
Principy distribuovaných systémů

NSWI035 ZS 2018/19

RNDr. Filip Zavoral, Ph.D.

Katedra softwarového inženýrství

<http://ksi.mff.cuni.cz/lectures/NSWI035>

1. Úvod

- ◆ motivace, funkce, architektury
- ◆ komunikace, spolehlivost, RPC, skupinová komunikace

2. Synchronizační algoritmy

- ◆ fyzické a logické hodiny, vyloučení procesů, volba koordinátora
- ◆ kauzální závislost, doručovací protokoly, virtuální synchronie

3. Konsensus

- ◆ detekce globálního stavu, armády a generálové
- ◆ Paxos, RAFT, etc

4. Distribuovaná sdílená paměť

- ◆ konzistenční modely, distribuované stránkování

5. Správa prostředků a procesů

- ◆ zablokování, distribuované algoritmy detekce
- ◆ vzdálené spouštění procesů, vyvažování zátěže, migrace

6. Replikace a konzistence

- ◆ replikace, aktualizační protokoly
- ◆ klientocentrické konzistenční modely, epidemické protokoly

- **Pokryvá většinu přednášené (a zkoušené) látky**
 - Tanenbaum, van Steen: Distributed Systems - Principles and Paradigms, 2nd ed
 - Chow, Johnson: Distributed Operating Systems & Algorithms
 - *není pokryto:* Paxos, Raft
 - wikipedia, články, videa z přednášek
- **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: Distributed Systems, 2nd ed
- slajdy ani velmi historické 'učební texty' ze studnic (*nepoužívat!*) nejsou skripta

- Tůma: **NSWI080 Middleware** [LS]
 - RMI, Sun RPC, .NET Remoting, CORBA, DCE, DCOM, SOAP, WSDL, UDDI, MQSeries, JMS, DDS, MPI, CAN, Chord, Pastry, JavaSpaces, EJB, OSGi, ...
- Bednárek, Yaghob, Zavoral: **NSWI150 Virtualizace a Cloud Computing** [ZS]
 - moderní aplikace principů distribuovaných systémů
- Yaghob, Kruliš: **NPRG042 Programování v paralelním prostředí** [LS]
 - jiný způsob řešení 'výkonného počítání'
- Kruliš: **NPRG058 Pokročilé programování v paralelním prostředí** [ZS]
 - nové architektury, GPGPU



“Definice”

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

**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

**je systém propojení množiny nezávislých uzelů,
který poskytuje uživateli dojem jednotného systému**

Hardwareová 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 uzelů, topologii, komunikaci apod.

distribuovaný systém - celá škála různých řešení

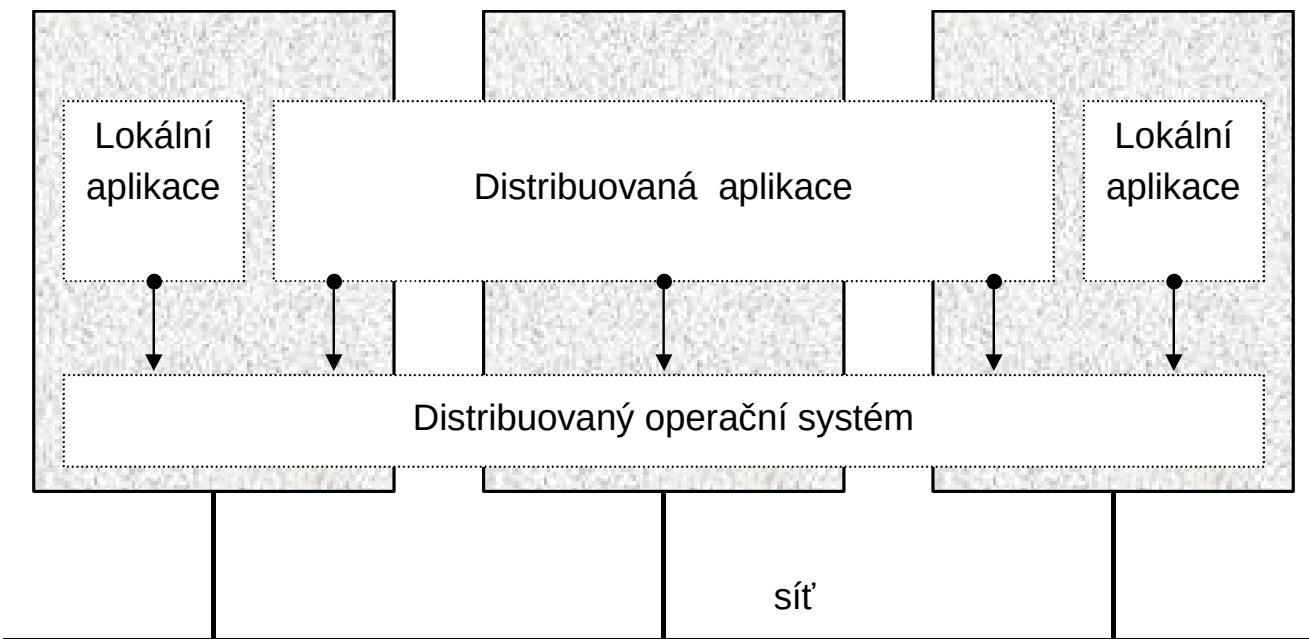
- ◆ multiprocesory na jedné základové desce
- ◆ 'celosvětový' systém pro miliony uživatelů
- ◆ moderní technologie: cluster, grid → cloud

Zaměření přednášky

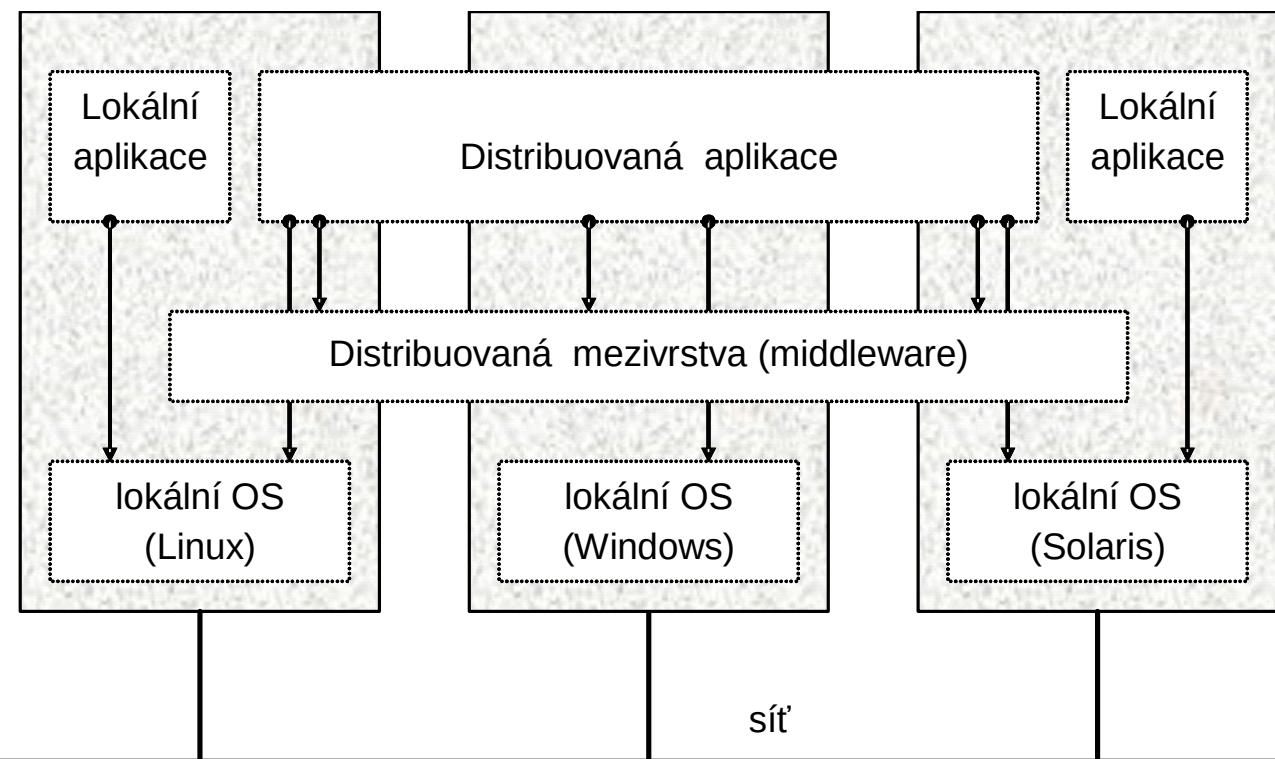
- ◆ principy, algoritmy, protokoly, synchronizace
- ◆ uzel = běžný počítač, běžná síť, softwarový systém

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

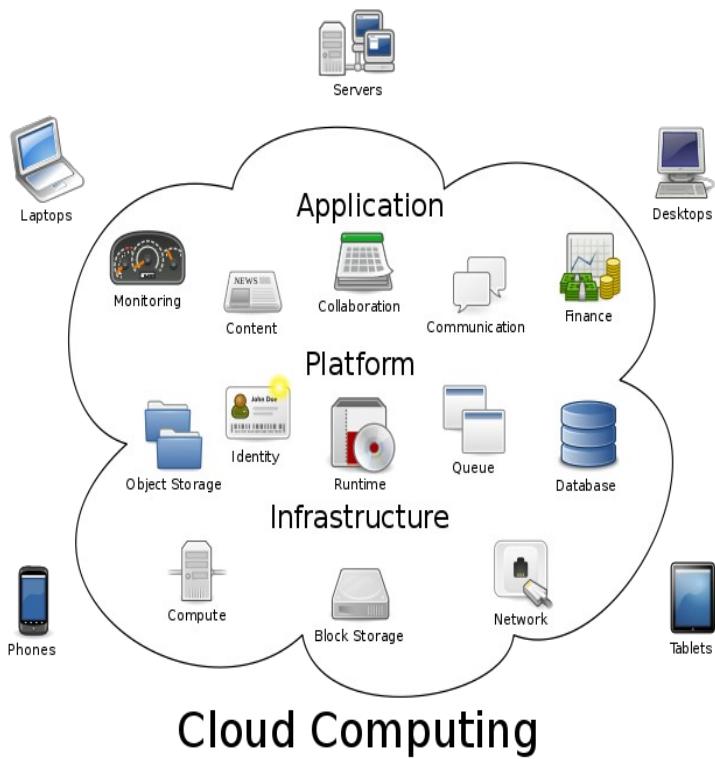


- softwarový průmysl - jiný směr, dominance Win/Ux
- vyžití principů a mechanismů distribuovaných systémů
- lokální OS, rozšiřující vrstva - distribuované prostředí + další služby
- prostředí pro distribuované aplikace - **middleware** (OSF DCE, CORBA, DCOM, Globe, ..)
- využití pro cluster / grid computing



201x - Cloud computing, XaaS

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

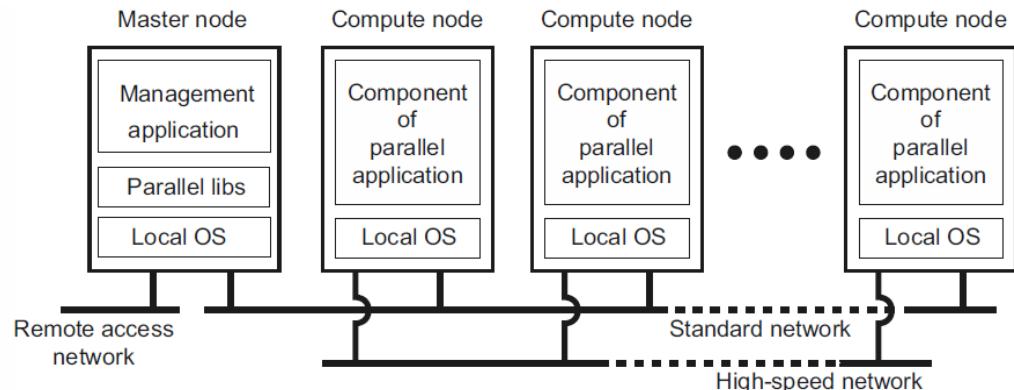


	Virtual Machines	Web Sites	Cloud Services
Data Management	SQL Database	Tables	Blobs
Networking	Virtual Network	Connect	Traffic Manager
Business Analytics	SQL Reporting	Hadoop	
Messaging	Queues	Service Bus	
Caching	Caching	CDN	
Identity	Windows Azure Active Directory		
High-Performance Computing	HPC Scheduler		
Media	Media Services		
Commerce	Marketplace		
SDKs	.NET	Java	PHP
	Python	Node.js	

NSWI150

High Performance Distributed Computing

- Cluster Computing
 - homogenní hw/OS, LAN
 - shared memory, DSM, MQ
 - batch \Rightarrow single system image
 - middleware

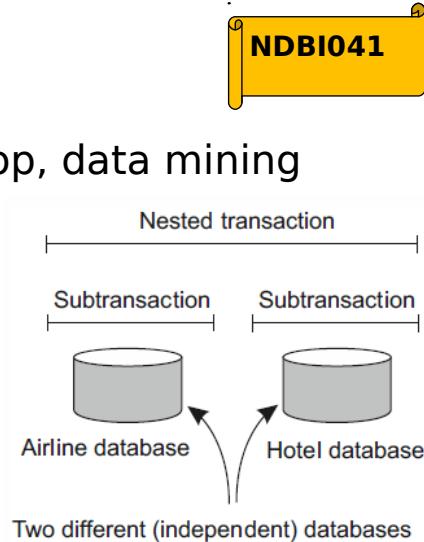


- Grid Computing
 - heterogenní hw/sw, WAN
 - odlišné algoritmy a protokoly
 - virtuální organizace - práva
 - fabric / connectivity / resource / application layer
 - service-oriented, web services
 - *OSGA - Open Grid Services Architecture*
- Cloud Computing
 - resource providers, utility computing
 - IaaS, PaaS, SaaS, ... *aaS
 - high availability - distributed protocols

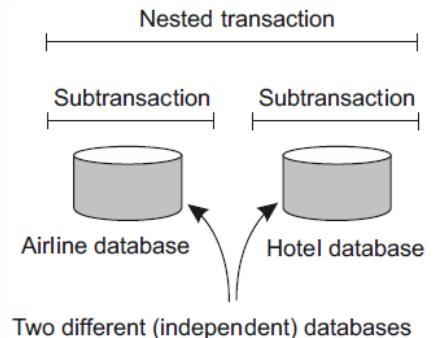
časté
vzájemné
propojení

- Distributed Information Systems
 - Distributed Data Processing
 - data-intensive computing, Big Data, NoSQL databáze, Hadoop, data mining
 - Distributed Transaction Processing
 - distribuované transakce, vnořené transakce, ACID
 - DB systémy, enterprise aplikace
 - RPC, transactional RPC
 - transakční monitor
 - Enterprise Application Integration
 - RPC, RMI / message-oriented middleware, ESB, publish-subscribe
 - tightly / loosely coupled communication
- Pervasive Systems
 - dynamická organizace
 - typicky velký a měnící se počet 'malých' uzelů
 - mikrouzly, senzory, sci-fi house, samoorganizace, kolektivní znalost a rozhodování
 - distribuovaní agenti, inteligentní brouci, inteligentní plíseň
- Peer-to-peer systems
 - DHT, file sharing, P2P computing

NDBI041



- Distributed Information Systems
 - Distributed Data Processing
 - data-intensive computing, Big Data, NoSQL databáze, Hadoop, data mining
 - Distributed Transaction Processing
 - distribuované transakce, vnořené transakce, ACID
 - DB systémy, enterprise aplikace
 - RPC, transactional RPC
 - transakční monitor
 - Enterprise Application Integration
 - RPC, RMI / message-oriented middleware, ESB, publish-subscribe
 - tightly / loosely coupled communication
- Pervasive Systems
 - dynamická organizace
 - typicky velký a měnící se počet 'malých' uzlů
 - IoT, mikrouzly, senzory, sci-fi house, samoorganizace, kolektivní znalost a rozhodování, distribuovaní agenti, inteligentní brouci, inteligentní plíseň, ...
- Peer-to-peer systems
 - DHT, file sharing, P2P computing



Motivace

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

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 API, komunikace mezi uzly, komunikace s uživatelem
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
 - *uzel:cesta/soubor* vs *služba:namespace/soubor*

■ Exekuční / Migrační

- ◆ procesy mohou běžet na libovolném uzlu / přemisťovat se mezi uzly

■ Replikační

- ◆ uživatel se nemusí starat počet a aktualizaci kopii 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?
- ◆ využití moderního hw



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%
→ nespolehlivost čtyř serverů $0.01^4 = 0.00000001$
- Leslie Lamport: distribuovaný systém je ...
"systém, který pro mě odmítá cokoliv vykonávat,
protože počítač, o kterém jsem nikdy neslyšel, byl náhodně vypnut"

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

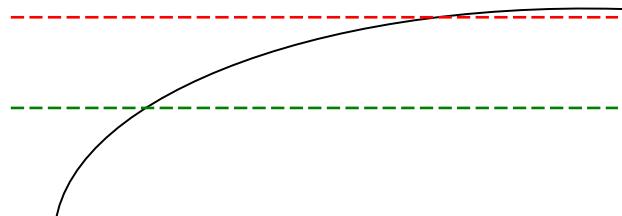
Spolehlivost

- Teorie: nespolehlivost jednoho serveru 1%
→ 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 vypnul"
- High Availability - distribuovaný konsensus



Škálovatelnost / Scalability

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

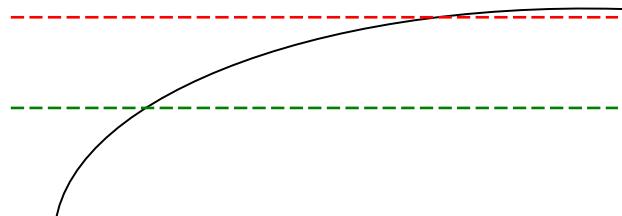


Výkonnost

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

Chyby návrhu distribuovaných systémů

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



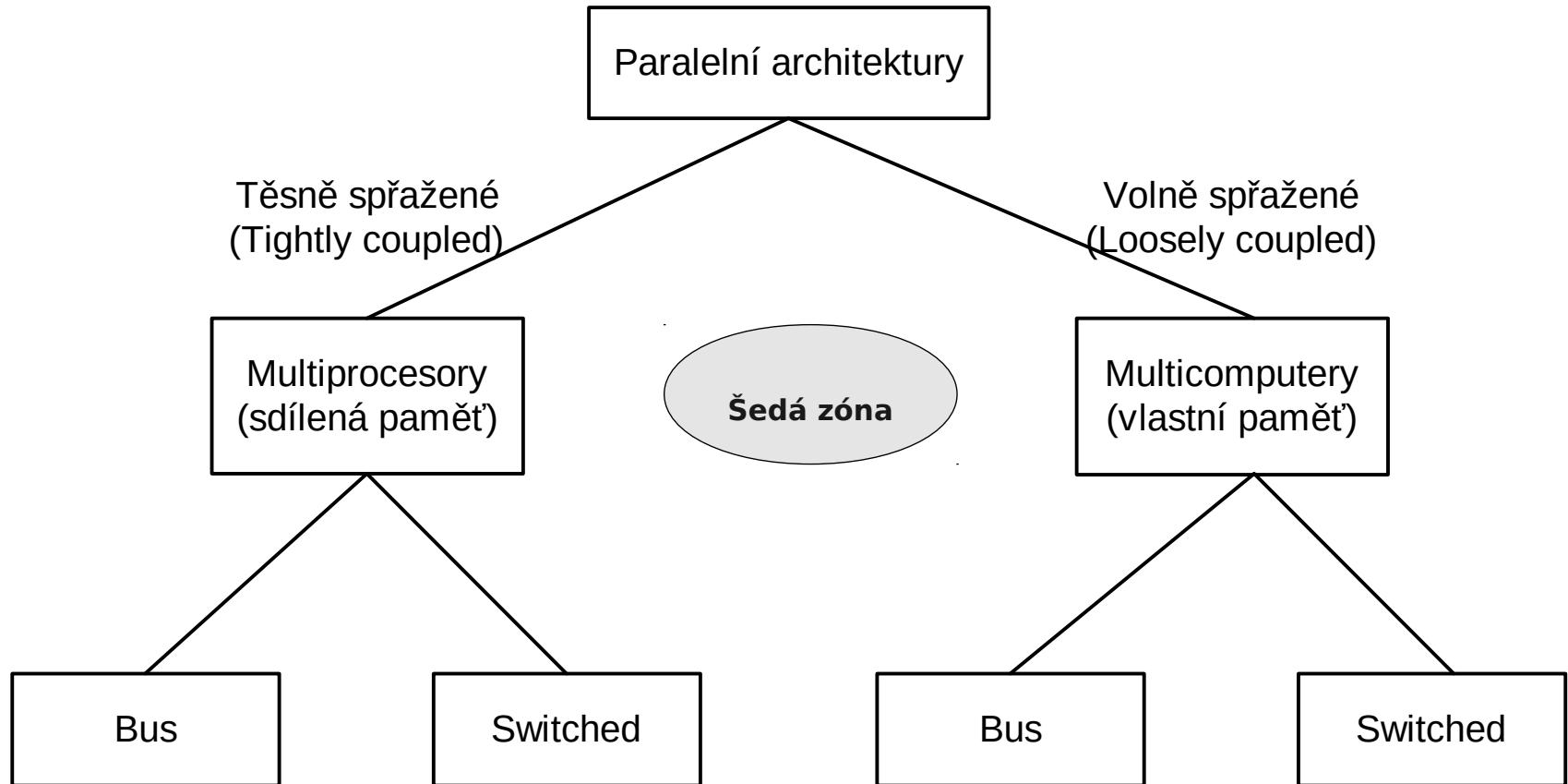
Výkonnost

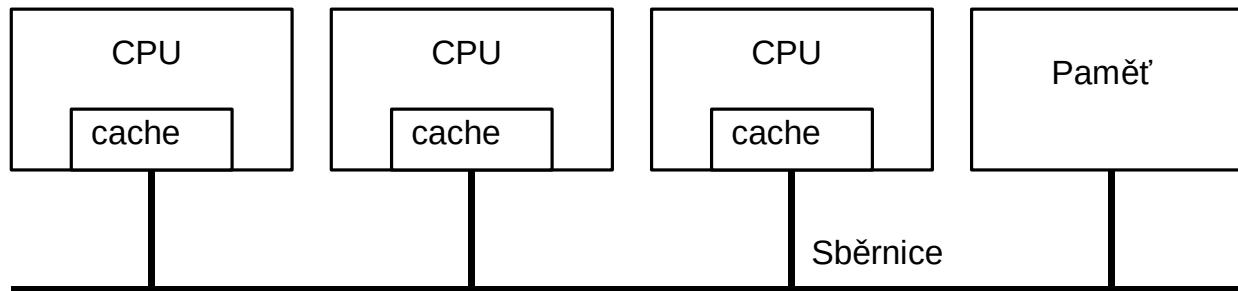
- Teorie: více uzlů ⇒ vyšší výkon
- Praxe: výrazně nižší než lineární nárůst výkonu
 - jednodušší architektura, menší synchronizace ⇒ nižší režie
- Příčiny: komunikace, synchronizace, komplikovaný software
- Granularita výpočetní jednotky

Chyby návrhu distribuovaných systémů

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

Paralelní architektury

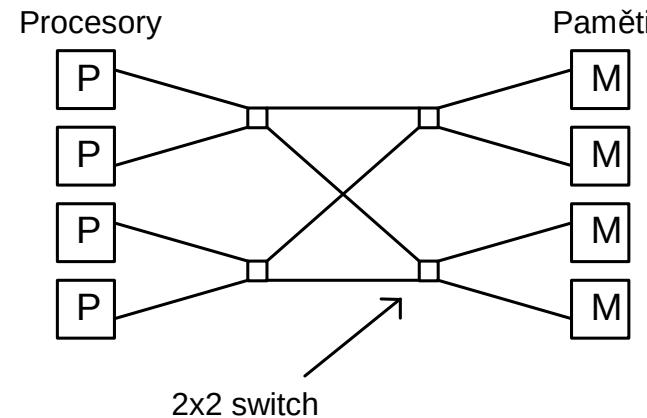
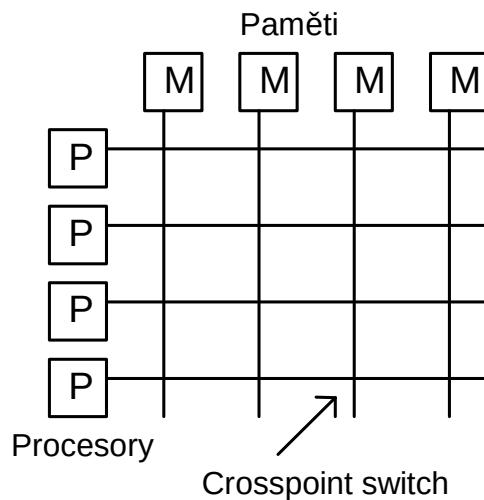




Sběrnicová architektura

- ◆ sdílená sběrnice mezi procesory a pamětí
- ◆ levná, běžně dostupná
- ◆ velmi slabá škálovatelnost
 - max. desítky uzelů
- ◆ cache – synchronizace

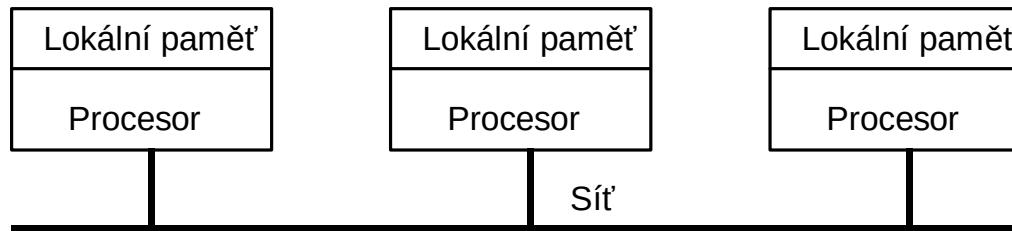
Multiprocesory - přepínačová architektura



Přepínačová (switched) architektura

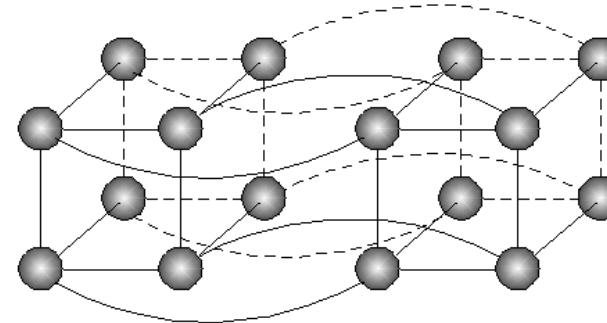
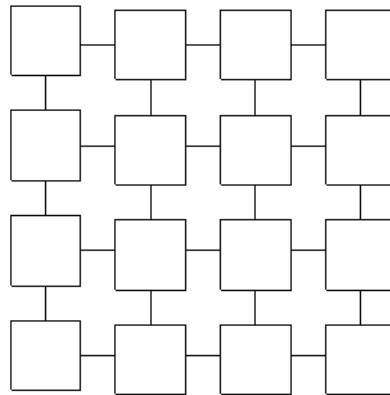
- ◆ Možné zapojení i většího počtu uzlů
- ◆ Mřízka - crosspoint switch
 - sběrnicová mřízka - procesory × paměťové bloky
 - nákladné - kvadratický počet přepínačů
- ◆ Omega network
 - postupně přepínaná cesta mezi procesorem a pamětí
 - při větším počtu úrovní pomalé
 - počet přepínačů $n * \log n$

nákladné
nízká škálovatelnost
⇒ NUMA



Sběrnicová architektura

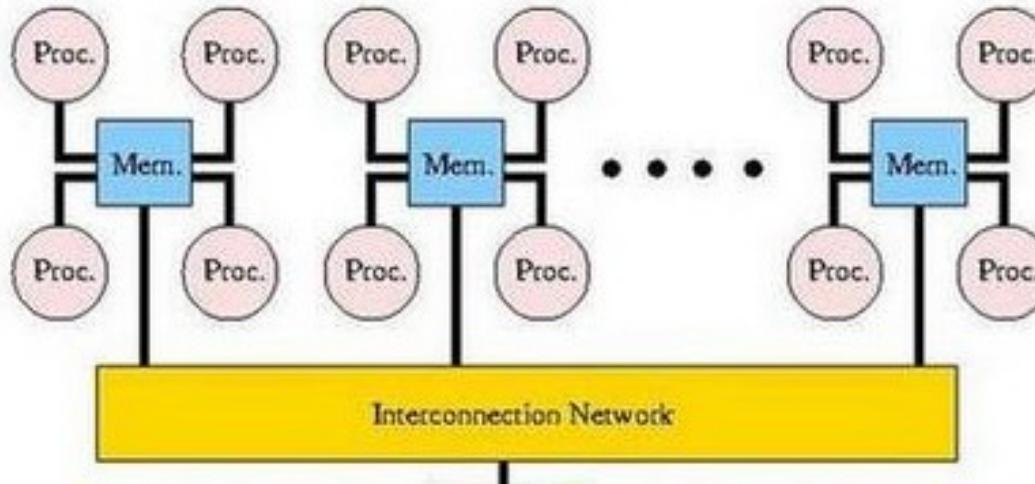
- ◆ vlastní paměť každého uzlu → výrazně nižší komunikace
 - složitější synchronizace
- ◆ teoreticky neomezený počet uzlů
- ◆ levná, běžně dostupná
 - 'normální' počítače propojené 'normální' sítí
 - datacentra - jiné provedení: processor pool, blade, ...
- ◆ cluster => grid



Přepínačová (switched) architektura

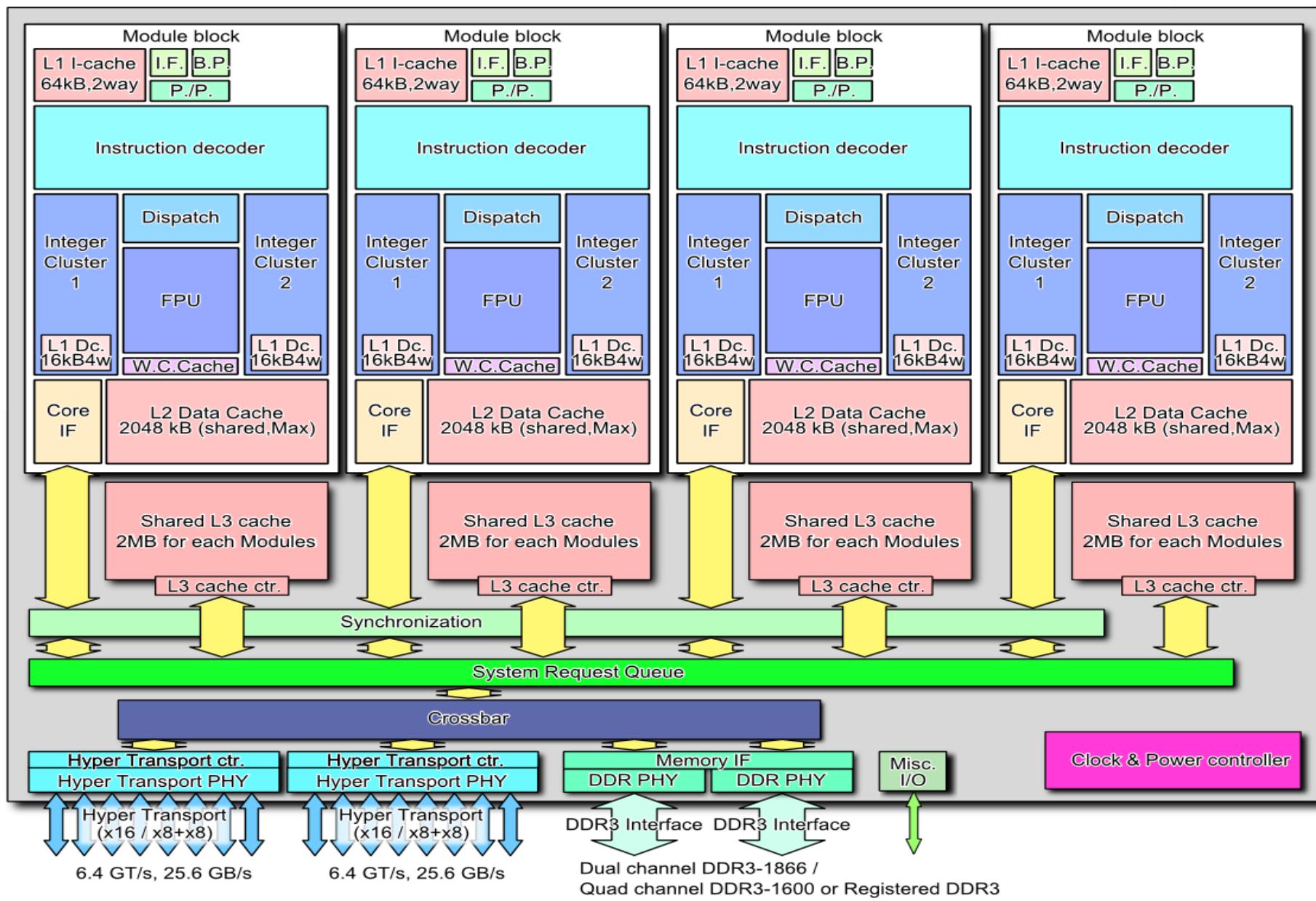
- ◆ Mřízka
 - relativně jednoduchá (dvourozměrná) implementace
 - vhodné pro řešení 'dvourozměrných' problémů - grafy, analýza obrazu, ...
- ◆ Hyperkrychle
 - n-rozměrná krychle, častý rozměr 4
 - každý uzel přímo propojen se sousedy ve všech rozměrech
 - supercomputers - desetitisíce až miliony uzel
 - 2018 IBM Summit - 9K Power9 CPU, 27K Tesla GPU, 200 petaflops, 250 PB RAM
 - 2015 Salomon - TI Ostrava - 52K cores, 14 TB RAM
 - » v době spuštění 40. na světě, 2018 vypadl z top500

Non Uniform Memory Access



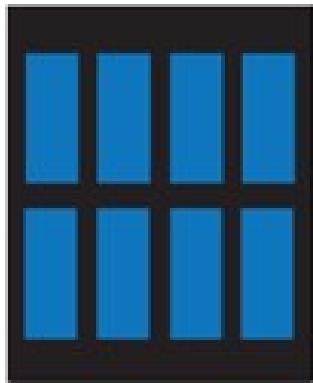
- S-COMA - Simple Cache-Only Memory Architecture
 - ◆ požadavek na nelokální data je propagován vyšší vrstvě
- ccNUMA - Cache Coherent NUMA
 - ◆ globální adresace, konzistence cache
- NUMA-faktor - rozdíl latence lokálních a vzdálených paměťových přístupů
 - ◆ NUMA-aware processing / scheduling

ccNUMA Bulldozer Architecture

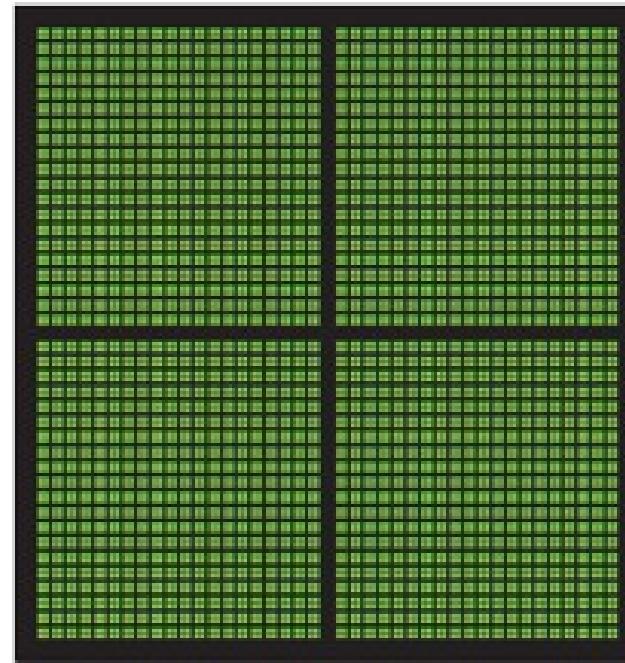


General-purpose computing on graphics processing units

- Moderní dostupný HW
 - NVidia Tesla, Kepler, Xeon Phi, ...
- Frameworks
 - CUDA, OpenCL, C++Amp

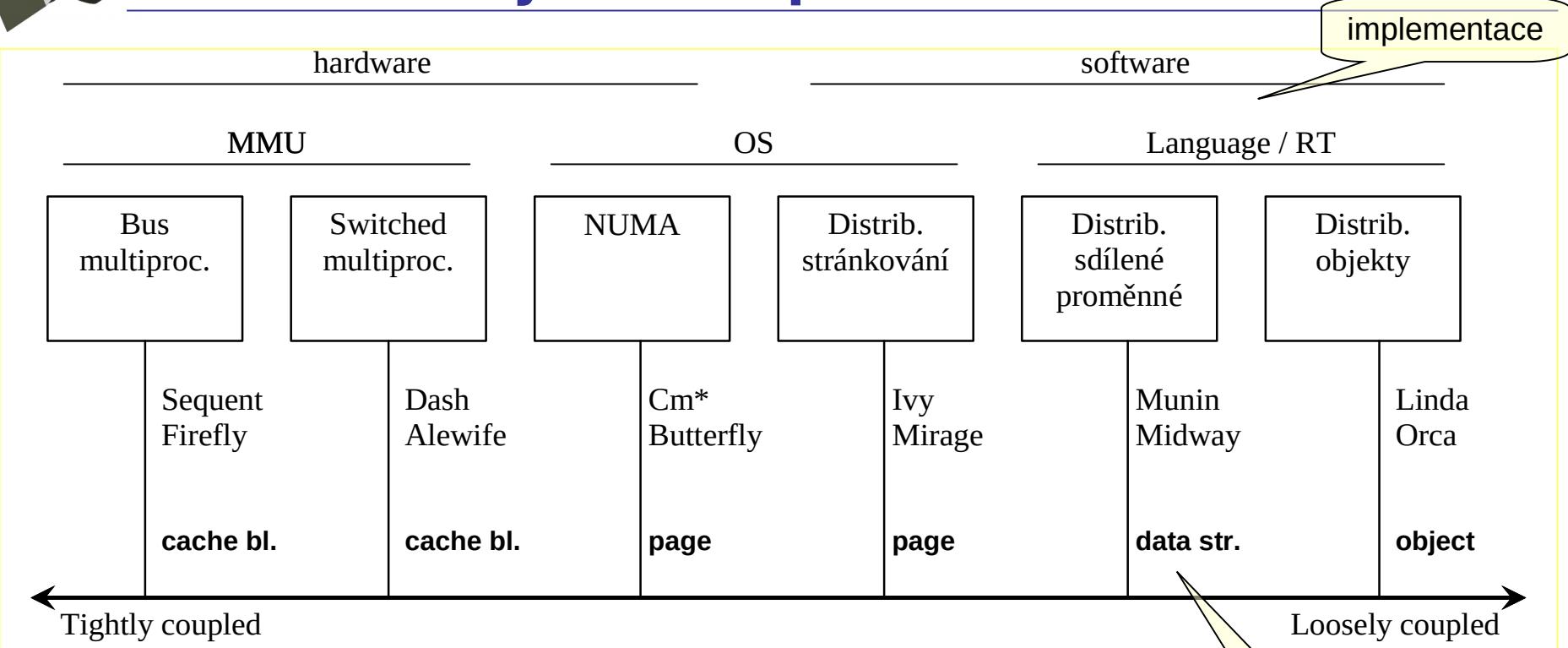


CPU
MULTIPLE CORES



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 Bennett 1990

Komunikace

Síťové protokoly
Spolehlivost komunikace
Idempotence, sémantika služeb
Havárie serveru, klienta
RPC, Message Passing
Skupinová komunikace

absence sdílené paměti → zasílání zpráv

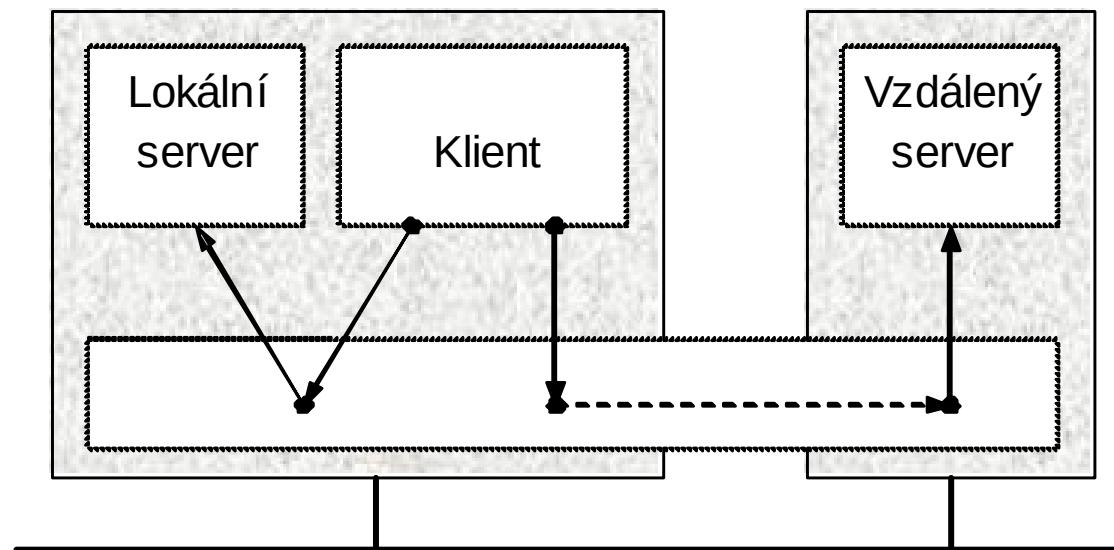
vyšší komunikační mechanismy

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

Jednotný komunikační mechanismus

klient-server model

efektivita - lokální / vzdálený přístup

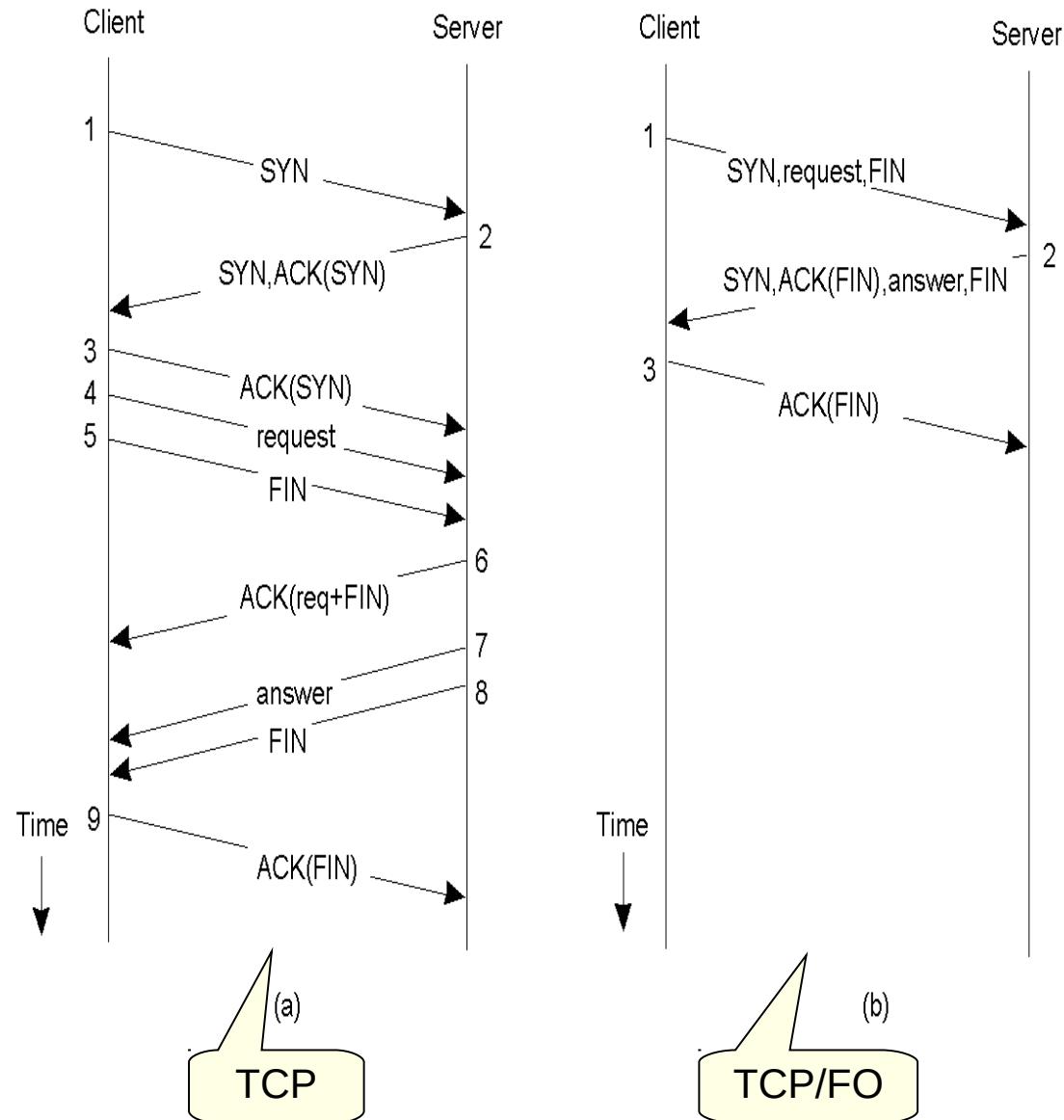


■ TCP - standard pro spolehlivý vzdálený přenos

- ◆ ale: většina komunikace v rychlé a spolehlivé LAN
- ◆ nevýhody:
 - složitost protokolu
 - vyšší latence
 - nižší propustnost

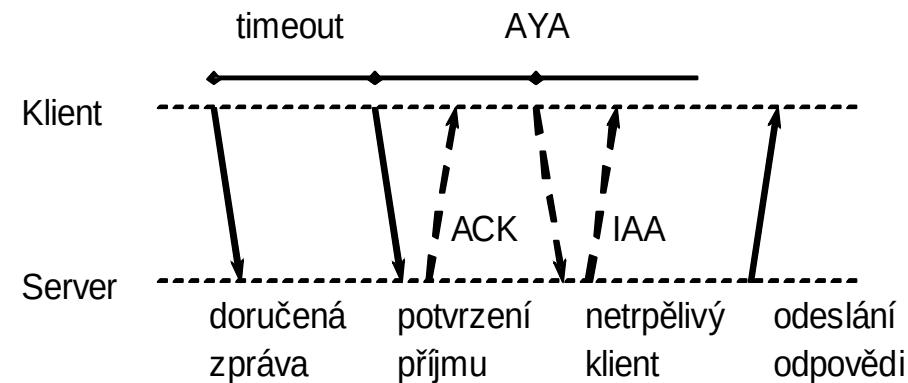
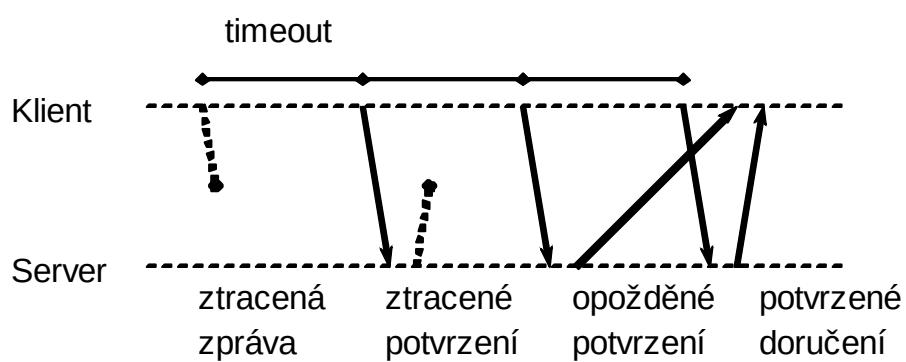
■ specializované protokoly pro (spolehlivé) lokální sítě

- ◆ optimistické
- ◆ vysoký výkon
- ◆ složitější zotavení z chyb
- ◆ T/TCP \Rightarrow TCP Fast Open
FLIP, VMTP



Zajištění spolehlivosti komunikace

- duplicita zpráv - číslování, zahzení - poslední zpráva / okno
- předbíhání zpráv - číslování, zpráva o nedoručení - režie, okno
- ztráta zprávy - klient vyslal žádost a neví
 - ◆ ztráta žádosti / ztráta odpovědi / příliš pomalá komunikace
 - ◆ typické řešení: potvrzování, timeout, opakování
- potvrzování → zvýšení režie → odpověď = potvrzení příjmu
- při delším zpracování - timeout serveru nebo klienta
- služební zprávy - ACK, NAK, AYA, IAA, CON, ...



Přenos dlouhých zpráv

■ jednotka přenosu = zpráva

- ◆ příliš dlouhá → rozdělení na pakety

■ co potvrzovat?

- ◆ každý paket?
- ◆ celá zpráva?

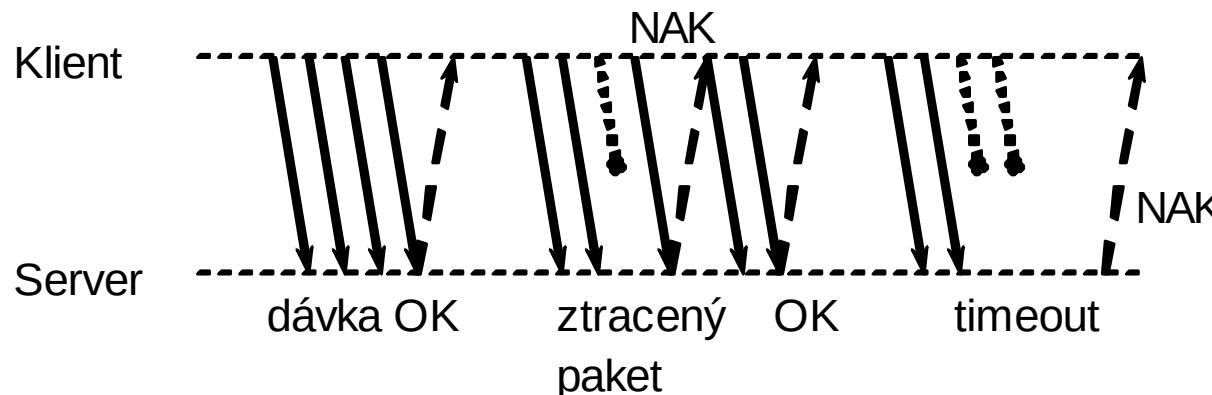
režie při správném přenosu !

režie při špatném přenosu !

Přenos dlouhých zpráv

- jednotka přenosu = zpráva
 - ◆ příliš dlouhá → rozdělení na pakety
 - co potvrzovat?
 - ◆ každý paket?
 - ◆ celá zpráva?
- režie při správném přenosu !
- režie při špatném přenosu !

- možné řešení - dávky (blast, burst, *buřty*)
 - ◆ 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 dávky

- ◆ jednodušší, ale hůře optimalizovatelný
 - vyšší zátěž sítě → velká chybovost
 - volná/spolehlivá síť → zbytečná režie

■ dynamické dávky

■ pevné dávky

- ◆ jednodušší, ale hůře optimalizovatelný
 - vyšší zátěž sítě → velká chybovost
 - volná/spolehlivá síť → zbytečná režie

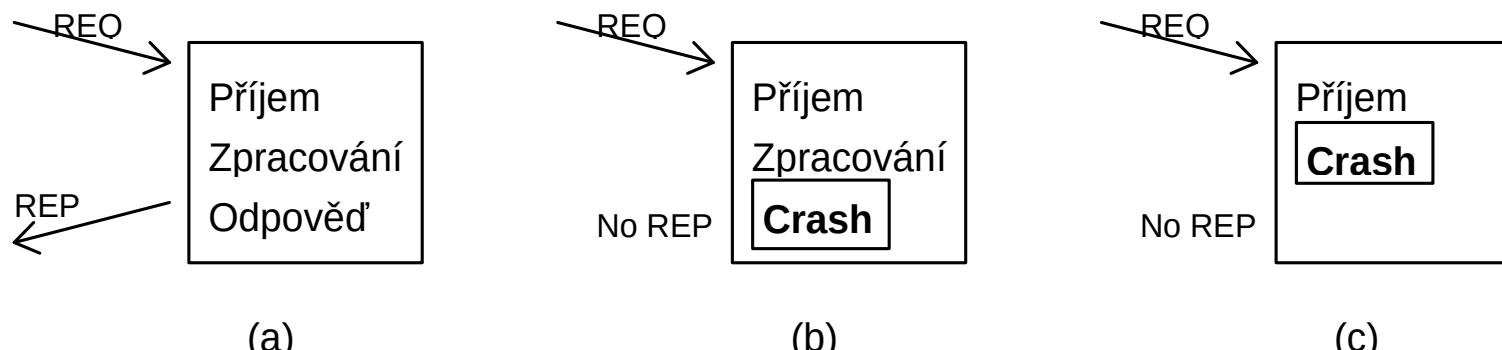
■ dynamické dávky

- ◆ úprava velikosti podle aktuální situace
 - n-krát správný přenos dávky → zvětšení
 - špatný přenos → zmenšení
- ◆ vysoká využitelnost kapacity aniž by jeden přenos omezoval ostatní
- ◆ korektnost implementace, konsensus o velikosti dávky!

(Ne)spolehlivost serveru / služeb

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 příliš náročné (transakce)



Sémanitka služeb

idempotentní služby

- ◆ nevadí opakované provedení služby
- ◆ sečti 1 + 1
- ◆ vyber 1000000 \$ z mého účtu ☺

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 neprovědla vícekrát
- ◆ idempotentní služby!

■ **at-most-once** sémantika

- ◆ služba se určitě neprověde vícekrát
- ◆ nelze zajistit, že se provede

sirotci (orphans)

typicky se neřeší

specializované dlouhotrvající výpočty

■ exterminace

■ reinkarnace

■ expirace



sirotci (orphans)

typicky se neřeší

specializované dlouhotrvající výpočty

■ exterminace

- ◆ klient po zrození sám zruší službu
- ◆ **zodpovědnost klienta**

■ reinkarnace

■ expirace



sirotci (orphans)

typicky se neřeší

specializované dlouhotrvající výpočty

■ exterminace

- ◆ klient po zrození sám zruší službu
- ◆ **zodpovědnost klienta**



■ reinkarnace

- ◆ klient po zrození nastartuje novou epochu
- ◆ server zruší všechny úlohy patřící do prošlých epoch
- ◆ **evidence na serveru, zodpovědnost klienta i serveru**

■ expirace

sirotci (orphans)

typicky se neřeší

specializované dlouhotrvající výpočty

■ exterminace

- ◆ klient po zrození sám zruší službu
- ◆ **zodpovědnost klienta**



■ reinkarnace

- ◆ klient po zrození nastartuje novou epochu
- ◆ server zruší všechny úlohy patřící do prošlých epoch
- ◆ **evidence na serveru, zodpovědnost klienta i serveru**

■ expirace

- ◆ úloha má přiděleno kvantum času
- ◆ při překročení si server vyžádá nové kvantum
- ◆ **zodpovědnost serveru, spolupráce klienta**

Vzdálené volání pomocí zpráv

```
main()
{
    struct message m1, m2;
    for(;;)
    {
        receive( &m1);
        check( &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

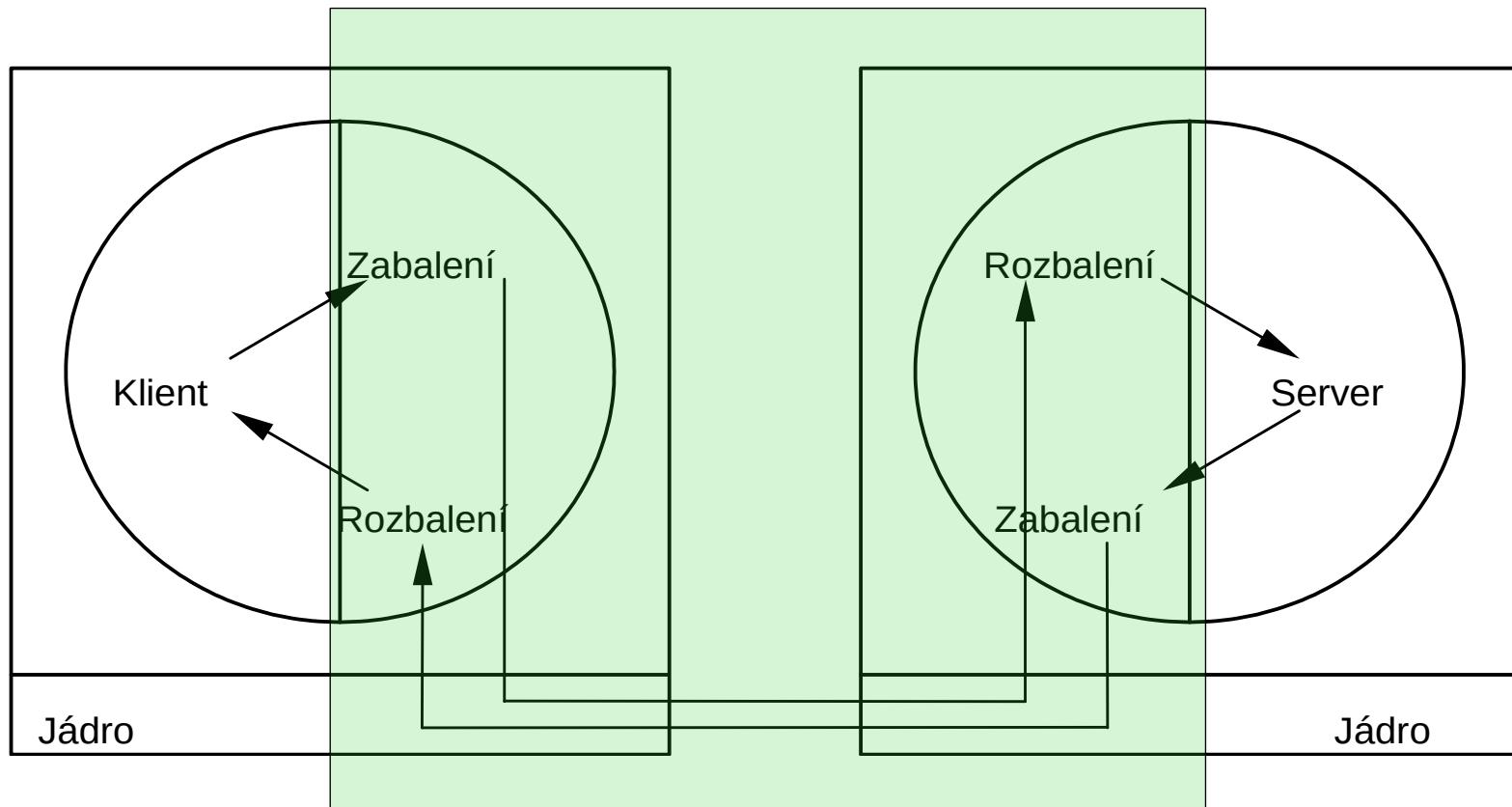
```
fnc()
{
    struct message m;
    m.opcode = SERVICE1;
    m.arg1 = ...;
    m.arg2 = ...;
    send( MY_SERVER, &m);
    receive( &m);
    if( m.retval != OK)
        error_handler( m.retval);
    process_data( &m.data);
}
```

RPC - vzdálené volání procedur

Mezi procesová komunikace příliš nízkoúrovňová

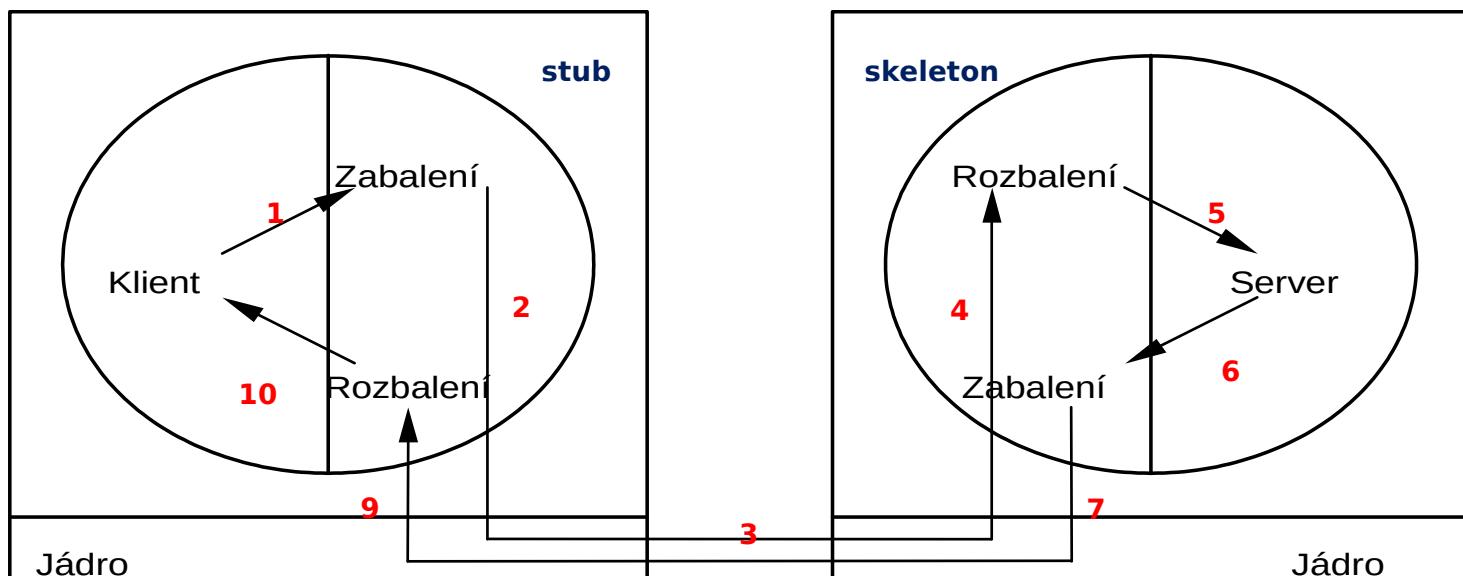
Idea: přiblížit zažitým mechanismům volání 'procedur'

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 uzlu, kde běží server
4. Vzdálené jádro předá zprávu skeletonu (*server-side stub*)
5. Skeleton rozbalí parametry a zavolá výkonnou funkci / službu
6. Server zpracuje požadavky a normálním způsobem se vrátí do skeletonu
7. Skeleton zabalí výstupní parametry do zprávy a zavolá jádro
8. Jádro pošle zprávu zpět klientu
9. Klientovo jádro předá zprávu stubu
10. Stub rozbalí výstupní parametry a vrátí se ke klientovi



Kompilace

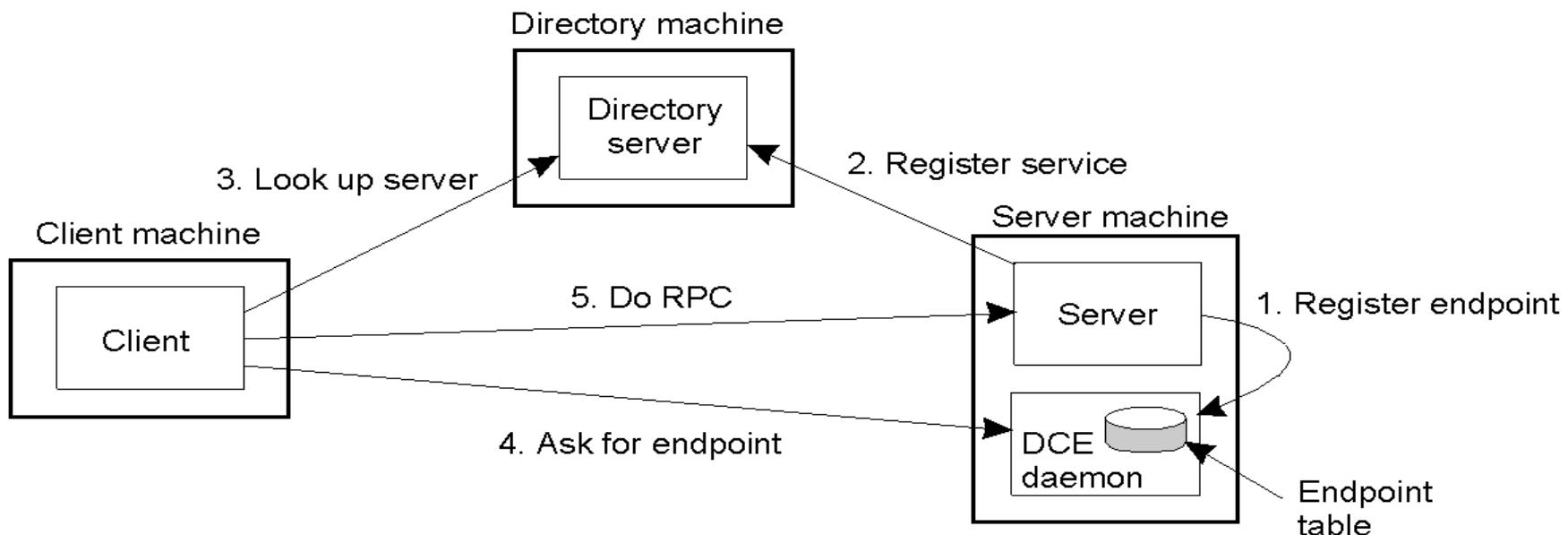
- ◆ Klient a server jsou programovány "lokálně"
- ◆ Vytvoří se stub (klient) a skeleton (server)

Spojení - odlišná životnost

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

Vyvolání

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



[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);

}

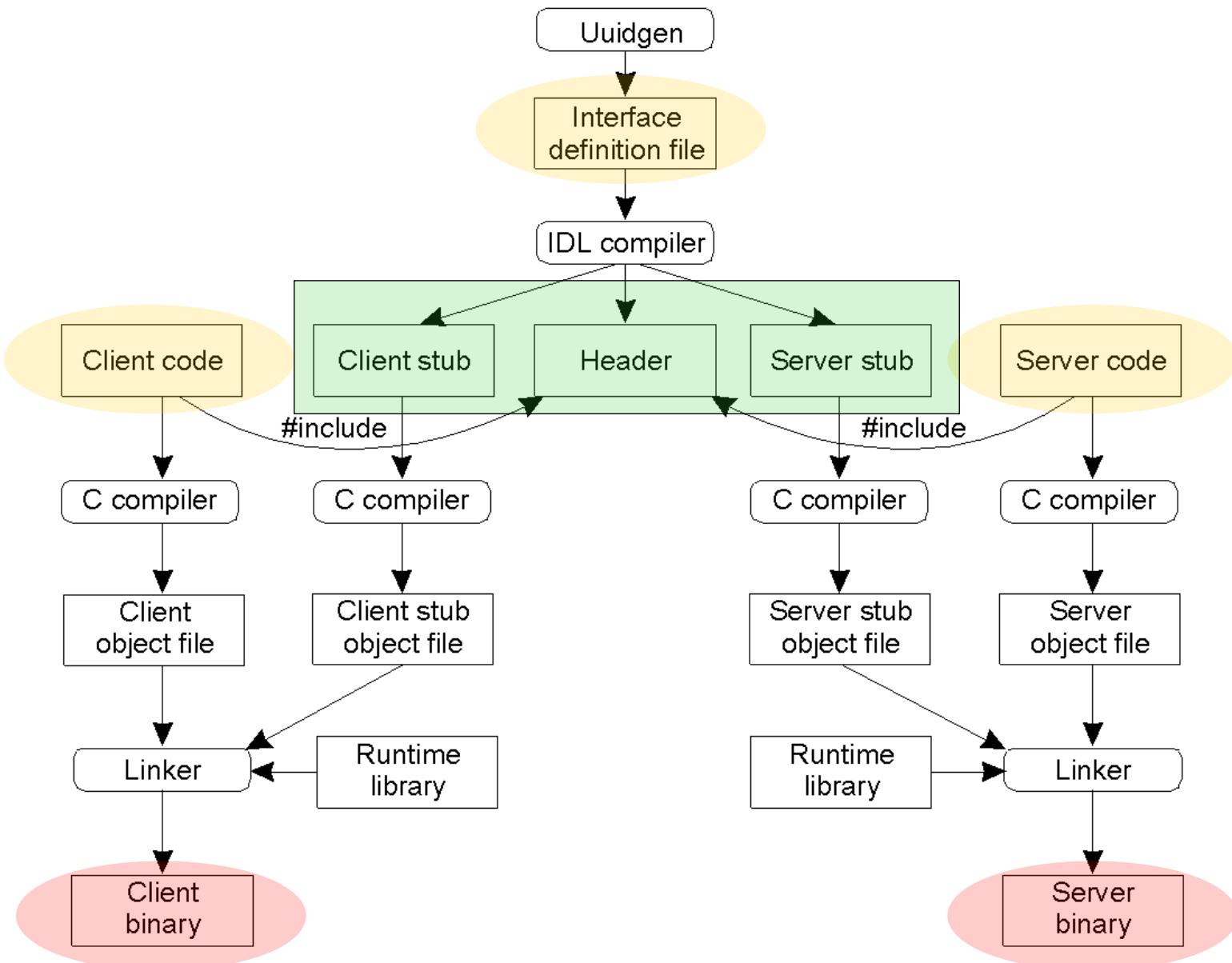
jednoznačný id rozhraní

definice rozhraní

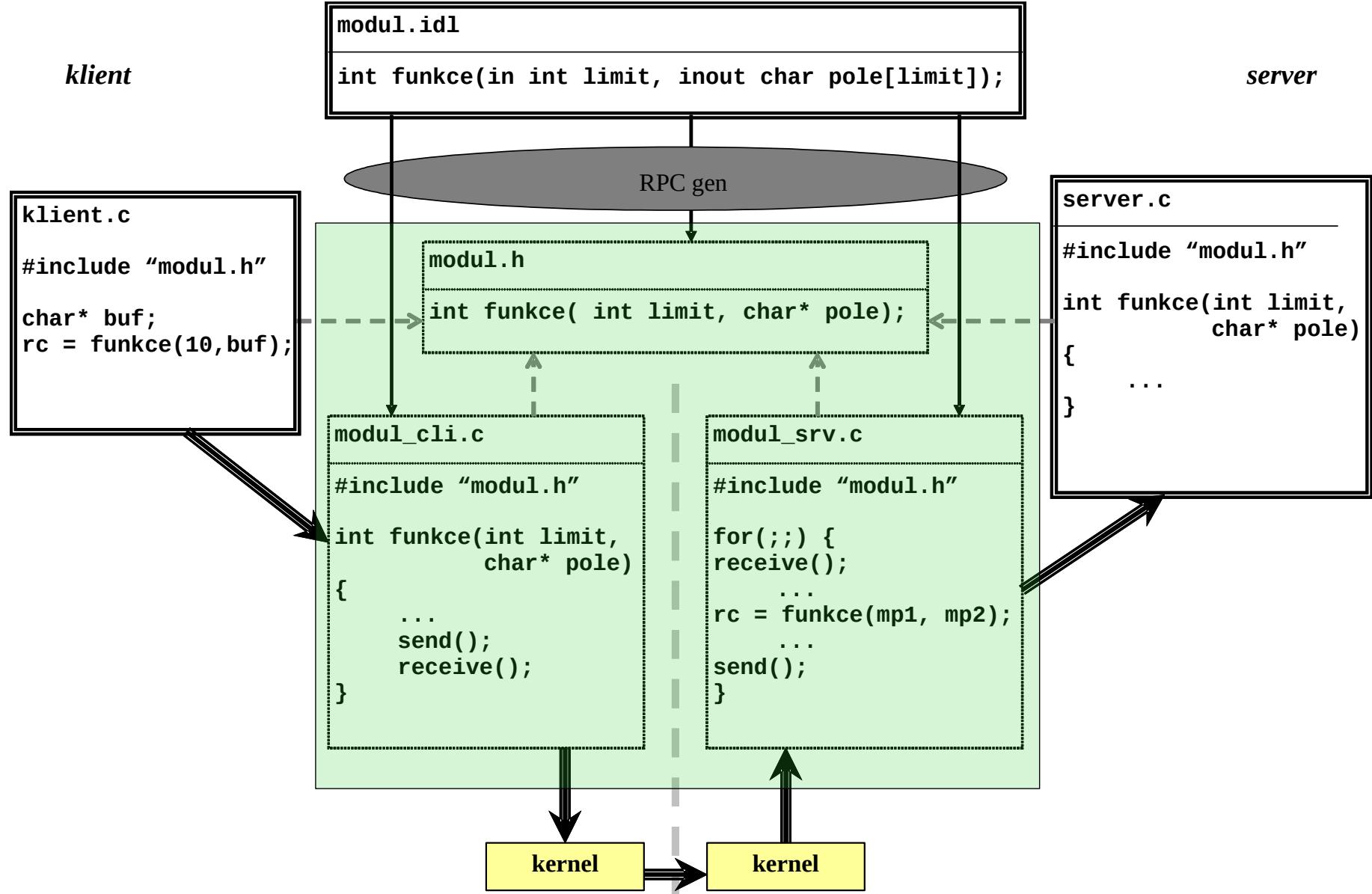
omezení velikosti polí

vstupní / výstupní parametry

RPC - komplikace modulů



RPC - komplilace modulů



Problémy

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

RPC obecně

□ užitečné, prakticky použitelné ... a používané

☒ není zcela transparentní, nutno dávat pozor na omezení

Problémy

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

RPC obecně

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

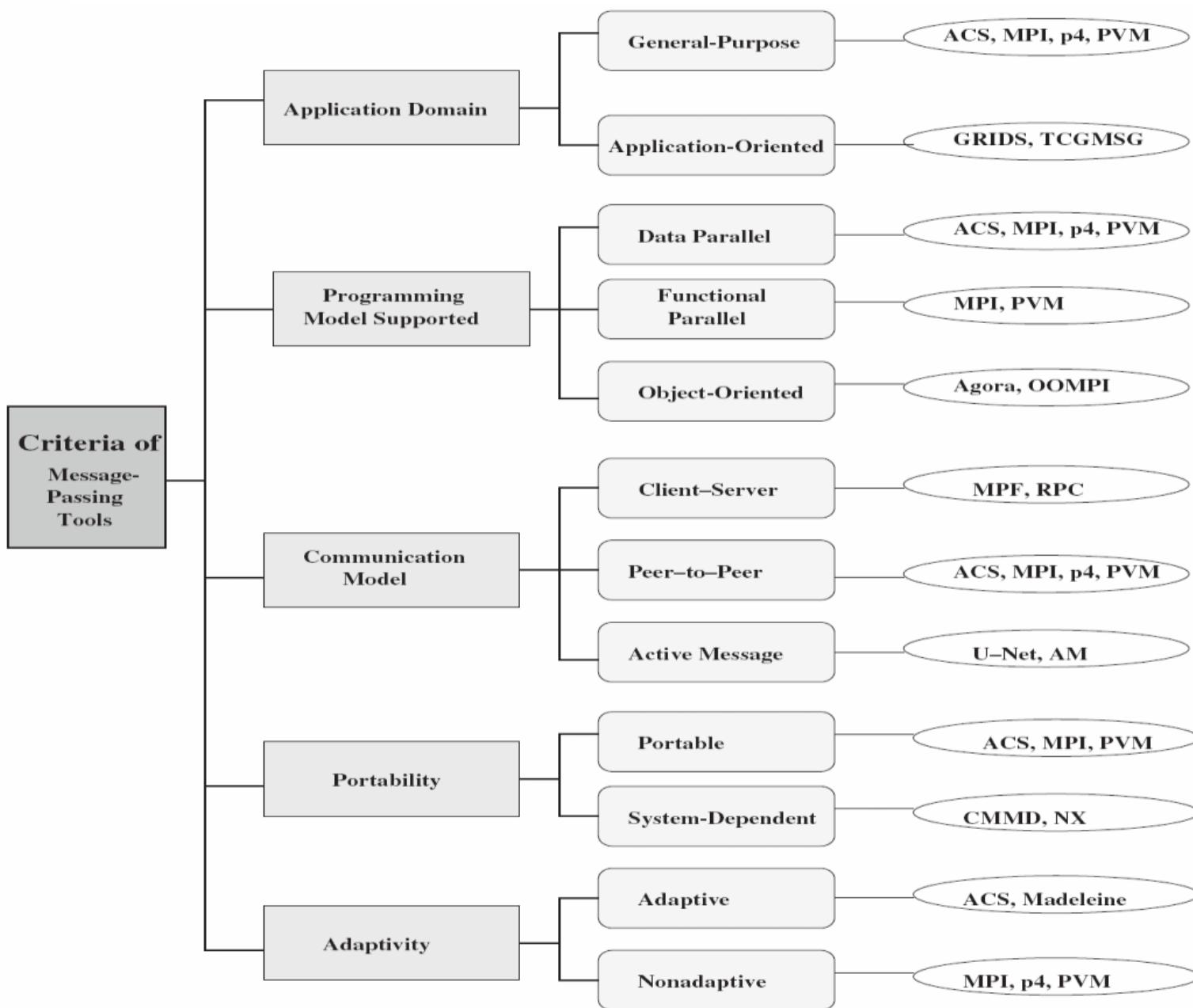
Další mechanismy:
SRPC
Asynchronous RPC
Doors, RMI
MPI, MQ, MOM, ...

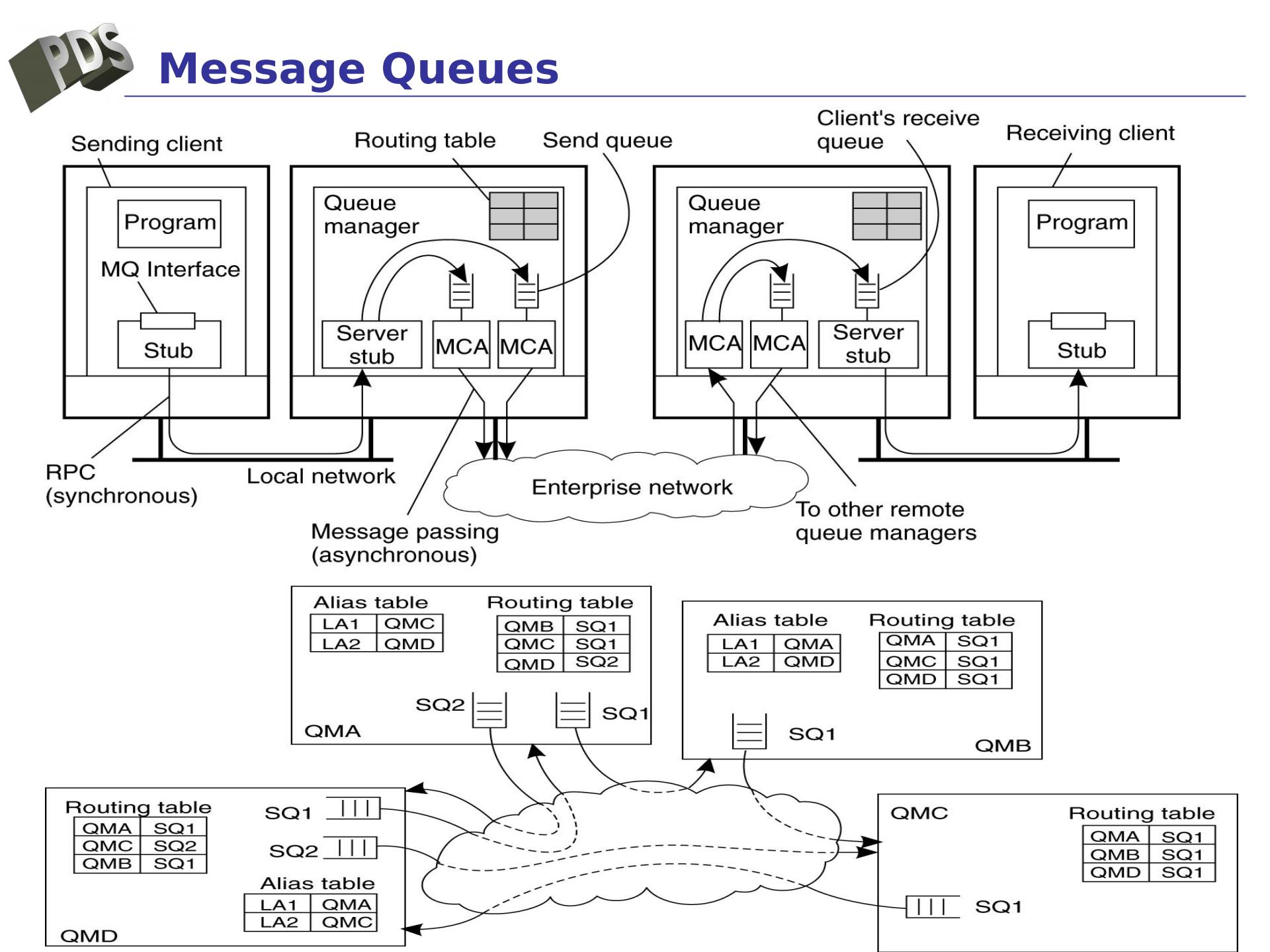


C++ RPC-related systems in use today

- XML or JSON over HTTP
 - - no IDL, hard to parse
- Google protobufs, Apache Thrift
 - + compact encoding, defensively coded (protobufs)
 - + good support for incrementally evolving messages
 - - complex encoding, not C++11
- Google FlatBuffers, Cap'n Proto
 - + same representation in memory and on wire, very fast
 - - less mature, non-deterministic wire format, bigger attack surface
- Apache Avro
 - self-describing messages containing schema
- Cereal
 - C++11 library for serialization
- Cisco RPC/XDR
 - used by Internet standards such as NFS
 - + simple, good features (unions, fixed- and variable-size arrays, ...)
 - - big endian, binary but rounds everything to multiple of 4 bytes

Message passing interfaces





Jeden odesílatel - více příjemců

■ Atomicita

- ◆ doručení **všem** členům nebo nikomu
- ◆ evidence a změna členství

■ Synchronizace

- ◆ doručování od různých odesílatelů
- ◆ příjem vs. doručení zprávy
- ◆ doručovací protokol - sémantika

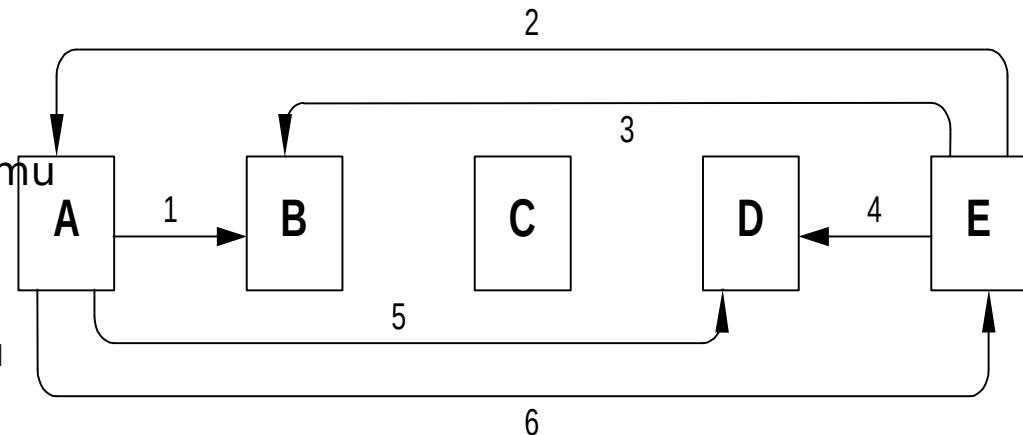
■ Uzavřená skupina

- ◆ pouze členové skupiny
- ◆ vhodné pro kooperativní algoritmy

■ Otevřená skupina

- ◆ zasílat zprávy může kdokoliv
- ◆ distribuované služby, replikované servery

■ Překrývající se skupiny

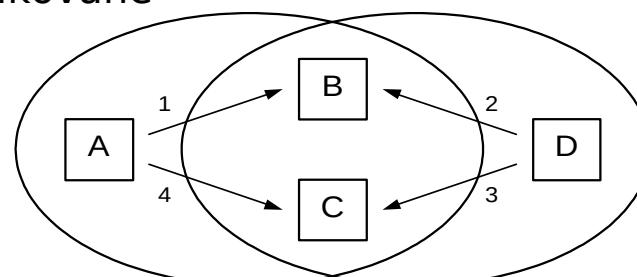


Uzel B:

*A: x += 1;
E: x *= 2;*

Uzel D:

*E: x *= 2;
A: x += 1;*



doručovací protokoly,
virtuální synchronie

Synchronizační algoritmy

Logické a fyzické hodiny

Distribuované vyloučení procesů

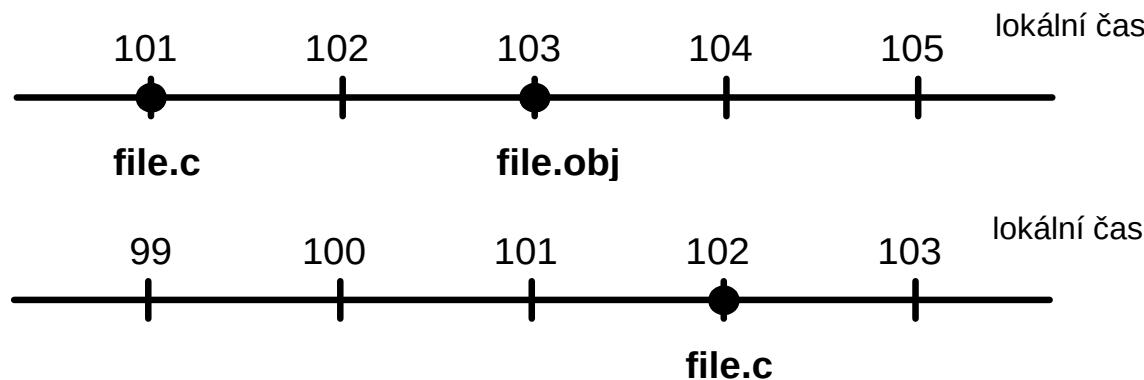
Volba koordinátora

Doručovací protokoly

Virtuální synchronie, členství

... → konsensus

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



Synchronizace s fyzickým časem

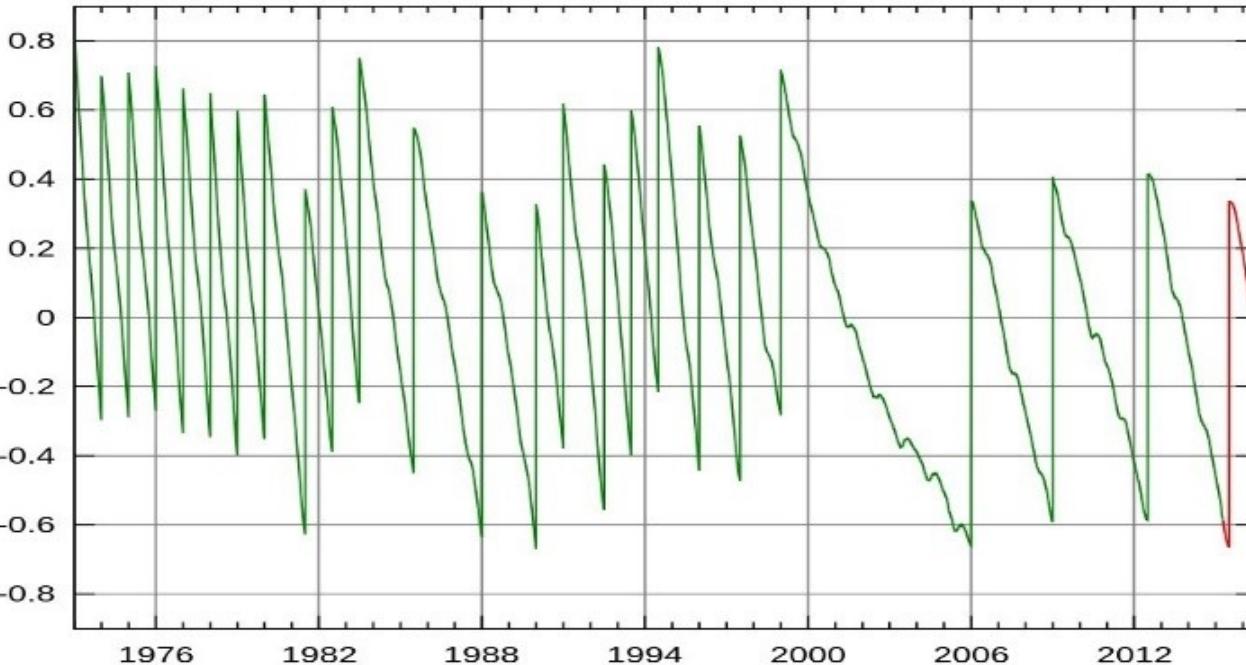
- jak sesynchronizovat lokální hodiny jednotlivých komponent systému
- jak sesynchronizovat jednotlivé hodiny mezi sebou

Fyzický čas

- astronomické měření – solární sec. = $1/86400$ solárního dne
- 1948 - atomové hodiny - $1\text{s} \approx 9$ 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

Fyzický čas

- astronomické měření – solární sec. = $1/86400$ solárního dne
- 1948 - atomové hodiny - $1\text{s} \approx 9$ mld přechodů atomu cesia 133
- 1950 - TAI - Bureau International de l'Heure - průměr asi 50 laboratoří
 - 1 TAI den je o cca 3 ms kratší než solární den
 - rozdíl >800 ms - Leap second ↵ Mezinárodní služba rotace Země (IERS)
 - 1972 Universal Coordinated Time UTC
- vysílání UTC (krátké vlny, satelit) - nepřesnost cca 0.1 - 10 ms



↳ Mezinárodní služba rotace Země (IERS)

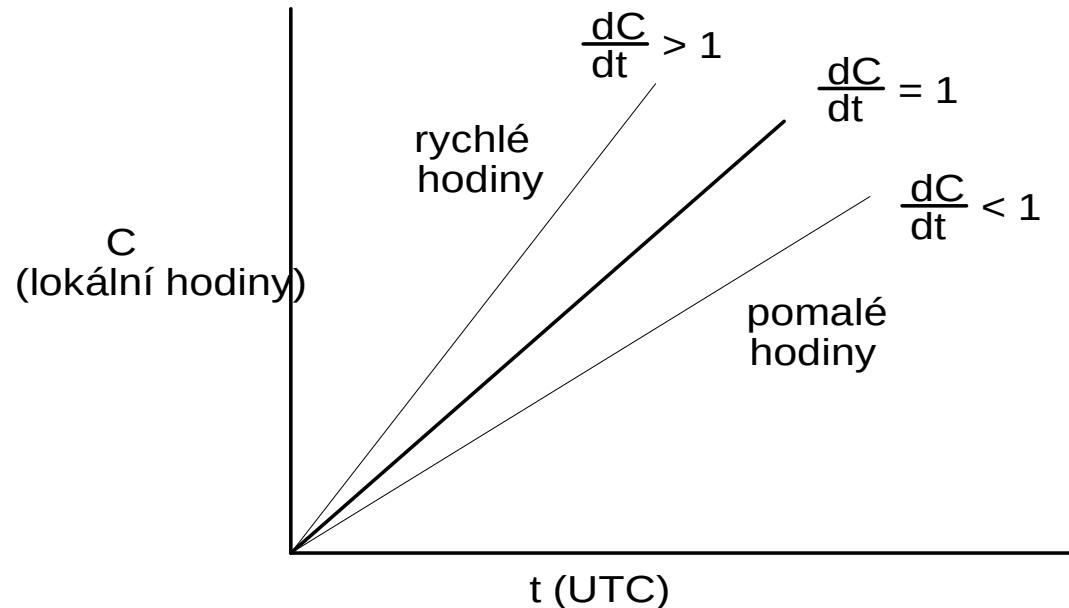
31.12.2016
23:59:**60**

Změna UTC 30.6.2012
neproběhla podle představ
MTU. Problémy hlásily servery
s OS **Debian** Linux, které
postihl krátký výpadek.
Poruchy hlásil také **Reddit**,
databáze **Cassandra** a
programy napsané v **Javě**.

Synchronizace fyzických hodin

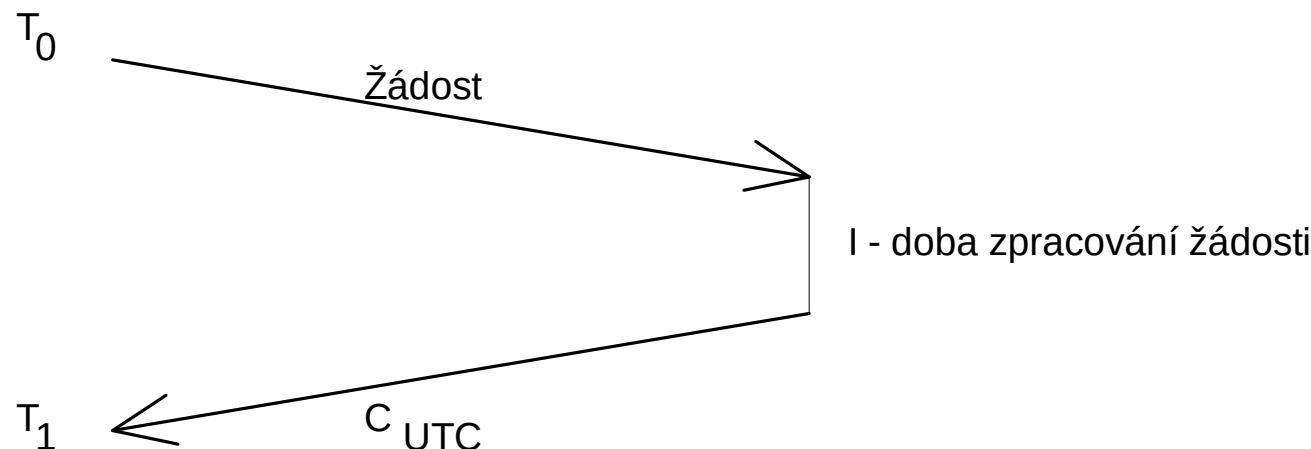
vnitřní hw hodiny C , fyzický čas t
hodiny na počítači p v čase t je $C_p(t)$
přesné hodiny: $C_p(t) = t \ \forall t$, tedy $dC/dt = 1$

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



maximální odchylka dvojice hodin $2\rho\Delta t$
požadovaná odchylka $\delta \Rightarrow$ intervaly max. $\delta/2\rho$

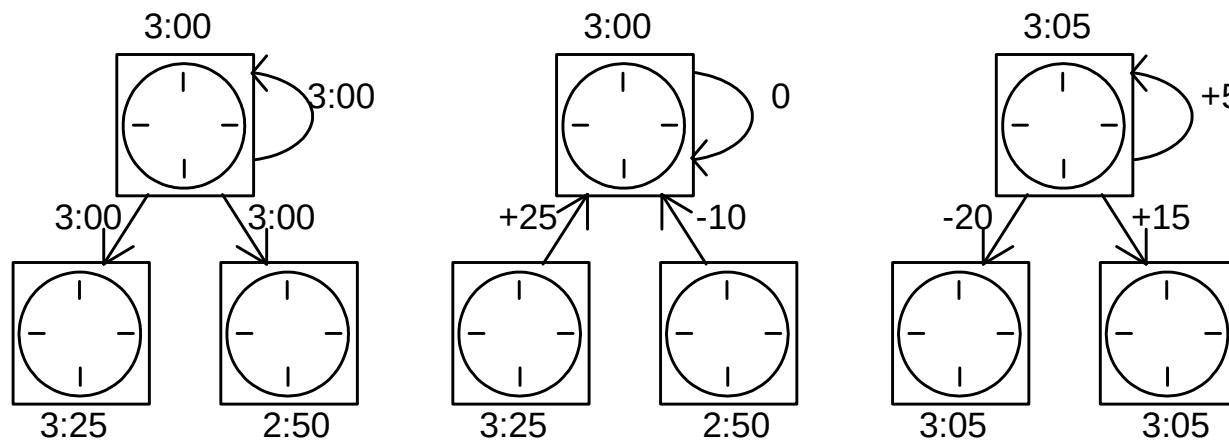
Cristianův algoritmus



- jeden 'přesný' pasivní time server (UTC)
- periodické dotazy v intervalech $< \delta/2\rho$
- $T = T_{\text{UTC}} + (T_1 - T_0 - I)/2$
- nikdy nepřeřizovat najednou !
 - (dis)kontinuita
 - postupná změna
 - zrychlování nebo zpomalování
- Flaviu Cristian 1989

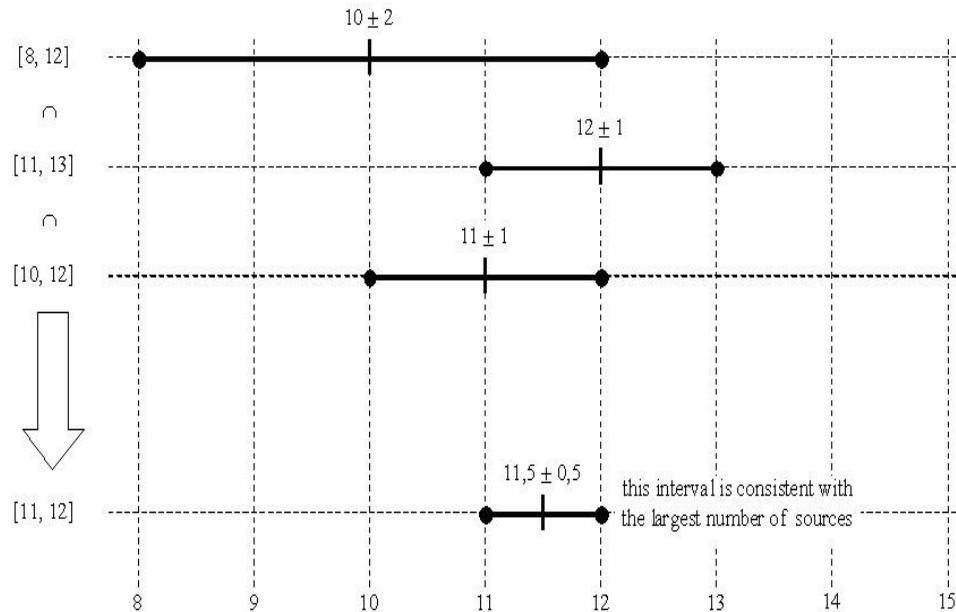


- aktivní time server
- periodicky se ptá ostatních na rozdíl času
- zahodí extrémy, spočítá průměr, vrátí rozdíl
- postupná synchronizace
- *Gusella and Zatti, Berkeley 1989*



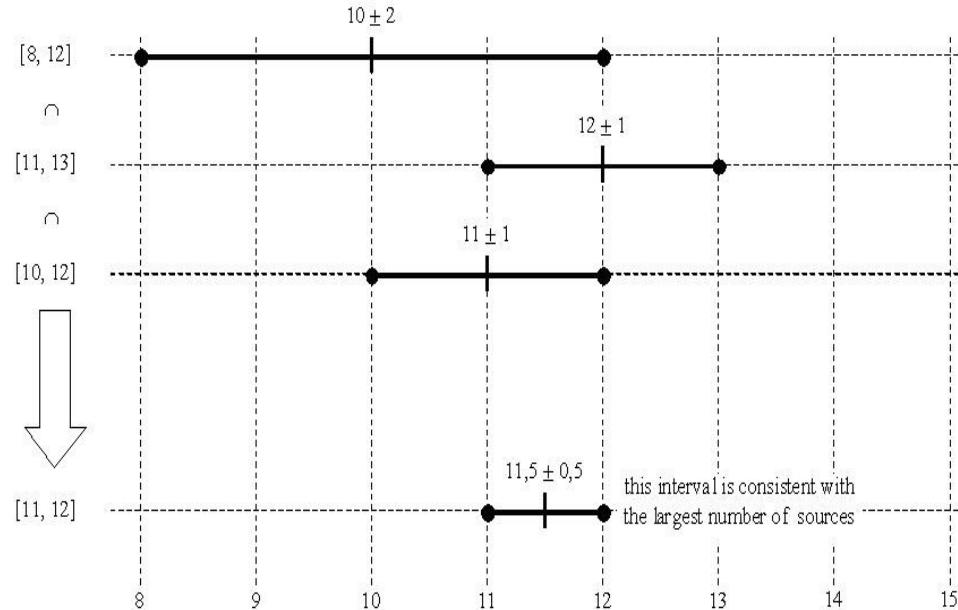
Intersection algorithm

- žádný server ani koordinátor
- resynchronizační intervaly pevné
 - i-tý interval: $T_0+iR \dots T_0+(i+1)R$
- broadcast (ve fyz. nestejný čas)
- případné zahození extrémů
- spočítání průměru a odchylky
- *Marzullo 1984*
 - modifikovaná verze použita pro N



Intersection algorithm

- žádný server ani koordinátor
- resynchronizační intervaly pevné délky R
 - i-tý interval: $T_0+iR \dots T_0+(i+1)R$
- broadcast (ve fyz. nestejný čas)
- případné zahodení extrémů
- spočítání průměru a odchylky
- *Marzullo 1984*
 - modifikovaná verze použita pro NTF



Intervalový čas

- čas není okamžik ale interval
 - porovnání časů - některé časové údaje jsou neporovnatelné
- DCE - 33 knihovních funkcí
 - time clerk (klient) - určování času, započítání odchylky
- Google TrueTime 2012 - $[\text{earliest}, \text{latest}] = \text{now} \pm \epsilon$
 - nepřesnost $200 \mu\text{s}$, synchronizace $30 \text{ s} \Rightarrow$ odchylka $\pm 6 \text{ ms}$
- Spanner - NewSQL distributed DB

Problém: fyzické hodiny nelze dostatečně přesně sesynchronizovat 😞

Lamport:

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

Problém: fyzické hodiny nelze dostatečně přesně sesynchronizovat 😞

Lamport:

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

$e1 \rightarrow^p e2$ uspořádání v rámci procesu/uzlu p
send(m), recv(m) je odeslání/příjem zprávy m

☠ Kauzální závislost →

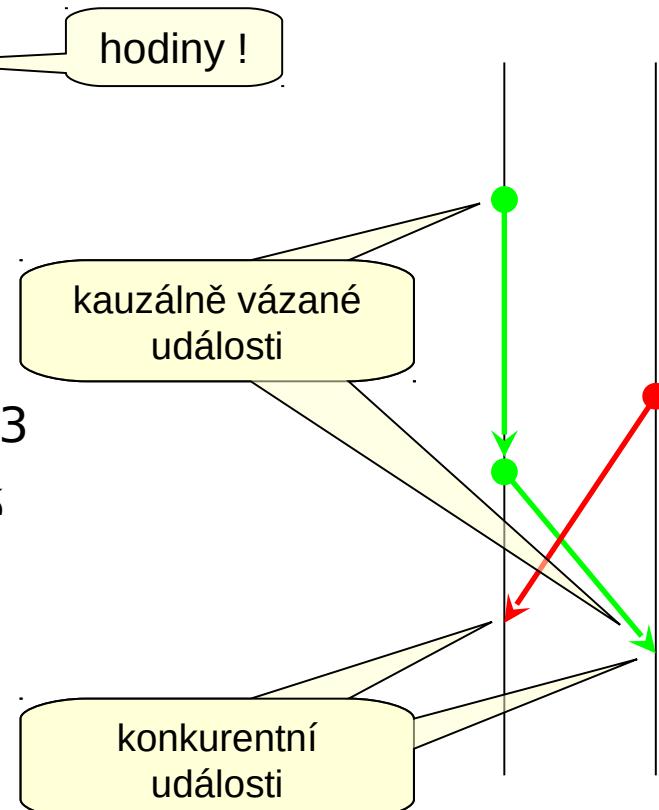
- jestliže $\exists p: e1 \rightarrow^p e2$ potom $e1 \rightarrow e2$
- $\forall m: \text{send}(m) \rightarrow \text{recv}(m)$
- jestliže $e1 \rightarrow e2 \wedge e2 \rightarrow e3$ potom $e1 \rightarrow e3$

Kauzálně předchází, kauzálně vázané/závislé

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

Logické hodiny

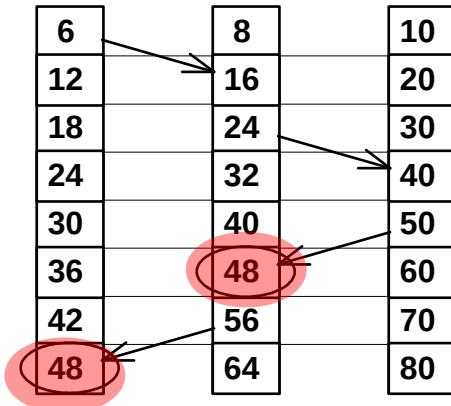
- událost a, čas $C(a)$
- jestliže $a \rightarrow b$ pak $C(a) < C(b)$



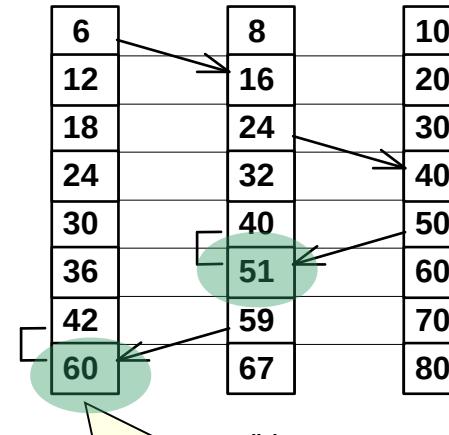
Synchronizace logických hodin

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 $C_j = \max(C_j(b), Tm+1)$



(a)



oprava lokálních hodin

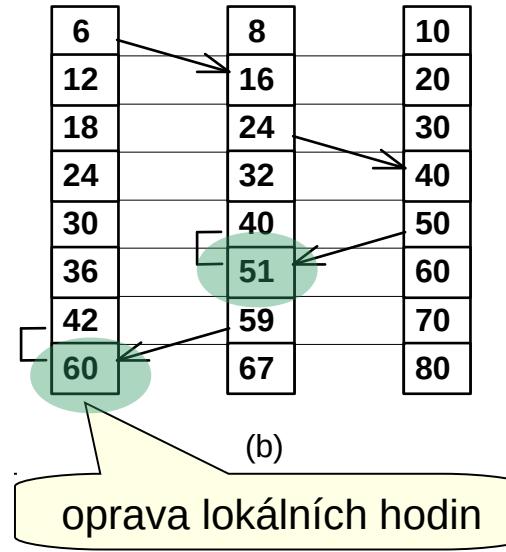
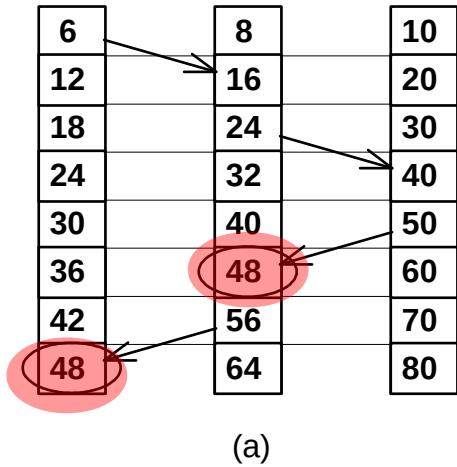


Leslie Lamport

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
- $C(a)=C(b) \& P_i < P_j \Rightarrow C'(a) < C'(b)$
- 'byrokratické' uspořádání

Kauzální závislost

- jestliže $\exists p: e1 \rightarrow_p e2$ potom $e1 \rightarrow e2$
- $\forall m: send(m) \rightarrow recv(m)$
- jestliže $e1 \rightarrow e2 \wedge e2 \rightarrow e3$ potom $e1 \rightarrow e3$

Platí ☺

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

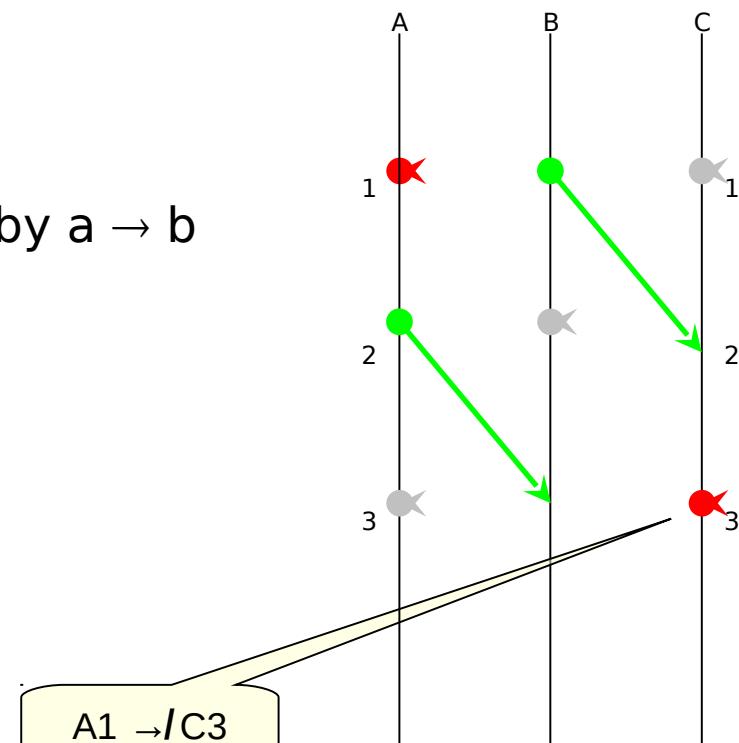
Neplatí ☹

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

Jak tedy zjistím kauzalitu událostí?

- později - vektorové hodiny

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



A1 → / C3

Vzájemné vyloučení procesů

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



■ Způsoby řešení

- ◆ centralizovaná ochrana přístupu
- ◆ permission-based
 - časové značky
 - volby
- ◆ token-based
 - token-passing
 - strom
 - token ring

Lamport ($3n$), *Ricart-Agrawala* ($2n$)

Maekawa (\sqrt{n}), *Agrawal-ElAbadi* ($\log n$)

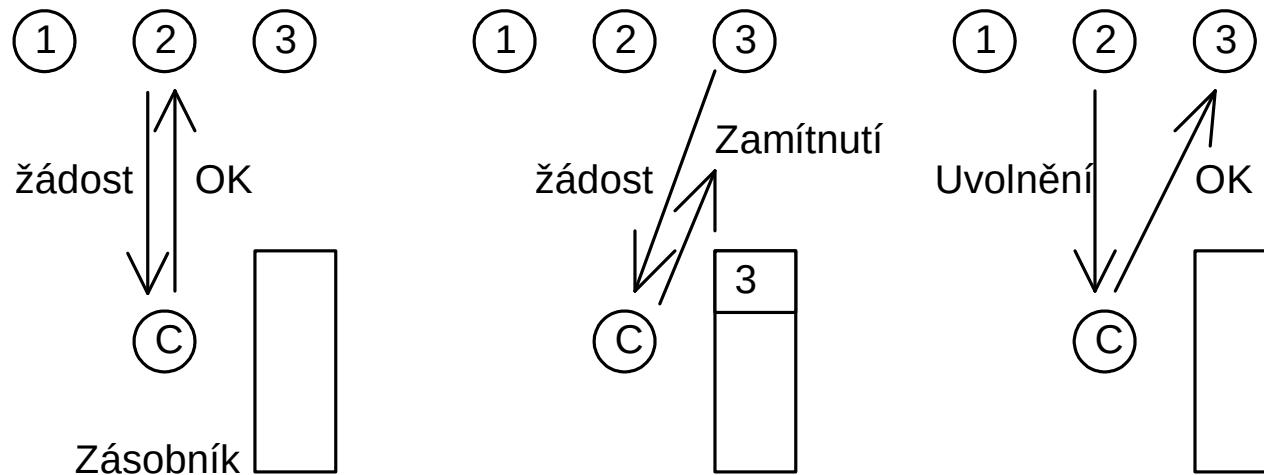
Suzuki-Kasami

Raymond

Centralizovaný algoritmus

Centralizovaný algoritmus

- ◆ jeden server s frontou - sekvencer
- ◆ žádost / potvrzení / zamítnutí / uvolnění
- ◆ centralizovaná komponenta
 - ideově nevhodné
 - výpadek serveru - ztráta informace, nutnost volby
 - výpadek klienta - problém vyhľadovení (starvace ↔ expirace)



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

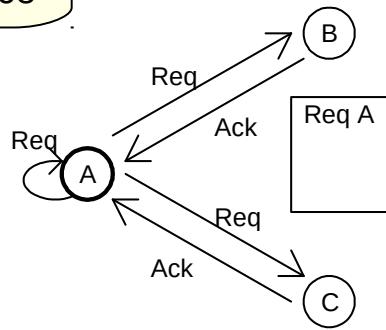
akce se zprávami send, add, del

Událost	Akce procesu p	
Žádost M_p	send $M_p = \{req, p, T_p\}$	uložení žádosti a odpověď s vlastním časem
Přijetí žádosti M_i	add M_i ; send $\{ack, p, T_p\}$	od všech přišlo novější potvrzení
Přijetí potvrzení A_i	add A_i	neviduju starší žádost
Podmínka vstupu M_p	$\forall i \neq p \exists \{ack, i, T_i\} : T_p < T_i$ $\& \forall \{req, i \neq p, T_i\} : T_p < T_i$	
Uvolnění R_p	send $\{rel, p, T_p\}$	
Přijetí uvolnění R_i	del $\{req, i, T_k\} : T_k < T_i$	po přijetí uvolnění smažu starší žádosti tohoto procesu

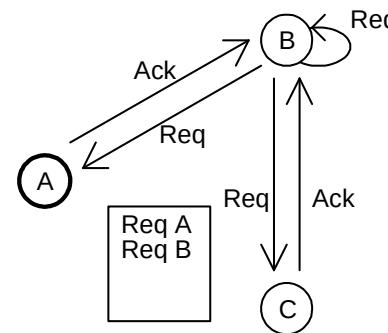
Komunikační složitost: $3(n-1)$

Lamportův algoritmus

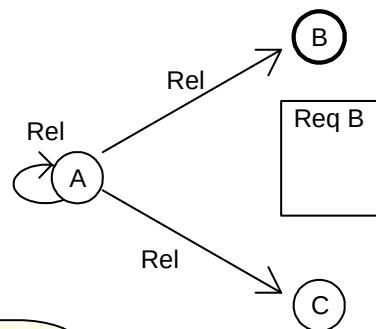
Vstup A
do kritické sekce



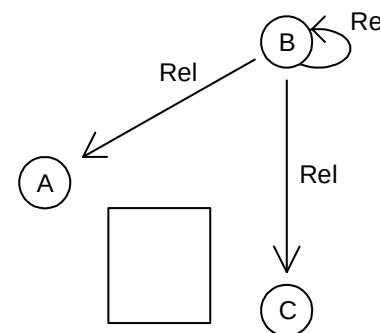
Žádost B o vstup



Uvolnění A
a vstup B



Uvolnění kritické
sekce



Lamportův algoritmus

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

program p : do forever \rightarrow

```
{  $\neg x_p$  }  
 $x_p, F_p := \text{true}, c_p$   
; {  $x_p$  }  
( ||  $q : p \neq q$  : send  $F_p$  to co  $p, q$   
; receive-acknowledgement from co  $p, q$   
{  $\neg x_q \vee F_p < F_q$  } {  $F_p < c_q$  }  
)  
; {  $x_p$  } {  $(\forall q : p \neq q : (\neg x_q \vee F_p < F_q) \wedge F_p < c_q)$  }  
Critical Section  $p$   
;  $x_p := \text{false}$ 
```

od

Vyloučení procesů - Ricart & Agrawala

Proces chce vstoupit do kritické sekce:

- zašle žádost s TM a čeká na odpovědi s potvrzením

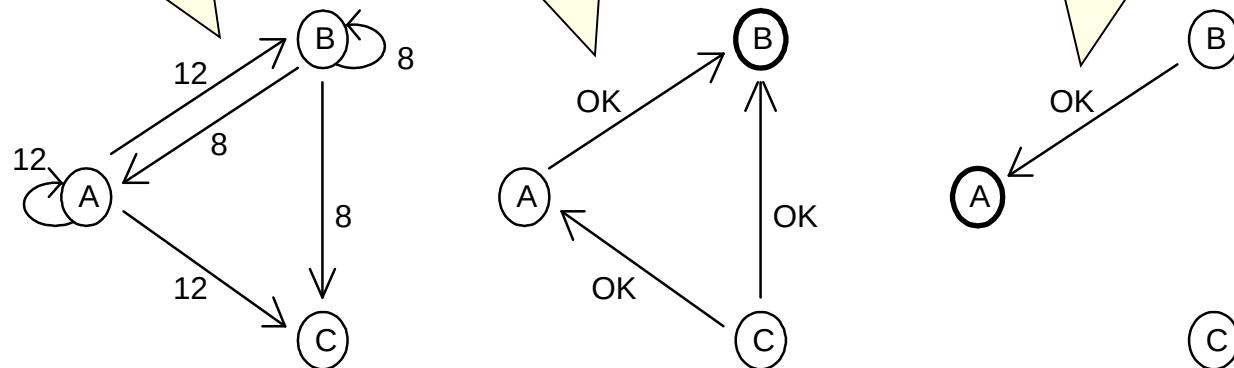
rozdíl: význam ACK

Proces přijme zprávu se žádostí:

- jestliže **není** v kritické sekci a ani do ní **nechce** vstoupit, pošle potvrzení
- 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:
 - žádost má **nižší** časovou značku (= starší), neodpovídá a zařadí žádost odesilatele do fronty
 - v opačném případě (žádost odesilatele je starší) pošle zpět potvrzení

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

frontě
Komunikacní složitost: $2(n-1)$ - lepší do výše liscárni



■ 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 **hlas**
 - 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
- ◆ při žádosti jiného procesu dá proces hlas
- ◆ 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 - 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 hlas
 - 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
- ◆ při žádosti jiného procesu dá proces hlas (pokud ještě nehlasoval)
- ◆ ☺ relativně odolný proti výpadkům
 - vydrží výpadek až poloviny procesů
- ◆ 😞 složitost = $O(n)$
 - není lepší než TS
- ◆ 💀 deadlock!
 - více procesů může dostat stejný počet hlasů

```
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

- ◆ $\forall p, q: S_p \cap S_q \neq \emptyset$
 - **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 voličů
- ◆ $\forall p, q: |S_i: p \in S_i| = |S_j: q \in S_j| = D$
 - p je obsažen ve stejném počtu D volebních okrsků
 - každý proces má stejnou zodpovědnost

korektnost
t

spravedlnost
ost

zodpovědno
st

komunikace
jen se svým
okrskem

ukále jak?

■ Důsledek - komunikační složitost $O(|S_p|)$

- ◆ cíl: minimalizovat velikost volebních okrsků **ukále jak?**

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
 - $N = M \Rightarrow K = D$
 - \rightarrow velikost okrsku \approx počet okrsků, ve kterých je každý proces
- každý proces q z okrsku S_p je obsažen v $D-1$ jiných okrscích
 - max. počet okrsků $(D-1)K+1$
- $M = K(K-1)+1 \rightarrow K = O(\sqrt{M})$

okrsky \bowtie procesy

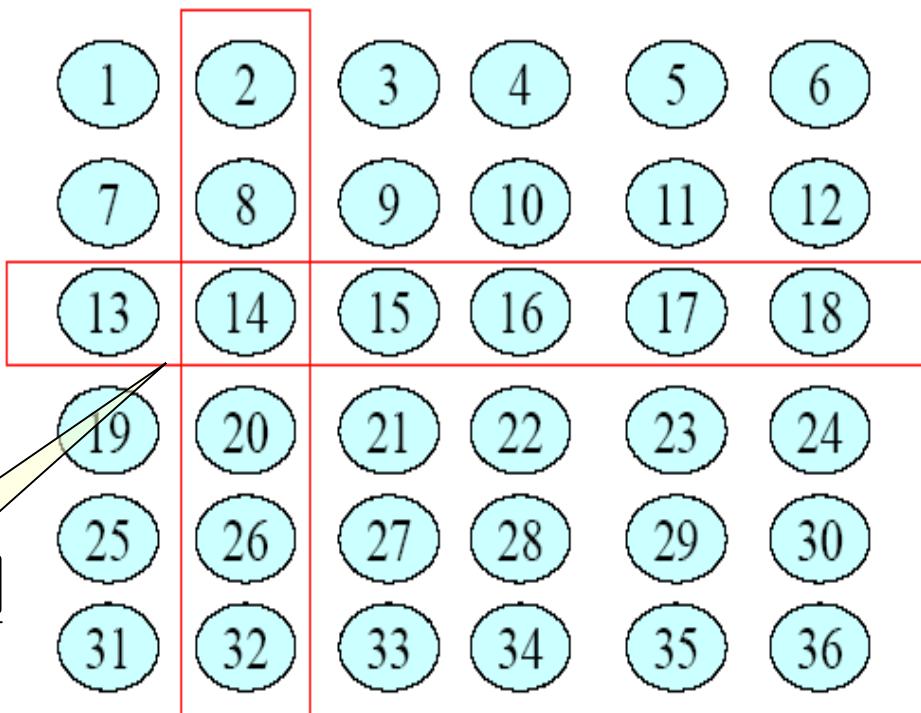
komunikační složitost

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

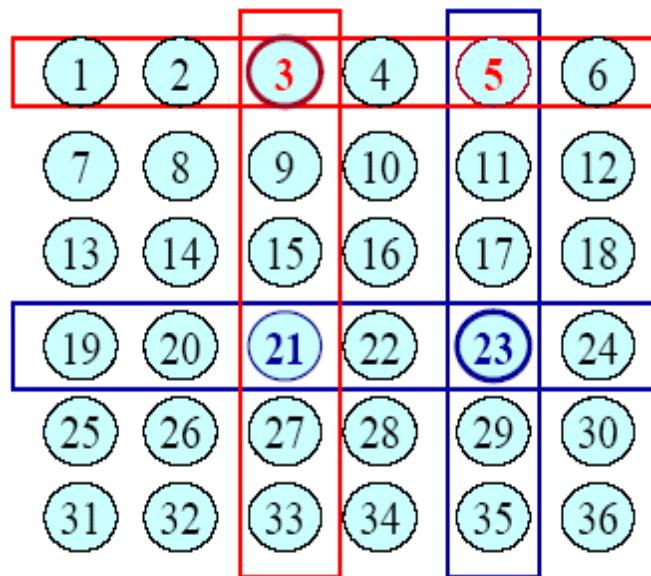
■ Algoritmus rozdělení procesů do okrsků

- ◆ optimální pro $M=K(K-1)+1$ složitý
 - vyžaduje restrukturalizaci při změně členství
- ◆ suboptimální pro $K = O(\sqrt{M})$ jednoduchý
 - prakticky použitelný
 - $M = n^2$, $P_{i,j}$, $S_{i,j}$, $1 \leq i,j \leq n$
 - $S_{i,j} = \bigcup P_{i,x} \cup \bigcup P_{y,j}$ pro $1 \leq x,y \leq n$

K - velikost okrsku
M - počet procesů



Maekawa - deadlock a jeho řešení

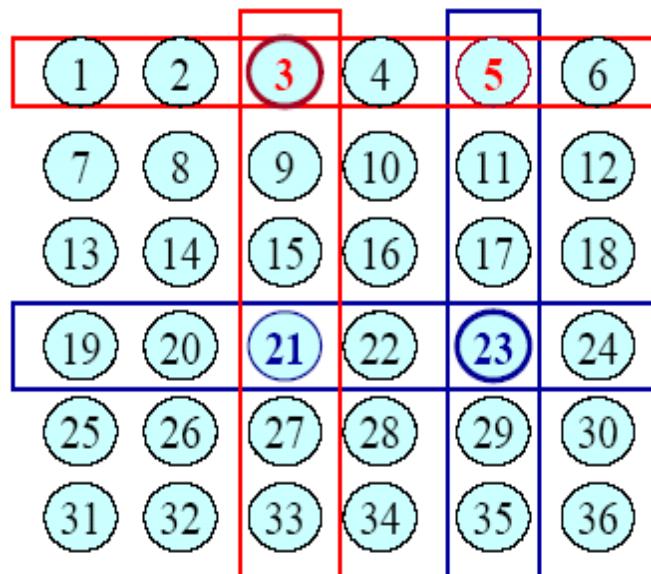


Soutěží P3 a P23

Možnost deadlocku
P21 volí P23
P5 volí P3

Nelze získat
všechny hlasů
DEADLOCK

Maekawa - deadlock a jeho řešení



Soutěží P3 a P23

Možnost deadlocku
P21 volí P23
P5 volí P3

novější

starší !

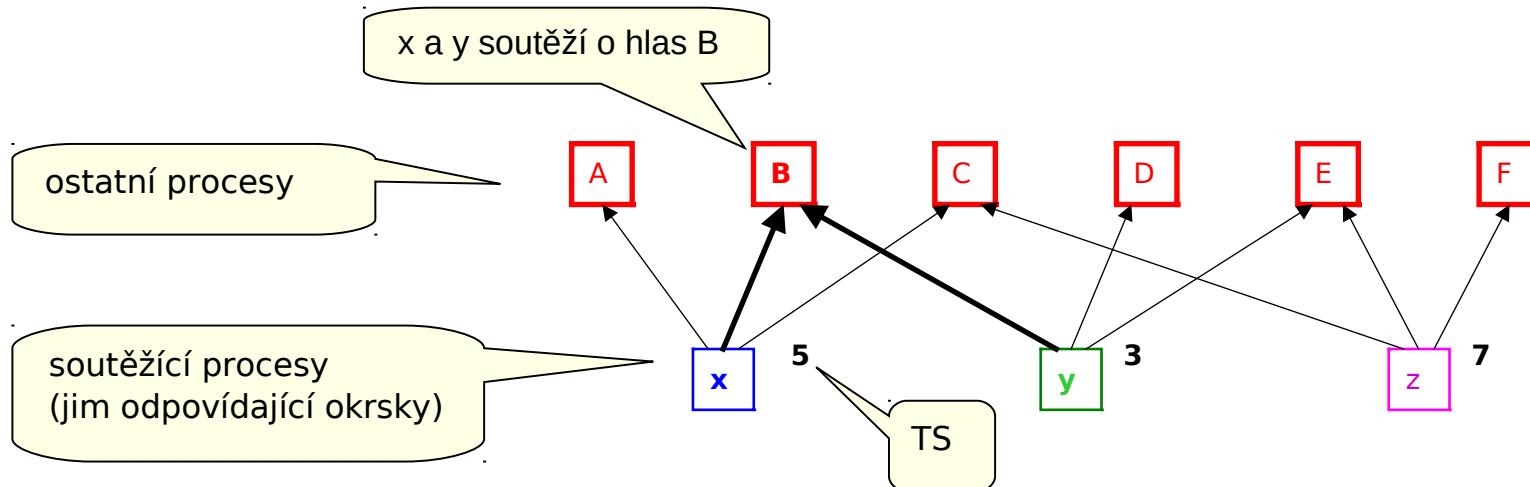
Race
Condition !!
Nevadí to ??

■ 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

Maekawa - deadlock a jeho řešení



■ Scénář 1

- ◆ yD, yE - ACK
- ◆ xA, xC - ACK
- ◆ **yB - ACK**
- ◆ $y \rightarrow CS$
- ◆ **xB - enqueue x**
- ◆ $y \leftarrow CS - REL$
- ◆ xB - ACK
- ◆ $x \rightarrow CS$

TS:
 $x > y$

■ Scénář 2

- ◆ yD, yE - ACK
- ◆ xA, xC - ACK
- ◆ **xB - ACK**
- ◆ $x \rightarrow CS$
- ◆ **yB - REJECT x - ignored**
- ◆ $x \leftarrow CS - REL$
- ◆ yB - ACK
- ◆ $y \rightarrow CS$

x:CS

■ Scénář 3

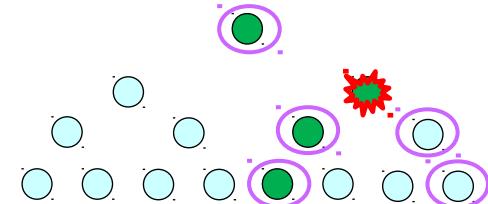
- ◆ yD, yE - ACK
- ◆ xA - ACK
- ◆ **xB - ACK**
- ◆ **yB - REJECT x - revoked**
- ◆ yB - ACK
- ◆ $y \rightarrow CS$
- ◆ $y \leftarrow CS - REL$
- ◆ xC, xB - ACK
- ◆ $x \rightarrow CS$

TS:
 $x > y$

Vyloučení procesů - stromové algoritmy

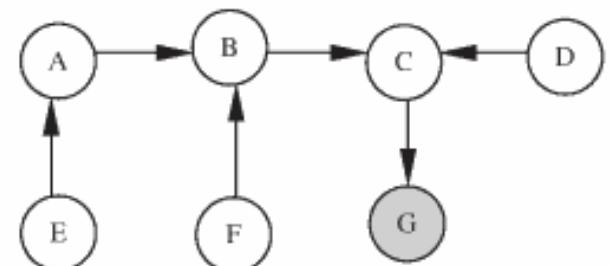
Agrawal & El Abbadi (1991)

- ◆ Volby - složitost $O(\ln(n))$
 - úplný binární strom
 - quorum = cesta od kořene k listu
 - fault tolerance
 - náhrada havarovaného uzlu cestou v L a R podstromu



Raymond (1989)

- ◆ **token-based** - kořen
- ◆ kostra, neorientovaný strom → orientovaný strom
- ◆ každý uzel udržuje
 - referenci na souseda směrem k token holderu
 - frontu žádostí
- ◆ forward žádosti otci
- ◆ vstup do kritické sekce
 - přenos tokenu
 - změna kořenu
 - přesměrování referencí - rekonfigurace

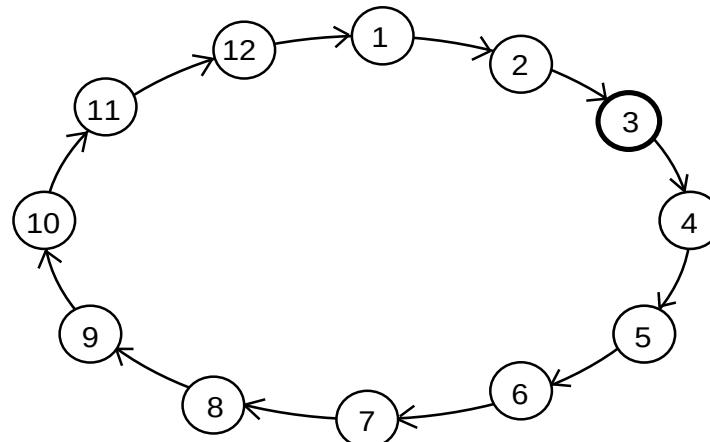


Suzuki & Kasami (1985)

- ◆ 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, klienta
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

Porovnání algoritmů vzájemného vyloučení

Algoritmus	Počet zpráv	Problémy
Centralizovaný	3	havárie koordinátora, klienta
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

Nevhodný není centralizovaný algoritmus,
ale centralizovaný problém!

Volba koordinátora

- Leader election
- korektnost!
 - za žádných okolností nesmí existovat více koordinátorů najednou
- koordinátor / leader nutný pro řadu distribuovaných algoritmů
 - distribuovaný konsensus, load balancing, ...
 - součást distribuovaných / cloud / cluster frameworků
- Způsoby řešení
 - ◆ detekce extrému
 - synchronní systémy - *Bully*
 - asynchronní / partitioned systémy - *Invitation*
 - kruhové algoritmy - *LeLann, Hirschback-Sinclair*
 - složitější topologie - mesh, torus, hypercubes
 - ◆ race-condition (+ randomizace)
 - RAFT

Volba koordinátora - bully algoritmus

■ Předpoklady

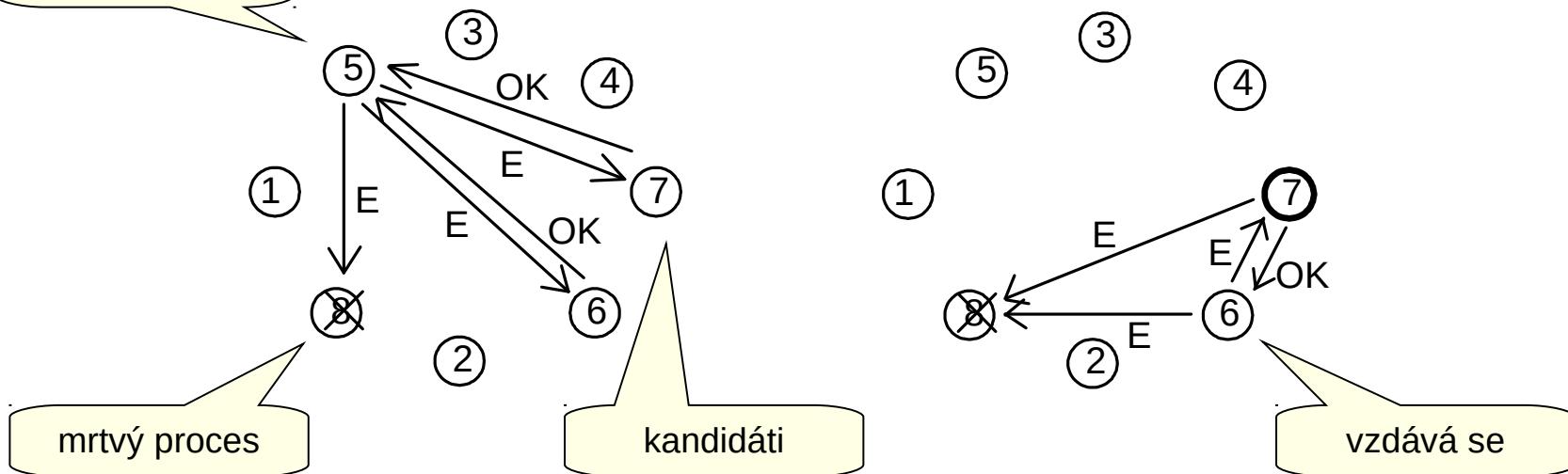
- ◆ omezená doba přenosu zprávy *(message propagation time)*
- ◆ omezená doba zpracování zprávy a odpovědi *(message handling time)*
Pozor!! Velmi silný požadavek
- ◆ **synchrone systém** - 'spolehlivý' detektor havárie: $2T_m + T_p$

■ 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



- 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 detektovat havárii
 - absence odpovědi neznamená havárii

Idea

- ◆ koordinátor je vázán na skupinu, skupiny lze štěpit
- ◆ všichni členové stejné verze skupiny (*pohled, view*) vidí stejného koordinátora

Invitation algoritmus (Garcia-Molina 1982)

Reálné prostředí

- ◆ havárie, neomezená doba reakce, nespolehlivá detekce havárie

Idea: koordinátor je vázán na skupinu, skupiny lze štěpit

koordinátor: pravidelná výzva *AreYouCoordinator*

příjem AYC koordinátorem

- ◆ sjednocení do skupiny vyššího koordinátora

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

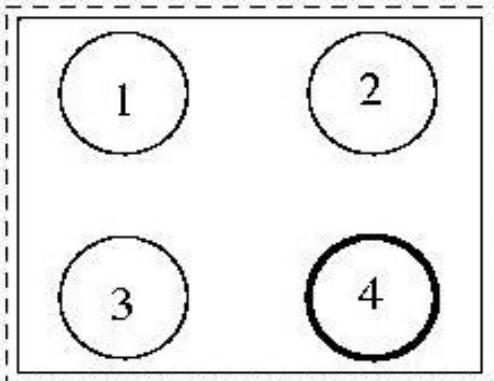
- ◆ prohlásí se za koordinátora vlastní nové skupiny
- ◆ pozvání ke sjednocení ostatním novým skupinám

konzistence relativní vzhledem ke skupině

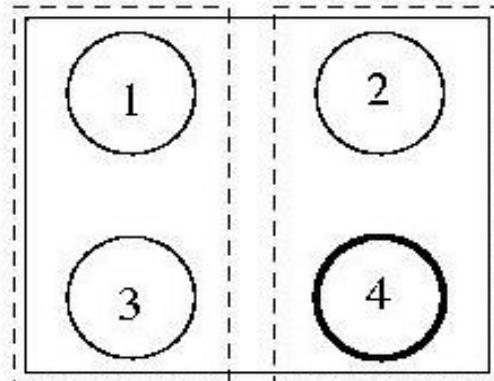
- ◆ procesy se shodují na členství ve skupině a na nějaké hodnotě
- ◆ koordinátor vyzve ostatní k připojení
 - pokud se nepřipojí, jsou konzistentní samy se sebou

silnější sémantiky - distribuovaný konsensus ↳ *později*

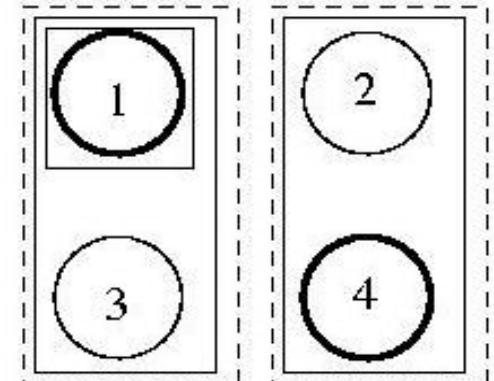
Invitation algoritmus - průběh



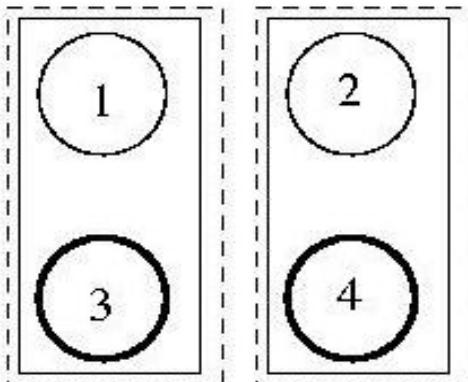
One partition. One group
Coordinator = Processor 4



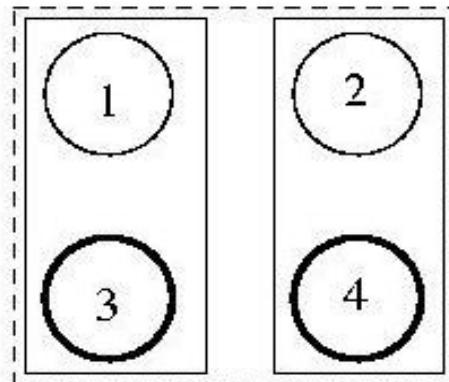
The network is partitioned, but
no one notices



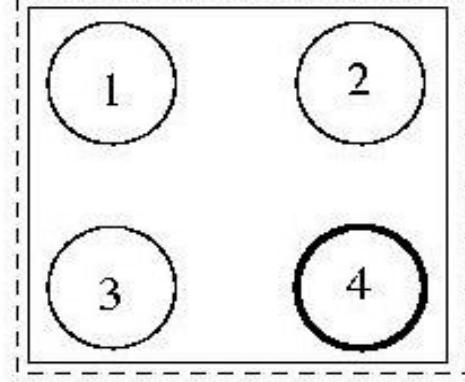
Processor 1 notices the partitioning,
and declares itself a coordinator.
It calls out and reaches processor 3.



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

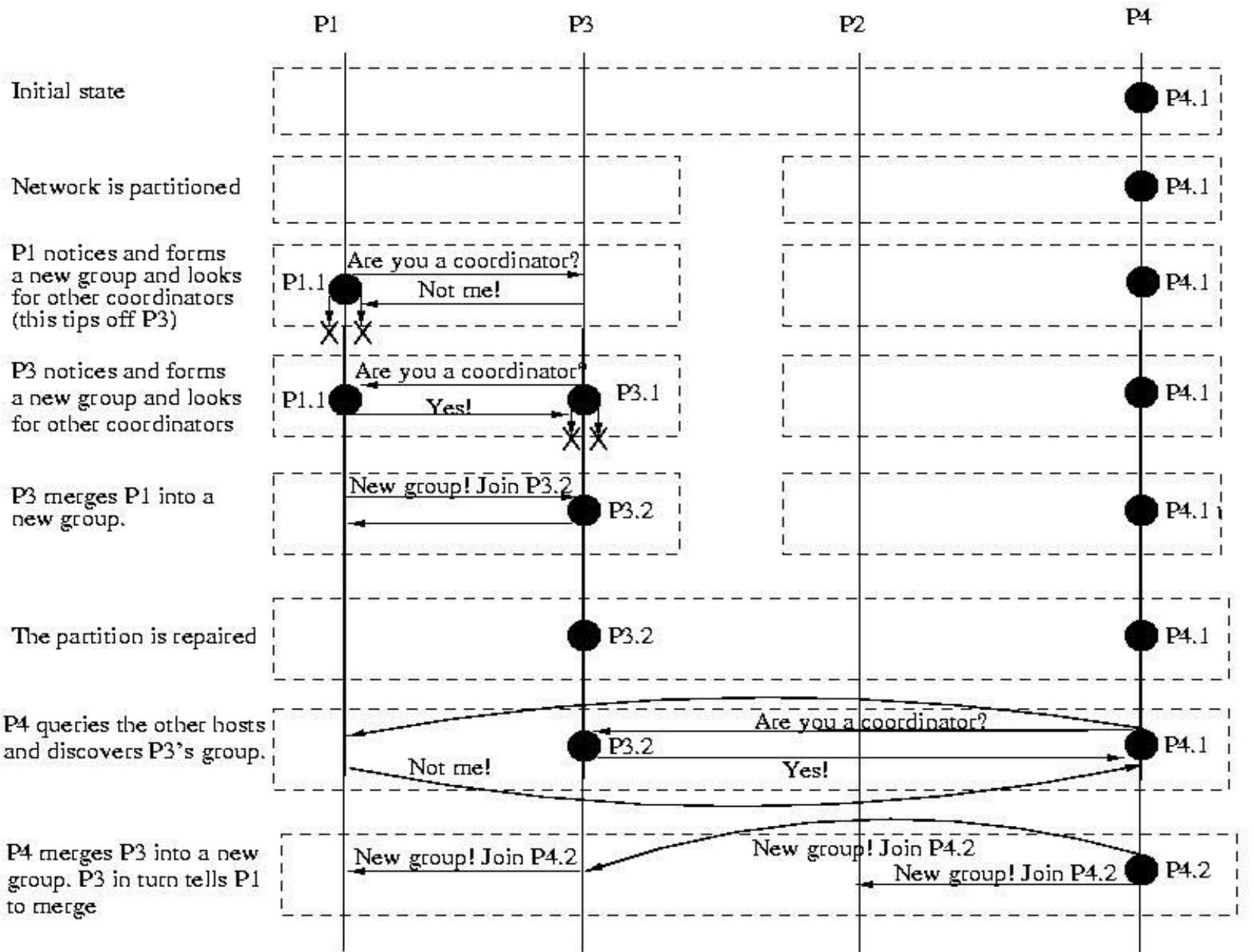


The network partition is repaired,
but none of the processors notice



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.

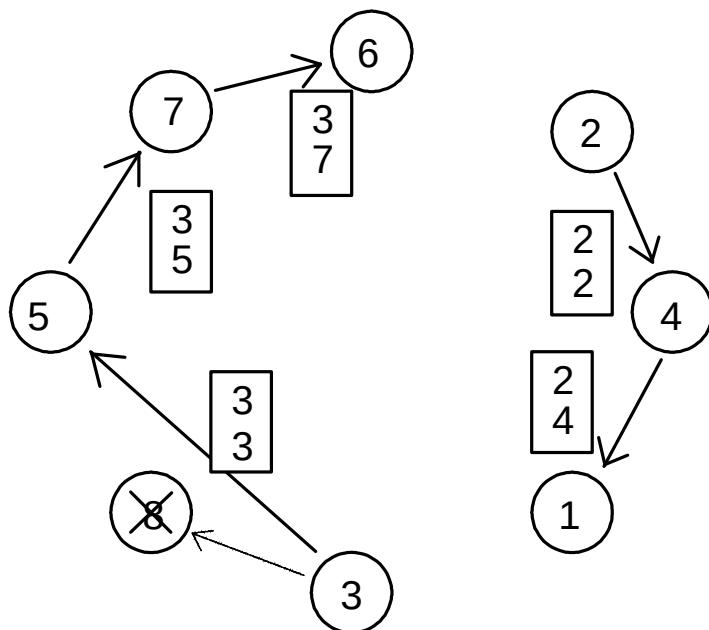
Invitation algoritmus - vyzývací zprávy



Volba koordinátora - kruhový algoritmus

Le Lann, Chang & Roberts (1979)

- 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 $O(n^2)$



HS algoritmus

Hirschback & Sinclair (1980)

- optimalizace komunikační složitosti
 - koordinátor okolí 2^k
 - složitost $O(n \log n)$
 - implementačně náročnější
 - obousměrné propojení
- výhodné při velkých skupinách a vysoké frekvenci konkurentních voleb

Chang & Roberts - pseudokód

```
boolean participant=false;  
int leader_id=null;
```

To initiate an election:

```
send(ELECTION<my_id>);  
participant:=true;
```

Upon receiving a message ELECTION(j):

```
if ( $j > my\_id$ ) then send(ELECTION( $j$ ));  
if ( $my\_id = j$ ) then send(LEADER<my_id>);  
if (( $my\_id > j$ )  $\wedge$  ( $\neg participant$ )) then  
    send(ELECTION<my_id>);  
  
participant:=true;
```

Propagace

Detekce

Vyšší kandidát

Upon receiving a message LEADER(j):

```
leader_id:=j;  
if ( $my\_id \neq j$ ) then send(LEADER( $j$ ));
```

Propagace
výsledku

Hirschback & Sinclair - pseudokód

To initiate an election (phase 0):

send(**ELECTION** $\langle my_id, 0, 0 \rangle$) to left and right;

Upon receiving a message **ELECTION** $\langle j, k, d \rangle$ from left (right):

if $((j > my_id) \wedge (d < 2^k))$ then

Propagace

send(**ELECTION** $\langle j, k, d + 1 \rangle$) to right (left);

if $((j > my_id) \wedge (d = 2^k))$ then

Obrátka

send(**REPLY** $\langle j, k \rangle$) to left (right);

if $(my_id = j)$ then announce itself as leader;

Upon receiving a message **REPLY** $\langle j, k \rangle$ from left (right):

if $(my_id \neq j)$ then

Propagace

send(**REPLY** $\langle j, k \rangle$) to right (left);

else

if (already received **REPLY** $\langle j, k \rangle$)

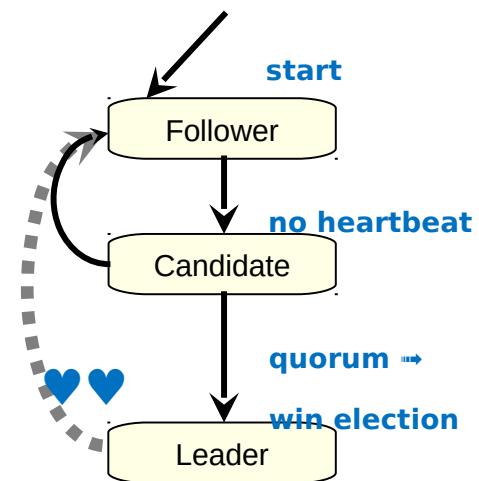
Další kolo

send(**ELECTION** $\langle j, k + 1, 1 \rangle$) to left and right;

Volba koordinátora - randomizovaný protokol

- Volba koordinátora (leadera)
 - Leader, Follower, Candidate
 - Leader - heartbeat
 - timeout - absence heartbeat
 - Follower \rightsquigarrow Candidate
 - vote for self, RequestVote
 - většinový počet hlasů \rightsquigarrow Leader
 - timeout \rightsquigarrow randomizovaná pauza, nová volba
 - $[T, 2T]$... cca 150-300 ms
 - chová se dobře při $T \gg$ doba přenosu zprávy
 - podmínky:
 - rychlá **lokální** síť
 - menší počet aktivních uzlů
 - randomizovaný přístup v praxi jednodušší
- RAFT consensus protocol (*Ongaro 2014*)

extrémní vlastnost X race condition





Doručovací protokoly

■ 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

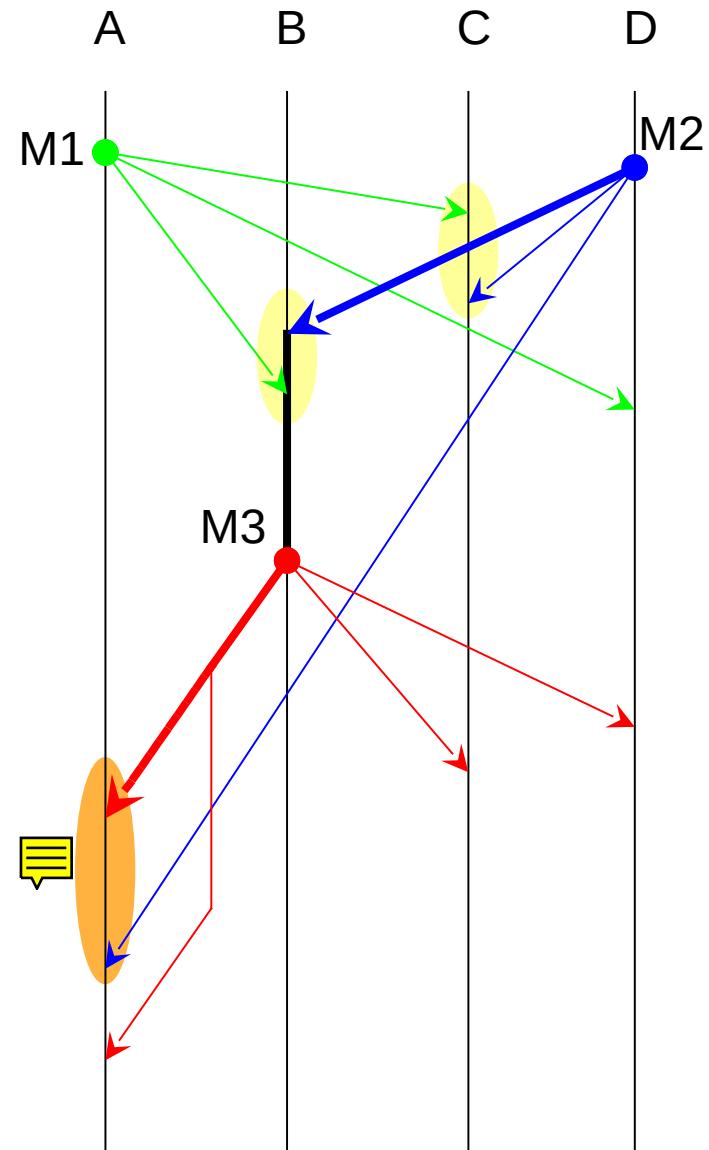
- ◆ kauzálně vázané zprávy ve správném pořadí
- ◆ konkurenční zprávy v libovolném pořadí

■ Kauzálně vázané zprávy

- ◆ obsah vázané zprávy **může být** ovlivněn
- ◆ není podstatné, jestli obsah **byl** ovlivněn

■ Konkurenční zprávy

- ◆ ty, které nejsou kauzálně vázané



Kauzální doručování

Kauzální závislost (*opakování je matkou ...*)

→ (kauzálně předchází) je relace definovaná:

- jestliže $\exists p: e1 \rightarrow_p e2$, potom $e1 \rightarrow e2$
- $\forall m: \text{send}(m) \rightarrow \text{recv}(m)$
- jestliže $e1 \rightarrow e2 \ \& \ e2 \rightarrow e3$ potom $e1 \rightarrow e3$

Kauzální uspořádání doručovaných zpráv

$\text{dest}(m)$ množina uzlů, kterým je zaslána zpráva m

$\text{deliver}_p(m)$ je událost doručení zprávy m uzlu p

- $m1 \rightarrow m2 \Rightarrow \forall p \in (\text{dest}(m1) \cap \text{dest}(m2)) : \text{deliver}_p(m1) \rightarrow_p \text{deliver}_p(m2)$

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

Vektorové hodiny

vektorové hodiny (vektorová časová značka, vector time, VT)

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
- $\forall \text{ send } p_i(m): VT(m) = ++VT(p_i)[i]$
- p_j při doručení m upraví $VT(p_j)$:

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

Inkrementace položky \approx
odesílající uzel

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 - konkurenční

odeslání zprávy m uzlem p_i

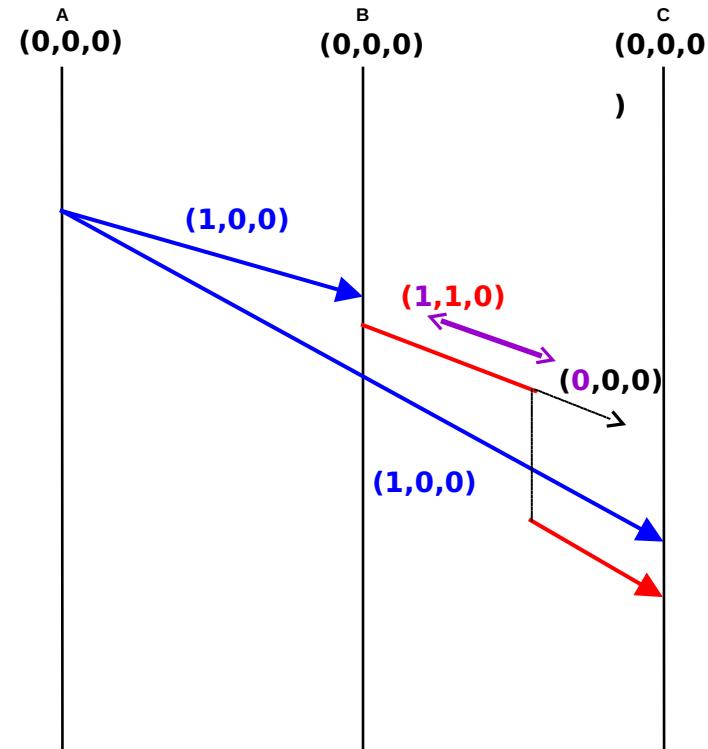
- $VT(p_i)[i]++$
- $VT(m) = VT(p_i)$

podmínky doručení

přijetí uzlem p_j

- proces $p_j \neq p_i$ pozdrží doručení m dokud:
 - ◆ $VT(m)[k] = VT(p_j)[k] + 1$ pro $k = i$
 - ◆ $VT(m)[k] \leq VT(p_j)[k]$

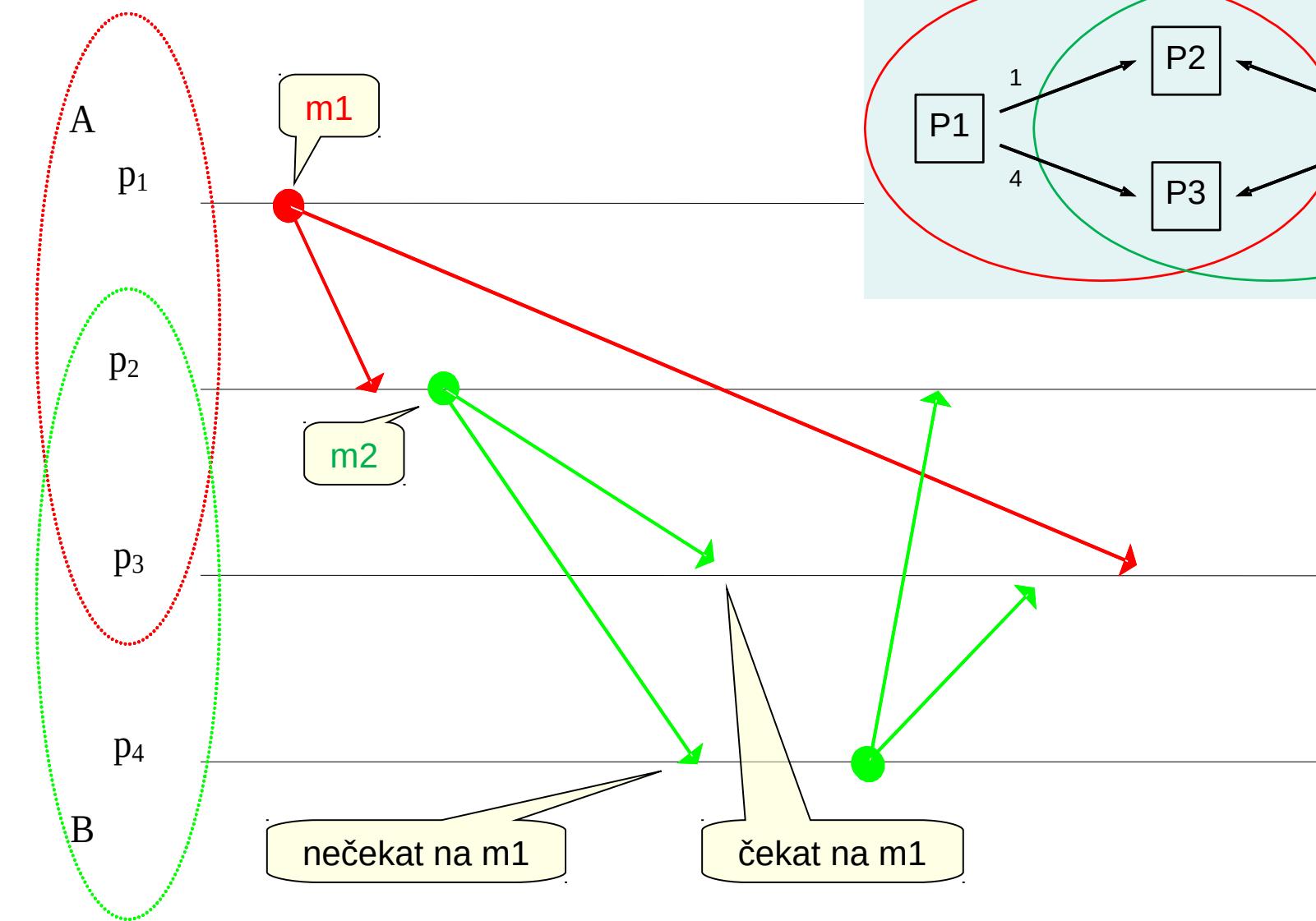
co to znamená?



doručení

- po doručení si uzel 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**

VT_a - časová značka skupiny g_a

VT_a[i] - počet multicastů uzlu p_i do g_a

S odesílanou zprávou posílá uzel časovou značku **všech** skupin, kterých je **členem**

odeslání uzlem p_i do skupiny g_a

- VT_a(p_i)[i]++
- VT(m) = U VT_b(p_i) : p_i ∈ g_b

VT všech skupin, kterých je **odesílatelem** členem

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

- uzel p_j ≠ p_i pozdrží m dokud neplatí:
 - ◆ VT_a(m)[i] = VT_a(p_j)[i] + 1
 - ◆ ∀k (p_k ∈ g_a & k ≠ i): VT_a(m)[k] ≤ VT_a(p_j)[k]
 - ◆ ∀b (p_j ∈ g_b): VT_b(m) ≤ VT_b(p_j)

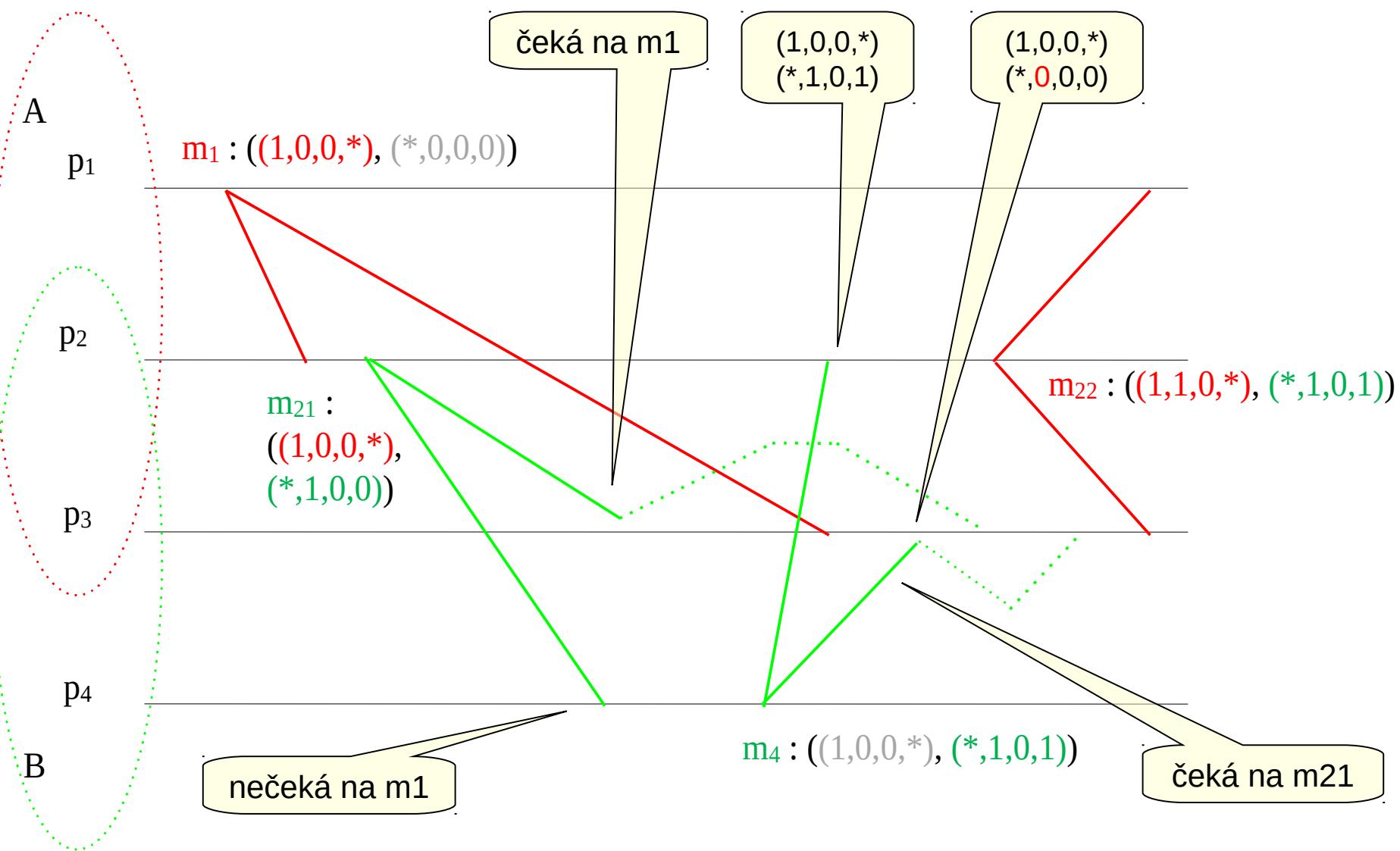
kauzalita vzhledem k skupině, do které byla **zaslána** zpráva

kauzalita vzhledem k ostatním skupinám **příjemce**

Doručení

- Po doručení si uzel p_j upraví VT

Kauzální doručování pro překrývající se skupiny

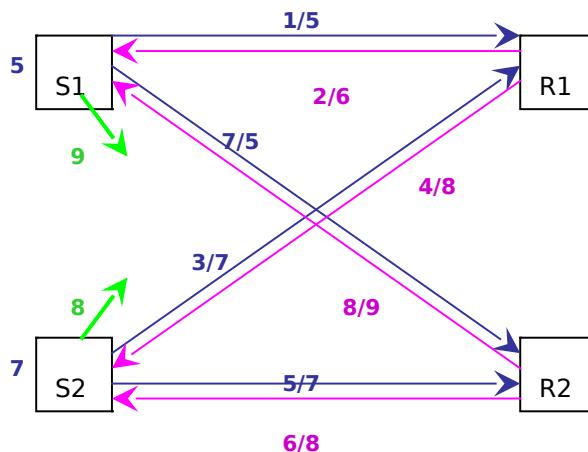


Distribuovaný total-order protokol

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

stačí skalární hodiny (časové značky) → jedno pořadí → jeden čítač
pri příjmu zprávy potvrzení odesilateli $TS(R_i)$
odesílatel po příjmu všech potvrzení odešle finalizační zprávu s $TSF = \max(TS(R_i))$

- nutnost jednoznačných TM - *byrokratické uspořádání*
po příjmu finalizační zprávy doručí příjemce zprávy podle TSF



pořadí zprávy / časová značka

msg	$TS(R_1)$	$TS(R_2)$	TSF
m0	2	2	2
m1	6	9	9
m2	8	8	8

zjednoznačnění lokálních
TS: $TS.\text{node_id}$

Komunikační složitost: $3n$

Distribuovaný total-order protokol

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

stačí skalární hodiny (časové značky)

při příjmu zprávy potvrzení odesilateli $TS(R_i)$

odesilatel po příjmu všech potvrzení odešle finalizační zprávu s $TSF = \max(TS(R_i))$

po příjmu finalizační zprávy doručí příjemce zprávy podle TSF

Sekvenční doručování - forma centralizované komponenty

Centralizované řešení efektivnější

Sekvencer

- ◆ komunikačně jednodušší
- ◆ nutnost výběru koordinátora
- ◆ serializace/uspořádání dle příjmu sekvencera
- ◆ možnost prioritního doručování

nemá smysl řešit centralizovaný problém ...



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

■ 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 zjistí které uzly zprávu přijaly?

■ Záplavový (flooding) algoritmus

- ◆ Při příjmu každé doposud nepřijaté zprávy ji každý uzel přepošle všem ostatním
- ◆ nakonec všichni zbývající členové zprávu přijmou
- ◆ spolehlivý, neefektivní

základní myšlenka:
přeposílat jen potřebným

■ 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 zjistí které uzly zprávu přijaly?

■ 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 zjistí které uzly zprávu přijaly?

Trans algoritmus

Trans - protokol pro spolehlivé kauzální doručování

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 $(m, m') \in G$

Zdrojem informací pro negativní potvrzování je G (nepřijaté zprávy)

orientace
ve směru potvrzení,
ne kauzality!

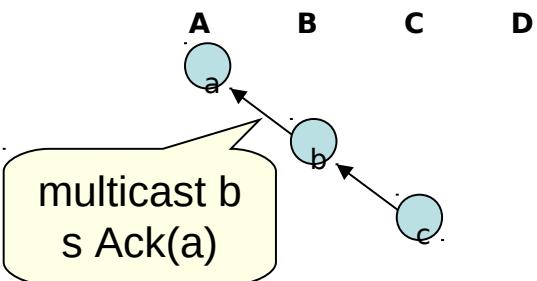
Implementace - 3 komponenty:

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

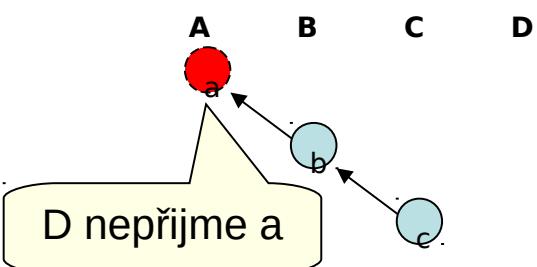
Potvrzovací mechanismy

■ Jak pořídit, aby uzly zjistily, které zprávy přijaly?

- ◆ *flooding*: 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 korektném kauzálním doručování jsou potvrzení zpráv tranzitivní

Trans algoritmus - odesílání

Implementace

- ack_list, nak_list - seznam zpráv pro potvrzení / nepřijatých zpráv
- undelivered_list - seznam přijatých ale ještě nedoručených zpráv
- přeposlání zprávy včetně ack/nak-listu

Trans_send(m)

```
m += nack_list  
m += ack_list  
ack_list = m  
G += m  
send m to every node
```

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
```

přeposlání vyžádaných zpráv všem

```
if n ∈ Gp
```

včetně ACK!

```
multicast n
```

```
if m ∈ Gp
```

```
exit
```

```
foreach ack(n) ∈ m && n ∉ Gp
```

zaznamenání nepřijaté zprávy

```
nak_list += n
```

```
if m ∈ nak_list
```

```
nak_list -= m
```

všechny kauzálně předcházející
byly doručeny

```
undelivered_list += m
```

```
G += m
```

```
foreach n ∈ undelivered_list && causal(n)
```

```
undelivered_list -= n
```

```
deliver n
```

všechny kauzální zprávy
které nebyly tranzitivně potvrzeny

```
foreach n ∈ Gp && causal(n)
```

```
&& neex( n': causal(n') ) && (n', n) ∈ Gp)
```

```
ack_list += n
```

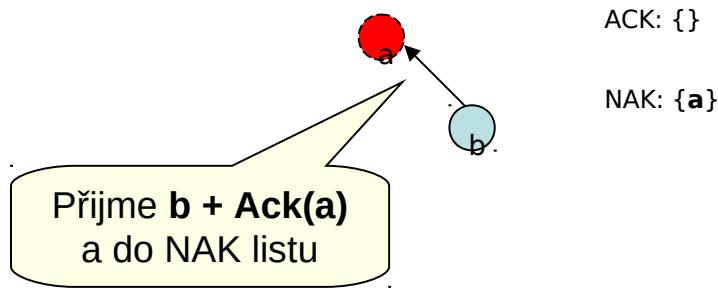
n' → n, n' potvrzuje zprávu n

```
foreach n ∈ Gp && stable(n)
```

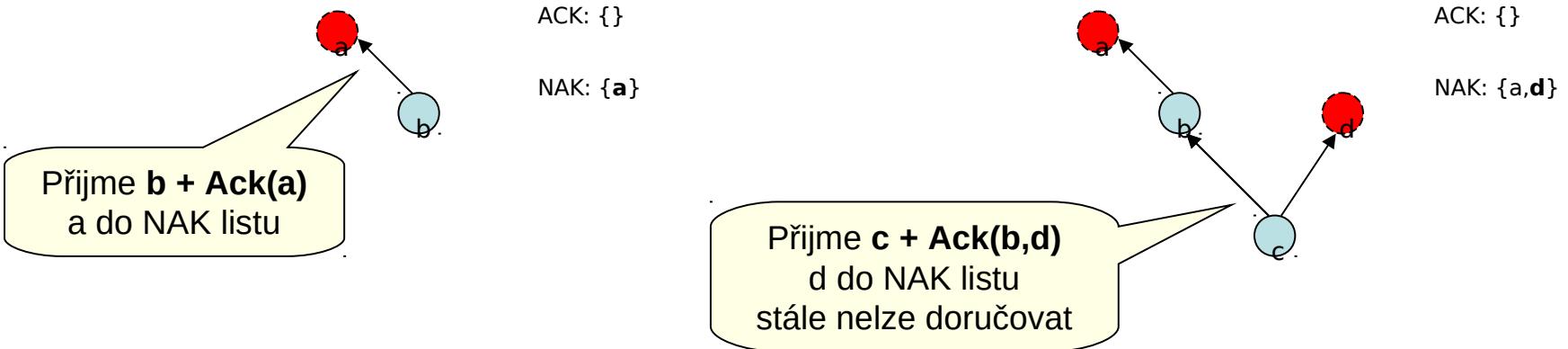
```
Gp -= n
```

vyčištění všech stabilních zpráv

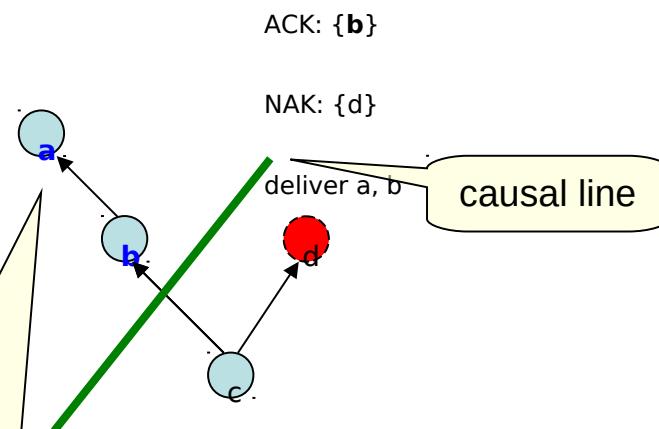
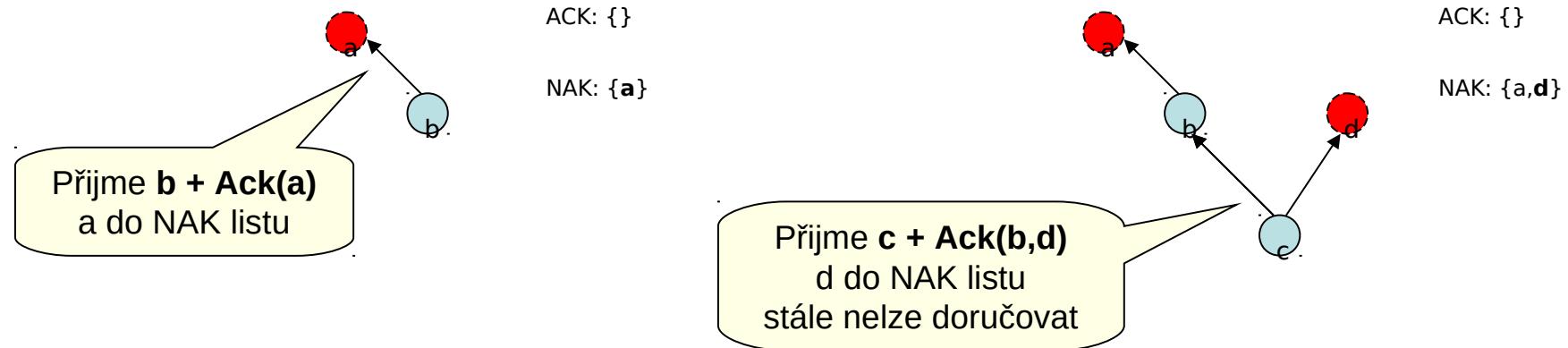
Průběh Trans algoritmu



Průběh Trans algoritmu

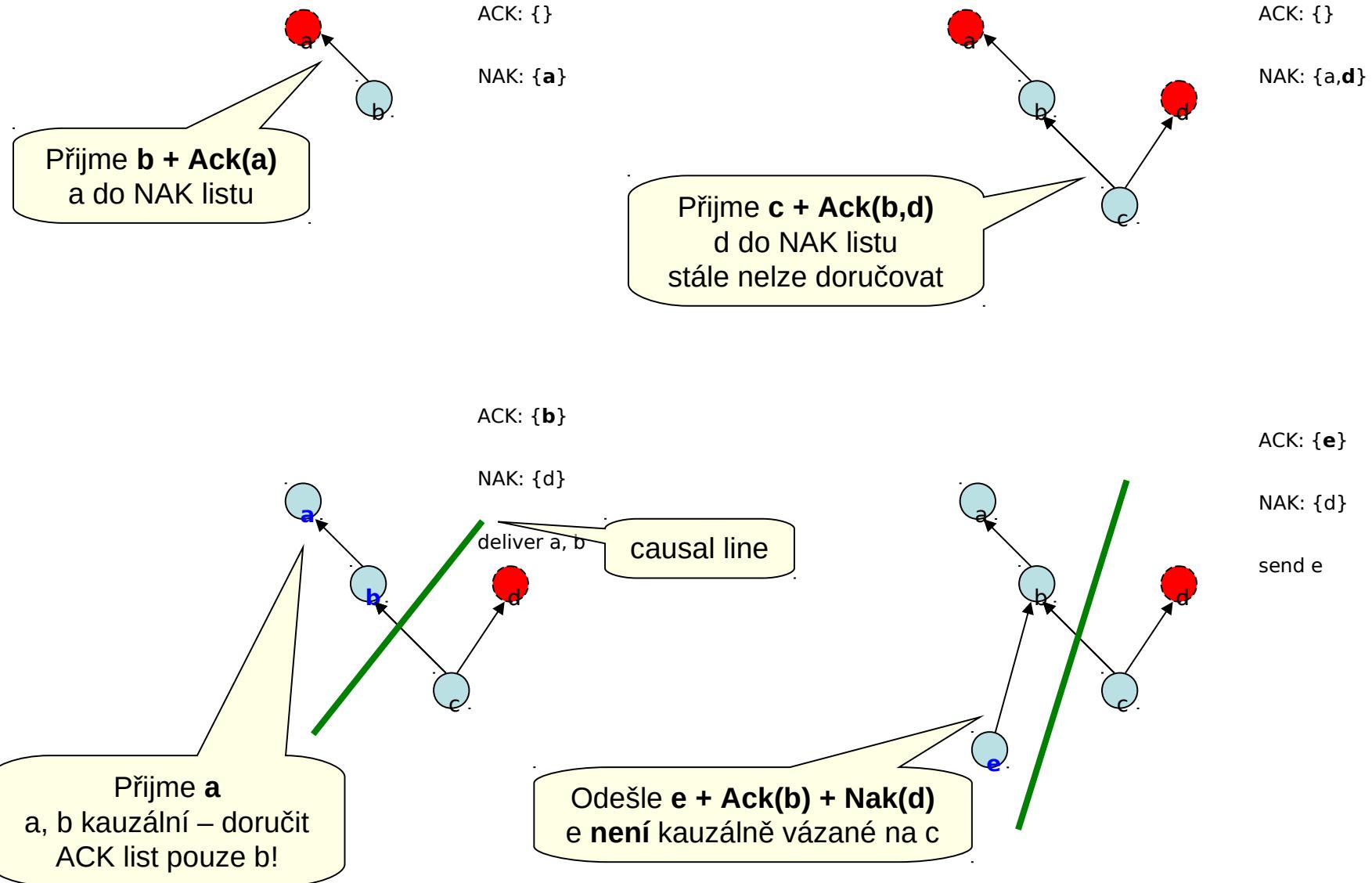


Průběh Trans algoritmu



Přijme **a**
a, b kauzální – doručit
ACK list pouze b!

Průběh Trans algoritmu



Trans protokol - pozorování

pohled na protokol - posun causal line a stable line

nově příchozí zpráva vně causal line

pokud všechny kauzálně předcházející zprávy jsou uvnitř,
pak i zprávu lze dát dovnitř causal line

Pozorování:

- je-li zpráva doručena spolehlivému uzlu,
 pak někdy každý nehavarující uzel zprávu dostane
- zprávy jsou doručovány v **kauzálním** uspořádání
- G reprezentuje graf závislosti v kauzálním uspořádání
- všechny uzly vytvářejí **stejný** graf závislostí
 - ◆ **pořadí** přidávání zpráv do grafu však může být **různé**
- jestliže uzel havaruje, paměťová náročnost je **neomezená** ! 😞☞☠️
 - ◆ slabina - pro praktické použití musí být Trans protokol doplněn protokolem pro **změnu členství** ve skupinách

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

Virtuální synchronie

- ◆ $p, q \in L^x, L^{x+1}$
- ◆ $\text{install } L_p^x < \text{deliver}_p(m) < \text{install } L_p^{x+1} \Rightarrow \text{install } L_q^x < \text{deliver}_q(m) < \text{install } L_q^{x+1}$

■ View-synchronous communication

■ pokud je zpráva m odeslána skupině s L^x před změnou na L^{x+1}

- ◆ bud' 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

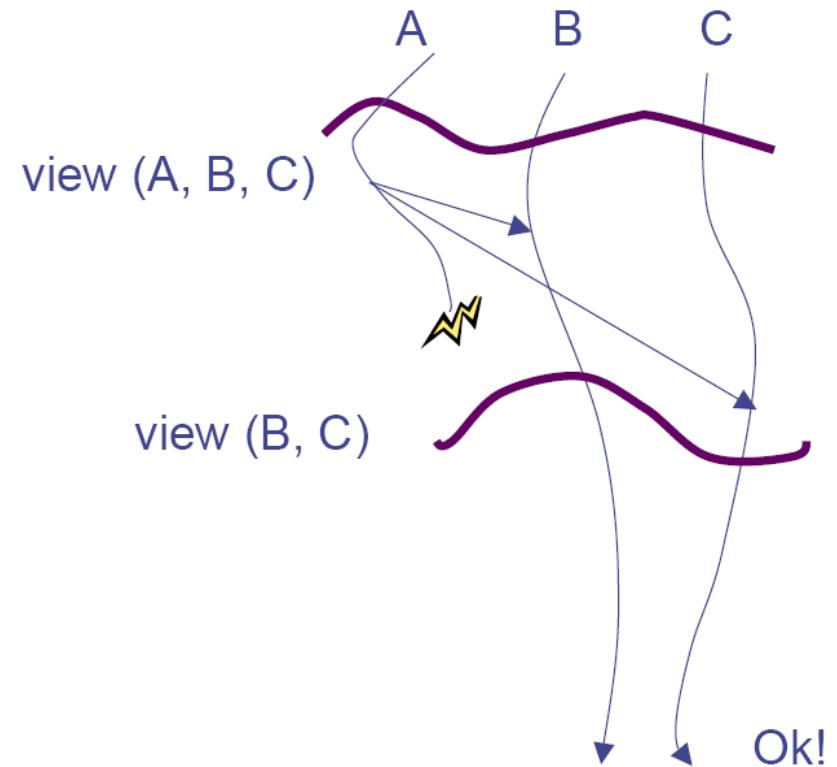
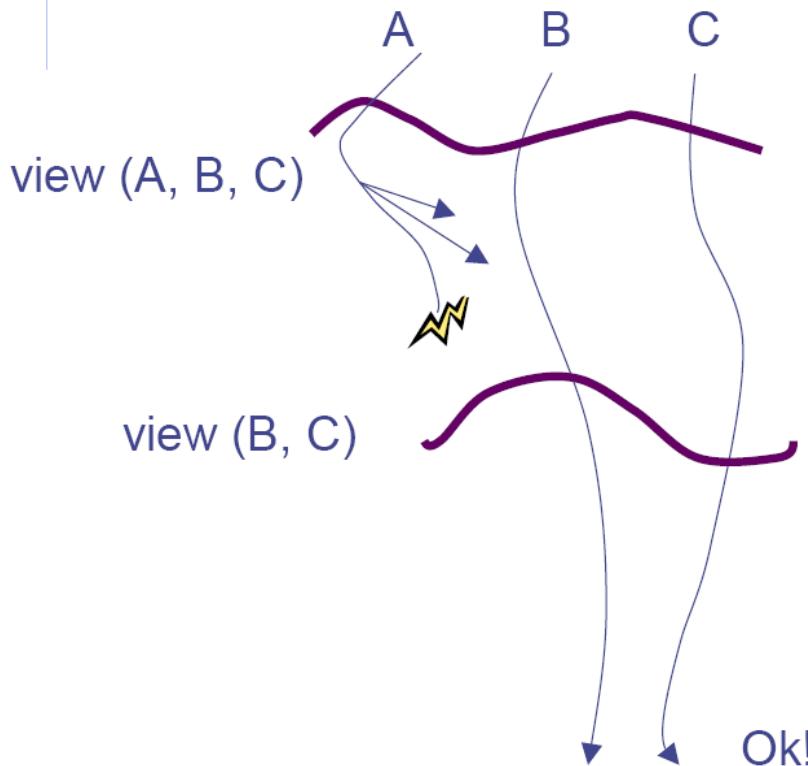
■ přesná formální definice konzistentní změny pohledů nejednotná

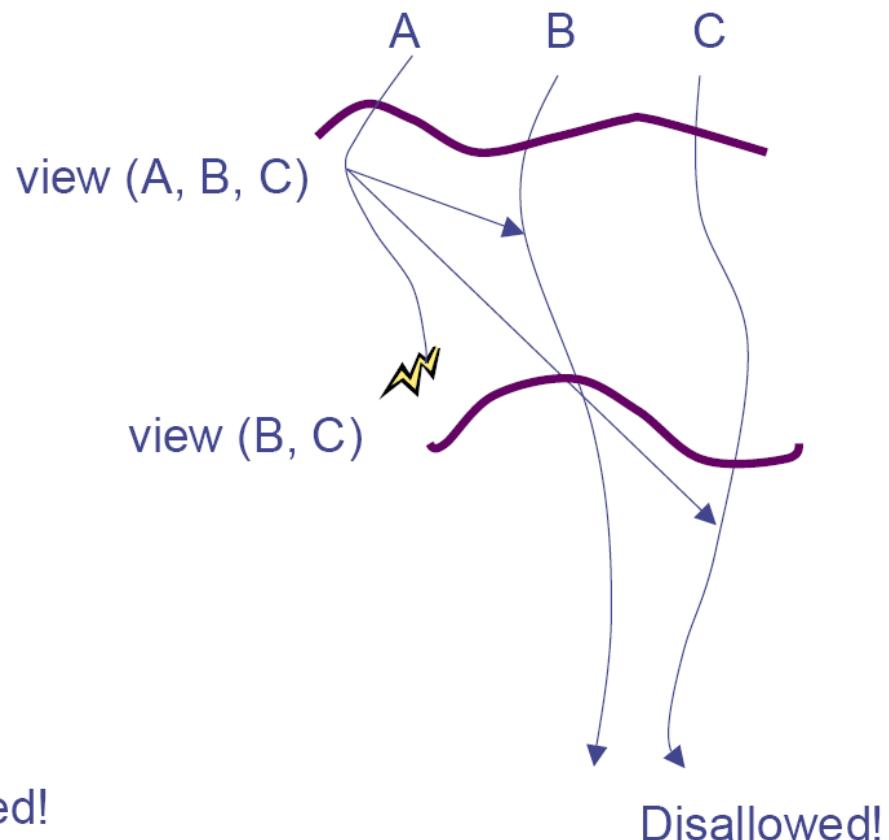
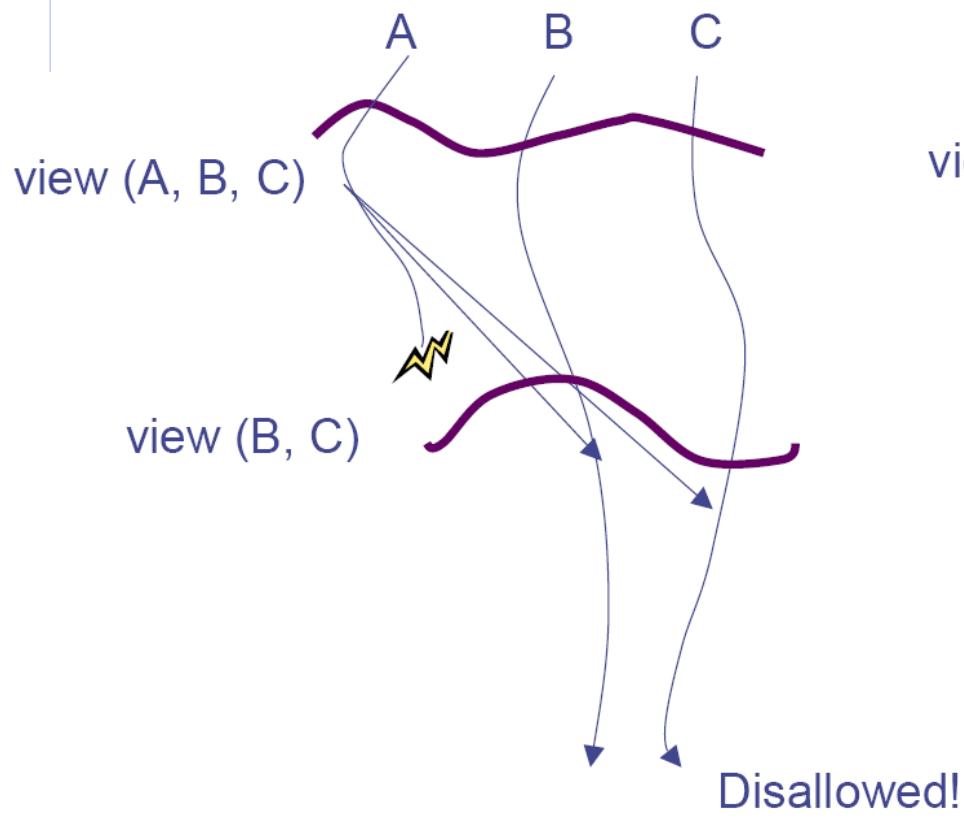
- ◆ konzistentní řezy, běhy, temporální logika, ...

■ podmínka vzájemné konzistence: $p \in Lq \Rightarrow q \in Lp$

- ◆ všechny uzly ve skupině udržují stejný L
- ◆ instalují nové pohledy ve stejném pořadí

Virtuální synchronie - přípustné doručování





Transis algoritmus

Transis - spolehlivý kauzální multicast, členství ve skupinách

- ◆ vychází z Trans
- ◆ Technion, '92-'02 - Amir, Dolev, Kramer, Malki
www.cs.huji.ac.il/labs/transis

Monotónnost protokolu

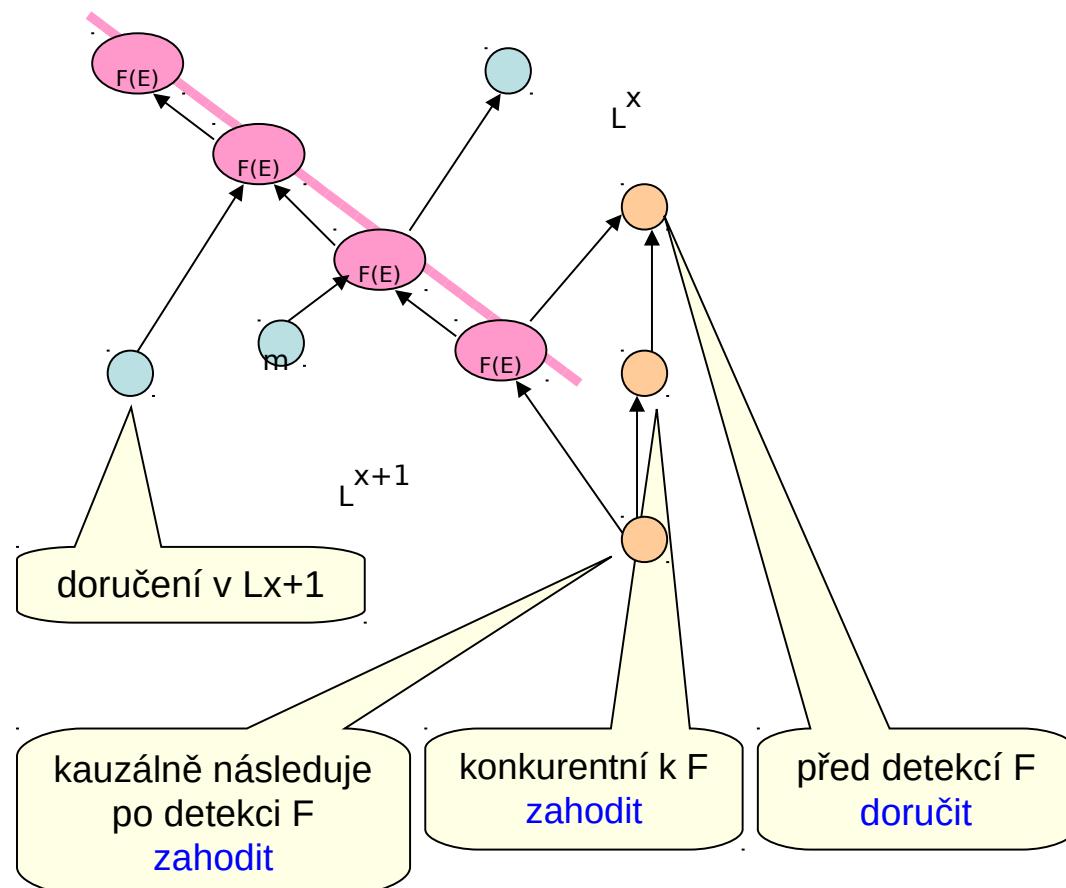
- ◆ nemožnost dosažení spolehlivého distribovaného konsensu (*později*)
- ◆ **paranoia** - jedno podezření stačí k vyloučení
 - k vyloučení stačí větší zpoždění - nerozeznatelné od havárie - *asynchronní*
- ◆ **jednosměrnost** - vyloučené procesy se již do algoritmu nevrací
 - lze se ale explicitně znovu 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

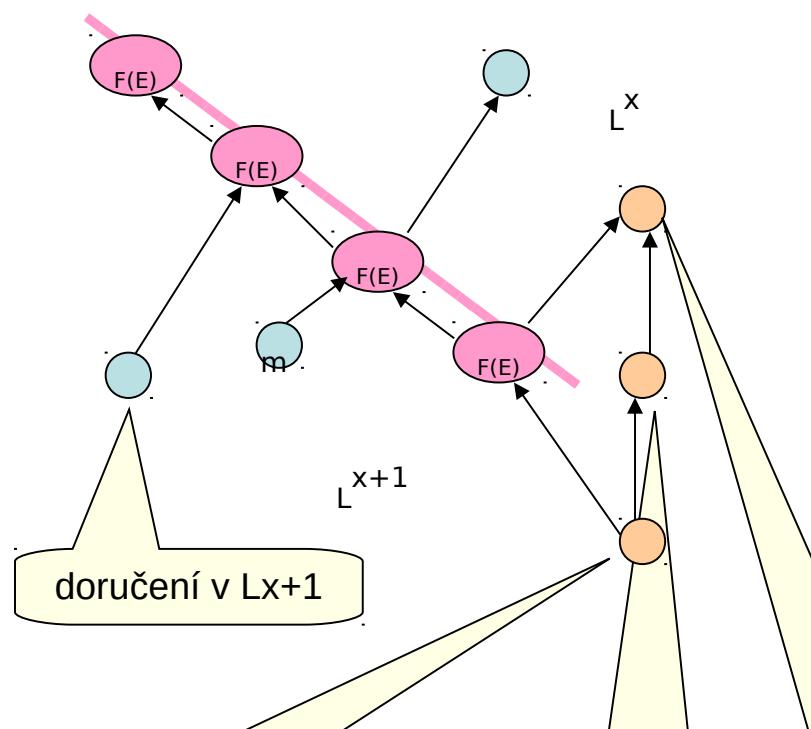
Průběh Transis algoritmus

A B C D E



Průběh Transis algoritmus

A B C D E

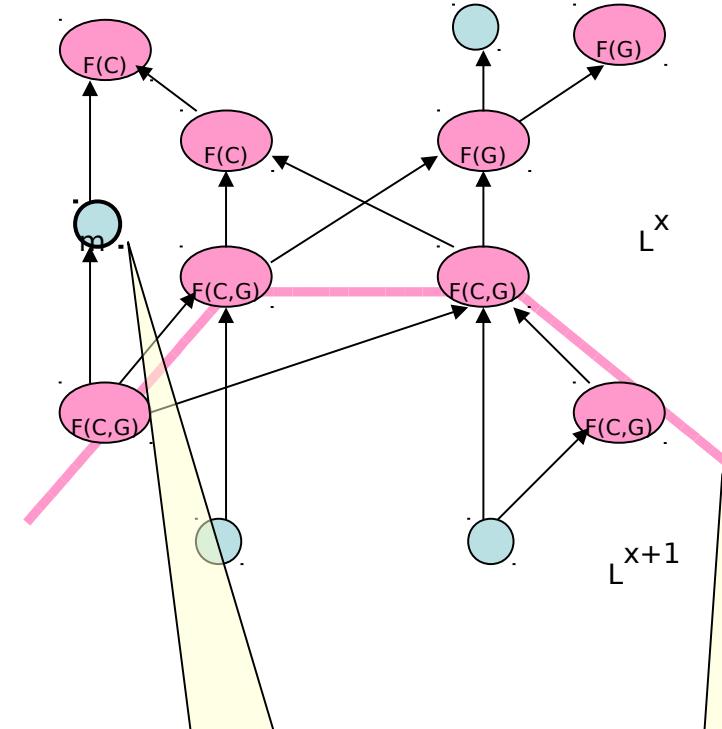


kauzálně následuje
po detekci F
zahodit

konkurenční k F
zahodit

před detekcí F
doručit

A B C D E G



nejdřív pozdržet
doručit ale v L^x

causal line
instalace pohledu

Transis algoritmus

- G graf kauzálních závislostí
stejný jako u Trans: m potvrzuje zprávu $m' \Rightarrow (m, m') \in G$
- L aktuální pohled (množina uzlů/procesů)
- F množina uzlů označených za havarované
- FAULT(q) zpráva s detekcí havárie uzlu q
- blocked blokované zprávy \rightarrow doručení v následujícím pohledu
- Last[i] uzly označené procesem i za havarované
- J uzly přidávané do pohledu
- JLast[i] uzly naposledy označené uzlem i za přidávané

přidávání uzlů do pohledu symetrické

Transis_failure(q)

$F += q$

Last[self] = F

Trans_send(FAULT, F)

detekce havárie uzlu q

Transis_deliver(m , sender)

doručení zprávy

Transis_new(q)

Transis_deliver_join(m , sender)

ekvivalent failure

F/J, Last/JLast

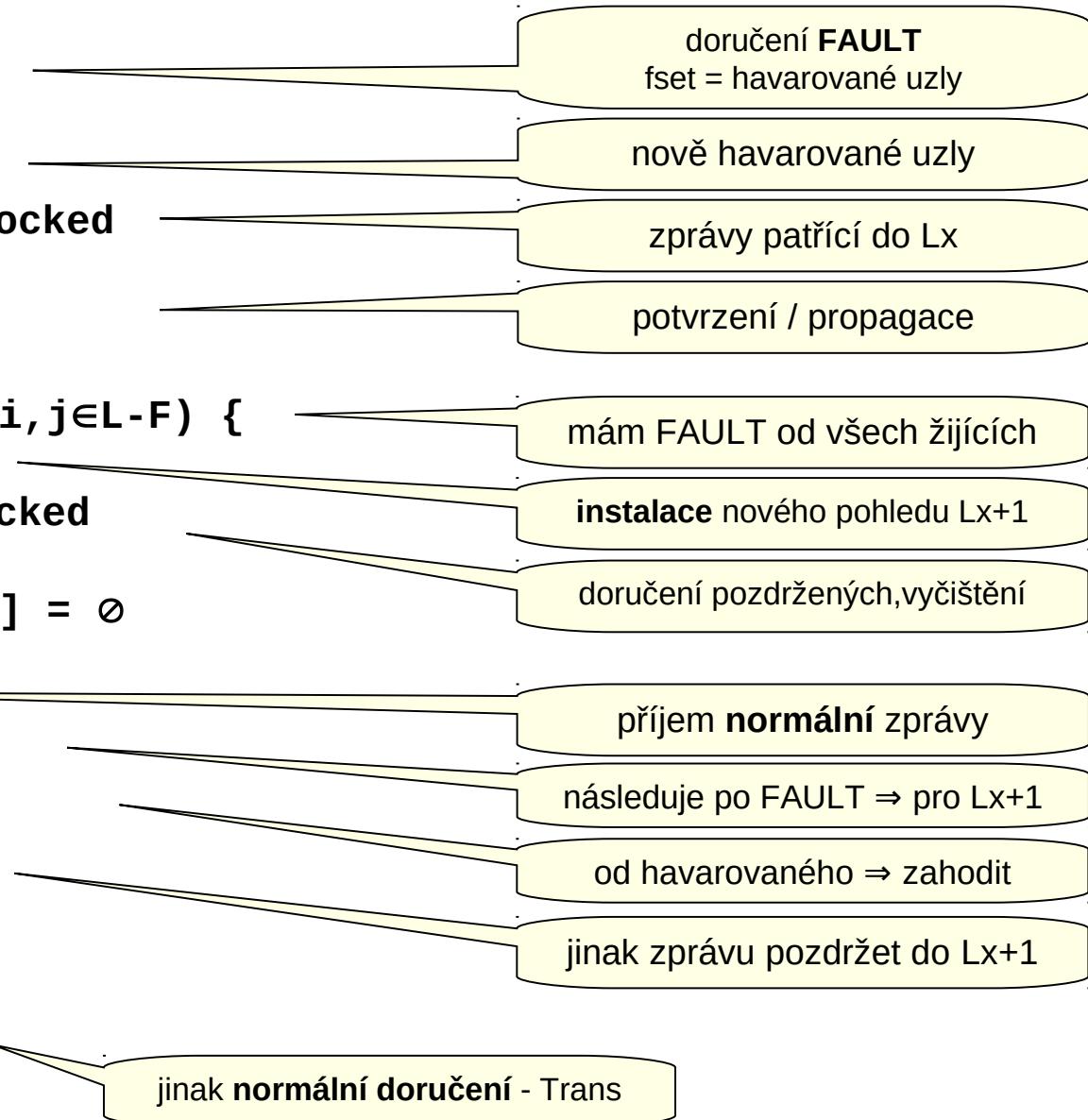


Transis algoritmus - odebrání uzlu, doručení

```

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

```



ISIS protokol - spolehlivé kauzální doručování

Základem doručovacích protokolů je reprezentace informace o doručení

- ◆ Trans - kauzální potvrzení
- ◆ ISIS - **maticové hodiny** (MT)

jiné využití

[opakování] vektorové hodiny:

- ◆ $VT_{p_i}[i]$ vlastní odeslané zprávy uzlu p_i
- ◆ $VT_{p_i}[k]$ pro $k \neq i$ zprávy přijaté od ostatních uzelů

idea: každý uzel zná $VT[]$ všech uzelů - udržuje matici

- ◆ $MT_{p_i}[j][k]$ co uzel p_i ví o doručení zpráv uzlu p_j od p_k

p_i při příjmu zprávy od p_j aktualizuje:

- $MT_{p_i}[i][j] = VT_m[j]$ co ví příjemce o sobě
- $MT_{p_i}[j][*] = VT_m[*$] co ví příjemce o odesílateli

■ stabilní zprávy

- = přijaté všemi členy skupiny
- doručeny VT od všech členů skupiny

■ srovnání ISIS a Trans

- Trans lze považovat za formu komprimace MT
- .. ale vyžaduje složitější datové struktury a doručovací algoritmus

ISIS protokol - členství ve skupinách

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

Každý uzel v L udržuje zprávu dokud není stabilní

detekce pomocí MT

Po příjmu zprávy o instalaci nového pohledu uzel:

- ◆ přepošle všechny nestabilní zprávy
- ◆ **flush** message - potvrzení instalace
- ◆ zatím pohled **neinstaluje**

Po příjmu **flush** od **každého** uzlu může **instalovat** nový pohled

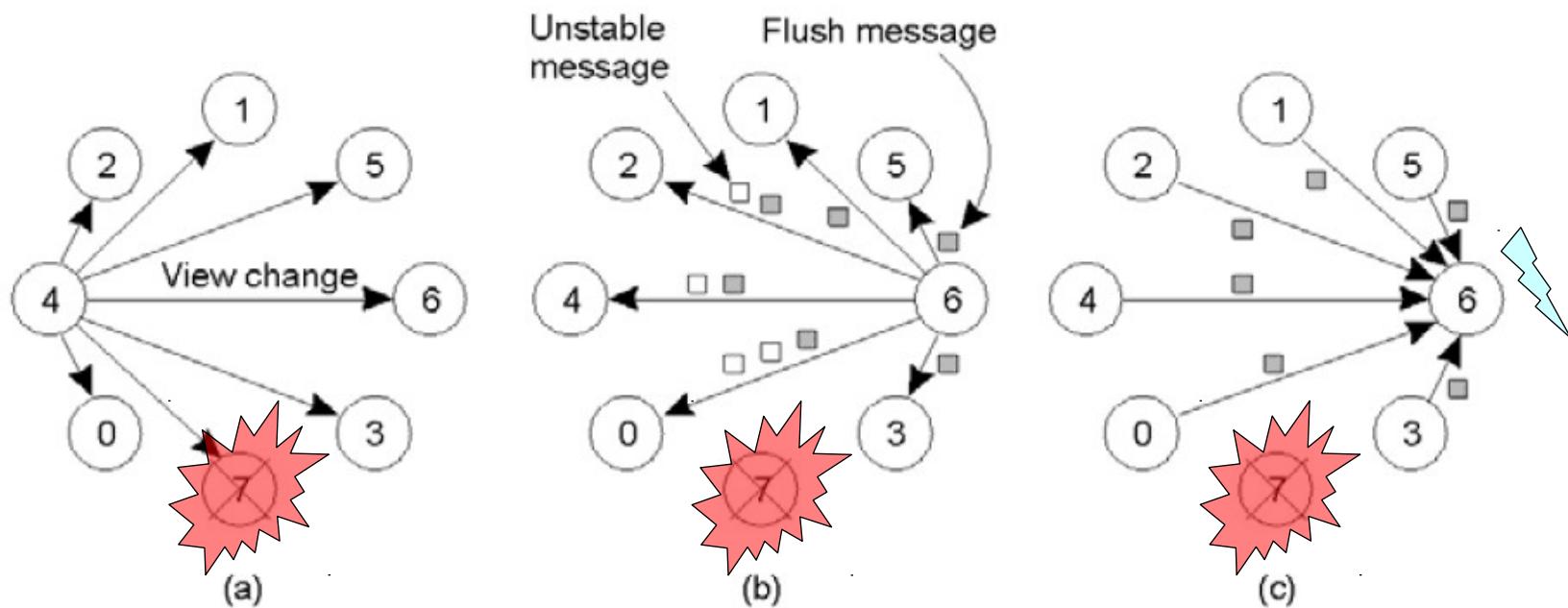
Zprávy od havarovaných uzlů

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

Reálné nasazení

- ◆ New York Stock Exchange, Swiss Stock Exchange
- ◆ French Air Traffic Control System
- ◆ Reuters, Bloomberg

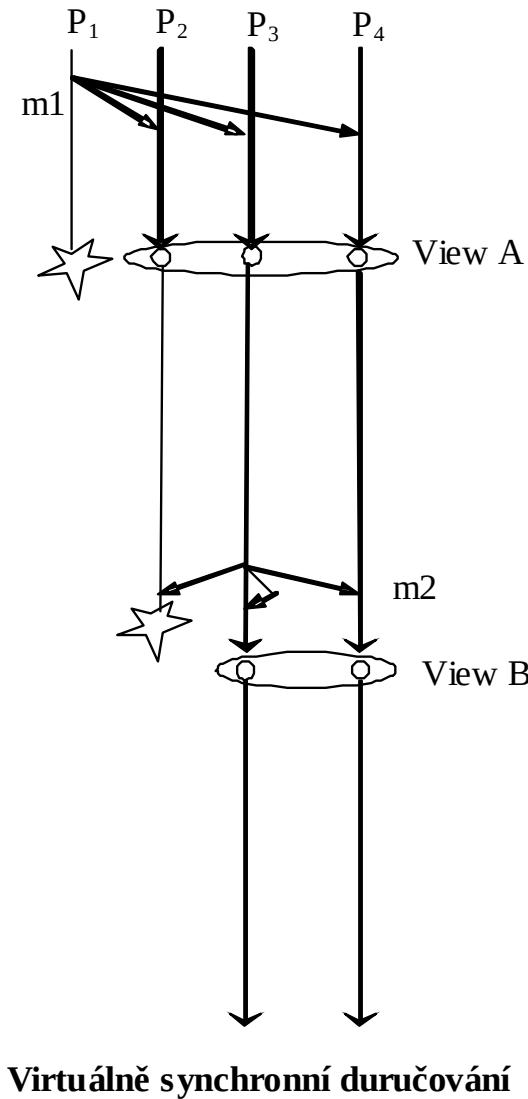
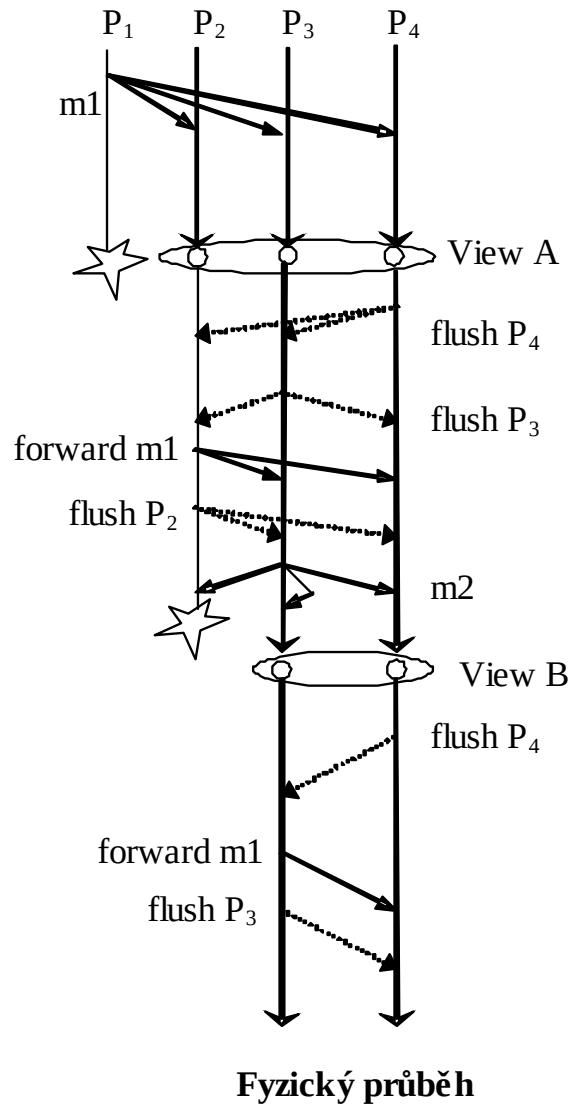
Instalace nového pohledu



Průběh instalace nového pohledu v ISIS

- proces 4 zjistil havárii procesu 7, rozesílá instalaci nového pohledu
- proces 6 rozesílá nestabilní zprávy a potvrzení instalace
- proces 6 po přijetí všech flush instaluje nový pohled

Průběh virtuální synchronie



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
virtuální synchronie, změna pohledu	Transis - tranzitivní ack Isis - maticové hodiny



Real-World Group Communication Systems

- ISIS, Horus, Ensemble, QuickSilver - Cornell University
- Spread - John Hopkins University
- JGroups
- DCS in WebSphere and AS/400 - IBM
- iBus - SoftWire Inc.
- Transis, Xpand - Hebrew University
- Cactus - Arizona State University
- Phoenix - EPFL
- Relacs - University of Bologna
- Totem - UCSB
- Amoeba - Vrije University, Amsterdam
- Appia - University of Lisboa
- NewTOP - New Castle
- HP
- Lucent Bell-Labs
- Stratus

...

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
 - ◆ nepotřebujte!
- volba koordinátora
 - ◆ detekce extrému
 - ◆ randomized race condition
- doručovací protokoly
 - ◆ sekvenční/kauzální doručování, spolehlivé doručování
 - ◆ virtuální synchronie, změny členství

Globální stav a konsensus

Ukončení distribuovaných výpočtů

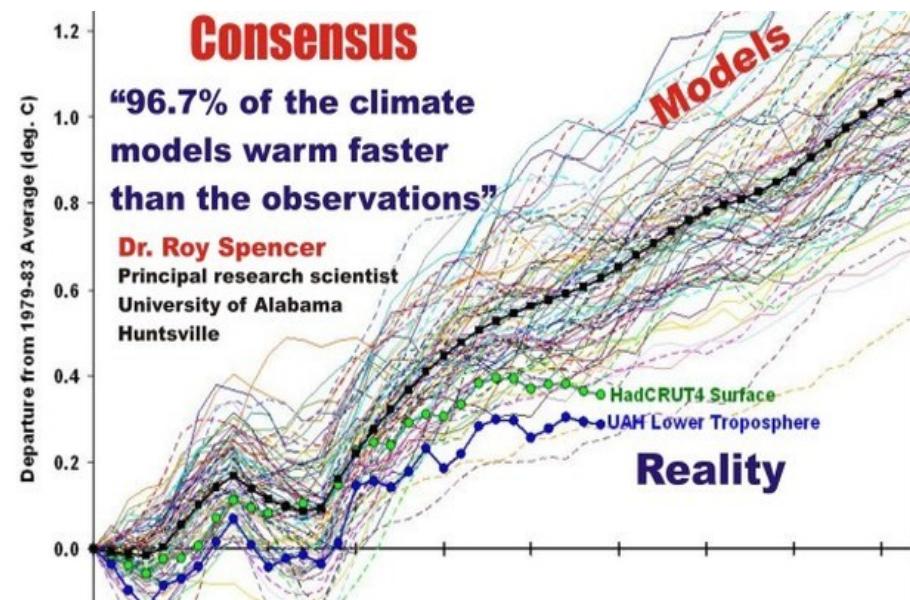
Kauzálně konzistentní globální stav

Detekce globálního stavu, značkový algoritmus

Problém dvou armád

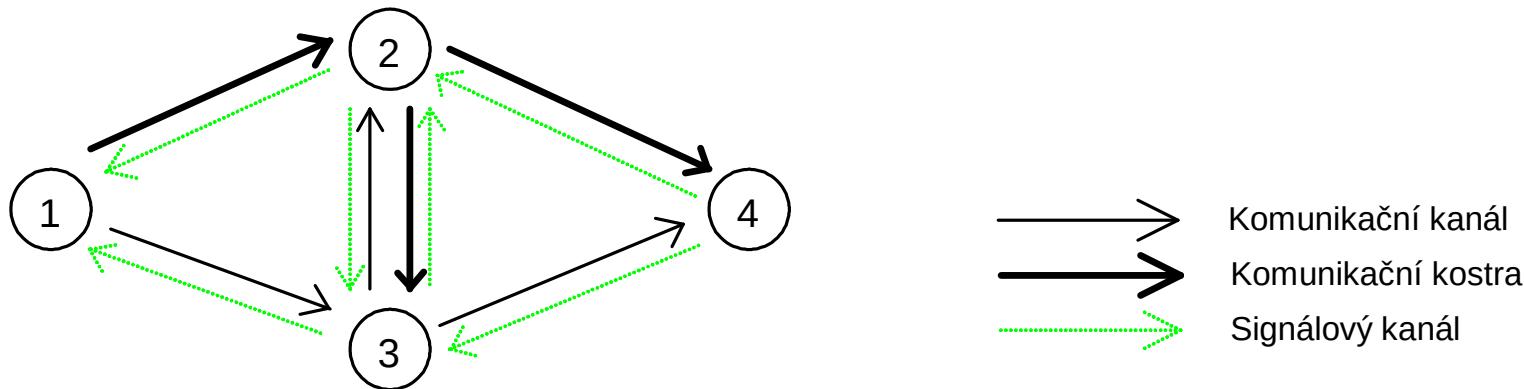
Problém Byzantských generálů

Paxos family, RAFT, ...



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 vrátí 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)

Strom

- ◆ každý listový proces při přechodu do stavu *ukončen* pošle signál svému otci
- ◆ proces dostane signály od všech svých synů, pošle signál svému otci
- ◆ iniciátor 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 **dynamicky** vytváří **orientovanou 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 signály od všech výstupních hran
 3. Poslat signál otci
- ◆ iniciátor dostane všechny signály - konec výpočtu

Strom

- ◆ každý listový proces při přechodu do stavu *ukončen* pošle signál svému otci
- ◆ proces dostane signály od všech svých synů, pošle signál svému otci
- ◆ iniciátor 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 **dynamicky** vytváří **orientovanou 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 signály od všech výstupních hran
 3. Poslat signál otci
- ◆ iniciátor dostane všechny signály - konec výpočtu

Strom

- ◆ každý listový proces při přechodu do stavu *ukončen* pošle signál svému otci
- ◆ proces dostane signály od všech svých synů, pošle signál svému otci
- ◆ iniciátor 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 **dynamicky** vytváří **orientovanou 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 signály od všech výstupních hran
 3. Poslat signál otci
- ◆ iniciátor dostane všechny signály - konec výpočtu

- Idea: dělení vah zpráv
 - každý proces má váhu, každá zpráva má váhu
 - váha iniciátora = W
 - váha ostatních procesů = 0
- Odeslání zprávy: předání části váhy zprávě
- Příjem zprávy: přičtení váhy zprávy k vlastní váze
- Proces ukončen: pošle zprávu s celou vahou procesu iniciátoru
- Konec výpočtu: váha iniciátora = W
- Problémy:
 - dělitelnost vah
 - havárie procesu / ztráta váhy

Značka - speciální zpráva odlišitelná od běžných zpráv

iniciační uzel vyšle žádost o ukončení všem výstupním hranám

- značku, že je kanál prázdný

■ příjem první značky:

- ◆ připraven - propagace značky výstupním hranám
- ◆ nepřipraven - negativní signál (zamítnutí)

■ příjem další značky: signál příchozímu kanálu / zamítnutí

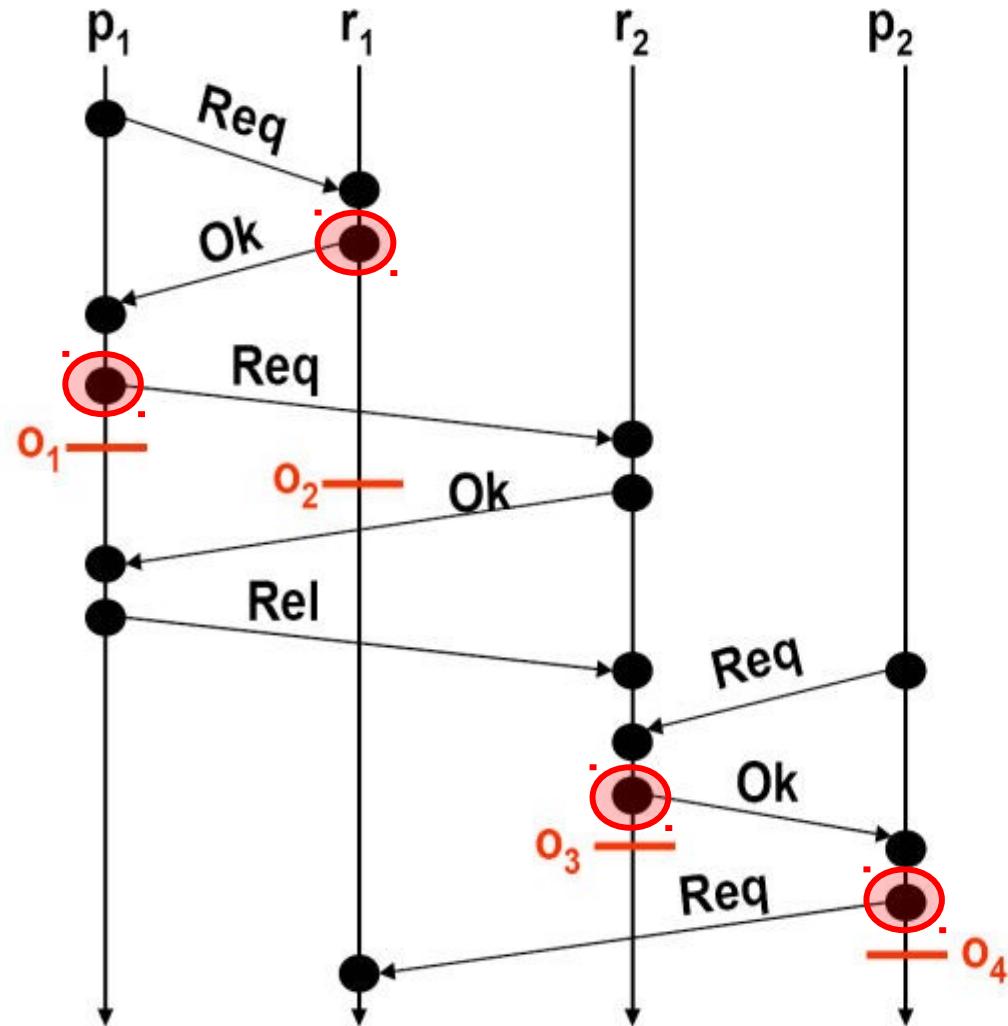
■ příjem zamítnutí: zamítnutí první žádosti

■ příjem všech potvrzení: signál první žádosti

aplikace obecnějšího algoritmu - detekce **globálního** stavu

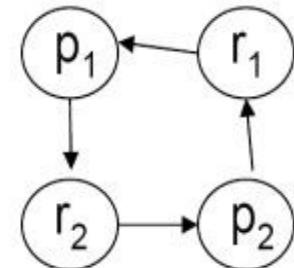
Detekce globálního stavu - motivace

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 $E = \{e\}$

Past / Future

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

(Kauzálně) **Konzistentní řez** $c : a \rightarrow b \text{ & } a \in F_c \Rightarrow b \in F_c$

Intuitivní význam: minulost nelze ovlivnit událostí v budoucnosti

Stav distribuovaného výpočtu je množina událostí, které se během výpočtu udaly

(Kauzálně) **Konzistentní stav** $S = P_c$, kde c je konzistentní řez

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

$S \xrightarrow{e} S'$ (S' je **dosažitelný** z S)

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

$S \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \dots S_{n-1} \xrightarrow{e_n} S_n$ Značíme $S \xrightarrow{s} S_n$

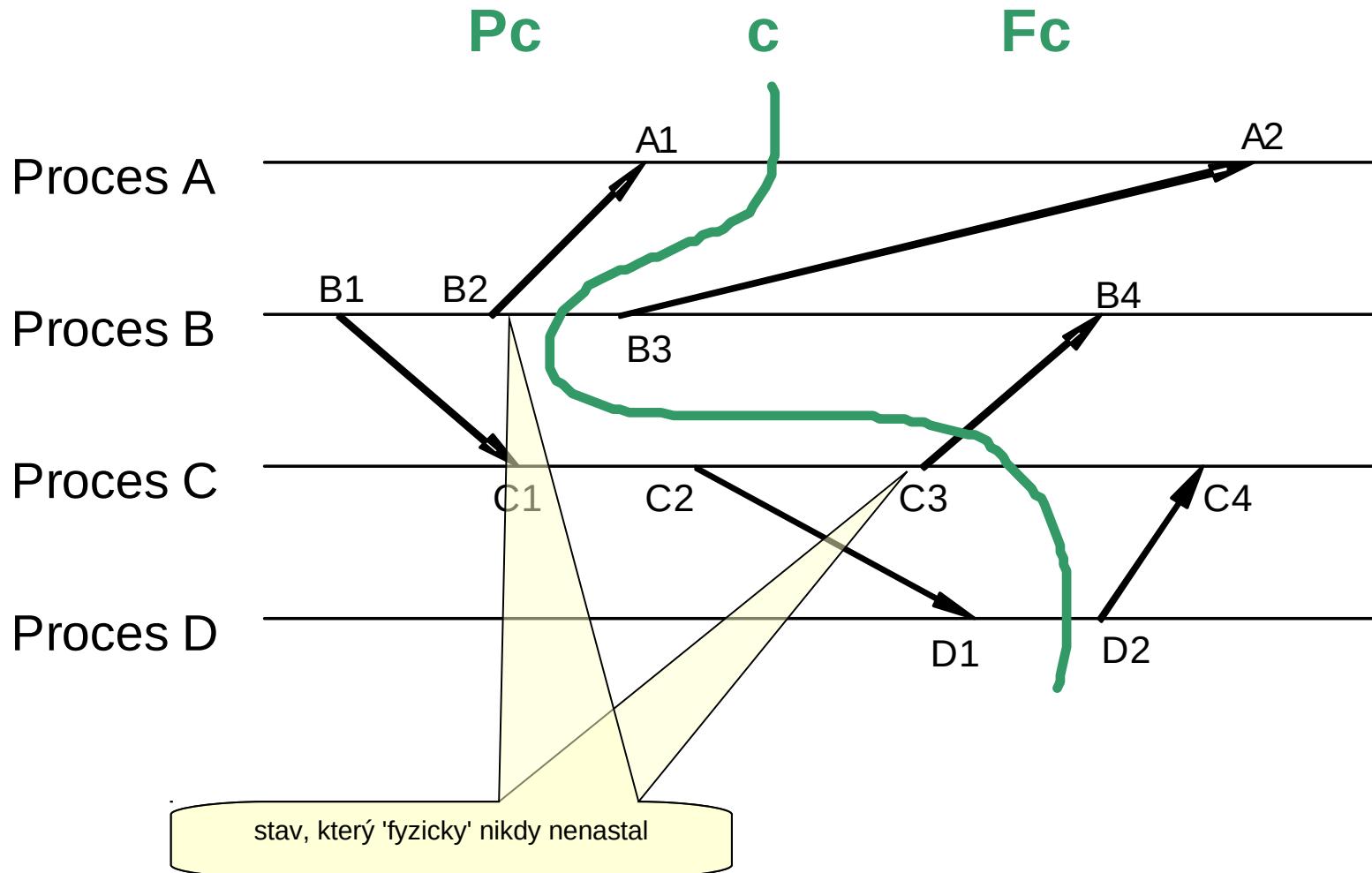
Zřejmě platí $S \xrightarrow{s} S_n \Rightarrow S \subseteq S_n$

Využití:

- ◆ obecně: detekce globálních vlastností
- ◆ detekce deadlocků
- ◆ garbage collection
- ◆ detekce ukončení výpočtů

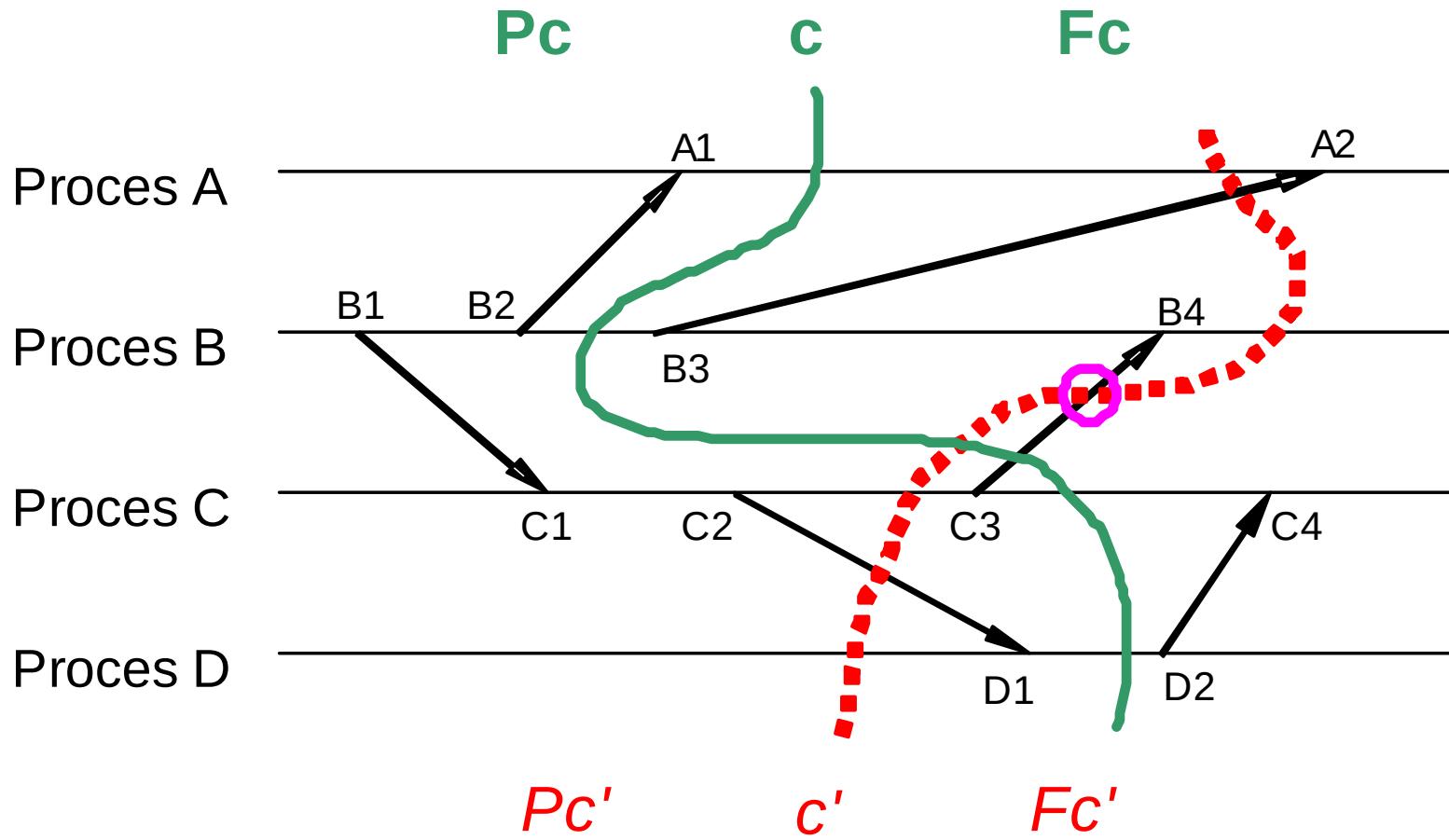
Konzistentní a nekonzistentní řez

c – konzistentní řez



Konzistentní a nekonzistentní řez

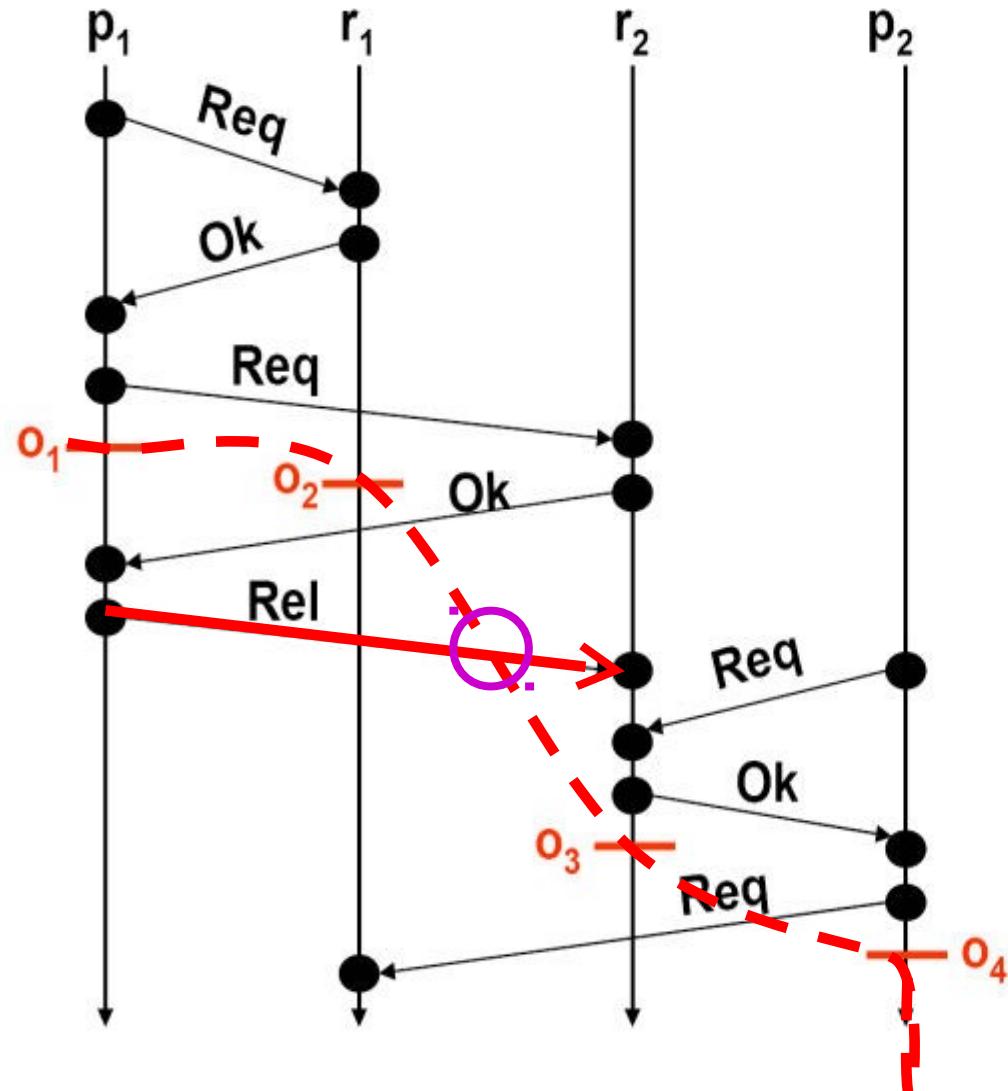
c – konzistentní řez



c' – nekonzistentní řez

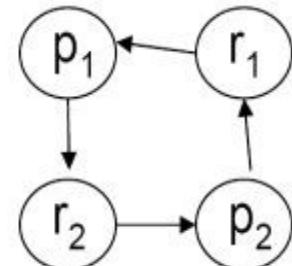
Důsledky nekonzistentního řezu

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!

stav uzlu: množina přijatých a odeslaných zpráv

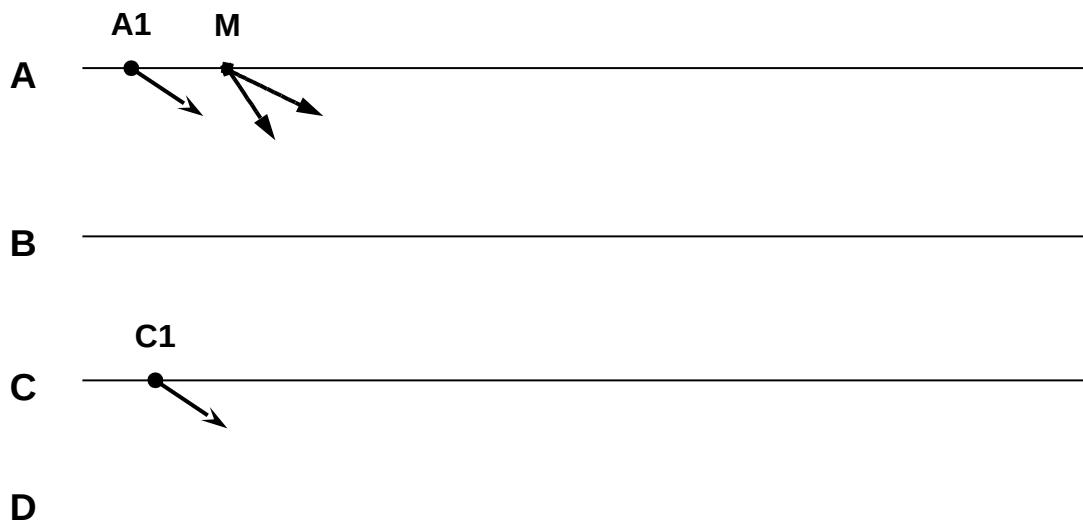
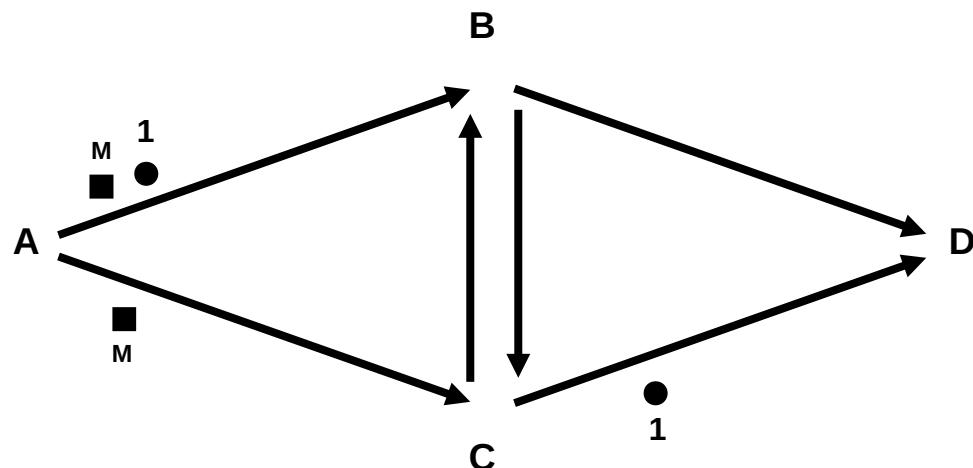
stav kanálu: množina zpráv, které byly kanálem odesány, ale ještě nebyly doručeny

Algoritmus:

- Iniciátor vyšle všem výstupním uzlům značku
 - Okamžik lokálního řezu
- 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 = zprávy došlé kanálem mezi příjmem první a této značky
- Konec algoritmu:
 - ◆ po přijmutí všech značek

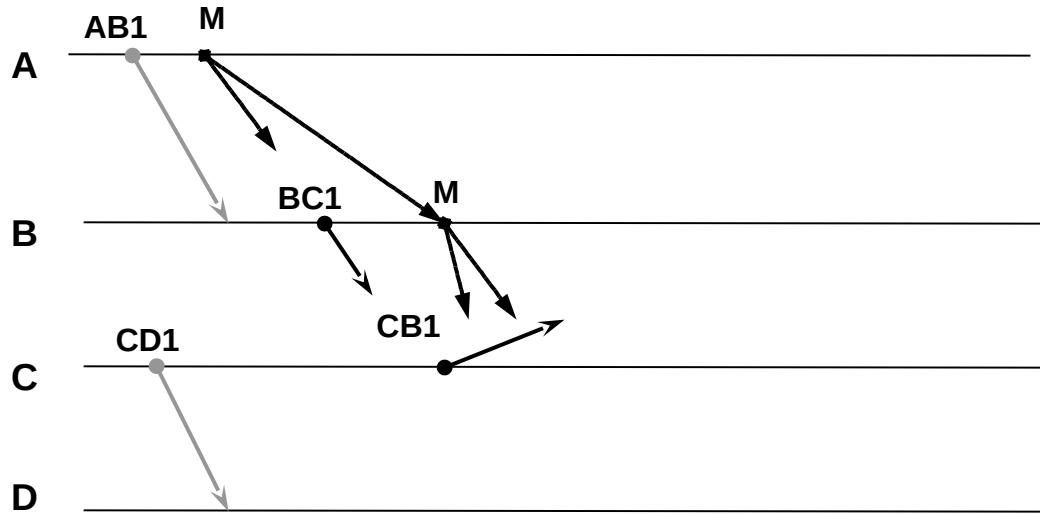
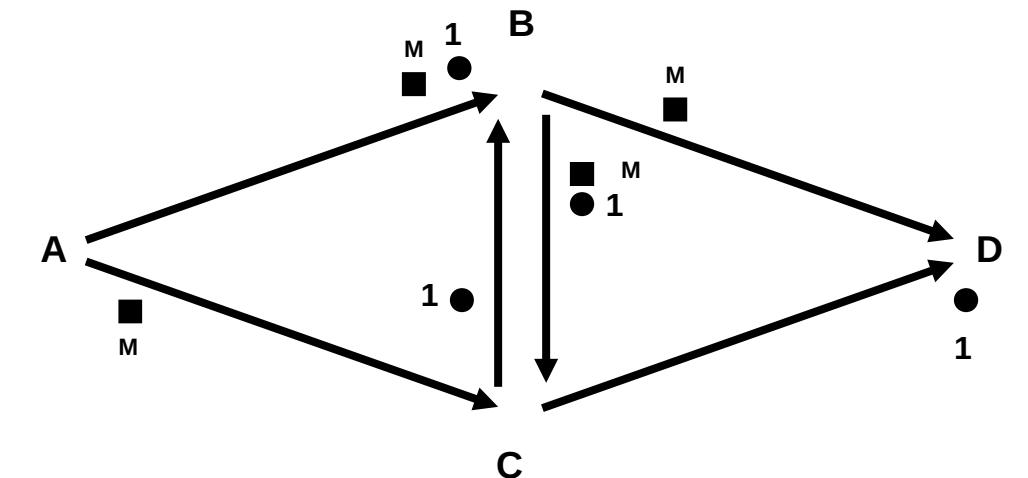
Zaznamenané stavy uzlů a kanálů definují (kauzálně) konzistentní stav systému
Chandy, Lamport (1985)

Průběh detekce GS 1



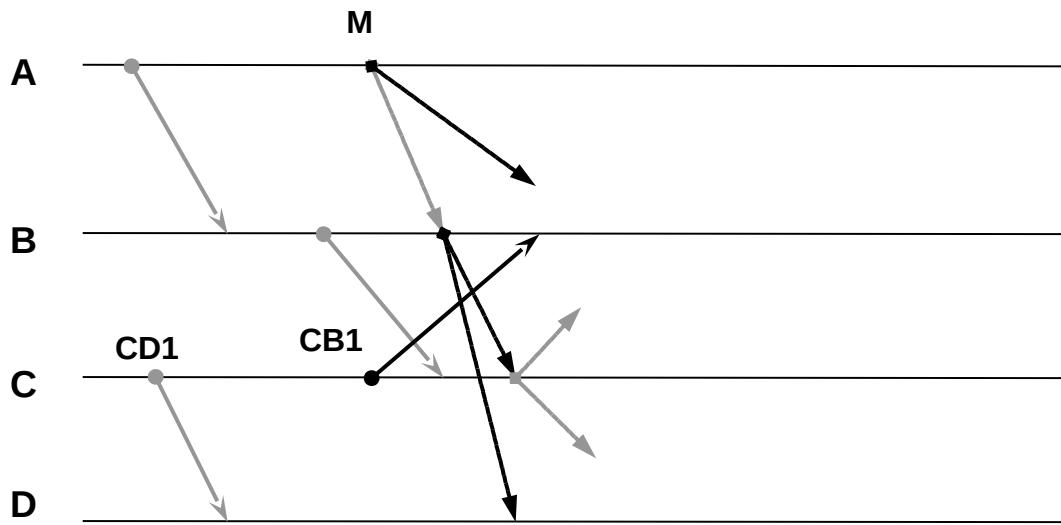
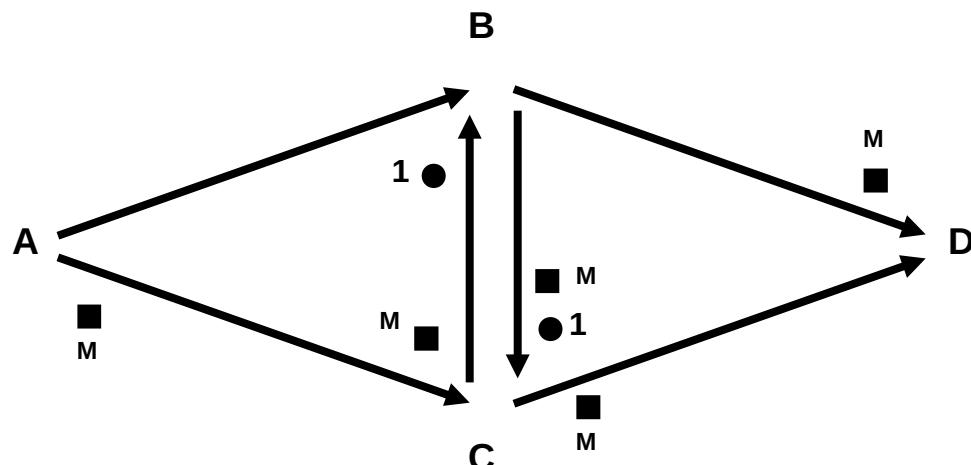
A	A→B	1
	A→C	0
B	A→B	
	C→B	
	B→C	
	B→D	
C	A→C	
	B→C	
	C→B	
	C→D	
D	B→D	
	C→D	

Průběh detekce GS 2



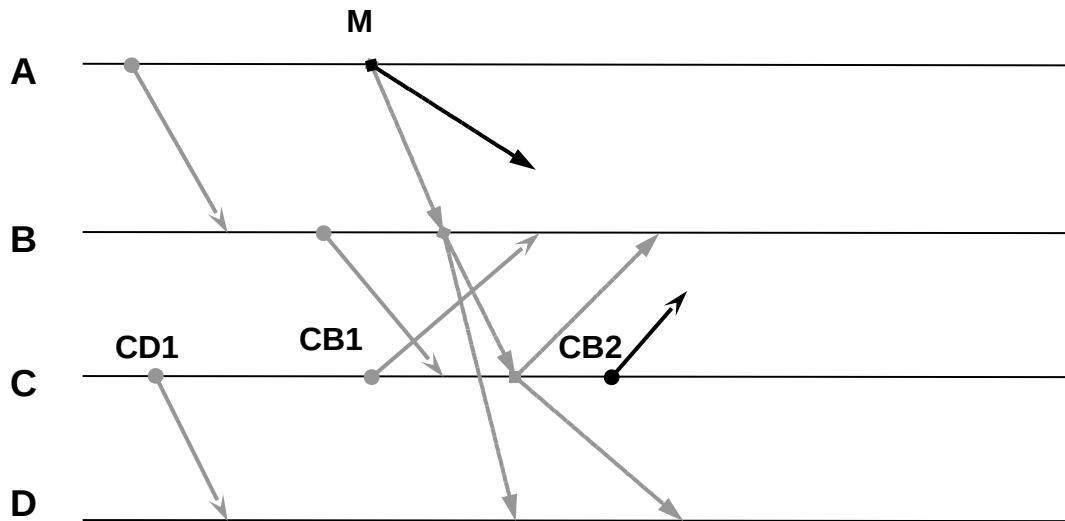
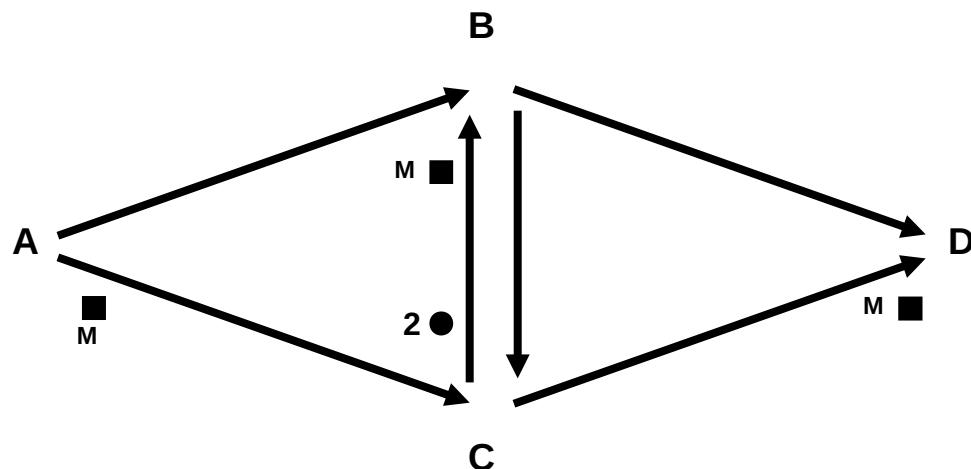
A	$A \rightarrow B$	1
	$A \rightarrow C$	0
B	$A \rightarrow B$	\diamond
	$C \rightarrow B$	
	$B \rightarrow C$	1
	$B \rightarrow D$	0
C	$A \rightarrow C$	
	$B \rightarrow C$	
	$C \rightarrow B$	
	$C \rightarrow D$	
D	$B \rightarrow D$	
	$C \rightarrow D$	

Průběh detekce GS 3



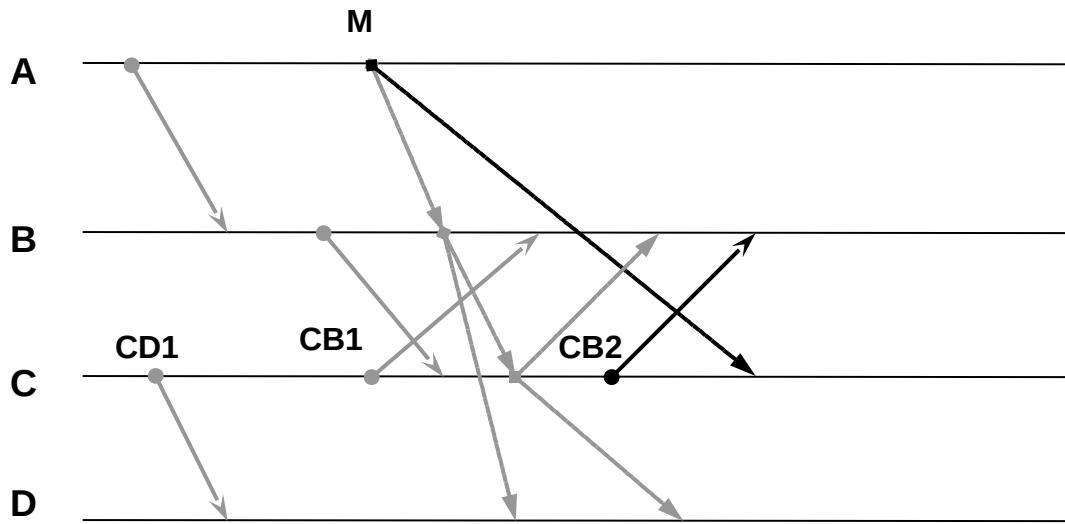
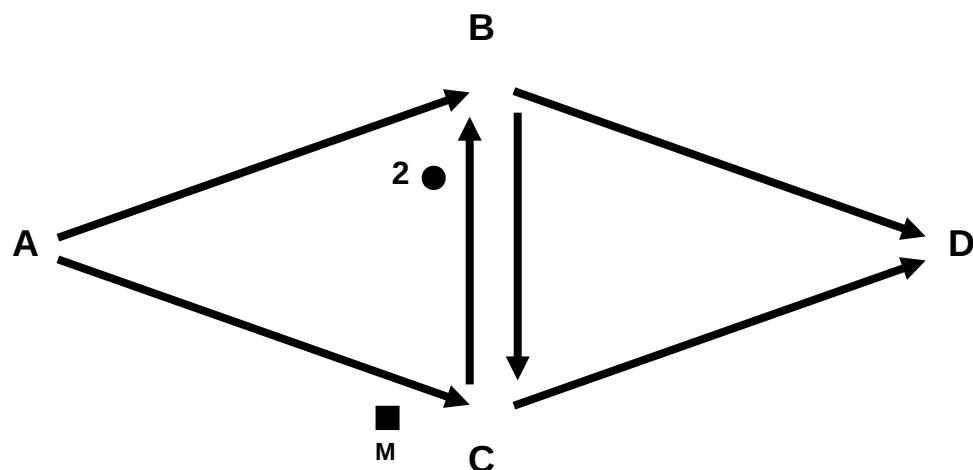
A	$A \rightarrow B$	1
	$A \rightarrow C$	0
B	$A \rightarrow B$	\diamond
	$C \rightarrow B$	1
	$B \rightarrow C$	1
	$B \rightarrow D$	0
C	$A \rightarrow C$	
	$B \rightarrow C$	\diamond
	$C \rightarrow B$	1
	$C \rightarrow D$	1
D	$B \rightarrow D$	\diamond
	$C \rightarrow D$	

Průběh detekce GS 4



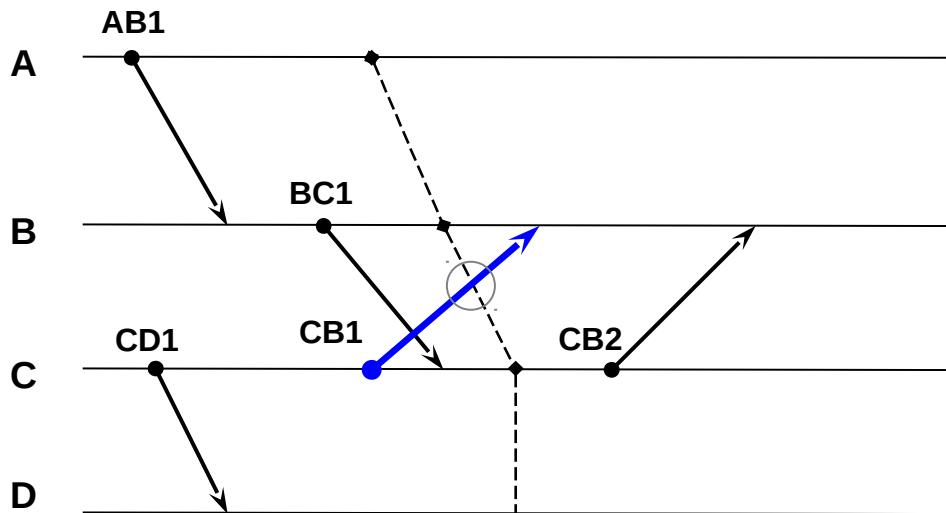
A	$A \rightarrow B$	1
	$A \rightarrow C$	0
B	$A \rightarrow B$	\diamond
	$C \rightarrow B$	$\diamond 1$
	$B \rightarrow C$	1
	$B \rightarrow D$	0
C	$A \rightarrow C$	
	$B \rightarrow C$	\diamond
	$C \rightarrow B$	1
	$C \rightarrow D$	1
D	$B \rightarrow D$	\diamond
	$C \rightarrow D$	\diamond

Průběh detekce GS 5



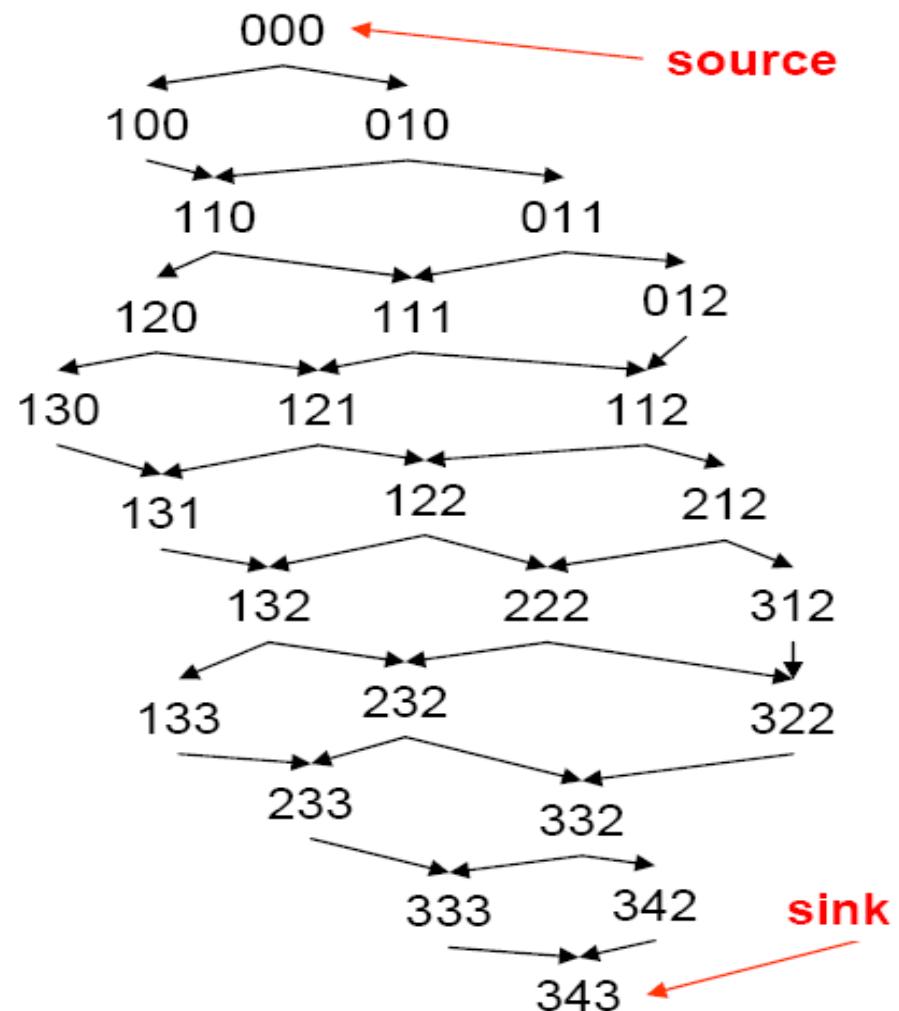
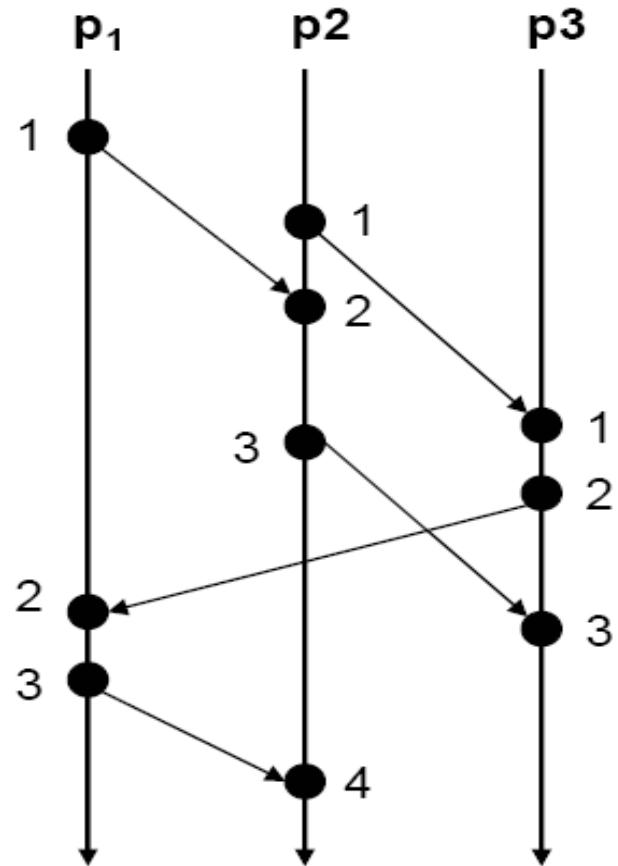
A	$A \rightarrow B$	1
	$A \rightarrow C$	0
B	$A \rightarrow B$	\diamond
	$C \rightarrow B$	$\diamond 1$
	$B \rightarrow C$	1
	$B \rightarrow D$	0
C	$A \rightarrow C$	\diamond
	$B \rightarrow C$	\diamond
	$C \rightarrow B$	1
	$C \rightarrow D$	1
D	$B \rightarrow D$	\diamond
	$C \rightarrow D$	\diamond

Průběh detekce GS - výsledek



A	$A \rightarrow B$	1
	$A \rightarrow C$	0
B	$A \rightarrow B$	\Diamond
	$C \rightarrow B$	$\Diamond 1$
	$B \rightarrow C$	1
	$B \rightarrow D$	0
C	$A \rightarrow C$	\Diamond
	$B \rightarrow C$	\Diamond
	$C \rightarrow B$	1
	$C \rightarrow D$	1
D	$B \rightarrow D$	\Diamond
	$C \rightarrow D$	\Diamond

Modely distribuovaných výpočtů



globální stavy, globální predikáty
stabilní vlastnosti, důkazy korektnosti
inevitable states - nevyhnutelné



Problém dvou armád

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

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:

(A1 neví, jestli A2 dostal potvrzení)

....

....

....

Distribuovaný konsensus



Problém dvou armád

- ◆ Početnější armáda B je rozdělena
- ◆ Úspěch pouze při synchronizovaném útoku, jinak 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:

.....

.....

formální důkaz indukcí

Problém dvou armád



Praktická řešení

- Agresivní strategie
 - první generál vyšle větší množství zpráv oznamující čas útoku
 - a zaútočí!
 - předpoklad: při vysokém počtu poslaných zpráv alespoň jedna projde
 - druhý generál nemusí ani odpovídat
- Pravděpodobnostní strategie
 - první generál vyšle větší množství (N_0) zpráv
 - kromě času útoku přidá i počet poslaných zpráv
 - druhý generál odpoví na každou zprávu, 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



Byzantine Generals Problem



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

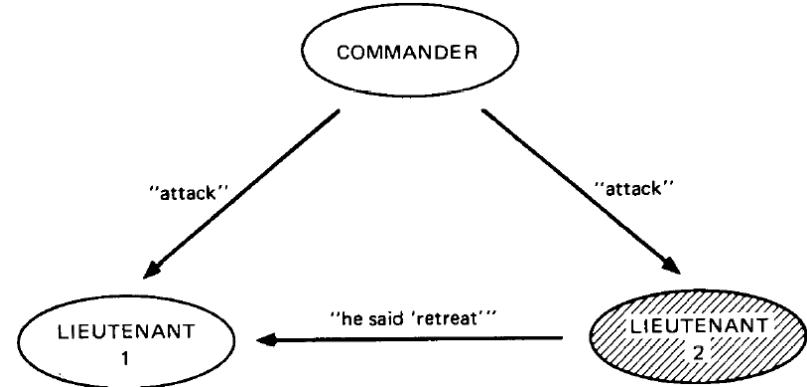
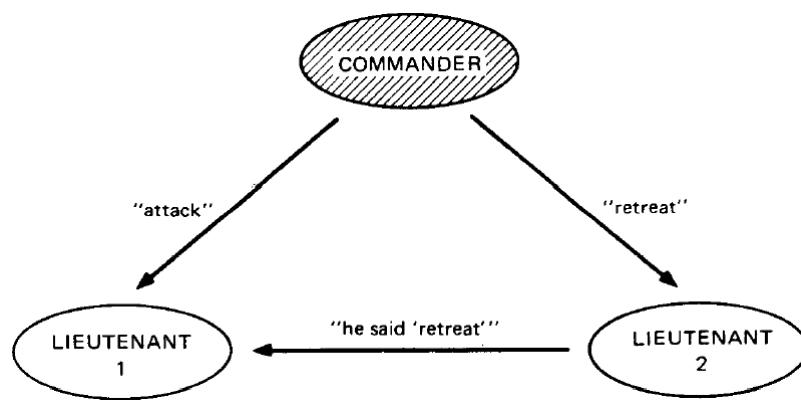
■ BÚNO: 1 generál, ostatní důstojníci

- ◆ jak generál tak důstojníci mohou být zrádci
- ◆ generál vydá rozkaz, důstojníci ho předají ostatním
- ◆ rozkaz bude vydán na základě většiny

■ Cíl protokolu:

- ◆ C1: uzly se shodnou na jedné hodnotě
 - *všichni loajální důstojníci vydají stejný rozkaz*
- ◆ C2: pokud hodnotu navrhl nehavarovaný uzel, uzly se shodnou na této hodnotě
 - *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

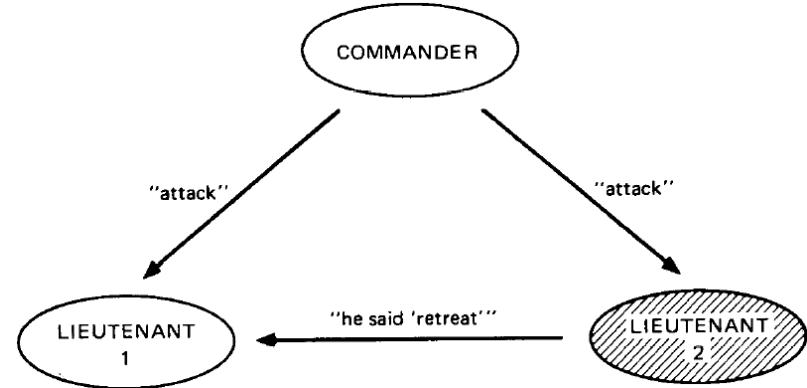
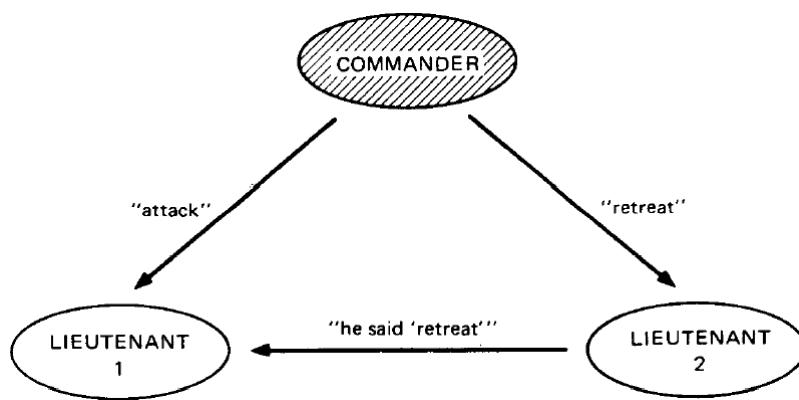
■ Zrádce důstojník

- ◆ 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 $n \leq 3m$ uzelů

Problém Byzantských generálů - 3 uzly



■ Zrádce generál

- ◆ různé rozkazy důstojníkům
- ◆ důstojník 1 dostane 2 různé rozkazy

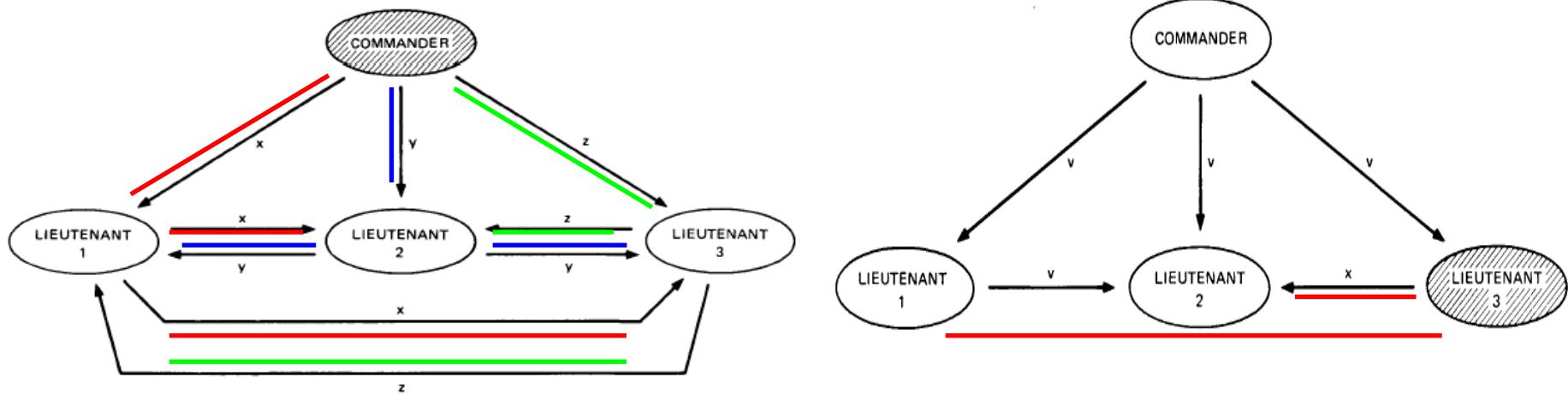
■ Zrádce důstojník

- ◆ 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 $n \leq 3m$ uzelů

Problém Byzantských generálů - 4 uzly



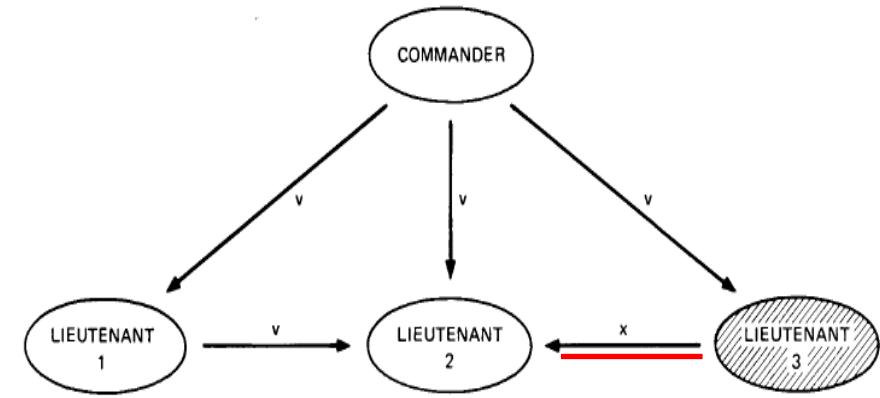
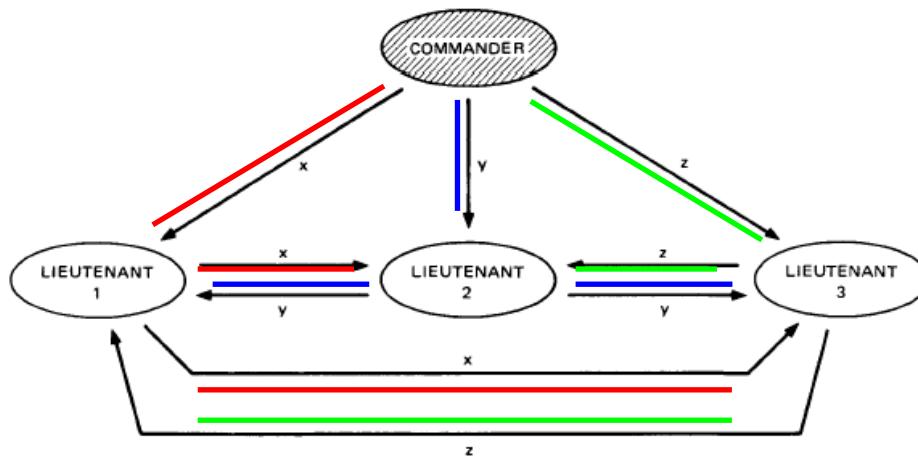
■ Zrácce generál

- ◆ alespoň 2 stejné rozkazy
 - důstojníci si vzájemně přepošlou většinový rozkaz (C2)
- ◆ 3 různé rozkazy
 - důstojníci se shodnou na tom, že generál je zrácce (C1)

■ Zrácce důstojník

- ◆ nejhorší případ: falešné předání rozkazu všem ostatním
- ◆ loajální důstojníci dostanou většinu správných rozkazů (C2)

Problém Byzantských generálů - 4 uzly



■ Řešení pro 4 uzly s 1 zrádcem existuje

Obecně: Pro m zrádců exsituje řešení pro $n \geq 3m+1$ uzelů

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

- Idea algoritmu: rekurzivní dle m (počet možných zrádců)
 - celkem $m+1$ kol
 - BGP(0)
 - generál pošle hodnotu všem důstojníkům
 - BGP(m)
 - důstojník jako generál
 - pošle vektor hodnot z BGP($m-1$) ostatním důstojníkům, kteří je ještě nemají
 - výběr většinové hodnoty
- exponenciální
počet zpráv !
- Karel řekl, že
Josef řekl, že ...

1 → 1

■ Byzantský konsensus (*Byzantine agreement*)

- ◆ 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

- ◆ každý uzel má iniciální hodnotu
- ◆ všechny loajální uzly se musí shodnout na společné hodnotě
- ◆ pokud je iniciální hodnota všech loajálních uzelů stejná, musí se shodnout na této hodnotě

■ Interaktivní konzistence

- ◆ 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

■ Byzantský konsensus (*Byzantine agreement*)

- ◆ 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

1 → 1

■ Konsensus

- ◆ 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

n → 1

■ Interaktivní konzistence

- ◆ 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

■ Byzantský konsensus (*Byzantine agreement*)

- ◆ 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

1 → 1

■ Konsensus

- ◆ 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

n → 1

■ Interaktivní konzistence

- ◆ 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

n → n

■ Byzantský konsensus (*Byzantine agreement*)

- ◆ 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

1 → 1

■ Konsensus

- ◆ 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

n → 1

■ Interaktivní konzistence

- ◆ 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

n → n

Konsensus & fault-tolerance

- Reálné prostředí
 - nespolehlivé uzly, neomezená doba zpracování a reakce, výpadky
 - nespolehlivá síť, neomezená doba doručení zprávy, ztráta, duplikace, pořadí
- **Paxos**
 - rodina protokolů pro asynchronní fault-tolerantní konsensus
 - výpadek není výjimka, ale běžný stav
 - Lamport:
 - 1989 *rejected!*
 - 1998 The Part-Time Parliament
 - 2001 Paxos Made Simple
 - výpadek F uzel tolerován při celkem $2F+1$ uzelích
 - formálně dokázaná korektnost
 - neřeší se úmyslné podvádění
 - Byzantine Paxos
 - základ řady moderních systémů:
 - Chubby (Google), Zookeeper (Yahoo), ...

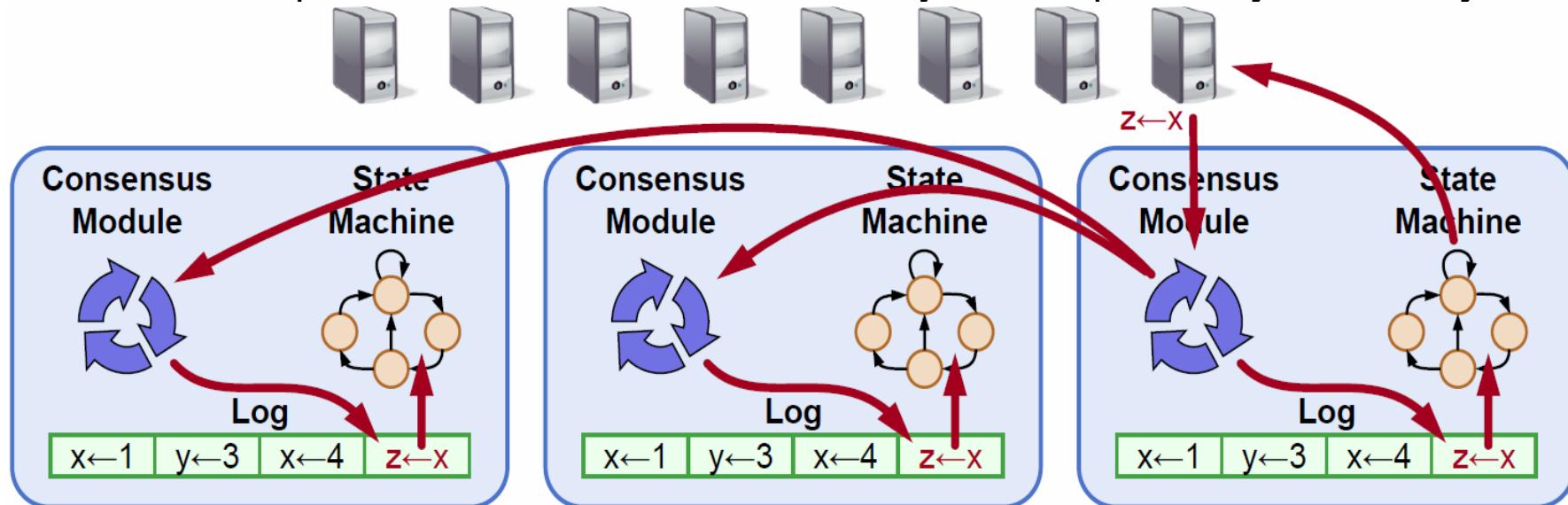


- Part-time parliament
 - poslanci (*legislators, procesy*) - schvalují zákony
 - poslíčci (*messengers, zprávy*) - doručují zprávy mezi poslanci
 - poslanci/poslíčci můžou na čas odejít (*havárie/výpadek*)
- Problém
 - poslanci nejsou ochotni zůstat na celé jednání
 - poslíček je důvěryhodný, ale nespolehlivý
 - nemění zprávy, může ale odejít na 5 minut, na 3 týdny nebo n
 - poslíček může zprávu doručit několikrát
- Způsob řešení
 - každý poslanec má zákoník a nesmazatelný inkoust
 - zaznamenává všechny návrhy



Konsensus & replikovaný konečný automat

- Konsensuální algoritmus - dosažení shody většiny uzelů
 - 5 serverů může pokračovat i když 2 servery havarují
 - větší počet havárií \Rightarrow zastavení algoritmu
- Typické využití: replikovaný konečný automat
 - **replicated state machine**
 - každý uzel zná **přechodové funkce**, udržuje **stav** (automatu) a log **příkazů**
 - jeden krok: distribuovaný konsensus o následujícím příkazu
 - každý uzel provede stejnou sekvenci příkazů \Rightarrow stejná posloupnost stavů
 - chování z pohledu klienta: komunikace s jedním spolehlivým stavovým





Paxos - role

Hlavní role

- **Proposer**
 - navrhuje nový stav, řeší konflikty
- **Acceptor**
 - přijímá nebo odmítá návrhy, sdružení v Quoru
- **Learner**
 - reprezentuje repliky, provádí změnu na základě akceptovaného stavu

Další role

- **Client** - zadává požadavek na změnu stavu
- **Leader** (Primary Proposer)
 - vybraný Proposer - pro přechod do dalšího stavu

Quorum

- libovolná většinová podmnožina Acceptorů
- zpráva od Acceptora je platná až po příjmu zpráv celého Quora

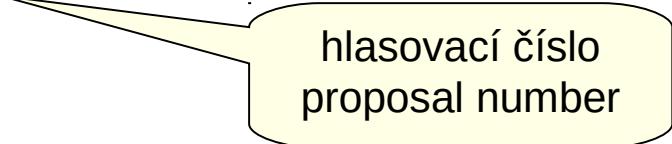
role ≠ uzel

- uzel typicky zastává několik rolí

Ballot numbers

Uspořádání

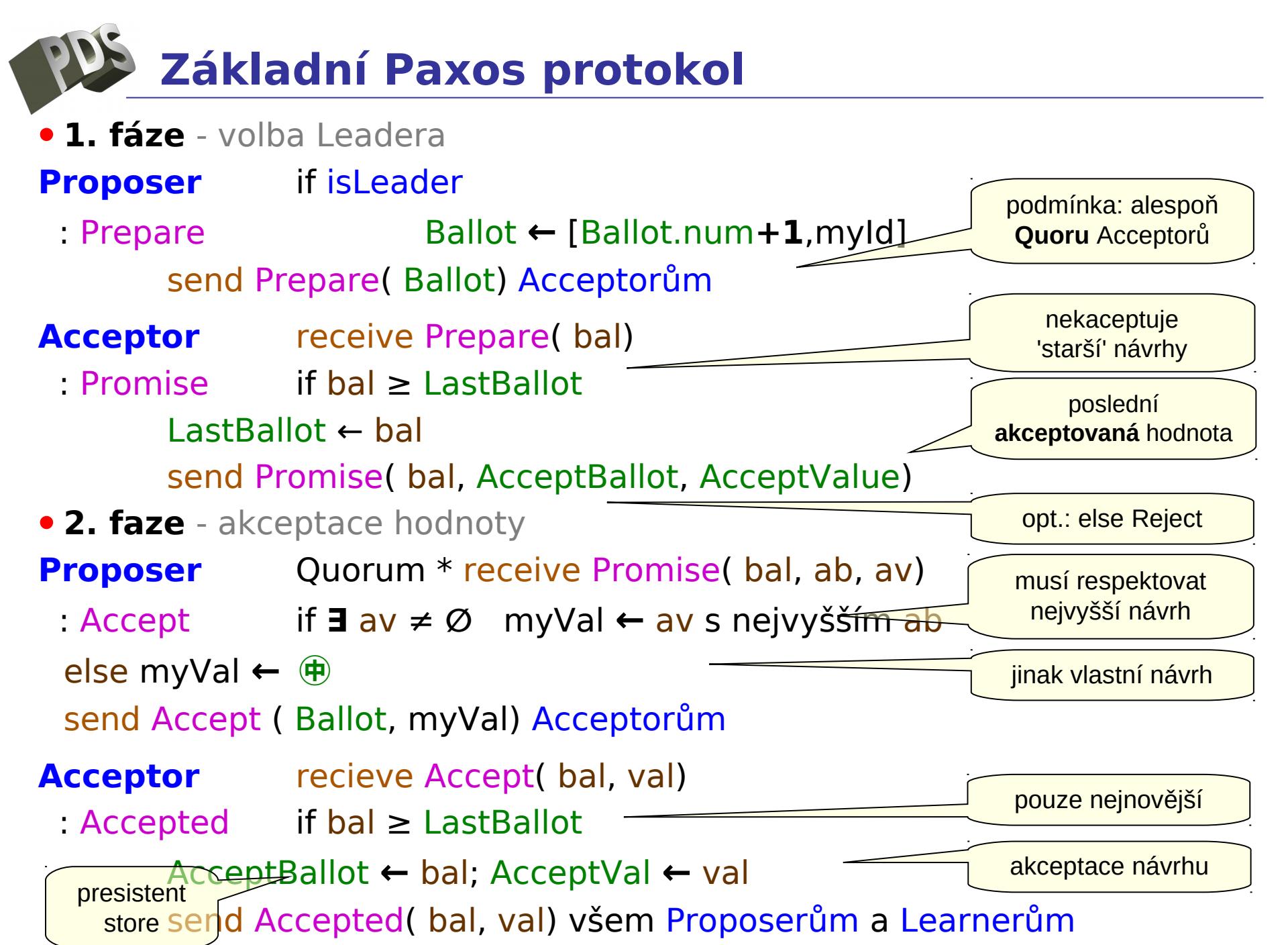
- **Ballot numbers** - sekvenční čísla [num, process id]
 - $[n_1, p_1] > [n_2, p_2]$
 - $n_1 > n_2$
 - $n_1 = n_2 \ \& \ p_1 > p_2$
 - lokálně monotónní globální uspořádání, jednoznačné
- Volba Ballot
 - poslední známý Ballot $[n, q]$
 - pak p zvolí $[n+1, p]$
- Procesy akceptují pouze zprávy s **nejvyšším** Ballot



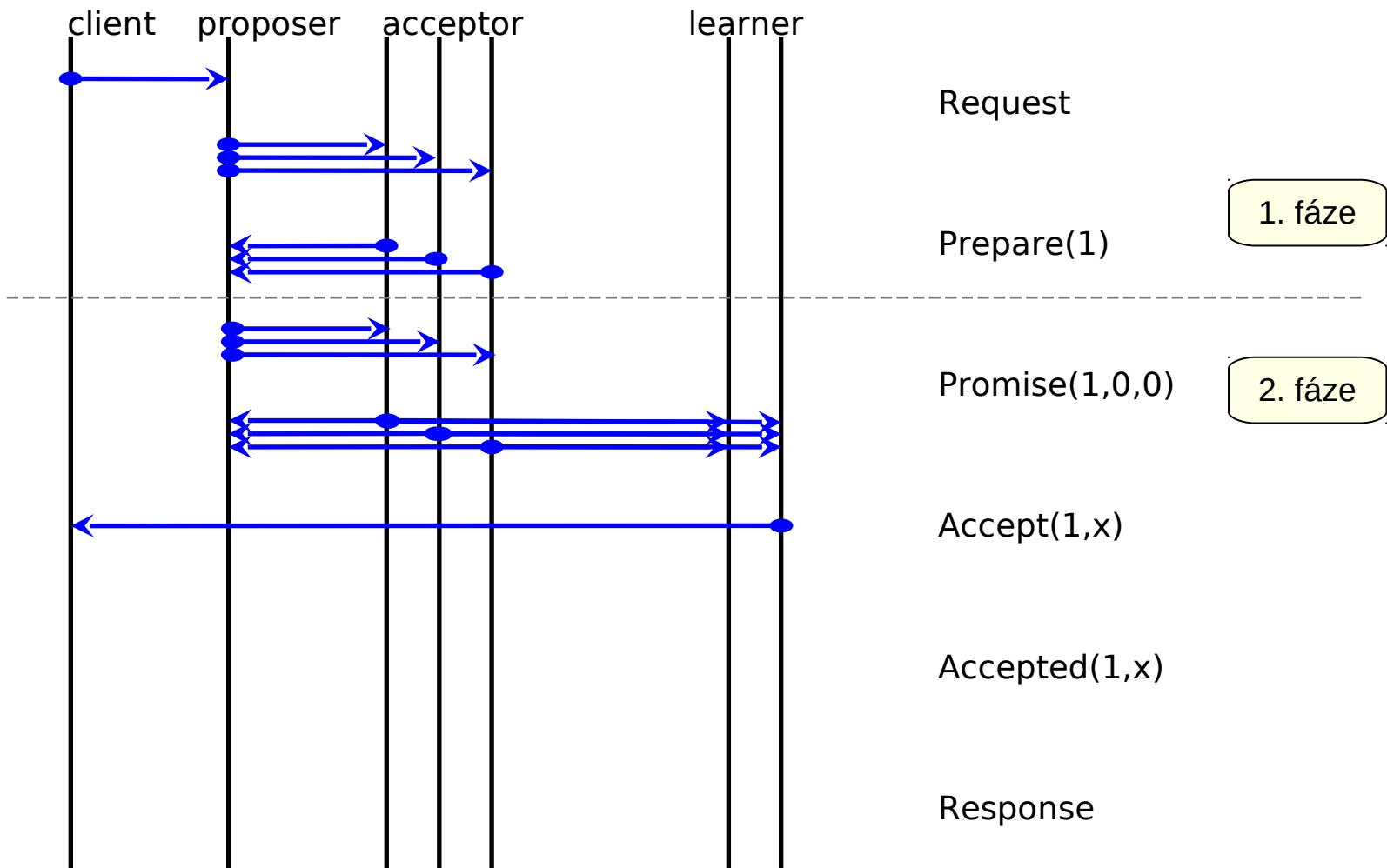
hlasovací číslo
proposal number

Proměnné (lokální pro každou roli)

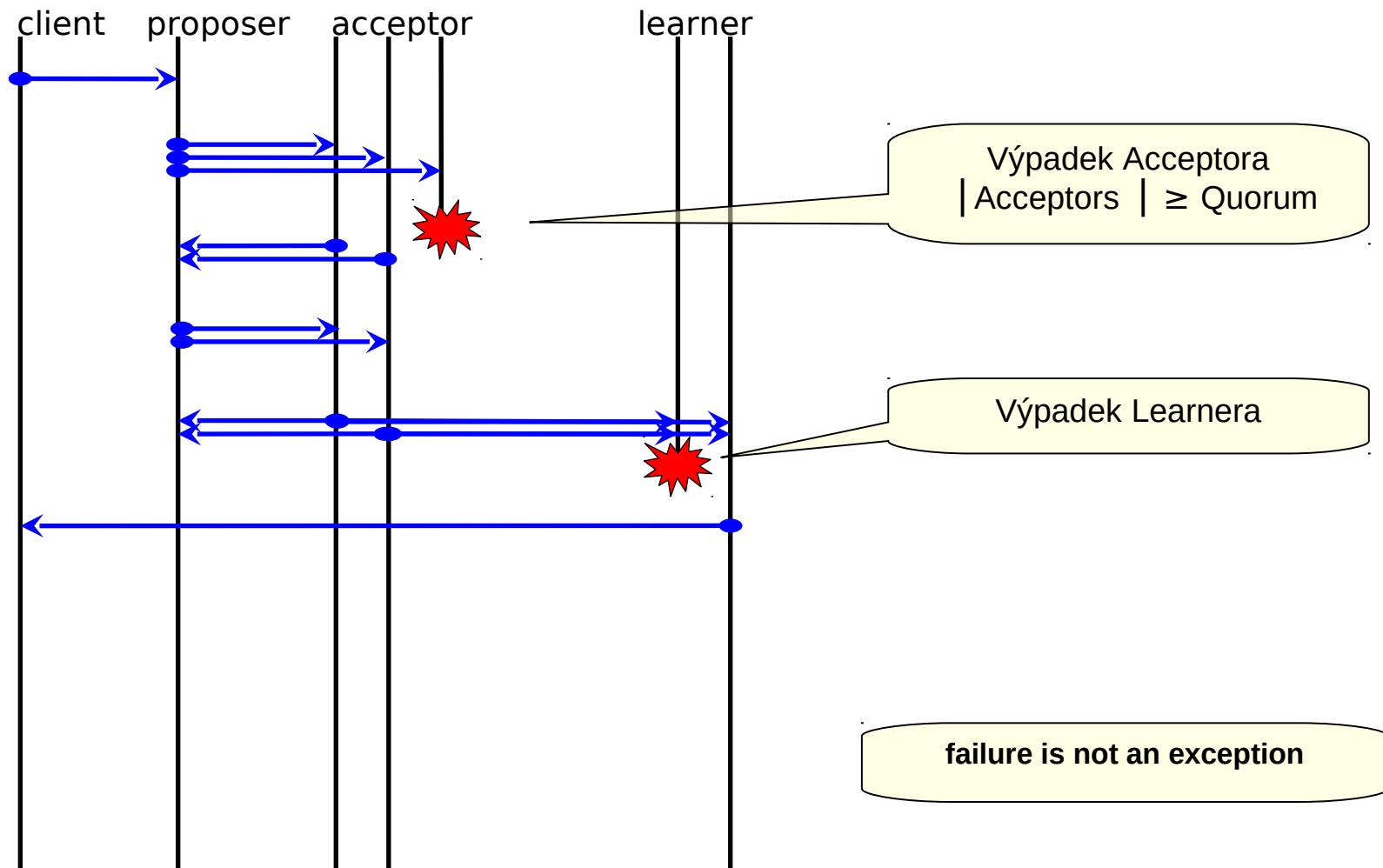
- LastBallot - poslední Ballot (návrh) iniciálně $[0,0]$
- AcceptBallot - poslední akceptovaný Ballot iniciálně $[0,0]$
- AcceptValue - poslední akceptovaná hodnota iniciálně \emptyset



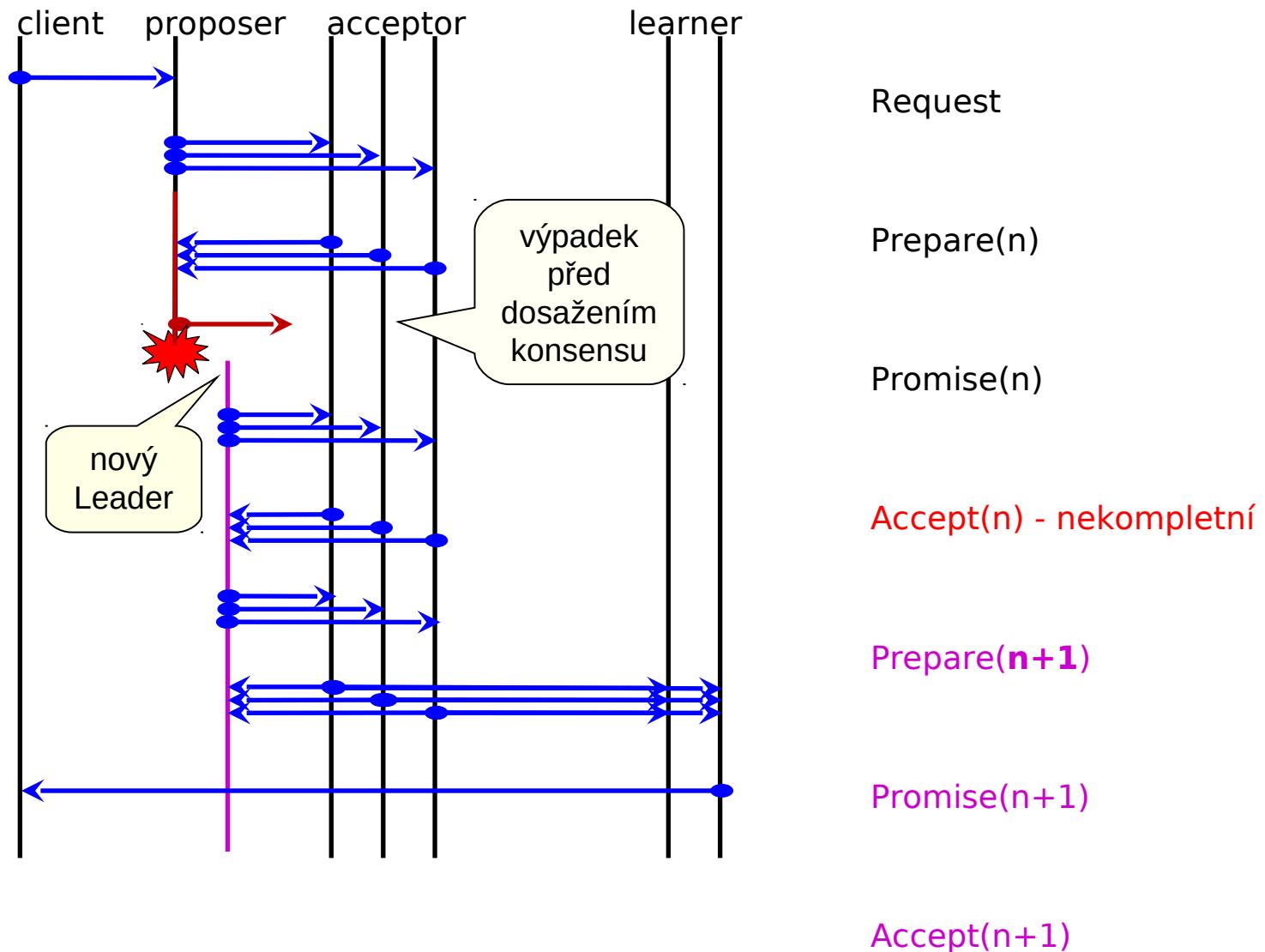
Paxos - základní komunikace



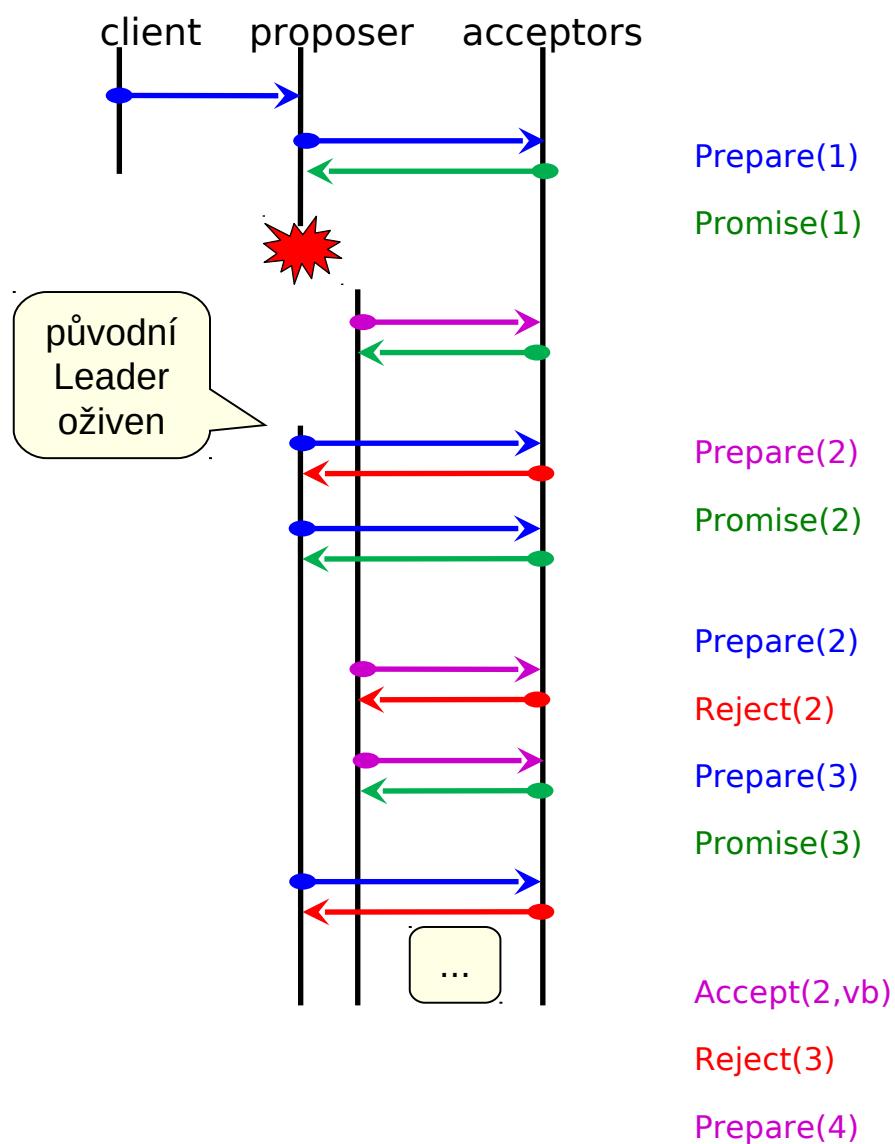
Paxos - výpadky bez režie



Paxos - výpadek Proposera



Paxos - competing Proposers



Konflikt

- více Proposerů se střídavě prohlašuje za Leadery
 - např. při oživení po havárii
 - mezitím volba nového Leadera
- konečnost není deterministicky zaručena
- praktická implementace OK
 - randomizovaný timeout
 - komunikace mezi Proposery
- není porušena korektnost!

- Fáze 1
 - neposílají se žádné hodnoty
 - sehnání dostatečného počtu Promise
 - Leader očekává většinu (Quorum) hlasů pro jeho návrh
 - zajištění konzistence s nižšími Ballot
- Fáze 2
 - Leader navrhuje hodnotu s nejvyšším Ballot z fáze 1
 - nebo může navrhнуть vlastní hodnotu
 - akceptace hodnoty ⇒ potvrzení leadera
- Stabilní úložiště
 - musí přežít výpadek uzlu
 - obsahuje informace, které si musí uzel pamatovat po znovuoživení
 - Acceptor uloží stav před vlastní odpovědí
- Počet zpráv Learnerům
 - každý Acceptor každému Learneru
 - jeden vybraný Learner / množina
 - více Learnerů ⇒ **vyšší spolehlivost / vyšší komunikace**

nesmazatelný inkoust

Paxos - invarianty, formální důkaz

$$I1(p) \triangleq \begin{aligned} & \text{[Associated variable: } outcome[p] \text{]} \\ & (outcome[p] \neq \text{BLANK}) \Rightarrow \exists B \in \mathcal{B} : (B_{qrm} \subseteq B_{vot}) \wedge (B_{dec} = outcome[p]) \end{aligned}$$

$$I2(p) \triangleq \begin{aligned} & \text{[Associated variable: } lastTried[p] \text{]} \\ & \wedge owner(lastTried[p]) = p \\ & \wedge \forall B \in \mathcal{B} : (owner(B_{bal}) = p) \Rightarrow \\ & \quad \wedge B_{bal} \leq lastTried[p] \\ & \quad \wedge (status[p] = \text{trying}) \Rightarrow (B_{bal} < lastTried[p]) \end{aligned}$$

$$I3(p) \triangleq \begin{aligned} & \text{[Associated variables: } prevBal[p], prevDec[p], nextBal[p] \text{]} \\ & \wedge prevBal[p] = MaxVote(\infty, p, \mathcal{B})_{bal} \\ & \wedge prevDec[p] = MaxVote(\infty, p, \mathcal{B})_{dec} \\ & \wedge nextBal[p] \geq prevBal[p] \end{aligned}$$

$$I4(p) \triangleq \begin{aligned} & \text{[Associated variable: } prevVotes[p] \text{]} \\ & (status[p] \neq \text{idle}) \Rightarrow \\ & \quad \forall v \in prevVotes[p] : \wedge v = MaxVote(lastTried[p], v_{pst}, \mathcal{B}) \\ & \quad \wedge nextBal[v_{pst}] \geq lastTried[p] \end{aligned}$$

$$I5(p) \triangleq \begin{aligned} & \text{[Associated variables: } quorum[p], voters[p], decree[p] \text{]} \\ & (status[p] = \text{polling}) \Rightarrow \\ & \quad \wedge quorum[p] \subseteq \{v_{pst} : v \in prevVotes[p]\} \\ & \quad \wedge \exists B \in \mathcal{B} : \wedge quorum[p] = B_{qrm} \\ & \quad \wedge decree[p] = B_{dec} \\ & \quad \wedge voters[p] \subseteq B_{vot} \\ & \quad \wedge lastTried[p] = B_{bal} \end{aligned}$$

$$I6 \triangleq \begin{aligned} & \text{[Associated variable: } \mathcal{B} \text{]} \\ & \wedge B1(\mathcal{B}) \wedge B2(\mathcal{B}) \wedge B3(\mathcal{B}) \\ & \wedge \forall B \in \mathcal{B} : B_{qrm} \text{ is a majority set} \end{aligned}$$

$$I7 \triangleq \begin{aligned} & \text{[Associated variable: } \mathcal{M} \text{]} \\ & \wedge \forall NextBallot(b) \in \mathcal{M} : (b \leq lastTried[owner(b)]) \\ & \wedge \forall LastVote(b, v) \in \mathcal{M} : \wedge v = MaxVote(b, v_{pst}, \mathcal{B}) \\ & \quad \wedge nextBal[v_{pst}] \geq b \\ & \wedge \forall BeginBallot(b, d) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \wedge (B_{dec} = d) \\ & \wedge \forall Voted(b, p) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \wedge (p \in B_{vot}) \\ & \wedge \forall Success(d) \in \mathcal{M} : \exists p : outcome[p] = d \neq \text{BLANK} \end{aligned}$$



Použití v distribuovaném plánovači

- klient spouští job v clusteru - zpráva scheduleru
každá replika scheduleru zastává všechny role - Proposer, Acceptor, Learner
právě jedna replika je Leader
- běžný bezchybový průběh:
 1. Leader pošle ostatním replikám **Prepare** s novým číslem jobu
 2. ostatní repliky vrátí **Promise** (pokud neviděli job s větším číslem)
 3. Leader přidělí jobu prostředky a pošle **Accept**
 4. ostatní odpoví **Accepted** a zaznamenají stav, Leader spustí job
- havárie - více replik se snaží být Leader, konkurentně zasílají vlastní návrhy
nakonec zvítězí návrh s nejvyšším číslem, jeho navrhovatel se stane Leaderem



Paxos - protokoly

- 2008 Google: Paxos Made Live
 - velký rozdíl mezi abstraktním popisem algoritmu a reálnou implementací
- 2014 Meling, Jehl: Paxos Explained from Scratch
 - podrobně motivace k jednotlivým částem protokolu, zdůvodnění
- 2015 Van Renesse, Altinbuken: Paxos Made Moderately Complex
 - formální popis a verifikace Multi-Paxosu

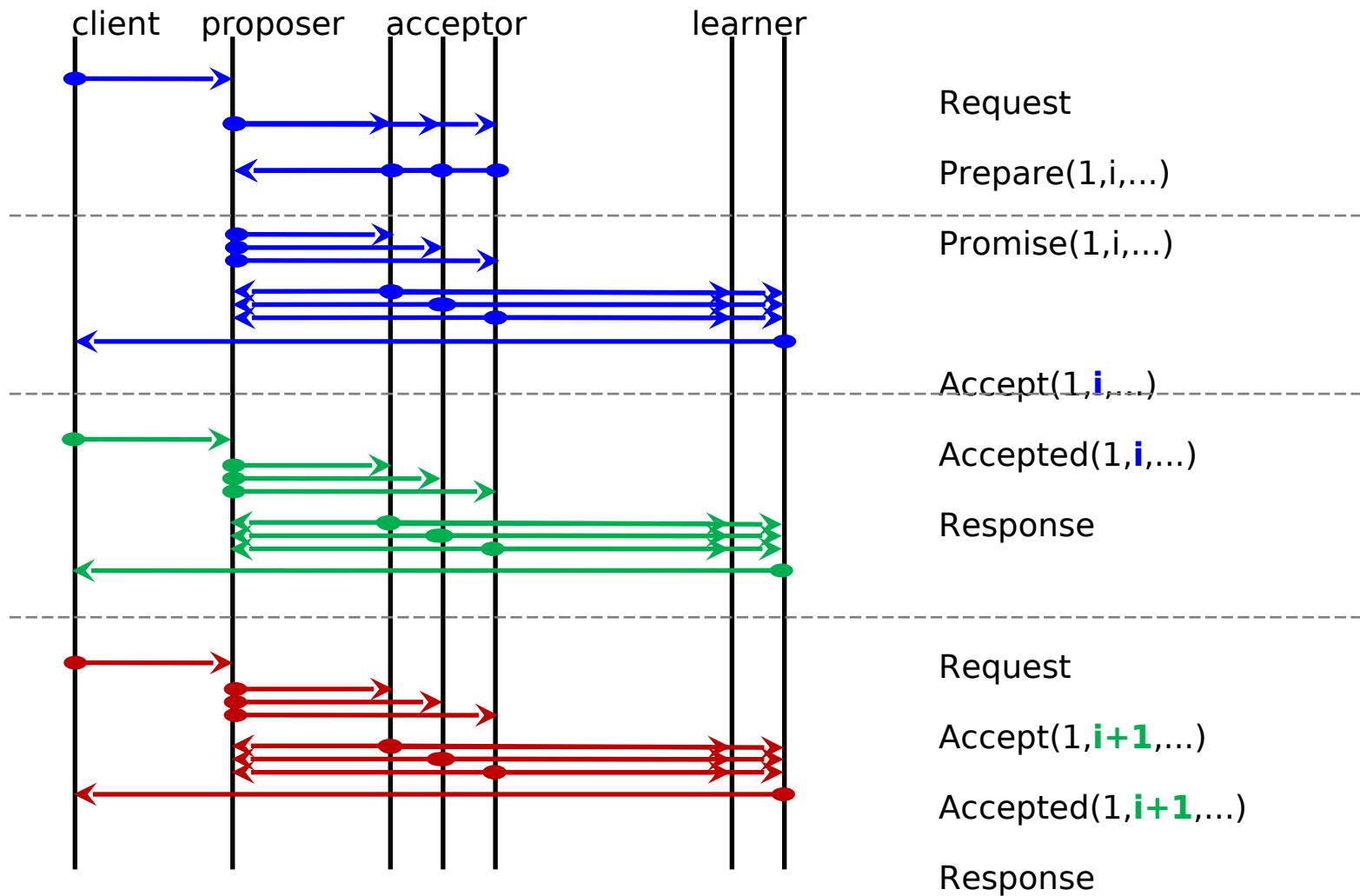
Paxos Family

- **Basic** Paxos - základní verze protokolu, konsensus pouze jedné hodnoty
 - prakticky málo použitelný \Rightarrow nepoužívaný
- **Multi**-Paxos - kontinuální opakování, základ typické implementace
- **Cheap** Paxos - rozšíření pro 'levnou' (nahraditelnou) toleranci havarovaných uzelů
- **Fast** Paxos - optimalizace doručování zpráv
- **Byzantine** Paxos - rozšíření pro záškodníky *byzantine failures*
- **Generalized** Paxos - optimalizace pro konkurentní komutativní operace
- další protokoly, optimalizace, rozšíření
 - Disk P., Vertical P., Stoppable P., Egalitarian P., Leaderless P., Mencius, ...

- Fáze 1
 - zajištění konzistence s nižšími Ballot
- Optimalizace
 - fáze 1 nutná jen při změně Leadera
 - implementační názvosloví: "view change", "recovery mode"
- Multi-Paxos
 - kontinuální pokračování Basic Paxos
 - nejčastější základ praktické implementace
 - stabilní Leader \Rightarrow pouze fáze 2
 - technicky protokol doplněn o číslo 'instance'
- Distributed State Machine
 - deterministické změny stavů dle akceptovaných hodnot

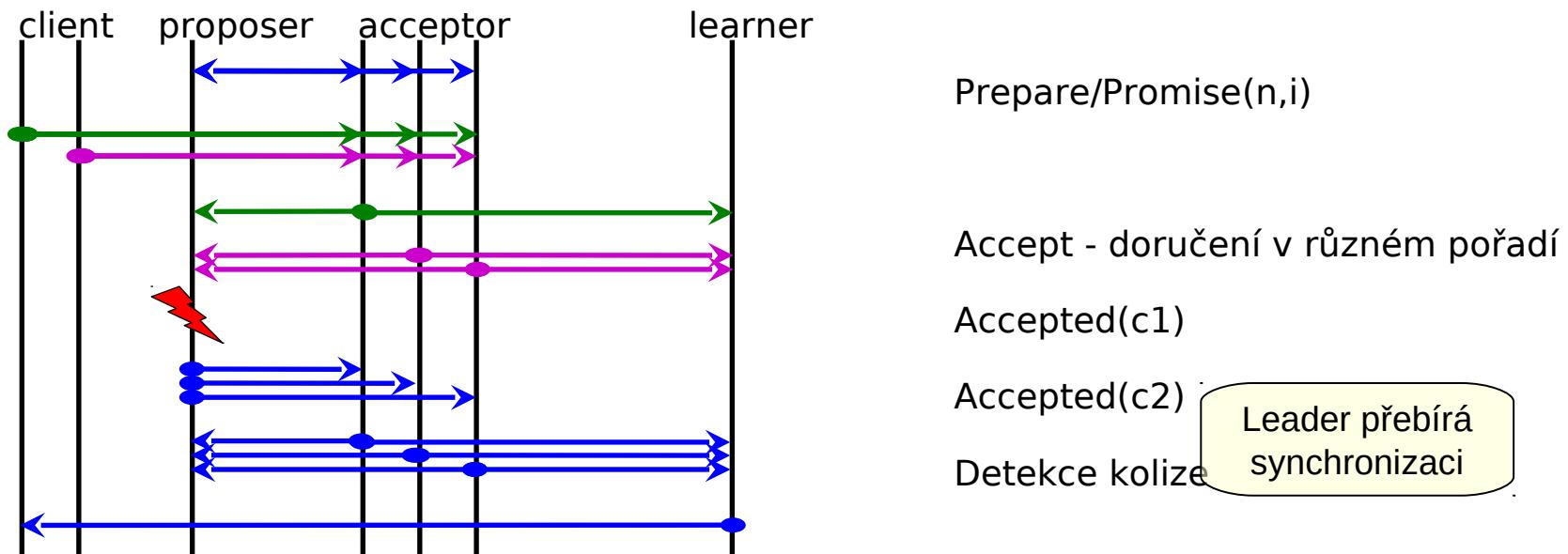
Accept(n,**i**,v1); Accepted(n,**i**,v1)
Accept(n,**i+1**,v2); Accepted(n,**i+1**,v2)

Multi-paxos - komunikace

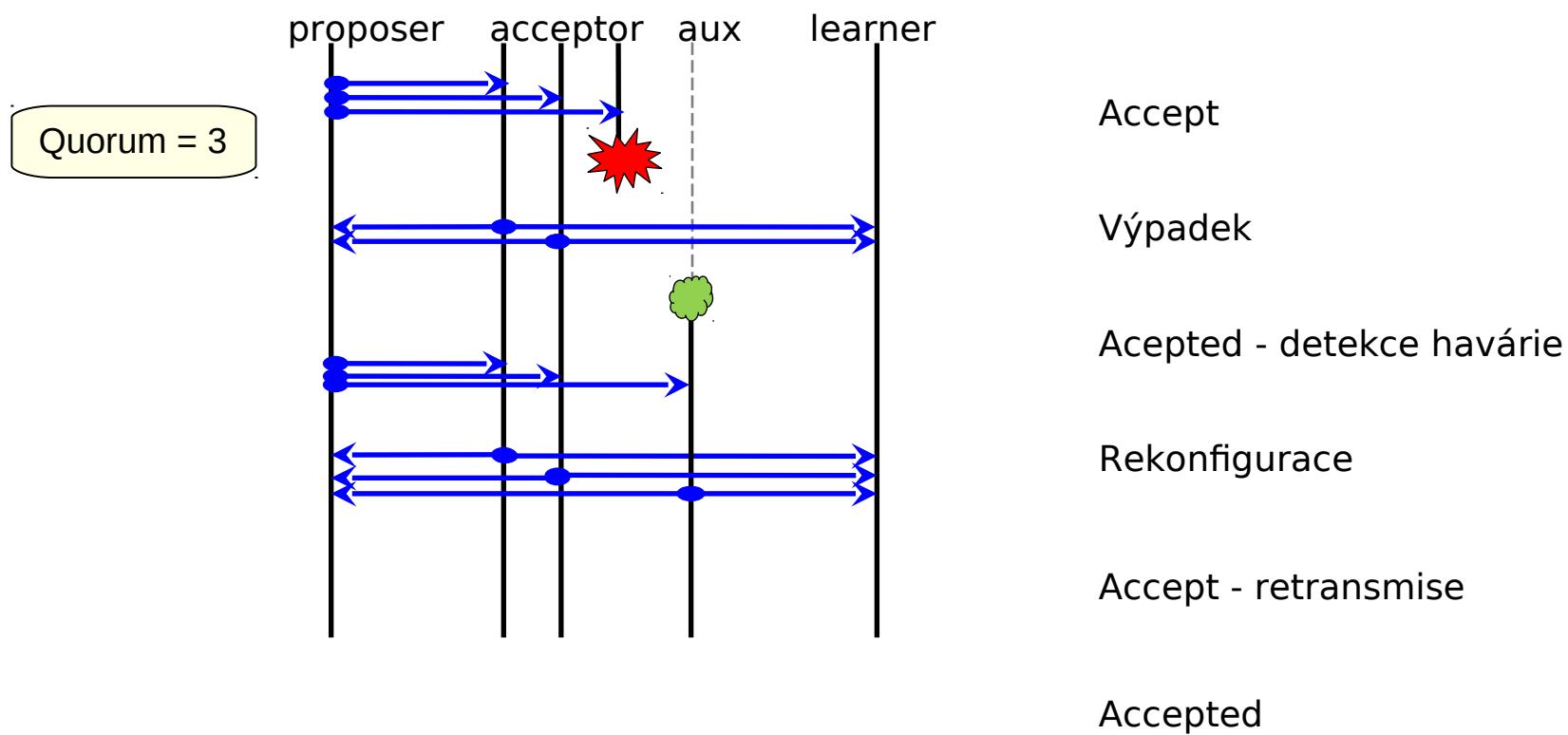


Fast Paxos

- Optimalizace - redukce prodlev při doručování zpráv
 - Basic Paxos - 3 zprávy do přijetí hodnoty
 - Fast Paxos - 2 zprávy - za cenu zaslání požadavku Clientem více uzelům
- Protokol
 - Client zašle Accept přímo Acceptorům
 - Accepted Leaderu a Learnerům
- Kolize - více konkurenčních Clientů
 - Leader určí pořadí a pošle serializovaně nové Accepty

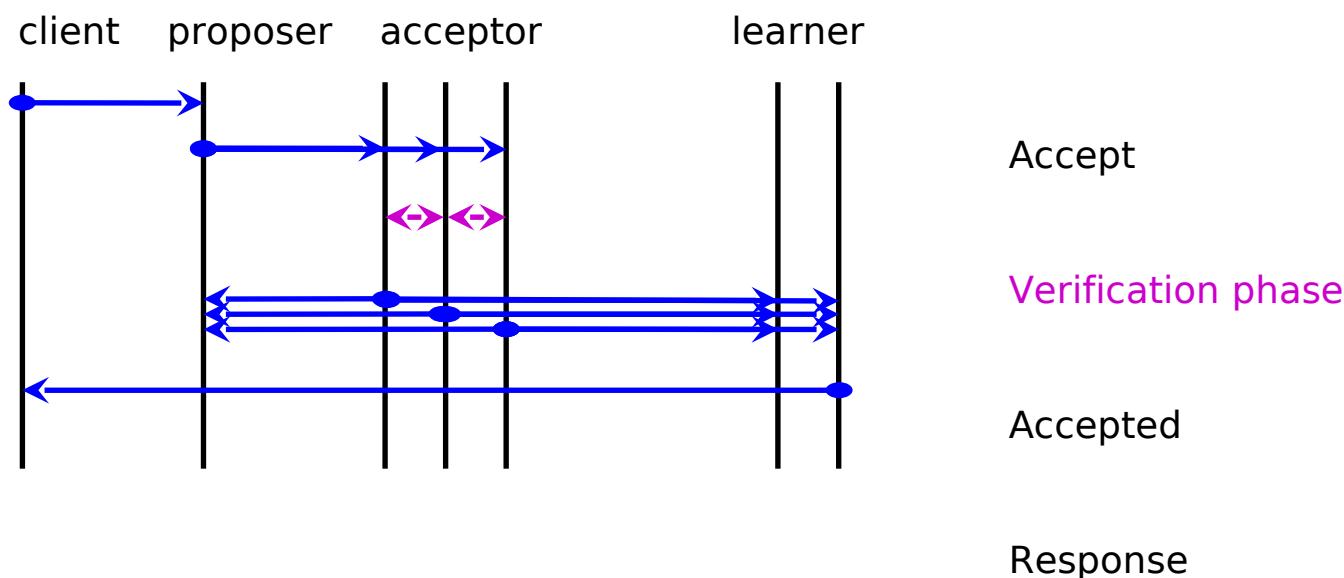


- Rozšíření - tolerance F havarujících uzlů při $F+1$ uzlech celkově
 - potřeba dalších F uzlů připravených v záloze
 - dynamická rekonfigurace po každé havárii
 - redukce uzlů potřebných k fault-tolerantnosti za cenu zvýšené režie při výpadku



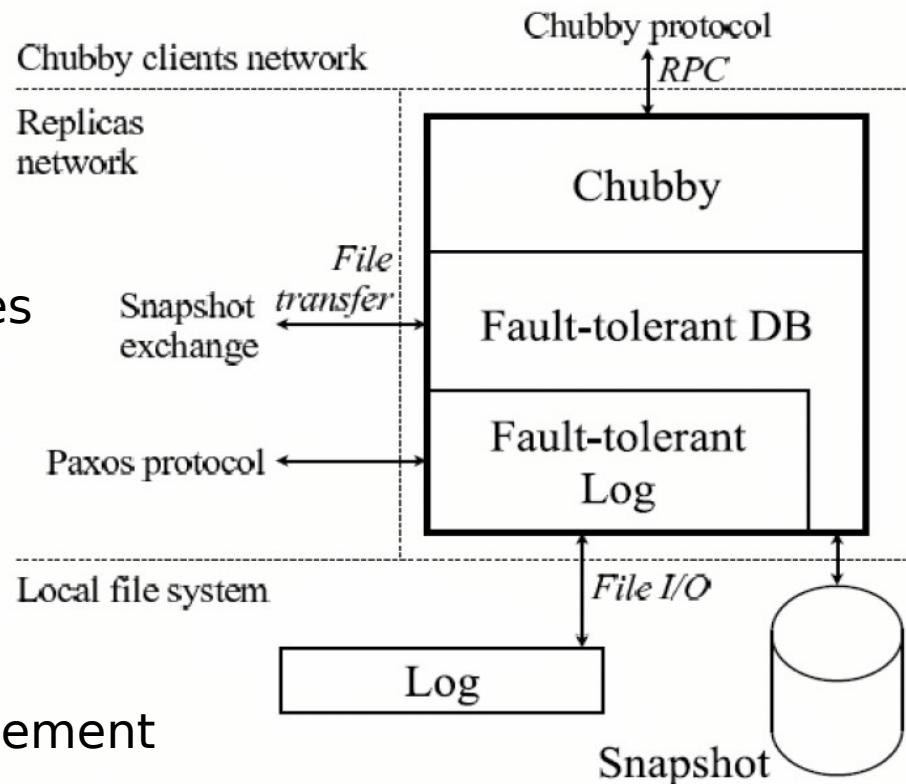
Byzantine Paxos

- Ochrana před (ne)úmyslně nekorektním chováním
 - uzel může měnit zprávy, nerespektovat protokol apod.
 - běžně se nepoužívá
 - nepředpokládá se zákeřné chování při havárii
- Přidáno kolo vzájemně ověřovacích zpráv - Verify
 - lze optimalizovat - broadcast zároveň s Accepted
- V případě nekonzistence si korektní Acceptoréi vzájemně přeposílají hodnoty
- Learner čeká, dokud nepřijme alespoň $F+1$ stejných rozhodnutí



Paxos - aplikace

- Google Chubby
 - fault-tolerant distributed coarse-grained locking
 - typicky 5 replik, 1 master
 - při havárii automatická volba nového leadera
 - hierarchický prostor jmen
 - filesystem API
 - exclusive/shared lock (rw/ro)
 - dobrovolná sémantika
- Google Borg / Omega / Kubernetes
 - cluster management systems
 - orchestrace kontejnerů
 - 10K uzlů
 - master - 5 replik
- FT cloud/cluster/container management





Paxos a RAFT

Paxos - nedostatky

- téměř 20 let de-facto standard pro distribuovaný konsensus
ale ...
- těžko pochopitelný
- náročná implementace
- nekompletní popis - reálně implementovatelná verze jen naznačena
 - shoda jen na jedné hodnotě, konečnost, membership management, efektivita, ...
- reálná implementace neverifikovaná
 - i přes formální důkaz základního protokolu

The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos :-) — NSDI reviewer

RAFT - Replicated And Fault-Tolerant protocol

- 2014 **Ongaro**, Ousterhout: In Search of an Understandable Consensus Algorithm
 - 3x rejected
- hlavní motivace: **srozumitelnost**, (relativní) snadnost implementace
- implementace: gorraft + desítky dalších
- použití: InfluxDB (HP & HAvail time series DB), etcd (distributed key-value store), ...
- Ousterhout: <https://www.youtube.com/watch?v=YbZ3zDzDnrw>

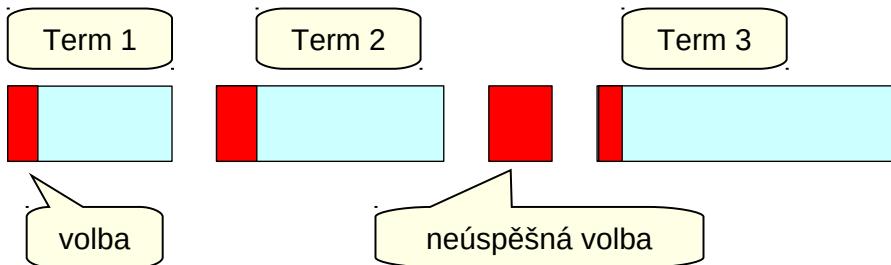
RAFT - role a komponenty

- Role
 - Leader - navrhuje hodnoty
 - Follower - akceptuje hodnoty, pouze pasivní, hlídá timeout
 - Candidate - dočasně při volbě Leadera

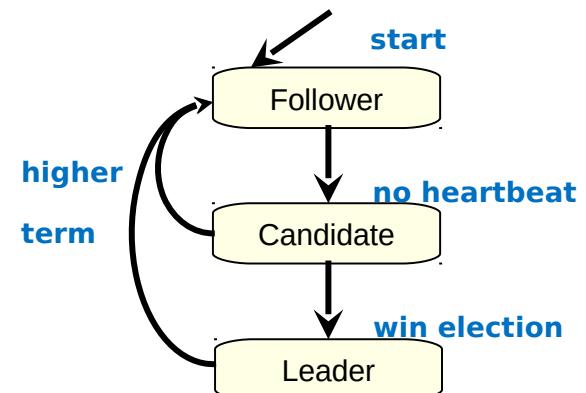
Hlavní komponenty:

- Volba Leadera
 - heartbeat, detekce havárie
 - volba nového Leadera
- Replikace logu
 - normální operace
 - Leader přijímá požadavky od klientů, přidá je do vlastního logu
 - Leader replikuje log Followerům
- Udržování konzistence
 - kontrola konzistence záznamů
 - pouze uzly s aktuálním logem se mohou stát Leaderem

RAFT - Terms



- Term
 - základ synchronizace
 - s každou zprávou, detekce zastaralých údajů
 - Term má jednoznačného Leadera
 - každý uzel udržuje lokální CurrentTerm
 - IncomingTerm > CurrentTerm
 - aktualizace, Leader → Follower
 - IncomingTerm < CurrentTerm
 - reject
- Heartbeat
 - Leader periodicky zasílá zprávy Followerům
 - timeout - volba nového Leadera
 - cca 100-500 ms

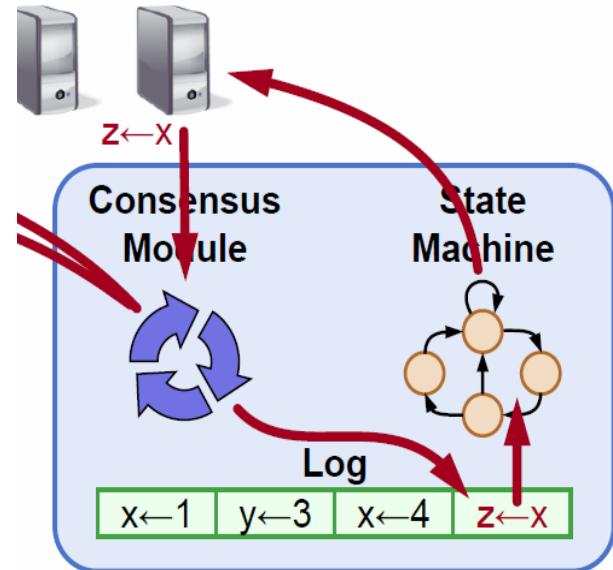


RAFT - normální průběh

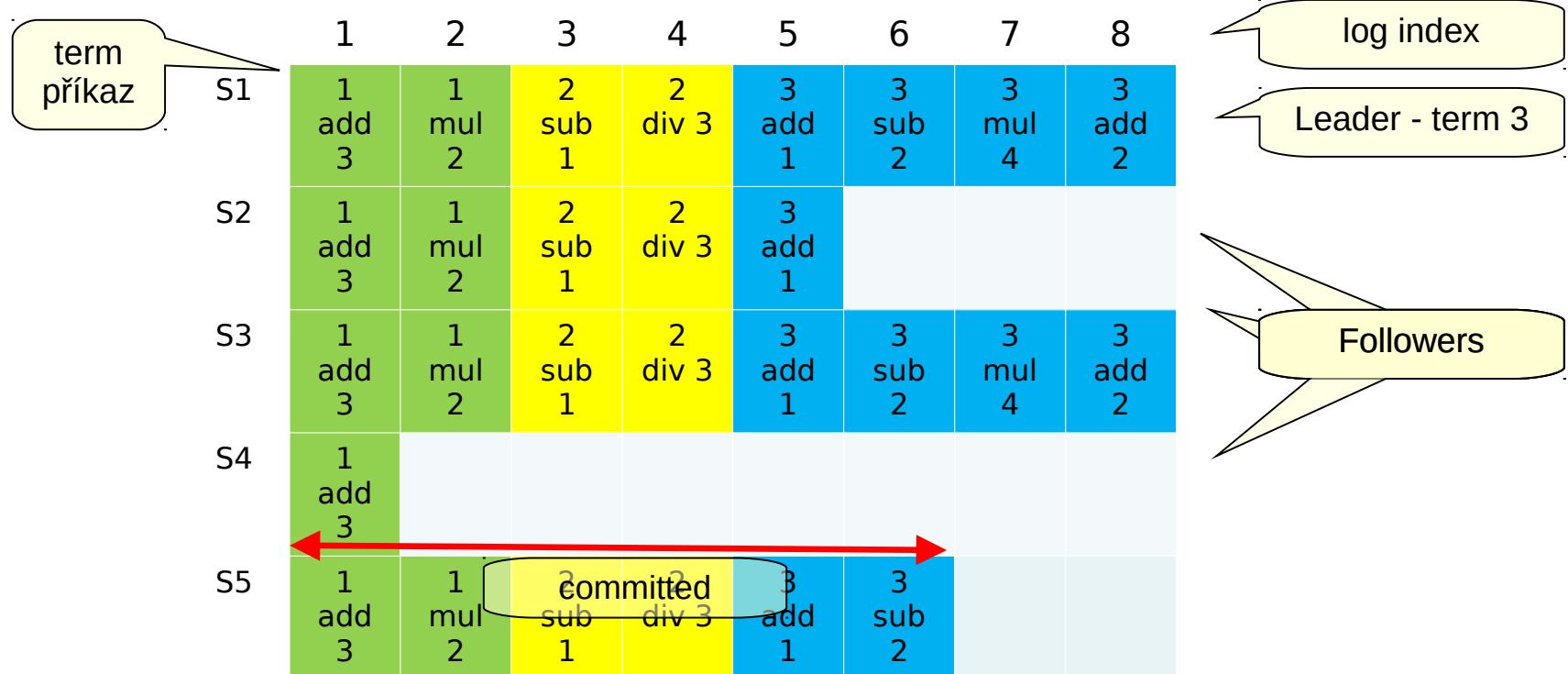
Normální průběh operací

- Klient pošle žádost Leaderu
 - Leader přidá záznam do svého logu
 - Leader pošle **AppendEntries** všem Followerům
- potvrzení příjmu **většinou** Followerů (quorum)
 - záznam (příkaz) **commitován**
 - Leader aplikuje záznam (provede příkaz) ve svém lokálním stavovém automatu
 - výsledek vrátí Klientovi
 - Leader oznamuje Followerům commitované záznamy v následných **AppendEntries**
 - po příjmu commitu Follower provede příkaz ve svém stavovém automatu
- havárie nebo zpoždění Followera
 - Leader opakuje **AppendEntries** s více záznamy - později
- optimální výkonnost při běžném provozu
 - jedno RPC (request/reply) pro libovolnou většinu uzlů

synchroznizace logu - za chvíli



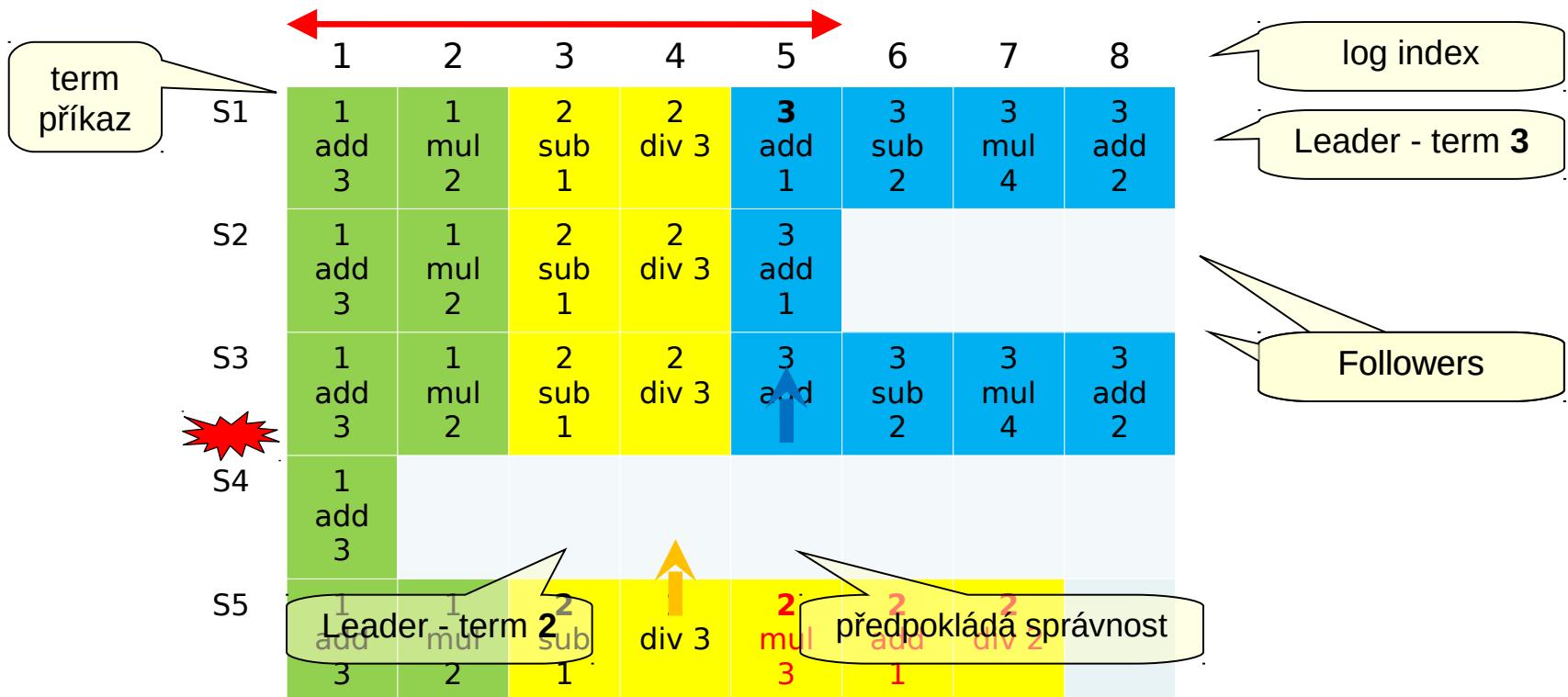
RAFT - struktura Logu



Log

- indexovaný
- perzistentní - musí se zotavit po havárii
- commit - záznam replikovaný na většině uzlů
- růst logu nerovnoměrný
- ne každý uzel se musí dozvědět o novém termu

RAFT - konzistence Logu



Možné nekonzistence

- havárie - přerušení komunikace
- Leader - záznamy bez potvrzení
- jinde možný nový term

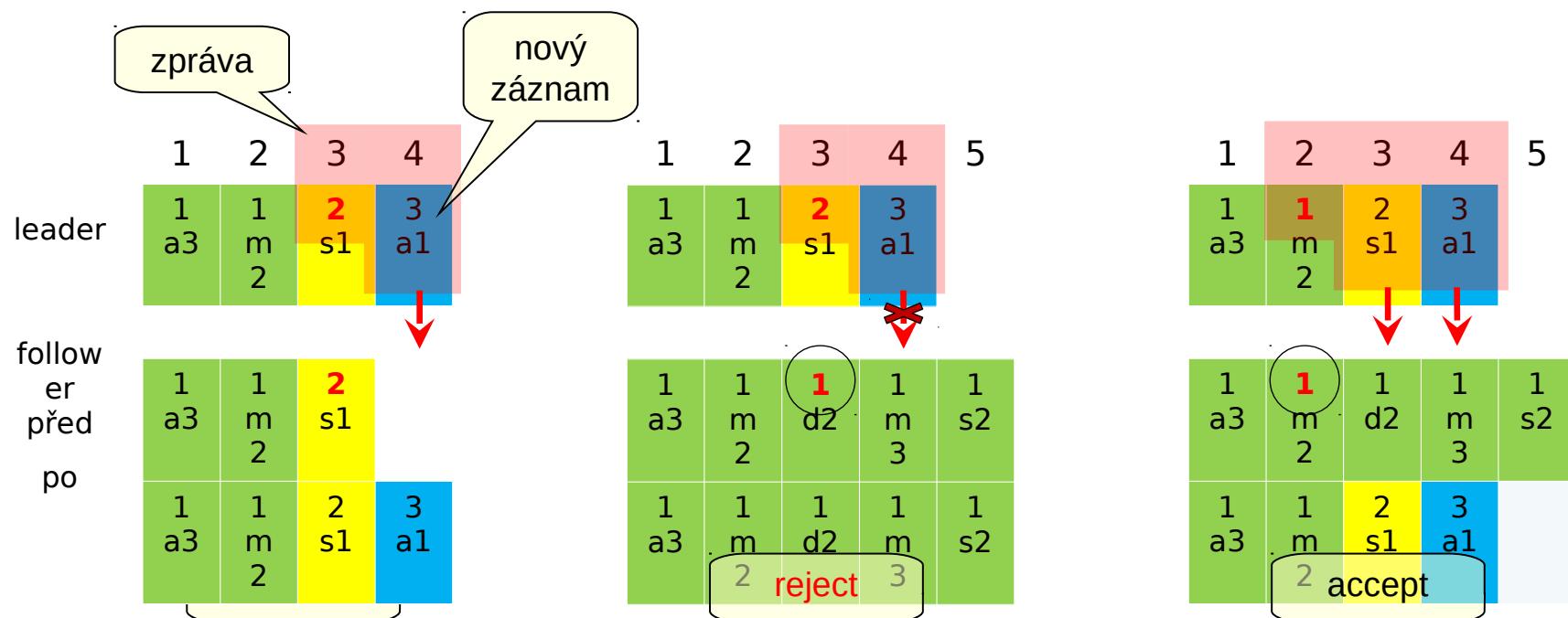
Konzistence logů

- záznam se stejným indexem a termem
 - vždy **shodná** hodnota
 - všechny **předcházející** hodnoty shodné
- commitovaný záznam
 - všechny předcházející hodnoty shodné
 - všichni budoucí Leaderi ho musí obsahovat

RAFT - AppendEntries

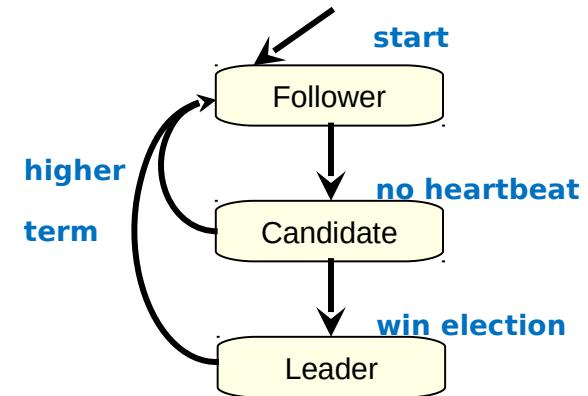
AppendEntries - kontrola konzistence

- <index, term> **předcházejícího** záznamu
- Follower: <index, term> musí souhlasit, jinak reject
 - → Leader: **nová** zpráva s **nížším** indexem
 - obsahuje všechny novější záznamy



RAFT - volba Leadera

- Detekce havárie
 - Leader: heartbeat
 - Follower: timeout
- Volba Leadera:
 - Follower \rightsquigarrow Candidate
 - vote for self, **RequestVote**
 - vyšší term \rightsquigarrow Follower
 - většinový počet hlasů \rightsquigarrow Leader
 - timeout \rightsquigarrow randomizovaná pauza, nová volba
 - $[T, 2T] \dots$ cca 150-300 ms
 - chová se dobře při $T \gg$ doba přenosu zprávy
 - randomizovaný přístup v praxi jednodušší
- **RequestVote** obsahuje poslední zaznamenaný $\langle \text{index}, \text{term} \rangle$ Candidate
 - uspořádání aktuálnosti logu dle $\langle \text{LastTerm}, \text{LastIndex} \rangle$
 - uzel odmítne dát hlas pokud má aktuálnější log
- Demo: <http://thesecretlivesofdata.com/raft/>

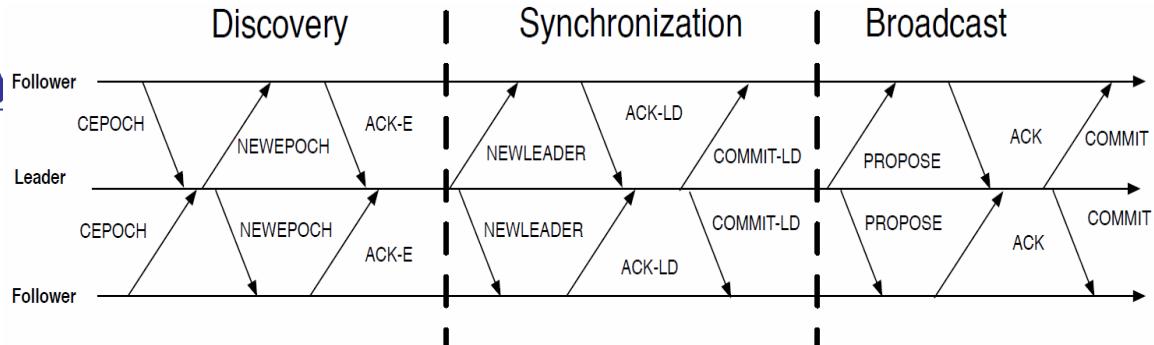




RAFT Protocol Summary

Followers	
<ul style="list-style-type: none"> • Respond to RPCs from candidates and leaders. • Convert to candidate if election timeout elapses without either: <ul style="list-style-type: none"> • Receiving valid AppendEntries RPC, or • Granting vote to candidate 	
Candidates	
<ul style="list-style-type: none"> • Increment currentTerm, vote for self • Reset election timeout • Send RequestVote RPCs to all other servers, wait for either: <ul style="list-style-type: none"> • Votes received from majority of servers: become leader • AppendEntries RPC received from new leader: step down • Election timeout elapses without election resolution: increment term, start new 	
Leaders	
<ul style="list-style-type: none"> • Initialize nextIndex for each to last log index + 1 • Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts • Accept commands from clients, append new entries to local log • Whenever last log index \geq nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful • If AppendEntries fails because of log inconsistency, decrement nextIndex and retry • Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers • Step down if currentTerm changes 	
RPCs:	
currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidatId that received vote in current term (or null if none)
Log Entry	
term	term when entry was received by leader
index	position of entry in the log

RequestVote RPC	
Invoked by candidates to gather votes.	
Arguments:	
candidatId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry
Results:	
term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote
Implementation:	
1. If term < currentTerm, return false 2. If term == currentTerm, check votedFor: - If null, grant vote and return true - If candidateId != votedFor, return false - If candidateId == votedFor, check log: - If log >= lastLogIndex and log term == lastLogTerm, grant vote and return true - Otherwise, return false	
AppendEntries RPC	
(step down if leader or candidate) Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat. If term == currentTerm, votedFor is null or candidatId, and candidate's log is at least as complete as local log, grant vote and reset election timeout	
Arguments:	
term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed
Results:	
term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm
Implementation:	
1. Return if term < currentTerm 2. If term > currentTerm, currentTerm \leftarrow term 3. If term == currentTerm and votedFor == null, set votedFor = candidatId 4. If term == currentTerm and votedFor == candidatId, check log: - If log >= prevLogIndex and log term == prevLogTerm, set commitIndex = log + 1 and return true - Otherwise, return false 5. If term == currentTerm and votedFor != candidatId, return false	



- Yahoo Zookeeper 2011
 - high-performance coordination service for distributed applications
 - synchronization, **consensus**, group membership, leader election, queues, event notifications, ...
 - HBase, Storm, Kafka
 - **ZAB** - Zookeeper Atomic Broadcast Protocol
 - na principech Paxosu, detailly protokolu odlišné
 - silnější předpoklady \Rightarrow jednodušší protokol
 - protokol integrovaný ve frameworku
 - hlavní rozdíl od Paxosu - účel, design protokolu:
 - Paxos: replikovaný stavový automat
 - Zab: **primary-backup** systems
 - delta state updates generated by a primary replica for its followers
 - replicas agree on the order
 - unlike client requests, updates must be applied in the **exact** original **order**
 - if a primary fails, a new primary **cannot reorder** uncommitted state updates
 - formální verifikace
 - Viewstamped Replication - MIT 1988, komplikovaný

- Server Replication
 - motivace / základní aplikace Paxos protokolu
 - Cloud Computing, grid, ...
 - Google Borg / Kubernetes, IBM SAN Volume Controller, MS Autopilot CMS, ...
 - deterministický stavový automat
 - uzly začínají ve stejném stavu, dostávají stejný vstup ve stejném pořadí
 - ↪ stejný stav a výstup
- Log Replication
 - log - propagace změn dat na uzly
- Synchronization Service
 - tradičně: kontrola konkurence pomocí zámků
 - konsensus protokol jako služba
 - Google Chubby - based on Multi-Paxos
 - originally a distributed lock service for coarse-grained synchronization
 - wider use cases: name service and repository of configuration data
 - ZooKeeper
 - simple code for exclusive/fair/shared locks

- Barrier Orchestration
 - large-scale graph processing systems based on Bulk Synchronous Parallel model
 - Google Pregel, Apache Giraph, Apache Hama
 - coordination between computing processes
 - double barrier - beginning and the end of each computation iteration
 - barrier thresholds may be reconfigured during each iteration
- Configuration Management
 - filesystem or key-value API - storing arbitrary data
 - durable and consistent storage for small data items
 - configuration metadata - connection details, feature flags, ...
 - can be watched for changes \Rightarrow reconfigure when parameters are modified
 - leader election, group membership, service discovery, metadata management

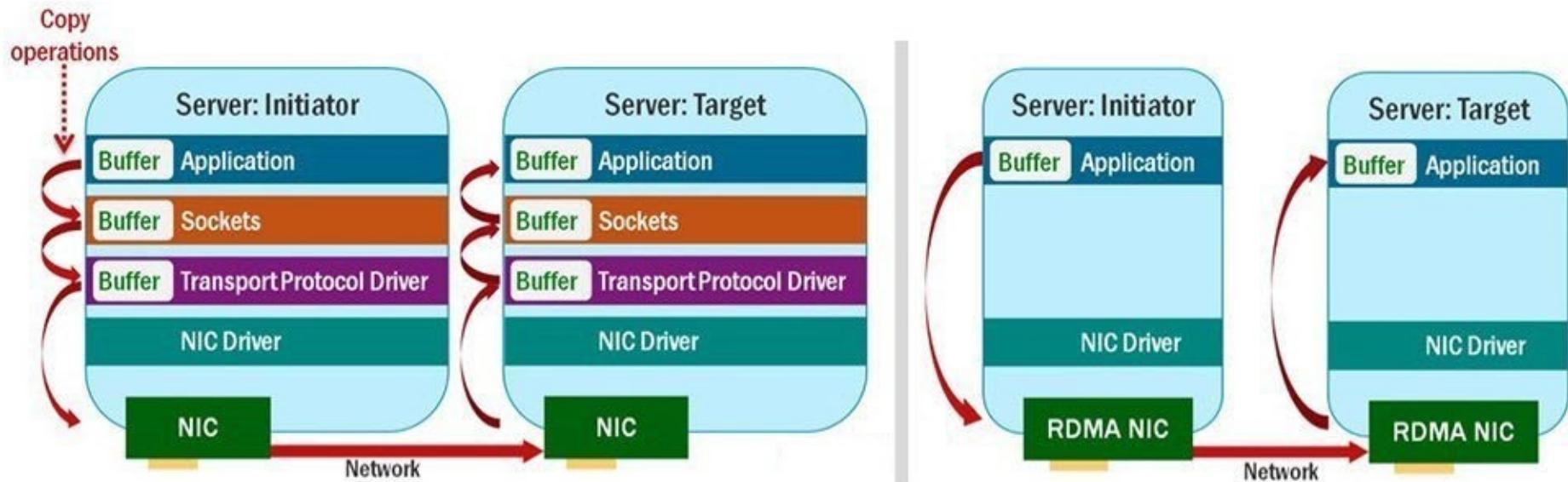


Distribuovaná sdílená paměť

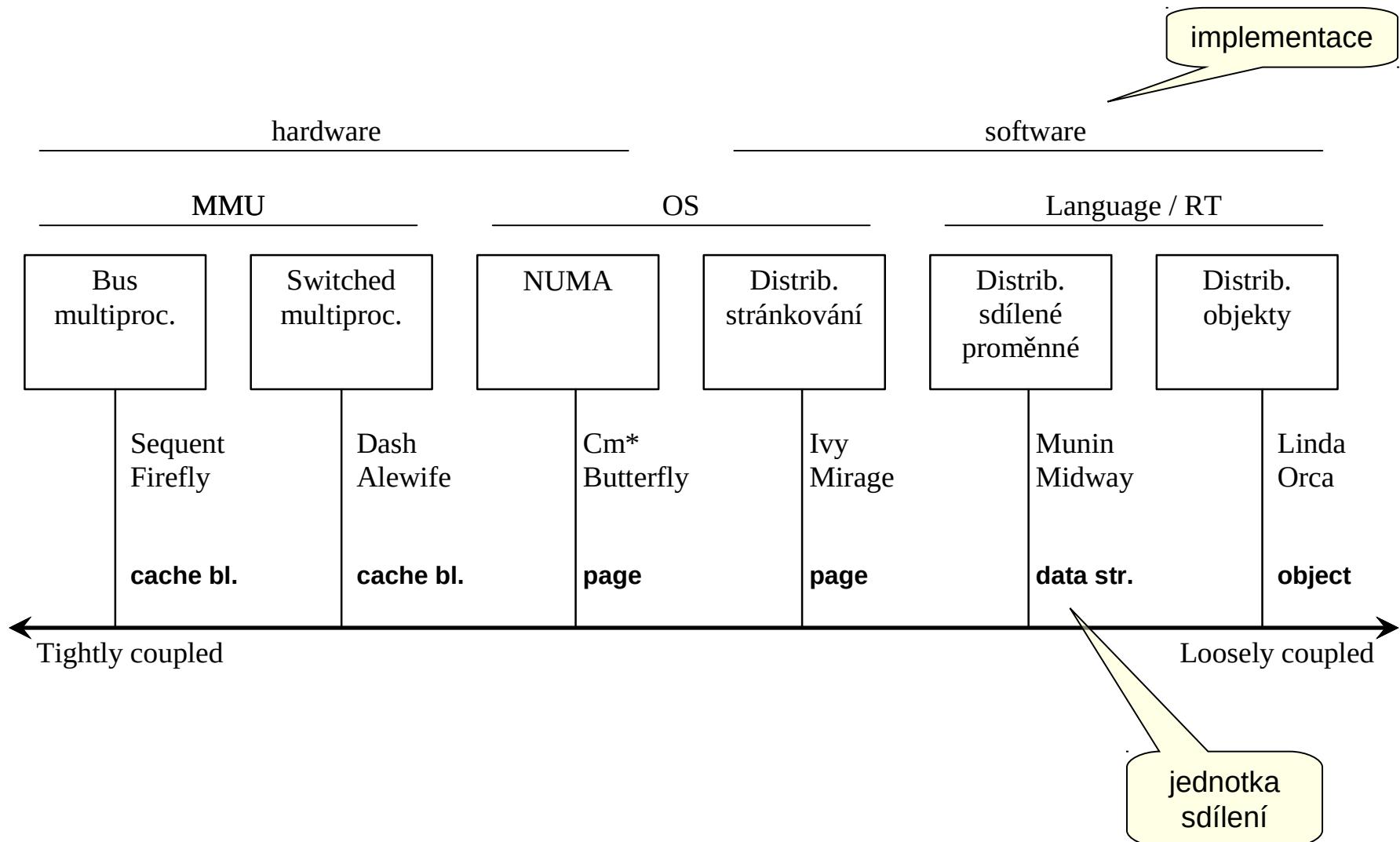
Distribuovaná sdílená paměť

■ Paralelní výpočty

- ◆ multiprocesory
 - malý počet procesorů
 - hardwarově náročné, drahé
- ◆ multicomputery
 - snadno dostupné
 - programování a synchronizace není prakticky (inženýrsky) dobře zvládnutá
 - pokus o řešení – distribuovaná sdílená paměť
 - 1986 Li & Hudak - DSM - množina uzlů sdílející adresový prostor
- ◆ RDMA



Mechanismy sdílení paměti



■ Primitiva

- ◆ Read, Write
- ◆ sdílený adresový prostor

■ Konzistenční model

- ◆ specifikace co implementace musí splňovat vzhledem k operacím čtení a zápis
- ◆ konzistenční modely pro virtuální paměť
- ◆ konzistenční modely pro knihovny / spec. jazyky - explicitní synchronizace

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 adresy x vrátí hodnotu uloženou při posledním zápisu na adresu x

- ◆ zajišťuje absolutní časové uspořádání
- ◆ nejsilnější, na jednoprocесорových 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: $R(x)0 \quad R(x)1$

paměť, která není striktně konzistentní

Sequential consistency

Výsledek výpočtu je ekvivalentní s

- všechny operace všech uzlů jsou vykonávány v nějakém sekvenčním uspořádání
- operace každého uzlu jsou vykonávány v pořadí daném 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: $R(x)1$ $R(x)1$

dvakrát spuštěný tentýž program – může dát různé výsledky

P1: $W(x)1$

P2: $R(x)0$ $R(x)1$

Sekvenční konzistence

P1:

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

P2:

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

P3:

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

šest instrukcí - $6! = 720$ možných uspořádání
pouze $3 \cdot (5!/4) = 90$ 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

Sekvenční a externí konzistence

■ 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

■ Externí konzistence

- ◆ **external consistency**
- ◆ distribuované databáze
- ◆ pro jakékoli dvě transakce T1, T2:
 - jestliže T2 začne operaci **poté**, co T1 ukončí operaci (commit)
 - potom logický čas T2 > logický čas T1
 - synchronizace hodin, nepřesnost \approx ms

Causal consistency

Kauzálně závislé zápisy musí být viděny všemi uzly 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 \approx odeslání, čtení \approx přijetí

	konkurenční zápisy		rozdílné čtení konkurenčních zápisů		
P1:	$W(x)1$		$W(x)3$		
P2:		$R(x)1$	$W(x)2$		
P3:		$R(x)1$		$R(x)3$	$R(x)2$
P4:		$R(x)1$		$R(x)2$	$R(x)3$

splňuje podmínu kauzální konzistence, nesplňuje sekvenční

P1: $W(x)1$

P2: $W(x)2$

P3: $R(x)2 \quad R(x)1$

P4: $R(x)1 \quad R(x)2$

kauzálně konzistentní rozvrh

P1: $W(x)1$

P2: $R(x)1 \quad W(x)2$

P3:

P4:

porušení kauzální konzistence
(přidaná operace čtení)

$R(x)2 \quad R(x)1$

$R(x)1 \quad R(x)2$

není kauzálně konzistentní

♦ implementace složitější

● vyžaduje vyžaduje udržování grafu závislostí zápisů na čtení

PRAM (Pipelined RAM) consistency

Zápisu prováděné jedním uzlem jsou viděny ostatními uzly v pořadí provádění.

Zápisu různých uzel mohou být viděny různými uzly 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:

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

P2:

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

P3:

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

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

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

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

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 ☺

➡ Eventual consistency

- 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

■ 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

- ◆ ☺ vyšší výkonnost
- ◆ ☹ ztráta transparentnosti

■ Speciální druh proměnné - **synchronizační proměnná**

- ◆ použití pouze pro synchronizační účely

S synchronizace

Acq vstup do kritické sekce (Acquire)

Rel výstup z kritické sekce (Release)

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

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

P2:

R(x)2 R(x)1 **S** R(x)2

P3:

R(x)1 R(x)2 **S** R(x)2

slabě konzistentní rozvrh

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

P2:

S R(x)1

porušení slabé konzistence

po synchronizaci musí
být data aktuální

- ◆ ☺ 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

Release consistency

1. Před přístupem ke sdílené proměnné musí být úspěšně ukončeny předchozí Acq() procesu.
2. Před provedením Rel() musí být ukončeny všechny předchozí zápisu 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

P1: **Acq** W(x)1 W(x)2 **Rel**

P2: **Acq** R(x)2 **Rel**

P3:

release consistency

R(x)1

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
 - optimizace síťového provozu

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 vázány 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 vstupovat a vystupovat z k sekce bez nutnosti komunikace
- ◆ proces, který není vlastníkem, musí požádat o vlastnictví

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

Nezávislá data x a y

Shrnutí konzistenčních modelů

bez SP

s SP

Konzistence	Vlastnosti
Striktní	Absolutní časové uspořádání
Sekvenční	Všechny události jsou vidět ve stejném pořadí, možná synchronizace s ext. časem
Kauzální	Kauzálně vázané události jsou vidět ve stejném pořadí
PRAM	Události jednoho uzlu jsou vidět ve stejném pořadí
Slow mem	Zápisu jednoho uzlu na jedno místo jsou vidět ve stejném pořadí
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 úrovni virtuální paměti
- ◆ procesy nemusí o DSM vůbec vědět
- ◆ standardní programovací jazyky / knihovny

■ Konzistenční modely s použitím synchronizačních proměnných

- ◆ speciální jazyky / knihovny / framework / middleware
- ◆ procesy musí být korektně naprogramovány
- ◆ potenciálně vyšší výkonnost

■ Obdoba virtuální paměti

- ◆ přístup k nemapované stránce - přerušení, obsluha, načtení
- ◆ read-only mapování, při zápisu přerušení → synchronizační akce

■ Obdoba virtuální paměti

- ◆ přístup k nenamapované stránce - přerušení, obsluha, načtení
- ◆ read-only mapování, při zápisu přerušení → synchronizační akce

■ Problémy

- ◆ nalezení stránky *jak nalézt distribuovaná data*
 - centralizovaný manager
 - replikovaný manager - indexace spodními n byty / hash
 - broadcast

■ Obdoba virtuální paměti

- ◆ přístup k nemapované stránce - přerušení, obsluha, načtení
- ◆ read-only mapování, při zápisu přerušení → synchronizační akce

■ Problémy

- ◆ nalezení stránky *jak nalézt distribuovaná data*
 - centralizovaný manager
 - replikovaný manager - indexace spodními n bity / hash
 - broadcast
- ◆ správa kopií *co dělat s kopii při čtení / zápisu*
 - broadcast
 - copyset - vlastník stránky udržuje množinu lokací kopií

■ Obdoba virtuální paměti

- ◆ přístup k nemapované stránce - přerušení, obsluha, načtení
- ◆ read-only mapování, při zápisu přerušení → synchronizační akce

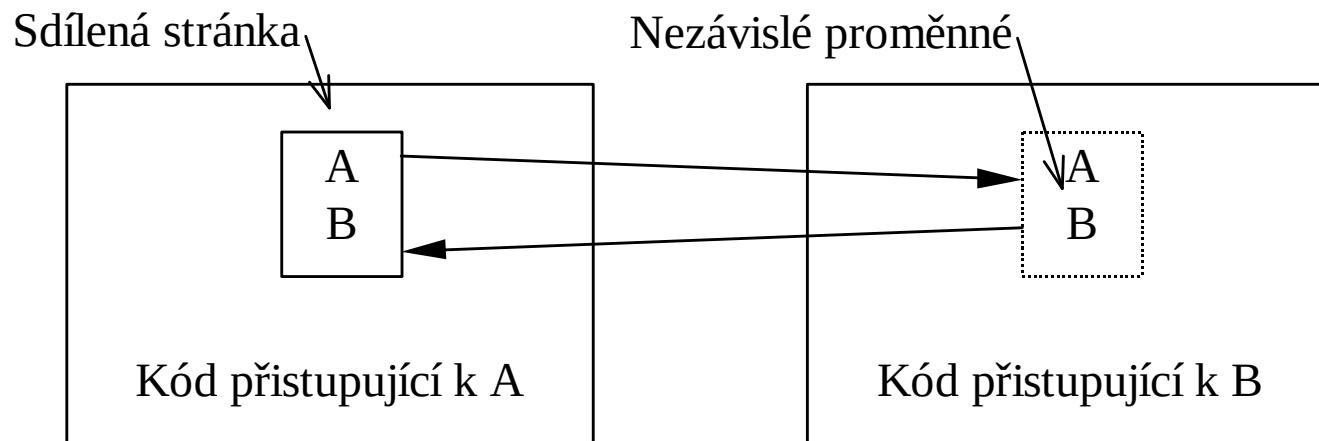
■ Problémy

- ◆ nalezení stránky *jak nalézt distribuovaná data*
 - centralizovaný manager
 - replikovaný manager - indexace spodními nábyty / hash
 - broadcast
- ◆ správa kopií *co dělat s kopii 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*

■ Problémy

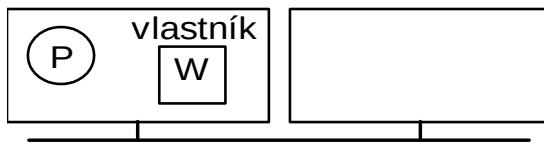
- ◆ falešné sdílení

nezávislá data na stejné stránce

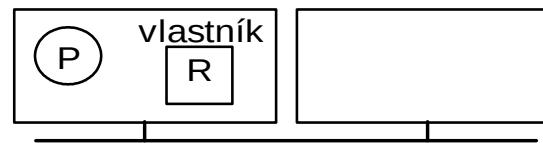
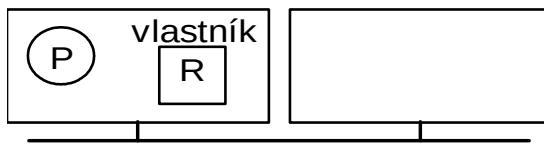
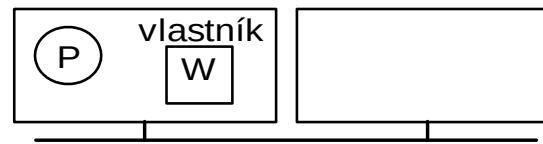


Sekvenčně konzistentní distribuované stránkování

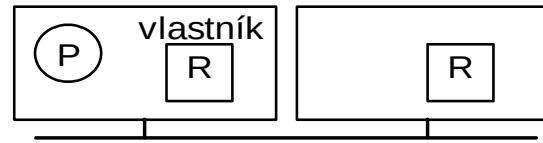
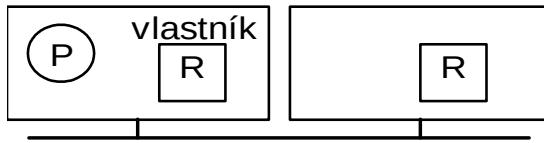
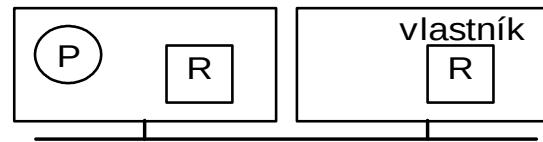
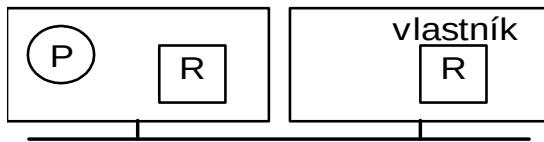
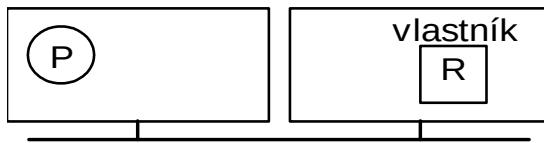
čtení



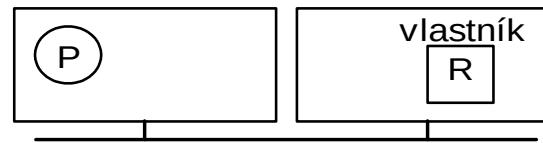
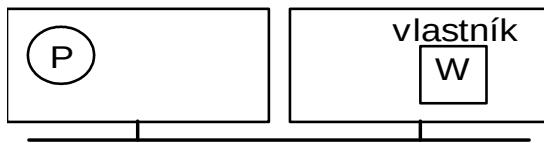
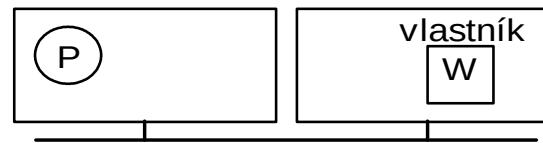
zápis



Povýšení

Invalidace,
povýšeníInvalidace,
změna vlastníka,
povýšení

Kopie

Invalidace, kopie,
změna vlastníka,
povýšeníDegradace,
kopieInvalidace, kopie,
změna vlastníka,
povýšení

kradení - viskozita

■ základní idea - graf závislostí

■ vektorové hodiny:

- ◆ VT_S jednotka granularity (stránky)
- ◆ VT_P procesy

■ $VT[i] \approx$ stránka i

■ výpadek stránky - přenos dat, $VT_P = \max(VT_S, VT_P)$

■ zápis do stránky - $VT_S = \text{inc}(VT_P)$

■ aktualizace VT_P - **zneplatnění stránek** i: $VT_{Si}[i] < VT_P[i]$

■ Problémy:

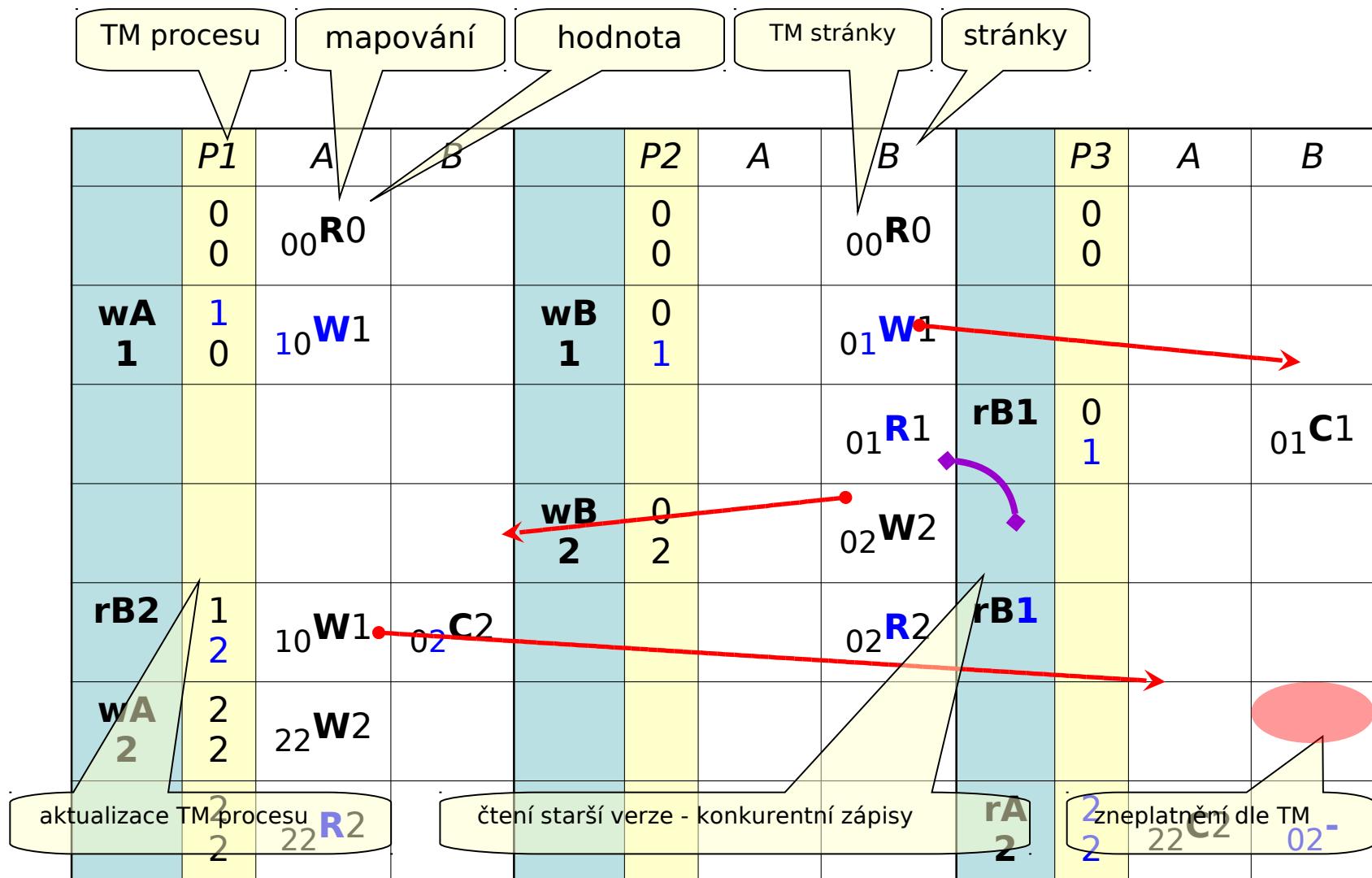
- ◆ velká prostorová režie (1 MB = 256 stránek = 512 B / stránku)
- ◆ propagace konkurenčních zápisů
 - lámky, bariéry, timeouty

na kterých stránkách
závisí obsah

z kterých stránek
znám data

jak OS pozná zápis?
změna mapování

Kauzálně konzistentní distribuované stránkování



W - právo zápisu, R - r/o vlastník, C - r/o kopie

- základní idea - graf závislostí
- vektorové hodiny:
 - ◆ VT_S jednotka granularity (stránky)
 - ◆ VT_P procesy
- $VT[i] \approx$ stránka i
- výpadek stránky - přenos dat, $VT_P = \max(VT_S, VT_P)$
- zápis do stránky - $VT_S = \text{inc}(VT_P)$
- aktualizace VT_P - **zneplatnění stránek** i: $VT_{Si}[i] < VT_P[i]$
- Problémy:
 - ◆ velká prostorová režie (1 MB = 256 stránek = 512 B / stránku)
 - ◆ propagace konkurenčních zápisů
 - zámky, bariéry, timeouty

■ 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

■ 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, ...

■ 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

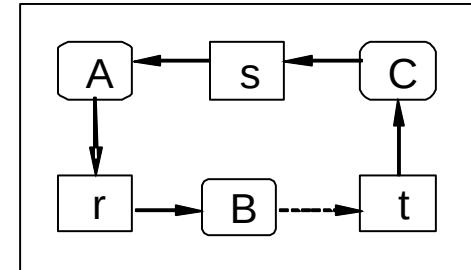
■ Distribuované objekty

- ◆ 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

- 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 - exception
- konvenční sdílená data (*nepatřící do žádné z výše uvedených kategorií*)
 - ◆ jako distribuované stránkování - single writer/many readers
 - ◆ sekvenční konzistence

Správa prostředků a procesů

- Centralizované řešení - resource manager
 - teoretické pokusy o distribuovanou správu
 - prakticky nepoužitelné - *distribuované vyloučení procesů*
- Uváznutí (deadlock)
 - ◆ časté řešení - pštrosí algoritmus
 - ◆ detekce - větší problém než u centralizovaných systémů
 - ◆ WFG - Wait-For-Graph
 - orientovaný graf (procesů, transakcí)
 - $P_1 \rightarrow P_2$ proces P_1 je blokován procesem P_2
 - $P \rightarrow R$ proces P žádá o prostředek R
 - $R \rightarrow P$ proces P drží prostředek R
 - orientovaná kružnice – uváznutí



■ Korektnost algoritmu detekce deadlocku

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

■ Metody konstrukce WFG

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

■ Ho-Ramamoorthy, Bernstein

■ Přenos informací:

- ◆ po každé změně lokálního stavu
- ◆ v pravidelných intervalech
- ◆ na požádání

■ Problém - falešné uváznutí kvůli zpoždění zpráv

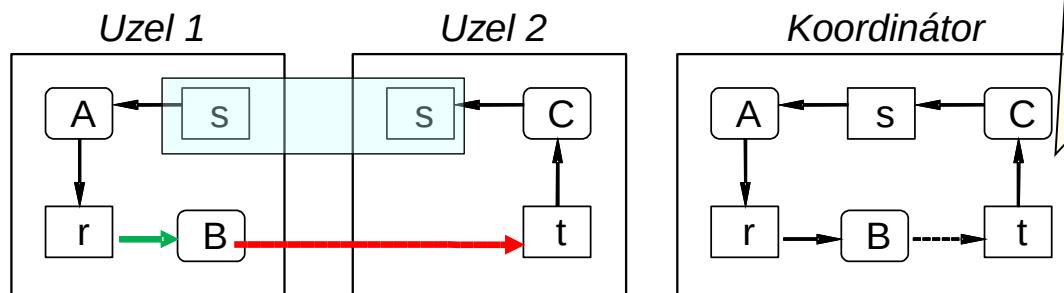
viz též: nekonzistentní řez

- ◆ řešení: logické hodiny, kauzální doručování
- ◆ alternativně: při detekci (*podezření*) kontrola

■ Rozšíření - hierarchický algoritmus

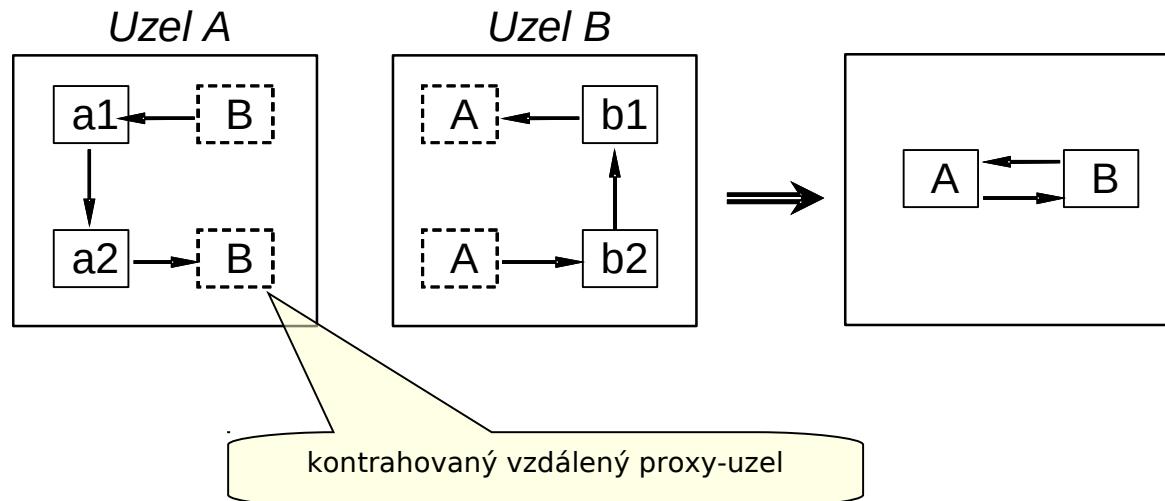
- ◆ uzel řeší deadlocky lokálně
- ◆ koordinátor řeší deadlocky podřízených uzlů

fyzické pořadí: **rel r->B, acq B->t**
 doručení: **acq B->t, rel r->B**

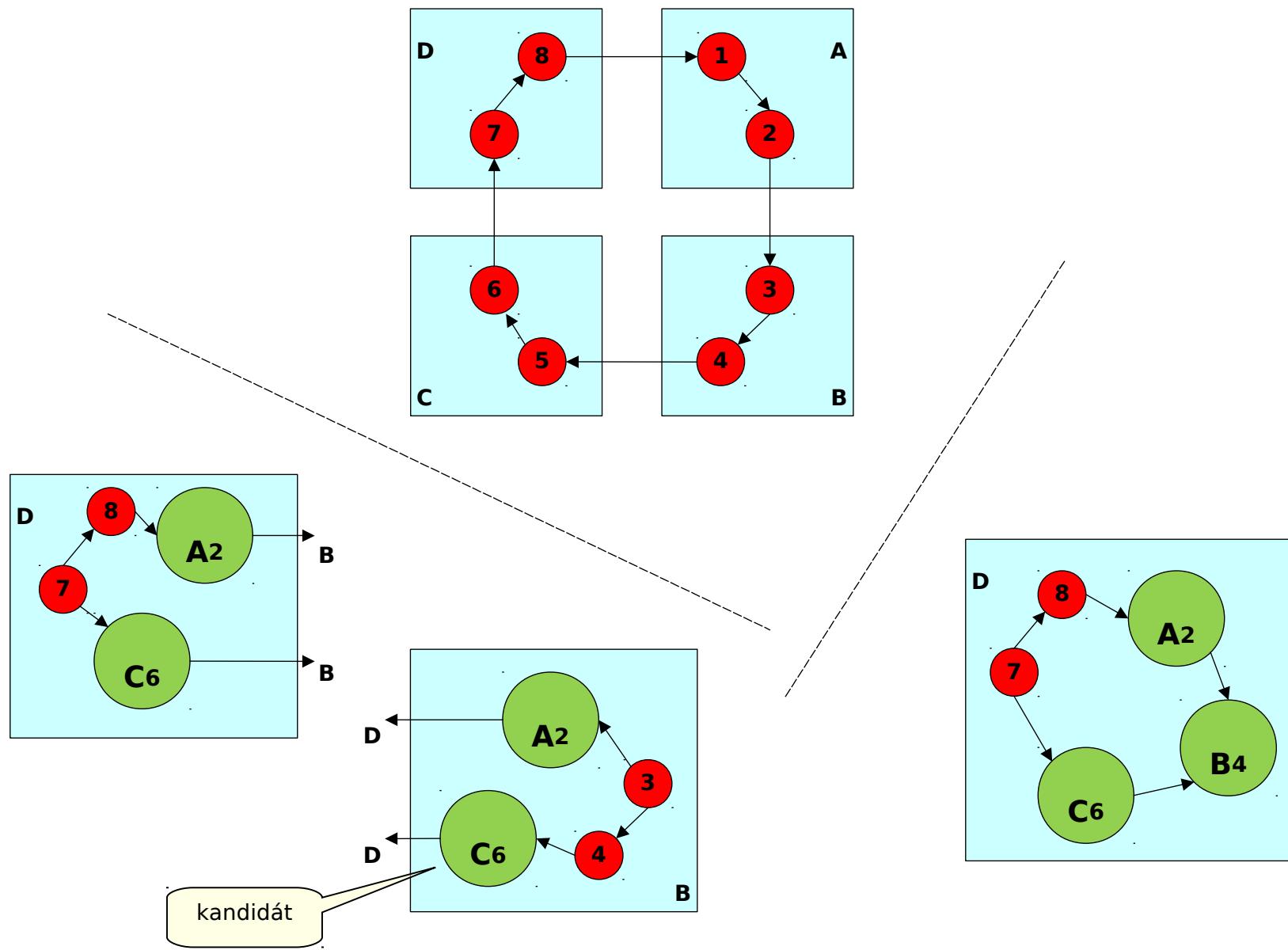


Path-pushing

- Obermarck, Menasce-Muntz, Gligor-Shattuck, Ho-Ramamoorthy
- WFG distribuovaný, uzly spravují lokální části WFG
- Sousedním uzlům jsou zasílány externí závislosti
- Některé publikované algoritmy/protokoly nekorektní
 - ◆ phantom deadlock

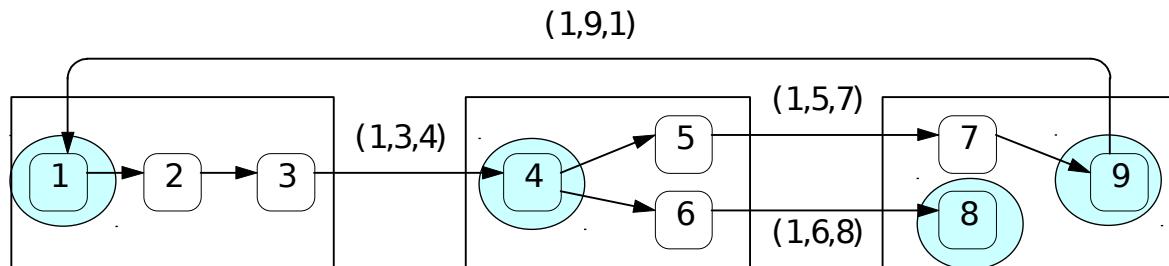


Path-pushing



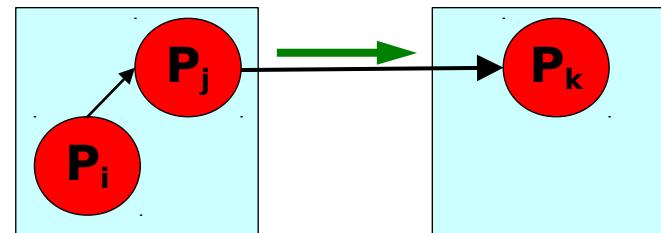
Edge-chasing

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



- Modely
 - ◆ and / or / n-nad-m

Edge-chasing Chandy-Misra-Haas



Sending the probe:

if P_i is locally dependent on itself then deadlock

else for(all P_j and P_k such that P_i is locally dependent upon P_j

& P_j is waiting on P_k & P_j and P_k are on different sites)

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

Receiving the probe(i,j,k):

if P_k is blocked & $\text{dependent}_k(i)$ is false & P_k has not replied to all requests of P_j { ~~dependent_k(i) := true;~~

if $k = i$ then P_i is deadlocked

else ...

send probe similarly

}

Pk ví o tom, že
Pi čeká na Pk

■ Diffusing computation

- ◆ Chandy-Misra, Chandy-Herrmann
- ◆ 'distribuovaný výpočet' podél WFG
 - aplikace značkového algoritmu pro detekci ukončení
- ◆ první zpráva: propagace, další zprávy: signál, všechny signály: signál otci
- ◆ 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
- ◆ varianty
 - celý WFG iniciátorovi
 - lokální kontrakce WFG , externí závislosti zaslány iniciátorovi
 - redukovaný WFG sousedům (ala edge-chasing)

■ Diffusing computation

- ◆ Chandy-Misra, Chandy-Herrmann
- ◆ 'distribuovaný výpočet' podél WFG
 - aplikace značkového algoritmu pro detekci ukončení
 - ◆ první zpráva: propagace, další zprávy: signál, všechny signály: signál otci
 - ◆ pokud se zpráva dostane k iniciátorovi, vracejí se signály
 - ◆ iniciátor po obržení signálu může rozhodnout
 - ◆ vhodné pro složitější modely (OR, n-nad-m)

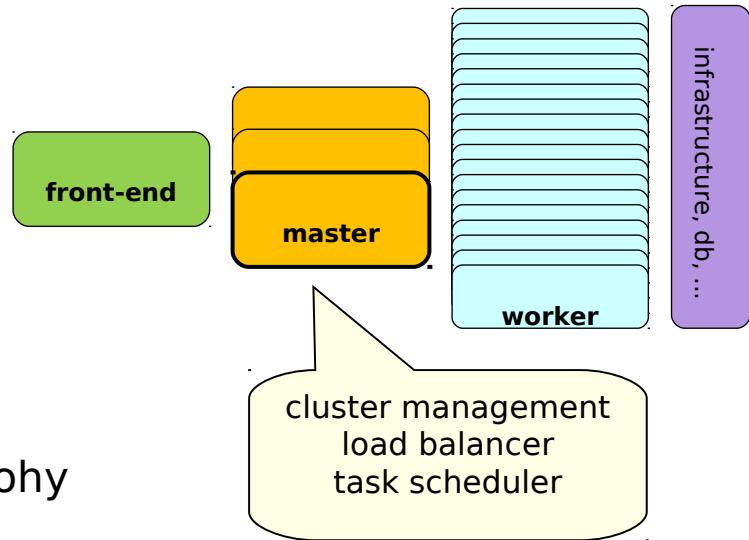
■ Detekce globálního stavu

- ◆ 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
 - ◆ varianty
 - celý WFG iniciátorovi
 - lokální kontrakce WFG , externí závislosti zaslány iniciátorovi
 - redukovaný WFG sousedům (ala edge-chasing)

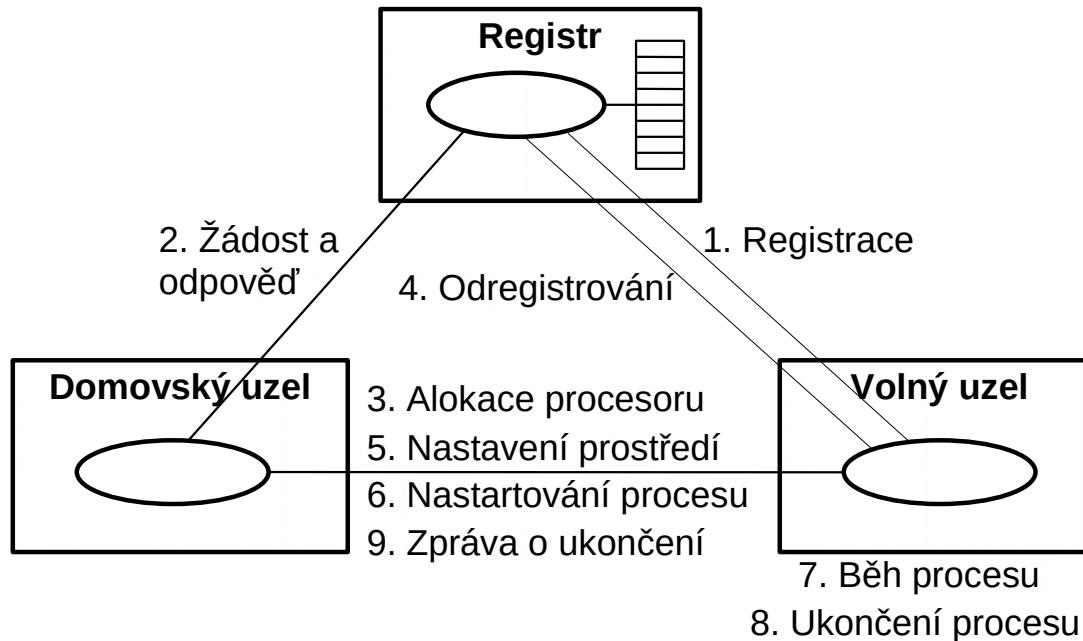
- Správa procesů v distribuovaných systémech
 - sdílet výpočetní sílu systému
 - rozdělovat zátěž na jednotlivé procesory
 - synchronizovat procesy a vést evidenci stavu
- Rozdílné cíle i prostředky
 - služby vnějším klientům, cluster
 - krátkodobé úlohy - minimalizace latence
 - dlouhodobé úlohy - maximalizace výkonu
 - centralizovaná / hierarchická správa
 - kooperativní systém, distribuované výpočty
 - rovnoměrné sdílení výkonu
 - decentralizovaná / peer-to-peer správa
- Load balancing / vyvažování zátěže
- Kooperativní systémy
- Migrace procesů / virtuálních strojů

Load balancing

- homogenní prostředí
- centralizovaná / replikovaná správa
- load balancer / task scheduler
- vyvažovací strategie
 - Round Robin
 - homogenní systém, krátké/srovnatelné úlohy
 - Weighted Round Robin
 - váha dle výkonnosti
 - Dynamic Round Robin
 - periodické měření aktuální výkonnosti, vyhlazení, klouzavý průměr
 - Least Connections, Weighted LC
 - nejmenší počet otevřených úloh
 - Random, Threshold
 - heterogenní výkonnost i úlohy, neznámé vlastnosti
 - Agent-Based Adaptive Balancing, Resource-based Scheduling
 - 'chytrější' algoritmy na základě aktuální výkonnosti a dalších parametrů
 - ne vždy efektivnější, výrazně vyšší režie



■ Spuštění vzdáleného procesu



■ Ukončení vzdáleného procesu

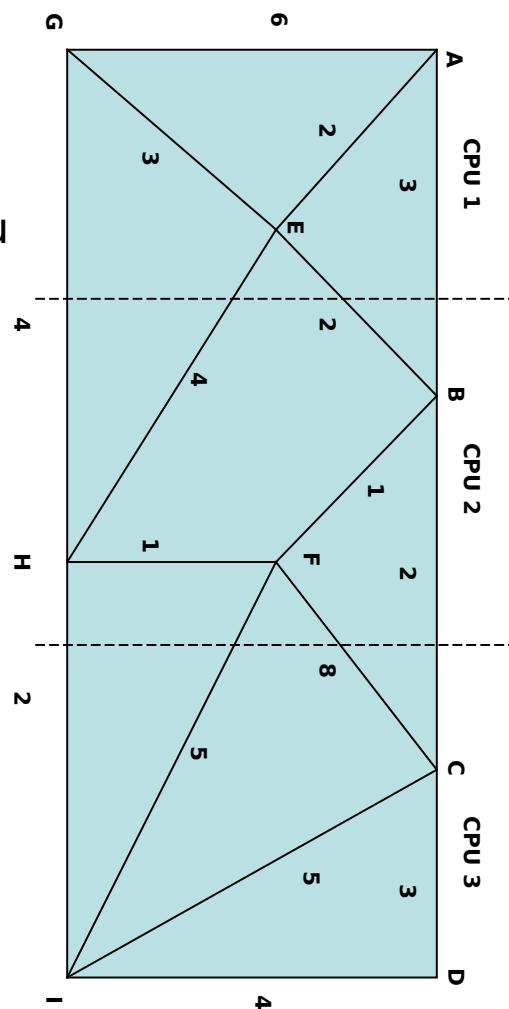
(uživatel se vrátil z oběda, potřeba restartu, ...)

- ◆ doběhnutí
- ◆ dokončení / rollback transakce
- ◆ zabítí
- ◆ čas na uložení / uzavření
- ◆ *migrace*

Kooperativní load balancing

- Distribuovaný výpočet, uzly inicují další procesy

- ◆ centralizované/hierarchické řízení přístupu
 - manažeři skupin, při neúspěchu žádost nadřazenému
 - up-down algoritmus ↗
- ◆ distribuovaný heuristický algoritmus
 - k náhodných výběrů cíle
 - server / receiver initiated
- ◆ deterministický grafový algoritmus
 - minimalizace komunikace
 - nutnost znalosti komunikační složitosti
 - optimální deterministický algoritmus – tok v sítích
- ◆ bidding (obchodní, nabídkový) algoritmus
 - procesy kupují výpočetní sílu, procesory ji nabízejí



Up-down algoritmus

■ Mutka-Livny 1987

- optimalizace: rovnoměrné sdílení výkonu

■ Koordinátor

- tabulka se záznamem pro každý uzel obsahující "trestné body"

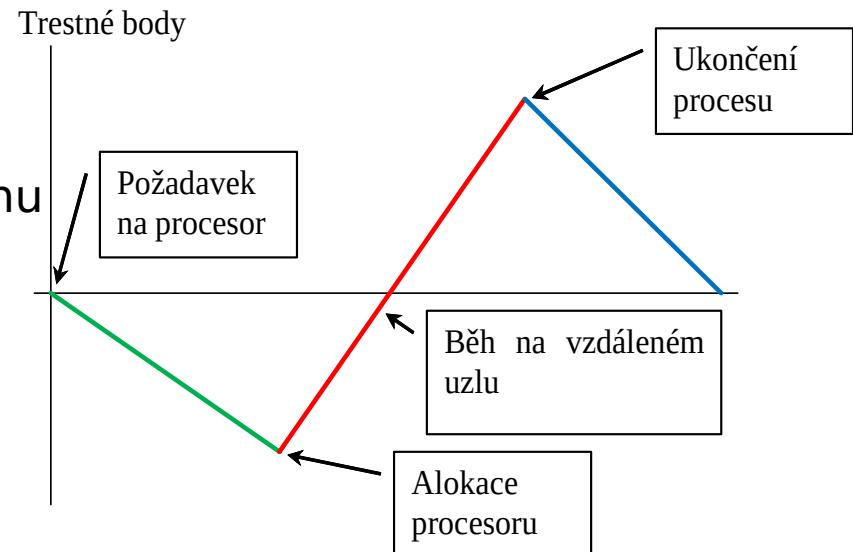
■ Při významné akci

(vytvoření procesu, ukončení procesu, tik hodin)

zpráva koordinátoru, který provede změny:

- každý proces běžící na jiném uzlu - **plus** trestné body
- každý neuspokojený požadavek - **mínus** trestné body
- jestliže nic z tohoto - směrem **k nule**

■ Uvolnění uzlu: proces z fronty (*neuspokojených požadavků*), jehož vysílající uzel má nejméně trestných bodů



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 ...
- ◆ komunikace s ostatními procesy
- ◆ reziduální dependence - žádná, domácí, průběžná, dočasná
- ◆ vícenásobná migrace - viskozita, hystereze

■ Migrace procesů ✽ virtuálních strojů, kontejnery

■ 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

■ 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
- ◆ Post-copying / 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, ...

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 - **cílový uzel** si proces „tahá“ k sobě
- ◆ Přesun adresového prostoru a paměti (*Cíl*)
- ◆ Forward čekajících zpráv na cílový uzel (*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

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 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 zdrojovém uzlu zavolá „Migrate Out“
- ◆ podrobnější informace o procesu na cílový uzel - „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 zdrojového uzlu

Stanford (Cheriton) 1984-86

sítově transparentní prostředí, 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 zdrojovém uzlu kopíruje adresový prostor; proces stále běží
- ♦ migrovaný logický host je zamražen, dokončí se kopírování modifikované paměti
- ♦ kopírování stavu z jádra zdrojového uzlu

Odmražení, přesměrování odkazů

- ♦ smaže se stará kopie logical hostu, nová se odmrazí
- ♦ broadcast nové adresy logical hostu



Technion (Barak, Shiloh, ...) (1977!) - 1988 - 2008 - ...!

FreeMosix, Linux Process Migration Infrastructure

Mosix 4.4.4 - 24.10.2017

Multicomputer **O**perating **S**ystem for **u**n**I**X

UNIX adaptovaný na distribuované prostředí, rozsáhlé **vyvažování zátěže**

Jeden z mála dlouhodobě **reálně používaných** distribuovaných systémů

⇒ Cluster Management System

Rozšíření struktury procesu:

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

Vlastní migrace

Příprava

- ◆ Cílový uzel zjistí, jestli si může dovolit příjem procesu
- ◆ Zajistí opravu komunikačních kanálů
- ◆ Nastaví se paměťové oblasti
- ◆ Přenos procesu

Přesun dat - adresový prostor

Rozběhnutí odmigrovaného procesu, vyčištění zdrojového uzlu

Berkley (*Douglis, Ousterhout, Welch*) 1992

- Předpoklady:
 - mnoho nevyužitých uzlů
 - po návratu vlastníka potřeba uvolnit (odmigrovat)
- Cíle
 - rychlosť, maximální transparency
- Návrh migrace
 - vlastní migrace jádro / sběr statistiky a rozhodování provádí uživatelský proces
 - původní idea: všechna volání jádra forwardovat na domácí uzel
 - transparency vs. reziduálních dependence
 - ze 106 volání Sprite: 91 lokálně, 11 forwardováno, 4 se řeší kooperací
- Virtuální paměť
 - kombinace přesunu veškeré paměti najednou a „copy-on-reference“
- **Jednotný mechanismus** migrace různých entit
 - každá součást stavu má definovány 4 rutiny
 - pre-migration, encapsulation, de-encapsulation, post-migration routine

MFF UK (Bednárek, Merta, Yaghob, Zavoral) 1994-97

Systém, který by

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

Hlavní části

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

Zkušenosti s T4

- výuka, studentské projekty, experimenty
 - ++ diplomové práce, PGDS
- další výzkum v oblasti distribuovaných systémů
 - + publikace (*málo*), výzkum jiným směrem
- pro běžné každodenní použití
 - aplikace !!! (Tetris, DOOM)

Migratory load balancing

■ Rozhodnutí o okamžiku migrace

- ◆ jak porovnávat zatížení uzelů
- ◆ udržování konzistence údajů
- ◆ volba migrujícího procesu
- ◆ volba příjemce

■ Migrační jednotka

- ◆ proces
- ◆ skupina procesů / task
- ◆ virtuální stroj, kontejner

■ Párový algoritmus (Bryant & Finkel, Charlotte)

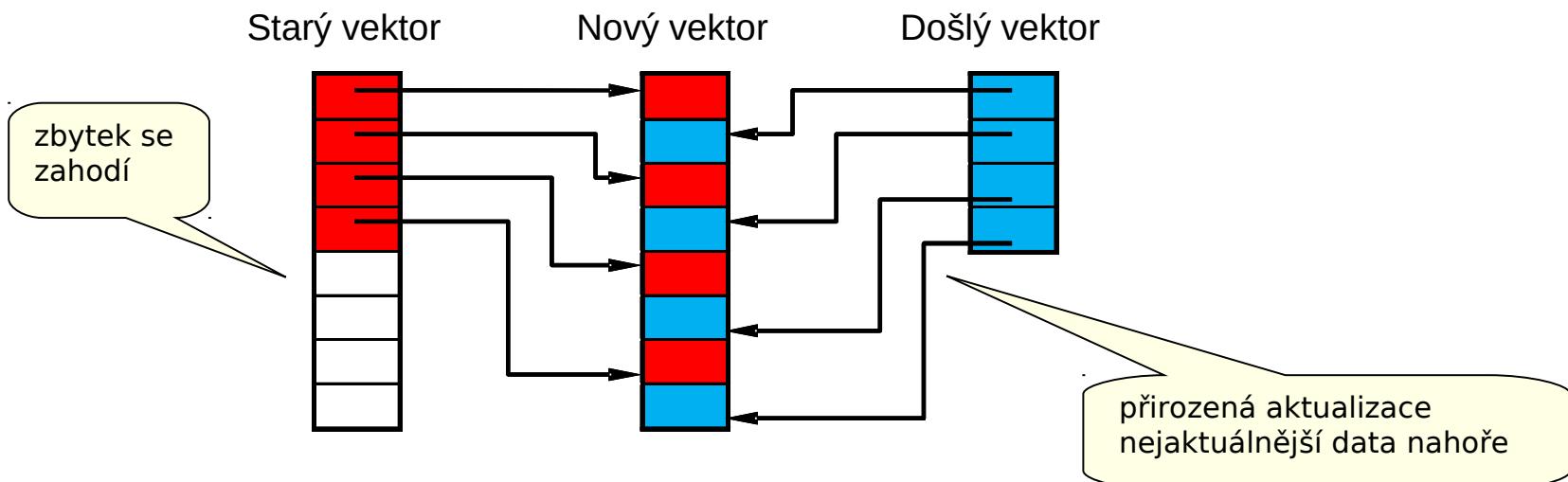
- ◆ vytvářejí se páry, které se vzájemně vyvažují
- ◆ A pošle B žádost o vytvoření páru se seznamem procesů
- ◆ B odmítne / vytvoří pár / migruje
- ◆ zatíženější uzel vybere proces podle míry vylepšení
 - $k_i = A_i / (B_i + AB_i)$
- ◆ významné zlepšení stavu \Rightarrow migrace a další proces, jinak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

- Rozhodnutí o okamžiku migrace
 - ◆ jak porovnávat zatížení uzelů
 - ◆ udržování konzistence údajů
 - ◆ volba migrujícího procesu
 - ◆ volba příjemce
- Migrační jednotka
 - ◆ proces
 - ◆ skupina procesů / task
 - ◆ virtuální stroj, kontejner
- **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

- ◆ distribuce a aktualizace zátěže
- ◆ pevný vektor zátěže L , $L[0] =$ vlastní zátěž
- ◆ periodicky každý uzel provádí:
 - zjistí vlastní zátěž
 - náhodně pošle polovinu vektoru
- ◆ při příjmu L' : $L_{n+1}[2i] = L[i]$, $L_{n+1}[2i+1] = L'[i]$
- ◆ k zátěži se přičte komunikační režie



■ Bidding algoritmus

- ◆ 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

- ◆ stochastic learning automata - zpětné učení

■ BDT algoritmus

- ◆ bayesian decision theory - posílání globálních stavů

neprosadily se
příliš komplikované

■ Centralizovaný / hierarchický algoritmus

- ◆ centralizovaný manažer, zná zátěž všech uzlů, velí
- ◆ virtualizace, datová centra: $\approx 1k\text{-}10k$ uzlů / manažer
- ◆ hierarchicky organizované skupiny, nadřazení manažeři

idea:
vyvažující se skupiny
nebývají příliš velké

■ 'Lokální' algoritmus

- ◆ známa pouze lokální zátěž, prahová hodnota
- ◆ žádost n uzlům, podprahová / nejlepší odpověď OK
- ◆ sender/receiver initiated

jednoduchý
suboptimální
použitelný 😊

■ Virtuální stroje

- ◆ primární: paměť, procesor
- ◆ sekundární: síťování, disky
- ◆ požadavky:
 - stejná architektura (společná podmnožina)
 - nízká latence síťového spojení
 - 100 ms RTT: long-distance live migration NY ↔ London
 - sdílený filesystem / migrace datastore

Replikace

- replikace dat / souborů - udržování více kopií na více db/fileserverech
 - ◆ spolehlivost (*reliability*) - při havárii serveru nejsou ztracena data
 - ◆ dostupnost (*availability*) - k datům lze přistupovat i při výpadku serveru
 - ◆ výkon (*performance*) - přístup k nejbližším datům, rozdělení výkonu

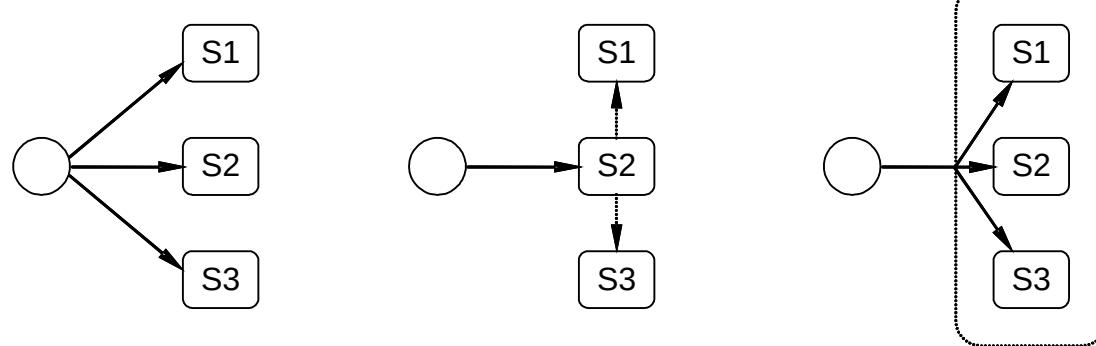
- explicitní replikace
 - ◆ klient se sám stará o udržování konzistence

- odložená replikace
 - ◆ zápis do primární repliky, aktualizace sekundárních

- skupinová komunikace
 - ◆ zápis y simultánně zasílány všem dostupným replikám

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

typicky knihovny



■ Problém aktualizací kopií

- ◆ primární kopie

kdy a které kopie budou aktualizovány

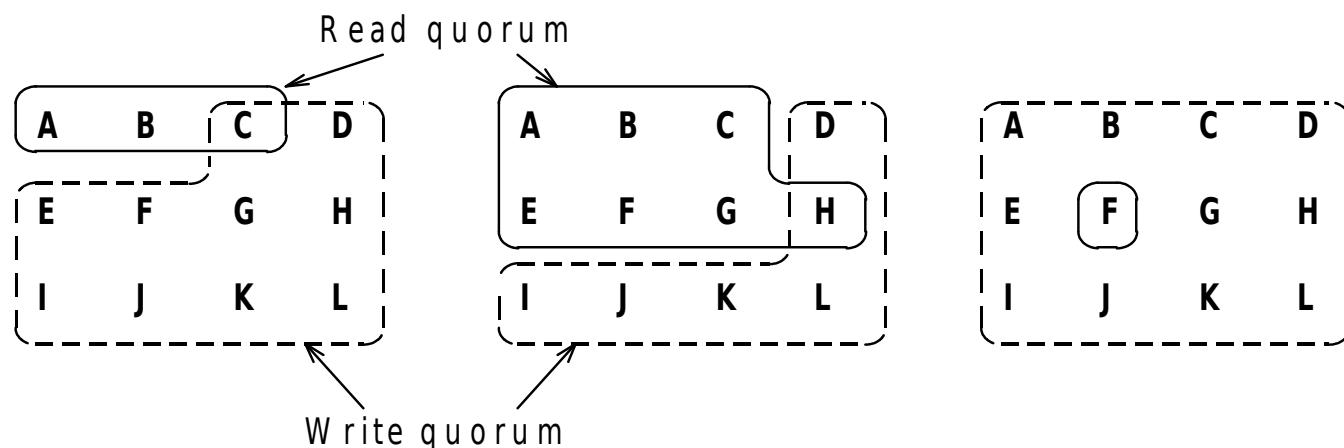
■ Problém aktualizací kopií

- ◆ primární kopie
- ◆ většinové hlasování (*majority voting*)
 - $Nr > N/2$
 - $Nw > N/2$

speciální případ váženého hlasování

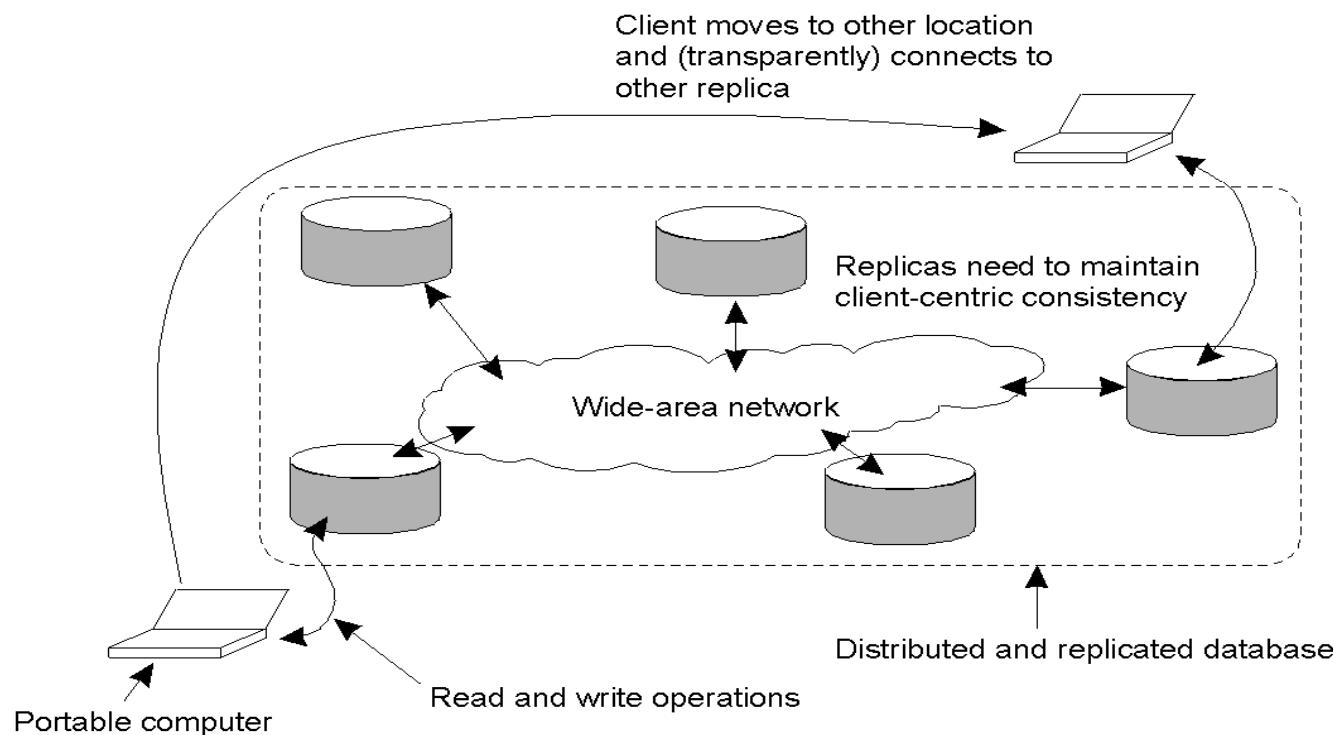
■ Problém aktualizací kopií

- ◆ primární kopie
- ◆ většinové hlasování (*majority voting*)
- ◆ vážené hlasování (*weighted/quorum voting*)
 - read / write quorum:
 - $Nr + Nw > N, 2 * Nw > N$
 - většinové hlasování \approx vážené při $Nr = Nw$
 - \approx sekvenční rozvrh (*one copy serializability*)
 - optimalizace čtení × zápis
 - optimalizace × availability
- ◆ dynamická kvóra, multidimenzionální kvóra



Klientocentrické konzistenční modely

- Konzistenční modely DSM - synchronizace dat mezi **různými klienty**
- Replikovaná databáze
 - ◆ zápisy málo časté, masivně paralelní čtení
 - ◆ není potřeba vzájemná synchronizace klientů
 - ◆ WWW + cache, mobile computing, News, ...
 - ◆ session consistency levels of cloud databases (Cosmos DB)
- Klientocentrický model - pohled na replikovaná data **jedním klientem**



■ Eventual consistency

Po ukončení všech zápisů budou všechny repliky v konečném čase aktualizovány

■ Problém: pohled jednoho klienta na data různých replik

Značení:

x_i hodnota proměnné x na replice L_i

$W(x_i)$ prvotní zápis hodnoty x_i

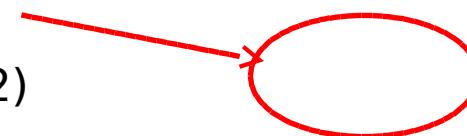
$S(x_i)$ posloupnost operací na L_i vedoucí k hodnotě x_i

$S(x_i ; x_j)$ posloupnost x_i předchází x_j , $S(x_i) \subset S(x_j)$

A, B - připojení jednoho klienta k různým replikám

A: $S(x_1)$ $R(x_1)$

B: $S(x_2)$ $R(x_2)$ $S(x_1;x_2)$



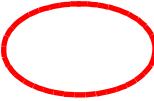
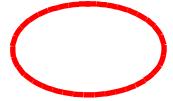
Příklad: při připojení k jedné replice uživatel vidí zprávy, po připojení k jiné replice některé zprávy (které již viděl) 'ještě' nevidí

■ Monotonic read consistency

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

A: $S(x_1)$ $R(x_1)$
B: $S(x_1;x_2)$ $R(x_2)$

Vyhovuje monotónnímu čtení

A: $S(x_1)$  $R(x_1)$ 
B: $S(x_2)$ $R(x_2)$ $S(x_1;x_2)$

Nevyhovuje monotónnímu čtení

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

■ Monotonic write consistency

Zápis proměnné je proveden před jakýmkoliv následným zápisem této proměnné

Vyhovuje monotónnímu zápisu

A: $W(x_1)$

B: $S(x_1)$ $W(x_2)$

Nevyhovuje monotónnímu zápisu

A: $W(x_1)$



B: $W(x_2)$

Příklad: SVN commit na různých replikách

■ Read your writes consistency

Zápis proměnné je proveden před jakýmkoliv následným čtením této proměnné

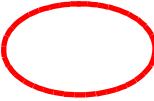
Vyhovuje č.v.z.

A: $W(x_1)$

B: $S(x_1; x_2)$

$R(x_2)$

Nevyhovuje č.v.z.

A: $W(x_1)$ 

B: $S(x_2)$ $R(x_2)$

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 stejně nebo novější hodnotě

Vyhovuje z.n.č.

A: $S(x_1)$ $R(x_1)$

B: $S(x_1; x_2)$ $W(x_2)$

Nevyhovuje z.n.č.

A: $S(x_1)$ $R(x_1)$

B: $S(x_2)$ $W(x_2)$

Příklad: zápis odpovědi do diskusního fóra 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: $\text{WID} = \text{repl_id} + \text{loc_id}$
- každý **klient** udržuje dvě množiny identifikátorů: **read-set**, **write-set**
- Monotónní čtení
 - ◆ **při čtení** replika **serveru** ověří podle **read-set** klienta 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 serveru ověří podle **write-set** klienta aktuálnost svých zápisů
 - chybějí-li nějaké, zapíše je
 - ◆ po zápisu si klient aktualizuje **write-set**
- Čtení vlastních zápisů
 - ◆ při čtení replika serveru ověří podle **write-set** aktuálnost svých zápisů
 - jiné možné řešení - forward čtení na aktuální repliku
- Zápisы následují čtení
 - ◆ aktualizace repliky podle **read-set**
 - ◆ aktualizace **read-set** i **write-set** klienta

Naïvní implementace

- každému **zápisu** je přiřazen globálně jednoznačný identifikátor **WID**
 - ◆ přiřazuje replika kde byl zápis proveden klientem: $\text{WID} = \text{repl_id} + \text{loc_id}$
- každý **klient** udržuje dvě množiny identifikátorů: **read-set**, **write-set**
- Monotónní čtení
 - ◆ při **čtení** replika serveru ověří podle **read-set** klienta 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 serveru ověří podle **write-set** klienta aktuálnost svých zápisů
 - chybějí-li nějaké, zapíše je
 - ◆ po zápisu si klient aktualizuje **write-set**
- Čtení vlastních zápisů
 - ◆ při čtení replika serveru ověří podle write-set aktuálnost svých zápisů
 - jiné možné řešení - forward čtení na aktuální repliku
- Zápisы následují čtení
 - ◆ aktualizace repliky podle read-set
 - ◆ aktualizace read-set i write-set klienta

Naïvní implementace

- každému **zápisu** je přiřazen globálně jednoznačný identifikátor **WID**
 - ◆ přiřazuje replika kde byl zápis proveden klientem: $\text{WID} = \text{repl_id} + \text{loc_id}$
- 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 serveru 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 serveru ověří podle **write-set** aktuálnost svých zápisů
 - jiné možné řešení - forward čtení na aktuální repliku
- Zápisы 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

Efektivnější implementace

■ Problém: read-set i write-set neomezeně rostou

■ Možné řešení: seskupení do relací (session)

■ Reprezentace množin - vektorové hodiny

- *Bayou - distributed high-availability service*
- 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

myšlenka:
stačí detektovat
poslední R/W
každé repliky

obecná pravidla:

- $VT(A)[i] = \max \{ TS \text{ operací } A \text{ iniciovaných na replice } Si \}$
- 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: $VT(RS)[j] = \max(VT(RS)[j], RCV(i)[j]) \quad \forall j$

● zápis: $VT(WS)[j] = \max(VT(WS)[j], RCV(i)[j]) \quad \forall j$

■ read/write-set reprezentuje poslední operace zápisu, které klient viděl/zapisoval

čte i data pocházející
z jiných replik

Epidemické protokoly

Možná implementace eventuální konzistence

100 000
1 000 000
.....

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

■ problém: kdy má uzel přestat infikovat

Epidemické protokoly

Možná implementace eventuální konzistence

Optimalizace komunikace ve VELMI rozsáhlých systémech, neřeší konflikty

Teorie epidemií - infekční nákaza

- rozšíření infekce co nejrychleji na co největší počet uzelů

Antientropie - server P náhodně vybere server Q k výměně dat

- možné výměny:
 - ◆ push, pull, push/pull
- 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: masivní škálovatel
- složitější topologie - hierarchie, rychlejší infekce

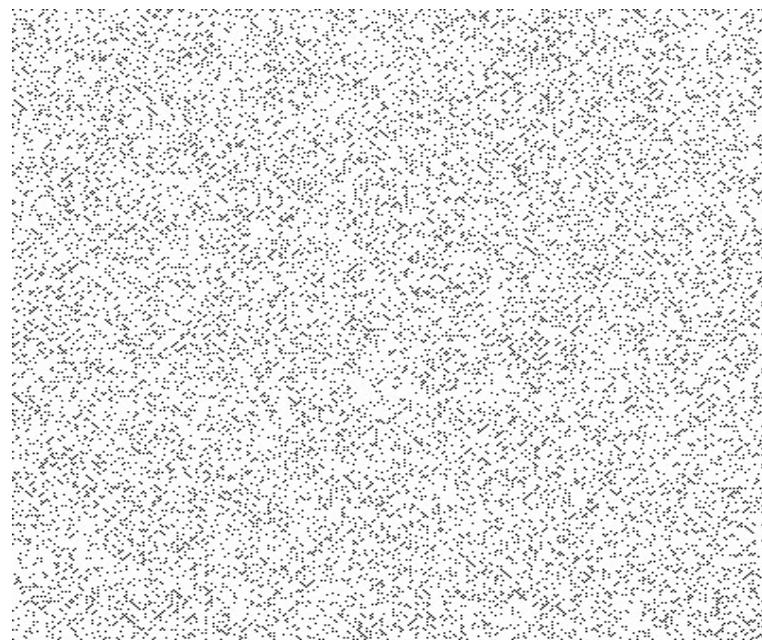


Problém mazání dat

- ◆ 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

- ◆ *mouchy, spočítejte se!*
- ◆ *která jste největší?*
- ◆ *kolik dohromady vážíte?*



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

- ◆ *mouchy, spočítejte se!*
 - ◆ podproblém: spočítejte průměr
 - uzel i: x_i = iniciální (libovolná) hodnota
 - epidemicky: $(x_i, x_k) = (x_i + x_k) / 2$
 - $x_i \rightarrow \text{avg}_{\forall n}(x_n)$
 - ◆ iniciátor $x_i = 1$, ostatní $x_i = 0$
 - $x_i \rightarrow 1 / N$
- ◆ *největší moucha*
 - $x_i = \text{vlastní váha}$
 - ep: $(x_i, x_k) = \max(x_i, x_k)$
 - $x_i \rightarrow \max_{\forall n}(x_n)$
 - ◆ *váha stáda*
 - průměrná moucha
 - velikost stáda

■ 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 ...

■ Živý výzkum v oblasti velmi rozsáhlých systémů

■ Nový hardware

- 😊 síť ≈ rychlá sběrnice, multicomputery ≈ multiprocesory
- 😊 rychlé optické kabely - přenos po síti rychlejší než na disk^{1.8²⁴ ≈ 1.3 mil ☺}
- 😊 velké disky - WORM, zapisovatelné, ...
- ☹️ velká paměť - mapované soubory, nestránkování
- ☺️ neblokové soubory - append v paměti rychlejší

Mooreův zákon:

$1.8^{24} \approx 1.3 \text{ mil}$ ☺

velikost disků:

1994: 100 MB

2018: 16 TB

■ Globální systémy

- ☺️ algoritmy - hierarchické / distribuované, masivní škálovatelnost a paralelizace
- 😊 lokalizace - znaková sada, ikony, národní prostředí
- 😊 potřeba datových dálnic
- 😊 problémy s prostorem jmen - strom příliš košatý
- ☺️ globální IS - VLDB, Cloud

■ Mobilní uživatelé

- 😊 konektivita v letadle
- 😊 ochrana dat a osobních údajů

■ ...



1. Úvod

- ◆ motivace, funkce, architektury
- ◆ komunikace, spolehlivost, RPC, skupinová komunikace

2. Synchronizační algoritmy

- ◆ fyzické a logické hodiny, vyloučení procesů, volba koordinátora
- ◆ kauzální závislost, doručovací protokoly, virtuální synchronie

3. Konsensus

- ◆ detekce globálního stavu, armády a generálové
- ◆ Paxos, RAFT, etc

4. Distribuovaná sdílená paměť

- ◆ konzistenční modely, distribuované stránkování

5. Správa prostředků a procesů

- ◆ zablokování, distribuované algoritmy detekce
- ◆ vzdálené spouštění procesů, vyvažování zátěže, migrace

6. Replikace a konzistence

- ◆ replikace, aktualizační protokoly
- ◆ klientocentrické konzistenční modely, epidemické protokoly



The End.



unused ...

Ocharna přístupu

■ Access Control Matrix

- user × resource → effective right

	Afile	Bfile	Cfile	Dfile
Anna	rwx	r	x	
Bill	r	rx	x	
Charles	r	r	rw	r
Damian		r		rw

■ Access Control List

- prostředek má seznam upravněných uživatelů

■ Capabilities

- certifikát
- uživatel drží oprávnění k prostředkům
- datová struktura umožňující jednoznačnou identifikaci objektu
 - obsahuje navíc přístupová práva pro držitele kapability
 - serializace, perzistence
- ochrana objektů, šifrování, redundance
- k jednomu objektu typicky několik různých kapabilit
 - vlastník, zapisovatel, čtenář, ... další druhy služeb
- uživatelským procesům znemožněno vlastní generování kapabilit i změny práv

■ Výhody

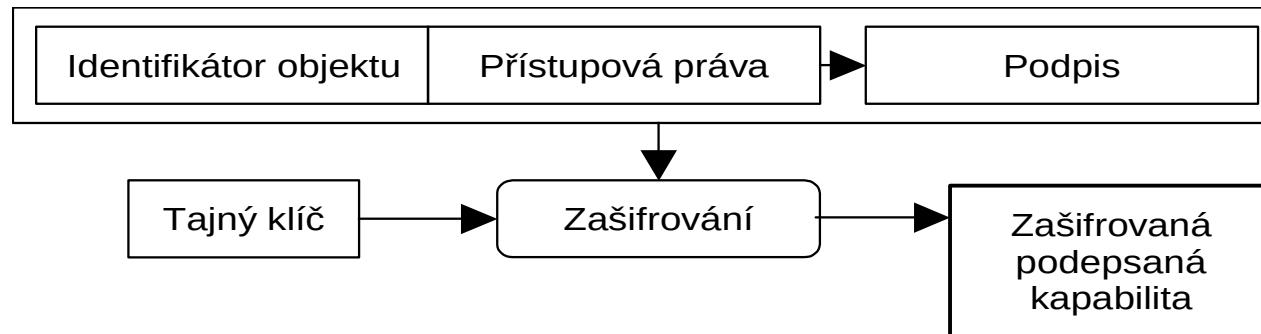
- ◆ snadný test oprávěnosti přístupu
 - škálovatelnost
- ◆ flexibilita
 - každý správce prostředků může na definovat vlastní druhy práv
- ◆ anonymita

■ Nevýhody

- ◆ kontrola propagace
 - copy bit, čítač - chráněný způsob přenosu
- ◆ review - seznam oprávněných uživatelů
- ◆ revocation - odejmutí práva
 - zrušení objektu a vytvoření nového, notifikace ostatních uživatelů
 - expirace

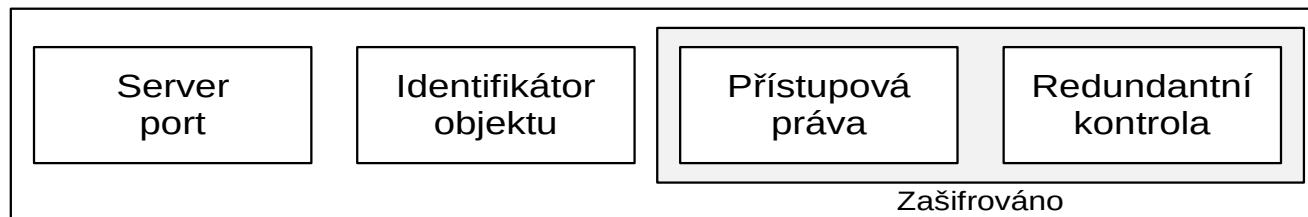
■ Kapabilita s podpisem

- ◆ 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



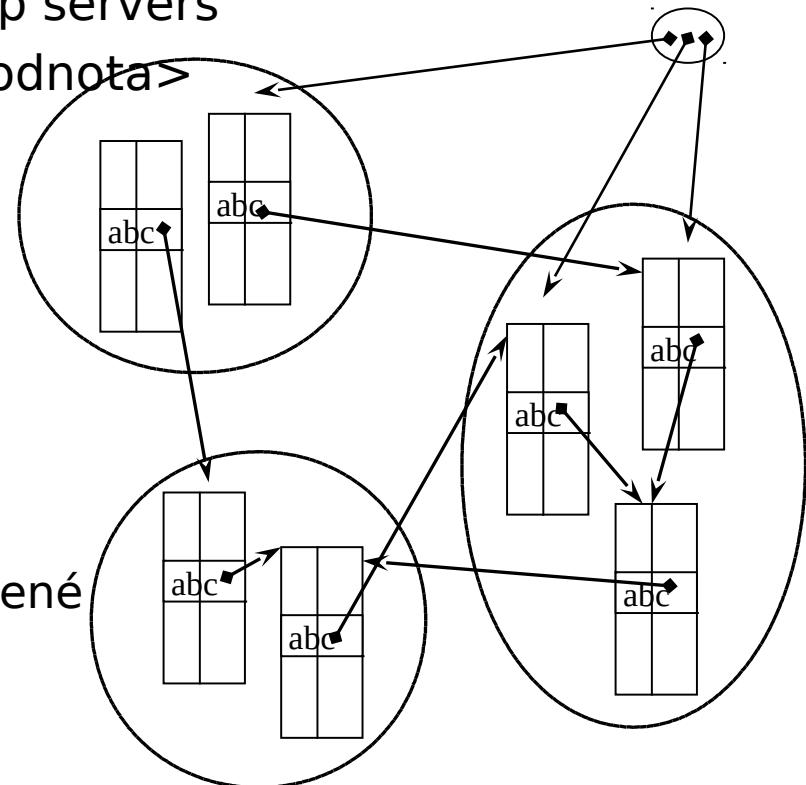
■ Kapabilita s redundantní kontrolou

- ◆ port serveru a identifikátor objektu jsou volně přístupné
- ◆ přístupová práva a náhodně vygenerované binární 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



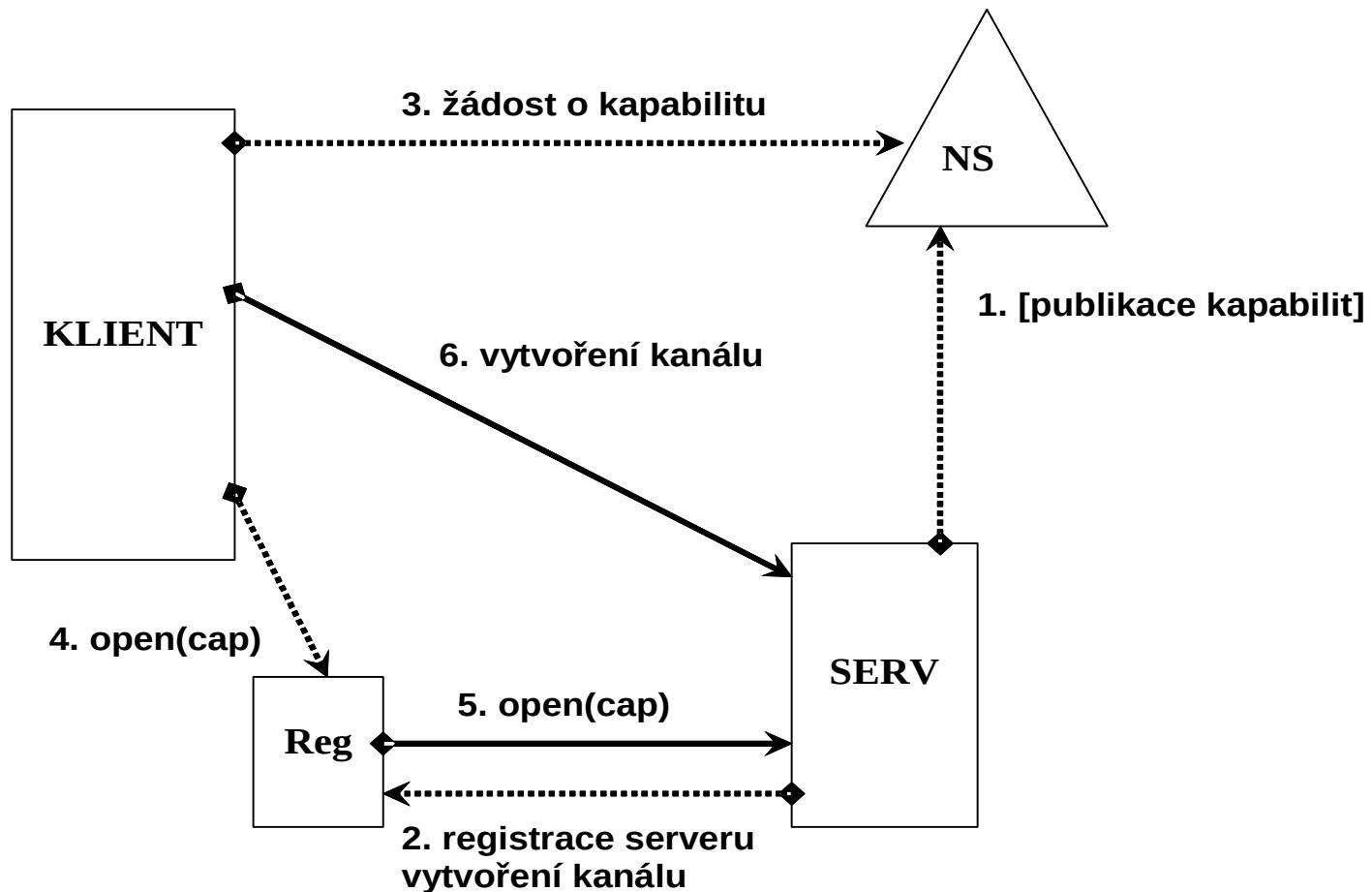
Distribuovaná správa jmen

- 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(jméno, 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

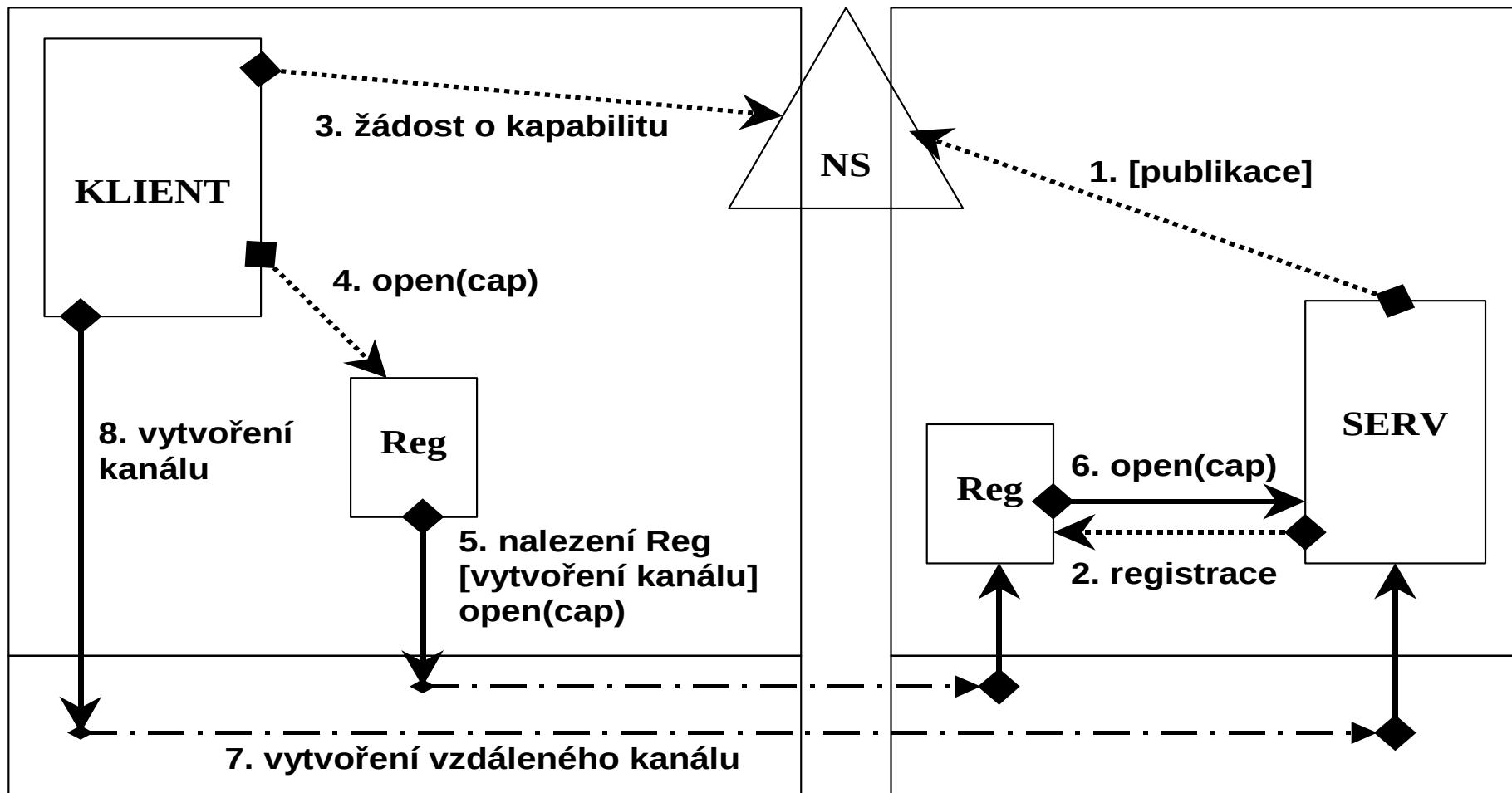
```
cap = ns -> resolve( "public/published_obj");
```

- otevření objektu - vytvoření tranzientní reference (portu)

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

■ Zprávy

- ◆ přesměrování - dočasné / trvalé
- ◆ migrace kanálů, spojení front



Klasifikace vyvažovacích algoritmů

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