_{i,j}$, $S_{i,j}$, $1 \le i,j \le n$
    - $S_{i,j} = \bigcup P_{i,x} \bigcup \bigcup P_{y,j}$ *pro 1≤x,y*



Maekawa - deadlock a jeho řešení



Soutěží P3 a P23

Možnost deadlocku
P21 volí P23
P5 volí P3

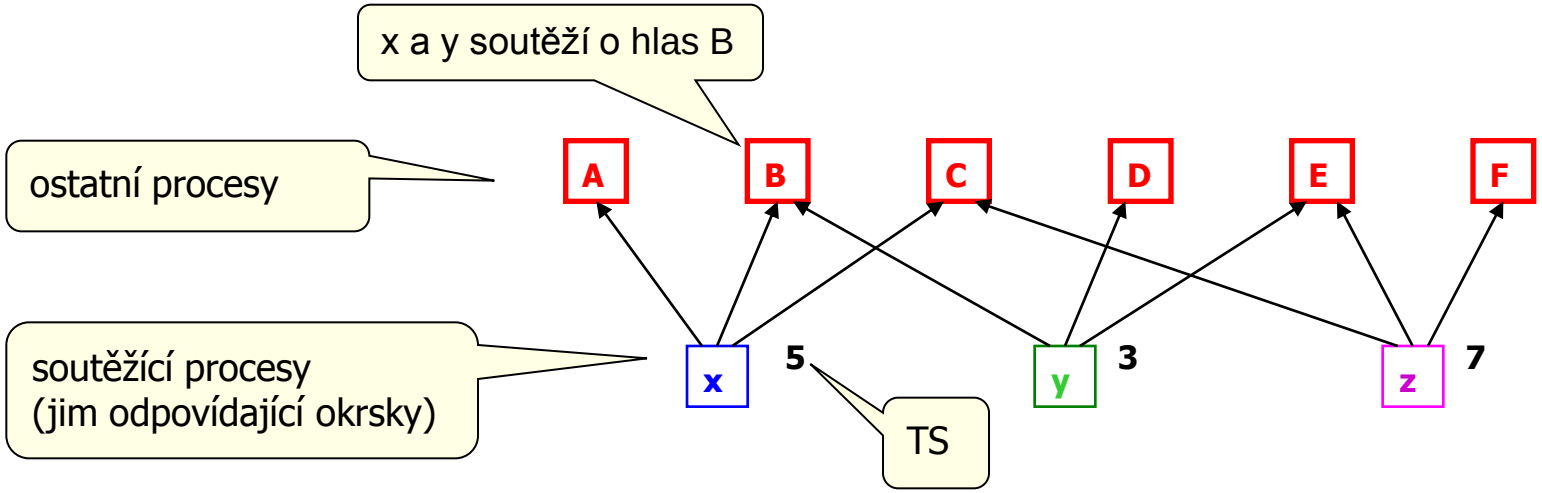
- Prevence deadlocku - logické hodiny
  - ♦ při příjmu žádosti procesu r s TS_r procesem p
    - p má volný hlas: potvrzení ACK_r
    - p dal již hlas jinému procesu q s $TS_q < TS_r$: zařadit do fronty
    - p dal již hlas jinému procesu q s $TS_q > TS_r$: pošle zprávu REJECT procesu q
      - pokud již q je v kritické sekci (dostal všechny potřebné hlasy), odpoví až po opuštění kritické sekce
      - pokud q ještě nemá všechny hlasy, vrátí hlas procesu p a ten ho předá procesu r

novější

starší !

Race Condition!!
Nevadí to ?

Maekawa - deadlock a jeho řešení



- Scénář 1
  - yD, yE - ACK
  - xA, xC - ACK
  - **yB - ACK**
  - y → CS
  - **xB - enqueue x**
  - y ← CS - REL
  - xB - ACK
  - x → CS

TS: x>y

- Scénář 2
  - yD, yE - ACK
  - xA, xC - ACK
  - **xB - ACK**
  - x → CS
  - **yB - REJECT x - ignored**
  - x ← CS - REL
  - yB - ACK
  - y → CS

x:CS

- Scénář 3
  - yD, yE - ACK
  - xA - ACK
  - **xB - ACK**
  - **yB - REJECT x - revoked**
  - yB - ACK
  - y → CS
  - y ← CS - REL
  - xC, xB - ACK
  - x → CS

TS: x>y

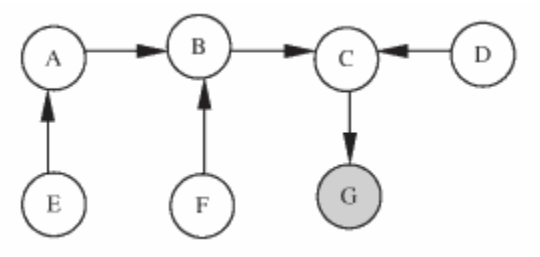
Vyloučení procesů - stromové algoritmy

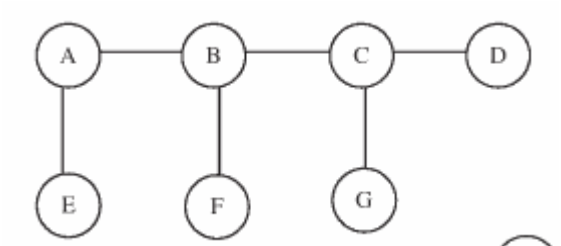
Agrawal & El Abbadi (1991)

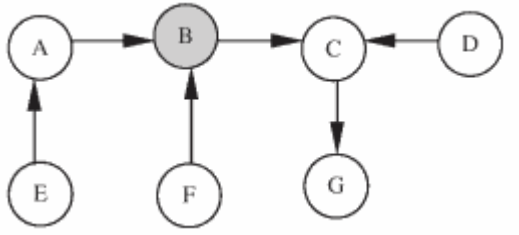
- Volby - složitost O(ln(n))
  - quorum = cesta od kořene k listu
  - fault tolerance
    - náhrada havarovaného uzlu jeho podstromy

Raymond (1989)

- token-based
- kostra, nezakořeněný strom
- každý uzel udržuje referenci na souseda směrem k token holderu
- při přenosu tokenu přesměrování referencí - rekonfigurace







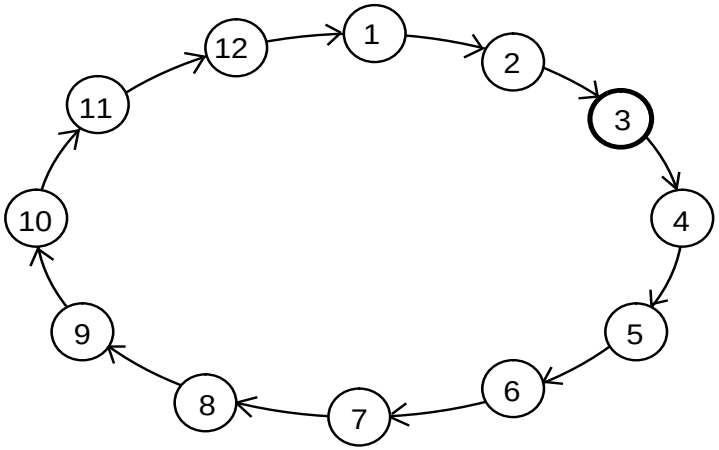
Vyloučení procesů - další algoritmy

Suzuki & Kasami

- přenos zprávy obsahující povolení ke vstupu do kritické sekce
- žádost o vstup - broadcast
- zpráva obsahuje frontu požadavků
- lze prioritní řazení

Token ring

- logický kruh
- přenos zprávy obsahující povolení ke vstupu do kritické sekce
- výpadek - broadcast, centrální registr



Porovnání algoritmů vzájemného vyloučení

Algoritmus	Počet zpráv	Problémy
Centralizovaný	3	Havárie koordinátora
Lamport	3(n-1)	výpadek libovolného uzlu
Ricart & Agrawala	2(n-1)	výpadek libovolného uzlu
Maekawa	2 $\sqrt{n}$	výpadek libovolného uzlu
Agrawal & El Abbadi	2 ln n	výpadek uzlu zvětšuje kvórum a složitost
Raymond	2 ln n	výpadek uzlu - rekonfigurace
Token ring	1 až ∞	ztráta peška, výpadek uzlu

Moudro:
Obvykle nemá smysl řešit centralizovaný problém distribuovaným algoritmem

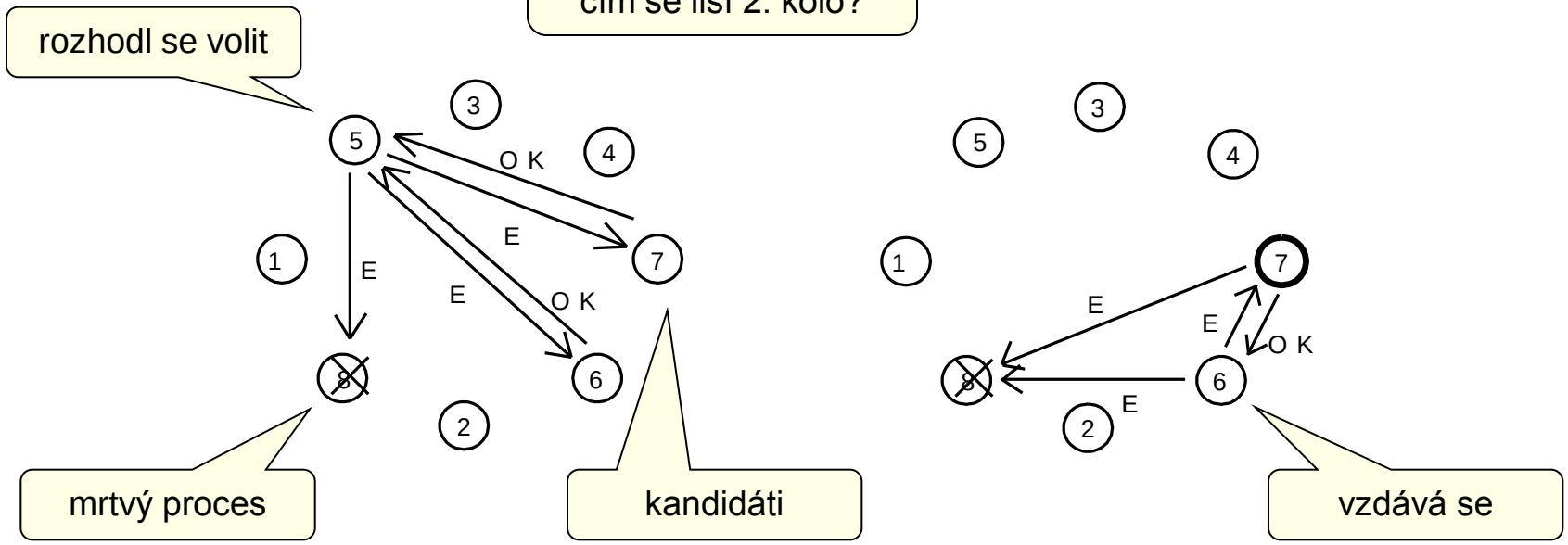
Volba koordinátora - bully algoritmus

- Předpoklady
  - omezená doba přenosu zprávy *(message propagation time)*
  - omezená doba zpracování zprávy a odpovědi *(message handling time)*
  - synchronní systém - '*spolehlivý*' detektor havárie: 2Tm+Tp

Pozor !! Velmi silný požadavek

- Když se proces rozhodne volit, zašle zprávu všem procesům s vyšší identifikací *(číslo procesu)*
  - když přijde odpověď, proces končí
  - když nepřijde nic, proces vyhrál, je novým koordinátorem, pošle zprávu všem ostatním
- Když proces přijme zprávu o volbě, vrátí zpět odpověď a vyšle žádosti všem vyšším procesům
- Volba se provede ve dvou kolech

proč zrovna ve dvou?
čím se liší 2. kolo?



Volba koordinátora - invitation algoritmus

- Asynchronní systém
  - nelze předpokládat nic o délce událostí
- Bully algoritmus
  - při překročení časových limitů možnost více koordinátorů
  - 💣☠✝👎☹

Invitation algoritmus *(Garcia-Molina 1982)*

Reálné prostředí

- Procesor: může havarovat (fail-stop), neomezená doba reakce
- Komunikace: rozpojení na segmenty, ztráta zpráv
- Nelze spolehlivě detekovat havárii: absence odpovědi neznamená havárii

Idea: koordinátor je vázán na skupinu, skupiny lze štěpit

- všichni členové stejné verze skupiny *(pohled, view)* vidí stejného koordinátora

Volba koordinátora - invitation algoritmus

Invitation algoritmus *(Garcia-Molina 1982)*

Reálné prostředí

- havárie, neomezená doba reakce, nespolehlivá detekce havárie

Idea: koordinátor je vázán na skupinu, skupiny lze štěpit

- všichni členové stejné verze skupiny *(pohled, view)* vidí stejného koordinátora

koordinátor: pravidelná výzva *AreYouCoordinator*

příjem AYC koordinátorem

- sjednocení do skupiny vyššího koordinátora

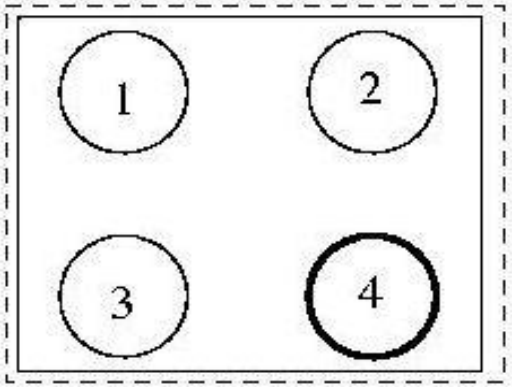
pokud člen skupiny neobdrží AYC svého koordinátora *(dostatečně dlouho)*

- prohlásí se za koordinátora vlastní nové skupiny
- pozvání ke sjednocení ostatním novým skupinám

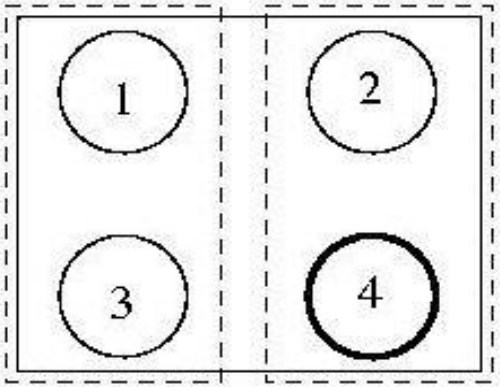
konzistence relativní vzhledem ke skupině

- procesy se shodují na členství ve skupině a na nějaké hodnotě
- koordinátor vyzve ostatní k připojení
  - pokud se nepřipojí, jsou konzistentní samy se sebou

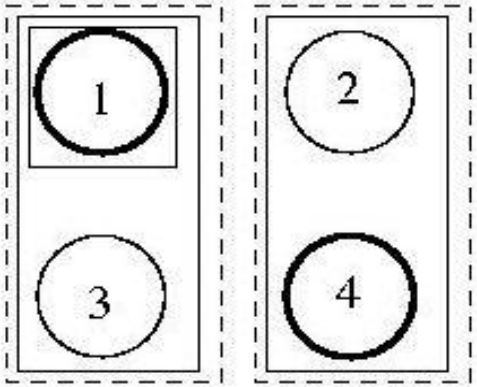
Invitation algoritmus - průběh



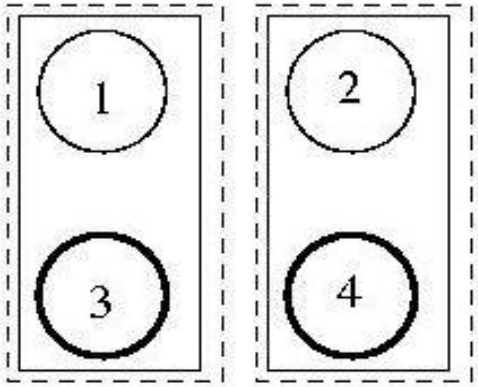
One partition. One group
Coordinator = Processor 4



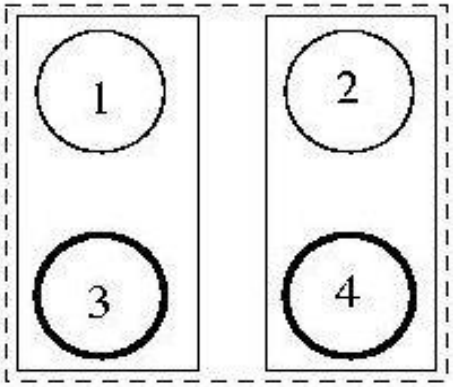
The network is partitioned, but no one notices



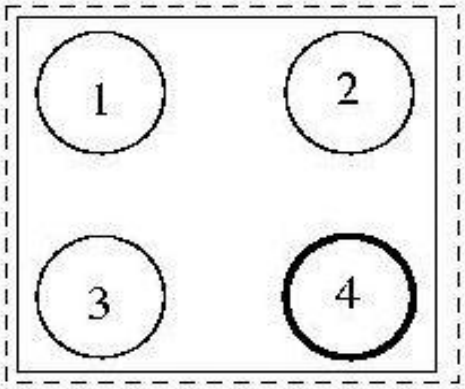
Processor 1 notices the partitioning. and declares itself a coordinator. It calls out and reaches processor 3.



Processor 3 realizes the partitioning and becomes a coodinator. Processor 1 retires from this role.

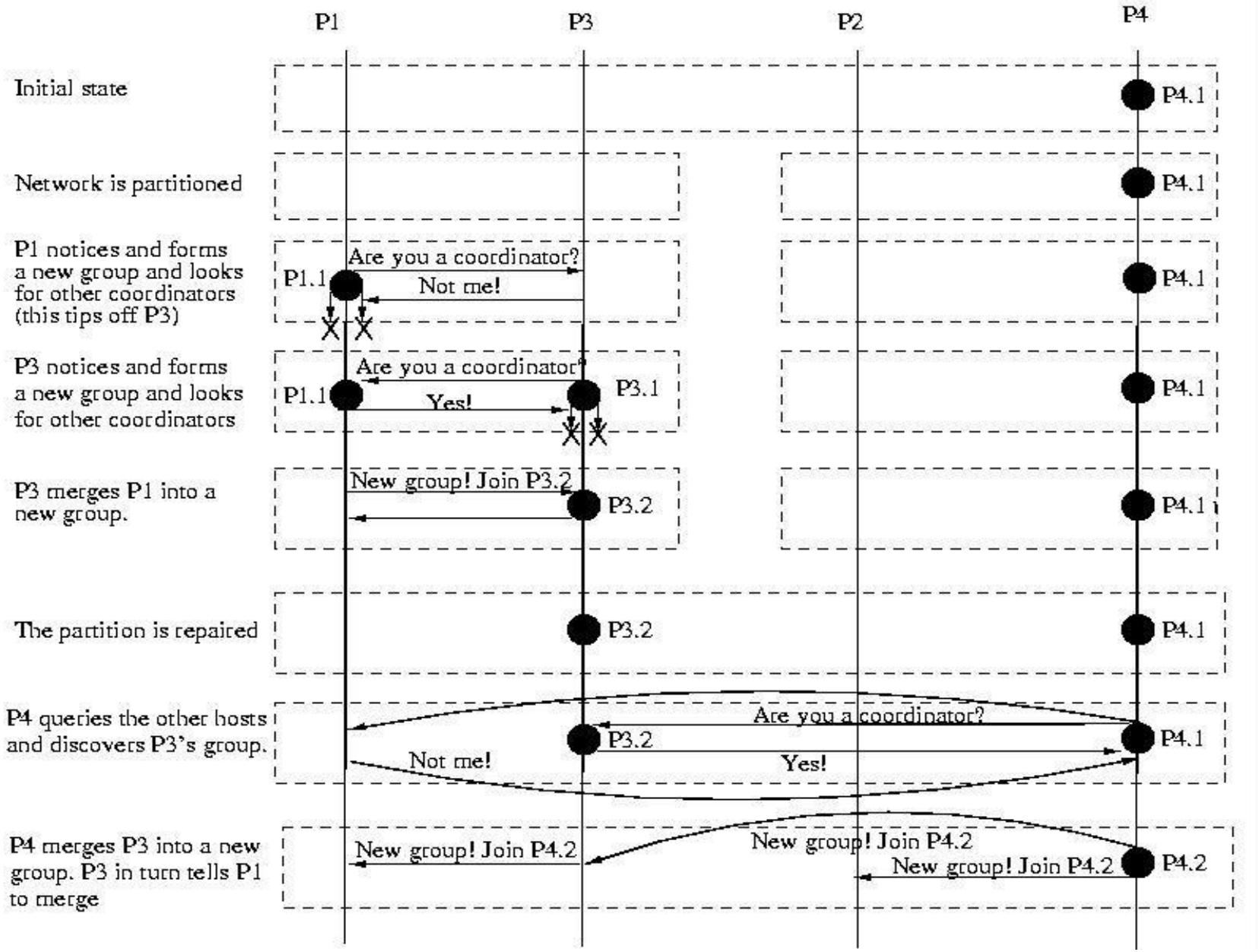


The network partition is repaired,. but none of the processors notice



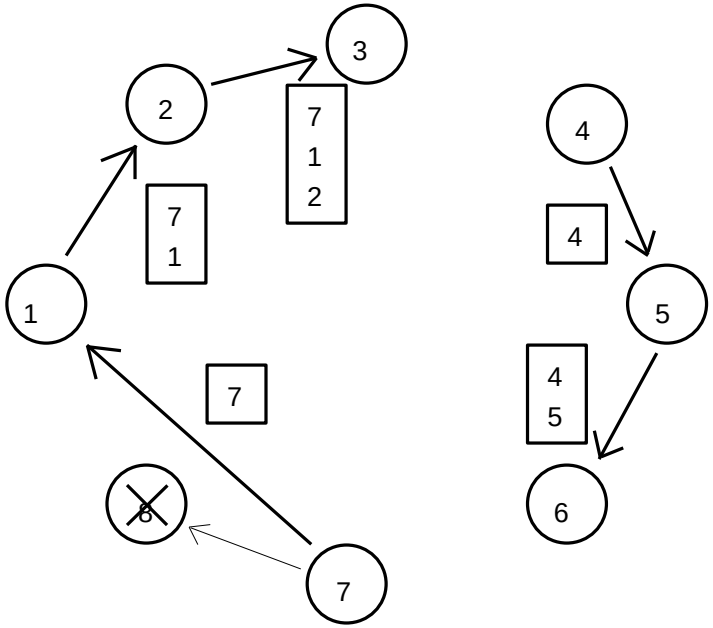
Either processor 3 or processor 4. discover that the network has been repaired. Processor 4 responds by announcing that it is coordinator and restoring global consistency.

Invitation algoritmus - vyzývací zprávy



Volba koordinátora - kruhový algoritmus

- proces se rozhodne volit, pošle zprávu následníkovi
- zpráva obsahuje čísla procesů (odesílatel a nejvyšší živý)
- po návratu obsahuje zpráva nového koordinátora
- následuje fáze oznámení
- stačí znát následníky a mít možnost zjistit následníka nedostupného uzlu
- složitost $O(n^2)$



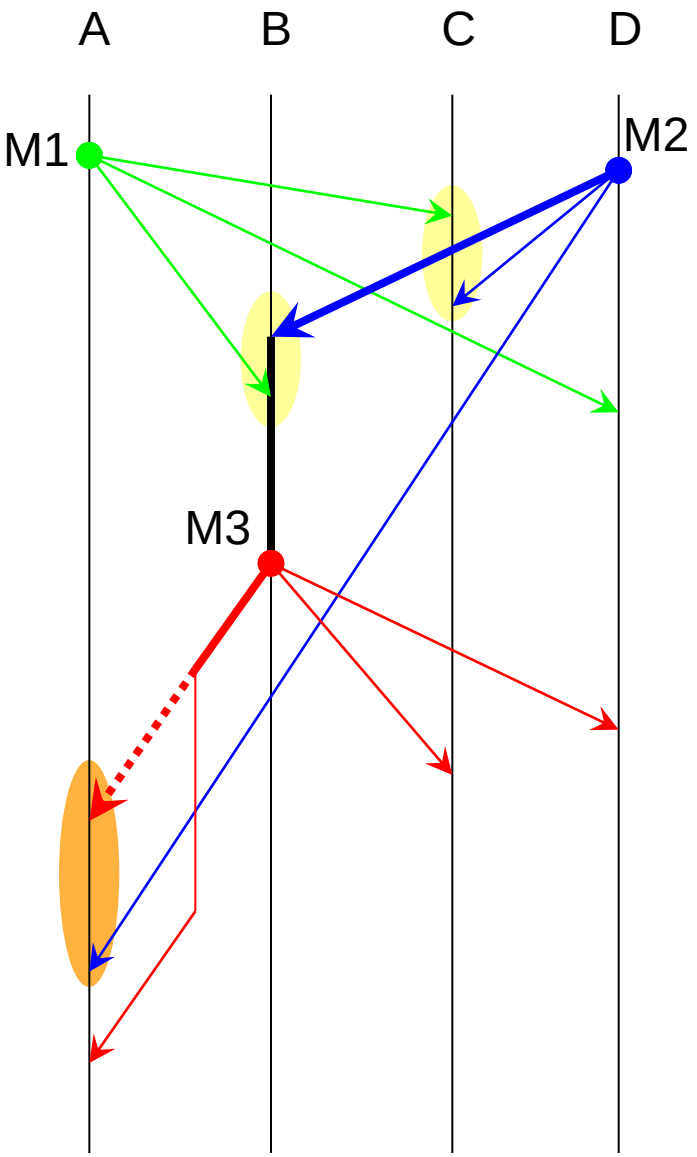
optimalizace komunikační složitosti

- koordinátor okolí 2^k
- složitost O(n log n)
- implementačně náročnější
- výhodné pouze při vysoké frekvenci konkurentních voleb a velkých skupinách

Doručovací protokoly skupinové komunikace

- Globální uspořádání
  - zprávy jsou doručovány v pořadí odeslání
  - nelze – neexistence globálních hodin
- Sekvenční uspořádání
  - všechny uzly doručí zprávu ve stejném pořadí
  - doručení nezávisí nutně na času odeslání
  - sekvencer, dvoufázový distribuovaný alg.
- Atomický multicast
  - spolehlivé sekvenční doručení
- Kauzální uspořádání
  - kauzálně vázané zprávy ve správném pořadí
  - konkuretní zprávy v libovolném pořadí

- Kauzálně vázané zprávy
  - obsah vázané zprávy **může být** ovlivněn
  - není podstatné, jestli obsah **byl** ovlivněn
- Konkurentní zprávy
  - ty, které nejsou kauzálně vázané



Kauzální doručování

Kauzální závislost *(opakování je matkou …)*

$\rightarrow$ *(kauzálně předchází)* je relace definovaná:

- jestliže $\exists p$: e1 $\rightarrow^p$ e2, potom e1 $\rightarrow$ e2
- $\forall$ m: send(m) $\rightarrow$ rcv(m)
- jestliže e1 $\rightarrow$ e2 & e2 $\rightarrow$ e3 potom e1 $\rightarrow$ e3

Kauzální uspořádání doručovaných zpráv

dest(m) množina procesů, kterým je zaslána zpráva m

$deliver_p(m)$ je událost doručení zprávy m procesu p

- $m1 \rightarrow m2 \Rightarrow \forall p \in (dest(m1) \cap dest(m2)) \ : \ deliver_p(m1) \rightarrow^p deliver_p(m2)$

Význam: Jestliže m2 je zpráva kauzálně závislá na m1, potom ve všech procesech, kterým budou doručeny obě tyto zprávy bude zachováno pořadí tohoto doručení

Vektorové hodiny

časové značky, vector clock, vector time, timestamps

Časová značka je vektor délky n, kde n je počet procesů ve skupině
časová značka procesu p: VT(p), časová značka zprávy: VT(m)

Obecná pravidla aktualizace časových značek

- při startu je $VT(p_i)$ nulový
- $\forall$ send $p_i(m)$: $VT(m) = ++VT(p_i)[i]$
- p_j doručující m upraví $VT(p_j)$:

  $\forall k \in 1..n: VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k])$
- $\forall\ m_i, m_j\ (i \neq j): VT(m_i) \neq VT(m_j)$

Pozor! Obecná pravidla neříkají KDY k doručení dojde

po složkách maxima ze stávajícího a přijatého vektoru

Porovnávání časových značek

- $VT1 \leq VT2 \Leftrightarrow \forall i: VT1[i] \leq VT2[i]$
- $VT1 < VT2 \Leftrightarrow VT1 \leq VT2 \ \&\ \exists i: VT1[i] < VT2[i]$

ne každá dvojice VT musí být porovnatelná - konkurentní

Doručovací protokol pro jednu skupinu

odeslání zprávy m procesem p_i

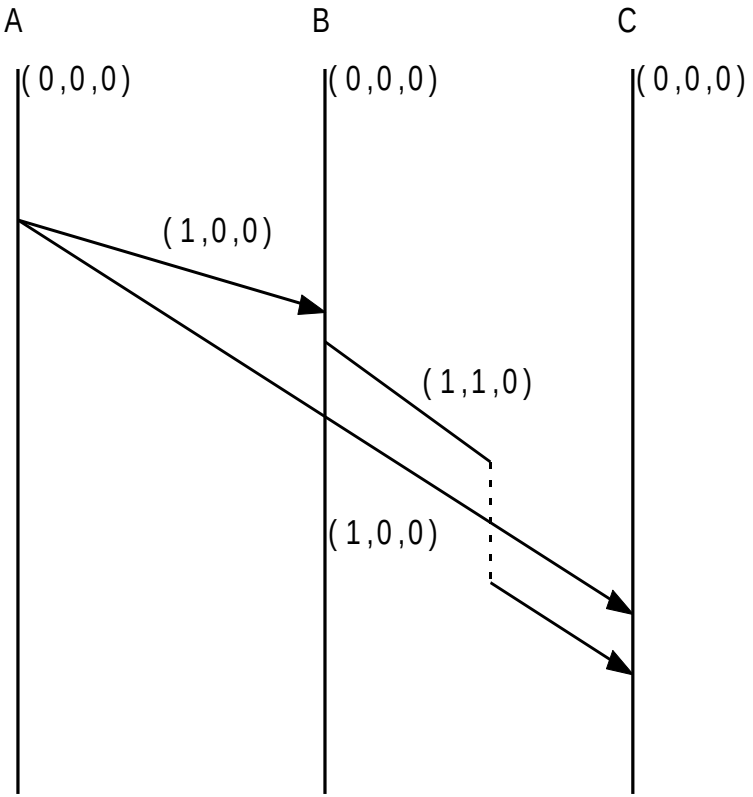
- $VT(p_i)[i]$ ++
- $VT(m) = VT(p_i)$

přijetí zprávy m procesem p_j

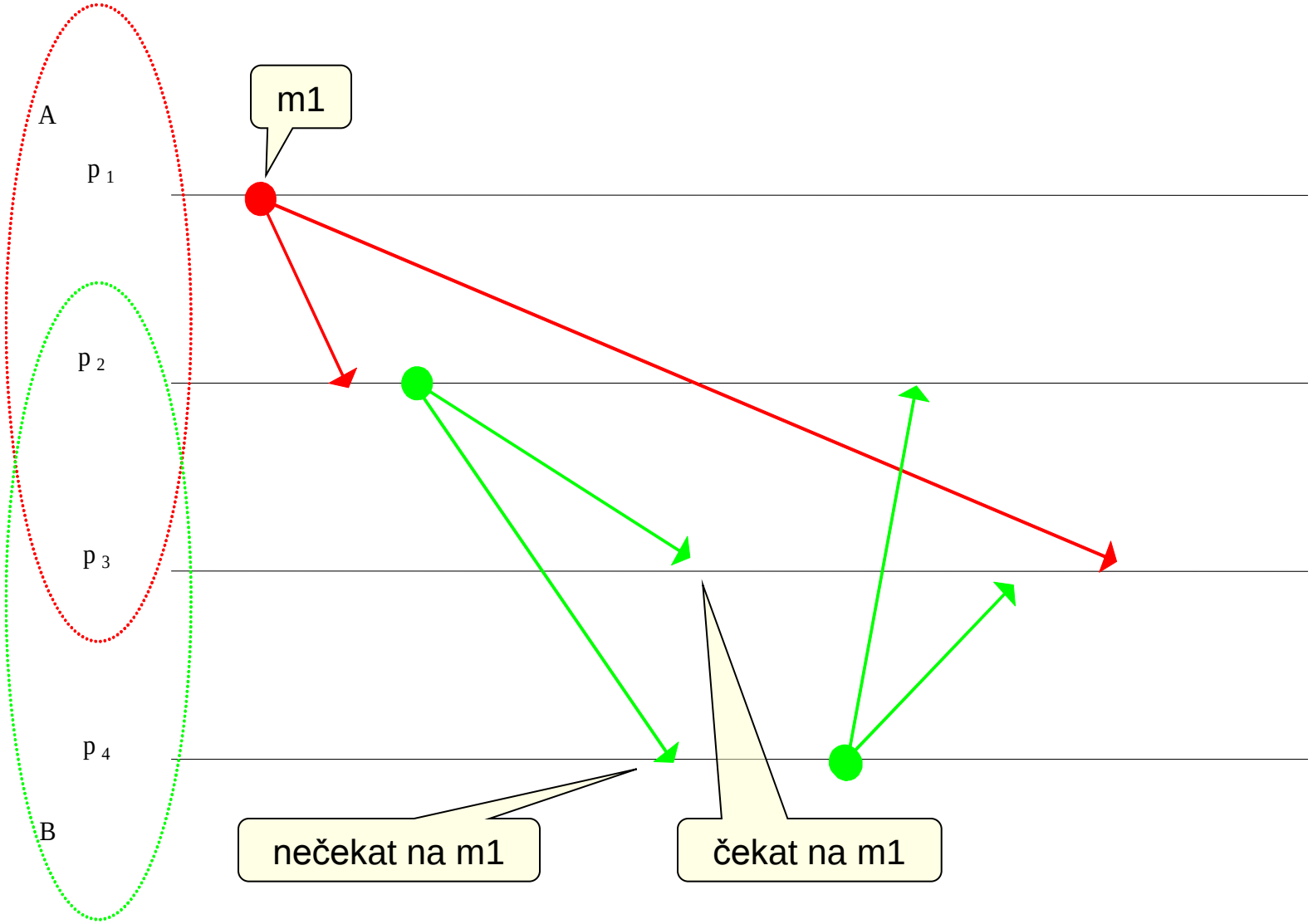
- proces pj ≠ pi pozdrží doručení m dokud:
  - $VT(m)[k] = VT(p_j)[k] + 1$ pro k = i
  - $VT(m)[k] \leq VT(p_j)[k]$ jinak

doručení

- po doručení si proces p_j upraví VT
  - $\forall k \in 1..n$: $VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k])$



Kauzální doručování a překrývající se skupiny



Doručovací protokol pro překrývající se skupiny

Vektory vektorových hodin - maticové hodiny

Časovou značku spojenou se skupinou g_a značíme VTa

VTa[i] vyjadřuje počet multicastů procesu p_i do skupiny g_a

S odesílanou zprávou posílá proces VT všech skupin, kterých je členem.

odeslání procesem p_i do skupiny g_a

- $VTa(p_i)[i]$++
- $VT(m) = \cup\ VTb(p_i)\ :\ p_i \in g_b$

VT všech skupin, kterých je odesílatel členem

Přijetí zprávy m procesem p_j (od p_i, VT(m), do g_a)

- Proces $p_j \neq p_i$ pozdrží m dokud neplatí:
  - $VTa(m)[i] = VTa(p_j)[i] + 1$
  - $\forall k\ (p_k \in g_a\ \&\ k \neq i)$: $VTa(m)[k] \leq VTa(p_j)[k]$
  - $\forall b\ (p_j \in g_b)$: $VTb(m) \leq VTb(p_j)$

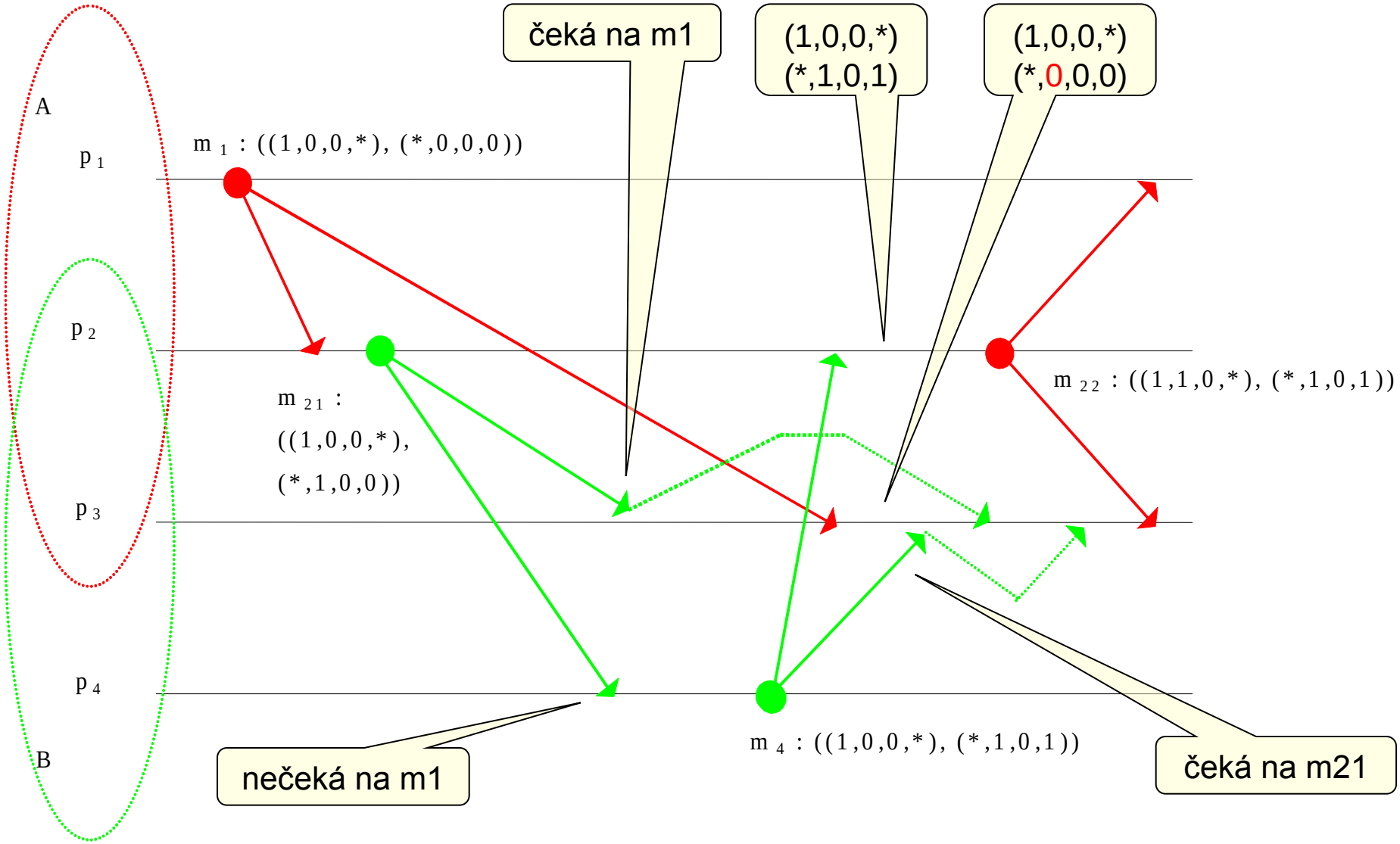
kauzalita vzhledem k skupině, do které byla zaslána zpráva

kauzalita vzhledem k ostatním skupinám příjemce

Doručení

- Po doručení si proces p_j upraví VT

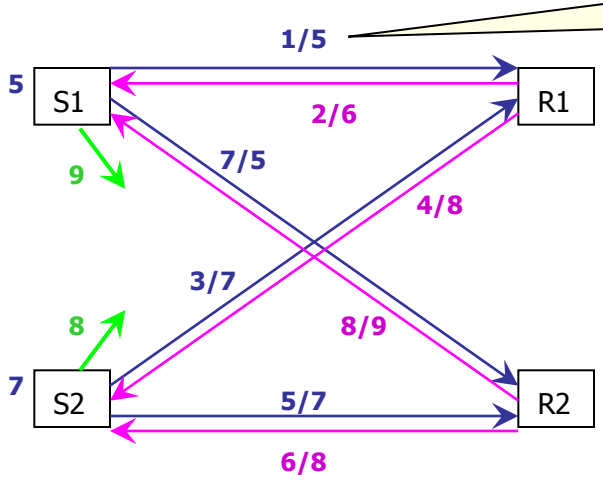
Kauzální doručování pro překrývající se skupiny



Distribuovaný total-order protokol

Totální / sekvenční doručování, total-order protocol
všechny zprávy jsou všem členům doručeny ve stejném pořadí

stačí skalární vektorové hodiny
při příjmu zprávy potvrzení odesilateli TSAi
odesilatel po příjmu všech potvrzení odešle finalizační zprávu s TSF = max(TSAi)
po příjmu finalizační zprávy doručí příjemce zprávy podle TSF



první zpráva,
časová značka 5

časové značky R1

msg	*TSA*	*TSF*
m0	2	deliv
m1	6	9
m2	8	8

zjednoznačnění
lokálních TS:
TS.proc_id

Komunikační složitost 3n
V praxi častěji řešení pomocí sekvenceru

- komunikačně jednodušší, nutnost výběru koordinátora, zaslání zpráv / pořadí

Spolehlivé doručování

'Naivní definice' - *všem členům skupiny bude doručena zpráva*

- Naivní implementace - spolehlivým kanálem včem členům skupiny odeslat zprávu
  - ♦ problémy:
    - ● výpadek příjemce
    - ● výpadek odesílatele
- Transakce
  - ♦ doručení pouze po commitu
  - ♦ funguje, ale příliš striktní
  - ♦ nepříjemné důsledky:
    - ● odesílatel neobdrží potvrzení - abort
      .. i když by stačilo něco méně drastického, např. redukce skupiny
    - ● havárie odesílatele po odeslání všech zpráv

Virtuální synchronie

Group **view** - množina uzlů ve skupině

Terminologie: delivery list, view, group membership, membership list ... **pohled**

Značení: L, L_i, L^x, L_i^x — globální pohled, lokální pohled procesu i, verze pohledu x

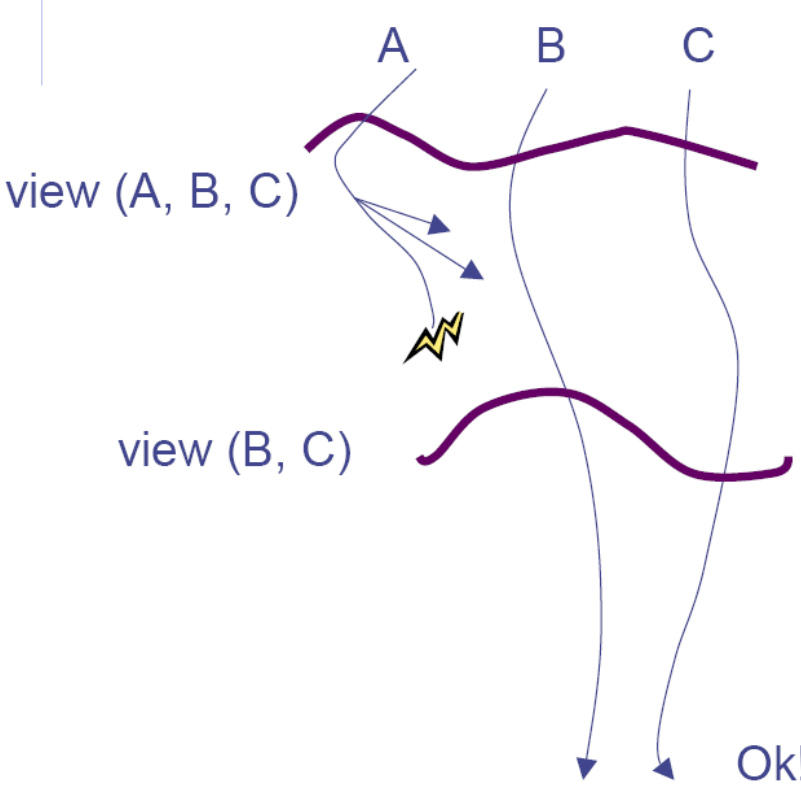
Algoritmus spolehlivého doručování je **virtuálně synchronní** když

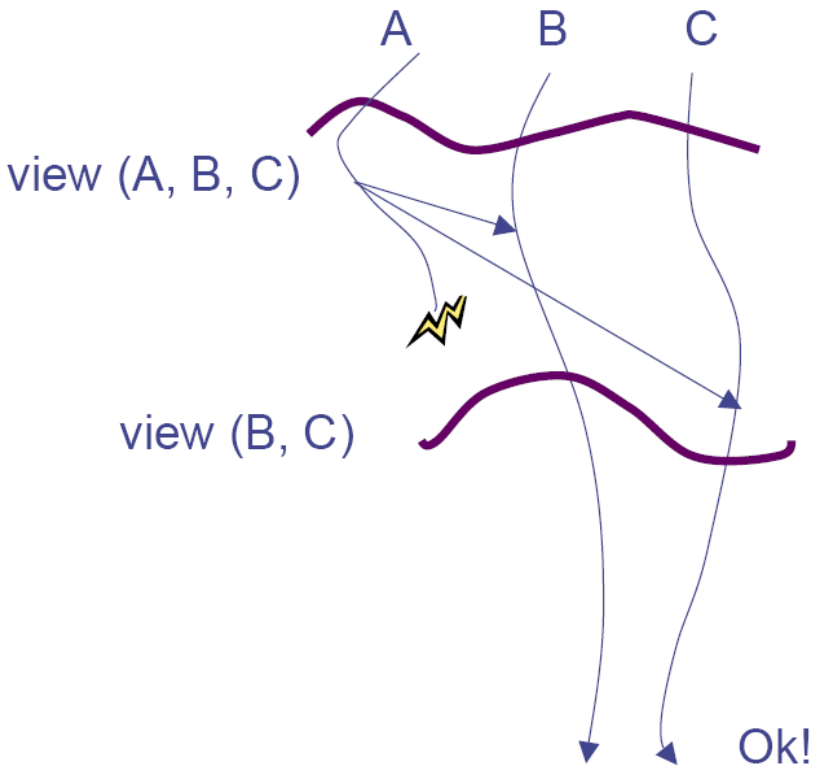
- všechny uzly ve skupině udržují stejný L
- pokud zpráva *m* je odeslána skupině s L^x před změnou na L^{x+1}
  - buď *m* doručí všechny uzly z L^x před provedením změny na L^{x+1}
  - nebo žádný uzel z L^x který provede změnu na L^{x+1} zprávu *m* nedoručí

Pozorování:

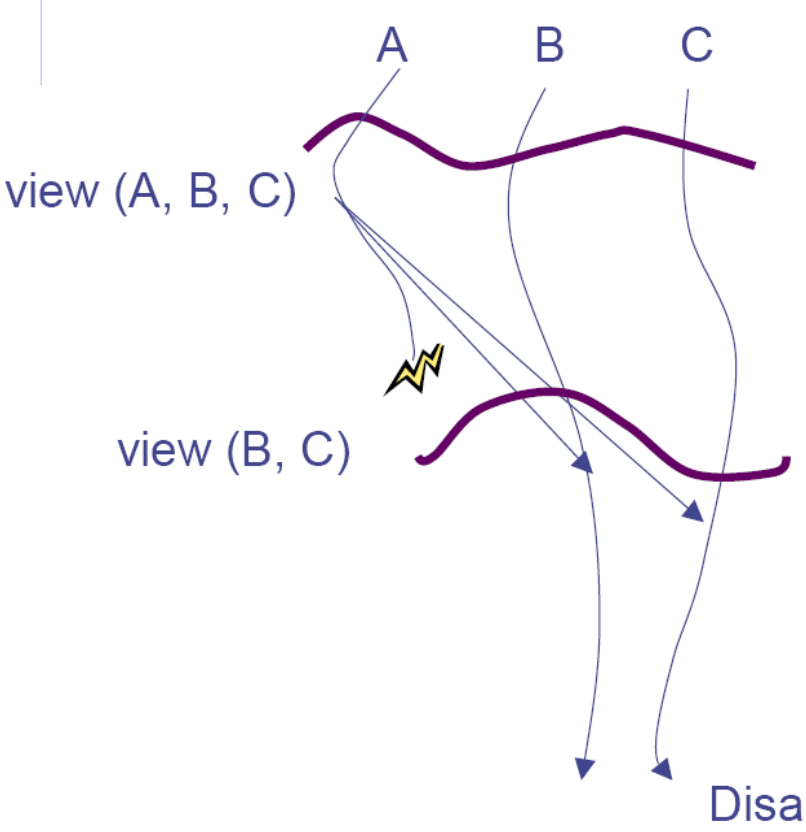
- Nemusí platit, že přijetí zprávy členem L implikuje doručení všem členům L
- *Odesílatel může také havarovat*

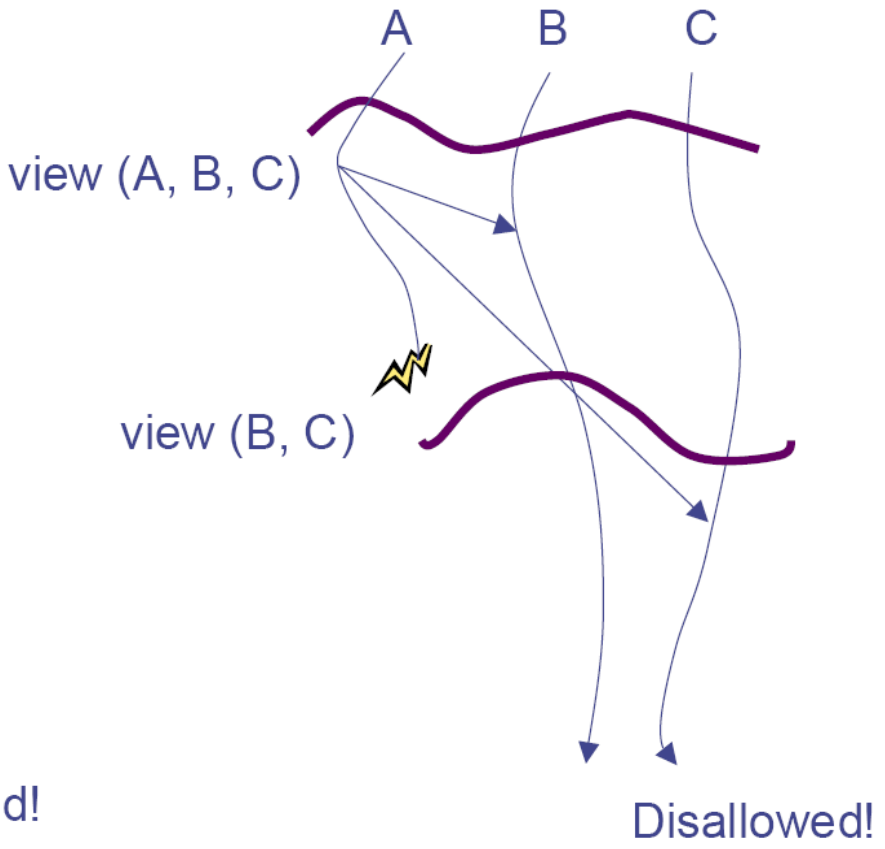
Virtuální synchronie - přípustné doručování





Virtuální synchronie - nepřípustné doručování





Spolehlivé kauzální doručování

- Záplavový (flooding) algoritmus
  - Při příjmu každé doposud nepřijaté zprávy ji každý uzel přepošle všem ostatním
  - nakonec všichni zbývající členové zprávu přijmou
  - spolehlivý, neefektivní

- Idea algoritmu s potvrzováním
  - p_i odesílatel zprávy, $p_j \in L$ příjemce zprávy, p_x havarovaný uzel
  - p_i odešle zprávu $\forall p \in L$, zprávu si uchová až do obdržení Ack(p) $\forall p \in L$ nebo do zjištění že p_x havaroval
  - p_j po příjmu zprávy odešle Ack(p_j) uzlu p_i, zprávu si uchová až do zjištění že zprávu přijaly $\forall p \in L$
  - jestliže p_j zjistí, že p_i havaroval, odešle zprávu $\forall p \in L$ o kterých neví, že zprávu přijaly

- 'Drobný' technický detail
  - jak p_j zjistí které uzly zprávu přijaly?

Trans algoritmus

Protokol pro spolehlivé kauzální doručování - *Melliar-Smith & co '90*

Potvrzení - graf závislostí (DAG)

Invariant:

- p rozesílá Ack(m) když přijal m a všechny kauzálně předcházející zprávy
- není nutné potvrzovat zprávy kauzálně předcházející m

transitive acknowledgements

Stabilní zpráva - přijata všemi členy skupiny

DAG G - **graf kauzality** celé skupiny

Gp - všechny zprávy které p přijal a ještě nejsou stabilní

Jestliže m potvrzuje zprávu m' pak (m, m') $\in$ G

orientace ve směru potvrzení, ne kauzality

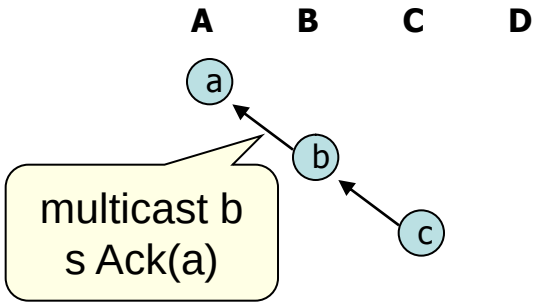
Zdrojem informací pro negativní potvrzování je G (nepřijaté zprávy)

Implementace - 3 komponenty:

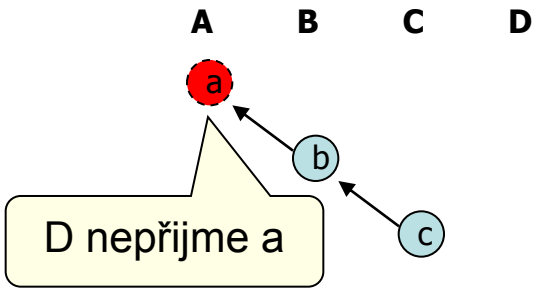
- příjem potvrzení a výpočet vlastních potvrzení
- detekce nepřijatých zpráv
- ukládání zpráv a detekce stabilních zpráv

Potvrzovací mechanismy

- Jak p_j zjistí které uzly zprávu přijaly?
  - rozesílání Ack(p) nejen odesílateli, ale $\forall p \in L$
    - neefektivní, mnoho potvrzení zbytečných - *n^2+n zpráv pro jedno doručení*
  - využití kauzality zpráv a piggybacking potvrzení



- D může po příjmu a, b+Ack(a), c+Ack(b) odvodit:
  - B přijal a
  - C přijal a i b



- D může po příjmu b+Ack(a), c+Ack(b) odvodit:
  - B přijal a
  - C přijal a i b
  - A odeslal a -> žádost o zaslání

Při korektním kauzálním doručování jsou potvrzení zpráv tranzitivní

Trans algoritmus - odesílání

Implementace

- přeposlání zprávy vždy včetně ACK
- ack_list, nack_list - seznam zpráv pro potvrzení / nepřijatých zpráv
- undelivered_list – seznam přijatých ale ještě nedoručených zpráv

```
Trans_send(m)
        m += nack_list
        m += ack_list
        ack_list = m
        G += m
        send m to every proc
```

nak_list zůstává až do příjmu zprávy

vyčištění ack_listu, přidat m

m do grafu kauzality

Trans algoritmus - příjem

```
Trans_Receive(m)

        foreach nak(n) ∈ m
                if n ∈ Gp
                        multicast n
        if m ∈ Gp
                exit
        foreach ack(n) ∈ m && n ∉ Gp
                nak_list += n
        if m ∈ nak_list
                nak_list -= m
        undelivered_list += m
        G += m
        foreach n ∈ undelivered_list && causal(n)
                undelivered_list -= n
                deliver n
        foreach n ∈ Gp && causal(n)
                   && neex( n': causal(n') && (n',n) ∈ Gp)
                ack_list += n
        foreach n ∈ Gp && stable(n)
                Gp -= n
```

včetně ACK!

zaznamenání nepřijaté zprávy

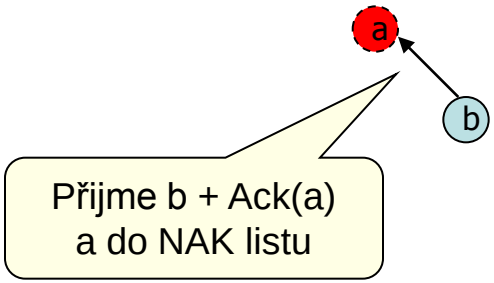
všechny kauzálně předcházející byly doručeny

všechny kauzální zprávy které nemohly být tranzitivně potvrzeny

n' -> n, n' potvrzuje zprávu n

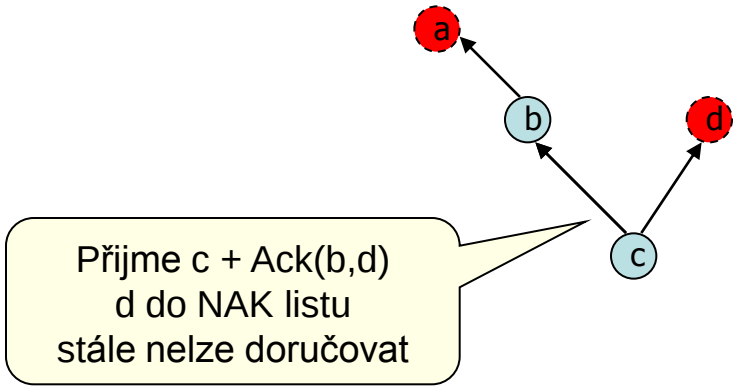
vyčištění všech stabilních zpráv

Průběh Trans algoritmu



ACK: {}

NAK: {**a**}

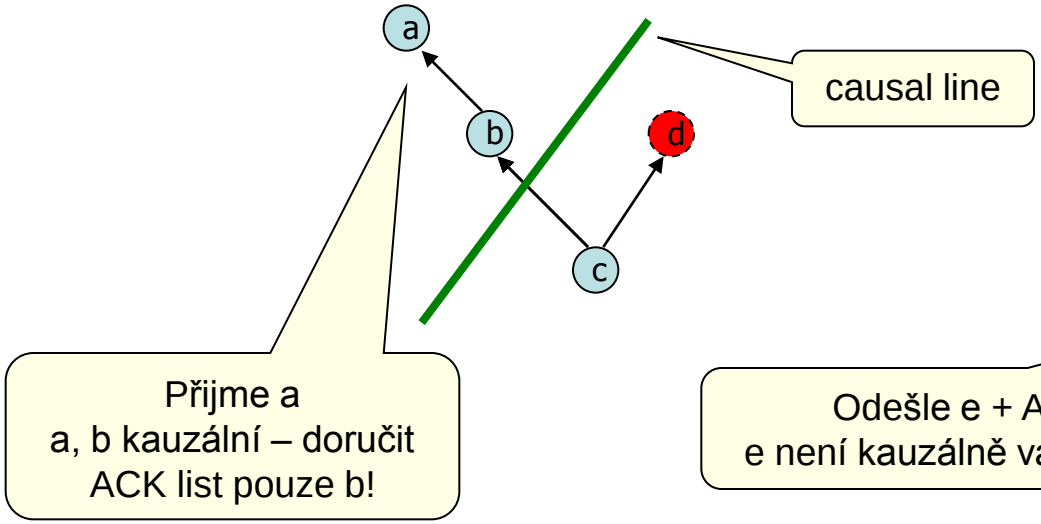


ACK: {}

NAK: {a,**d**}

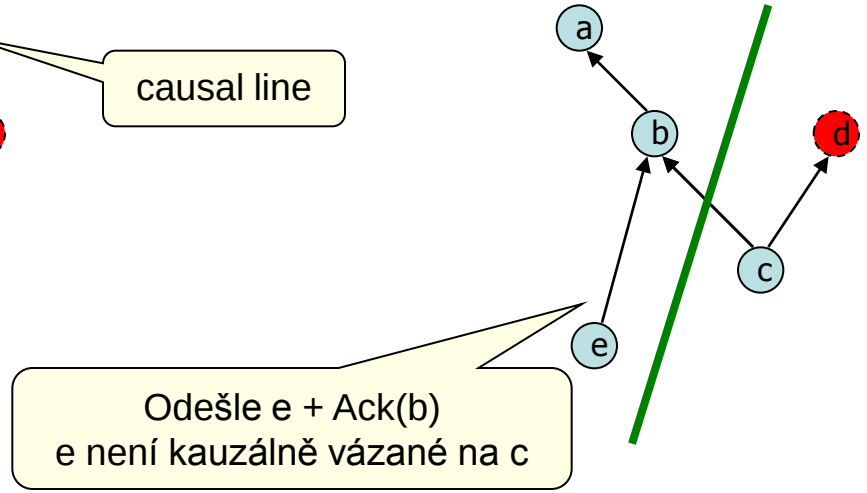
ACK: {**b**}

NAK: {d}



ACK: {**e**}

NAK: {d}



Trans protokol – pozorování

pohled na protokol - posun causal line a stable line

nově příchozí zpráva vně casual line

pokud všechny kauzálně předcházející zprávy jsou uvnitř,

pak i zprávu lze dát dovnitř causal line

Pozorování:

- je-li zpráva doručena spolehlivému procesu,

  pak někdy každý nehavarující proces zprávu dostane
- zprávy jsou doručovány v kauzálním uspořádání
- G reprezentuje graf závislosti v kauzálním uspořádání
- všechny procesy vytvářejí stejný graf závislostí
  - pořadí přidávání uzlů však může být různé
- jestliže procesor havaruje, paměťová náročnost je neomezená ! ☹👎
  - slabina - pro praktické použití musí být Trans protokol doplněn nějakým protokolem pro změnu členství ve skupinách

Členství ve skupinách

důležitá součást spolehlivých doručovacích protokolů

podmínka vzájemné konzistence: $p \in L_q \Rightarrow q \in L_p$

striktní chování - asynchronní shoda

- nelze *motivace a 'důkaz' později*
- slabší *podobně jako např. D2PC*
- typicky pesimistické modely
  - považuje za havarovaný i ten uzel, který je detekován od jediného uzlu

- Sekvence pohledů (group view sequence) $L^0, L^1, L^2, \ldots$
  - hodnota L^i je shodná pro všechny $p \in L^i$
  - uzly instalují pohledy ve stejném pořadí
- Virtuální synchronie
  - $p, q \in L^x, L^{x+1}$
  - $\text{install } L_p^x < \text{deliver}_p(m) < \text{install } L_p^{x+1} \Rightarrow \text{install } L_q^x < \text{deliver}_q(m) < \text{install } L_q^{x+1}$

doručení zprávy respektuje pohledy

- Přesná formální definice konzistentní změny pohledů
  - konzistentní řezy, běhy, temporální logika, ...
  - nejednotná

Členství ve skupinách - Transis algoritmus

Transis - spolehlivý kauzální multicast, členství ve skupinách

- vychází z Trans, založen na principech Trans a ISIS
- Technion, '92 - Amir, Dolev, Kramer, Malki http://www.cs.huji.ac.il/labs/

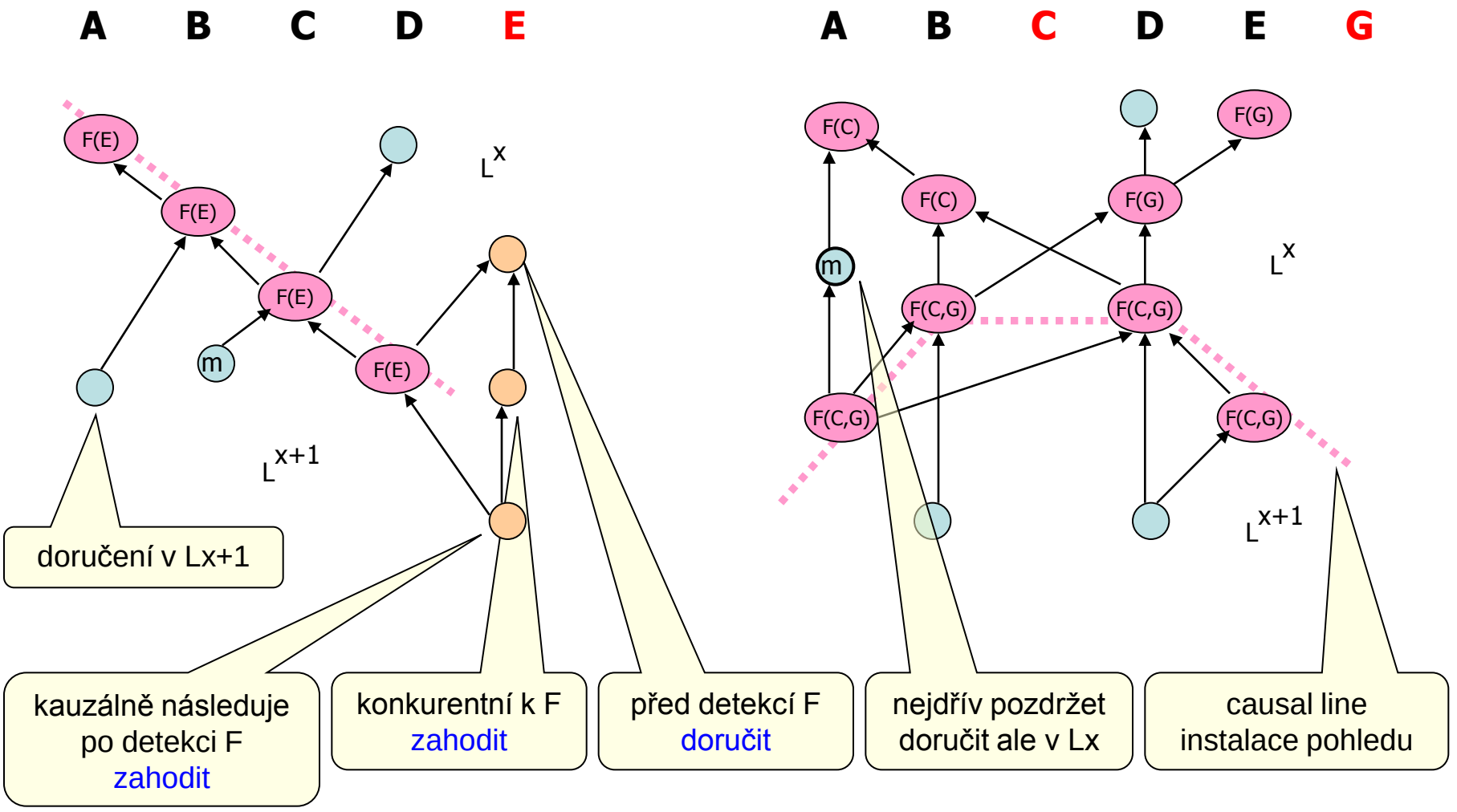
Monotónnost protokolu

- **paranoia** - jedno podezření stačí k vyloučení
  - k vyloučení stačí větší zpoždění - nerozeznatelné od havárie
- **jednosměrnost** - vyloučené procesy se již do algoritmu nevrací
  - lze se ale explicitně znovu připojit

Idea: konzistentní změny pohledů, doručování v rámci pohledů

- při detekci havárie zpráva FAULT(q), při přijetí její přeposlání
- kauzální hranice pohledu
  - doručení předcházejících zpráv
  - pozdržení zpráv kauzálně následujících
- konkurentní detekce různých havárií - společná hranice, doručení obou zpráv
  - doručení pozdržených zpráv
- kauzální doručení zpráv havarovaného procesu vzhledem ke změně pohledu

Průběh Transis algoritmus



Transis algoritmus

G	graf kauzálních závislostí stejný jako u Trans: m potvrzuje m' $\Rightarrow$ (m,m') $\in$**G**
L	aktuální pohled (množina procesů)
F	procesy označené za havarované (Fault)
Fault(q)	zpráva s detekcí havárie procesu q
blocked	zprávy blokované pro doručení v následujícím pohledu
Last[i]	procesy označené procesem i za havarované
J	procesy přidávané do pohledu
JLast[i]	procesy naposledy označené procesem i za přidávané

přidávání uzlů do pohledu symetrické

```
Trans_failure(q)
        F += q
        Last[self] = F
        Trans_send(FAULT,F)
```

detekce havárie procesu q

```
Trans_deliver(m, sender)
```

doručení zprávy

```
Trans_new(q)
Trans_deliver_join(m, sender)
```

ekvivalent failure
F/J, Last/JLast

Transis algoritmus - odebrání uzlu, doručení

```
Trans_deliver(m,sender)
    if( sender∈F && (m,FAULT(F'))∈G && sender∈F')
        discard m
    else if( F≠∅ && (m,FAULT(F'))∈G)
        blocked += m
    else if( m == FAULT(fset)) {
        Last[sender] = fset
        if( fset-F ≠ ∅) {
            deliver deliverable msgs from blocked
            F += fset
            Trans_send(FAULT,F)
        }
        if( Last[i] == Last[j] ∀i,j∈L-F) {
            deliver msgs from blocked that precede every FAULT(F)
            install L -= F
            deliver all from blocked
            F = ∅
            foreach( i∈L)
                Last[i] = ∅
        }
    } else
        deliver(m,sender)
```

zahození m v Lx+1 od havarovaného odesílatele

pozdržení zprávy do Lx+1

doručení FAULT

nově havarované procesy

zprávy patřící do Lx

potvrzení / propagace

FAULT od všech žijících

doručení Lx

instalace nového pohledu

doručení pozdržených,vyčištění

normální doručení

ISIS protokol - spolehlivé kauzální doručování

Základem doručovacích protokolů je způsob přenosu informace o doručení

- Trans - kauzální potvrzení
- ISIS - maticové hodiny

vektorové hodiny:

- $VTp_i[i]$ vlastní odeslané zprávy procesu p_i
- $VTp_i[k]$ pro $k \neq i$ zprávy přijaté od ostatních procesů

idea - každý proces zná VT[] všech procesů - udržuje matici

- $VTp_i[j][k]$ co proces p_i ví o doručení zpráv procesu p_j od p_k

při příjmu zprávy od p_i k p_j se aktualizuje $VTp_j[i]$ a $VTp_j[j]$

co ví příjemce o sobě

$VTp_j[j][i]=VTm[i]$, $VTp_j[i][*]=VTm[*]$

co ví příjemce o odesílateli

stabilní zprávy (přijaté všemi členy skupiny) - doručeny VT od všech členů skupiny

- porovnání s Trans
  - Trans potřebuje potvrdit max. M jiných zpráv - lze nahradit VT

Trans lze považovat za formu komprimace VT

- optimistický algoritmus
  - relativně nízká režie při správném přenosu, vysoká režie při chybách

ISIS protokol - členství ve skupinách

Cíl: všechny zprávy zaslané do pohledu L jsou doručeny všem žijícím procesům v L před instalací nového pohledu

Každý proces v L udržuje zprávu dokud není stabilní

Po příjmu zprávy o instalaci nového pohledu proces přepošle všechny nestabilní zprávy a potom potvrzení instalace (flush message), zatím pohled neinstaluje

Po příjmu flush od každého procesu může instalovat nový pohled

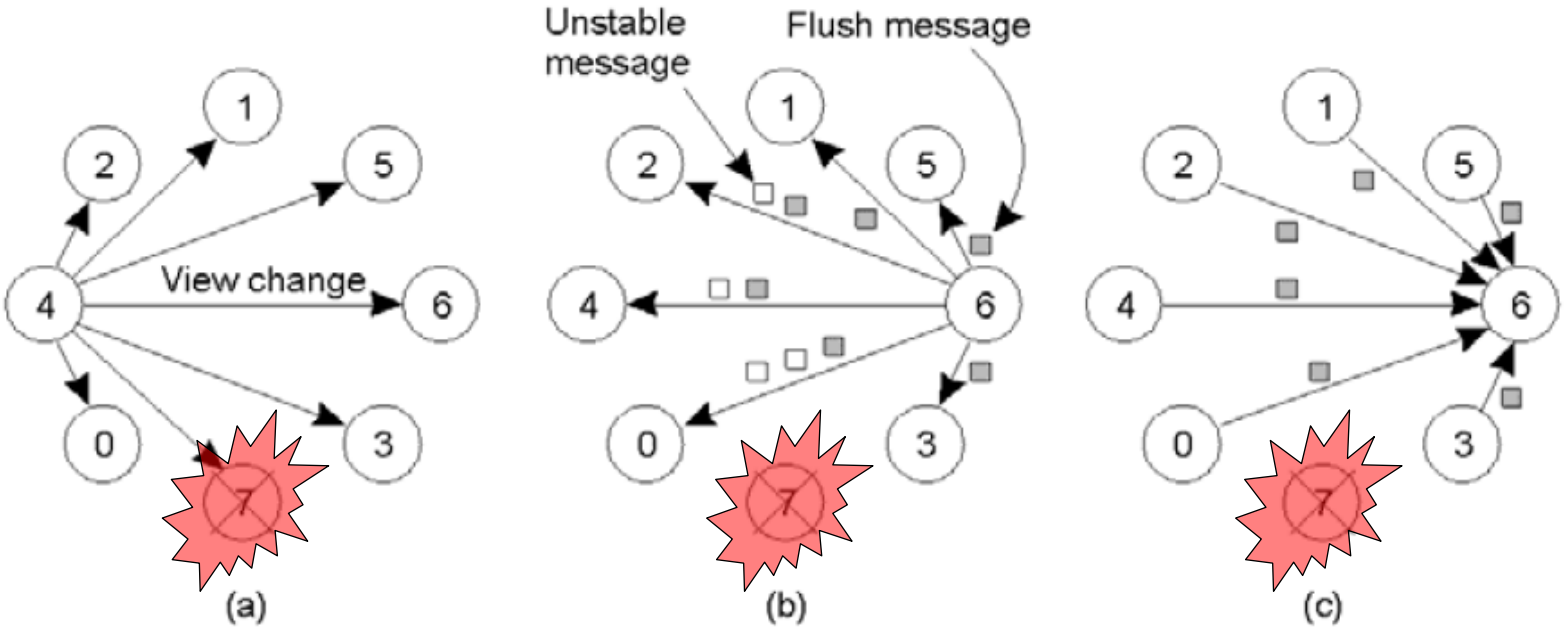
Zprávy od havarovaných procesů

- každý proces udržuje seznam 'havarovaných' procesů aktuálního pohledu
- s každou zprávou se rozesílá seznam, při příjmu se sjednotí s vlastním
- zprávy od havarovaných procesů se zahazují

Reálné nasazení

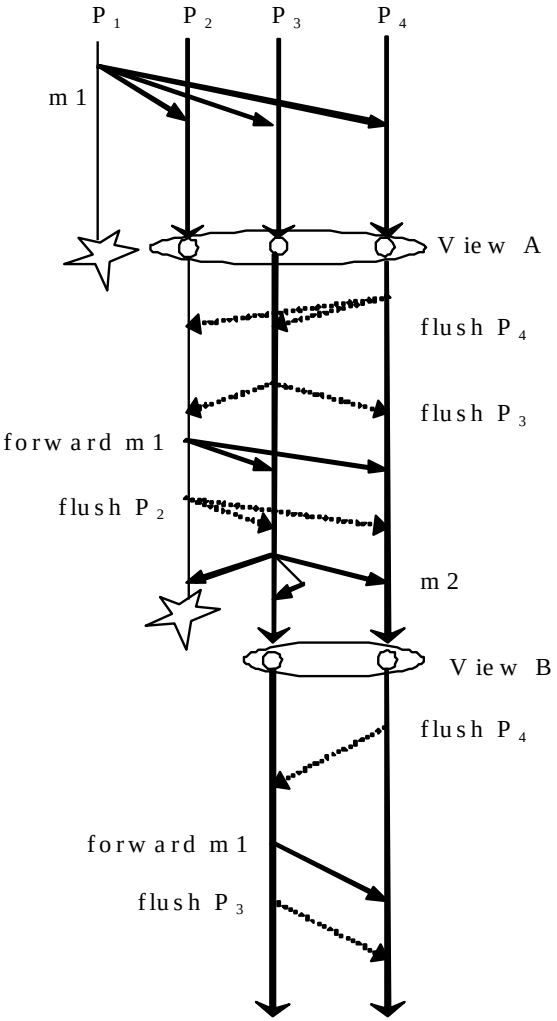
- New York Stock Exchange
- Swiss Stock Exchange
- French Air Traffic Control System

Instalace nového pohledu

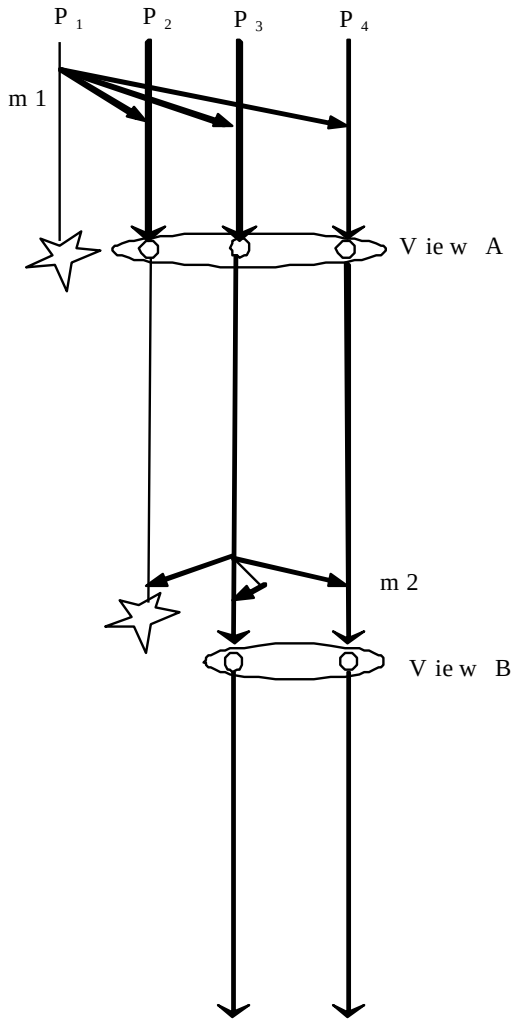


Průběh instalace nového pohledu v ISIS
a) proces 4 zjistil havárii procesu 7, rozesílá instalaci nového pohledu
b) proces 6 rozesílá nestabilní zprávy a potvrzení instalace
c) proces 6 po přijetí všech flush instaluje nový pohled

Průběh virtuální synchronie



Fyzický průběh



Virtuálně synchronní duručování

Doručovací protokoly - shrnutí

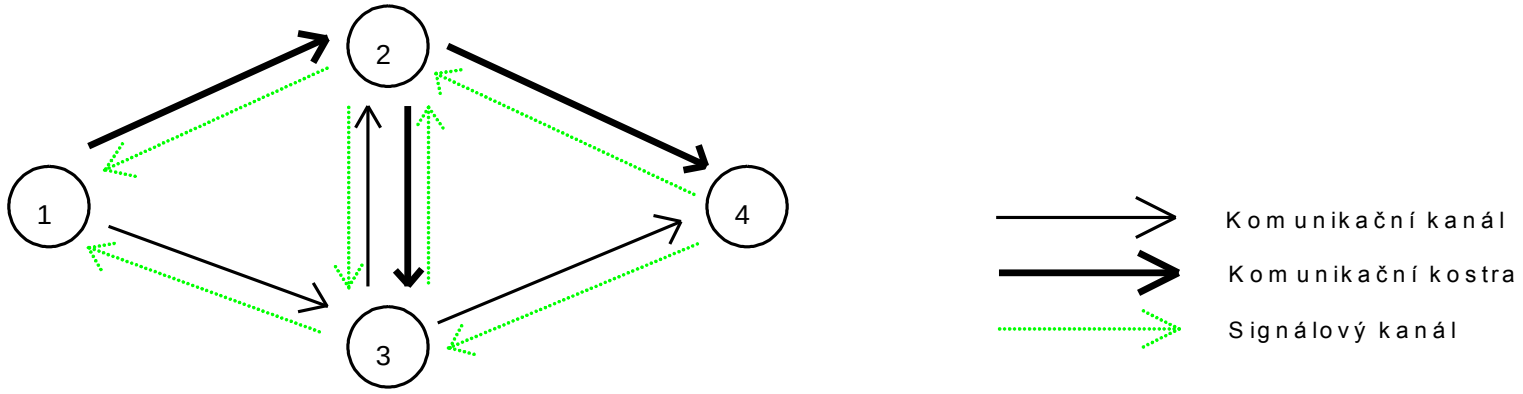
Shrnutí problémů doručovacích protokolů

principy a protokoly, které je řeší

problém	protokol
sekvenční doručování	skalární hodiny sekvencer
kauzální doručování	vektorové hodiny
překrývající se skupiny	maticové hodiny
nespolehlivá komunikace, stabilní zprávy	Trans
měnící se skupiny virtuální synchronie	Transis a Isis

Ukončení distribuovaných procesů

orientovaný graf uzlů, vstupní / výstupní kanály
signální kanály - signalizace ukončení



- Iniciační proces začíná výpočet, vyšle podél svých výstupních hran zprávu
- Proces se při příjmu zprávy dostane do stavu *výpočet*
  - může dále posílat zprávy podél výstupních hran
  - může přijímat zprávy ze vstupních hran
  - když proces skončí výpočet, dostane se do stavu *ukončen*, zprávy neposílá, může přijímat
  - při přijetí zprávy se proces zrestartuje do stavu *výpočet*

Algoritmus ukončení:

- všechny procesy jsou ve stavu *ukončen*, pak se v konečném čase toto všechny procesy dozví
- stačí, když se to dozví iniciační uzel (může ostatním uzlům poslat zprávu)

Dijkstra-Scholten (DS) algoritmus

Strom

- každý listový proces při přechodu do stavu *ukončen* pošle signál svému otci
- proces dostane signály od všech svých synů, pošle signál svému otci
- iniciační proces dostane všechny signály - konec výpočtu

Acyklický orientovaný graf (DAG)

- s každou hranou je asociován čítač - tzv. *deficit*
  - rozdíl mezi počtem zpráv došlých tímto datovým kanálem a počtem signálů poslaných signálním kanálem zpět
- končící proces vyšle každým signálním kanálem tolik signálů, aby deficit = 0

Obecný (orientovaný) graf

- problém: neexistují listy, které by mohly rozhodnout o ukončení výpočtu
- řešení: výpočet vytvoří dynamicky kostru grafu
  - otec = uzel, od kterého přišla první zpráva
- algoritmus ukončení pro jeden proces:
  1. Poslat signál podél všech vstupních hran kromě hrany k otci
  2. Čekat na signál od všech výstupních hran
  3. Poslat signál otci
- iniciační proces dostane všechny signály - konec výpočtu

Značkový (TM) algoritmus

Značka - speciální zpráva odlišitelná od všech ostatních

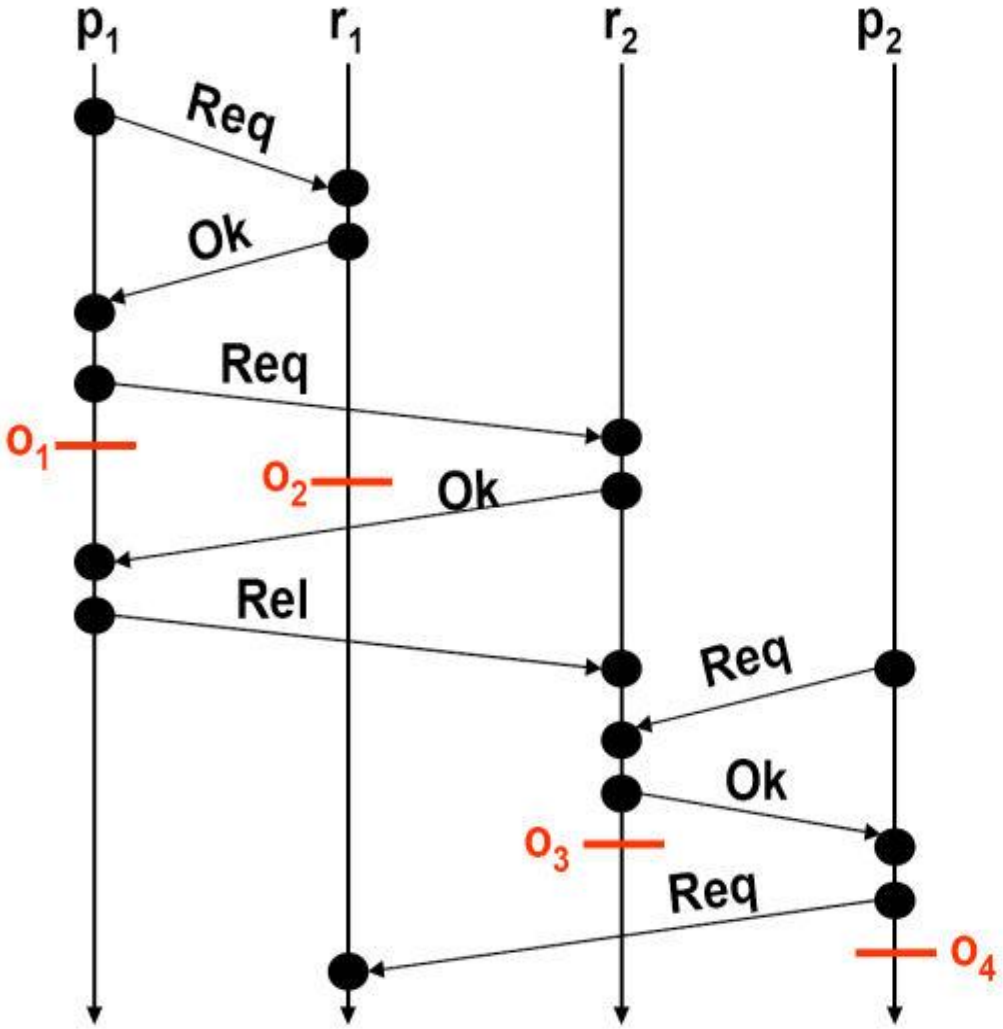
Iniciační uzel vyšle žádost o ukončení (značku, že je kanál prázdný) všem výstupním hranám

- Příjem první značky:
  - ♦ připraven - propagace značky výstupním hranám
  - ♦ nepřipraven - negativní signál (zamítnutí)
- Příjem další značky: signál / zamítnutí
- Příjem zamítnutí: zamítnutí první žádosti
- Příjem všech potvrzení: signál první žádosti

Speciální případ obecnějšího algoritmu - detekce globálního stavu

Detekce globálního stavu - motivace

Detekce falešného deadlocku



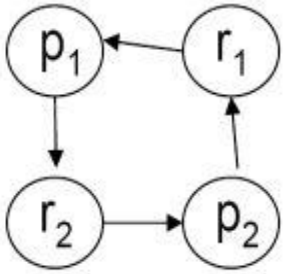
Inference from

o_1: p_1 waits for r_2

o_2: r_1 waits for p_1

o_3: r_2 waits for p_2

o_4: p_2 waits for r_1



From $O=\{o_1,o_2,o_3,o_4\}$ the deadlock detector concludes there is a deadlock!

Konzistentní stav

Množina událostí v systému E = {e}

Řez c je disjunktní rozdělení E na P_c a F_c : $P_c \cup F_c = E \;\&\; P_c \cap F_c = \varnothing$

Konzistentní řez c : $a \rightarrow b \;\&\; a \in F_c \Rightarrow b \in F_c$

Stav (distribuovaného) procesu je množina událostí, které se v procesu udály

Konzistentní stav *S* = P_c, kde *c* je konzistentní řez

S konzistentní stav, e: S' = S $\cup$ e je konzistentní stav:

$S \rightarrow^e S'$ (S' je **dosažitelný** z S)

Posloupnost událostí s = (e_1, e_2, ..., e_n) se nazývá **rozvrh** (schedule), jestliže

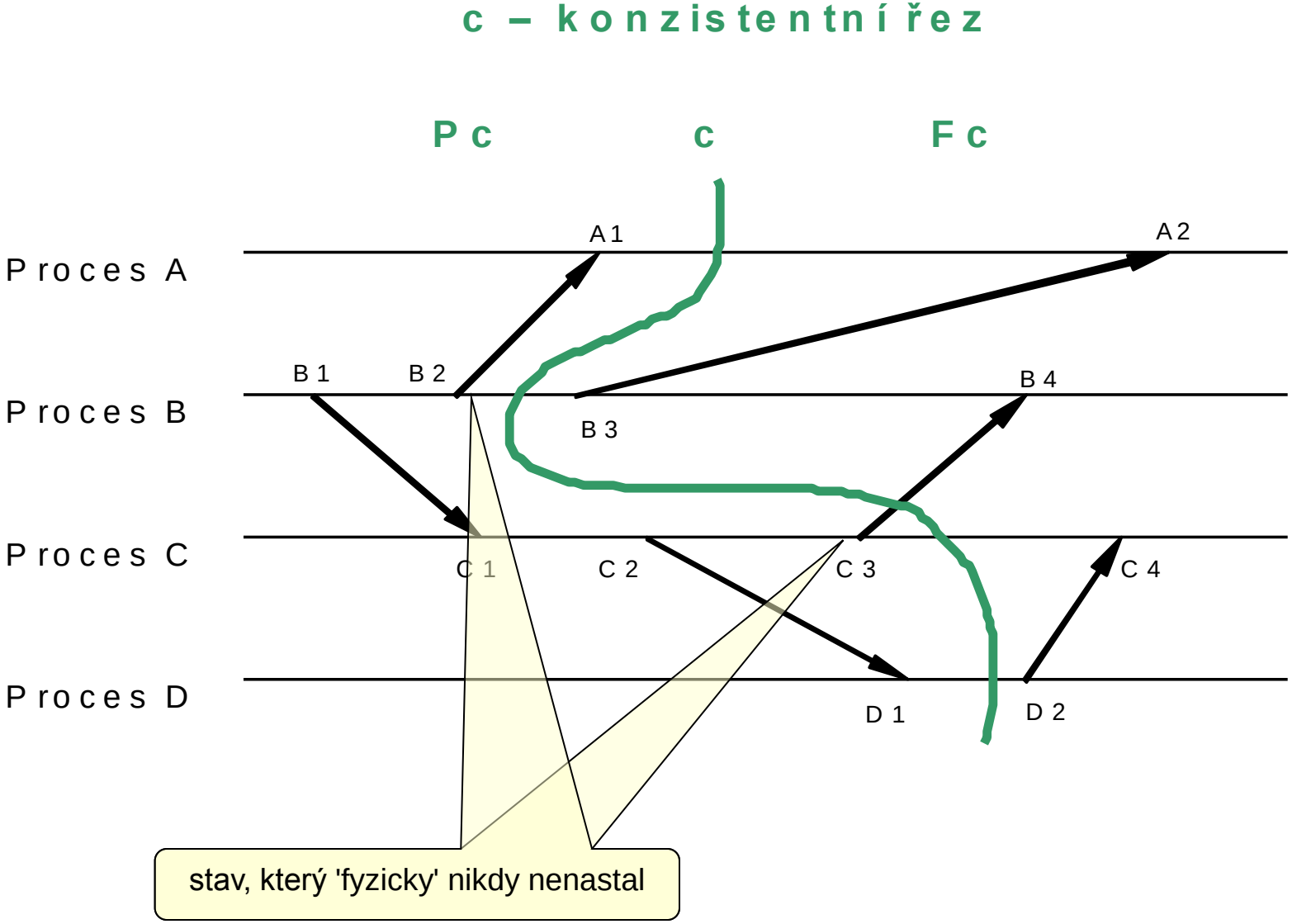
$S \rightarrow^{e1} S_1 \rightarrow^{e2} S_2 \ldots S_{n-1} \rightarrow^{en} S_n$ Značíme $S \rightarrow^s S_n$

Zřejmě platí $S \rightarrow^s S_n \Rightarrow S \subseteq S_n$

Využití:

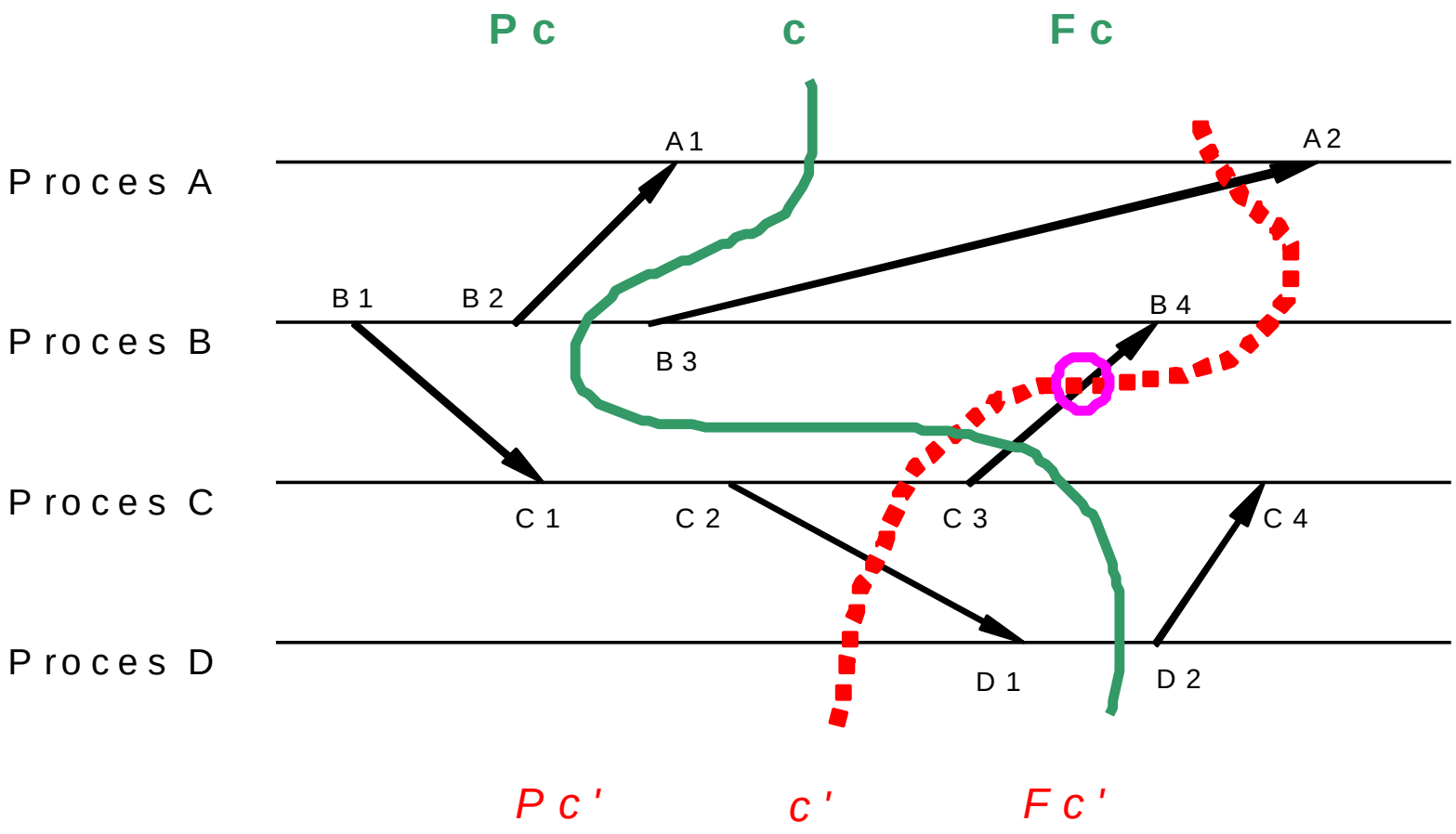
- d. detekce deadlocků
- d. garbage collection
- detekce ukončení d. výpočtů
- obecně detekce globálních vlastností

Konzistentní a nekonzistentní řez



Konzistentní a nekonzistentní řez

c – konzistentní řez



c' – nekonzistentní řez

Algoritmus detekce globálního stavu

stav uzlu: množina přijatých a odeslaných zpráv

stav kanálu: množina zpráv, které byly kanálem odeslány, ale ještě nebyly doručeny

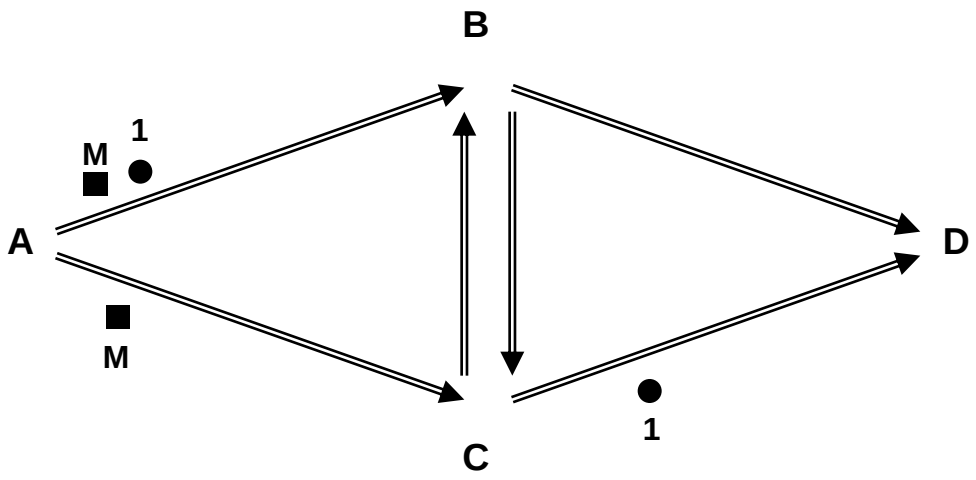
Algoritmus:

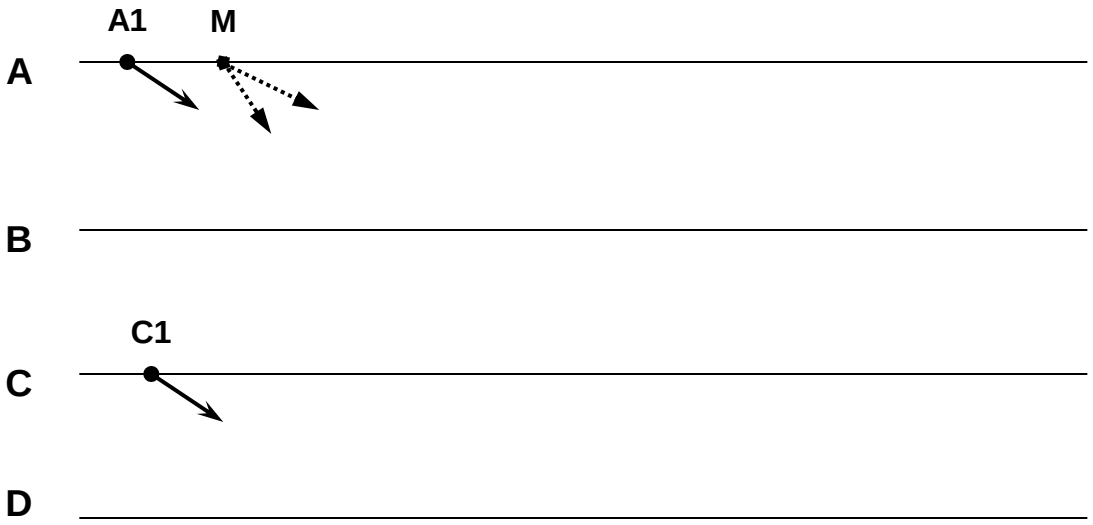
- Iniciátor vyšle všem výstupním uzlům značku
- Příjem první značky:
  - Uzel si zapamatuje poslední přijaté a odeslané zprávy = stav uzlu
  - Stav všech příchozích kanálů označí za prázdný
  - Vyšle všem výstupním uzlům značku
- Příjem zpráv od uzlů, od kterých ještě nepřišla značka:
  - Zapamatuje si čísla zpráv
- Příjem značek od dalšího uzlu:
  - Stav kanálu = zaznamenané zprávy došlé od uzlu první a touto značkou
- Konec algoritmu:
  - po přijmutí všech značek

Zaznamenané stavy uzlů a kanálů definují konzistentní stav systému

Chandy, Lamport (1985)

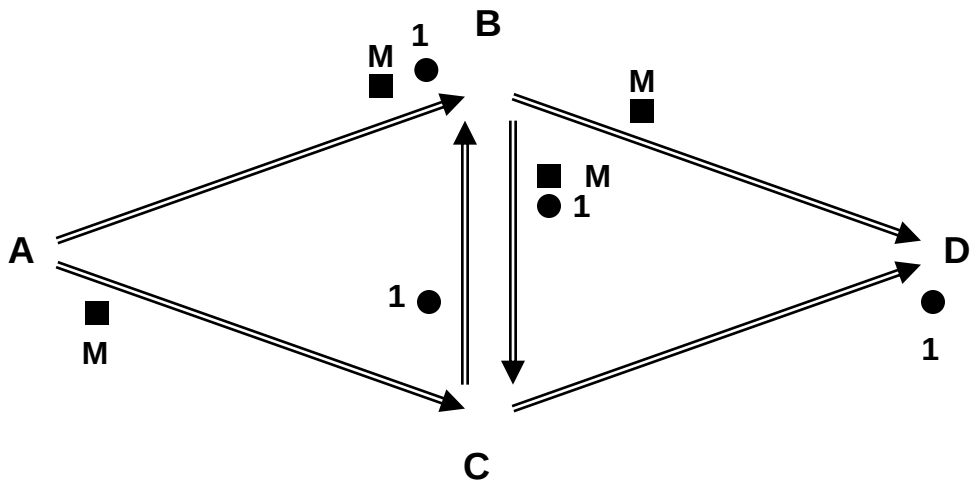
Průběh detekce GS 1

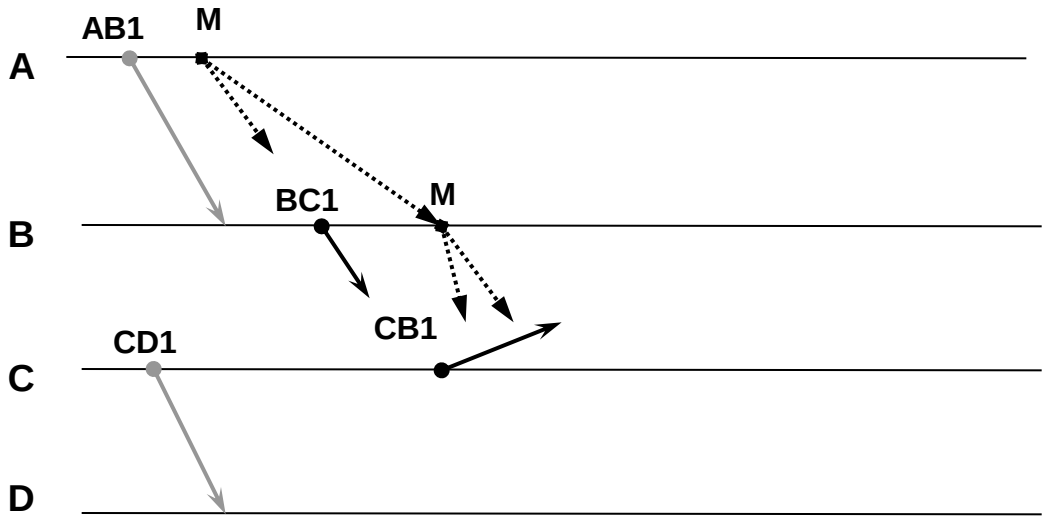




A	A→B	1
	A→C	0
B	A→B	
	C→B	
	B→C	
	B→D	
C	A→C	
	B→C	
	C→B	
	C→D	
D	B→D	
	C→D	

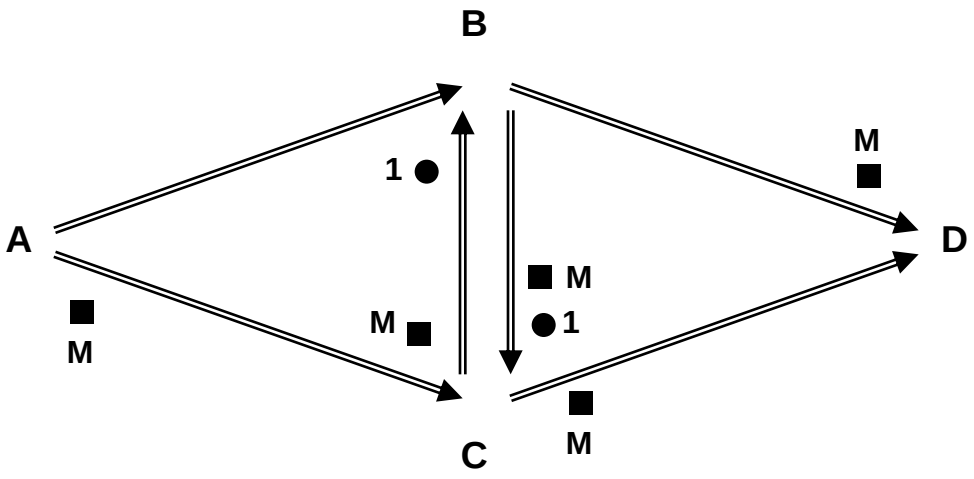
Průběh detekce GS 2

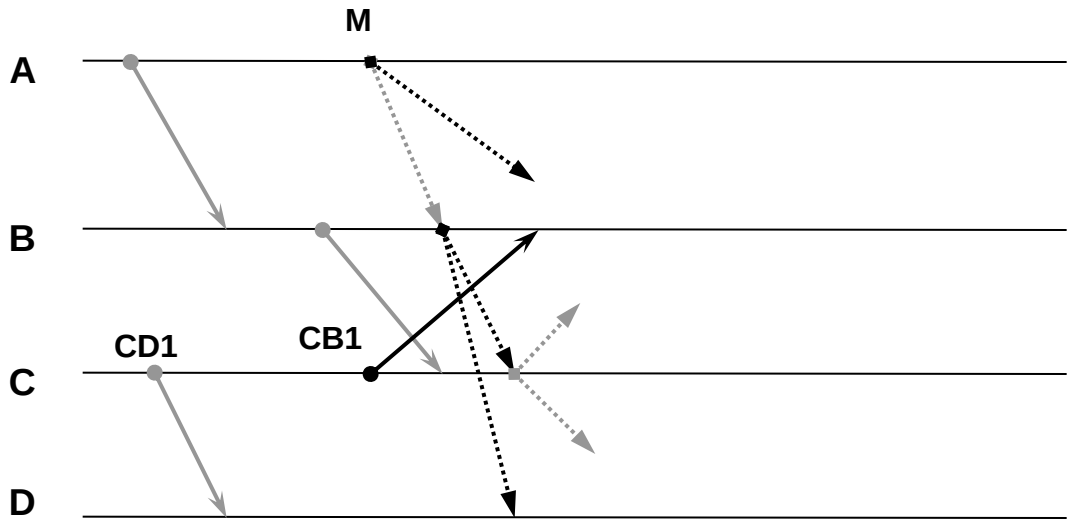




A	A→B	1
	A→C	0
B	A→B	◇
	C→B	
	B→C	1
	B→D	0
C	A→C	
	B→C	
	C→B	
	C→D	
D	B→D	
	C→D	

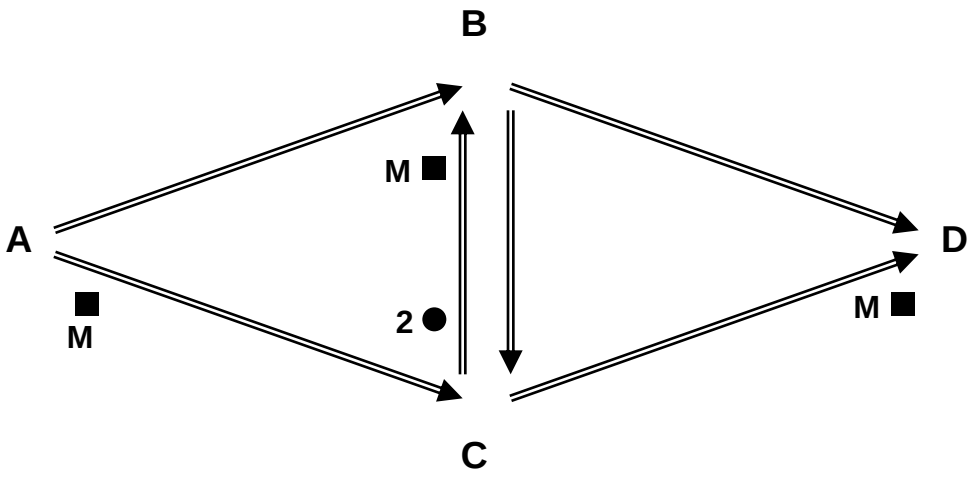
Průběh detekce GS 3

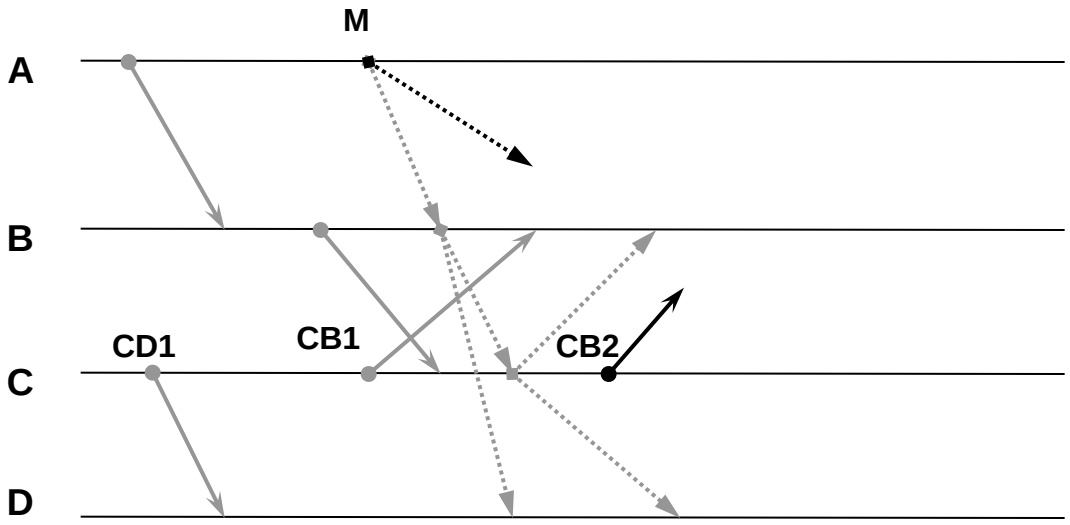




A	A→B	1
	A→C	0
B	A→B	◇
	C→B	1
	B→C	1
	B→D	0
C	A→C	
	B→C	◇
	C→B	1
	C→D	1
D	B→D	◇
	C→D	

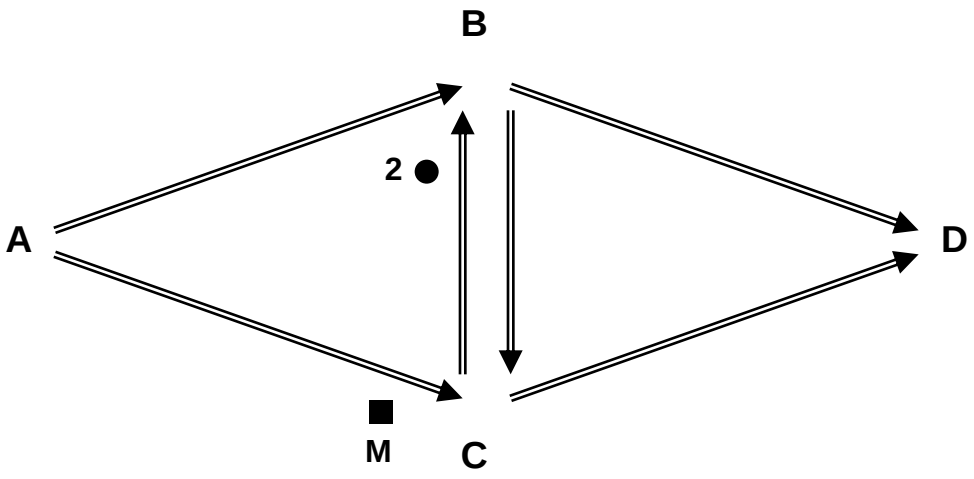
Průběh detekce GS 4

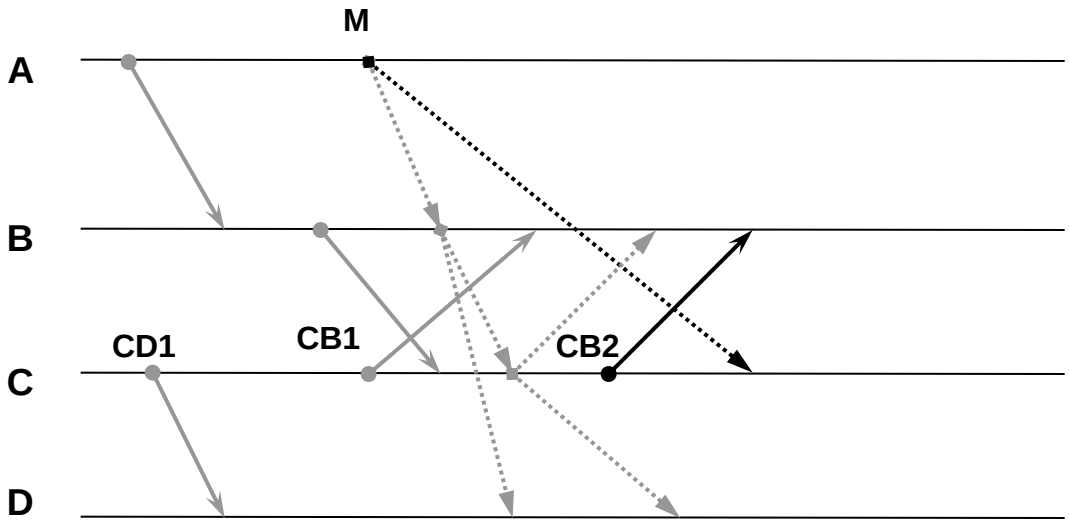




A	A→B	1
	A→C	0
B	A→B	◇
	C→B	◇1
	B→C	1
	B→D	0
C	A→C	
	B→C	◇
	C→B	1
	C→D	1
D	B→D	◇
	C→D	◇

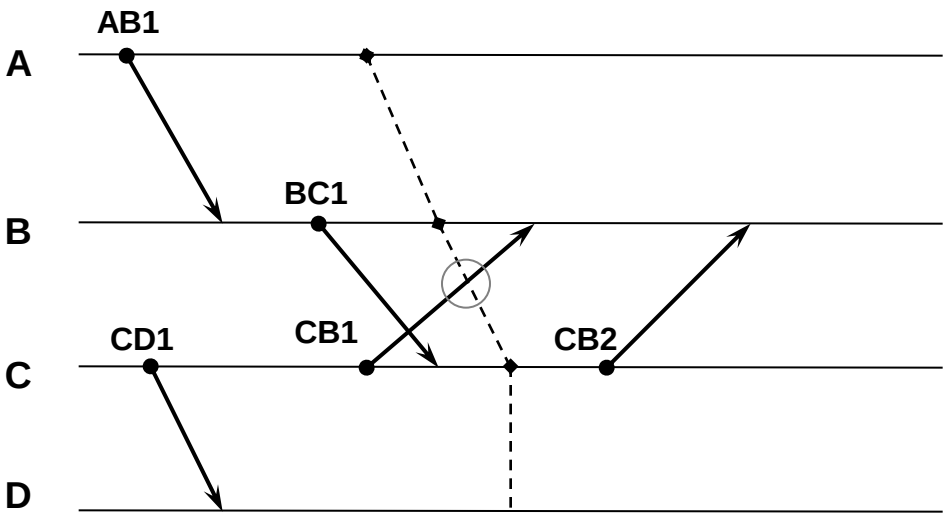
Průběh detekce GS 5





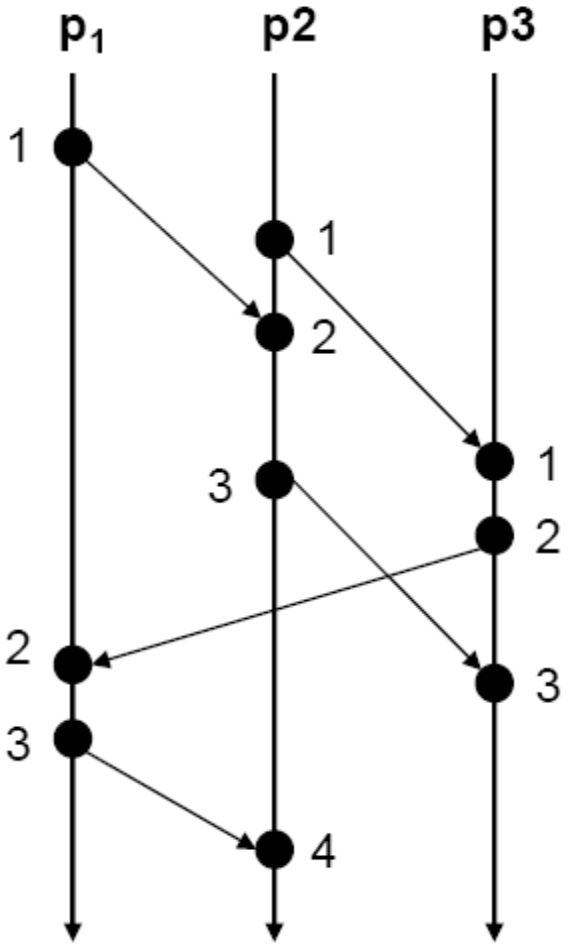
A	A→B	1
	A→C	0
B	A→B	◇
	C→B	◇1
	B→C	1
	B→D	0
C	A→C	◇
	B→C	◇
	C→B	1
	C→D	1
D	B→D	◇
	C→D	◇

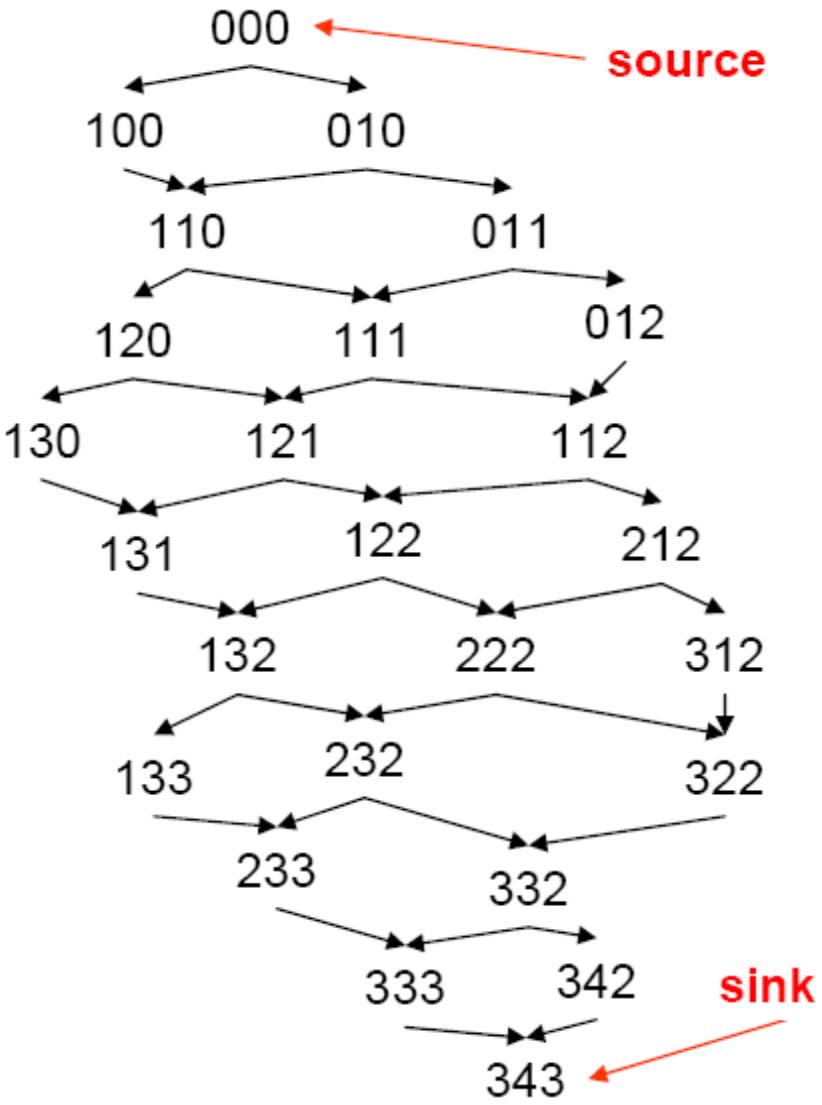
Průběh detekce GS - výsledek



A	A→B	1
	A→C	0
B	A→B	◇
	C→B	◇1
	B→C	1
	B→D	0
C	A→C	◇
	B→C	◇
	C→B	1
	C→D	1
D	B→D	◇
	C→D	◇

Modely distribuovaných výpočtů





globální stavy, globální predikáty

stabilní vlastnosti, důkazy korektnosti

Distribuovaný konsensus

Problém dvou armád

- Početnější armáda B je rozdělena
- Úspěch pouze při synchronizovaném útoku, jinak drtivá porážka
- Obě části musí mít jistotu, že druhá část začne útok také
- Komunikace pouze nespolehlivým kurýrem - zajetí

Striktní řešení NEEXISTUJE !!!

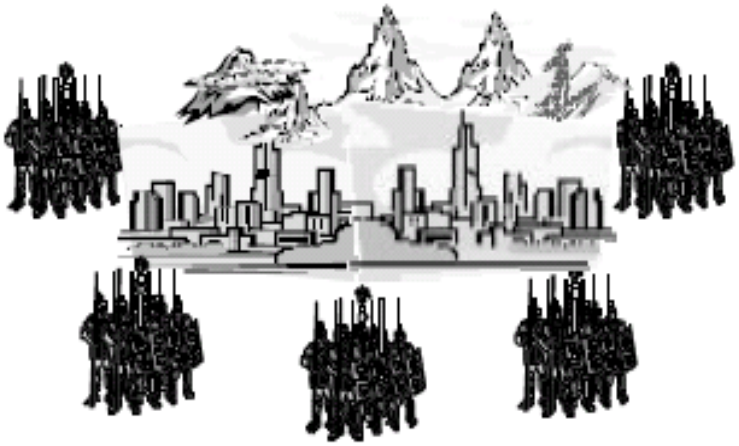
A1->A2:	Attack!	*(A1 neví, jestli zprávu A2 dostal)*
A2->A1:	ACK	*(A2 neví, jestli A1 dostal potvrzení)*
A1->A2:	ACK	*(A1 neví, jestli A2 dostal potvrzení)*
A2->A1:	ACK	*(A2 neví, jestli A1 dostal potvrzení)*
A1->A2:	ACK	*(A1 neví, jestli A2 dostal potvrzení)*
A2->A1:	ACK	*(A2 neví, jestli A1 dostal potvrzení)*
A1->A2:	ACK	*(A1 neví, jestli A2 dostal potvrzení)*
.....	.....	.....
...	....	.....

Problém dvou armád

Praktická řešení

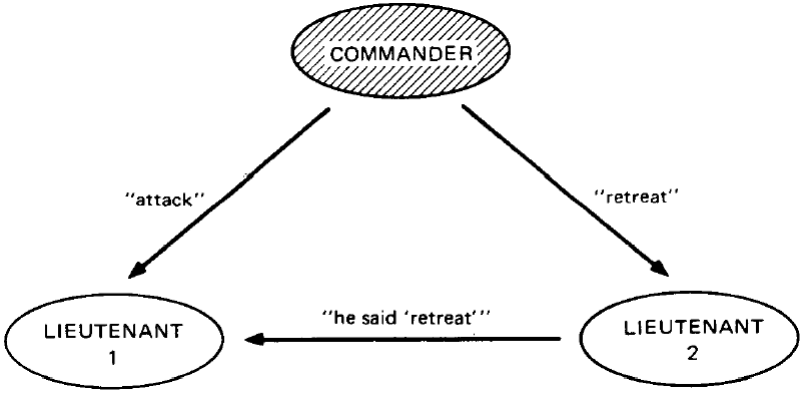
- Agresivní strategie
  - první generál vyšle větší množství zpráv oznamující čas útoku
  - a zaútočí tak jako tak
  - předpokládá se, že při vysokém počtu poslaných zpráv alespoň jedna projde
  - druhý generál zde ani nemusí odpovídat.
- Pravděpodobnostní strategie
  - první generál vyšle větší množství (N_0) zpráv
  - kromě času útoku přidá i to, kolik jich poslal
  - druhý generál odpoví na každou, kterou obdržel (N_1)
  - počet odpovědí vrácených prvnímu generálovi je N_2
  - oba generálové znají přibližnou míru úspěšnosti, že posel zprávu pronese

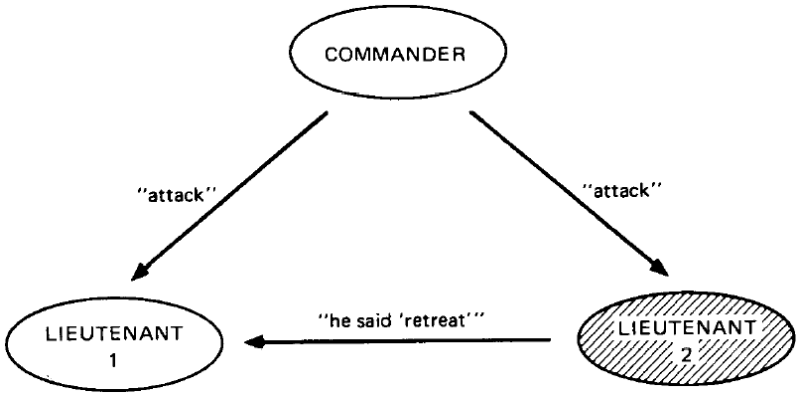
Problém Byzantských generálů



- Předpoklady:
  - uzly mohou libovolně havarovat
  - havarovaný uzel se může chovat zákeřně!
  - spolehlivá komunikace (časově neohraničená)
- Úkol:
  - někteří generálové jsou zrádci
  - všichni loajální generálové se musejí rozhodnout shodně
  - každý generál se rozhoduje na základě informací obdržených od ostatních generálů
- BÚNO: 1 generál, ostatní důstojnící
  - jak generál tak důstojníci mohou být zrádci
  - generál vydá rozkaz, důstojnící ho předají ostatním
  - rozkaz bude vydán na základě většiny
- Cíl:
  - C1: všichni loajální důstojníci vydají stejný rozkaz
  - C2: Je-li generál loajální, pak každý loajální důstojník vydá rozkaz generála

Problém Byzantských generálů - 3 uzly



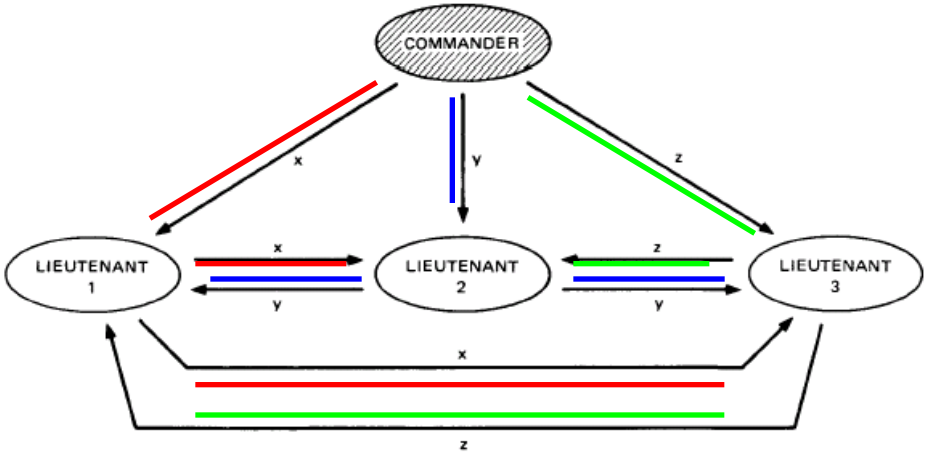


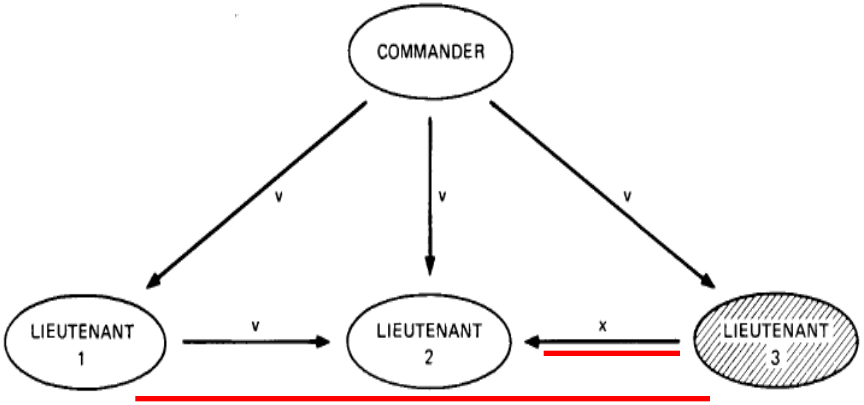
- Zrádce generál
  - různé rozkazy důstojníkům
  - důstojník 1 dostane 2 různé rozkazy
- Zrádce důstojník
  - falešné předání rozkazu
  - důstojník 1 dostane 2 různé rozkazy

Řešení pro 3 uzly s 1 zrádcem neexistuje

Obecně: Pro m zrádců, pak neexistuje řešení pro $n \leq 3m$ uzlů

Problém Byzantských generálů - 4 uzly





- Zrádce generál
  - ♦ alespoň 2 stejné rozkazy
    - ● důstojníci si vzájemně přepošlou většinový rozkaz (C2)
  - ♦ 3 různé rozkazy
    - ● důstojníci se shodnou na tom, že generál je zrádce (C1)
- Zrádce důstojník
  - ♦ nejhorší případ: falešné předání rozkazu všem ostatním
  - ♦ loajální důstojníci dostanou většinu správných rozkazů (C2)

👍 Řešení pro 4 uzly s 1 zrádcem existuje

Obecně: Pro m zrádců exsituje řešení pro n ≥ 3m+1 uzlů

Lamport, Shostak, Paese: The Byzantine Generals Problem, ACM '82

Klasifikace problémů distribuovaného konsensu

- Byzantský konsensus *(Byzantine agreement)* — $1 \rightarrow 1$
  - inicátor zvolí hodnotu, rozešle ji všem uzlům
  - všechny loajální uzly se musí shodnout na stejné hodnotě
  - pokud je iniciátor loajální, hodnota se musí shodovat s iniciální
- Konsensus — $n \rightarrow 1$
  - každý uzel má iniciální hodnotu
  - všechny loajální uzly se musí shodnout na společné hodnotě
  - pokud je iniciální hodnota všech loajálních uzlů stejná, musí se shodnout na této hodnotě
- Interaktivní konzistence — $n \rightarrow n$
  - každý uzel má iniciální hodnotu
  - všechny loajální uzly se musí shodnout na společném vektoru
  - hodnota položek vektoru odpovídajících loajálním uzlům se musí shodovat s jejich iniciální hodnotou

$BK \cong K \cong IK$

Synchronizace v DS - shrnutí

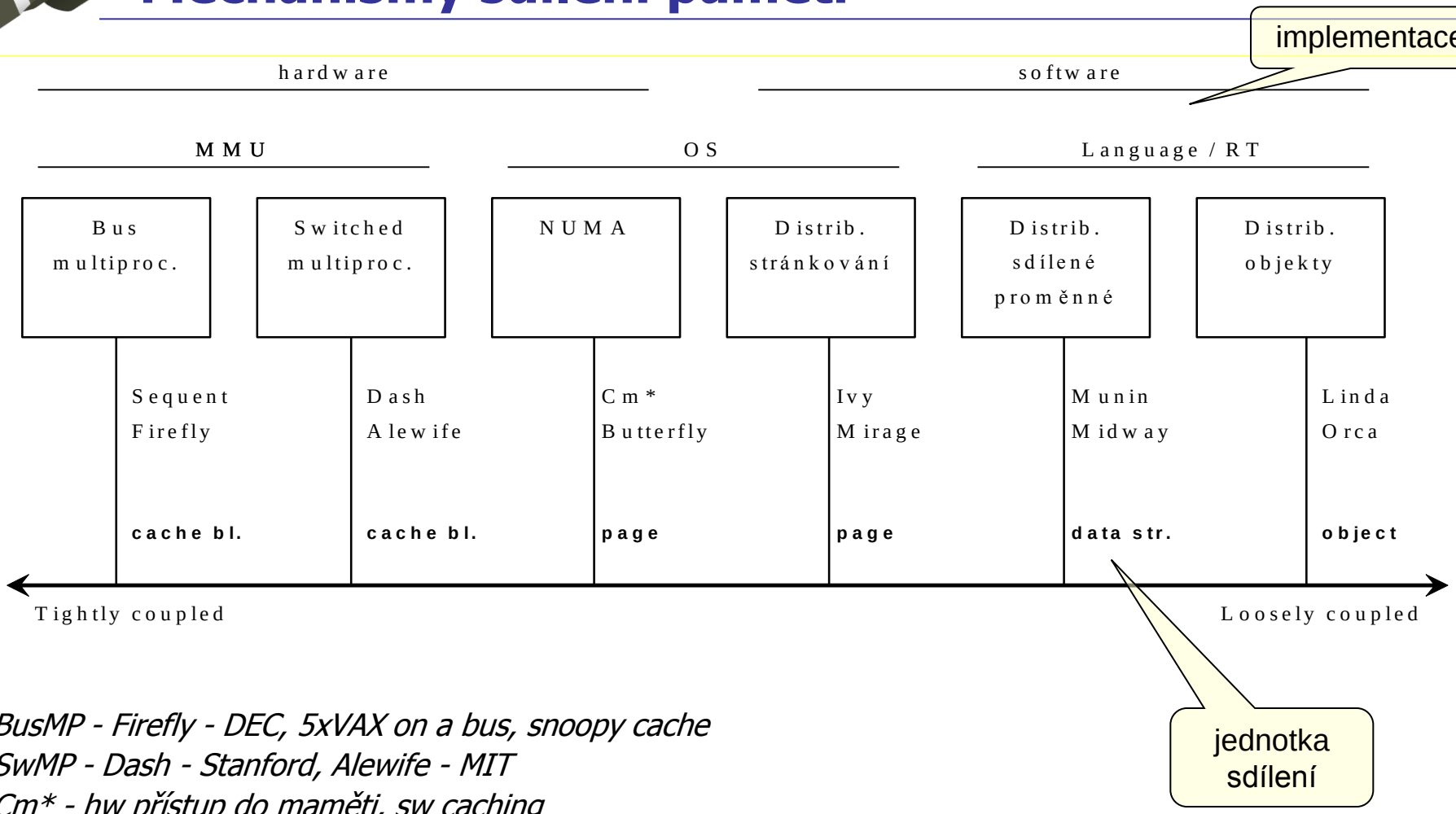
- fyzické hodiny
  - synchronizace vůči přesnému zdroji nebo navzájem, úprava rychlosti tiků
- logické hodiny
  - kauzalita událostí, neexistence globálního času, relace 'předchází'
- distribuované vyloučení procesů
  - absence sdílené paměti
  - centralizované, časové značky, voting, token passing
- volba koordinátora
- doručovací protokoly
  - sekvenční/kauzální doručování, spolehlivé doručování
  - virtuální synchronie, změny členství
- detekce globálního stavu
  - konzistentní řez, značkový algoritmus
- konsensus
  - dvě armády, problém byzantských generálů

Distribuovaná sdílená paměť

- Paralelní výpočty
  - multiprocesory
    - malý počet procesorů
    - příliš komplikované a drahé
  - multicomputery
    - snadno dostupné
    - programování a synchronizace není prakticky (inženýrsky) dobře zvládnutá
    - RPC – problémy
    - Pokus o řešení – distribuovaná sdílená paměť
    - 1986 Li & Hudak – DSM - množina uzlů sdílející adresový prostor

- Distribuovaná sdílená paměť (DSM)
  - Konzistenční modely
  - Distribuované stránkování
  - Distribuované sdílené proměnné a objekty

Mechanismy sdílení paměti



BusMP - Firefly - DEC, 5xVAX on a bus, snoopy cache
SwMP - Dash - Stanford, Alewife - MIT
Cm - hw přístup do maměti, sw caching*

Distributed paging - Ivy, Li ... T4
Mirage - Bershad 1990
Munin - Benett 1990

Konzistenční modely DSM

- Konzistenční model
  - Specifikace co implementace musí splňovat vzhledem k operacím čtení a zápis

Značení:

W(x)a	zápis hodnoty a do proměnné x
R(x)a	při čtení z proměnné x je vrácena hodnota a
S	synchronizace
Acq	vstup do kritické sekce (Acquire)
Rel	výstup z kritické sekce (Release)
Pi	proces i

Striktní konzistence

Strict consistency, atomic consistency

Jakékoliv čtení z paměti z adresy x vrátí hodnotu uloženou při posledním zápisu na adresu x.

- zajišťuje absolutní časové uspořádání
- nejsilnější, na jednoprocesorových systémech je tradičně zajištěna
- všechny zápisy **okamžitě** všude viditelné
- podmínka: musí existovat přesný globální čas
- ideální pro programování ☺, v distribuovaném systému nedosažitelné ☹

```
a=1; a=2; print(a);
```

vytiskne vždy 2

P1: W(x)1
P2: R(x)1

striktní konzistence

P1: W(x)1
P2: R(x)0 R(x)1

paměť, která není striktně konzistentní

Sekvenční konzistence

Sequential consistency

Výsledek výpočtu je stejný, jako kdyby všechny operace všech procesorů byly vykonávány v nějakém sekvenčním uspořádání a operace každého procesoru jsou vykonávány v pořadí specifikovaném programem.

- o něco slabší model
  - snadno implementovatelná
  - příjemná pro programování
  - povoleno libovolné prokládání instrukcí na různých procesorech
  - **všechny** procesy vidí **stejné** pořadí změn paměti
  - změny nejsou propagovány okamžitě, není zaručena velikost zpoždění (sec, min)

P1: W(x)1
P2: R(x)1 R(x)1

P1: W(x)1
P2: R(x)0 R(x)1

dvakrát spuštěný tentýž program – může dát různé výsledky

Sekvenční konzistence

```
P1:          P2:          P3:
a=1;         b=1;         c=1;
print(b,c);  print(a,c);  print(a,b);
```

šest instrukcí - 6!=720 možných uspořádání

pouze 3*(5!/4)=90 nenarušuje konzistenci

```
a=1;         a=1;         b=1;         b=1;
print(b,c);  b=1;         c=1;         a=1;
b=1;         print(a,c);  print(a,b);   c=1;
print(a,c);  print(b,c);  print(a,c);  print(a,c);
c=1;         c=1;         a=1;         print(b,c);
print(a,b);  print(a,b);  print(b,c);  print(a,b);
```

Výstup: (skutečný výstup, např. na obrazovce)

```
001011       101011       010111       111111
```

Signatura: (výstupy jednotlivých procesů v pevném pořadí)

```
001011       101011       110101       111111
```

Sekvenční konzistence

```
a=1;            b=1;            c=1;
print(b,c);     print(a,c);     print(a,b);
```

- signatura
  - spojení výstupů procesů v pevném (předem daném) pořadí
  - konkrétní proložení instrukcí procesů, a tím i pořadí paměťových referencí
  - celkem 2^6=64 možných signatur
  - ne všechny odpovídají sekvenční konzistenci
    - např. 000000, 001001 neodpovídají žádnému sekvenčnímu proložení instrukcí

P1: W(a)1 R(b)0 R(c)0
P2: W(b)1 R(a)1 R(c)0
P3: W(c)1 **R(a)0** R(b)1

signatura 001001 je nedosažitelná v sekvenčně konzistenčním modelu

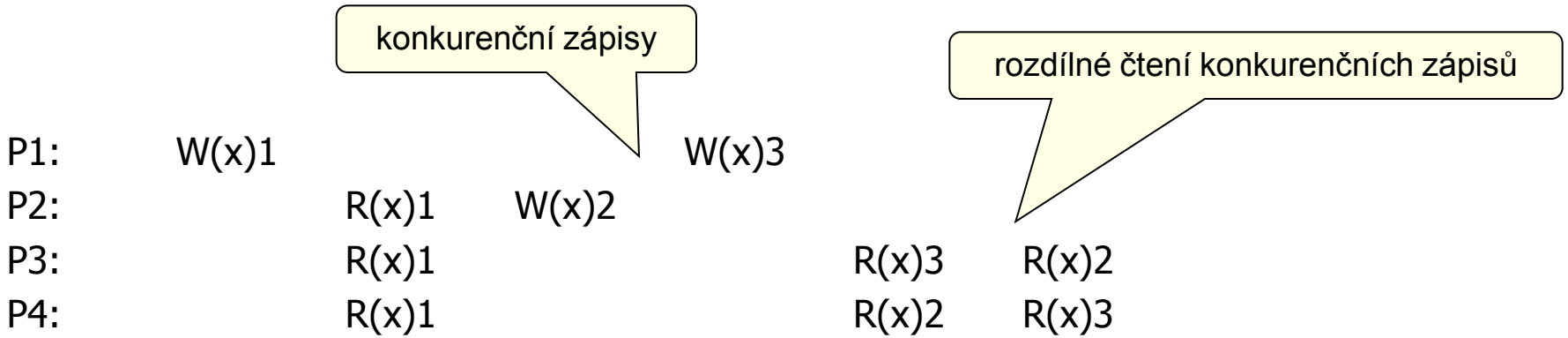
- Sekvenční konzistence
  - příjemný model pro programování
  - výkonnost není příliš velká, dokázáno (Lipton & Sandberg, 1988):
    - čas čtení *r*, čas zápisu *w* a čas přenosu zprávy (paketu) *t* : $r + w \geq t$
    - optimalizace protokolu pro čtení znamená delší čas pro zápis a naopak

Kauzální konzistence

Causal consistency

Kauzálně vázané zápisy musí být viděny všemi procesy ve stejném pořadí.
Konkurenční zápisy mohou být viděny v různém pořadí.

- rozlišuje události, které jsou potenciálně závislé a které ne
- kauzálně závislé zápisy
  - pokud W(x)a a (druhý proces) R(x) W(y)b, pak W(y)b je kauzálně závislý na W(x)a
- analogie k zasílání zpráv: zápis ≈ odeslání, čtení ≈ přijetí



splňuje podmínku kauzální konzistence, nesplňuje sekvenční

Kauzální konzistence

P1: W(x)1

P2: W(x)2

P3: R(x)2 R(x)1

P4: R(x)1 R(x)2

kauzálně konzistentní rozvrh

porušení kauzální konzistence
(přidaná operace čtení)

P1: W(x)1

P2: R(x)1 W(x)2

P3: R(x)2 R(x)1

P4: R(x)1 R(x)2

není kauzálně konzistentní

- implementace složitější
  - vyžaduje vyžaduje udržování grafu závislostí zápisů na čtení

PRAM konzistence

PRAM (Pipelined RAM) consistency

Zápisy prováděné jedním procesem jsou viděny ostatními procesy v tom pořadí, ve kterém byly prováděny.
Zápisy různých procesů mohou být viděny různými procesy různě.

- srovnání se sekvenční konzistencí:
  - v PRAM neexistuje jednotný pohled na rozvrh
- snadná na implementaci
  - nezáleží na pořadí, v němž různé procesy vidí přístupy k paměti
  - je nutné dodržet pořadí zápisů z jednoho zdroje

P1 přečte b dříve než uvidí jeho zápis
P2 přečte a dříve než uvidí jeho zápis

není možné při libovolném globálním uspořádání instrukcí

```
P1:
a=1;
if(b==0) kill(P2);
```

```
P2:
b=1;
if(a==0) kill(P1);
```

možný výsledek: oba procesy budou zabity

PRAM konzistence

P1:	***P2:***	***P3:***
`a=1;`	`b=1;`	`c=1;`
`print(b,c);`	`print(a,c);`	`print(a,b);`
`a = 1;`	`a = 1;`	`b = 1;`
`print(b,c);`	`b = 1;`	`print(a,c);`
`b = 1;`	**`print(a,c);`**	`c = 1;`
`print(a,c);`	`print(b,c);`	**`print(a,b);`**
`c = 1;`	`c = 1;`	`a = 1;`
`print(z,b);`	`print(a,b);`	`print(b,c);`

Výstup:

00	**10**	**01**

PRAM-konzistentní lokální rozvrhy

Slow memory

Zápisy jedním procesem do jednoho místa musí být viděny ve stejném pořadí

Nejslabší model - lokální zápis, pomalá nesynchronizovaná propagace
Neposkytuje žádnou synchronizaci

P1: W(x)1 W(x)2 W(y)3 W(x)4
P2: R(y)3 **R(x)1** R(x)2 R(x)4

zápis do x před zápisem do y ještě není propagován

Ještě slabší modely?
Casual Consistency (Bednárek 2009)
Příležitostná konzistence - když je příležitost, tak to je konzistentní ☺

Konzistenční modely se synchronizační proměnnou

- Doposud uvedené konzistenční modely velmi restriktivní
  - vyžadují propagaci všech zápisů všem procesům
  - málo efektivní
  - ne všechny aplikace vyžadují sledování všech zápisů, natož pak jejich pořadí
  - typická situace:
    - proces v kritické sekci ve smyčce čte a zapisuje data
    - ostatní procesy nemusí jednotlivé zápisy vidět, není nutné, aby byly propagovány
  - paměť ale neví, že proces je v kritické sekci, musí propagovat všechny zápisy

- Řešení: nechat proces ukončit kritickou sekci a poté rozeslat změny ostatním
- Speciální druh proměnné - **synchronizační proměnná**
  - použití pouze pro synchronizační účely

S	synchronizace
Acq	vstup do kritické sekce (Acquire)
Rel	výstup z kritické sekce (Release)

Slabá konzistence

Weak consistency

1. Přístup k synchronizačním proměnným je sekvenčně konzistentní.
2. Přístup k SP není povolen, dokud neskončí všechny předchozí zápisy.
3. Přístup k datům není povolen, dokud nebyly dokončeny všechny předchozí přístupy k SP.

- zavedení synchronizační proměnné (synchronizační operace)
- bod 1: všechny procesy vidí všechny přístupy k SP ve stejném pořadí
- bod 2: před přístupem k SP budou dokončeny všechny předchozí zápisy
- bod 3: při přístupu k obyčejným proměnným jsou dokončeny všechny předchozí přístupy k SP
- provedením synchronizace před čtením se zajistí aktuální verze dat

Slabá konzistence

P1: W(x)1 W(x)2 **S**
P2: R(x)2 R(x)1 **S**
P3: R(x)1 R(x)2 **S**

slabě konzistentní rozvrh

P1: W(x)1 W(x)2 **S**
P2: **S R(x)1**

po synchronizaci musí být data aktuální

porušení slabé konzistence

- ☺ odpadá nutnost propagací všech zápisů
- ☹ paměť při přístupu k SP nerozezná vstup a výstup z kritické sekce
  - pokaždé musí vykonat akce potřebné pro oba případy
- řešení: rozlišení vstupu (Acq) a výstupu (Rel) z kritické sekce

Výstupní konzistence

Release consistency

1. Před přístupem ke sdílené proměnné musí být úspěšně ukončeny předchozí Acq() procesu.
2. Před provedením Rel() musí být ukončeny všechny předchozí zápisy i čtení prováděné procesem.
3. Acq() a Rel() musí být PRAM konzistentní.

- po Acq() jsou všechny lokální kopie aktuální
- po Rel() jsou propagovány změny ostatním procesům
- při správném párování Acq() a Rel() je výsledek jakéhokoliv výpočtu ekvivalentní sekvenčně konzistentní paměti

Výstupní konzistence

P1: **Acq** W(x)1 W(x)2 **Rel**
P2: **Acq** R(x)2 **Rel**
P3: R(x)1

release consisteny

bez přístupu k SP
libovolně stará hodnota

- Možnosti implementace
  - Eager release consistency *(horlivá, dychtivá)*
    - po Rel() se propagují změny všem procesům
    - optimalizace přístupové doby
  - Lazy release consistency *(líná)*
    - po Rel() se nic nepropaguje, až při Acq() jiného procesu
    - optimlizace síťového provozu

Vstupní konzistence

Entry consistency

1. Acq() k SP není povolen, dokud nebyly provedeny všechny aktualizace chráněných sdílených dat procesu.
2. Exkluzivní přístup procesu k SP (= zápis) je povolen pouze v případě, že žádný jiný proces nepřistupuje k SP, a to ani neexkluzivně (= čtení).
3. Po exkluzivním přístupu k SP si příští neexkluzivní přístup libovolného procesu k SP musí vyžádat aktuální kopii dat od vlastníka SP.

- sdílená data jsou vázána na SP, při přístupu se synchronizují pouze tato data
- přístup k datům a SP může být exkluzivní (RW) nebo neexkluzivní (RO)
- každá SP má vlastníka - proces, který k ní naposledy přistupoval
- vlastník může opakovaně vystupovat do a vystupovat z kritické sekce bez nutnosti komunikace
- proces, který není vlastníkem, musí požádat o vlastnictví

Nezávislá data x a y

P1: **Acq(Lx)** W(x)1 **Acq(Ly)** W(y)2 **Rel(Lx)** **Rel(Ly)**

P2: **Acq(Lx)** R(x)1 R(y)0

P3: **Acq(Ly)** R(y)2

Shrnutí konzistenčních modelů

	Konzistence	*Vlastnosti*
bez SP	Striktní	Absolutní časové uspořádání
	Sekvenční	Všechny události jsou vidět ve stejném pořadí
	Kauzální	Kauzálně vázané události jsou vidět ve stejném pořadí
	PRAM	Události jednoho procesu jsou vidět ve stejném pořadí
	Slow mem	Zápisy jednoho procesu na jedno místo jsou vidět ve stejném pořadí
s SP	Slabá	Sdílená data jsou konzistentní po synchronizaci
	Výstupní	Sdílená data jsou konzistentní po opuštění kritické sekce
	Vstupní	Sdílená data vázaná na kritickou sekci jsou konzistentní při vstupu do kritické sekce

- Konzistenční modely bez použití synchronizačních proměnných
  - implementace možná na úrovní virtuální paměti
  - procesy nemusí o DSM vůbec vědět
  - standardní programovací jazyky / knihovny
- Konzistenční modely s použitím synchronizačních proměnných
  - speciální jazyky nebo knihovny
  - procesy musí být korektně naprogramovány
  - vyšší výkonnost

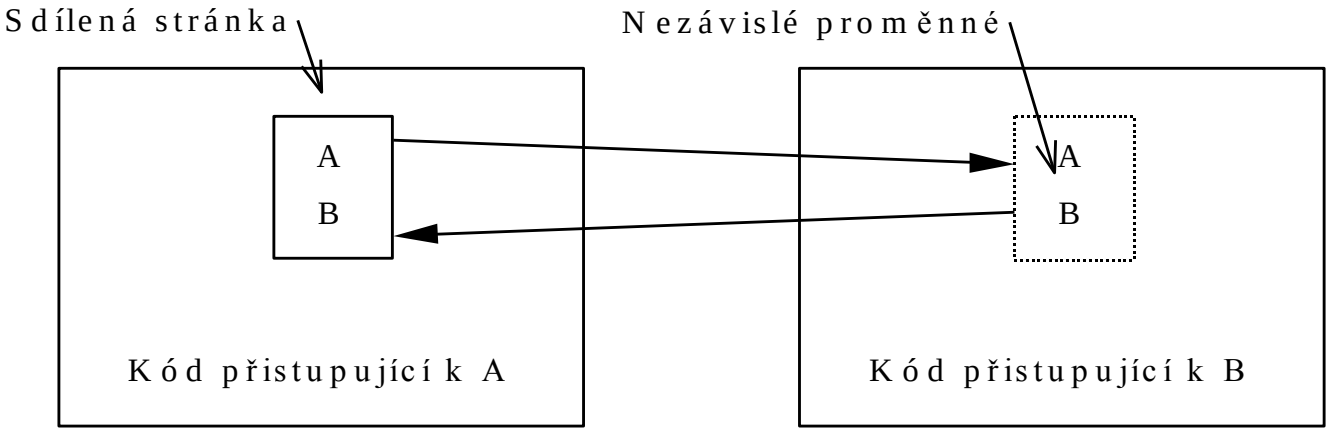
Distribuované stránkování

- Obdoba virtuální paměti
  - ♦ přístup k nenamapované stránce - přerušení, obsluha, načtení
- Problémy
  - ♦ replikace → konzistence — *jak udržovat data konzistentní*
    - namapování read-only, při zápisu synchronizační akce
    - invalidace vs. aktualizace
  - ♦ nalezení stránky — *jak nalézt distribuovaná data*
    - broadcast
    - centralizovaný manager
    - replikovaný manager - indexace spodními n bity / hash
  - ♦ správa kopií — *co dělat s kopiemi při čtení / zápisu*
    - broadcast
    - copyset - vlastník stránky udržuje množinu lokací kopií
  - ♦ uvolňování stránek — *kterou stránku uvolnit*
    - nevlastněná read-only kopie
    - vlastněná replikovaná kopie - přenos vlastnictví
    - lokální heuristika (LRU, ...)
  - ♦ falešné sdílení — *nezávislá data na stejné stránce*

Distribuované stránkování - falešné sdílení

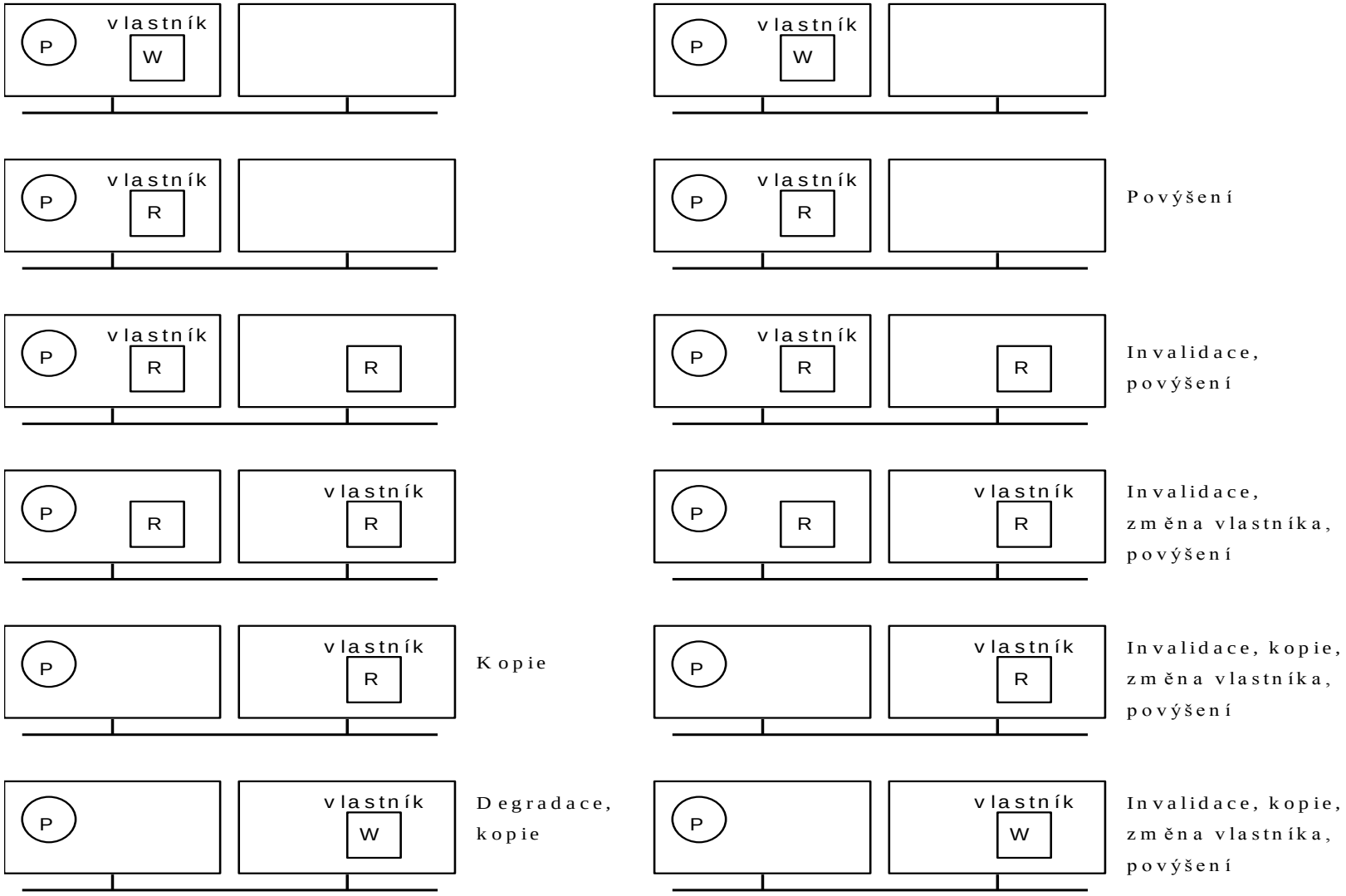
- Problémy
  - falešné sdílení

nezávislá data na stejné stránce



Sekvenčně konzistentní distribuované stránkování

čtení



Kauzálně konzistentní distribuované stránkování

- základní idea – graf závislostí
- vektorové hodiny:
  - VT_S jednotka granularity (stránky)
  - VT_P procesy
- VT[i] ≈ stránka i
- výpadek stránky – přenos dat, $VT_P = \max(VT_S , VT_P)$
- zápis do stránky – $VT_S = inc(VT_P)$
- aktualizace VT_P – zneplatnění stránek i: $VT_{Si}[i] < VT_P[i]$

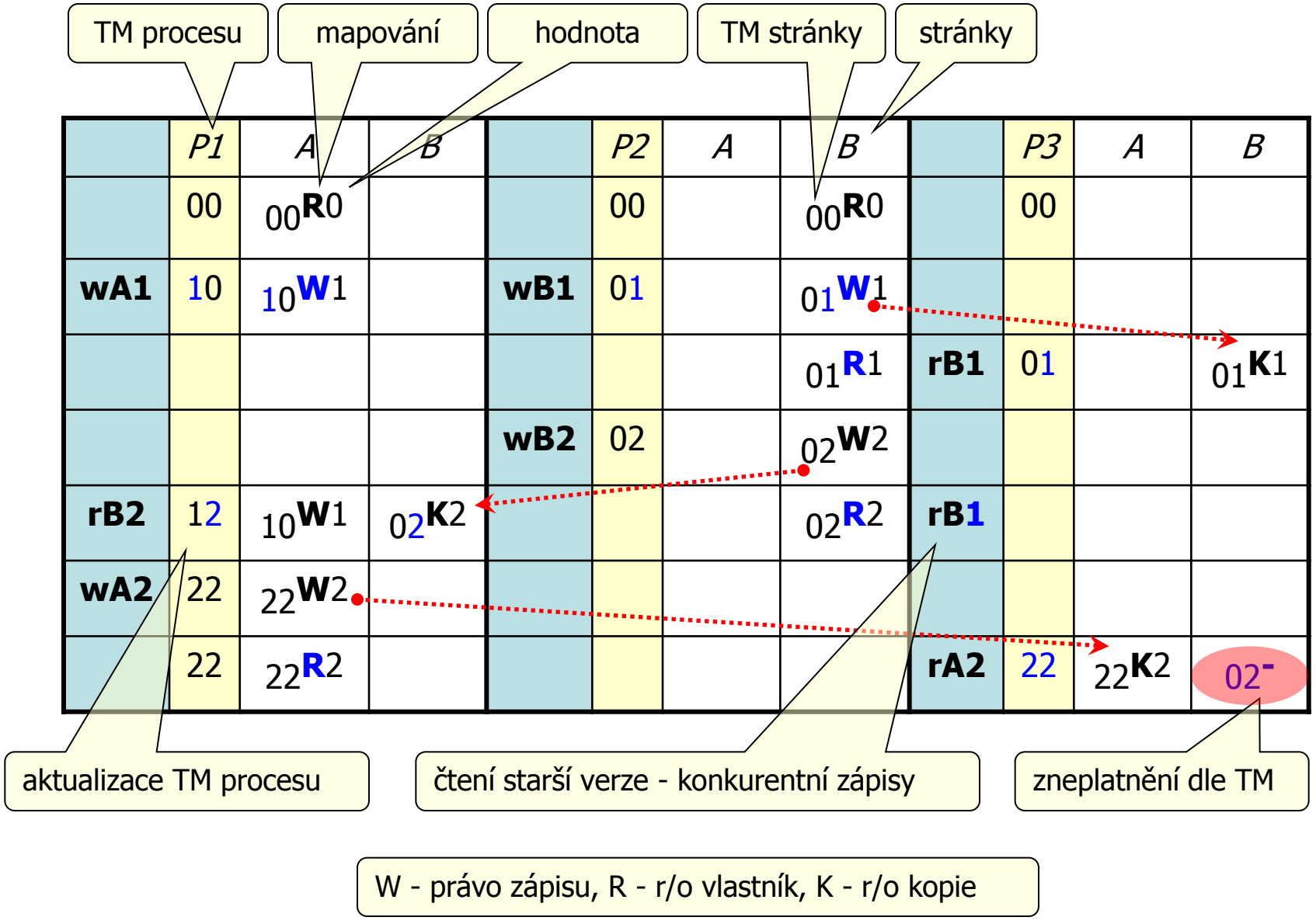
na kterých stránkách závisí obsah

z kterých stránek znám data

jak OS pozná zápis?
změna mapování

- Problémy:
  - velká prostorová režie (1 MB = 256 stránek = 512 B / stránku)
  - propagace konkurentních zápisů
    - zámky, bariéry, timeouty

Kauzálně konzistentní distribuované stránkování



Kauzálně konzistentní distribuované stránkování

- základní idea – graf závislostí
- vektorové hodiny:
  - VT_S jednotka granularity (stránky)
  - VT_P procesy

na kterých stránkách závisí obsah

z kterých stránek znám data

- VT[i] ≈ stránka i
- výpadek stránky – přenos dat, $VT_P = \max(VT_S , VT_P)$
- zápis do stránky – $VT_S = inc(VT_P)$
- aktualizace VT_P – zneplatnění stránek i: $VT_{Si}[i] < VT_P[i]$

- Problémy:
  - velká prostorová režie (1 MB = 256 stránek = 512 B / stránku)
  - propagace konkurentních zápisů
    - zámky, bariéry, timeouty

Distribuované sdílené proměnné

- Implementace na úrovni knihoven
  - potenciálně replikovaná distribuovaná databáze
  - typicky konzistenční model se synchronizačními proměnnými
- Výhody
  - potenciálně lepší výkonnost
  - eliminace falešného sdílení
- Nevýhody
  - nepodporováno přímo operačním systémem
  - nutnost implementace pro různé jazyky
  - nutnost rekompilace
- Příklad implementace: Munin 1990 Benett & spol.

- Distribuované objekty
  - díky zapouzdření flexibilnější - komunikace a synchronizace v metodách
  - distribuovaná data potomkem základní distribuované třídy
  - Class Definition Language - automatické generování hlaviček a kódu
  - základní "distribuovaná" třída, dědění vlastností, CORBA, Java RMI, ...

Middleware

Implementace sdílených dat - Munin

- ordinary / shared / synchronization variables
- sdílené proměnné - *read-only, migratory, write-shared, conventional*
- read-only
  - při výpadku je v adresáři nalezen vlastník, žádost o read-only kopii dat
- migratory
  - acquire/release protokol implementující eager release consistency
  - při opuštění kritické sekce se propagují změny
  - data chráněná SP migrují na uzel v kritické sekci
- write-shared
  - stránky iniciálně r/o, při zápisu kopie s původním obsahem r/w, označí se dirty
  - po release se porovná stránka s původní, změny se propagují
  - propagace na ne-dirty stránku akceptace, jinak word-po-wordu porovnání
  - sjednocení dat / konflikt - runtime-error
- konvenční sdílená data *(nepatřící do žádné z výše uvedených kategorií)*
  - jako distribuované stránkování - single writer/many readers
  - sekvenční konzistence

Identifikace a správa prostředků

- Identifikace a přístup k objektům
  - který objekt má být použit (identifikace)
  - kde je tento objekt umístěn (adresa)
  - jak je možné se k němu dostat (cesta)

- Druhy a struktura identifikací
- Kapability, distribuovaná správa jmen, přístup k objektům
- Uváznutí (deadlocks)
- Distribuované hashovací tabulky

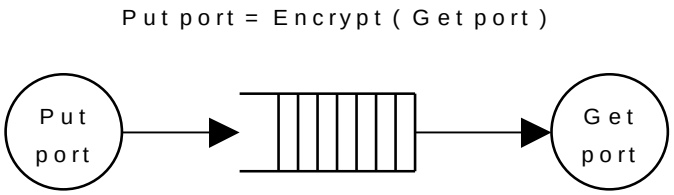
Trvanlivost identifikací, prostory jmen

- Dynamická jména *(dočasná, tranzientní)*
  - typicky vázána na životnost procesu
  - deskriptory otevřených objektů, porty, adresy DSM
- Statická jména *(trvalá, perzistentní)*
  - nezávislá na procesech
  - jména souborů, kapability
- Převod dočasných a trvalých jmen
  - freeze / melt
  - *port / kapabilita*

- Prostor jmen
  - separátní prostory - file system, registry, URL, identifikace procesů, ...
  - jednotný - distribuovaný name server
  - lokační a replikační transparentnost

Systémová jména, matice přístupových práv

- Systémová jména
  - interní identifikace objektů, optimalizace pro strojové zpracování
  - typicky binární řetězce pevné délky i struktury
  - chráněné objekty jádra - porty



dvojice přístupu k portu
služební / uživatelský

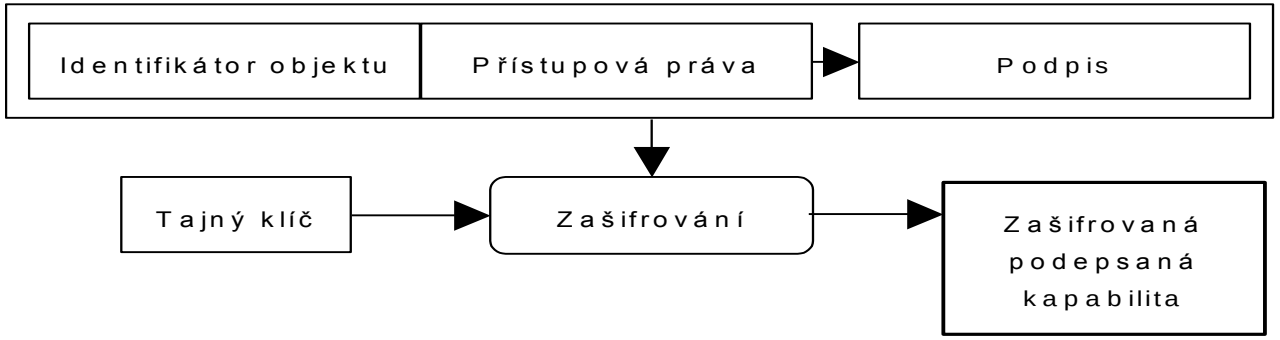
- Access Control Matrix - user $\times$ resource $\rightarrow$ effective right
  - Capabilities — uživatel drží oprávnění k prostředkům
  - Access Control List — prostředek má seznam uprávněných uživatelů

Kapability

- Kapability (capabilities)
  - datová struktura umožňující jednoznačnou identifikaci objektu
  - obsahuje navíc přístupová práva pro držitele kapability
  - ochrana objektů, šifrování, redundance
  - k jednomu objektu typicky několik různých kapabilit
    - vlastník, písař, čtenář, ... další druhy služeb
  - uživatelským procesům znemožněno vlastní generování kapabilit i změny práv
- Výhody
  - snadný test oprávěnosti přístupu
  - flexibilita - každý správce prostředků může nadefinovat vlastní druhy práv
- Nevýhody
  - kontrola propagace
    - copy bit, čítač - chráněný způsob přenosu
  - review - seznam oprávněných uživatelů
  - revocation - odejmutí práva
    - zrušení objektu a vytvoření nového, notifikace ostatních uživatelů
  - garbage collection

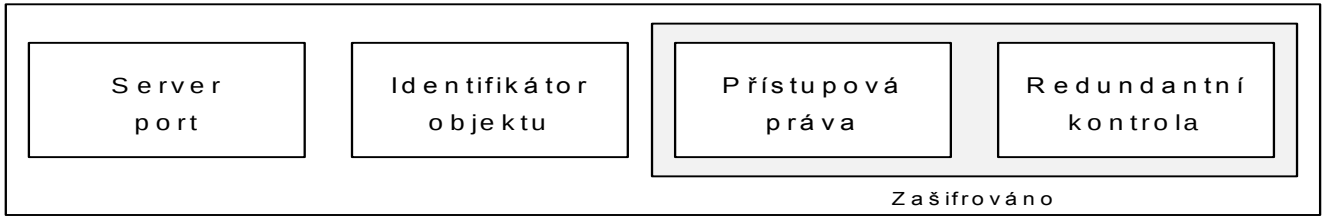
Implementace kapabilit

- Kapabilita s podpisem
  - hlavní část - identifikace objektu a přístupových práv
  - platná kapabilita je rozšířena o podpis, který je vypočítán z obsahu hlavní části
  - celá kapabilita zašifrována tajným klíčem
    - ochrana proti odvození podpisové funkce uživatelskými procesy
  - řídkost: ze všech binárních čísel velikosti kapability jsou platné jen ty, jejichž podpis odpovídá zbytku kapability



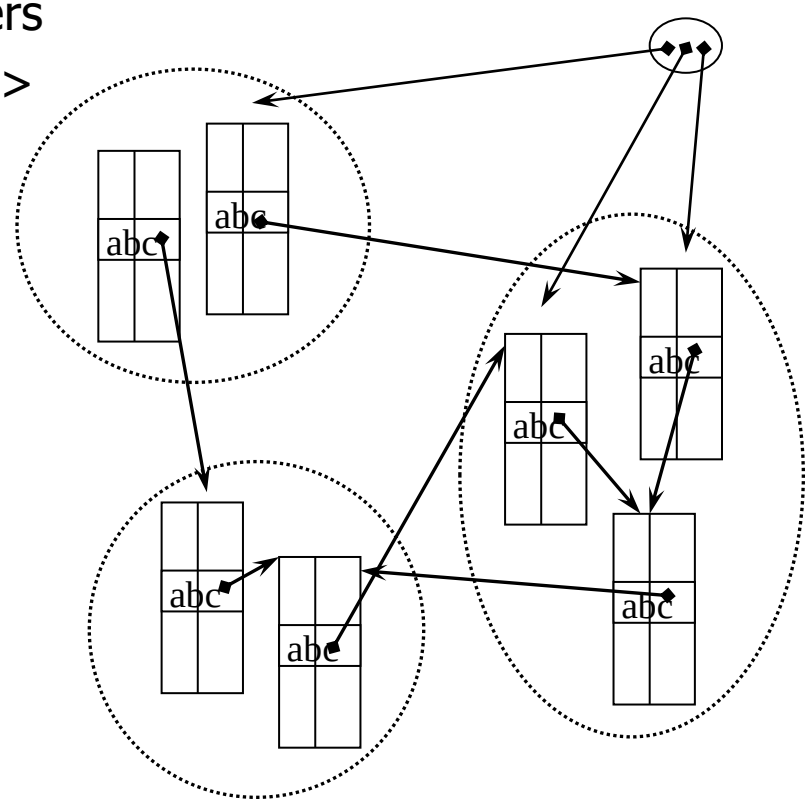
Implementace kapabilit

- Kapabilita s redundantní kontrolou
  - port serveru a identifikátor objektu jsou volně přístupné
  - přístupová práva a náhodně vygenerované binárním číslo pevné délky - zakódováno
  - ochrana serveru
    - port, dostatečně velké náhodně generované číslo
  - ochrana přístupových práv
    - zašifrování pole s přístupovými právy spolu s redundantní kontrolou
  - uživatelskému procesu znemožněna změna přístupových práv
    - proces nemůže svá práva změnit, nezná dešifrovací funkci, ani ji nemůže odvodit
    - náhodně generovaná redundance
  - služba serveru - vygenerování kapability s menšími právy



Distribuovaná správa jmen

- Name servers, directory services, lookup servers
- Adresáře – množina položek <jméno, hodnota>
- Hodnoty:
  - primitivní
    - čísla, řetězce, binární data, ...
  - perzistentní reference
    - trvalé odkazy na objekty, kapability
  - tranzientní reference
    - odkazy na živé objekty, porty, kanály
  - odkazy na jiné adresáře – lokální / vzdálené

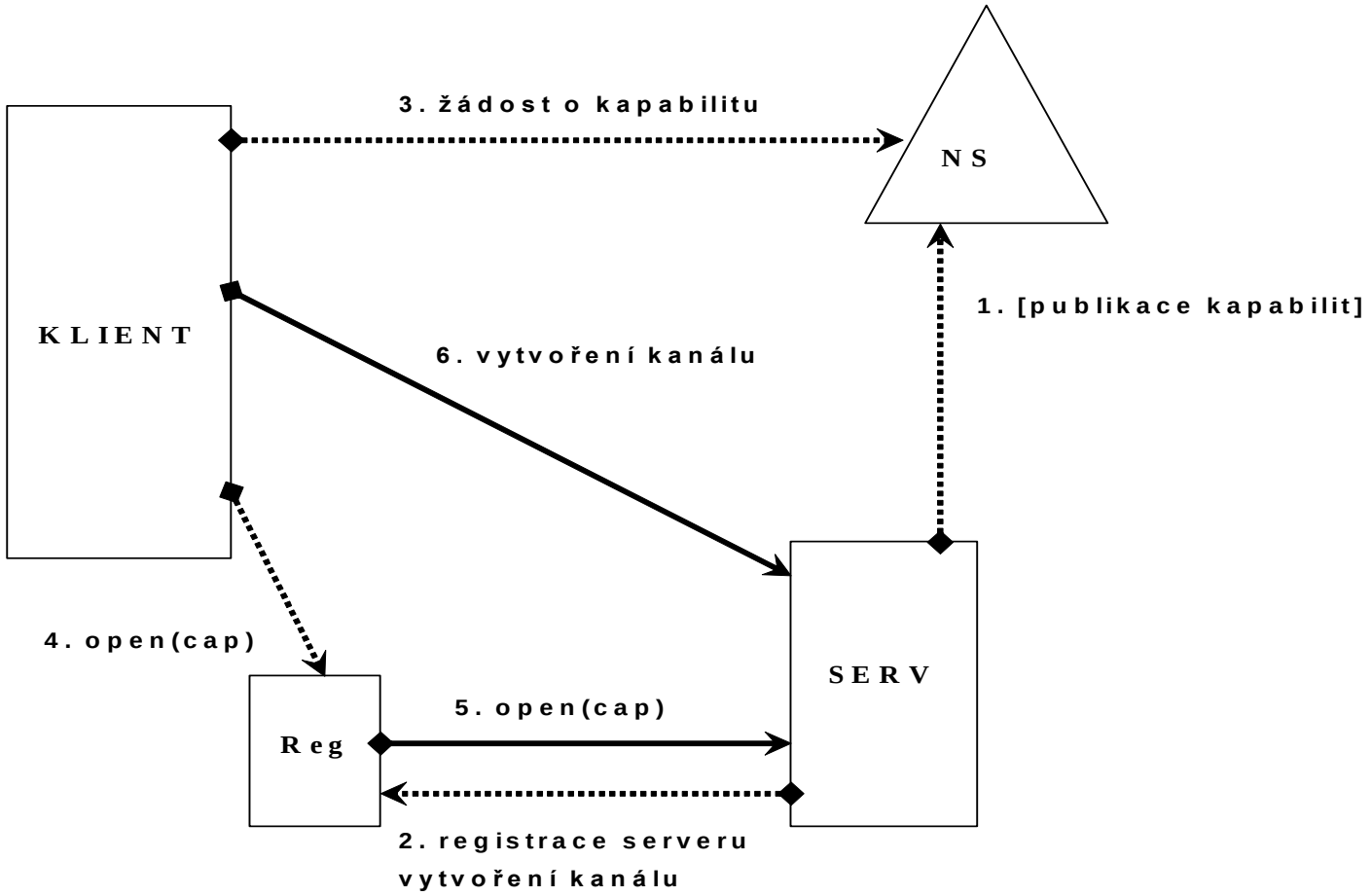


- Základní operace:
  - **`adresář -> set( jmeno, hodnota);`**
  - **`adresář -> lookup( slozene_jmeno);`**
    - `adresář -> lookup("A") -> lookup("B") -> lookup("C");`

- Klientský proces – několik základních 'adresářů'
  - filesystem, objekty, služby a servery, registry, ...

Přístup k objektům

- Server spravuje objekty *(identifikace, operace nad nimi)*
  - každý objekt má lokální identifikátor spravovaný serverem
  - na něm definovány operace / služby
  - server publikuje řídící port a porty pro otevřené objekty



Služby pro přístup k objektům

- vytvoření objektu, tranzientní reference

```
obj_port = service_port -> create_object(...);
```

porty ≈ objekty
služby ≈ metody

- operace nad objektem

```
obj_port -> op(...);
```

- freeze – vytvoření kapability (perzistentního odkazu)

```
cap = obj_port -> freeze();
```

- zrušení tranzientní reference (ne objektu)

```
obj_port -> drop();
```

- uložení kapability s plnými právy

```
ns -> add( "mydata/cap/this_obj", cap);
```

- vytvoření kapability s restringovanými právy

```
cap2 = service_port -> cap_restrict( cap, 0101);
```

- zveřejnění restringované kapability

```
ns -> add( "public/published_obj", cap2);
```

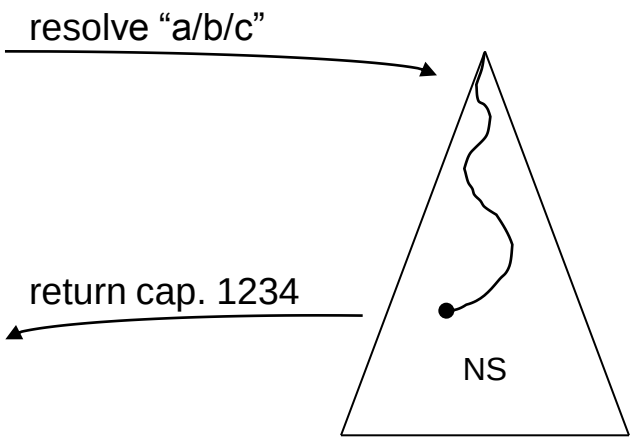
- vyzvednutí kapability

```
cap = ns -> resolve( "public/published_obj");
```

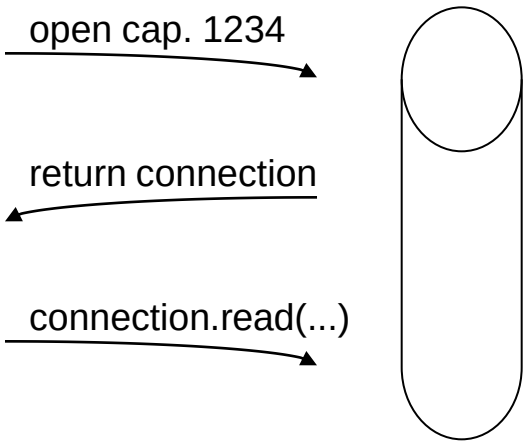
- otevření objektu - vytvoření tranzientní reference (portu)

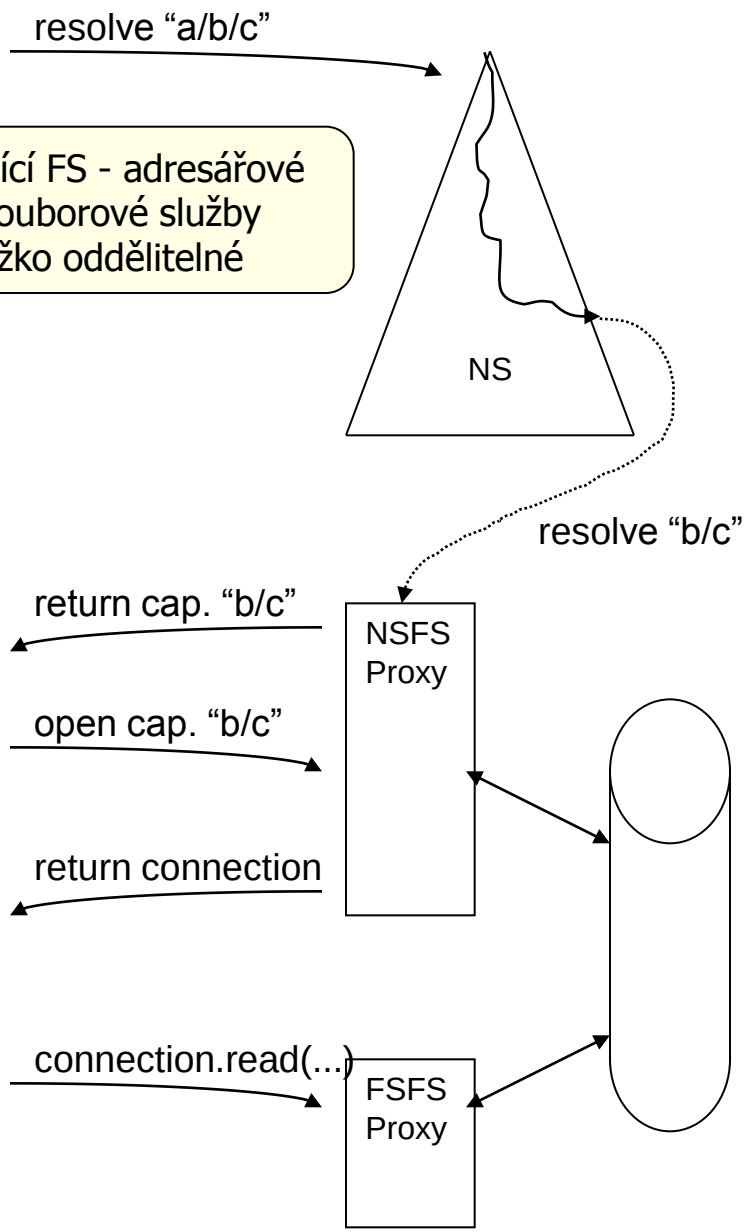
```
obj_port = service_port -> melt( cap);
```

Spolupráce name serveru a souborového serveru



oddělené adresářové a souborové (objektové) služby

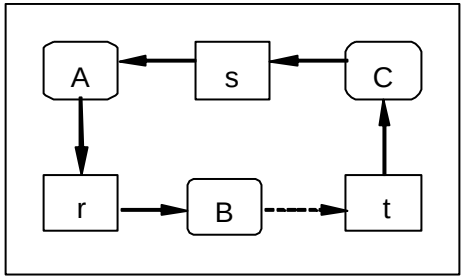




Distribuovaná správa prostředků

- Centralizované řešení - resource manager
- Pokusy o distribuovanou správu
  - prakticky nepoužitelné - *distribuované vyloučení procesů*
- Migrace - identifikace, komunikace
- Replikované servery – aktualizační protokol

- Uváznutí (deadlock)
  - typické řešení - pštrosí algoritmus
  - detekce - větší problém než u centralizovaných systémů
  - WFG – Wait-For-Graph
    - orientovaný graf (procesů, transakcí)
    - P1 → P2 proces P1 je blokován procesem P2
    - orientovaná kružnice – uváznutí



Algoritmy detekce deadlocků

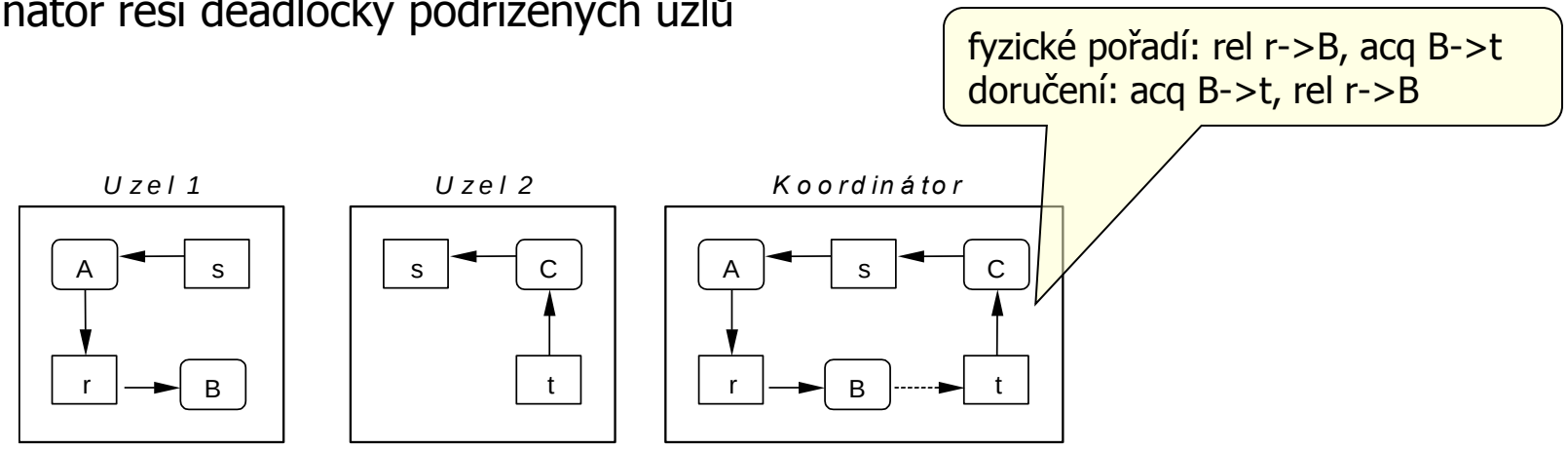
- Korektnost algoritmu detekce deadlocku
  1. Každý existující deadlock je v konečném čase detekován
  2. Detekovaný deadlock musí existovat
     - Pozor na vnořené deadlocky - phantom deadlock

- Modely deadlocků
  - single model
  - AND model
  - *OR model*
  - *m-out-of-n model*
  - *AND-OR model*

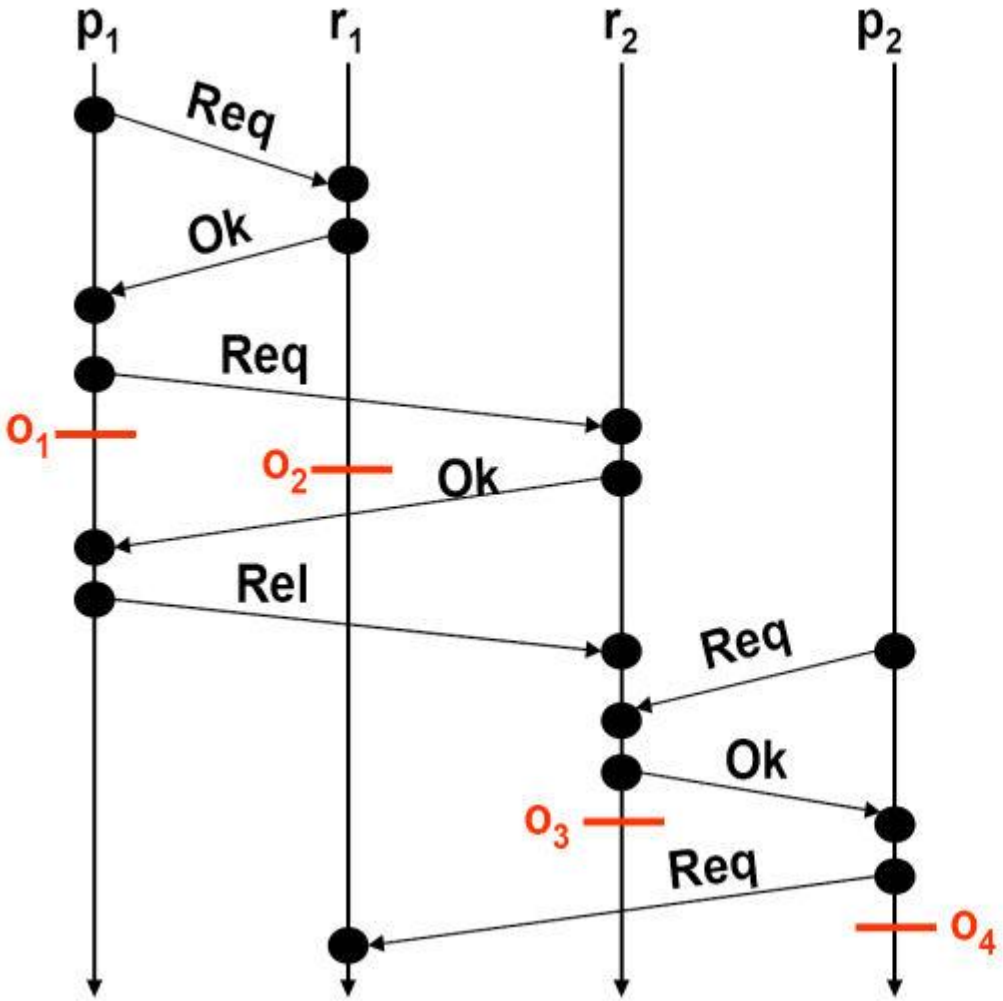
- Metody konstrukce WFG
  - centralizované řešení, hierarchické
  - path-pushing - kontrakce a distribuovaná kolekce WFG
  - probe based: edge-chasing, diffusing computation
  - detekce globálního stavu

Centralizovaný algoritmus

- *Ho-Ramamoorthy, Bernstein*
- Přenos informací:
  - po každé změně lokálního stavu
  - v pravidelných intervalech
  - na požádání
- Problém - falešné uváznutí kvůli zpoždění zpráv
- Řešení – logické hodiny, kauzální doručování
- Rozšíření - hierarchický algoritmus
  - uzel řeší deadlocky lokálně
  - koordinátor řeší deadlocky podřízených uzlů



Detekce falešného deadlocku



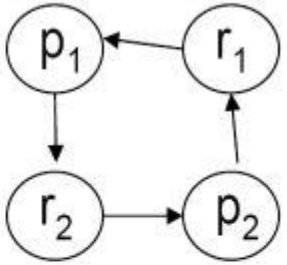
Inference from

o_1: p_1 waits for r_2

o_2: r_1 waits for p_1

o_3: r_2 waits for p_2

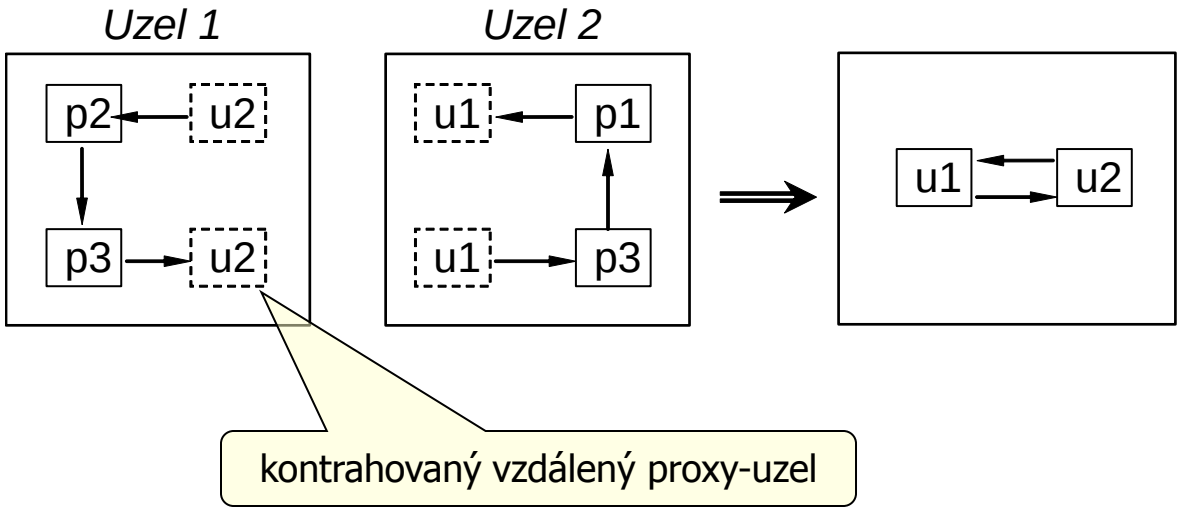
o_4: p_2 waits for r_1



From $O=\{o_1,o_2,o_3,o_4\}$ the deadlock detector concludes there is a deadlock!

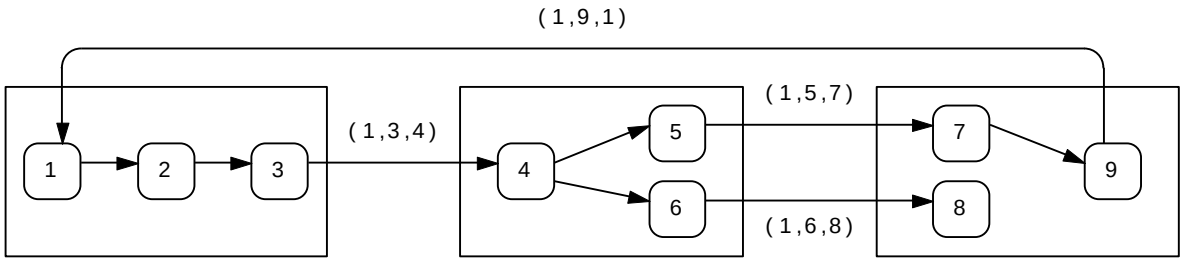
Path-pushing

- *Obermarck, Menasce-Muntz, Gligor-Shattuck*
- WFG distribuovaný, uzly spravují lokální části WFG
- Sousedním uzlům jsou zasílány externí závislosti
- Mnoho publikovaných algoritmů/protokolů nekorektních
  - phantom deadlock



Edge-chasing

- *Chandy-Misra-Haas, Stankovic, Singhal-Kshemkalyani, Roesler*
- Speciální zprávy (probes) jsou zasílány podél WFG
  - proces rozešle zprávu všem procesům, kterými je blokován
  - pokud se zpráva vrátí odesilateli – deadlock (orientovaná kružnice)
- Paralelní spuštění - overkill
  - zpráva zároveň hledá vhodného kandidáta



Edge-chasing Chandy-Misra-Haas

Sending the probe:

if Pi is locally dependent on itself then deadlock.

else for(all Pj and Pk such that Pi is locally dependent upon Pj

& Pj is waiting on Pk & Pj and Pk are on different sites)

send probe(i,j,k) to the home site of Pk

Receiving the probe(i,j,k):

if Pk is blocked & $dependent_k(i)$ is false & Pk has not replied to all requests of Pj

{ $dependent_k(i) := true$;

if k = i then Pi is deadlocked

else ... *(send)*

}

Pk ví o tom, že
Pi čeká na Pk

Další algoritmy detekce deadlocků

- **Diffusing computation**
  - *Chandy-Misra-Haas, Chandy-Herrmann*
  - 'distribuovaný výpočet' končí v okamžiku nalezení cyklu
  - pokud se zpráva dostane k iniciátorovi, vracejí se signály
  - iniciátor po obržení signálu může rozhodnout
  - vhodné pro složitější modely *(OR, n-nad-m)*

- **Detekce globálního stavu**
  - *Bracha-Toueg, Kshemkalyani-Singhal*
  - jestliže WFG $\rightarrow$ WFG' a proces je v deadlocku v WFG, pak je v deadlocku i v WFG'
    - existuje-li deadlock, pak existuje i v konzistentním řezu
  - při příjmu značky (okamžik řezu) uzel zaznamená lokální WFG
  - lokální kontrakce WFG, externí závislosti zaslány iniciátorovi
  - distribuovanější varianta - redukovaný edge-chasing na WFG

Distribuované procesy

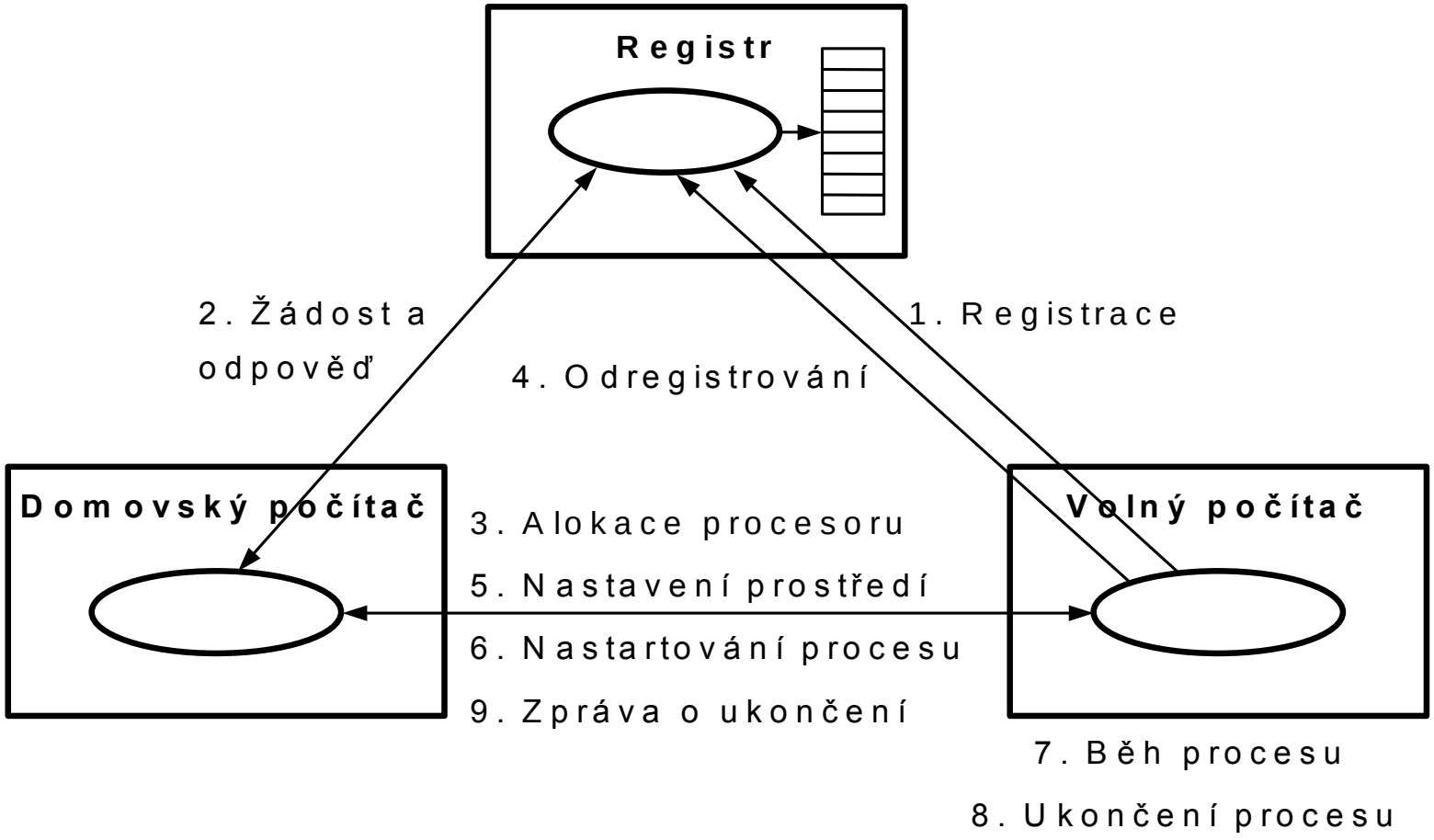
- Správa procesů v distribuovaných systémech
  - sdílet výpočetní sílu systému
  - rozdělovat zátěž na jednotlivé procesory
  - provádět operace na vzdálených procesech
  - synchronizovat procesy a vést evidenci stavu

- Vzdálené spouštění procesů
- Alokace procesorů
- Migrace procesů
- Vyvažování zátěže

Vzdálené spouštění procesů

- Nalezení volného počítače
  - alokační algoritmy
- Znalost vlastní zátěže
  - procento využití procesoru
    - periodické měření - vyhlazení (nějaký klouzavý průměr)
- Spuštění vzdáleného procesu
  - transparentnost
  - přenesení kódu a dat – nezajímavé
    - heterogenní prostředí
  - vytvoření prostředí odpovídající domovskému počítači
    - kontexty (fs, naming, environment)
    - systémová volání – vzdálená / přesměrování
- Hostitelský počítač přestane být volný *(uživatel se vrátil z oběda, potřeba restartu)*
  - doběhnutí
  - zabití
  - čas na uložení / uzavření
  - *migrace*

Vzdálené spouštění procesů



Klasifikace alokačních algoritmů

- zda jsou předem známy údaje o procesech, podle kterých se algoritmus rozhoduje
  - deterministické / heuristické
- do jaké míry se provádí optimalizace
  - optimální / suboptimální
- co se snaží optimalizovat
  - využití CPU / čas odpovědi / přidělování prostředků / komunikační složitost, ...
- zda umožňují přemístění již rozběhnutého procesu
  - migrační / nemigrační
- rozdíly mezi procesory a speciální požadavky procesů na vlastnosti procesorů
  - homogenní / heterogenní
- podle charakteru algoritmu
  - centralizované / distribuované
- podle jakých informací se provádí rozhodnutí o vzdáleném spuštění procesu
  - lokální / globální
- kdo iniciuje vzdálené spuštění procesu
  - odesilatel / příjemce / symetrický

Alokace procesorů

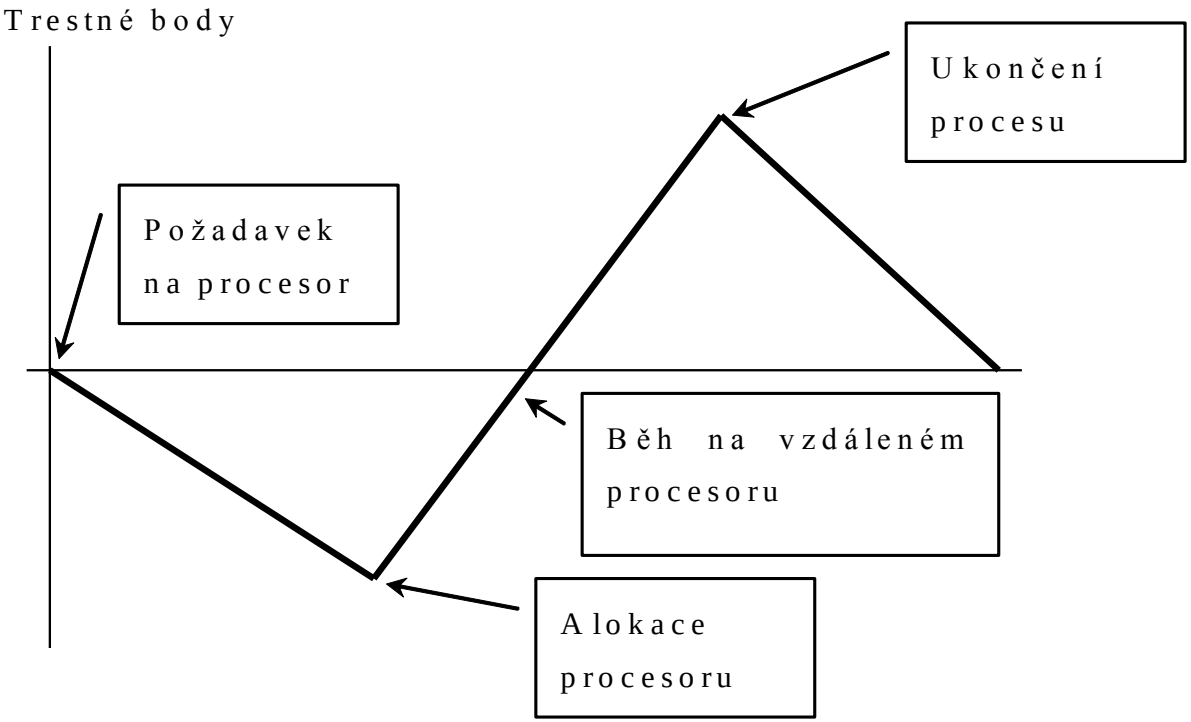
- Implementace alokačních algoritmů
  - deterministický grafový algoritmus
    - minimalizace komunikace
    - nutnost znalosti komunikační složitosti
    - optimální deterministický algoritmus – tok v sítích
  - hierarchický algoritmus
    - manažeři skupin, při neúspěchu žádost vyšším místům
  - distribuovaný heuristický algoritmus
    - *k* náhodných výběrů cíle
    - server / receiver initiated
  - bidding (obchodní, nabídkový) algoritmus
    - procesy kupují výpočetní sílu, procesory ji nabízejí
  - up-down algoritmus

Up-down algoritmus

- Mutka-Livny 1987
- Optimalizace na stejnoměrné sdílení výkonu
- Koordinátor
  - tabulka se záznamem pro každý procesor obsahující "trestné body"
- Při každé významné akci *(vytvoření procesu, ukončení procesu, tik hodin)* zpráva koordinátoru, který provede změny:
  - každý proces běžící na jiném počítači - plus trestné body
  - každý neuspokojený požadavek - mínus trestné body
  - jestliže nic z tohoto - směrem k nule
- V okamžiku uvolnění procesoru se vezme ten proces z fronty *(neuspokojených požadavků)*, jehož vysílající procesor má nejméně trestných bodů

Up-down algoritmus

Homogenní
Nemigrační
Spravedlivý
Heuristický
Centralizovaný
Suboptimální
Globální
Symetrický



Migrace procesů

korektní a transparentní přenesení procesu během výpočtu

- Motivace:
  - vyvažování zátěže; optimalizace - I/O, komunikační; přemístění; shutdown
- Korektnost
  - ostatní procesy nejsou migrací podstatně ovlivněny
  - po ukončení stav odpovídá stavu bez migrace
- Transparentnost
  - proces o migraci neví a nemusí spolupracovat
  - zůstanou zachovány vazby na komunikující procesy
  - není narušena komunikace
    - ... v konečném důsledku
- Problémy k řešení
  - přenesení rozpracovaného stavu ...
  - přenesení adresového prostoru ...
  - reinicializace vazeb na ostatní procesy
  - komunikace s ostatními procesy
  - reziduální dependence - žádná, domácí, průběžná, dočasná
  - vícenásobná migrace - viskozita

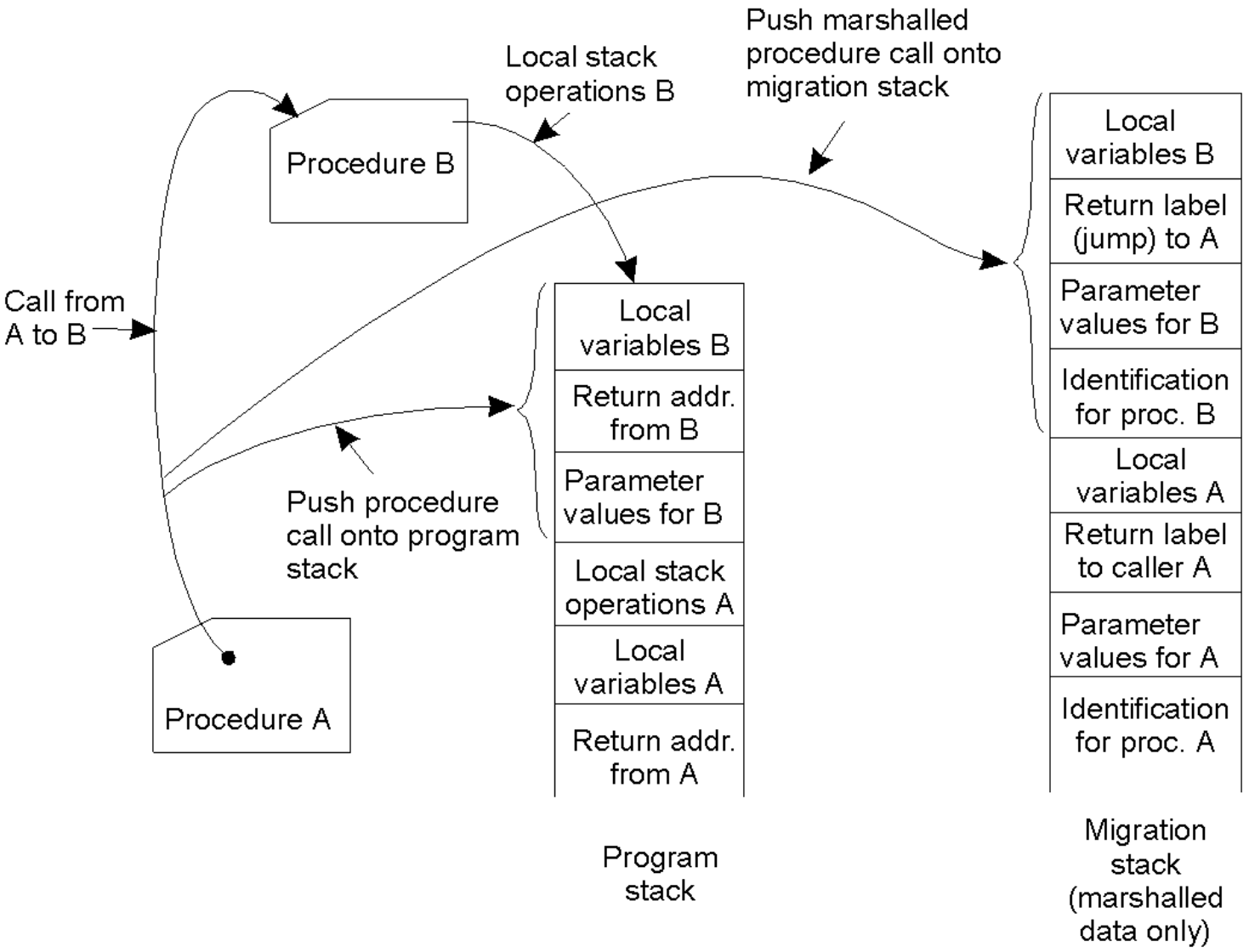
Implementace migrace - přenos procesu

- Přenos procesu
  - vyjmutí ze stavu, zmražení, speciální stav
  - oznámení příjemci o migraci, alokace procesu
  - přenos stavu - registry, zásobník, stav procesu
  - přenos kódu / adresového prostoru
  - přesměrování / doručení zpráv
  - dealokace procesu, vyčištění
  - vazby na nové jádro, nastartování procesu
    - přesunutí části stavu spolu s procesem (obsah VM)
    - forwardování (některých) požadavků (komunikace s konzolí)
    - použití odpovídajícího prostředku na cílové stanici (fyzická paměť)
  - dokončení přenosu vazeb, dočištění
    - dočasná reziduální dependence, přesměrování zpráv

Implementace migrace - virtuální paměť

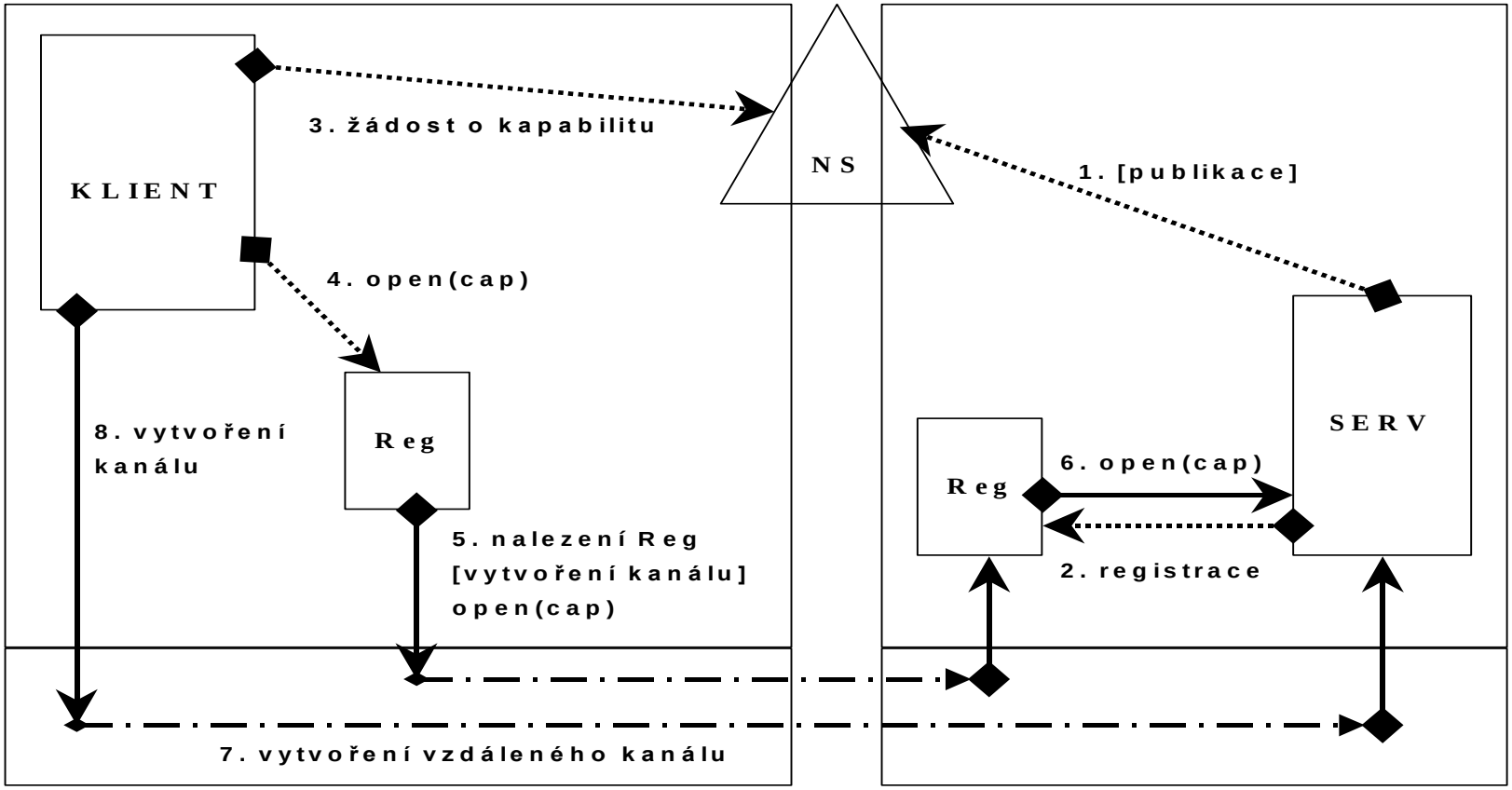
- Virtuální paměť
  - přenesení celé VM při migraci
    - výhody - eliminace reziduálních dependencí
    - nevýhody - prodloužení doby zmražení procesu, mnohdy zbytečné přesouvat celý obsah virtuálního adresového prostoru
  - Pre-copying
    - výhoda - proces je zmražen pouze po dobu přesunu malého množství dat
    - nevýhoda - některé části se kopírují vícekrát - prodloužení celkové doby migrace
  - Copy-on-reference
    - nejprve se přenese stav procesu potřebný pro běh (registry, kanály, ...)
    - přesun adresového prostoru odložen
    - stránky na cílové stanici označeny jako neprezentní (jako by byly odloženy na disk)
    - při přístupu na stránku se obsah přenese
    - nutnost evidence různých zdrojů dat pro každou stránku
      - swap, vícenásobná migrace, DSM, ...

Migrace v heterogenních systémech

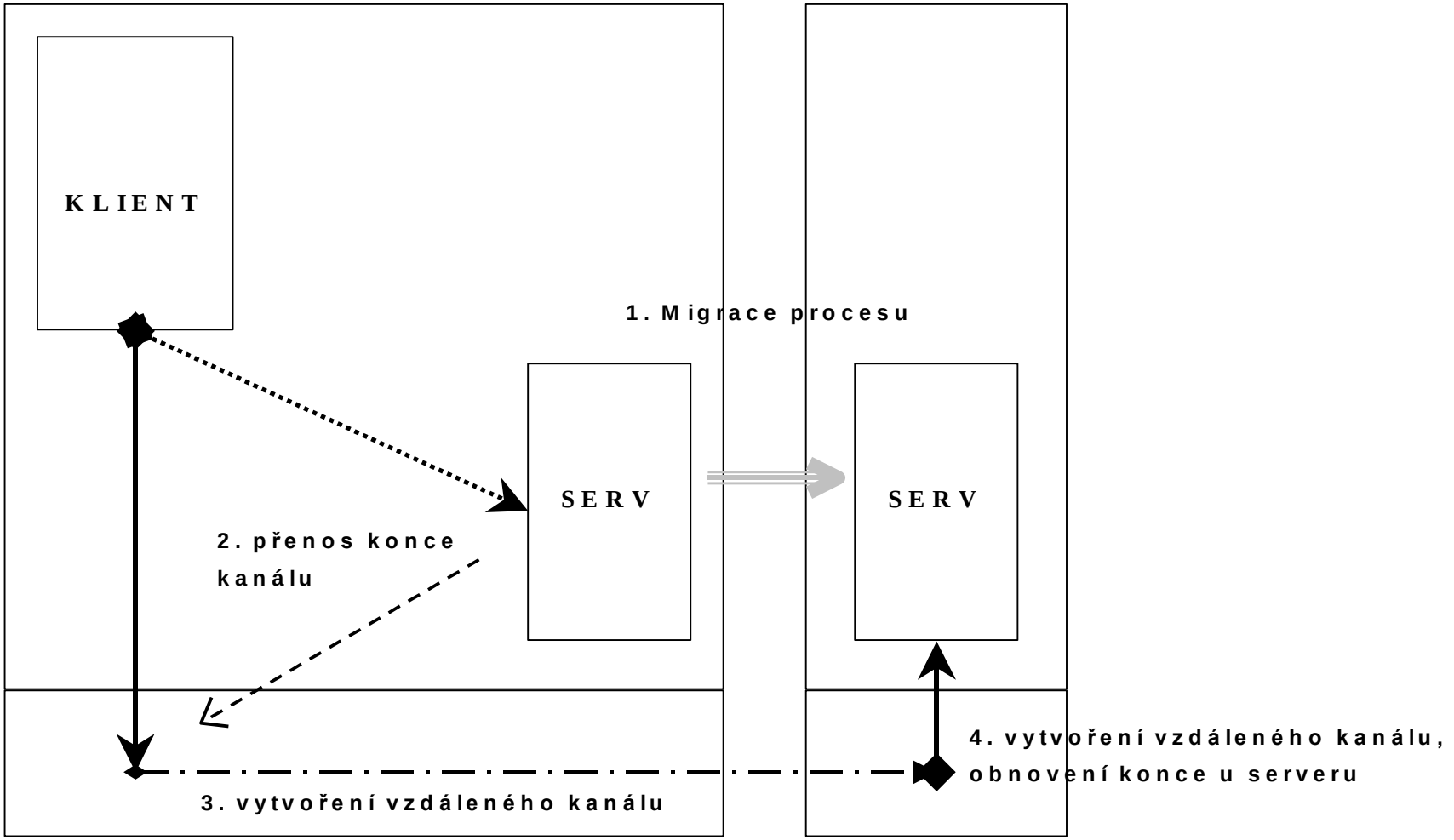


Implementace migrace - zprávy

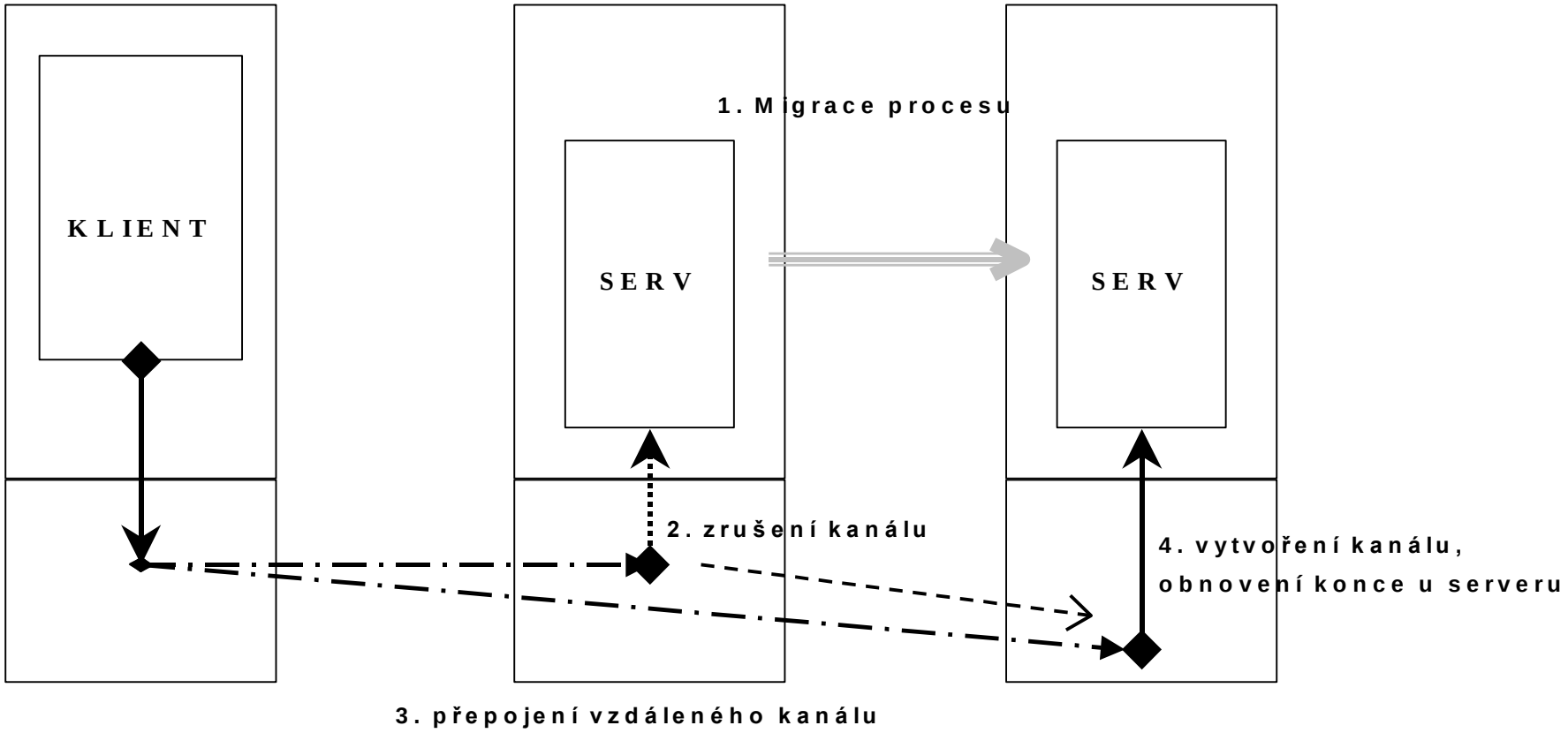
- Zprávy
  - přesměrování - dočasné / trvalé
  - oznámení předem - problém: komu všemu
  - opakování zpráv - no ACK, NAK
  - migrace kanálů, spojení front



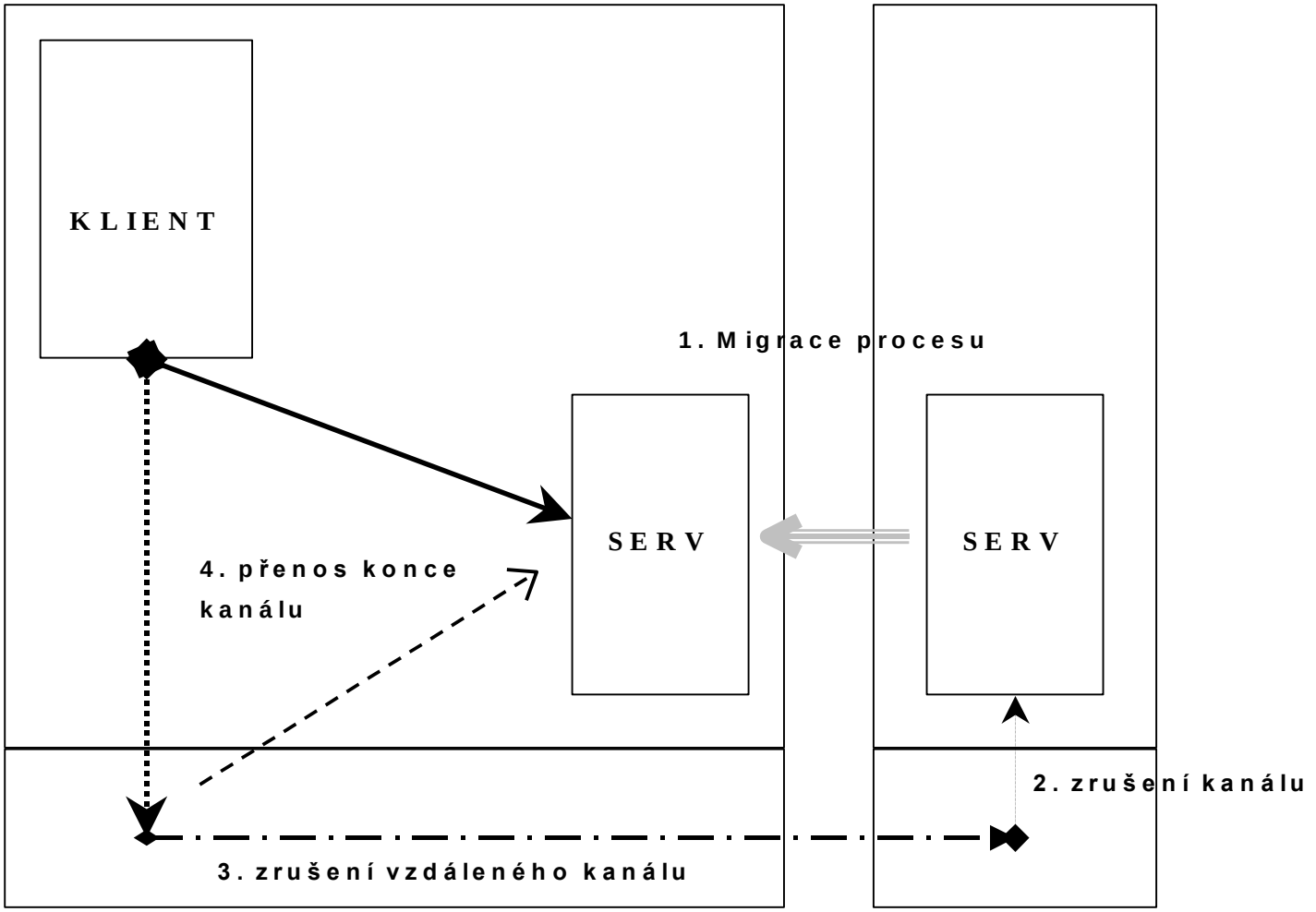
Migrace serveru vzhledem k lokálnímu klientu



Migrace serveru vzhledem k vzdálenému klientu



Migrace serveru ke klientu



Přehled některých migračních systémů

- DEMOS/MP
- Charlotte
- V
- MOSIX
- Sprite
- další ...

  Amoeba, Emerald, Spice, Accent, T4, ...

DEMOS/MP

Berkeley v r. 1983

IPC pomocí linků spojených s procesy, migrace v jádře

plná transparentnost, jednotné komunikační rozhraní nezávislé na poloze procesů

Migrace

- Vyjmutí procesu (*Zdroj*) - stav „v migraci", vyjmutí z fronty
- Podrobnější informace pro cíl (*Zdroj*) - velikost procesu, alokované prostředky
- Alokace stavu na cíli (*Cíl*) - datové struktury pro proces, alokace
- Přesun stavu (*Cíl*) - zajímavé - cílová stanice si proces „tahá" k sobě
- Přesun adresového prostoru a paměti (*Cíl*)
- Forward čekajících zpráv na cílovou stanici (*Zdroj*)
- Vyčištění stavu na zdroji (*Zdroj*) - veškerá data kromě adresy (pro forwarding)
- Restart procesu (*Cíl*)

Forwardování zpráv - 3 druhy zpráv podle toho jak přicházely

- Odeslané ale nepřijaté před migrací - přeneseny při migraci
- Odeslané po migraci za použití staré adresy - průběžný forward
- Odeslané po migraci s využitím nové adresy - není problém

Charlotte

Uni Wisconsin (Bryant, ..) 1985-87

IPC pomocí linků nezávislých na umístění procesů, možnost přesouvat linky
migrační politika v uživatelském procesu, migrace v jádru
nezůstávají reziduální dependence, snaha o maximální fault-toleranci
migrační strategie - sběr statistiky - sběr 50-80 ms, rozesílání 5-8 s

- o počítači: počet procesů, komunikační linky, využití CPU, síťová komunikace
- o procesu: doba běhu, stav procesu, využití CPU, síťová komunikace

Průběh migrace

- Negotiation
  - výměna informací mezi procesy, proces na zdrojové stanici zavolá „Migrate Out"
  - podrobnější informace o procesu na cílovou stanici, cílová stanice zavolá „Migrate In"
- Vlastní přesun procesu
  - přesun obrazu procesu
  - nastavení komunikačních linků - jádra na počátcích linků obdrží novou pozici konce,
  - přesun stavu - deskriptory procesu, komunikačních linků, událostí a zpráv
- Clean-up - vyčištění zdrojové stanice

Stanford (Cheriton) 1984-86

síťově transparentní prostředí a meziprocesová komunikace, zotavení ze ztráty zprávy

Návrh migrace

předmětem migrace „logical host" - adresový prostor, možnost několika procesů

minimalizace doby zamražení - na úkor celkové doby migrace - precopying

Mechanika migrace

Inicializace cíle

- vytvoří se nový logical host

Precopying stavu

- proces na zdrojové stanici provádí kopírování stavu (adresový prostor); proces stále běží
- migrovaný logický host je zamražen, dokončí se kopírování modifikované paměti
- kopírování stavu z jádra zdrojové stanice

Odmražení, přesměrování odkazů

- smaže se stará kopie logical hostu, nová se odmrazí
- PID se naváže na novou kopii logical hostu, navázání na nový počítač (cache)
- broadcast nové adresy logical hostu

MOSIX

Technion (Barak, Shiloh, ...) (1977-) 1988-2004(!)

UNIX adaptovaný na distribuované prostředí, rozsáhlé vyvažování zátěže

Jeden z mála prakticky používaných distribuovaných systémů

Rozšíření struktury procesu:

čas od poslední migrace, čas na procesoru, příčina migrace, statistika IPC

Vlastní migrace

Příprava

- Cílová stanice zjistí, jestli si může dovolit příjem procesu
- Zajistí opravu komunikačních kanálů
- Nastaví se paměťové oblasti
- Čeká na doruční procesu - až bude proces doručen, rutina jej probudí a vrátí se

Přesun dat - adresový prostor

Rozběhnutí odmigrovaného procesu

Na zdrojové stanici se smažou lokální data procesu

Sprite

Berkley (Douglis, Ousterhout, Welch) 1992

- Předpoklady:
  - mnoho nevyužitých stanic
  - po návratu vlastníka potřeba uvolnit (odmigrovat)
- Cíle
  - rychlost, maximální transparence
- Návrh migrace
  - vlastní migrace jádro / sběr statistiky a rozhodování provádí uživatelský proces
  - původní idea: všechna volání jádra forwardovat na domácí stanici
  - transparence vs. reziduálních dependence
  - později ústup - kombinace s voláním na cílové stanici
  - ze 106 volání Sprite: 91 lokálně, 11 forwardováno, 4 se řeší kooperací
- Virtuální paměť
  - kombinace přesunu veškeré paměti najednou a „copy-on-reference"
  - nevýhoda: paměť se přesouvá po síti dvakrát
- Jednotný mechanismus migrace různých entit
  - každá součást stavu má definovány 4 rutiny
  - pre-migration, encapsulation, de-encapsulation, post-migration routine

T4

Systém, který by

- byl dostatečně efektivní a použitelný jako konvenční operační systém
- poskytoval dostatečně silné prostředí pro použití jako distribuovaný systém

Hlavní části

- meziprocesová komunikace
- vzdálená komunikace
- přenosové protokoly
- name services
- podpora pro "vyšší" distribuované služby
  - distribuovaná sdílená paměť
  - migrace procesů, load balancing

Zkušenosti s T4

- základ pro výuku a platforma pro studentské projekty a experimenty
  - ++ diplomové práce, PGDS
- základ dalšího výzkumu v oblasti distribuovaných systémů
  - +- články, konference (málo), výzkum jiným směrem
- pro běžné každodenní použití
  - -- aplikace !!!

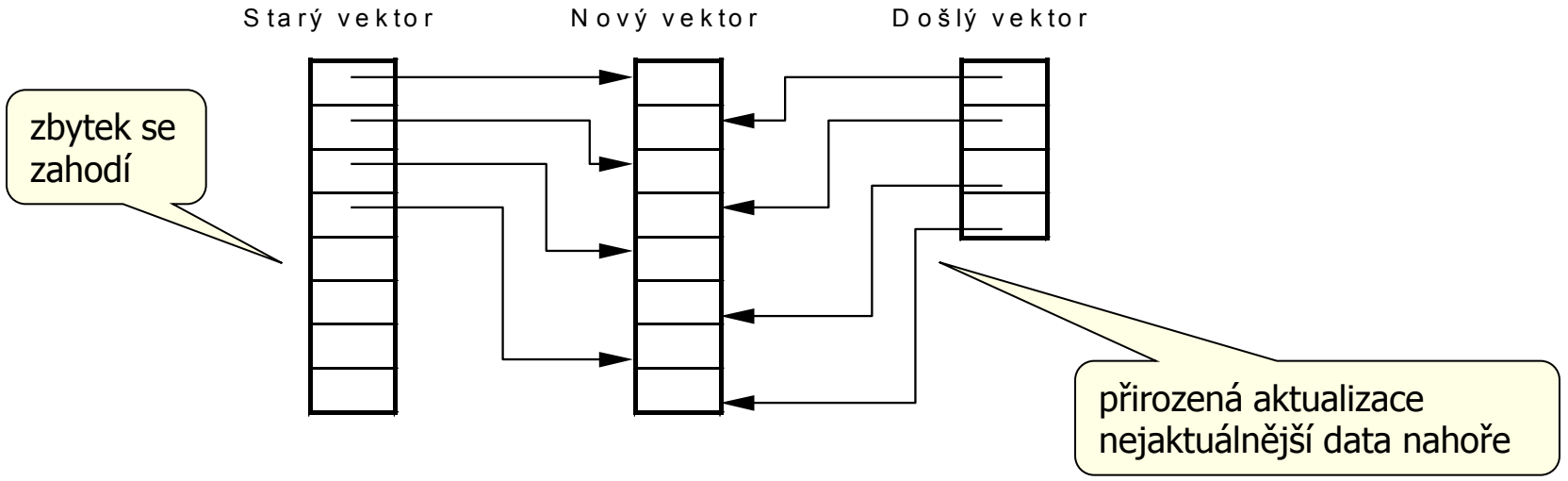
Vyvažování zátěže - load balancing

- Rozhodnutí o okamžiku migrace
  - jak porovnávat zatížení uzlů
  - udržování konzistence údajů
  - volba migrujícího procesu
  - volba příjemce
  - přenos a běh procesu

- **Párový algoritmus** (Bryant & Finkel, Charlotte)
  - vytvářejí se páry, které se vzájemně vyvažují
  - A pošle B žádost o vytvoření páru se seznamem procesů
  - B odmítne / vytvoří pár / migruje
  - zatíženější uzel vybere proces podle míry vylepšení
    - $k_i = A_i / (B_i + AB_i)$
  - významné zlepšení stavu – migrace a další proces, jinak konec

Vektorový algoritmus

- Barak & Shiloh - Mosix
  - pevný vektor zátěže L, L(0) = vlastní zátěž
  - periodicky každý uzel provádí:
    - zjistí vlastní zátěž
    - náhodně pošle polovinu vektoru
  - při příjmu L': L(2i) = L(i), L(2i+1) = L'(i)
  - k zátěži se přičte komunikační režie



Vyvažovací algoritmy

- **Bidding algoritmus** (Stankovic & Sidhu)
  - McCulloch-Pittsova vyhodnocovací procedura
  - vyhodnocovací buňka - excitátory, inhibitory, výstup
  - výstup vyšší - OK, nižší - migrace, nula - nelze migrovat (inhibitor)
  - migrace:
    - 'žádost o nabídku' (RFB) až do vzdálenosti d
    - odpovědi s nabídkou se zkorigují o režii
      - nejlepší nabídka / žádná nabídka: d++
  - problém: obtížná kvantifikace vlastností procesů

- SLA algoritmus (Stankovic)
  - stochastic learning automata - zpětné učení
- BDT algoritmus
  - bayesian decision theory - posílání globálních stavů

neprosadily se
příliš komplikované

Vyvažovací algoritmy

- **Centralizovaný / hierarchický algoritmus**
  - centralizovaný manažer, zná zátěž všech počítačů, velí
  - hierarchicky organizované skupiny, nadřazení manažeři

myšlenka:
vyvažující se skupiny
nebývají příliš velké

- **Lokální algoritmus**
  - známa pouze lokální zátěž, prahová hodnota
  - žádost n počítačům, podprahová / nejlepší odpověď OK

velmi jednoduchý
velmi suboptimální
velmi použitelný ☺

- hlediska srovnání vyvažovacích algoritmů:
  - efektivita a režie vlastního algoritmu
  - komunikační složitost:
    - přenos detailních popisů procesů
    - přenos globálního stavu (zátěž, . . .)
  - aktualizace údajů

Distribuovaná správa souborů

Identifikační a adresářové služby

Adresářový modul, Kontrola přístupu, Umisťovací modul, Lokační

Middleware

Replikační služby

Konzistence dat, Multiple copy update

Transakční služby

Transakční operace, Modul zotavení, Kontrola konkurence

Transakce

Souborové služby

Přístup k souborům, Přístup k atributům

Blokové a diskové služby

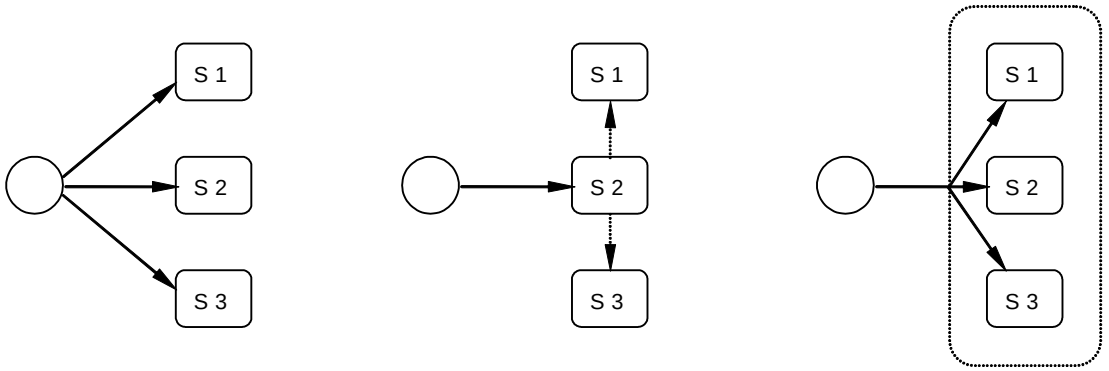
Modul správy bloků, Ovladače zařízení, Caching

Operační systémy

Replikace

- replikace souborů - udržování více kopií na více fileserverech
  - spolehlivost *(reliability)* - při havárii serveru nejsou ztracena data
  - dostupnost *(availability)* - k souborům lze přistupovat i při výpadku serveru
  - výkon *(performance)* - lze přistupovat k nejbližším datům, rozdělení výkonu
- explicitní replikace
  - 'uživatel' se sám stará o udržování konzistence
- odložená replikace
  - zápis do primární repliky, aktualizace sekundárních
- skupinová komunikace
  - zápisy simultánně zasílány všem dostupným replikám

typicky knihovny



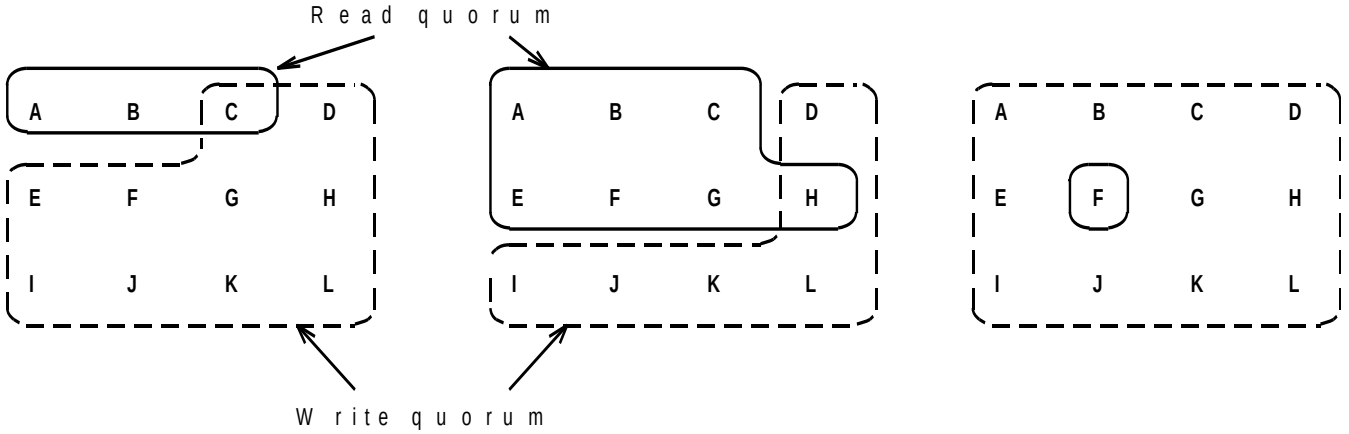
Aktualizační protokoly

- Problém aktualizací kopií
  - primární kopie
  - většinové hlasování *(majority voting)*
    - Nr > N/2
    - Nw > N/2

speciální případ váženého hlasování

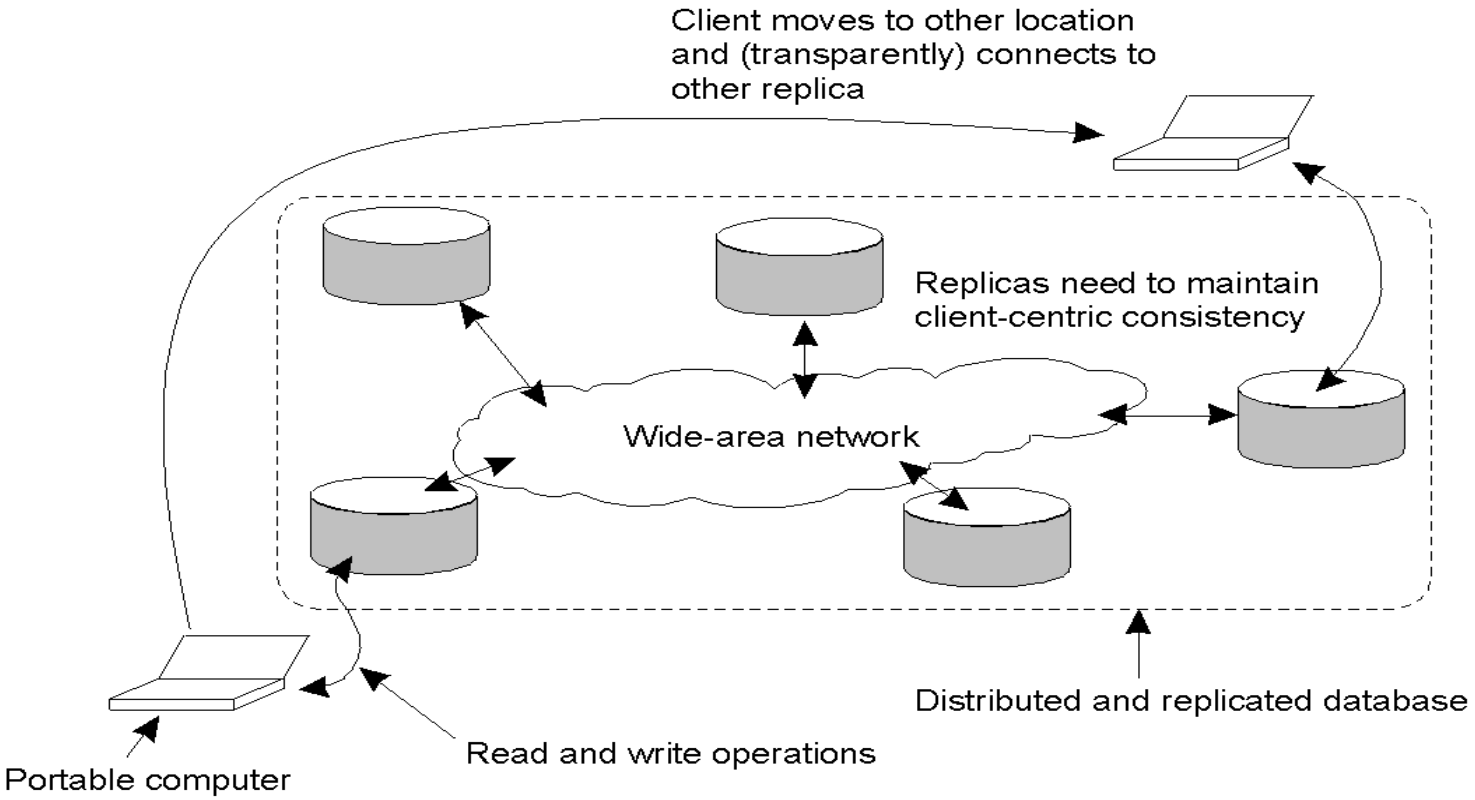
Aktualizační protokoly

- Problém aktualizací kopií
  - primární kopie
  - většinové hlasování *(majority voting)*
  - vážené hlasování *(weighted voting)*
    - read / write quorum: Nr + Nw > N, 2 * Nw > N
      - *většinové hlasování ≈ vážené při Nr = Nw*
    - optimalizace čtení zápis
    - problém při *'totální optimalizaci'* (Nr=1) - potenciální nemožnost zápisu
  - hlasování s duchy *(voting with ghosts)*
    - bezdatový (dummy, ghost) server, obsahuje pouze verze, žádná data
    - neúčastní se hlasování o čtení, pouze o zápisu
  - dynamická kvóra



Klientocentrické konzistenční modely

- Konzistenční modely DSM - pohled na sdílená data různými procesy
- Replikovaná databáze
  - ♦ zápisy málo časté, masivně paralelní čtení
  - ♦ není potřeba vzájemná synchronizace klientů
  - ♦ WWW + cache, mobile computing, News, ...
- Klientocentrický model - pohled na replikovaná data jedním procesem



Eventuální konzistence

- Eventual consistency

> Po ukončení všech zápisů budou všechny repliky v konečném čase aktualizovány

- Problém: pohled jednoho procesu na data různých replik

Značení:

x_i , $x_i[t]$	hodnota proměnné x na replice L_i (v čase t)
$W(x_i)$	prvotní zápis hodnoty x_i
$S(x_i)$, $S(x_i[t])$	posloupnost operací na L_i vedoucí k hodnotě x_i
$S(x_i;x_j)$	posloupnost x_i předchází x_j , $S(x_i) \subset S(x_j)$

A, B - připojení jednoho klienta k různým replikám

A:	S(x1)		R(x1)		
B:		S(x2)		R(x2)	S(x1;x2)

Příklad: při připojení k jedné replice uživatel vidí zprávy, po připojení k jiné replice některé zprávy (které již viděl) 'ještě' nevidí

Monotonní čtení

- Monotonic read consistency

Po přečtení hodnoty x všechna další čtení vrátí stejnou nebo novější hodnotu

A: S(x1) R(x1)
B: S(x1;x2) R(x2)

Vyhovuje monotónnímu čtení

A: S(x1) R(x1)
B: S(x2) R(x2) S(x1;x2)

Nevyhovuje monotónnímu čtení

Příklad: při připojení k jiné replice uživatel vidí všechny dosud přečtené zprávy

Monotonní zápis

- Monotonic write consistency

Zápis proměnné je proveden před jakýmkoliv následným zápisem této proměnné

A: W(x1)

B: S(x1) W(x2)

Vyhovuje monotónnímu zápisu

A: W(x1)

B: W(x2)

Nevyhovuje monotónnímu zápisu

Příklad: CVS commit na různých replikách

Čtení vlastních zápisů

- Read your writes consistency

Zápis proměnné je proveden před jakýmkoliv následným čtením této proměnné

A: W(x1)
B: S(x1;x2) R(x2)

Vyhovuje č.v.z.

A: W(x1)
B: S(x2) R(x2)

Nevyhovuje č.v.z.

Příklad: po aktualizaci webové stránky si neprohlížím kopie z cache

Zápisy následují čtení

- Writes follow reads consistency

Zápis proměnné po předchozím čtení této proměnné je proveden na stejné nebo novější hodnotě

A: S(x1) R(x1)
B: S(x1;x2) W(x2)

Vyhovuje z.n.č.

A: S(x1) R(x1)
B: S(x2) W(x2)

Nevyhovuje z.n.č.

Příklad: zápis odpovědi do newsgroups se provede tam, kde je i přečtená hodnota

Naïvní implementace

- každému zápisu je přiřazen globálně jednoznačný identifikátor WID
  - přiřazuje replika kde byl zápis proveden klientem: WID = repl_id + loc_id
- každý klient udržuje dvě množiny identifikátorů: read-set, write-set
- Monotónní čtení
  - při čtení replika serveru ověří podle read-set aktuálnost svých zápisů
    - při chybějících zápisech provede synchronizaci nebo forwarduje čtení
  - po čtení si klient aktualizuje read-set podle repliky ze které četl
- Monotónní zápis
  - při zápisu replika ověří podle write-set aktuálnost svých zápisů
    - chybějí-li nějaké, zapíše je
  - po zápisu si klient aktualizuje write-set
- Čtení vlastních zápisů
  - při čtení replika ověří podle write-set aktuálnost svých zápisů
    - jiné možné řešení - forward čtení na aktuální repliku
- Zápisy následují čtení
  - aktualizace repliky podle read-set
  - aktualizace read-set i write-set klienta

Problém: neomezený růst read/write set

Efektivnější implementace

- Problém: read-set i write-set neomezeně rostou
- Možné řešení: seskupení do relací (session)
  - typicky vázané na aplikaci nebo modul
  - při ukončení / restartu smazání množin
  - neřeší problém trvale běžících aplikací
- Jiné řešení: reprezentace množin - vektorové hodiny (Bayou)
  - replika serveru při zápisu přiřadí WID a lokální TS(WID)
  - každá replika Si udržuje RCV(i)[j] - TS poslední operace zápisu přijatá Si od Sj
  - při přijetí žádosti o čtení nebo zápis replika vrátí aktuální RCV(i)
  - read-set i write-set jsou reprezentovány vektorovými hodinami
  - obecná pravidla:
    - VT(A)[i] = max TS operací z A iniciovaných na replice Si
    - sjednocení: VT(A+B)[i] = max(VT(A)[i], VT(B)[i])
    - test podmnožiny: $A \subseteq B \Leftrightarrow VT(A) \leq VT(B)$
  - po přijetí TS si klient aktualizuje read-set nebo write-set
    - čtení: VT(RS)[j] = max(VT(RS)[j], RCV(i)[j]) $\forall j$
    - zápis: VT(WS)[j] = max(VT(WS)[j], RCV(i)[j]) $\forall j$
  - read/write-set reprezentuje poslední operace zápisu, které klient viděl/zapisoval

Epidemické protokoly

Implementace eventuální konzistence

Optimalizace komunikace ve **VELMI** rozsáhlých systémech, neřeší konflikty

Teorie epidemií - infekční nákaza

- rozšíření infekce co nejrychleji na co největší počet uzlů

Antientropie - server P náhodně vybere server Q k výměně dat

- možné výměny:

push

$P \rightarrow Q$ při velkém počtu infikovaných uzlů malá pravděpodobnost rozšíření

pull

$P \leftarrow Q$ zpočátku pomalé rozšiřování, nakonec infekce celé množiny

push/pull

$P \leftrightarrow Q$ lze spojit výhody předchozích postupů

Epidemické protokoly

Implementace eventuální konzistence

Optimalizace komunikace ve VELMI rozsáhlých systémech, neřeší konflikty

Teorie epidemií - infekční nákaza

- rozšíření infekce co nejrychleji na co největší počet uzlů

Antientropie - server P náhodně vybere server Q k výměně dat

- možné výměny:
  - push - při velkém počtu infikovaných uzlů malá pravděpodobnost rozšíření
  - pull - zpočátku pomalé rozšiřování, nakonec infekce celé množiny
  - push/pull - lze spojit výhody předchozích postupů
- push problém: kdy má uzel přestat infikovat

Gossiping

- při nákaze infikovaného uzlu se s pravd. 1/k uzel uvede do klidového stavu
- oblíbená kombinace: Gossiping + periodický Pull

- výhoda epidemických protokolů: rozšiřitelnost
- složitější topologie - hierarchie, rychlejší infekce

Epidemické protokoly

Problém mazání dat

- naivní implementace: smazání dat → smazání všech informací o datech
  - důsledek: entropie jiným kanálem data zase obnoví
- vylepšení: certifikát smrti (death certificate) - záznam o smazání
  - problém: neomezený nárůst certifikátů o historických datech
- řešení: v konečném čase certifikát zanikne
  - podmínka: konečná doba rozšiřování infekce

Aplikace - agregace dat

- *uzly, spočítejte se!*
- podproblém: spočítejte průměr
  - uzel i: x_i = iniciální (libovolná) hodnota
  - epidemicky: $(x_i, x_k) = (x_i + x_k) / 2$
  - $x_i \rightarrow avg_{\forall n}(x_n)$
- iniciátor $x_i = 1$, ostatní $x_i = 0$
  - $x_i \rightarrow 1 / N$

Další zajímavé protokoly

- Distribuční protokoly
  - kolik a kde umístit repliky
  - permanent, server-initiated, client-initiated
- Další konzistenční protokoly
  - primary-based - remote-write, local-write
  - replicated-write - active replication
  - cache coherence
  - lazy replication, causal consistency, ...
- Živý výzkum v oblasti velmi rozsáhlých systémů

Trendy '94 ..... 18 years ago

- Nový hardware
  - ☺ síť ≈ rychlá sběrnice, multicomputery ≈ multiprocesory
  - ☺ rychlé optické kabely - přenos po síti rychlejší než na disk
  - ☺ velké disky - WORM, zapisovatelné, ...
  - ☹ velká paměť - mapované soubory, nestránkování
  - ☹ neblokové soubory - append v paměti rychlejší
- Globální systémy
  - 😐 algoritmy - centralizované / hierarchické / distribuované, masivní škálovatelnost
  - ☺ lokalizace - znaková sada, ikony, národní prostředí
  - ☺ potřeba datových dálnic
  - ☺ problémy s prostorem jmen - strom příliš košatý
  - 😐 globální IS - VLDB
- Mobilní uživatelé
  - 😐 konektivita v letadle
  - ☺ ochrana dat a osobních údajů
- ...

Mooreův zákon:

$1.8^{18} \approx 40\ 000$

to be continued ...

- NSWI080 Middleware
  - Petr Tůma
  - CORBA, DCE, DCOM, EJB, JMS, RMI, JavaSpaces, MQ, ..., ..., ...
- NPRG042 Programování v paralelním prostředí
  - Jakub Yaghob
  - OpenMP, Cluster Open MP, Threading Building Blocks, MPI
- NSWI152 Vývoj cloudových aplikací
  - Filip Zavoral, Jaroslav Keznikl, Tomáš Pop
  - Goggle App Engine, Amazon Elastic Cloud, Windows Azure