

Principy distribuovaných systémů 2021/2022



Přednášející: Filip Zavoral
Sepsal: repetent za účelem složení zkoušky

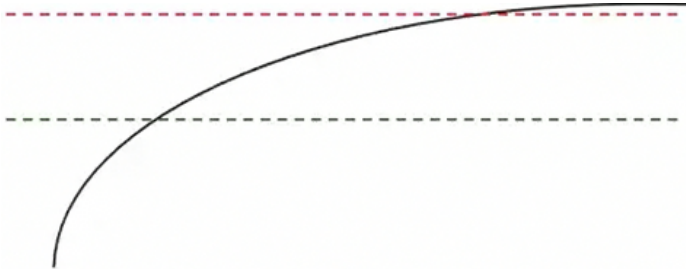
Obsah

Video 1	2
Video 2	8
Video 3	15
Video 4	19
Video 5	29
Video 6	32
Video 7	37
Video 8	47
Video 9	53
Video 10	57
Video 11	66
Video 12	71
Video 13	85
Video 14	91
Video 15	96

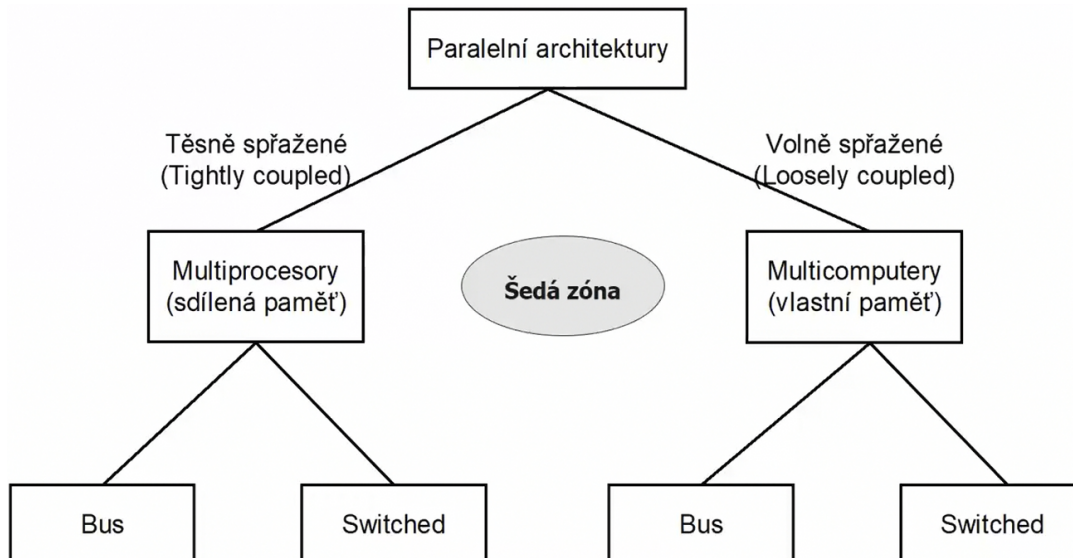
Video 1

- Distribuované algoritmy jsou základem jakéhokoliv cloudu
- Distribuovaný systém propojení množiny **nezávislých** uzlů, který poskytuje uživateli dojem **jednotného** systému
 - Uzly jsou nezávislé počítače s vlastním procesorem a pamětí
 - Komunikují pomocí síťového rozhraní
- Celá škála provedení distribuovaných systémů
 - Cloud (dnes moderní)
 - Množina počítačů v rámci racku
- 199x distribuované operační systémy
 - Amoeba
 - T4 (vznikl na MFF UK)
 - Podpora pro komunikaci přímo součástí jádra OS
 - Neexistoval rozdíl mezi lokální a distribuovanou aplikací
 - Přímé jádro OS komunikaci a synchronizaci řešilo samo bez aplikací
 - Neosvědčilo, složité komplexní problémy → nedotaženo do použitelného stavu
 - Skončilo to jako akademický proof-of-concept
 - Vývoj těchto OS skončil a šel jiným směrem
- 200x distribuované frameworky a aplikace
 - Distribuovanost se přesunula vně jader
 - Nad jádrem OS byla vrsta middleware, nad ním fungovaly distribuované aplikace
 - Lokální aplikace
 - Přímý přístup k nativnímu rozhraní OS
 - Distribuovaná aplikace
 - Přístup k middlewaru
- 201x cloud computing
 - Service-oriented computing (SaaS, PaaS, IaaS)
 - Spojení virtualizace + aplikace distribuovaných algoritmů + webové rozhraní
- HPC (High Performance Computing)
 - Velmi častá aplikace distribuovaných systémů
 - Výpočty, kde by jeden uzel nestačil nebo neměl dostatečnou výkonnost
 - Cluster Computing
 - Cluster = množina uzlů, které jsou jednotné z hlediska HW a OS
 - Propojeny standardní sítí
 - Často se používá sdílená paměť, RDMA
 - V jednom datovém centru nebo laborce
 - Těsné propojení sítí s nízkou latencí
 - Vysoká spolehlivost
 - Grid Computing
 - Rozvolnění vazeb mezi uzly
 - Širší geografické rozptýlení
 - Odlišný HW a SW
 - Žádné těsné vazby
 - Daleko větší latence

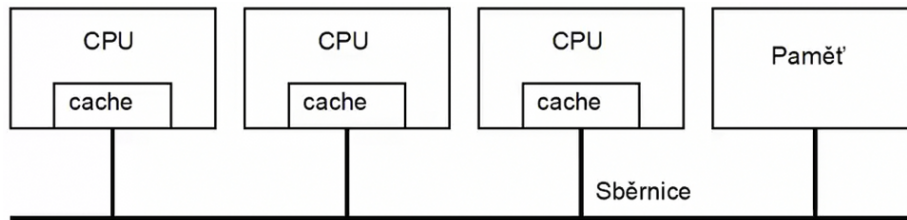
- Cloud Computing
 - Využívá oba dva koncepty (Cluster Computing i Grid Computing)
 - Propojuje je
- Rozsáhlejší informační systémy
 - Distribuovaný není výpočet ale data
 - Více info viz předmět [NDBI040](#)
- Pervasivní systémy
 - “Chytré domácnosti”
 - Velký počet rychle se měnících malých uzlů
 - Senzory, IoT
- Proč vlastně děláme distribuované systémy?
 - Když víme, že to je komplikované
 - **Peníze**
 - Síť běžných PC může nabídnout srovnatelný výkon se superpočítačem za mnohem menší peníze (souvisí s vertikální a horizontální škálovatelností viz předmět [NSWI150](#))
 - Rozšiřitelnost
 - Nejdřív nám náš počítač dostačoval, protože jsme si ho koupili s nějakou rezervou
 - Už nám ale nestačí
 - Vyhodit a koupit nový?
 - Finančně náročné
 - Vedle tohoto počítače si koupit další
 - Jak rozložit zátěž?
 - Distribuované systémy
 - Spolehlivost
 - Máme jeden počítač, ten vypadne, je konec
 - Když máme 100 uzlů a 2 vypadnou, tak se sice sníží výkon, ale přežijeme to
 - Výkon
 - Způsob, kterým můžeme porazit technologické limity
 - Je možné dosáhnout většího výkonu než nejvýkonnější technologický možný počítač
 - Distribuovanost
 - “Inherentní” distribuovanost problému
 - Některé problémy jsou sami od sebe svojí povahou distribuované
- Cíle návrhu
 - Transparentnost
 - Aby na vyšších vrstvách SW distribuovanost nebyla vidět
 - Přizpůsobivost
 - Každý uzel je schopný samostatné funkčnosti
 - Nespolehlivé sítě, HW, ...
 - Složitý problém, více si povíme v kapitole o distribuovaném konsensu

- Každý uzel vykonává rozhodnutí nezávisle na ostatních
 - Jakákoliv komunikace s ostatními uzly je o mnoho řádu pomalejší než jakákoliv operace v rámci uzlu
 - Otevřenost
 - Specifikovat rozhraní, ale implementace může být na každém uzlu jiná
 - Podpora různých OS napříč distribuovaným systémem
 - Migrace procesů a prostředků
 - Z uzlu na uzel
 - Transparentně
 - Hodí se, když potřebujeme odstavit nějaký uzel
 - Spolehlivost
 - Mnoha desetiletí palčivý problém
 - S počtem uzlů by spolehlivost měla růst
 - Dlouho spíše opačně
 - V poslední dekádě zlepšení spolehlivost díky Cloud Computingu a nasazení a implementaci algoritmů pro distribuovaný konsensus
 - Škálovatelnost
 - Je jiné vymyslet systém pro desítky nebo stovky uzlů oproti systému pro sto miliony uzlů
 - Některé algoritmy techniky, které jsou použitelné pro rozumný počet uzlů často nejsou aplikovatelné nad nějaký limit
 - **Vyhnout se čemukoliv centralizovanému**
 - Neexistují přesné globální hodiny
 - Výpadek jednoho uzlu nesmí způsobit nefunkčnost celého systému
 - Výkonnost
 - Teorie: více uzlů → vyšší výkon
 - V praxi výkon nestoupá lineárně
 - Výrazně vzrůstá komunikace a synchronizace mezi uzly
 - U spousty distribuovaných systémů můžeme narazit na hranici, kdy v rámci jeho technologií přidání dalšího uzlu už nepřinese žádný další výkon
- 
- Vyhnout se algoritmům, které vyžadují synchronizaci všech uzlů, pokud máme rozsáhlý distribuovaný systém
 - Chyby návrhu distribuovaných systémů
 - Síť je spolehlivá

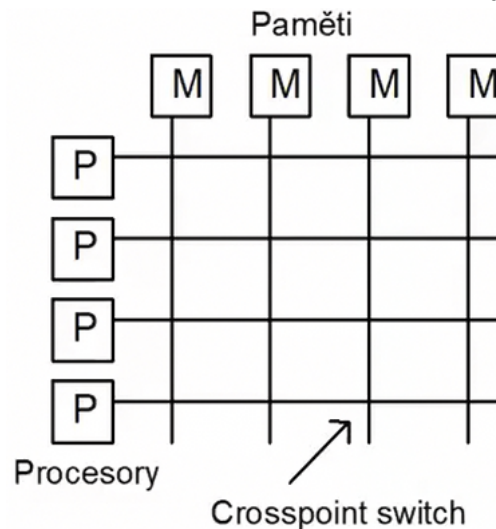
- Sít' je zabezpečena
- Nulová latence
- Topologie se nemění
- Jeden administrátor
- Neomezená kapacita sítě
- Paralelní architektury
 - Budeme dělit podle jejich těsnosti spřažení
 - Těsně spřažené
 - Mají nějakou sdílenou paměť
 - Říká se jim multiprocesory
 - Volně spřažené
 - Uzly mají vlastní paměť
 - Nemáme žádnou sdílenou paměť
 - Říká se jim multicomputery
 - Hrubé dělení, protože mezi jima existuje šedá zóna, kde nějaká sdílená paměť je
 - S kým je ale sdílena? (ne se všemi uzly)



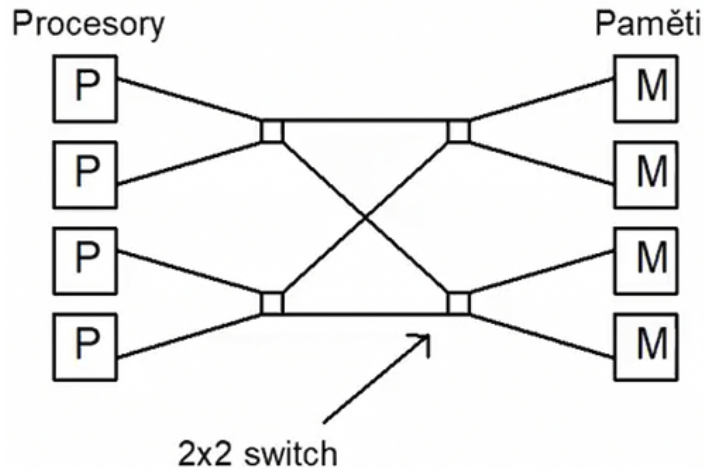
- Multiprocesory (bus – sběrníková architektura)
 - Nejběžnější z čeho můžeme vyrobit distribuovaný systém
 - Máme několik procesorů, které jsou sběrnicí připojeny k paměti
 - Když procesor potřebuje přístup, tak po sběrnici pošle požadavek a paměťový modul po ní vrátí data
 - Když máme více než jeden procesor, tak velkou roli v efektivitě hraje cache
 - Redukuje potřebu příliš častého přístupu ke sběrnici
 - V jednom čase totiž může na sběrnici být pouze jeden požadavek
 - **Vyniká velice malou škálovatelností**



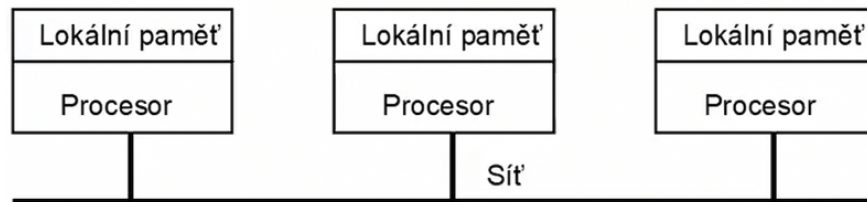
- Multiprocessory (switched – přepínačová architektura)
 - Procesory jsou umístěny nezávisle na sobě jako paměťové moduly
 - Od každého procesoru i paměťového modulu vede část sběrnice
 - Na všech křižovatkách jsou crosspoint switche, které umožňují přepnout na příslušnou odbočku
 - Pokud nedochází k častému promíchání toků (přepínání) tak je to o dost efektivnější než sběrniceová architektura
 - Hodně finančně náročné
 - Místo jedné sběrnice máme $CPU \cdot MEM$ sběrnic a kvadratický počet switchů
 - Není sériová výroba → dražší
 - Problémy v SW
 - Přicházíme o transparentnost (musíme mít na paměti, že vyvíjíme SW pro tuto architekturu se znalostí její konfigurace)



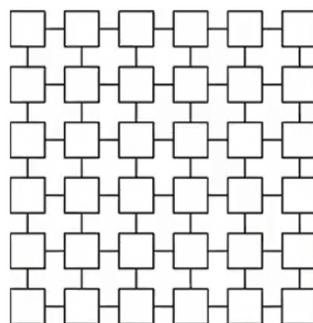
- Omega network
 - Místo kvadratického počtu crosspoint switchů existuje řešení používající $n \cdot \log n$ switchů
 - Je to levnější
 - Zvýšení latence při vyšším počtu úrovní



- Obecně specializované multiprocesorové architektury jsou dost drahé a nezvyšují míru škálovatelnosti výrazně, v praxi se nepoužívají
- Spíše se používají různé hybridní architektury, které umožňují přístup do paměti jiných uzlů → NUMA
- Multicomputery (bus – sběrníková architektura)
 - Základem typických distribuovaných systémů
 - Uzel má svůj vlastní procesor a lokální paměť, která je výhradně přístupná tomu procesoru
 - Uzly komunikují pomocí sítě
 - Levné a výrazně více škálovatelné
 - “Normální” počítače
 - Blady v DC

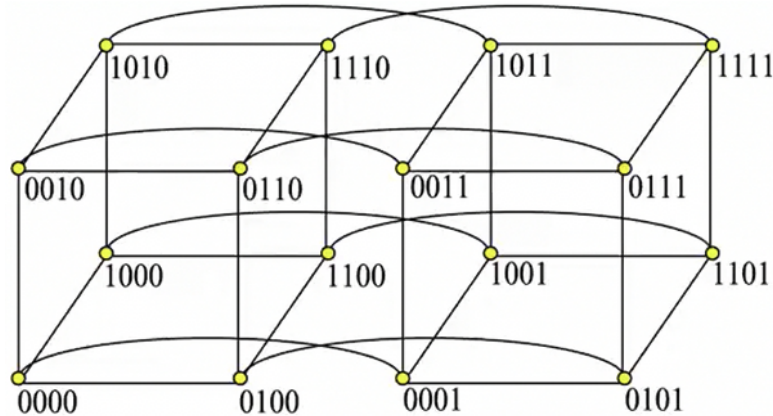


- Multicomputery (switched – přepínáčová architektura)
 - Mřížková architektura
 - Vhodné pro řešení dvourozměrných problémů
 - Grafy, analýza obrazu

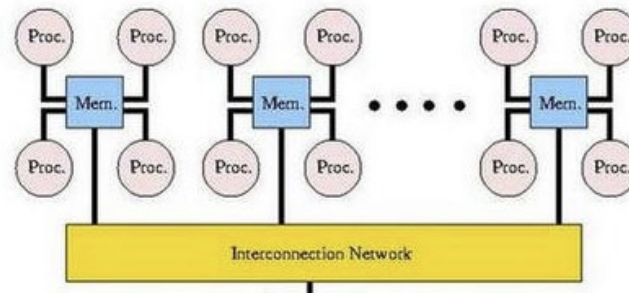


- Hyperkrychle

- n -rozměrná krychle (obvykle ale 4)
- Většina současných superpočítačů



- NUMA (Non Uniform Memory Access)
 - Vracíme se do šedé zóny (viz první obrázek Paralelní architektury)
 - K menší množině procesorů je připojen paměťový modul
 - Těchto bloků je několik
 - Všechny jsou propojeny sítí/sběrnicí
 - Nebudeme se o nich bavit → [NPRG042](#), [NSW1143](#)



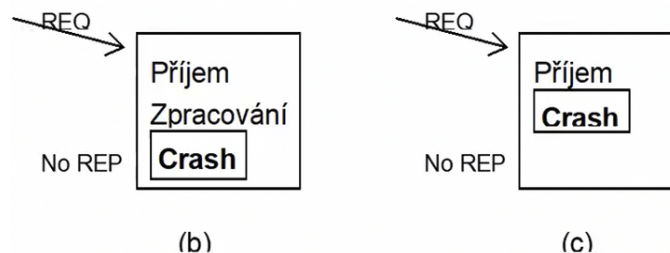
- GPGPU
 - Obecné počítání na grafických kartách
 - Dost odlišné od běžného paralelního počítání
 - CPU má "pár" jader, každé jádro můžeme separátně ovládat
 - GPU má tisíce jader, tisíce jader ovládáme jedním kódem
 - Programování není jednoduché a dost se odlišuje
 - [NPRG058](#)

Video 2

- Neexistence sdílené paměti
 - Předávání informací pomocí zasílání zpráv
 - Nízkoúrovňové primitivum
 - Dnes existují mechanismy vyšší úrovně (RPC)
- Klienti nerozlišují lokální a distribuovaný přístup
 - Komunikaci řeší middleware, jádro nebo něco jiného
 - Je nutné ale počítat s latencí, případně vzdálená komunikace může selhat

- TCP
 - Spolehlivý
 - Komplikovaný (velká režie komunikace)
 - V prostředí, kde nepotřebujeme řešit nespolehlivou komunikaci nevýhodné
 - V rámci racku, kde máme spolehlivou a rychlou síť mezi servery
 - Výpadek je velmi výjimečný
 - Obecně není vhodný
 - Nemá limit na max dobu doručení zprávy
 - Některé algoritmy vyžadují, aby existovala spolehlivá metoda na detekci toho, jestli zpráva přišla, nebo ne
- Specializované protokoly
 - Snaží se redukovat systémové a synchronizační zprávy
 - Výrazné navýšení rychlosti
 - Snížení latence
 - Často se používá piggybacking
 - Různé systémové a synchronizační zprávy jsou přilepeny k užitečným zprávám
 - (Normálně by tyto zprávy byly samostatné)
 - Příklad: TCP/FO (TCP Fast Open)
- Zajištění spolehlivosti komunikace
 - Přímou síťovým HW
 - Na SW úrovni
 - Duplicita nebo předbírání zpráv
 - Všechny zprávy mezi každým odesílatelem a příjemcem jsou postupně očíslovány
 - Když příjemce vidí zprávu se stejným číslem, tak ji zahodí
 - Když se zprávy předběhly tak je zahodí nebo bufferuje
 - Nepřijatá zpráva
 - Klient vyslal žádost a čeká ...
 - Mohla se ztratit zpráva se žádostí/odpovědí
 - Nebo se nic neztratilo a kanál je zahlcen a zpráva ještě nedorazila
 - Základní ošetření tohoto je (timeout, ACK, repeat)
 - Nastavím si timeout a pokud nedostanu do timeoutu ACK, tak pošlu zprávu znovu
 - Obvykle je vhodné na SW úrovni zajistit spolehlivost a škálovatelnost jednoduchostí nějakého řešení než vytvářením teoreticky dokonalým protokolem, který by mohl být příliš komplikovaný
- Přenos dlouhých zpráv
 - Doposud jsme považovali za jednotku přenosu zprávu
 - Když se zpráva pošle, tak potvrzení na celou zprávu
 - Když je zpráva příliš dlouhá (přeposlání celé DB, celého videa)
 - Rozdělíme zprávu na packety
 - Co ale potvrzovat?
 - Potvrzování každého packetu

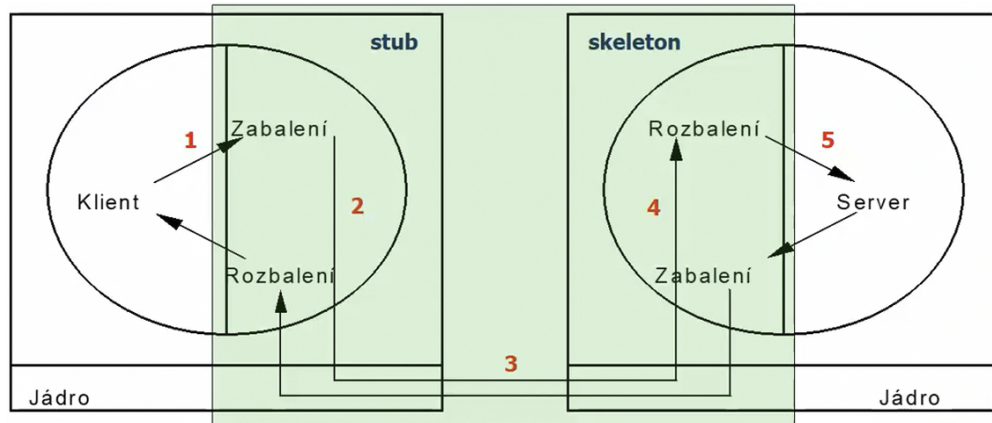
- Neúměrně vzroste režie
- Na spolehlivých lokálních sítích zbytečné
 - Režie při správném přenosu!
- Potvrzování celé zprávy
 - Režie při špatném přenosu!
- Dávky (blast, burst, buřty)
 - Budeme potvrzovat dávky (definovaný počet packetů)
 - Po úspěšného přeposlání dávky se pošle potvrzení
 - Při špatném přenosu se opakuje pouze dávka (záleží ale na protokolu – některé mohou být chytřejší a opakovat pouze část dávky)
 - Dostanu neočekávaný packet
 - timeout
 - Jak správně určit velikost dávky?
 - Pevné dávky
 - Jednoduché ale neadaptabilní
 - Když je síť pod větší zátěží může mít větší chybovost
 - Když je nezahlcena, tak může mít lepší spolehlivost
 - Dynamické dávky
 - Velikost dávky je upravována podle aktuální situace
 - n -krát správný přenos → zvětšení
 - Špatný přenos → zmenšení
 - O kolik zmenšovat a zvětšovat je implementačně závislé
 - Zmenšení obvykle výrazné (na polovinu)
 - Zvětšení obvykle postupně pomalé
 - Složitější implementovat, je potřeba jasný konsensus mezi oběma stranami o velikosti dávky
- Nespolehlivost serveru / služeb
 - Nespolehlivost nemusí být pouze na komunikačním kanálu, ale i na konkrétní uzlu
 - Klient odešle zprávu, čeká a nic se nestane
 - Komunikační problém nenastal
 - Problém je, že klient neví jestli havárie přišla před zpracováním zprávy nebo až po



- Obecně **nejde zjistit**, zda se operace **provedla**
 - I když lze, tak to pro některé služby je příliš náročné (transakce)
- Abychom se s tím mohli vypořádat, tak je důležitý koncept idempotentních služeb

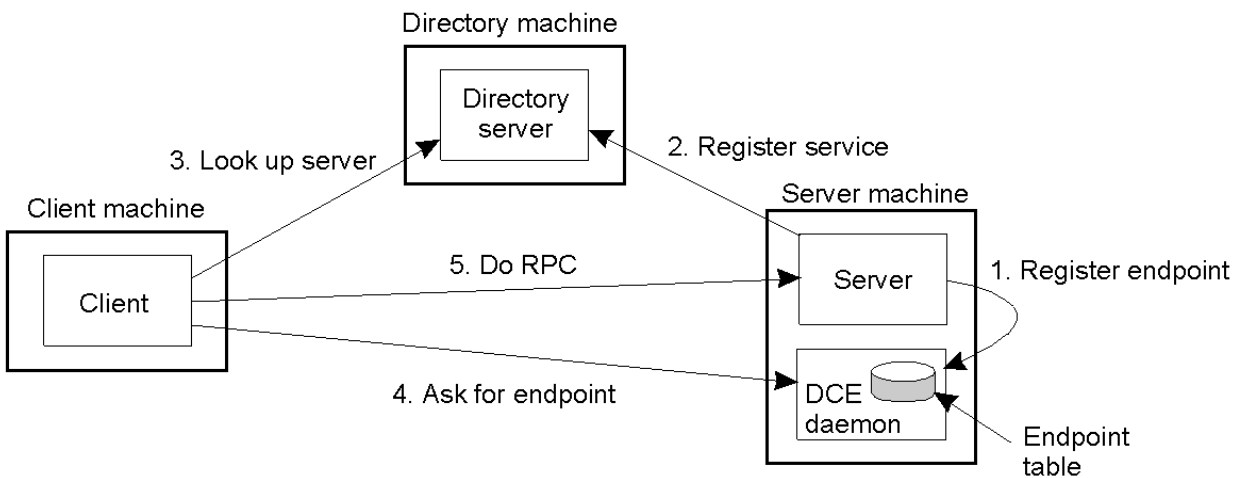
- Idempotentní služby
 - Nevadí opakované provedení služby
 - Sečti 1+1 👍
 - Ztratíme jenom trochu výpočetního výkonu, protože operaci server provede vícekrát
 - Vyber \$1 000 000 z mého účtu 🙄
 - (u mě je idempotentní, protože se mi z účtu stejně nic nevybere, když tam milion nemám 🤪)
 - Exactly once sémantika
 - Ideální
 - Každý požadavek se provede právě jednou
 - Pro některé služby to nejde
 - At-least-once sémantika
 - Služba se určitě alespoň jednou provede
 - Negarantujeme, že provede právě jednou
 - Nevadí, pokud jsou služby idempotentní služby
 - Při velké chybovosti ztrácíme maximálně výkon
 - At-most-once sémantika
 - Nutné, pokud nemáme k dispozici idempotentní služby
 - Služba se určitě neprovede vícekrát
 - Sem patří výběr z účtu
 - Nelze zajistit, že se provede
- Havárie klienta
 - Kromě serveru mohou havarovat i klienti
 - Klient odešle požadavek a umře
 - Server počítá a počítá a dělá výpočet pro mrtvého klienta
 - Typicky to, že umřel klient nějaké běžící služby se neřeší
 - Obvykle se nejedná o dlouhotrvající výpočty
 - Toto by mohl být problém v distribuovaných systémech, kde se provádějí dlouhotrvající výpočty
 - Kdyby nikdo neodebíral výpočet, tak by to byl problém (finanční)
 - Pokud to stojí za řešení v našem distribuovaném systému tak řešení existují
 - Exterminace
 - Zodpovědnost klienta
 - Klient má persistentní log
 - Když se zrovna narodí, tak prozkoumá svůj log, jestli náhodou neměl rozpracovaný job
 - Rozhodne se, jestli mu služba k něčemu bude nebo ne
 - Případně zruší službu
 - Reinkarnace
 - Založeno na pojmu epocha
 - Zodpovědnost klienta i serveru
 - Evidence (epocha, služba) je na straně serveru

- Klient má evidenci epochy a spolehlivou inkrementaci na starost
 - Jednoduché celé číslo
- Klient po svém narození inkrementuje svoji epochu
- S každým požadavkem klient zároveň posílá i svoji epochu
- Když server přijme žádost od klienta, kde je vyšší číslo epochy než je u služeb, které ještě zpracovává, tak zruší všechny služby, které spadají do předchozích epoch
- Expirace
 - Klient nemusí mít představu jak dlouho výpočet bude trvat, tak výpočtu přidělí tolik, kolik je ochoten utratit
 - Každá úloha má přidělené quantum času
 - Server počítá jenom pokud má quantum času
 - Když vyprší, tak server vrátí zprávu, že žije a že počítá, ale že to nestihl a vyžádá si pro další výpočet další quantum času
 - Klient si vyžádá pokračování výpočtu se specifikací nového quanta času
- Komunikace pomocí zpráv je z pohledu programátorského rozhraní komplikovaná
 - RPC (Remote Procedure Call)
 - Populární, užitečné a prakticky používané
 - Idea: Přizpůsobit mechanismus na volání funkcí/metod
 - Dvě části stub a skeleton
 - Stub u klienta
 - Skeleton u serveru
 - Oboje automaticky vygenerované části
 - Zajišťují komunikaci mezi klientem a serverem
 - Jak to probíhá
 - Klient zavolá běžně lokální funkci
 - Tím se zavolá klientský stub
 - Zabalí se parametry zprávy
 - Zpráva se pošle
 - Server přijme zprávu, zpráva se dostane do skeletonu
 - Skeleton zavolá výkonou funkci na serveru
 - Po zpracování server vrátí výsledek do skeletonu
 - Ze skeletonu se zpráva pošle
 - Klient přijme zprávu, zpráva se dostane do stubu
 - Stub ji rozbalí a vrátí ji klientovi



■ Spojení

- Jiný uzel uchovává tabulky serverů a služeb, které jsou poskytovány
 - Tzv. directory server
- Nově nastartovaný server vždy
 - Vytvoří endpoint do své tabulky
 - Zaregistruje si svoji službu na directory serveru
- Když klient potřebuje nějakou službu
 - Zeptá se directory serveru, kdo poskytuje službu, kterou potřebuje, directory server mu odpoví
 - Spojí se serverem, který poskytuje službu, a požádá ho o endpoint, server mu odpoví
 - (Potom může klient provádět RPC)



■ Rozhraní

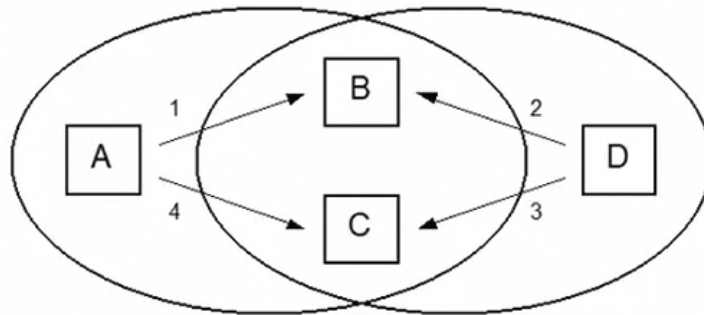
- Obvykle se definuje v IDL (Interface Description Language)
 - Jazyk nezávislý na programovacích jazycích sloužící k popisu rozhraní

■ Problémy

- Transparentnost
 - Stále musíme mít na paměti, že tato funkce není lokální

- Neexistuje sdílená paměť → žádné globální proměnné
 - Žádné předávání ukazatelů
 - (Možné řešení DSM – distribuovaná sdílená paměť, těžkopádné)
- Reprezentace dat
 - Problém pokud máme distribuovaný systém na bázi gridu
 - Big-endian vs Little-endian
- Komunikační chyby
 - Každý klientu by měl být připravený na to, že jakákoliv služba může být nedostupná
- Skupinová komunikace (jeden odesílatel, více příjemců)
 - Nepodporuje
- RPC systémy
 - Vznikali někde od roku '87 až do teď
 - Cisco RPC/XDR ('87)
 - Google FlatBuffers ('14)
- Skupinová komunikace
 - Jeden odesílatel, více příjemců
 - Atomicita
 - Doručení **všem** členům, nebo **nikomu**
 - Není vůbec triviální
 - Uzly vznikají a zanikají, mají havárie
 - Jak řešit → virtuální synchronie (Přednáška 7)
 - Synchronizace
 - Pořadí doručení je důležité zachovat podle pořadí odesílání
 - Co když jeden uzel pošle serveru $x += 2$ a druhý $x *= 2$
 - Na straně příjemce je nutné rozlišovat mezi příjmem zprávy a doručením zprávy
 - Příjem zprávy
 - Nějaká spodní komunikační vrstva fyzicky přijme zprávu a zařadí ji do bufferů
 - Doručení zprávy
 - Zpráva se z interních komunikačních bufferů dostane do SW, který na tuto zprávu čeká
 - Doručovací protokoly
 - Určují podmínky pro doručení přijatých zpráv
 - Definují sémantiku, jak komunikace funguje
 - Budeme se bavit o celé řadě
 - Kauzální sémantika
 - Sekvenční doručování
 - ...
 - Organizace skupin
 - Uzavřená skupina

- Všechny uzly, které se mají účastnit komunikace, jsou někde registrovány
- Vhodné pro kooperativní algoritmy
 - Množina uzlů spolupracuje na řešení distribuovaného výpočtu
- Mohou do ní odesílat zprávy pouze její členové
- Otevřená skupina
 - Definovaná příjemci
 - Zaslat zprávu může kdokoli (i někdo vně skupinu)
 - Vhodné pro distribuované služby a replikované servery
- Překrývající se skupiny
 - V rámci samostatných skupin je doručování zpráv synchronizované
 - Zprávy, které jsou zasílány uzlům, které jsou součástí i jiných skupin, tak mohou být nesynchronizovány

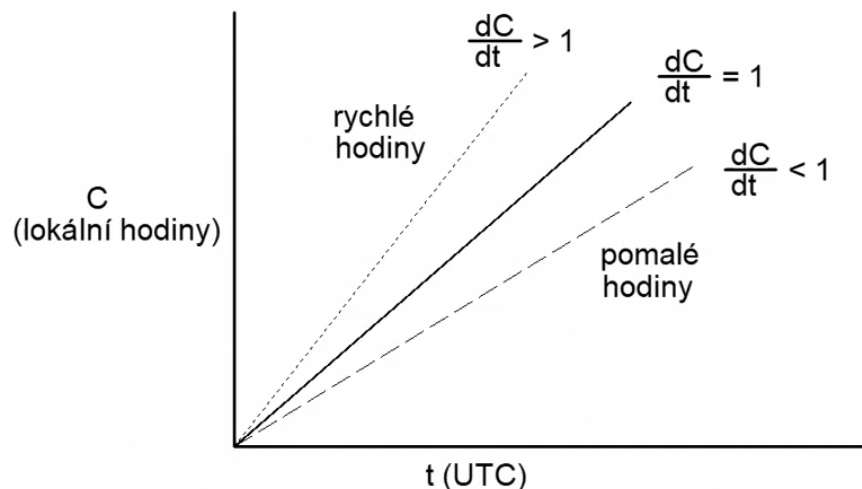


- Budeme si povídat o doručovacích protokolech pro skupinové komunikace mnohem podrobněji (Přednáška 6), nejprve ale potřebujeme probrat synchronizační algoritmy

Video 3

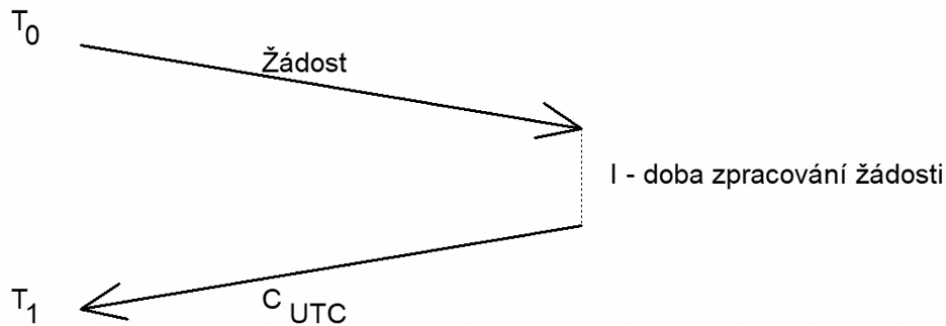
- Proč je synchronizace potřeba a proč je tak důležitá
 - Nemáme ani bit sdílené paměti
 - Veškerá komunikace pomocí zpráv
 - Neexistuje možnost mít na jednom místě všechny informace ohledně distribuovaném systému → rozprostřené informace mezi uzly
 - Vyloučení havarijních komponent
 - Umět se vypořádat s tím, že nějaký uzel odesílá i nesmysly
 - Neexistence společných hodin
- Synchronizace hodin s fyzickým časem
 - Jak sesynchronizovat lokální hodiny každého uzlu
 - Jak sesynchronizovat hodiny mezi sebou
- Fyzické hodiny (obecně)
 - Astronomické měření
 - Původně se čas měřil podle zdánlivého pohybu Slunce okolo Země
 - 1 solární sekunda = 1/86400 solárního dne

- Dlouho to stačilo (až do 2. sv. války)
- 1948 - atomové hodiny
 - $1s \approx 9$ mld přechodů atomu cesia 133
- 1950 – TAI (temps atomique international)
 - Atomové hodiny mohou mít nějakou drobnou nepřesnost
 - průměr z 50 (dnes 200) laboratoří
 - Každá laboratoř má svoje atomové hodiny
 - 1 sekunda je průměr z naměřených hodnot (s vyloučením extrémů)
 - 1 TAI den je cca o 3 ms kratší než solární den
- UTC (Universal Coordinated Time)
 - Když rozdíl napočítaného TAI času oproti pozorovatelnému astronomickému času založenému se odchýlí o více než 600ms, tak se v určitý čas vloží/odebere do TAI času přestupná sekunda
- Synchronizace fyzických hodin (co to znamená)
 - Každý uzel má hw hodiny C , fyzický čas t
 - Hodiny na počítači p v čase t je $C_p(t)$
 - Přesné hodiny (tikající podle TAI sekund): $C_p(t) = t \quad \forall t$
 - tedy $\frac{\partial C}{\partial t} = 1$ (derivative hodin podle času je 1)
 - Bohužel tak jednoduché to není, protože HW má v sobě nepřesnost danou výrobní technologií
 - Míra nepřesnosti ρ : $1 - \rho \leq \frac{\partial C}{\partial t} \leq 1 + \rho$
 - ρ je odchylka od standardního času



- Dvoje hodiny se za nějaký čas můžou rozsynchronizovat maximálně o $2\rho\Delta t$
 - My bychom chtěli, aby se nám žádné dvoje hodiny nerozsynchronizovali o maximální tolerovatelnou odchylku δ
 - Potom musíme zařídit, aby se nám všechny hodiny, které chceme seřizovat sesynchronizovávaly max v intervalu $\frac{\delta}{2\rho}$
- Ukážeme si několik algoritmů pro synchronizaci času

- Cristianův algoritmus
 - Když uzel potřebuje zjistit, jak moc se jeho hodiny odlišují od aktuálního času, tak pošle zprávu jednomu time serveru (považovaný za časovou autoritu – typicky napojen na UTC čas)
 - Server mu vrátí skutečný čas
 - Když se klientovi hodiny odlišují o více než nějakou maximální odchylku, tak si hodiny upraví
 - Je potřeba uvažovat dobu přenosu zpráv a dobu zpracování žádosti

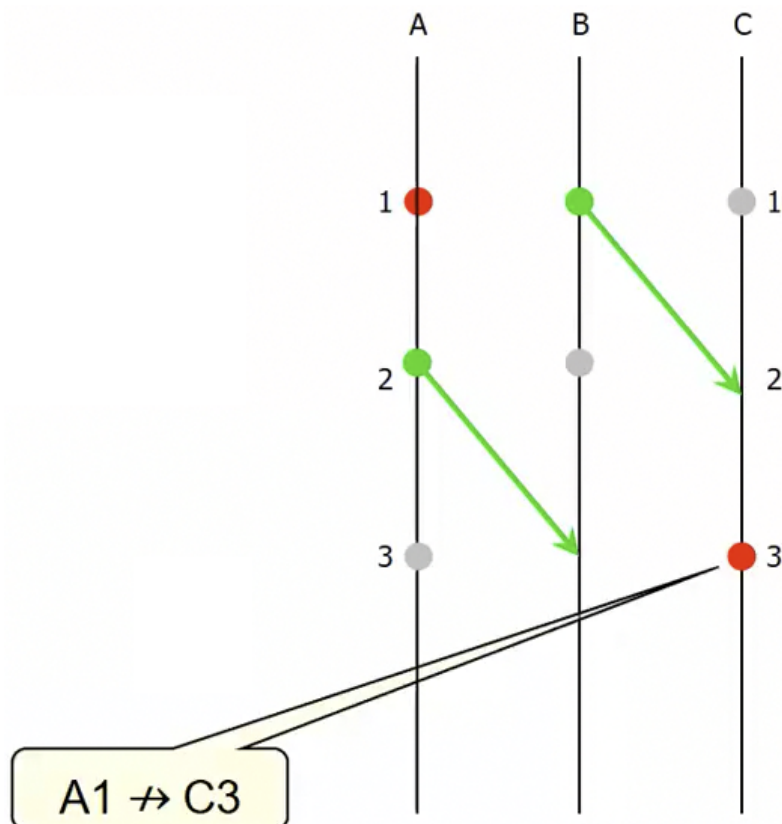


- $T = T_{UTC} + \frac{T_1 - T_0 - I}{2}$
 - **Nikdy nepřehřizovat najednou**
 - Zrychlovat nebo zpomalovat lokální čas
- Berkeley algoritmus
 - Synchronizace hodin vůči (aktivnímu) serveru, který ale nemá přesný zdroj času
 - V pravidelných intervalech se aktivní server zeptá všech ostatních uzlů kolik mají hodin (zároveň jim pošle svoje hodiny)
 - Zase se nějak uvažuje doba přenosu a zpracování zpráv
 - Server potom co dostane odpovědi tak zahodí extrém (nesmysly) a spočítá průměr
 - Spočítá pro jednotlivé uzly (včetně sebe) rozdíly o kolik by se měli upravit hodiny
 - Odešle zprávu se změnou jednotlivým uzlům, které si hodiny zas upraví (zpomalí nebo zrychlí)
- Intersection algoritmus
 - Zajímavý a často používaný koncept v distribuovaných systémech je, že čas se nechápe jako jeden přesný bod → je to interval
 - Čas od do (nebo střední hodnota a nějaká odchylka)
 - Upravená verze se používá pro NTP
 - Funguje čistě distribuovaně, žádný centrální server
 - Všechny uzly v pravidelných intervalech rozesílají informace o jejich času a jakou si evidují odchylku
 - Jakmile přijdou na uzel zprávy o čase, tak zahodí extrém a spočítá průměr mezi časy a vnitřní hodiny si přeřídí
- Použití intervalového času
 - Celá řada frameworků a knihoven
 - DCE (Distributed Computed Environment) – 33 knihovných funkcí
 - Google TrueTime (2012)

- Logické hodiny
 - Fyzické hodiny není možné dostatečně přesně sesynchronizovat
 - Důležité ale je **pořadí událostí**, nikoliv přesný čas
 - Nekomunikující procesy nemusí být sesynchronizovány
 - Na základě tohoto Leslie Lamport zdefinoval kauzální závislost
 - $e_1 \rightarrow^p e_2$
 - Uspořádání událostí v rámci procesu/uzlu p
 - Událost e_1 se udála dříve/před událostí e_2 na procesu/uzlu p
 - $send(m)$, $rcv(m)$ je odeslání/přijem zprávy m
 - 🧠 **Kauzální závislost** (značíme $\rightarrow$)
 - Jestliže $\exists p: e_1 \rightarrow^p e_2$ potom $e_1 \rightarrow e_2$
 - $\forall m: send(m) \rightarrow rcv(m)$
 - Jestliže $e_1 \rightarrow e_2 \wedge e_2 \rightarrow e_3$ potom $e_1 \rightarrow e_3$
 - Říkáme: kauzálně vázané, kauzálně závislé, kauzálně předchází
 - V distribuovaných systémech je typicky mnoho událostí, které nejsou kauzálně vázané, jsou tzv. konkurentní
 - Konkurentní události značíme: $e_1 \nrightarrow e_2$
 - **Def logických hodin**
 - Jednoduchý čítač
 - Musí platit následující
 - Událost a , čas $C(a)$
 - Jestliže $a \rightarrow b$ pak $C(a) < C(b)$
 - Hodiny musí být porovnatelné
- Synchronizace logických hodin
 - Synchronizace podle přijímání zpráv
 - Všechny události na každém uzlu mají přiřazenou časovou značku
 - V okamžiku, kdy komunikují, tak přilepí časovou značku odeslání zprávy
 - Pokud proces i vysílá v čase $C_i(a)$ zprávu m tak potom $T_m = C_i(a)$
 - T_m je značka zprávy
 - Synchronizace přichází v okamžiku příjmu zprávy
 - Pokud proces j přijme zprávu m v čase $C_j(b)$ tak potom

$$C_j = \max(C_j(b), T_m + 1)$$
 - Proces si upraví hodnotu svých logických hodin
 - $+ 1$ je technický detail
 - Tady narozdíl od seřizování fyzických hodin dochází k přenastavení času ihned
 - Na hodnotu logického času není nic jiného navázaného
 - **Vždycky pouze dopředu**
 - Zúplnění
 - Někdy se tomu říká “byrokratické uspořádání”

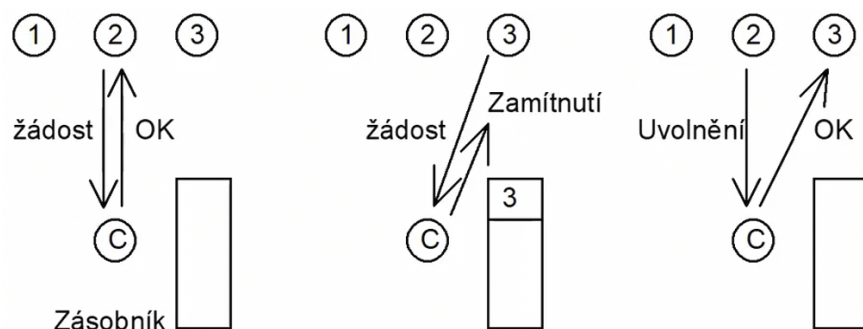
- Pro účely některých algoritmů (zejména synchronizačních algoritmů) je důležité, aby dvě události, které jsou odlišné, měly dvě různé hodnoty logických hodin
- Nesmí platit, že časové značky dvou různých událostí se mohou shodovat
- Událost a v procesu i , událost b v procesu j
- $C(a) = C(b) \wedge P_i < P_j \Rightarrow C'(a) < C'(b)$
- Logické hodiny a kauzalita
 - **Platí**
 - Jestliže $a \rightarrow b$ pak $C(a) < C(b)$
 - **Neplatí**
 - Jestliže $C(a) < C(b)$ pak by bylo hezké aby $a \rightarrow b$
 - Přitom tohle by se nám hodilo, protože potřebujeme detekovat kauzalitu, víme jenom hodnotu logických hodin



Video 4

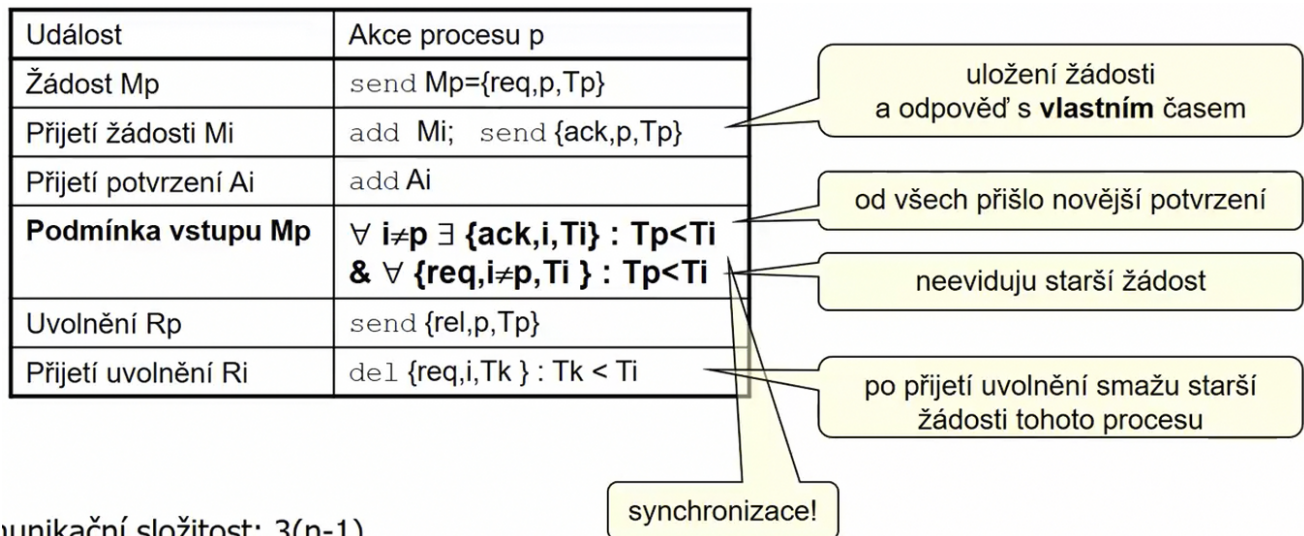
- Vzájemné vyloučení procesů = jakým způsobem zajistit exkluzivní přístup k něčemu
 - Nemáme paměť, takže žádný semafor
- Je třeba vzít v úvahu
 - Havárie, výpadky, prodlevy v komunikaci
- Složitý problém
- Způsoby řešení

- Centralizované řešení
 - Nějaký server řeší přístup k prostředku
- Permission-based
 - Distribuovaně každý uzel kontroluje podmínku
 - Když je podmínka splněna, tak potom algoritmus povolí vstup konkrétnímu procesu
 - Všechny uzly musí důsledně dodržovat onu podmínku
 - Algoritmy
 - Lamport ($3n$)
 - Ricart-Agrawala ($2n$)
 - 前川 ($\sqrt{n}$)
 - 1985年
 - Agrawal-ElAbadi ($\log n$)
- Token-based
 - Existuje něco jako token (pešek)
 - Jenom ten, kdo má token může vstoupit do kritické sekce
 - Algoritmy
 - Suzuki-Kasami
 - Raymond
- Centralizovaný algoritmus
 - Na každou kritickou sekci bude server, který přijímá žádosti od klientů a eviduje si je ve frontě a prvního pustí do kritické sekce
 - Když vystoupí klient z kritické sekce, tak tam pošle dalšího
 - Několik různých obdobných implementací
 - Ideově nevhodné → centrální komponenta
 - Single-point-of-failure
 - Špatně se škáluje
 - Zmíníme jeden z hromady problémů
 - Havárie klienta potom, co se klient dostal do kritické sekce
 - Možné řešit např. pomocí časových kvót a timeoutů



- Lamportův algoritmus
 - Čistě distribuovaný algoritmus, všechny uzly jsou si rovny
 - Proces když chce vstoupit do kritické sekce, tak pošle zprávu všem ostatním a čeká jednak až
 - dorazí odpovědi od všech ostatních

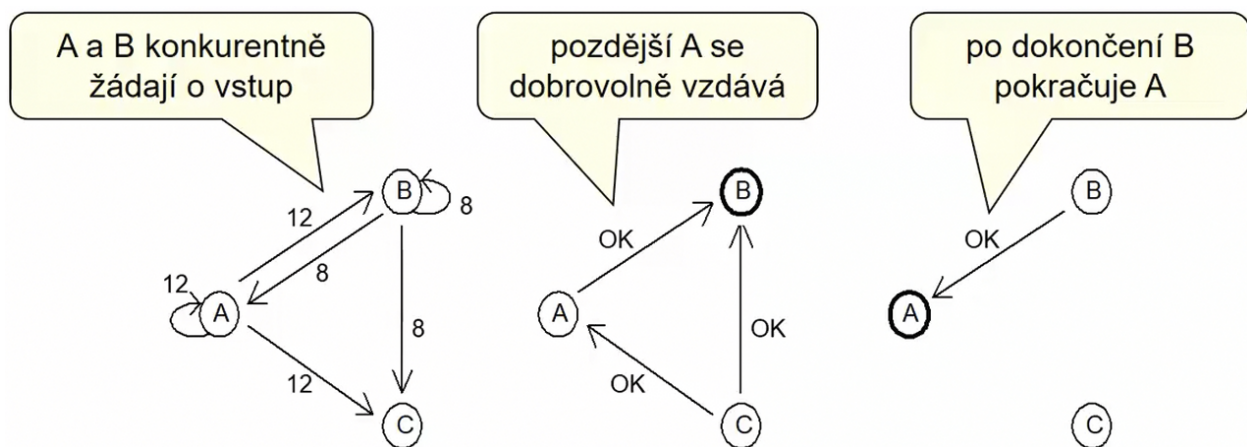
- a všechny žádosti v jeho frontě mají větší časovou značku
 - (každá zpráva má časovou značku a vstup do kritické sekce je řízen časovými značkami)
- Časové značky
 - Forma logických hodin, synchronizují se při příjmu zprávy
 - Když přijde zpráva s vyšší časovou značkou než jakou mám, tak přijímající proces si nastaví své lokální logické hodiny na alespoň takovou hodnotu jako je časová značka přijaté zprávy
- Existují tři typy zpráv
 - *release*
 - Uvolnění kritické sekce
 - *request*
 - Požadavek na vstup do kritické sekce
 - *ack*
 - Potvrzení pro vstup do kritické sekce
- Existují tři akce se zprávami
 - *send*
 - Odeslání zprávy
 - *add*
 - Přidání zprávy do své prioritní fronty (prioritní dle značek)
 - *del*
 - Smazání zprávy z mé fronty
- Jak to funguje



komunikační složitost: $3(n-1)$

- $n - 1$ je procesů, ze kterými každý žadatel komunikuje
- 3 jsou zprávy
 - Zpráva se žádostí
 - Zpráva zpátky s potvrzením
 - Zpráva o uvolnění kritické sekce

- $3(n - 1)$ je hodně, tak byly publikovány různé algoritmy, které komunikační složitost optimalizují
- Ricart & Agrawala
 - Hlavní rozdíl je ve významu potvrzení
 - U Lamporta potvrzení znamená – jo vidím tvoji zprávu, potvrzuji, že jsem dostal zprávu se žádostí
 - Tady znamená – ano souhlasím s tím, aby si vstoupil do kritické sekce
 - Zprávy s žádostí i potvrzením existují také
 - Jak to funguje
 - Proces chce vstoupit do kritické sekce
 - Zašle žádost s časovou značkou a čeká na odpovědi s potvrzením
 - Proces přijme zprávu ze žádostí
 - Není v kritické sekci a ani do ní nechce vstoupit → pošle potvrzení
 - Je v kritické sekci → neodešle potvrzení, požadavek zařadí do své fronty
 - Není v kritické sekci, chce ale vstoupit
 - Porovná časovou značku s vlastní žádostí
 - Vlastní žádost má menší časovou značku (je starší) → neodešle potvrzení a zařadí odesílatele do fronty
 - Žádost odesílatele má menší časovou značku (vlastní je mladší) → pošle potvrzení

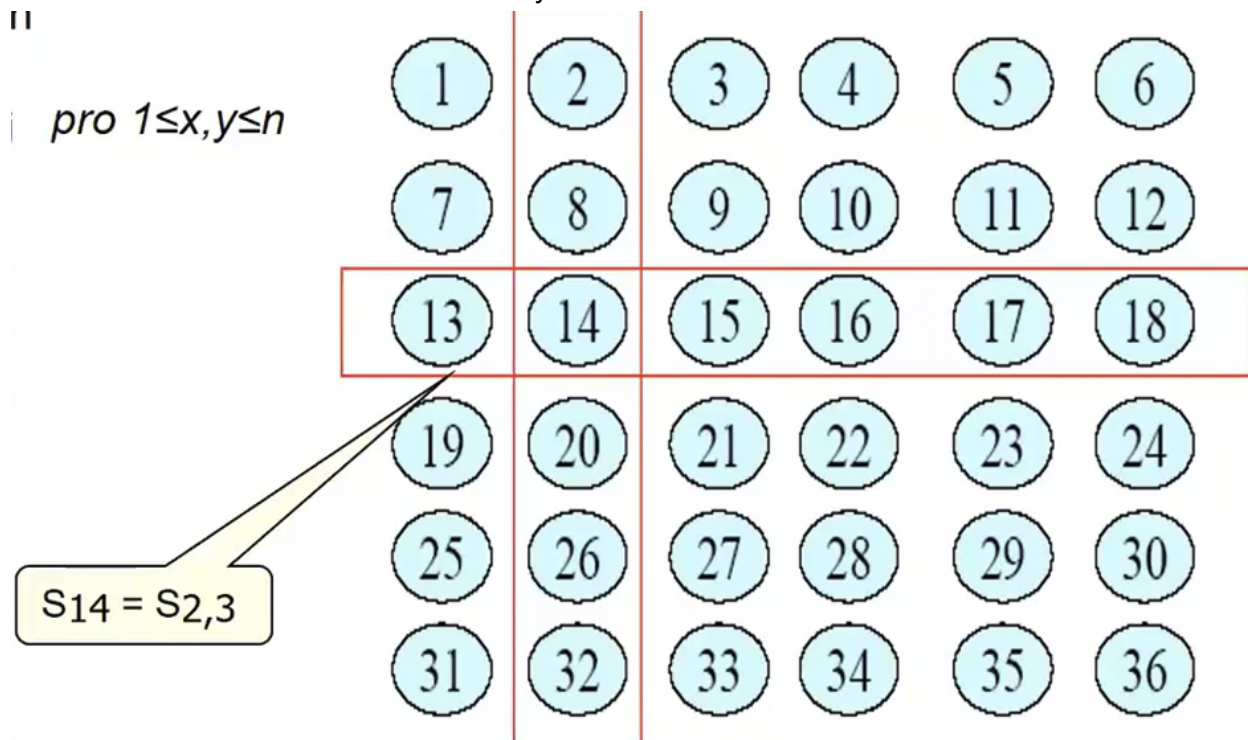


- Princip voleb (stále vyloučení procesů)
 - Základní princip (stejný pro všechny algoritmy)
 - Každý proces má jeden hlas
 - Může ho dát sobě (když touží po vstupu do kritické sekce)
 - Nebo ho dá jinému žadateli
 - Procesy se snaží získat hlasy ostatních procesů
 - Když chci vstoupit do kritické sekce, tak si požádám o hlasy
 - Když proces dostane více hlasů než ostatní, tak můžu vstoupit do kritické sekce, jinak čekám tak dlouho než dostanu dostatečný počet
 - Jednotlivé algoritmy se liší v tom

- Jak se pozná zda nějaký proces dostal více hlasů než jiný
 - Jak hlasovat a počítat výsledky
 - Kdy proces ví, že vyhrál
- Naivní algoritmus
 - Při žádosti jiného proces dá proces hlas (pokud ještě nehlasoval)
 - Odolný proti výpadkům
 - Vydrží výpadek až poloviny procesů
 - Komunikační složitost $O(n)$
 - Není jinak lepší než algoritmy, které jsou založené na porovnání značek
 - Rychle může dojít k deadlocku
 - Více procesů může dostat stejný počet hlasů
- 前川 (1985年)
 - Každému procesu p je přidělený volební okrsek S_p
 - S_p je nějaká dobře definovaná množina ostatních procesů
 - V okamžiku, kdy nějaký proces dostane všechny hlasy vlastního okrsku, tak může vstoupit do kritické sekce
 - Podmínky pro volební okrsky
 - Korektnost
 - $\forall p, q \text{ procesy} : S_p \cap S_q \neq \emptyset$
 - Každá dvojice kandidátu musí mít alespoň jednoho společného voliče
 - Při remíze, alespoň jeden společný volič rozhodne $\rightarrow$ každý proces může volit pouze jednou
 - Spravedlnost
 - $\forall p, q \text{ procesy} : |S_p| = |S_q| = K$
 - Velikost volebních okrsků je stejná
 - Zodpovědnost
 - $\forall p, q \text{ procesy} : \#S_i = \#S_j = D$, kde $p \in S_i$ a $q \in S_j$
 - p je obsažen ve stejném počtu D volebních okrsků
 - Každý proces má stejnou zodpovědnost za to jakým způsobem jednotlivé procesy vstupují do kritické sekce
 - Důsledek (podmínek)
 - Komunikační složitost algoritmu, který respektuje podmínky je $O(|S_p|)$
 - Proces nepotřebuje komunikovat se všemi ostatními procesy, pouze s vlastním volebním okrskem, kde sbírá hlasy
 - Potřebujeme minimalizovat volební okrsky, abychom minimalizovali komunikační složitost
 - Značení
 - N ... počet okrsků
 - K ... velikost okrsků
 - M ... počet procesů

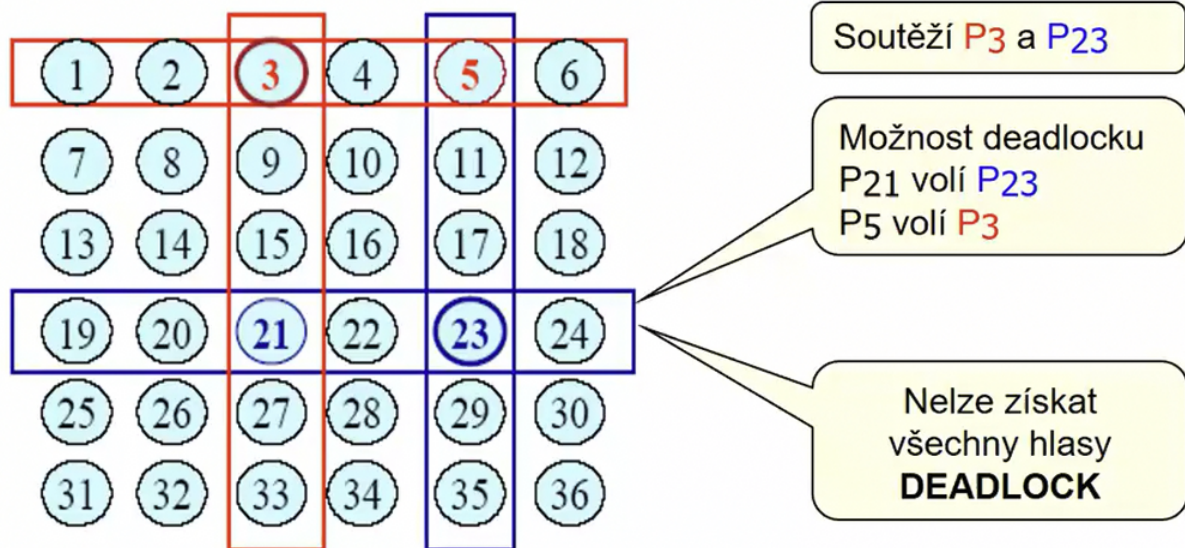
- D ... v kolika okrscích je každý proces členem
- Rozdělení okrsků
 - Jak velké okrsky?
 - V kolika okrscích má být každý proces?
- $N \cdot K = D \cdot M$
- Proces $\sim$ okrsek
 - Každý proces musí získat všechny hlasy vlastního okrsku
 - $N = M \Rightarrow K = D$
 - **Velikost** okrsku $\sim$ **počet** okrsků, ve kterých je proces členem
- Každý proces p z okrsku S_q je obsažen v $D - 1$ jiných okrscích
 - Tím pádem max počet okrsků je $(D - 1)K + 1$
- $M = K(K - 1) + 1 \Rightarrow K = O(\sqrt{M})$
 - Komunikační složitost
- Jak ale skládat okrsky?
 - Optimální algoritmus, který by přesně využíval $M = K(K - 1) + 1$ je složitý
 - Vyžaduje restrukturalizace při změně počtu členství
 - Existuje suboptimální algoritmus pro $K = O(\sqrt{M})$
 - Prakticky použitelný
 - Jednoduchý

II



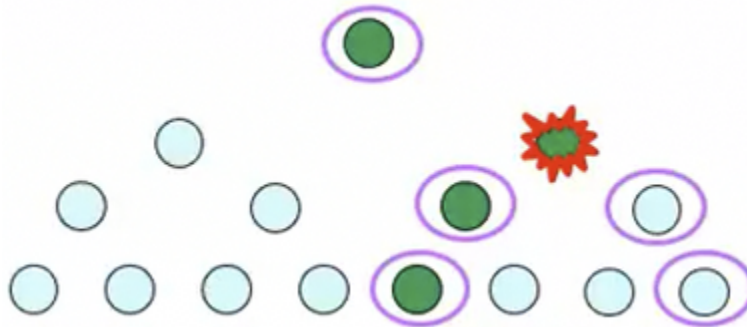
- Všechny procesy uspořádáme do "čtverce"
- Volební okrsek každého procesu je sjednocení vertikály a horizontály
- Na obr. je zvýrazněný volební okrsek procesu 14

Maekawa - deadlock a jeho řešení



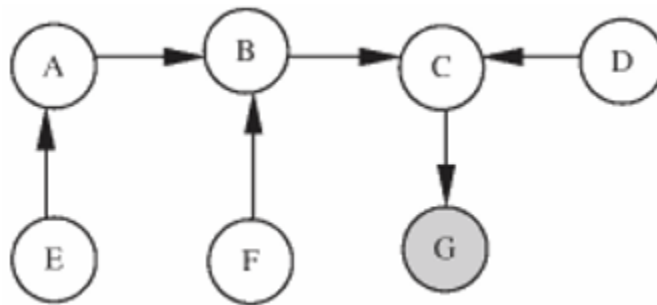
- Může se stát, že P_5 bude volit P_3 a P_{21} bude volit P_{23}
- P_3 i P_{23} oba dostanou $N - 1$ potřebných hlasů a čekají v deadlocku
- Zachrání nás logické hodiny – prevence deadlocku
 - Každá žádost je označována logickými hodinami jejího odesílatele
 - V okamžiku, kdy přijde zpráva se žádostí procesu r s TS_r (časovou značkou procesu r)
 - Má volný hlas
 - Odešle potvrzení *ACK* procesu p
 - Dal již hlas jinému procesu q
 - Pokud $TS_q < TS_r$ tak zařadí do fronty
 - Pokud $TS_q > TS_r$ tak pošle zprávu *REJECT* procesu q
 - Pokud již q je v kritické sekci, tak ho ignoruje a pracuje dál a až opustí kritickou sekci tak zpátky dá potvrzení a funguje to jako v předchozím případě
 - Pokud q ještě nemá všechny hlasy, vrátí hlas procesu p a ten ho předá procesu r
 - **TOHLE JE REGULERNÍ RACE CONDITION**
 - Jak race condition vyřešil 前川
 - Je to race condition, protože záleží jestli se proces stihne nebo nestihne vrátit do kritické sekce
 - Nevadí nám to, cílem algoritmu není přesný deterministický postup, v kterém procesy mají v jakém pořadí jít do kritické sekce

- Jediným cílem porovnávání časových značek je, aby nedošlo k deadlocku
- V okamžiku, kdy je nějaký proces (byť neoprávněně) je zvolený a je schopný se dostat do kritické sekce, tak potom už nemůžeme být v deadlocku a to nám stačí
- Agrawal & El Abbadi (1991)
 - Lepší komunikační složitost než 前川
 - Složitost $O(\log n)$
 - Volební okrsky jsou uspořádány podél stromu
 - Procesy jsou organizovány binárním stromem
 - Každý proces potřebuje získat cestu od kořene k listu
 - Potřebuje komunikovat s logaritmickým počtem uzlů
 - Speciálně navržený pro dosažení fault tolerantnosti
 - V případě kdy vypadne jakýkoliv uzel, tak hlas jednoho uzlu je možné nahradit dvěma cestami

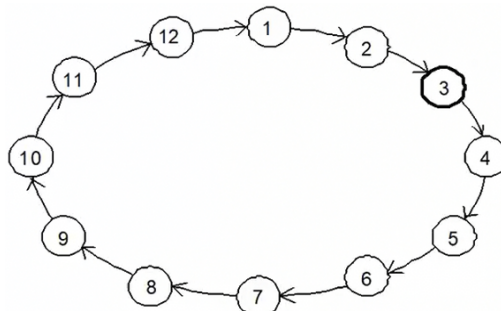


- Quorum = cesta od kořene k listu
- Nikdy nenašel praktické uplatnění
 - Nesplňuje podmínku symetrie
 - Jak vypadají quora?
 - Kořen je obsažen ve všech volebních okrcích
 - Účastní se všech voleb, je maximálně přetížen
- Existují různé vylepšení tohoto algoritmu
 - Strom je postupně otáčí...
 - Spíš obskurní
- Raymond (1989)
 - Všechny předchozí algoritmy, které jsme rozebírali byly permission-based
 - Založené na podmínce, kdy uzel zkoumá podmínku a když je splněna, tak může vstoupit do kritické sekce
 - Tento algoritmus je token-based
 - Založen na předávání tokenu
 - V celém systému je jedna speciální entita, která se přesouvá mezi uzly
 - Pouze ten uzel, který má tuto entitu (token), tak může vstoupit do kritické sekce
 - Všechny procesy jsou organizovány do orientovaného stromu
 - Uzel, který má token je v kořenu, všechny orientované hrany vedou k tomuto kořenu

- Když jakýkoliv proces potřebuje zažádat o vstup do kritické sekce, tak žádost přepośle podél orientovaných hran blíže k uzlu ke kořeni, až zpráva dojde do kořene
- Zároveň uzel s tokenem (kořen) udržuje frontu žádostí na vstup do kritické sekce
- Když uzel s tokenem vystoupí z kritické sekce, tak se podívá na frontu a prvním ve frontě pošle token a celou frontu
 - Nepošle mu to přímo, ale proti směru orientovaných hran sousednímu vrcholu
 - V každém kroku se obrátí orientace hrany
 - Takhle se token s frontou budou přeposílat a otáčet hrany než se dostane ke správnému vrcholu, z kterého se udělá nový kořen



- Suzuki–Kasami
 - (Token-based algoritmy byla publikována celá řada)
 - Tohle je jeden z prvních publikovaných
 - Vyžaduje broadcast zprávu všem uzlům
 - Nepoužitelné pro rozsáhlé distribuované systémy
 - Hodí se spíše do clusterového prostředí
 - Broadcast se posílá, když proces vyžaduje vstup do kritické sekce
 - Proces, který drží token, broadcast zpracuje (vloží si žádost do fronty), ostatní ho ignorují
 - Až proces bude chtít opustit kritickou sekci, tak přepośle token s frontou tomu procesu, který je na prvním místě
 - Umožňuje prioritně řadit do fronty (s každým požadavkem může být evidována cena)
- Token ring
 - Jiný typ token-based algoritmy



- Logický kruh, který nevyžaduje broadcast pro funkčnost

- Komunikace pouze mezi dvojicí uzlů
- Token postupně putuje mezi procesy po kruhu
- Když proces dostane token tak se podívá, jestli chce vstoupit do kritické sekce
 - Pokud ano, tak si token nechá
 - Pokud ne, tak ho pošle dál
- Dost vytěžuje síť i když nikdo nechce vstoupit do kritické sekce (speciálně když chráníme spoustu kritických sekcí)
 - Dokola budou lítat prázdné tokeny

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

- Zprávy
 - Nejlépe na tom je centralizovaný algoritmus
- Havárie
 - U centralizovaného velký problém havárie koordinátora
 - Agrawal & El Abbadi
 - (ten stromový algoritmus)
 - Jediný navržený pro dosažení funkčnosti při haváriích
 - Zvětšuje se ale quorum
 - (jak víme není ale symetrický)
 - Havárie uzlu u všechny ostatních výjma Raymond algoritmu způsobí totální nefunkčnost
- Chtěli jsme nevyhovující centralizovaný algoritmus nahradit něčím distribuovaným
- Výsledek je, že všechny distribuované algoritmy jsou horší než centralizované řešení
- Jak je možné, že něco distribuovaného je horší než centralizované řešení, když jsme si říkali, že právě centralizované řešení jsou původem veškerého zla

Moudro:

Obvykle nemá smysl řešit centralizovaný problém distribuovaným algoritmem

Nevhodný není centralizovaný algoritmus, ale centralizovaný problém!

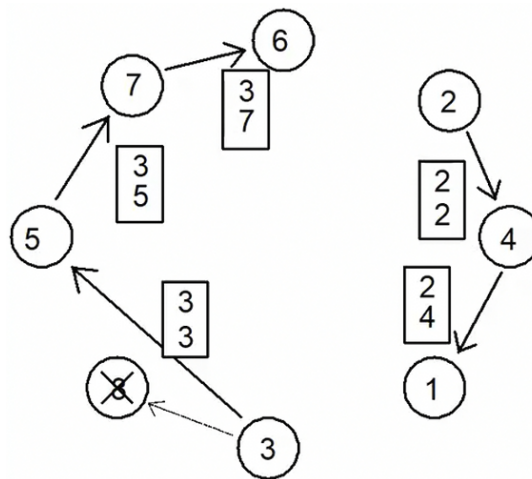
- Problém není v implementaci, ale v problému samotném
- V distribuovaném systému bychom neměli mít nic takového jako kritická sekce
- Kritická sekce je příkladem centralizované služby
- **V distribuovaném algoritmech/systémech vyhnout se potřebě centralizované komponenty i v podobě kritické sekce**
- Kritická sekce výrazně snižuje škálovatelnost řešení
- Není vhodné řešit centralizovaný problém distribuovaným algoritmem → centralizovanost zůstane, jenom se přesune

Video 5

- Budeme se zabývat algoritmy pro výběr koordinátora (leader election)
- Velice častý a běžný problém
- Koordinátor / leader je důležitý pro celou řadu distribuovaných algoritmů
 - Distribuovaný konsensus, load balancing
- Implementace algoritmu pro výběr koordinátora je typickou součástí cloudových frameworků
- Důležitá korektnost → nemůže být více než **jeden** leader
- Dva přístupy k řešení tohoto problému
 - Detekce extrému
 - Uzly jsou víceméně rovnoprávné, ale nějak rozlišitelné být musí → obvykle jsou očíslovány
 - Algoritmy se snaží v rámci identifikátorů najít takový, který má nejmenší/největší identifikaci a zároveň není havarovaný
 - Race-condition (+ randomizace)
 - RAFT
 - Založeno na náhodě
 - Zjednodušeně: Uzel, kterému se něco povede stihnout dříve, tak se stává koordinátorem
- Bully algoritmus
 - Má netriviální předpoklady
 - Nejdelší možná doba přenosu zprávy je omezena konstantou

- To samé pro dobu zpracování a přípravu odpovědi
 - Říkáme, že předpokládá synchronní systém
- Uzel pošle zprávu se žádostí a očekává odpověď
- Když odpověď nedojde do doby 2x čas přeposílání zprávy + čas na její zpracování, tak ví, že je něco špatně
 - Nefunguje komunikace
 - Uzel havaroval
- Nejde používat např. na síti, která využívá TCP/IP protokol
- Použitelné v rychlých a spolehlivých sítích v clusterovém prostředí s použitím vlastního lokálního protokolu pro komunikaci
- Když se proces rozhodne volit koordinátora
 - Typicky se děje na začátku výpočtu, nebo když komunikace se stávajícím koordinátorem selže
 - Proces zašle zprávu všem procesům s vyšší identifikací a čeká na odpověď
 - Když od nich přijde odpověď tak proces skončí (prohrál)
 - Když nepřijde nic, proces vyhrál a stává se novým koordinátorem, pošle zprávu všem ostatním
- Když proces přijme zprávu o volbě
 - Vráť zpět odpověď a vyšle tu samou žádost všem procesům s vyšším číslem identifikace
 - Toto druhé kolo rozhodne o vítězi
 - Druhé kolo rozhodne, rozdíl mezi prvním a druhým kolem volby je v tom, kdy se o volbě dozví nový koordinátor (ten co se ním stane)
 - V prvním běhu nějaký uzel zjistí, že je zapotřebí nová volba
 - Obeznámí všechny kandidáty
 - Pokud dostane odpověď, tak ví, že někdo vhodnější je naživu, už se o to nezajímá
 - V druhém kole už budoucí výherce už o volbě ví (z prvního kola). V druhém kole nedostane žádnou odpověď
- Invitation algorithm
 - Předpokládá asynchronní systém
 - Nelze předpokládat nic o délce událostí
 - Bully algoritmus nebude fungovat, protože při překročení časových limitů budeme mít více koordinátorů
 - Mnohem lépe odpovídá požadavkům reálného prostředí
 - Vhodnější více pro gridové prostředí než clusterové
 - Rozsáhlejší distribuované systémy na nespolehlivých sítích
 - Všechny uzly jsou rozděleny na segmenty, které mezi sebou komunikují
 - Když se přeruší komunikace s nějakým uzlem nebo množinou uzlů, tak se segmenty rozpojí a každý segment funguje nezávisle na sobě
 - V rámci každého segmentu se ustanoví koordinátor, který pravidelně komunikuje s ostatními uzly v segmentu
 - Jak funguje

- Koordinátor posílá heartbeaty
 - V pravidelných krátkých subsekundových intervalech posílá zprávu všem ostatním v segmentu
 - Pokud přijme zprávu někdo, kdo si sám myslí, že je koordinátorem, tak dojde k propojení (dříve rozdělených) segmentů
 - Novým koordinátorem se stane ten kdo má větší identifikaci
- Pokud uzel neobdrží do timeoutu heartbeat
 - Uzel odvodí, že už není ve stejném segmentu jako jeho původní koordinátor
 - Neřeší jestli koordinátor havaroval nebo jestli nejde spojení k němu
 - Sám za sebe vytvoří nový segment (jednoprvkový) a prohlásí se za koordinátora (a posílá všem ostatním heartbeaty)
- Le Lann, Chang & Roberts (1979)
 - Zástupce tzv. **kruhových algoritmů**
 - Tenhle je jeden z těch jednodušších
 - Uzly **nemusí znát** identifikaci všech ostatních členů skupiny, dokonce ani nemusí vědět, kdo všechno je ve skupině
 - Vhodnější pro rozsáhlejší distribuované systémy
 - Stačí, aby uzel znal svého následníka a existovala možnost jak zjistit následovníka havarovaného uzlu



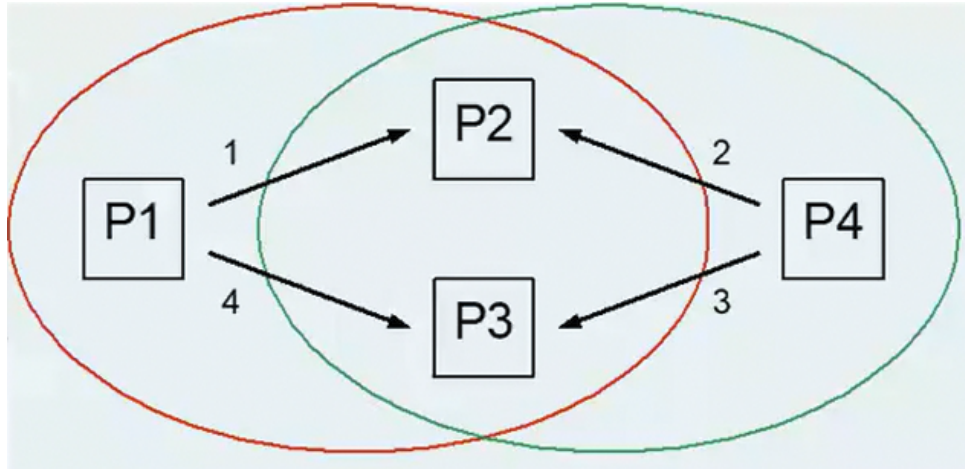
- V okamžiku, kdy uzel detekuje havárii (na obr. uzel 3) tak iniciuje volbu
- Pošle ve směru následníka zprávu, ve které bude
 - Jeho vlastní identifikace (aby se poznalo, že zpráva obkroužila kolečko)
 - Aktuální nejlepší kandidát na koordinátora
 - V okamžiku odeslání tam dá sebe sama
 - Až zpráva obkrouží kolečko bude obsahovat číslo nového koordinátora
- Detekci havarovaného uzlu může najednou provést více uzlů, takže těchto zpráv tam může probíhat více (maximálně n , což je počet uzlů)
- Zpráva proběhne n krát

- Celková složitost $O(n^2)$
- Existuje celá vylepšení tohoto kruhového algoritmu
- HS algoritmus
 - Hirschberg & Sinclair (1980)
 - Vylepšení předchozího
 - Když nějaký uzel zasílá zprávu o nové volbě, tak tato zpráva neobkrouží celé kolečko, ale okolí se exponenciálně zvětšuje
 - Do dalšího kola postoupí jenom ten kandidát, který v rámci lokálního okolí vyhrál
 - Celková složitost je $O(n \log n)$
 - Velmi implementačně náročný
 - Velká konstanta schovaná uvnitř O
- Randomizované protokoly
 - V předchozích algoritmech se deterministicky hledal uzel, který má největší hodnotu identifikace
 - Tady je úplně jedno, který kandidát zvítězí, podstatné je, aby zvítězil pouze jeden
 - Typickou používané v prostředí, kde je rychlá lokální síť s malou latencí a poměrně malým počtem aktivním uzlů
 - Velmi často se používá pro protokoly distribuovaného konsensu
 - 3, nebo 5 uzlů se účastní
 - Pro randomizovanou volbu leadera jsou jednotlivé uzly ve třech různých rolích
 - Dosavadní koordinátor se nazývá **leader**
 - Posílá heartbeat v pravidelných intervalech
 - Když nějaký uzel nedostane heartbeat po dosažení timeoutu
 - Prohlásí sám pro sebe dosavadního leadera za havarovaného a z iniciuje novou volbu → stává se **kandidátem**
 - V rámci každého kola voleb má každý uzel jeden hlas (uzel, který inicioval volbu tak volí sám sebe), všem ostatním uzlům pošle zprávu *RequestVote*
 - Když jiný uzel dostane tuto zprávu a ještě nehlasoval, tak svůj hlas odevzdá tomuto kandidátovi
 - Když jakýkoliv kandidát dostane nadpoloviční počet hlasů, tak se další zprávou prohlásí za nového leadera
 - Stejný zisk hlasů (potencionální deadlock) se řeší pomocí timeoutu a nové volby
 - Ukázalo se jako lepší řešení než u 前川
 - Když do timeoutu neobdrží kandidát příslušný počet hlasů, tak všechny uzly randomizovaně počkají čas $[T, 2T]$
 - Kde $T \gg$ (nejdelší) doba přenosu zprávy
 - Když T je vhodně zvoleno, tak se algoritmus chová dobře
 - Pokud uzly mají implementovaný 2-fázový commit, tak je možné provést volbu jiným způsobem, např. pomocí generátoru náhodných čísel
 - 2-fázový commit zajistí, že volbu vyhraje jen jeden
 - Apache Zookeeper

Video 6

- Budeme se zabývat doručovacími protokoly skupinové komunikace
- Ideální stav je globální uspořádání
 - Zprávy jsou doručovány v pořadí odesílání
 - Již víme, že nelze – neexistence globálních hodin
- Sekvenční uspořádání
 - (slabší forma doručování)
 - Všechny zprávy jsou pomyslně seřazeny do jednoho rozvrhu, všechny uzly doručí zprávy v rámci tohoto rozvrhu
 - Doručení nezávisí nutně na času odesílání
 - Důležité je pouze dodržení sekvenčnosti
 - Všechny uzly dodržují stejný rozvrh
 - Ukážeme si dva algoritmy
 - Distribuovaný a centralizovaný (total-order protokol, sekvencer)
- Pokud bychom sekvenční doručování doplnili o zaručení spolehlivosti → říkáme tomu **atomický multicast**
- Teď si budeme povídat o kauzálním doručování
 - Ne vždycky je potřeba, aby všechny zprávy byly seřazeny do jedné sekvence
 - Obvykle stačí uspořádání, kde ve správném pořadí jsou doručovány na všech uzlech jen ty zprávy, které jsou na sebe kauzálně vázané
 - Ostatní (konkurentní) zprávy mohou být doručeny v libovolném pořadí
- Kauzální uspořádání doručovaných zpráv
 - $dest(m)$... množina uzlů, kterým je zaslána zpráva m
 - $deliver_p(m)$... je událost doručení zprávy m uzlem p
 - $m_1 \rightarrow m_2 \Rightarrow \forall p \in dest(m_1) \cap dest(m_2) : deliver_p(m_1) \rightarrow^p deliver_p(m_2)$
 - Jestliže je m_2 je zpráva kauzálně závislá na m_1 potom na všech uzlech, kterým budou doručeny obě tyto zprávy, bude zachováno pořadí doručení
 - **Problém:** jak poznat, že $m_1 \rightarrow m_2$
 - Navíc když ještě m_1 nebyla doručena
 - Řešení: vektorové hodiny
- Vektorové hodiny
 - Složitější forma logických hodin
 - Vektor je organizovaný podle všech procesů
 - Každý proces má svůj vektor, složky odpovídají jednotlivým procesům ve skupině
 - Vektor délky n , kde n je počet procesů ve skupině
 - $VT(p)$... časová značka procesu
 - $VT(m)$... časová značka zprávy
 - S každou zprávou jde přibalený vektorový čas (značka)
 - Pravidla aktualizace časových značek
 - Při startu je $VT(p_i)$ nulový
 - Po každé když proces p_i odešle zprávu m , tak $VT(m) = ++VT(p_i)[i]$

- p_j při doručení m upraví své $VT(p_j)$ takto:
 - $\forall k \in 1 \dots n : VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k])$
 - Po složkách maxima ze stávajícího a přijatého vektoru
 - $\forall m_i, m_j, i \neq j : VT(m_i) \neq VT(m_j)$
 - Porovnávání časových značek
 - $VT_1 \leq VT_2 \Leftrightarrow \forall i: VT_1[i] \leq VT_2[i]$
 - $VT_1 < VT_2 \Leftrightarrow VT_1 \leq VT_2 \wedge \exists i: VT_1[i] < VT_2[i]$
- Kauzální doručovací protokol
 - V okamžiku odeslání zprávy m uzlem p_i
 - $VT(p_i)[i]++$
 - $VT(m) \leftarrow VT(p_i)$
 - V okamžiku přijetí zprávy m uzlem p_j od odesílatele p_i
 - Přijetí = zpráva fyzicky přijde ze síťového rozhraní
 - Proces p_j pozdrží doručení m dokud neplatí oboje
 - $VT(m)[i] = VT(p_j)[i] + 1$
 - Právě přijatá zpráva musí být nejbližší vyšší
 - Předchozí zprávy musí být přijaté
 - $VT(m)[k] \leq VT(p_j)[k]$ pro $k \neq i$
 - Přijímající vektor musí být alespoň tak velký jako jsou hodnoty příslušných složek vektoru v přijímané zprávě
 - Příjemce musí od ostatních procesů mít alespoň takové informace, které měl odesílající proces při odesílání původní zprávy
 - Příjemce v okamžiku doručení musí mít přijaté alespoň ty zprávy od všech ostatních procesů, které předcházely odeslání nyní přijímané zprávy
 - Po doručení zprávy m uzlem p_j
 - Doručení = po splnění všech podmínek daných protokolem je zpráva schválena pro další zpracování
 - $\forall k \in 1 \dots n: VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k])$
 - Po složkách se vezmou maxima
- Kauzální doručování a překrývající se skupiny



- I když máme sesynchronizované doručování v rámci jedné skupiny, tak když posíláme zprávy na procesy, které jsou v průniku, tak zprávy vzájemně (vůči červené a zelené) skupině nejsou sesynchronizovány
 - Může i nemusí nám to vadit
- Jedna možnost co s tím dělat
 - Všechny skupiny, které mají neprázdný průnik, tak bychom sjednotili do jedné velké skupiny
 - Kauzalitu doručování bychom zaručili v jedné velké skupině
 - Výhoda je, že protokol je stejně jednoduchý jako předtím
 - Takhle tranzitivně by se nám při sjednocování skupin mohli dostat do jedné skupiny procesy, které spolu vůbec nesouvisí
 - Mohlo by docházet ke ztrátě výkonu, kvůli synchronizaci zpráv, které spolu nesouvisí
- Pokud nám záleží na synchronizace mezi procesy, které jsou součástí několika skupin, tak se to dá dělat následujícím způsobem
- Kauzální doručovací protokol pro překrývající se skupiny
 - Vektorové hodiny si rozšíříme na maticové hodiny
 - Matice má tolik vektorů, kolik je skupin
 - $VT_a \dots$ časová značka skupiny g_a
 - $VT_a[i] \dots$ počet zpráv uzlu p_i do skupiny g_a
 - S každou zprávou, kterou proces odesílá pošle časovou značku všech skupin, kterých je členem
 - Když uzel p_i odesílá do skupiny g_a
 - $VT_a(p_i)[i]++$
 - $VT(m) = \cup_{g_b: p_i \in g_b} VT_b(p_i)$
 - Do zprávy přibalí všechny vektory, jejichž je odesílající proces členem
 - Přijetí zprávy m uzlem p_j (od p_i do g_a)

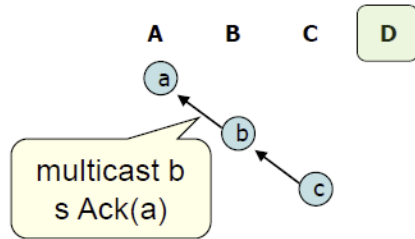
- Uzel $p_j \neq p_i$ pozdrží m dokud neplatí (první dvě jsou úplně stejné jako v případě jedné skupiny)
 - $VT_a(m)[i] = VT_a(p_j)[i] + 1$
 - Vektorové hodiny odpovídající skupině, do které je zpráva odeslána mají na složce vektoru, která odpovídá odesílajícímu procesu nejbližší vyšší hodnotu
 - $\forall k(p_k \in g_a \wedge k \neq i): VT_a(m)[k] \leq VT_a(p_j)[k]$
 - Kauzalita vzhledem ke skupině, do které byla zpráva zaslána
 - $\forall b(p_j \in g_b): VT_b(m) \leq VT_b(p_j)$
 - Kauzalita vzhledem k ostatním skupinám, kterých je příjemce členem
 - Příjemce musí v okamžiku doručení zprávy mít doručení zprávy od ostatním skupin, kterých je sám členem
 - Nová podmínka
 - Po doručení zprávy m si uzel p_j upraví **všechny** $VT_a(p_j)$
 - $VT_a(p_j) = \max(VT_a(p_j), VT_a(m)) \quad \forall a: p_j \in g_a$
- To bylo povídání o kauzálních protokolech, teď si ukážeme dva protokoly pro sekvenční doručování
 - Sekvenční doručování spočívá v tom, že všechny uzly respektují jeden rozvrh, kde nemusí nezbytně platit, že doručení zpráv odpovídá fyzickému času
 - U sekvenčního doručování nepotřebujeme detekovat kauzalitu
 - Stačí nám jedna časová řada, nepotřebujeme vektorové ani maticové hodiny → jednoduché skalární hodiny
- Distribuovaný total-order protokol
 - Funguje dvou fázově
 - Každý odesílatel nejprve odešle zprávu všem příjemcům, ty ji přijmou ale zatím nedoručují
 - Na každou přijatou zprávu odpoví zprávou $TS(R_i)$ s potvrzením, která obsahuje časovou značku příjmu zprávy, kterou potvrzují
 - Při příjmu zprávy se logické hodiny **sesynchronizují**
 - Ve chvíli kdy odesílatel dostane zpátky všechny potvrzení, tak odešle finalizační zprávu s finalizační časovou značkou $TSF = \max(TS(R_i))$
 - Maximum ze všech časových značek
 - Po příjmu finalizační zprávy doručí příjemce zprávu podle TSF
 - Časové značky příjemců určují pořadí doručení zpráv
 - **Nutnost jednoznačných** časových značek → byrokratické uspořádání
 - Komunikační složitost: $3n$
 - Datová zpráva, potvrzení, finalizační zpráva
 - Nešlo by to udělat elegantněji? → sekvencer
- Jiný způsob pro sekvenční doručování je použití **sekvenceru**

- Sekvencer je nějaký uzel, který poslouchá jak se mu doručují zprávy
- Jakýkoliv uzel, který chce rozeslat skupinovou zprávu, tak pošle jenom jednu zprávu sekvenceru
- Sekvencer podle toho v jakém pořadí mu dojdou zprávy, tak jim postupně dává vyšší a vyšší číslo a zprávy přeposílá všem příjemcům
- Forma centralizovaného řešení
 - Stejně jako u protokolů pro vzájemné vyloučení
 - Vylepšení pomocí distribuovaných řešení jsou jenom horší
 - Nyní je centralizovanou komponentou rozvrh – sekvenčnost doručování, na kterém se musí všichni shodnout
- Sekvenční doručování je jedna z forem centralizované komponenty → pokuste se ji vyhnout
 - Pokud ho nutně potřebujeme, tak nejjednodušší implementace je pomocí centralizované komponenty → sekvencer

Video 7

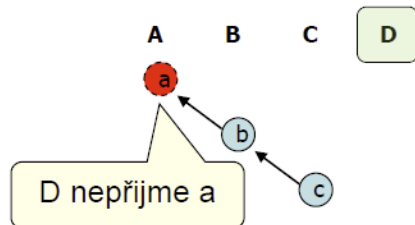
- Doposud jsme neřešili spolehlivost, předpokládali jsme, že zpráva bude doručena
- Spolehlivost na úrovni komunikačních kanálů se dá vyřešit na síťové vrstvě vhodným HW (Fibre Channel, ...)
- Daleko větší problém jsou výpadky uzlů
 - Může vypadnout příjemce
 - Může vypadnout odesílatel (to je ještě horší)
 - Který mohl odeslat zprávu polovině příjemců (druhou polovinu nestihl a umřel)
- Výpadek může být dočasný, nebo trvalý
- Distribuované transakce
 - Jedna z možností jak to řešit
 - Doručení zprávy doběhne až po commitu
 - Příliš striktní pro účely skupinové komunikace
 - Chtěli bychom něco slabšího/jednoduššího
- Spolehlivé kauzální doručování
 - Záplavový algoritmus
 - Když příjemce přijme zprávu, kterou neviděl, tak ji přepoše všem ostatním ve skupině
 - Všichni zprávu dostanou n krát, pokud nedošlo k havárii
 - Robustní a spolehlivé
 - Velmi neefektivní
 - Potřebujeme n^2 počet zpráv
 - Idea algoritmu s potvrzováním
 - Přeposílat zprávy by se nemělo automaticky všem, ale jenom těm potřebným
 - p_i odesílatel zprávy, p_j ostatní členové skupiny (příjemci), p_x je havarovaný uzel

- p_i odešle zprávu všem členům skupiny a zprávu si uchová do té doby, než obdrží potvrzení od všech uzlů skupiny, nebo zjistí od jiných uzlů, že p_x havaroval
- p_j při příjmu zprávy pošle potvrzení uzlu p_i a přijatou zprávu si uchová do té doby než zjistí, že zprávu přijali všechny uzly ve skupině
- Pokud p_j zjistí, že p_i havaroval, tak zprávu odešle všem uzlům, o kterých neví, že zprávu přijaly
- **Jak uzel p zjistí, které uzly zprávu přijaly?**
- Trans algoritmus
 - *Protokol pro spolehlivé kauzální doručování*
 - Na každém uzlu implementován **graf závislostí**
 - Pro účely potvrzování zpráv ($Ack(m)$)
 - Invariant
 - Když p rozesílá $Ack(m)$, tak ním potvrzuje **všechny** kauzálně předcházející zprávy (tomuto se říká transitive acknowledgment)
 - Nemusí potvrzovat každou individuální zprávu zvlášť
 - **Stabilní zpráva** = zpráva, která je přijata všemi členy skupiny
 - Graf závislostí G (někdy též graf kauzality)
 - Jedná se o DAG
 - Je grafem kauzality celé skupiny
 - Každý uzel si graf vytváří sám
 - Různé uzly si ho můžou vytvářet v odlišném pořadí, ale ve výsledku je distribuovaně stejný
 - Pokud nějaká zpráva m potvrzuje m' pak $(m, m') \in E(G)$
 - Orientace ve směru **potvrzování**, ne kauzality
 - Umožňuje zachytit i nepřijaté zprávy
 - 3 komponenty
 - Příjem potvrzení a výpočet vlastních potvrzení
 - Detekce nepřijatých zpráv
 - Ukládání zpráv a detekce stabilních zpráv
 - Jak p_j zjistí, které uzly zprávu přijaly?
 - Flooding
 - Rozesílání potvrzení všem uzlům ve skupině
 - Neefektivní, mnoho potvrzení zbytečných $\rightarrow O(n^2)$
 - Využití **kauzality** zpráv a **piggybackingu** potvrzení
 - Piggybacking
 - Místo speciální individuální zprávy jenom s potvrzením, se k běžné datové zprávě přidají dodatky, které označují potvrzování už nějakých existujících zpráv



♦ D může po příjmu a , $b + \text{Ack}(a)$, $c + \text{Ack}(b)$ odvodit:

- B přijal a
- C přijal a i b



♦ D může po příjmu $b + \text{Ack}(a)$, $c + \text{Ack}(b)$ odvodit:

- B přijal a
- C přijal a i b
- A odeslal $a \rightarrow$ žádost o zaslání

■ Při korektním kauzálním doručování jsou potvrzení zpráv tranzitivní

■ Implementace

- Každý uzel má
 - *ack_list* ... seznam zpráv, které přijal, ale musí na ně ještě odeslat *ACK*
 - *nak_list* ... seznam zpráv, které uzel nepřijal, ale z přijatých potvrzení od ostatních uzlů ví, že existují
 - *undelivered_list* ... seznam přijatých zpráv, které ale ještě nejsou doručeny
 - Typicky proto, že jsou kauzálně vázané a předchůdce ještě nepřišel
- Navíc si každý uzel ještě eviduje v rámci grafu
 - Které zprávy jsou kauzální
 - Které jsou stabilní
- Zprávu uzel vždy odesílá včetně jeho *ack_listu* a *nak_listu*
- Odeslání zprávy m

Trans_send(m)

$m \mathrel{+}= \text{nak_list}$

nak_list zůstává až do příjmu zprávy

$m \mathrel{+}= \text{ack_list}$

vyčištění *ack_listu*, přidat m

$\text{ack_list} = m$

m do grafu kauzality

$G \mathrel{+}= m$

send m to every node

- Příjem zprávy m (jenom zhruba)
 - Nejdřív se podívá na *nak*y přilepené s m
 - Pokud tyto zprávy má, tak je odesílateli zašle
 - Potom se podívá na *ack*y přilepené s m
 - Pokud mezi nima vidí zprávu, kterou nemá ve svém grafu závislostí, tak si ji přidá do *nak_listu*
 - Pokud zprávu m má v *nak_listu*, tak ji z něho odebere

- Zatím zprávu m **nedoručuje** → nevíme jestli máme přijaté a doručené všechny kauzální předchůdce
- Zprávu m si přidá do svého grafu a *undelivered_listu*
- Teď řeší, kdy je možné doručit zprávy z *undelivered_listu*
 - Každou zprávu je možné doručit, pokud byly všechny kauzálně předcházející zprávy doručeny
 - V okamžiku, kdy všechny kauzálně předcházející zprávy jsou doručeny, tak zprávu doručí a vyhodí z *undelivered_listu*
- Každou zprávu z G , která **nebyla potvrzena**, ale všechny kauzální předchůdci **byli doručeni**, přidá do *ack_listu*
- Všechny stabilní zprávy smaže z G
 - Stabilní zpráva = od všech ostatních uzlů na ni dostal potvrzení

Trans_Receive (m)

```

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

```

přeposlání vyžádaných zpráv všem

včetně ACK!

detekce nepřijaté zprávy

všechny kauzálně předcházející
byly doručeny

všechny kauzální zprávy
které nebyly tranzitivně potvrzeny

$n' \rightarrow n$, n' potvrzuje zprávu n

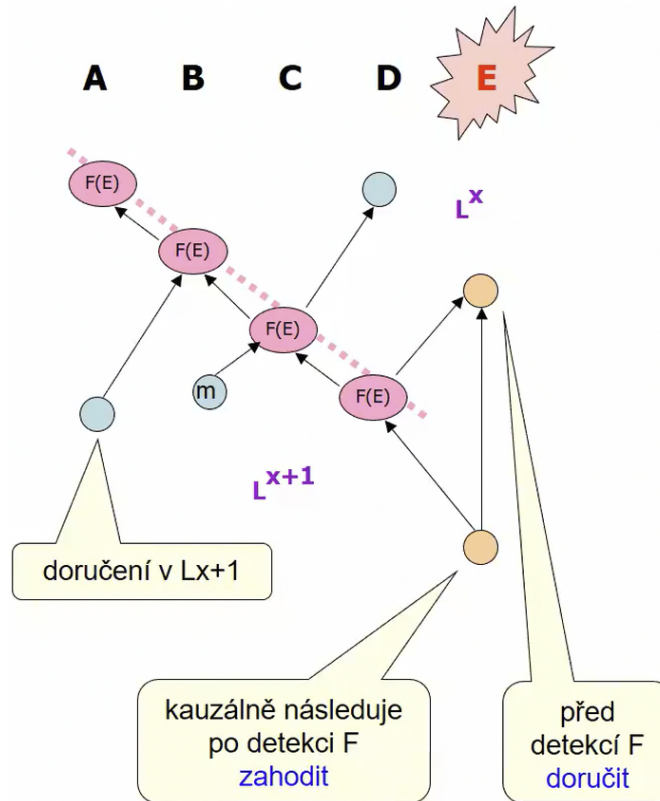
vyčištění všech stabilních zpráv

- Pozorování
 - Každý uzel postupně posouvá kauzální a stabilní hranici
 - Různé uzly mohou dělat posuny v různém pořadí

- Každá nově příchozí zpráva je mimo kauzální hranici a v případě, že jsou doručeny všechny její kauzální předchůdci, tak ji je možné doručit
- Když zprávu doručíme spolehlivému uzlu, tak potom pomocí *ACK* a *NAK* je zpráva nakonec doručena každému nezhavarovanému uzlu
- *ACK* rozesílá informace o tom, které zprávy jsou k dispozici
- *NAK* slouží k tomu, aby si příjemci, který daný zprávy nepřijali, si o ně mohli říct
- Zprávy jsou doručovány v kauzálním pořadí
- Jestliže uzel havaruje, tak paměťová náročnost protokolu je **neomezená**
 - Každý uzel si schovává zprávy, které nebyly doručeny všem uzlům
 - Slabina, pro praktické použití musí být trans protokol doplněn protokolem pro **změnu členství** ve skupinách
 - Když je nějaký uzel dostatečně důvěryhodně nařčený z toho, že havaroval, tak ostatní uzly si musí příslušný uzel vyřadit z evidence
- Virtuální synchronie
 - Abychom mohli podrobněji zkoumat chování doručovacích protokolů v prostředí, kde dochází v výpadku uzlům, tak se seznámíme s virtuální synchronií
 - Aktuální množina uzlů ve skupině = group view (nebo pouze view)
 - Český pohled
 - Značení
 - L
 - Spodní index – jak pohled vnímá konkrétní proces
 - Horní index – verze konkrétního pohledu
 - Definice
 - $p, q \text{ procesy} \in L^x, L^{x+1}$
 - p i q jsou součástí dvou po sobě následujících pohledů
 - $install L_p^x < deliver_p(m) < install L_p^{x+1} \Rightarrow install L_q^x < deliver_q(m) < install L_q^{x+1}$
 - Pokud nějaká zpráva je na uzel p doručena v rámci pohledu L_p^x (po instalaci pohledu L_p^x , ale před instalací L_p^{x+1} z pohledu uzlu p), tak potom to samé platí pro všechny ostatní procesy, které jsou součástí dvou následujících pohledů
 - Takováto komunikace se nazývá synchronní vůči pohledu
 - Pokud je zpráva m odeslána skupině s L^x před změnou na L^{x+1}
 - Buď m doručí **všechny** uzly z L^x před provedením změny na L^{x+1}
 - Nebo **žádný** uzel z L^x který provede změnu na L^{x+1} zprávu m nedoručí

- Všechny definice (i různé alternativní) dodržují **podmínku vzájemné konzistence**
 - $p \in L_q \Rightarrow q \in L_p$ (musí platit pro všechny procesy, které jsou členem nějakého konkrétního pohledu)
 - Všechny uzly ve skupině udržují stejný pohled L
 - Instalují si nové pohledy ve stejném pořadí (a jsou v nich vzájemně obsaženi)
- Pokud havaruje odesílatel zprávy, tak že havárie bude rychlejší než jeho odeslání, tak se nic nestane → nejde rozeznat havárii při “snaze o odeslání zprávy” se stavem beze snahy
- Pokud se to povede odeslat zprávu alespoň jednomu uzlu a pak umře, komunikační mechanismu potom musí zařídit, že zprávu musí doručit všem ostatním přeživším uzlům pohledu
- Je nepřípustné
 - Aby zpráva po havárii odesílatele byla doručena v následujícím pohledu
 - V následujícím pohledu už havarovaný odesílatel není přítomný
 - Aby jedna zpráva byla doručena některým uzlům v jednom pohledu a jiným v následujícím pohledu
- Transis algoritmus
 - *Spolehlivý kauzální multicast*
 - Rozšiřuje protokol Trans o detekci havarovaných komponent a reakci na zprávy o jejich detekci
 - Navíc oproti Trans algoritmu **podporuje** členství ve skupinách
 - **Monotónnost** protokolu
 - Vynucena **nemožností dosažením spolehlivého distribuovaného konsensu**
 - Paranoia
 - V Socialistickém/komunistickém zákonodárství už jenom podezření a spáchání trestního činu proti režimu stačilo ke zbavení všech funkcí, výhod a zařazení (i když se potom nic neprokázalo)
 - Funguje podobně, stačí jenom podezření jednoho uzlu, že nějaký uzel je havarovaný a ostatní uzly toto podezření musí respektovat
 - K jejímu vyvolání stačí pouze jedna zpráva, která je zpožděná nad timeout (nerozeznatelné od havárie)
 - Jednosměrnost
 - Procesy, které jsou paranoicky vyloučené, se zpátky automaticky nevrací, ale mohou se explicitně připojit (stejným způsobem jako úplně nové procesy)
 - Idea protokolu
 - Kauzální hranice pohledu
 - Doručení předcházejících zpráv
 - Pozdržení zpráv kauzálně následujících

- Všechny uzly si postupně instalují pohledy
- Všechny zprávy jsou doručovány v rámci konkrétního pohledu
- Pohledy jsou instalovány na jednotlivé uzly ve **stejném** pořadí
- Při detekci havárie (ať už faktické nebo domnělé)
 - Každý proces posílá všem zprávu *FAULT*
 - Po detekci *FAULT* zpráv si uzly instalují takové hranice, aby
 - Všechny zprávy, které přišly **před** *FAULT*em kauzálně předcházely změně hranice pohledu
 - Všechny zprávy, které přišly **po** *FAULT*u kauzálně následovaly na informaci o změně pohledu
 - Pokud uzel obdržel *FAULT*y od **všech procesů**, tak
 - Instaluje nový pohled
 - Doručí zprávy, které kauzálně následují změně hranici pohledu
- Detekcí havárií může být zároveň několik v rámci jednoho pohledu
- Scenario
 - Uzel *E* je označený uzlem *A* jako havarovaný (*A* je prvním, který zjistil havárii)
 - Zprávu o havárii přepošle ostatním
 - Když uzel přijme zprávu o havárii, tak okamžik příjmu je pro něho hranice pohledu
 - Každý uzel si takto nastaví svoji lokální hranici
 - Zprávy, které uzel přijal **před** okamžikem příjmu zprávy o havárii, které kauzálně předcházejí doručí v původním pohledu
 - Zprávy, které kauzálně **následují** doručení zprávě o havárii, jsou pozdrženy do dalšího pohledu → po instalaci jsou doručeny



- Implementace (detekci havárie)
 - G ... graf kauzálních závislostí (stejný jako u Trans algoritmu)
 - L ... aktuální pohled – množinu uzlů
 - F ... množina uzlů označených za havarované
 - $FAULT(q)$... zpráva s detekcí havárie uzlu q
 - $blocked$... blokové zprávy (k doručení v následujícím pohledu)
 - $Last[i]$... uzly označené procesem i za havarované
- Detekce havárie

Transis_failure(q)

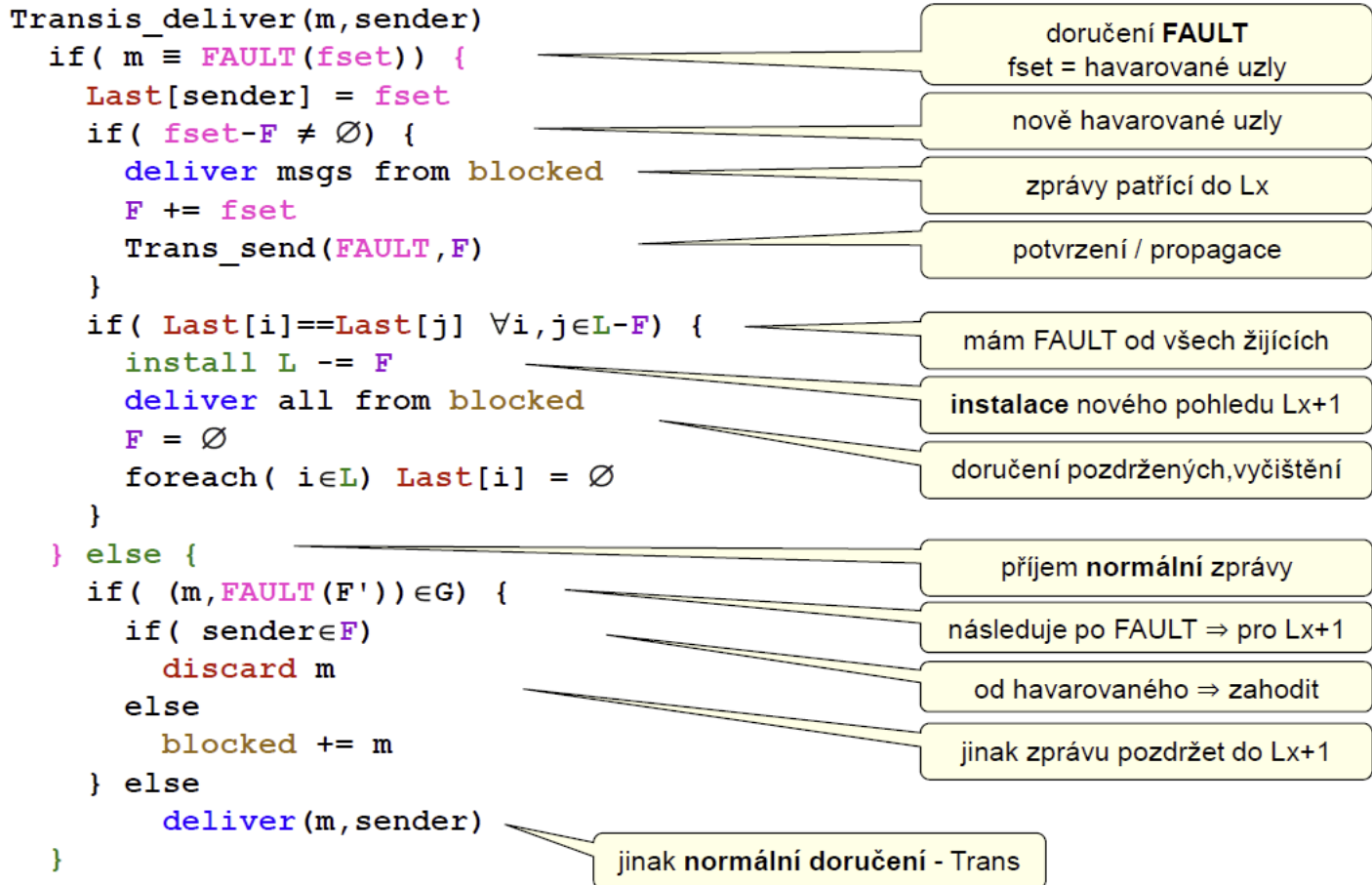
$F += q$

$Last[self] = F$

Trans_send($FAULT, F$)

- Doručení

detekce havárie uzlu q



- ISIS protokol

- Základem protokolů Trans a Transis je kauzální potvrzování zpráv
- Tento protokol funguje jinak a používá maticové hodiny (MT)
- (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 uzlů
- S každou zprávou je odeslán **vektor**
 - p_i odešle $MT_{p_i}[i][*]$
- Každý uzel udržuje matici, která je složena ze všech vektorů všech ostatních uzlů
- Každý uzel zná poslední informace, které mu přišly zároveň se zprávou od všech odesílatelů
- $MT_{p_i}[j][k]$... co uzel p_i ví o doručení zpráv uzlu p_j od p_k
 - Jeho vlastní vektor je $MT_{p_i}[i][*]$
- p_i **při příjmu** zprávy od p_j aktualizuje
 - $MT_{p_i}[i][j] = VT_m[j]$

- Aktualizuje si vlastní hodiny
- $MT_{p_i}[j][*] = VT_m[*]$
 - Aktualizuje hodiny odesílatele
- Podle maticových hodin mohou být detekovány stabilní zprávy
 - Na každý uzel jsou doručeny vektorové hodiny všech členů skupiny, které odpovídají odeslané zprávě
- Členství ve skupinách – jak funguje
 - Cíl: Před instalací nového pohledu musí být doručeny všechny zprávy, které byly doručeny alespoň jednomu přeživšímu procesu
 - Každý uzel v L si udržuje všechny zprávy, které zatím nejsou stabilní (pozná podle maticových hodin)
 - Potom co uzel dostane zprávu o instalaci nového pohledu
 - Typicky potom co nějaký jiný uzel zdetekuje havárii
 - Tak přepośle všechny nestabilní zprávy všem ostatním uzlům
 - Potom pošle speciální *FLUSH* zprávu
 - Tímto oznamuje, že je připraven přejít na nový pohled
 - Zatím neinstaluje
 - Čeká na přijetí *FLUSH* zpráv od všech ostatních uzlů
 - Až dostane *FLUSH* zprávy od všech uzlů může instalovat nový pohled
 - Zprávy od havarovaných uzlů
 - Každý uzel si udržuje svůj seznam “havarovaných uzlů”
 - S každou zprávou rozesílá jejich seznam, při příjmu sjednotí s vlastním
 - Zprávy od havarovaných uzlů se zahazují
- Reálně tento protokol byl nasazen
- Porovnání ISIS a Trans
 - *ACK* Trans algoritmu je možné považovat za určitou formu komprimace maticových hodin
 - Ale vyžaduje složitější datové struktury a složitější doručovací algoritmus
- Shrnutí doručovacích protokolů



Doručovací protokoly - shrnutí

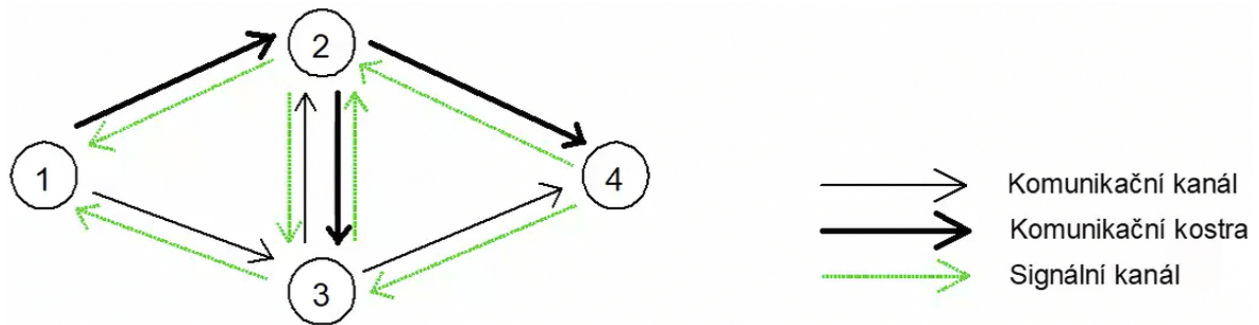
Shrnutí problémů doručovacích protokolů
principy a protokoly, které je řeší

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

- Pro sekvenční doručování jsme si ukazovali dva protokoly, jeden distribuovaný za použití skalárních hodin a druhý centralizovaný se sekvencerem.
- Pro kauzální doručování jsme si uváděli několik protokolů, pokud jsme neřešili překrývající se skupiny, stačili nám vektorové hodiny, jinak jsme používali maticové hodiny.
- Pro zajištění komunikace v nespolehlivém prostředí a pro detekci stabilních zpráv jsme si ukázali algoritmus Trans
- Pro implementaci virtuální synchronie jsme si ukázali Transis a ISIS

Video 8

- Budeme si povídat o globálním stavu v distribuovaných systémech a o dosažení konsensu
 - Tyto dvě témata spolu úzce souvisí
- Konsensus = něco na čem se všechny uzly distribuovaně dohodnou
- Diffusing Computation
 - Jednoduchý model pro distribuované výpočty
 - Uzly jsou vzájemně propojeny jednosměrnými komunikačními kanály
 - Dvousměrný se namodeluje dvěma komunikačními kanály na obě strany



- V opačném směru oproti datovým kanálům jsou signální kanály
 - Nejsou určeny pro aplikační data
 - Pouze pro signály v rámci synchronizačních protokolů (pro synchronizaci uzlů)
- Ukážeme si jak se počítá dynamická kostra (počítá se dynamicky během výpočtu)
 - Děje se to elegantně během distribuovaného výpočtu
 - Uzel u předá **první** zprávu uzlu v
 - v označí u za svého otce v rámci kostry
- Každý uzel je v **aktivním** nebo **pasivním** stavu
- **Aktivní stav**
 - Něco počítá
 - Během výpočtu může pomocí komunikačních kanálů posílat zprávy dalším uzlům
 - Příjemci mohou být v aktivním nebo pasivním stavu → po příjmu zprávy se uzel dostane do aktivního stavu
- **Pasivní stav**
 - Dostane se do něho, když nějaký výpočet skončil a už nemá, co by počítal
- V diffusing computation se dají dělat různá zjištění
 - **Kdy celý distribuovaný výpočet skončí?**
- Problém ukončení distribuovaných procesů
 - Protokol pro detekci ukončení distribuovaného výpočtu by měl detekovat konec výpočtu v konečném čase po tom, co všechny uzly jsou v pasivním stavu
- Dijkstra-Scholten (DS) algoritmus
 - Detekce ukončení distribuovaného výpočtu
 - Pokud vlastní graf výpočtu je strom
 - Triviální
 - Každý listový proces při přechodu do pasivního stavu pošle signál otcí
 - Pokud mezilehlý uzel dostane signály od všech svých synů, tak také pošle signál svému otcí
 - Celý výpočet běží tak dlouho, než se signály dostanou ke kořenu
 - Pokud graf výpočtu je DAG
 - Původní stromový algoritmus se dá vylepšit drobným vylepšením → přidáme deficit

- Deficit je jednoduchý čítač asociovaný s každou hranou
 - Rozdíl mezi počtem zpráv, které došly touto hranou a počtem signálů poslaných signálním kanálem zpět
 - Proces, který je v pasivním stavu počká než jeho výstupní kanály mají deficit nulový = přišly všechny signály
 - V okamžiku, kdy je deficit nulový na **výstupních** kanálech, tak pošle tolik signálů, aby na **vstupních** kanálech byl deficit nulový
 - (Funguje to, protože v DAGu existuje vrchol, který nemá žádné výstupní hrany → to bude ten, který bude ukončen jako první)
 - Zesložnění oproti stromové variantě spočítá v tom, že jeden uzel může mít několik otců a deficit se řeší to, kolik kterými kanály přišlo zpráv, abychom zpátky poslali stejný počet zpráv (v obráceném směru)
- Pokud graf výpočtu je obecný orientovaný graf
 - Nemáme listové uzly
 - Přijde nám vhod dynamicky vytvářená kostra
 - Otcem v rámci kostry je uzel, od kterého přišla první zpráva
 - Každý proces
 - Pošle signál všemi signálními kanály kromě kanálu k otci
 - Počká na signály od všech komunikačních kanálů (výstupních hran)
 - Potom co dostane příslušný počet signálů tak pošle signál otci
 - Když se signály dostanou od všech synů kořene, tak konec výpočtu
- Huangův algoritmus
 - Využívá váhy
 - Každá zpráva má váhu
 - Iniciátor má celkovou váhu (nějaké velké číslo)
 - Ostatní procesy mají na začátku váhu nulovou
 - Když nějaký proces (iniciátor) chce po jiném něco počítat, pošle mu zprávu a do zprávy dá část své váhy
 - Při příjmu zprávy si každý příjemce přičte váhu zprávy ke své váze
 - Proces, když je ukončen, tak pošle iniciátoru zprávu s celou vlastní vahou
 - Pokud se iniciátor dostane na svoji původní váhu → konec výpočtu
 - Problémy
 - Dělitelnost vah
 - Havárie procesu = ztráta váhy
- Značkový (TM) algoritmus (detekce ukončení výpočtu)
 - Kromě běžných zpráv je zde speciální zpráva odlišitelná od běžných zpráv → **značka**
 - V okamžiku, kdy iniciační uzel má podezření na ukončení distribuovaného výpočtu, tak pošle značku na **všechny výstupní kanály**
 - Trochu obdobné protokolu ISIS
 - Příjem první značky
 - Prohlásí kanál za kostru
 - Využívá principy dynamické kostry

- Uzel je aktivní
 - Pošle negativní signál
 - Uzel je pasivní
 - Propaguje značku všemi výstupními kanály
- Příjem další značky
 - Pošle signál **zpátky** příchozímu kanálu, když mu ze všech výstupních kanálů přišly signály
 - Jinak pošle negativní signál **zpátky** příchozímu kanálu
- Příjem negativního signálu
 - Pošle negativní signál první žádosti
- Když uzel přijme všechny signály na svoje značky
 - Pošle signál své první žádosti
- Když se signály dostanou zpátky k iniciátorovi, tak konec výpočtu
- Tento algoritmus je pouze aplikace obecnějšího (značkového) algoritmu na detekci globálního stavu
- Konzistentní stav
 - Nejprve si povíme, co to vlastně konzistentní stav je
 - Budeme ho chtít zjišťovat
 - E ... množina událostí v systému
 - Řez c je rozdělení E na dvě disjunktní množiny P_c a F_c (past, future)
 - Řez je právě hranice mezi minulostí a budoucností
 - (Kauzálně) konzistentní řez c
 - Splňuje $a \rightarrow b \wedge a \in F_c \Rightarrow b \in F_c$
 - Minulost nemůžeme ovlivnit jakoukoliv událostí v budoucnosti
 - Stav distribuovaného výpočtu je množina událostí, které se během výpočtu udály
 - (Kauzálně) konzistentní stav $S = P_c$, kde c je (kauzálně) konzistentní řez
 - (Kauzálně) konzistentní stav odpovídá nějakému (kauzálně) konzistentnímu řezu

Stav distribuovaného výpočtu je množina událostí, které se během výpočtu udály (Kauzálně) **Konzistentní stav** $S = P_c$, kde c je konzistentní řez

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

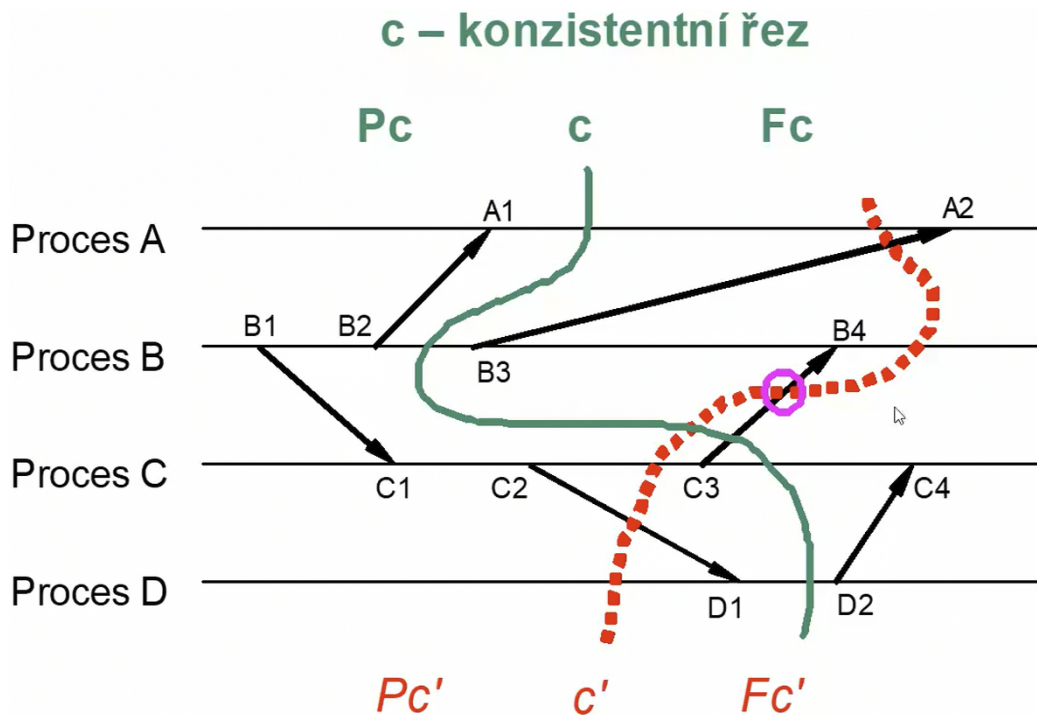
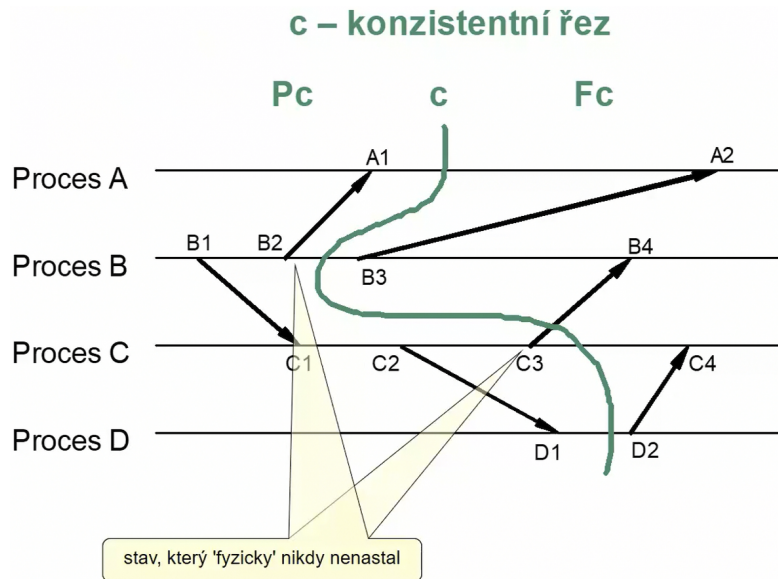
$S \xrightarrow{e} S'$ (S' je **dosažitelný** z S)

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

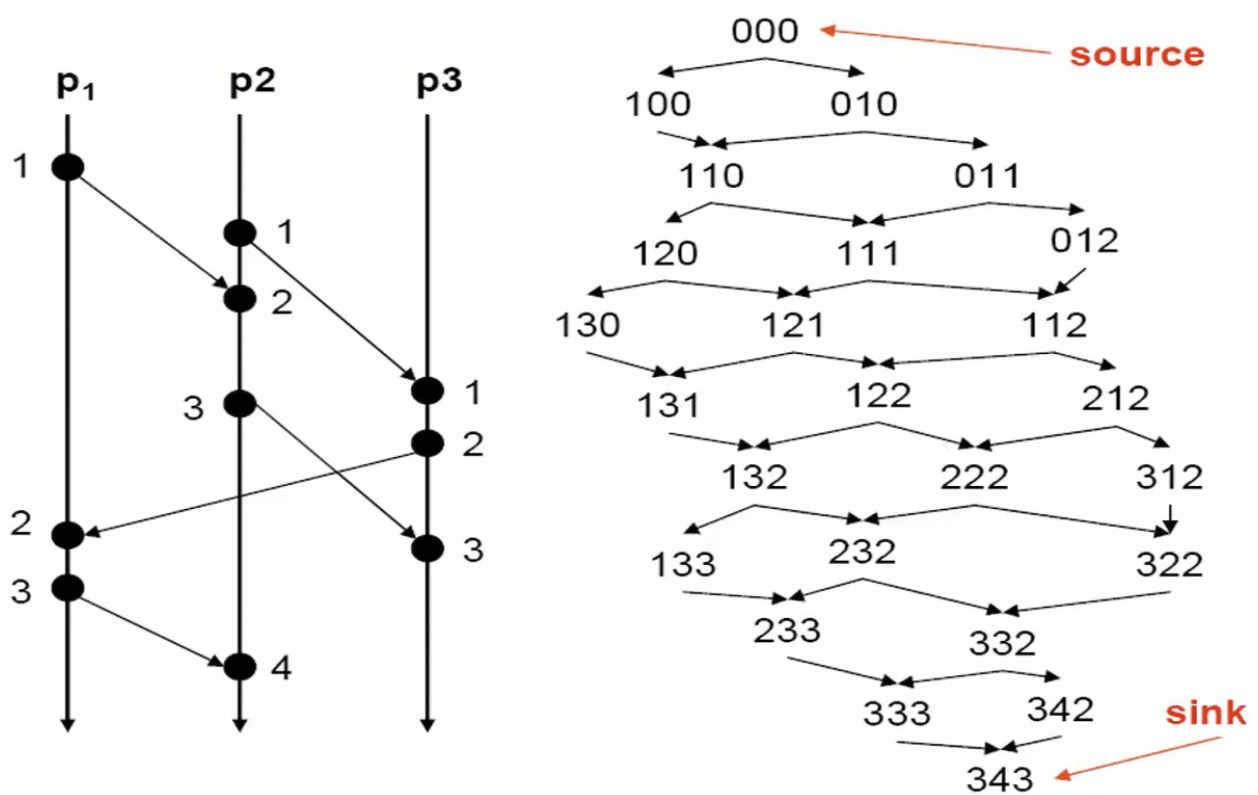
- Využití
 - Detekce deadlocku
 - Garbage collection
 - Detekce globálních vlastností
- Konzistentní řez



c' – nekonzistentní řez

- Značkový algoritmus detekce globálního stavu
 - Jeho výsledkem je zaznamenání **stavu uzlu a kanálů**
 - **Stavu uzlu a kanálů** dohromady definuje kauzálně konzistentní stav systému
 - = stav, který odpovídá kauzálně konzistentnímu řezu
 - Stav uzlu
 - Množina přijatých a odeslaných zpráv
 - Stav kanálu

- Množina zpráv, které byly do kanálu odeslány v minulosti, ale přijaty budou až v budoucnosti (ještě nebyly doručeny)
- Jak funguje
 - Existuje speciální zpráva (značka)
 - Iniciátor, když se rozhodne detekovat globální stav, tak všemi výstupními kanály pošle značku
 - Každý uzel v okamžiku příjmu první značky ze svých vstupních kanálů si zapamatuje svůj aktuální stav (okamžik řezu)
 - Dále si uzel označí všechny vstupní kanály jako prázdné a čeká na příjem značky z těch kanálů, ze kterých ještě nepřišla
 - Uzel propaguje značku podél všech výstupních kanálů
 - Čeká dokud mu ze vstupních kanálů nepřijdou další zprávy se značkou
 - Když mu přijde zpráva (ne značka) z kanálu, ze kterého ještě značka nepřišla
 - Zapamatuje si tuto zprávu (bude ji potřebovat pro stav kanálu)
 - Když mu přijde značka z kanálu, ze kterého ještě nepřišla
 - Definuje stav tohoto kanálu jako zprávy došlé mezi příjmem první značky a této značky
 - Když ze všech vstupních kanálů dostal značku, tak pro tento uzel algoritmus skončil, takže svému otci (v proti směru) pošle svůj stav
- Množina zaznamenaných stavů uzlů a kanálů definuje kauzálně konzistentní stav systému
- Aplikace
 - Obecně detekce libovolných globálních vlastností
 - Deadlock, garbage collection, ...
- Modely distribuovaných výpočtů
 - Většinou jsme si ukazovali kauzální časové diagramy
 - Každý proces má svoji časovou osu
 - Jednotlivé události jsou na časové ose znázorněny
 - Používají se ale i jiné modely než kauzální časové diagramy
 - Pro zkoumání distribuovaných algoritmů a jejich vlastností
 - Umožňuje znázornit přechody mezi stavy
 - Každý stav je identifikovaný



Video 9

- Armádní tematika
- Budeme se věnovat distribuovanému konsensu
- Problém dvou armád
 - Jsme generálem větší armády B
 - Armáda je rozdělena na půlku na dvou místech
 - Bojujeme proti armádě W
 - W je menší, ale je koncentrována na jednom místě
 - Nedokážeme porazit W pouze jednou částí naší armády, víme, že vyhrájeme pouze pokud synchronně obě naše armády zaútočí



- Zajímavý vojenský strategický problém
- Jediná možná komunikace mezi armádou je pomocí tajných kurýrů přes území nepřítele

- Nepřátelské území je hlídané a chráněné
- Pokud by nepřítel přišel na našeho kurýra, tak ho zajme...
- Zpráva nebude doručena druhé části naší armády
- Jaké vymyslet řešení "protokol", aby se obě armády dohodly
 - Že buď obě armády zaútočí společně
 - Nebo neútočit
- V řeči informatiky
 - Máme dvě komponenty, které komunikují pomocí zasílání zpráv
 - Zasílání zpráv je nespolehlivé
 - Potřebujeme protokol, kde obě části budou spolehlivě vědět
 - Že zpráva byla doručena
 - Druhá komponenta bude vědět o tom, že první komponenta ví o tom, že zpráva byla doručena

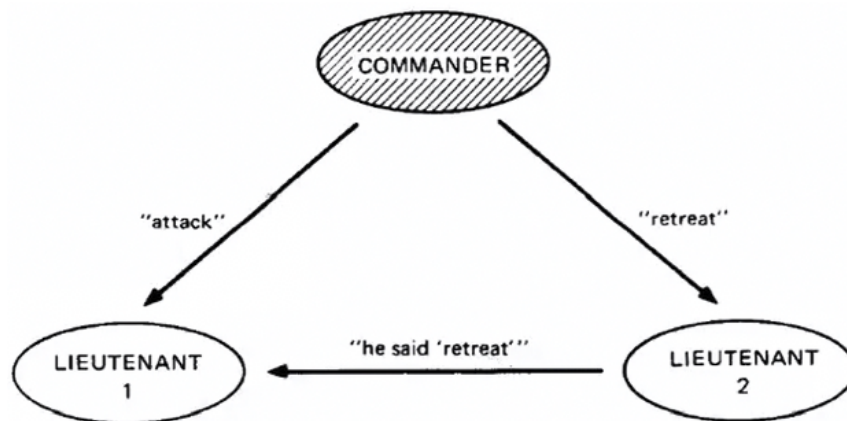
Striktní řešení NEEXISTUJE !!!

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

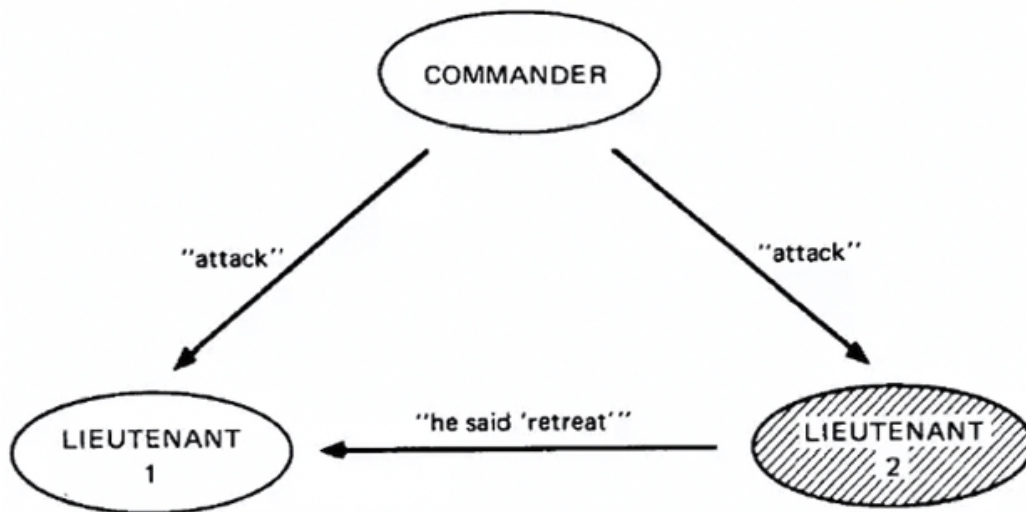
formální důkaz indukci

- Celá řada důsledků
 - Např. paranoia u Transis algoritmu
- Praktická řešení
 - Agresivní strategie
 - První generál vyšle větší množství zpráv oznamující čas útoku
 - Bude doufat, aby alespoň jeden kurýr dorazil
 - A zaútočí
 - (druhý generál nemusí ani odpovídat)
 - Pravděpodobnostní strategie
 - Spočívá ve zkoumání prostředí a propustnosti terénu
 - První generál vyšle větší množství zpráv a každému kurýrovi kromě zprávy předá ještě informaci o tom, kolik kurýrů bylo posláno
 - Když alespoň jeden kurýr dojde na druhou stranu, tak to samé udělá druhá strana zpátky a když se kurýr dostane zpátky k první armádě (pokud se dostane), tak oba generálové vědějí nějakou pravděpodobnost, že se zpráva dostane do cíle
- **Tohle je problematika spolehlivých uzlů, ale nespolehlivé komunikace**
- Problém Byzantských generálů
 - Jednotlivé uzly nejsou spolehlivé = **někteří generálové jsou zrádci**
 - Název pochází z historie → důvěryhodnost byzantských generálů byla nízká
 - Nepřestane plnit rozkazy, ale začne se chovat zákeřně

- Bezpečnost distribuovaného systému oproti úmyslným škůdcům
- **Všichni loajální generálové se musejí rozhodnout stejně**
- **Každý generál se rozhoduje na základě informací obdržených od ostatních generálů**
- BÚNO se problém dá zjednodušit
 - Pouze 1 generál a ostatní jsou důstojníci
 - Generál i důstojníci mohou být zrádci
 - Generál vydá rozkaz, důstojníci ho předají ostatním
 - Každý velitel vydá rozkaz, který bude vydán na základě většiny
- **Cíl protokolu**
 - C1
 - Všichni loajální důstojníci vydají stejný rozkaz
 - (Uzly se shodnou na jedné hodnotě)
 - C2
 - Je-li generál loajální, pak každý loajální důstojník vydá rozkaz generála
 - (Pokud hodnota navrhl nehavarovaný uzel, uzly se shodnou na této hodnotě)
- Zrádce generál



- 1 generál a 2 důstojníci
- Různé rozkazy důstojníkům
 - Důstojníci si předají zprávu a z jedné strany je útok a z druhé ústup
- Pokud by generál dal stejný rozkaz, tak by se jeho neloajálnost neprojevila
- Zrádce důstojník

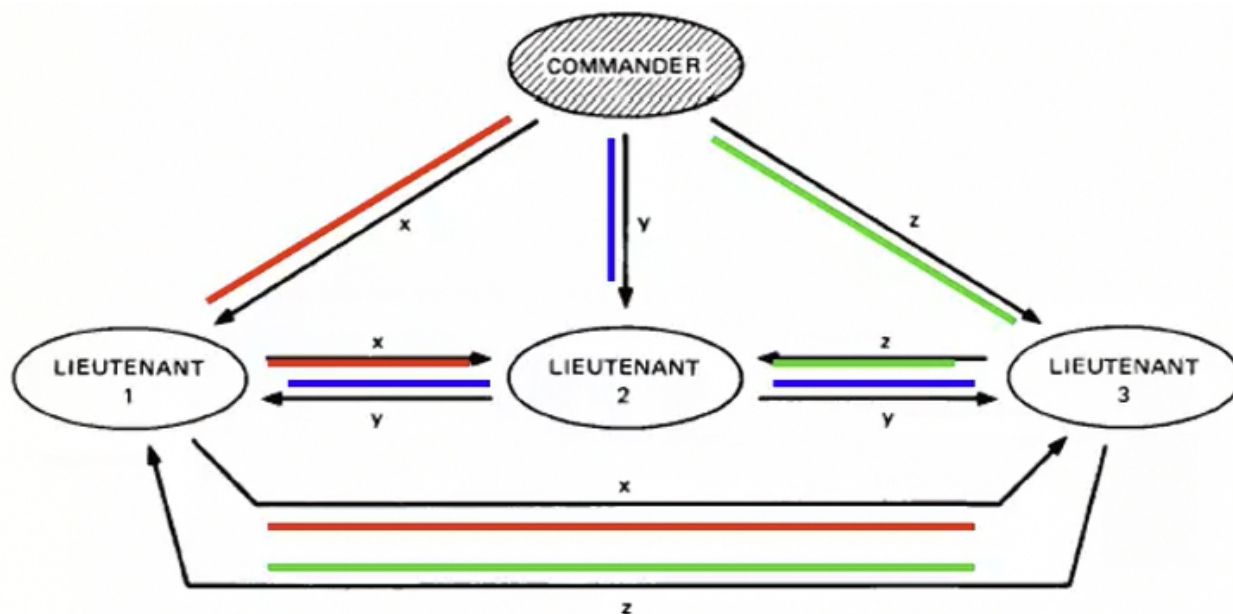


- Generál zavelí na obě strany útočit a zrádce důstojník by přeposlal ústup
- Zase jeden důstojník by z každé strany dostal jinou informaci
 - Z jeho pohledu opět nerozhodnutelné

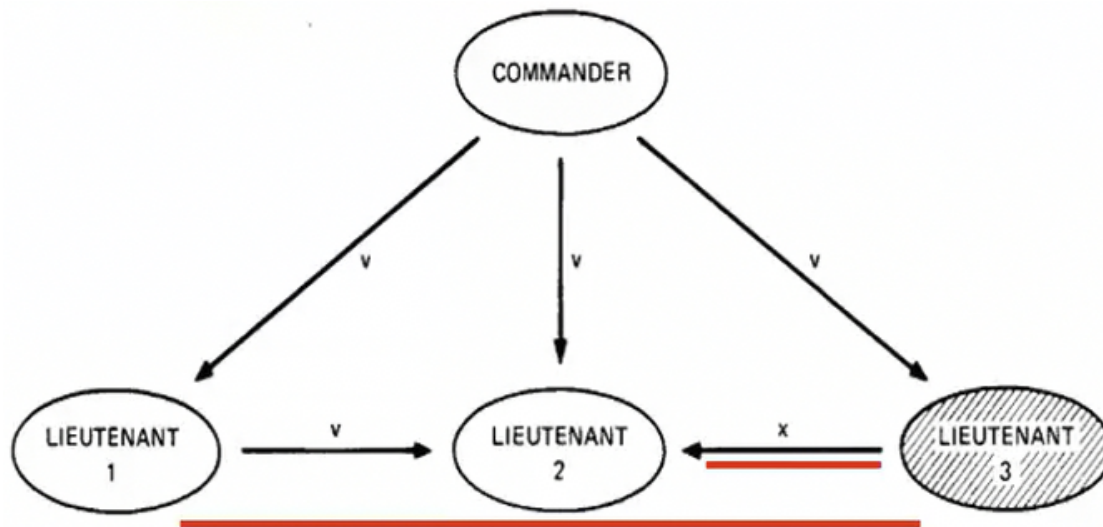
💡 Řešení pro 3 uzly s 1 zrádcem neexistuje

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

- Zrádce generál (zvětšíme počet uzlů)



- To nejhorší co generál může udělat je vydat na každou stranu rozdílný rozkaz
 - Důstojníci se shodnou, že generál je zrádce
- Pokud vydá alespoň dva stejné rozkazy (důstojníci se zařídí podle většinového rozkazu)
- Zrádce důstojník (zvětšíme počet uzlů)



- Nejhorší co důstojník může udělat je předání falešného rozkazu všem ostatním důstojníkům
 - Ostatní důstojníci si vzájemně pošlou zprávu co dostanou od generála
 - Snadno odvodí, že důstojník se špatnou zprávou je zrádce
- Řešení pro 4 uzly s jedním zrádcem existuje
- Obecně: Pro m zrádců existuje řešení pro $n \geq 3m + 1$ uzlů
 - Existuje algoritmus, který tento problém řeší
 - Podle počtu zrádců ale exponenciálně roste počet zpráv
 - Jeho nevýhoda je tedy exponenciální složitost
 - Vzájemně si důstojníci přeposílají všechny informace, co jim přišly
- Praktické řešení problému
 - Předchozí popsané řešení se (už jenom z hlediska efektivity) nepoužívá
 - Řešíme pomocí kryptografie, digitální podpis

Video 10

- Nyní se budeme zabývat protokoly pro dosažení distribuovaného konsensu, které jsou použitelné v reálném prostředí
- Reálné prostředí
 - Nespolehlivá síť
 - Neomezená doba doručení zprávy
 - Ztráta zprávy, duplikace, pořadí
 - Nespolehlivé uzly, náhlé havárie
 - **Nepředpokládají se byzantské uzly**, všichni dodržují protokol
- Paxos



- Klasika mezi konsensuálními protokoly
- Celá rodina protokolů
- *Výpadek není výjimka, ale běžný stav*
- Když chceme tolerovat výpadek F uzlů, tak potřebujeme $2F + 1$ uzlů celkově
- Základ moderních distribuovaných systémů (cloudů)
- Autor: Leslie Lamport
 - 1989 odmítnut
 - Napsal článek o zákonodárcích na ostrově Paxos
 - Článek byl pojat jako pohádka se spoustou řeckých názvů v řečtině
 - Velmi obtížné číst
 - Naštval se a článek schoval do šuplíku
 - Na univerzitě bylo celá řada spolupracujících kolegů a o článku věděli a na jeho základě dělali různé experimentální systémy → akorát je nemohli publikovat, protože si nikdo nedovolil vydat článek
 - 1998 se Lamport nechal přemluvit a vydal speciální vydání v původním znění
 - 2001 Paxos Made Simple (přepsaný článek)
 - In 'plain' English
 - Podle Zavorala to není úplně jednoduché čtení
- Co vlastně znamená distribuovaný konsensus?
 - Typické využití je pro replikovaný stavový automat
 - Klient komunikuje se serverem a potřebuje, aby server byl absolutně spolehlivý
 - Uzly, které implementují funkčnost služby jsou replikované (neběží jeden uzel ale 3 uzly, nebo více podle úrovně služby)
 - Klient komunikuje s jedním uzlem, který je označen jako primární (leader)
 - Předá mu nějaký pokyn

- Pokyny postupně mění stav celého systému
- Cílem je tyto pokyny replikovat na jednotlivých uzlech
 - Všechny uzly budou mít stejnou posloupnost příkazů a stejný stav
- Distribuovaný stavový automat (jak funguje)
 - Klient kontaktuje primární repliku, ta si jeho pokyn zapíše do logu a pošle zprávu o změně stavu ostatním replikám
 - Ty si zapíší taky změnu do svého stavu a zpátky potvrdí primární replice, že zprávu dostali
 - Potom co primární replika dostane dostatečný počet shodných odpovědí (více jako $\frac{1}{2}$), tak záznam, který má v logu považuje za schválený a na základě něho si může upravit svůj vnitřní stav
 - Aplikace
 - Cluster management systémy (Borg u Googlu)
- Cílem protokolů je z pohledu klienta, aby klient komunikoval s celou množinou distribuovaných uzlů jako s jedním spolehlivých stavovým automatem
- Part-Time Parliament



- Vrátime se k Lamportově článku na úrovni pohádky
- Poslanci
 - Schvalují zákony
- Poslíčci
 - Doručují zprávy mezi poslanci
- Poslanci i poslíčci jsou nespolehliví
 - Mohou na čas i na trvalo odejít
- Poslíčci jsou důvěryhodní
 - Když doručí, tak nemění zprávu
- Poslíček může doručit zprávu několikrát
- Poslanci nejsou ochotni zůstat na celé jednání
- Poslanci si zapisují nesmazatelným inkoustem všechny zákony, o kterých jednali
 - (V informatice perzistentní datové úložiště)
- Paxos
 - Role (jednotky v rámci algoritmu, nejsou to uzly, jeden uzel má obvykle více rolí)
 - Client

- Zadává požadavky na změnu (primárně komunikuje s proposerem)
- Proposer
 - Navrhuje nové stavy
 - Řeší konflikty
 - Jsou replikované
 - V každém okamžiku je jeden proposer primární
 - Primary proposer = Leader
 - Řeší, které návrhy jsou právě platné
- Acceptor
 - Přijímají nebo odmítají návrhy
 - Jsou sdruženi v quoru (viz níže)
- Learner
 - Reprezentují repliky, které provedou změnu dle zadaného příkazu
- Quorum
 - Libovolná nadpoloviční podmnožina Acceptorů
 - Libovolná zpráva je platná až po příjmu celým quorem
- Ballot numbers
 - Pro detekci konfliktů a uspořádání příkazů jsou použité sekvenční čísla
 - Jedná se o dvojici $[num, process\ id]$
 - Hlasovací číslo, proposal number
 - $[n_1, p_1] > [n_2, p_2]$
 - $n_1 > n_2 \vee (n_1 = n_2 \wedge p_1 > p_2)$
 - Jednoznačně definované globální uspořádání
 - Volba Ballot
 - Každý uzel si eviduje poslední známý Ballot $[n, p]$
 - Pak proces p volí $[n + 1, p]$
 - Procesy akceptují pouze zprávy s **nejvyšším** Ballotem
- Každá role si udržuje (lokálně) několik údajů
 - Poslední známý Ballot (*LastBallot*)
 - (návrh)
 - Poslední akceptovaný Ballot (*AcceptBallot*)
 - (potvrzen quorem acceptorů)
 - Poslední akceptované hodnota (*AcceptValue*)
 - (udržuje se s posledním akceptovaným Ballotem)
- Základní Paxos protokol
 - Tohle je made simple of made simple
 - Rozdělen na dvě fáze
 - V první fázi jde hlavně o zvolení leadera
 - V druhé fázi jde o propagaci hodnoty
 - Každá fáze má dvě části
 - Každá část zhruba znamená zaslání zprávy
 - 1. Fáze

- Proposer
 - Zvolí si nový ballot $Ballot \leftarrow [Ballot.num + 1, myId]$
 - Pošle zprávu *Prepare* s *Ballot* všem acceptorům (stačí odeslat alespoň quoru)
 - Když acceptor přijme zprávu *Prepare*
 - Tak se podívá podle ballotu jestli to není zastaralá zpráva
→ když jo, tak mu vrátí *Reject*
 - Zpráva je zastaralá, pokud $bal < LastBallot$
 - Zapamatuje si ballot číslo ($LastBallot \leftarrow bal$)
 - Vrátí zpátky proposeru zprávu *Promise* s hodnotami $bal, AcceptBallot, AcceptValue$
 - Cílem je, aby proposer, který pošle několik zpráv *Prepare*, dostal alespoň quorum *Promisů* → prohlásí se za vítěze voleb a pokračuje druhou fází
- 2. fáze
- Akceptace hodnoty
 - Proposer potom, co dostane dostatečný počet *Promisů*, tak pošle zprávu *Accept* s *Ballot* a *myVal* všem acceptorům
 - Musí respektovat nejvyšší návrh
 - $myVal \leftarrow AcceptValue$ s nejvyšším *AcceptBallot*
 - Pokud žádný návrh od acceptorů na hodnotu není, tak může použít vlastní návrh na hodnotu
 - Acceptorů když dostanou takovou zprávu *Accept*
 - Zapamatují ballot i hodnotu v ní poslanou bal, val
 - Dále tyto hodnoty přepošlou všem proposerům a learnerům ve zprávě *Accepted*, pokud $bal \geq LastBallot$ (ballot přijmutý ve zprávě *Accept* není zastaralý)



Základní Paxos protokol

... made simple of made simple

• 1. fáze - volba Leadera

Proposer if isLeader

: **Prepare** $\text{Ballot} \leftarrow [\text{Ballot.num} + 1, \text{myId}]$
 send **Prepare**(Ballot) **Acceptorům**

podmínka: alespoň
Quoru Acceptorů

Acceptor receive **Prepare**(bal)

: **Promise** if $\text{bal} \geq \text{LastBallot}$

$\text{LastBallot} \leftarrow \text{bal}$

 send **Promise**(bal , **AcceptBallot**, **AcceptValue**)

neakceptuje
'starší' návrhy

poslední
akceptovaná hodnota

• 2. fáze - akceptace hodnoty

Proposer Quorum * receive **Promise**(bal , ab , av)

: **Accept** if $\exists \text{av} \neq \emptyset$ myVal $\leftarrow$ av s nejvyšším ab

else myVal $\leftarrow \oplus$

send **Accept**(Ballot , myVal) **Acceptorům**

opt.: else Reject

musí respektovat
nejvyšší návrh

jinak vlastní návrh

Acceptor receive **Accept**(bal , val)

: **Accepted** if $\text{bal} \geq \text{LastBallot}$

$\text{AcceptBallot} \leftarrow \text{bal}$; $\text{AcceptVal} \leftarrow \text{val}$

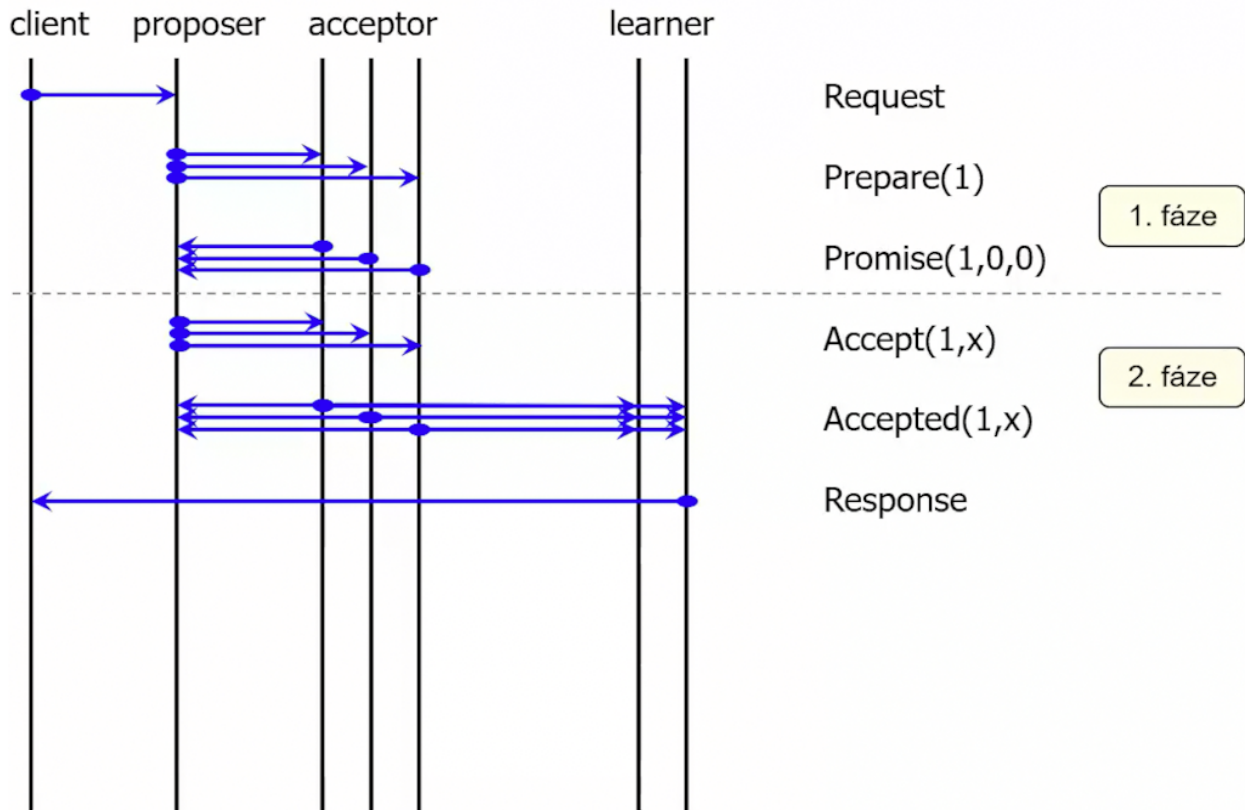
 send **Accepted**(bal , val) všem **Proposerům** a **Learnerům**

presistent
store

pouze nejnovější

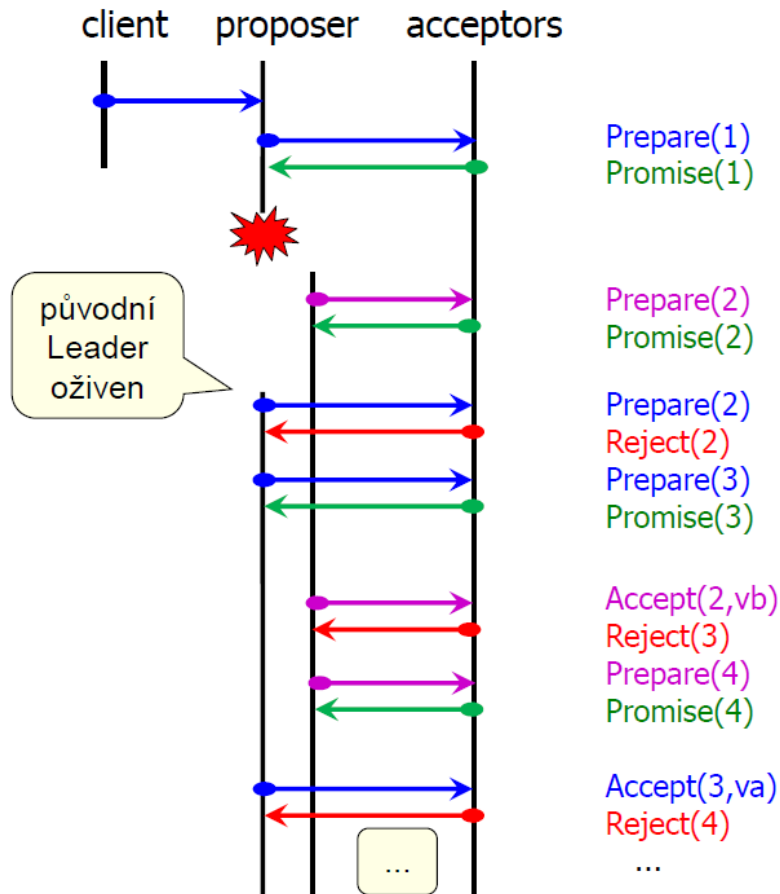
akceptace návrhu

Paxos - základní komunikace



■ Výpadky

- Když vypadne jeden acceptor tak se neděje vůbec nic
- Běh algoritmy by byl pozastaven, pokud acceptoři by nebyli schopni dát dohromady quorum
- Úplně stejně se chová výpadek learnera
 - Dokud je k dispozici alespoň jeden learner
- Když vypadne primární proposer
 - Zvolí se nový leader, který bude mít ballot o jedna větší
- Ještě je problém, kdyby se starý proposer, který dostal quorum *Promise* probudil potom co nový dostal také quorum *Promise* (ani jeden z nich nestihl odeslat *Accept*)
 - Viz obr. níže
 - Algoritmicky není zaručena konečnost při nejhorším možném způsobu prokládání
 - Konečnost není deterministická, ale dochází k randomizaci, takže to někdy prostě skončí
 - Prakticky to je OK



- Pozorování
 - Fáze 1
 - Neslouží k zasílání hodnot
 - Slouží k detekování leadera
 - Fáze 2
 - Leader navrhuje hodnoty s nejvyšším Ballotem z fáze 1
 - Aby celý protokol fungoval je nutné po dobu výpadku zachovat stav rolí
 - Perzistentní úložiště (nesmazatelný inkoust)
 - Jakým způsobem posílat *Accepted* zprávy learnerům
 - Dvě možnosti
 - Každý learner má přiděleného acceptora
 - Každý acceptor posílá *Accepted* zprávu každému learnerovy (kvadratická komunikační složitost)
- Existuje formální důkaz v TLA basic varianty Paxosu
- Je potřeba znát **princip algoritmu a chování při výpadcích jednotlivých rolí** na zkoušku i státnice
- **Použití v distribuovaném plánovači**
 - Implementujeme cluster management system
 - Umožňuje spouštění jobů

- Máme několik replik
 - Každá zastává všechny role (proposer, acceptor, learner)

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

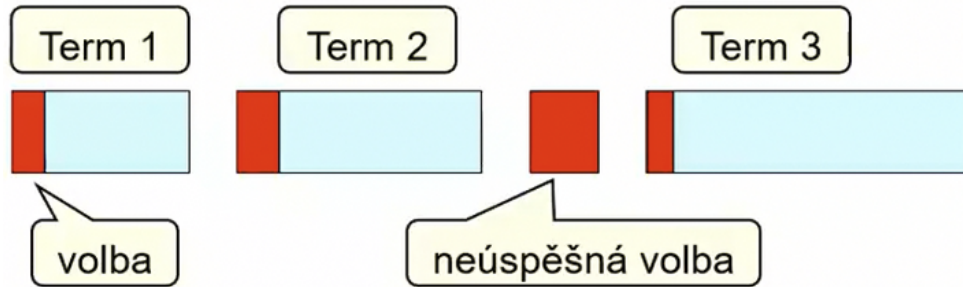
- Základní verze Paxosu je velmi málo použitelná → nepoužívaná
 - **Slouží ke konsensu pouze jedné hodnoty**
 - Pro jiného klienta musí **celý protokol** proběhnout znovu
- Vylepšení Paxosu
 - **Multi-Paxos**
 - Kontinuální opakování
 - **Základ typické implementace**
 - 2. fáze se opakuje tak dlouho, dokud nedojde k nějaké havárii
 - Založen na pozorování, že první fáze je nutná jenom v případě havárie
 - Rozhodli se v Googlu implementovat
 - Místo 1 až 2 týdnů jim to trvalo 3/4 roku
 - 2008 Paxos Made Live
 - článek o tom, jak je reálná implementace odlišná od abstraktního popisu
 - 2015 Paxos Made Moderately Complex
 - Článek o formálním popisu a verifikaci Multi-Paxosu
 - Cheap Paxos
 - Tolerance F havarujících uzlů při $F + 1$ uzlech celkově
 - Potřeba dalších F uzlů připravených v záloze
 - Fast Paxos
 - Optimalizace doručovaných zpráv
 - Generalized Paxos
 - Optimalizace pro komutativní operace
 - Byzantine Paxos
 - Rozšíření pro záškodníky
 - Ochrana před nekorektním chováním
 - Mezi fáze Accept a Accepted je přidána verifikační fáze, kde se zprávy ověřují
- Paxos - aplikace
 - Jedním z prvních byl Google
 - Google Chubby
 - Distribuovaný zamykací systém
 - Cluster management systémy

- Google Borg
 - Omega
 - Kubernetes
- Paxos - nedostatky
 - Téměř 20 let de-facto standardem pro distribuovaný konsensus
 - Těžko pochopitelný
 - Náročná implementace
 - Nekompletní popis
 - Vůbec není jasné jak implementovat
 - Reálná implementace neverifikovatelná

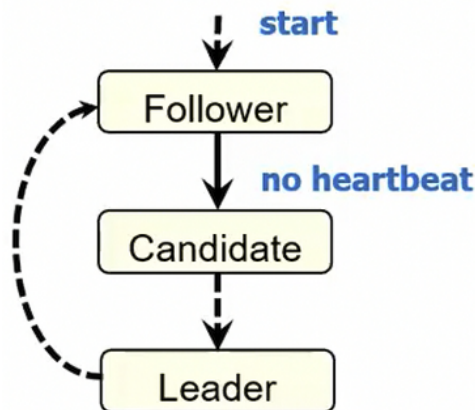
Video 11

- Přestože je Paxos přes 20 let standardem, je velmi kritizovaný pro jeho
 - Komplikovanost
 - Náročnou implementaci
 - Nekompletní popis (kompletní popis existuje pouze pro základní verzi)
- RAFT (Replicated And Fault-Tolerant Protocol)
 - 3x byl článek odmítnut
 - Stal se populárním, protože hlavní motivací pro jeho vytvoření byla
 - Srozumitelnost
 - (relativní) snadnost implementace
 - Implementace Paxosu jsou jednotky, rok po vydání RAFTu jich bylo stovky
 - Reálně používaný v celé řadě frameworků, ačkoliv se jedná o mladý protokol
 - Role
 - 3 základní, jedna z nich je dočasná
 - Leader
 - Primární replika, která navrhuje hodnoty
 - Follower
 - Akceptuje hodnoty
 - Jsou pouze pasivní (čekají na zprávu od leadera)
 - Hlídají, jestli pravidelně přichází heartbeat od leadera
 - Candidate
 - Dočasně při volbě leadera
 - Hlavní komponenty
 - Volba leadera
 - Alespoň jeden z followerů zdetekoval havárii (nedostal heartbeat)
 - Replikace logu
 - Normální operace
 - Leader přijímá požadavky od klientů, přidá je do vlastního logu
 - Log replikuje followerům
 - Udržování konzistence
 - Jednotlivé uzly se průběžně udržují konzistentní s tím co aktuální leader si udržuje ve svém logu

- Term
 - Základ synchronizace
 - Vždy začíná volbou leadera
 - Pokud je úspěšná, tak po celou dobu termu je leader jednoznačně zvolený a schválený followerama
 - Když se nepovede, tak term ani nezačne

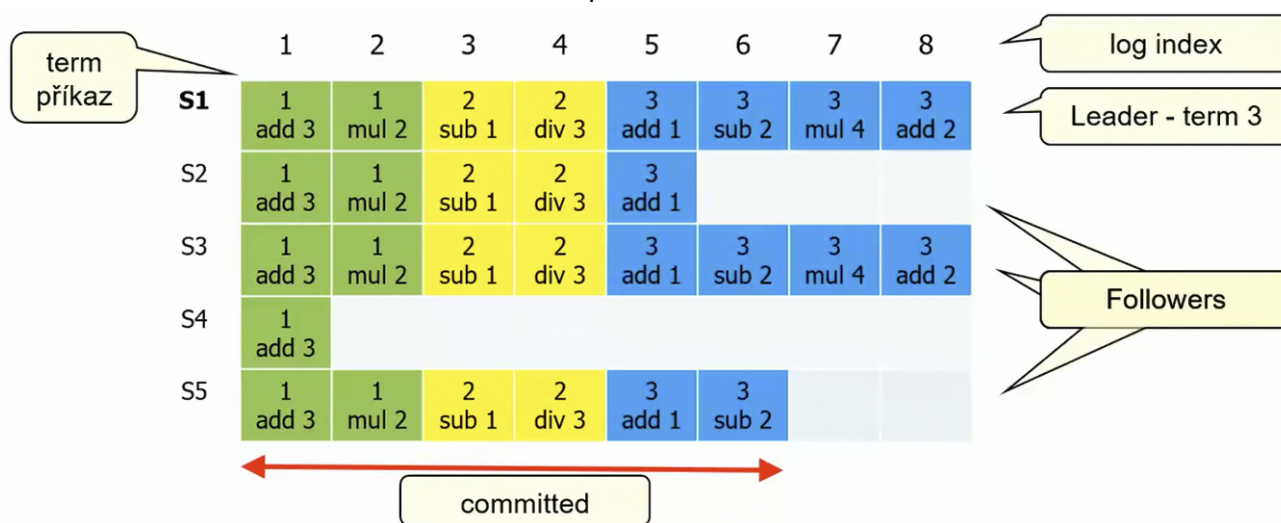


- V rámci termu mají všechny uzly, jednoznačně určeného leadera
 - Každý uzel si udržuje lokální *CurrentTerm*, když se odlišuje od jakékoliv zprávy, kterou dostane
 - $IncomingTerm > CurrentTerm$
 - Příjemce je zaostalý a proměškal volbu nového leadera
 - Potřebuje se zaktualizovat
 - $IncomingTerm < CurrentTerm$
 - Zprávu zahodí, protože to je zpráva od někoho opožděného
- Heartbeat
 - Leader periodicky zasílá zprávy followerům
 - Timeout → volba nového leadera
 - Obvykle v řádech stovek ms

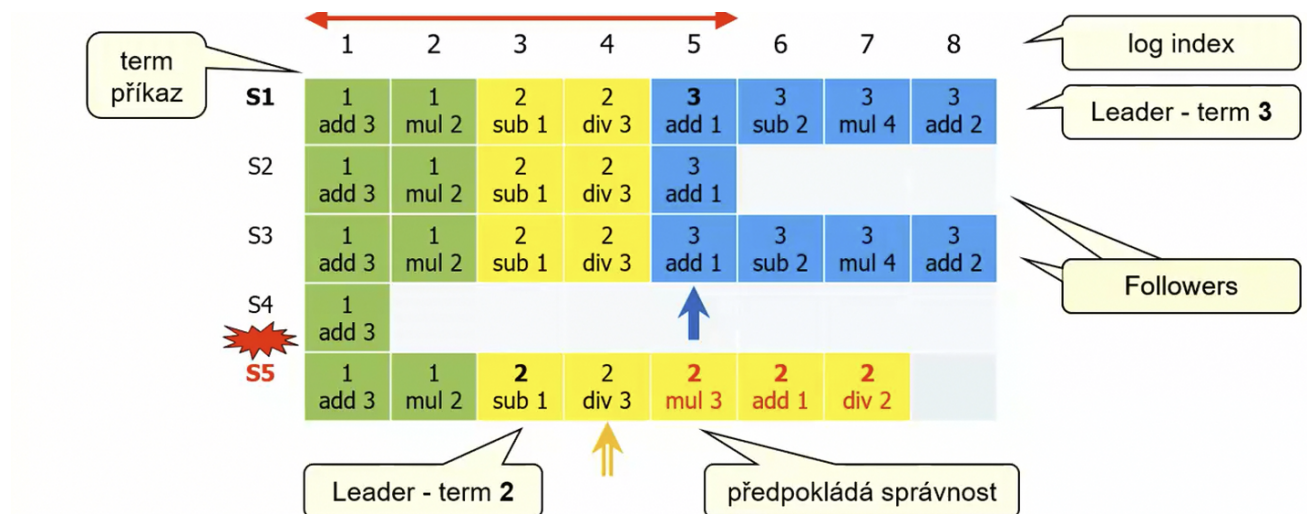


- Normální průběh RAFTu
 - Klient pošle zprávu leaderovi
 - Leader si zprávu zapíše do logu
 - Všem followerům pošle zprávu *AppendEntries* a čeká
 - Příkaz co má v logu neaplikuje, má ho jenom zapsaný

- Potom co na *AppendEntries* přijde odpověď od quora followerů (libovolná většina followerů), tak záznam z logu se stane commitovaným a aplikuje ho do svého lokálního stavového automatu
- Leader oznamuje followerům commitované záznamy v následných *AppendEntries*
- Follower po příjmu commit provede příkaz ve svém stavovém automatu
 - Postupně followeri následují přechody stavů leadera
- Havárie nebo zpoždění followera
 - Leader přepošle zprávu *AppendEntries* opožděnému followeru
- Při běžném provozu stačí jedna zpráva od leadera k followerům a její potvrzení pro jednu operaci, která je vyvolaná klientem
- Log
 - Pro vlastní synchronizaci
 - Každý uzel si eviduje svůj vlastní
 - Indexovaný
 - V každém záznamu je dvojice číslo termu, příkaz/data
 - Růst logu je nerovnoměrný
 - Persistentní
 - Musí se zotavit po havárii



- Udržování konzistence logu
 - Commit je pro udržování konzistence maximálně důležitý



Možné nekonzistence

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

Konzistence logů

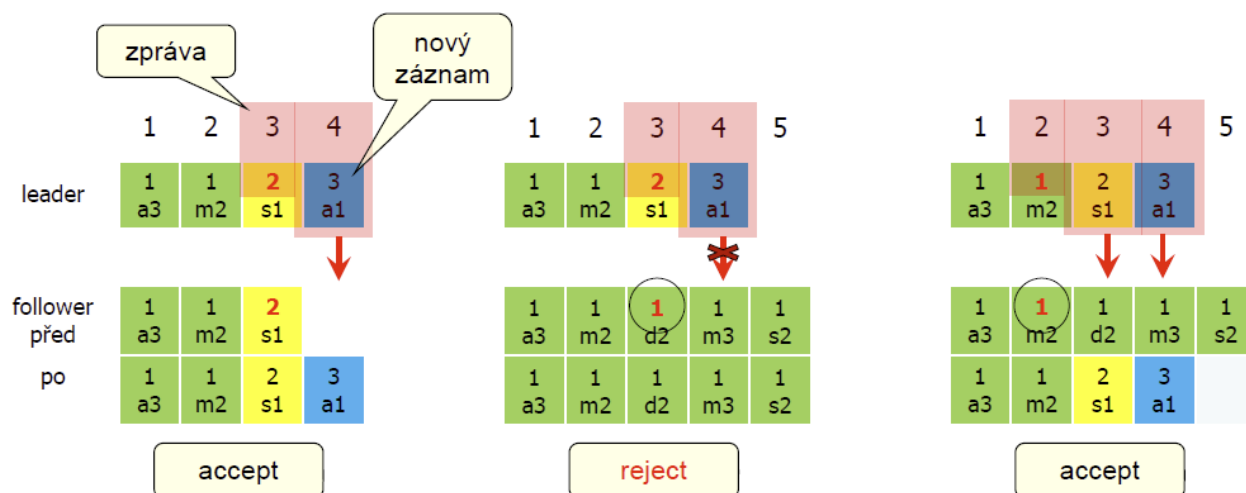
- záznam se stejným indexem a termem
 - vždy **shodná** hodnota
 - všechny **předcházející** hodnoty shodné
- commitovaný záznam
 - všechny předcházející hodnoty shodné
 - všichni **budoucí** Leaderi ho musí obsahovat

○ *AppendEntries* – jak přesně funguje

- Kromě nové hodnoty v ní leader posílá i index a term předcházejícího záznamu
- Když tuto zprávu přijme follower, tak zkontroluje, že index a term souhlasí s tím co si on eviduje, jinak ji odmítne
 - Tím dojde ke kontrole, že má s leaderem stejný log
- Leader potom pošle novou zprávu, kde bude více informací (předchozí index a termy), zkrátka to, co příjemce neeviduje

AppendEntries - kontrola konzistence

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

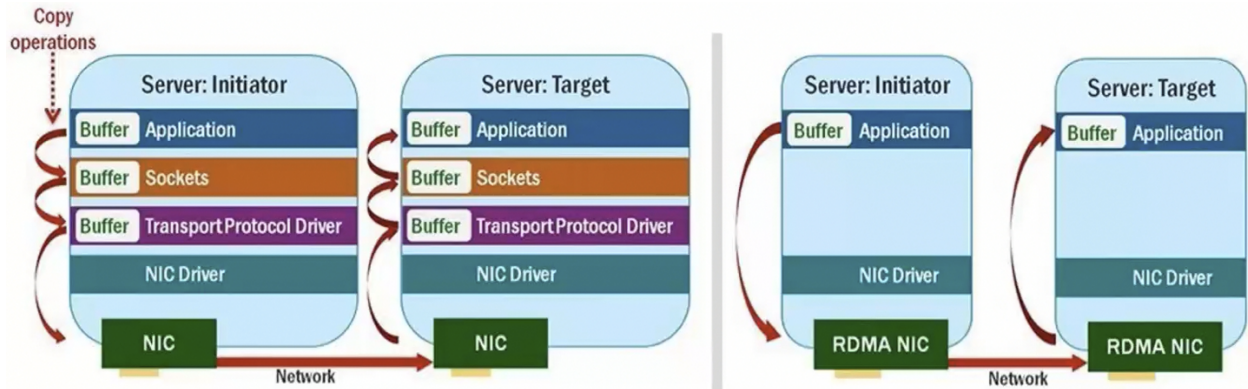


- Volba leadera
 - V případě, kdy dojde k havárii → dostatečnou dobu nedoručí heartbeat
 - Follower, který to zjistí se stane kandidátem
 - Zahájí se nová volba
 - Každý kandidát hlasuje sám pro sebe
 - Potom všem ostatním uzlům pošle zprávu *RequestVote*
 - Zpráva *RequestVote* obsahuje poslední zaznamenaný <index, term> kandidáta
 - Přijmutí zprávy *RequestVote*
 - Zpráva *RequestVote* obsahuje vyšší term než má příjemce → je opožděn (nepřišli mu nějaké zprávy)
 - Přesune se do role followera
 - Pošle hlas
 - Pokud je term žadatele **neaktuální** → odmítne dát hlas
 - Když přijde většinový počet hlasů (odpovědí na zprávu *RequestVote*), tak se stane leaderem
 - Pokud nedoručí k volbě za stanovený timeout tak nastane randomizovaná pauza
 - Všechny uzly si nastaví budík na čas $[T, 2T]$ (v řádu stovek ms)
- Další protokoly
 - Existují i další protokoly kromě Paxosu a RAFTu
 - ZAB (Zookeeper Atomic Broadcast Protocol)
 - Založen na principech Paxosu

- Technické detaily jsou odlišné
 - Viewstamped Replication
 - Ještě horší než Paxos
- Využití konsensu
 - **Replikace služeb**
 - Základní motivace
 - Replikace logu
 - V DB a jiných distribuovaných úložištích
 - Synchronizační služby
 - Tradičně se to dělalo pomocí zámků
 - V Googlu postavili na Paxosu distribuovaný zámkový systém Chubby
 - Bariérová orchestrace
 - Když děláme výpočet na rozsáhlých datech
 - Slouží k synchronizace jednotlivých uzlů mezi fázemi výpočtu
 - Configuration management
 - Distribuovaná vysoce spolehlivá služba pro ukládání Key/Value párů
 - **Obecně jsou algoritmy pro konsensus nejpoužívanějším a nejviditelnějším výsledkem teorie distribuovaných systémů**

Video 12

- Dnes distribuovaná sdílená paměť
- Paralelní výpočty
 - Multiprocesory
 - Malá škálovatelnost, hw náročné, drahé
 - Máme sdílenou paměť
 - Multicomputery
 - (typický distribuovaný systém, kde uzly jsou nezávislé)
 - Snadno dostupné
 - Snadno implementovatelné
 - Komunikace a synchronizace není prakticky dobře zvládnutá
 - Nemáme distribuovanou paměť → implementace náročnějších paralelních výpočtů není jednoduchá
 - Museli bychom všechno řešit přes nějaké zprávy
 - Pokus o řešení → distribuovaná sdílená paměť
 - 1986 Li & Hudak – DSM
 - Jeden z prvních článků
- Hlavně v 90. letech byl velký vývoj v oblasti distribuované sdílené paměti
- Nic prakticky použitelné
 - Hlavní překážka byla výkonnost
- Dnes jsou sítě již mnohem výkonnější
 - Masivním nárůstem cloudu a clusterových infrastruktur



- Výkonný síťový HW, který podporuje RDMA
 - Remote Direct Memory Access
 - Síťová karta přímo umožní zapsat data do výsledného bufferu, kde jsou očekávané
- Přenos po síti je dnes o několik řádů rychlejší
- Latence je mnohem lepší
- Dnes distribuovaná sdílená paměť narůstá na popularitě
 - Součástí distribuovaných a cloudových frameworků
- Můžeme rozlišit tři úrovně, kde je distribuovaná paměť implementována
 - Distribuované stránkování
 - Distribuované sdílené proměnné
 - Distribuované objekty
 - Pro objektové frameworky
- Konzistenční model
 - Specifikace, co implementace musí splňovat vzhledem k operacím čtení a zápis
 - Co se přesně má stát při zápisu a čtení
 - Primitiva
 - Read, Write
 - Sdílený adresový prostor

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 (někdy též atomická konzistence)
 - První konzistenční model, o kterém si budeme povídat

- Jakékoliv čtení z adresy x vrátí hodnotu uloženou při posledním zápisu na adresu x
 - Tradičně intuitivně zajištěna na jednoprocessorových systémech
- Zajišťuje absolutní časové uspořádání
- Je nejsilnější ze všech modelů, o kterých si budeme povídat
- Všechny zápisy jsou okamžitě všude viditelné
- Podmínka: **musí existovat přesný globální čas**
- Ideální pro programování → intuitivně ho očekáváme
- V distribuovaném systému nedosažitelné

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

vytiskne vždy 2

- Pro striktně konzistentní model platí následující

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

striktní konzistence

- Pro paměť, která není striktně konzistentní platí následující

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

paměť, která není striktně konzistentní

- Sekvenční konzistence
 - O něco slabší model
 - Všechny operace na jednom uzlu jsou všemi uzly v distribuovaném systému vnímané ve stejném pořadí
 - Všechny operace jsou seřazeny do nějakého jednoho rozvrhu, který všechny uzly respektují
 - Distribuovaný systém se v zásadě chová jako centralizovaný systém
 - **Oproti striktnímu modelu není v sekvenční konzistenci nutná synchronizace s fyzickým časem**
 - Když se na dvou různých uzlech provede zhruba ve stejný fyzický čas nějaká operace, tak implementace jednu z těchto operací prohlásí za dřívější (z hlediska sekvenčního rozvrhu) a druhou za pozdější a je jedno, která byla fyzicky provedena dříve
 - Na všech uzlech je ale dodržován stejný rozvrh operací
 - Stále se v něm ale dobře programuje (moc nepoznáme rozdíl oproti striktní konzistenci)

- Implementace distribuované sdílené paměti v tomto modelu je v celku jednoduchá → použitelné
- Velikost zpoždění samotným modelem **není nijak zaručena**
 - Implementace tohoto modelu může pozdržet některé zápisy o několik sekund/minut, to je nepříjemné chování, které může být problémové
- Později uvidíme vylepšení tohoto modelu o synchronizaci s fyzickým časem
- Níže je klasický případ race condition, který může nastat v tomto modelu

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

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

dvakrát spuštěný tentýž program – může dát různé výsledky

- Podíváme se na jednoduchý prográmek a jak se na něm sekvenční konzistence chová

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

šest instrukcí - $6!=720$ možných uspořádání
pouze $3*(5!/4)=90$ nenarušuje konzistenci

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

Výstup: (skutečný výstup, např. na obrazovce)

001011	101011	010111	111111
--------	--------	--------	--------

Signatura: (výstupy jednotlivých procesů v pevném pořadí)

001011	101011	110101	111111
--------	--------	--------	--------

- Signatura se používá pro zkoumání distribuovaných rozvrhů
- Signatura
 - **Výstupy jednotlivých procesů v pevném pořadí**
 - Pomůže ověřit, jestli výstup odpovídá sekvenční konzistenci
 - Konkrétní proložení instrukcí procesů, a tím i pořadí paměťových referencí
 - Např. 000000 neodpovídá žádnému sekvenčnímu rozvrhu
 - Každý rozvrh musí dodržovat to, že se nejdříve zapíšu hodnoty a pak se přečtou

- 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

- První dvě nuly říkají, že proces P1 vytisknul nuly
- Na začátku do proměnné a P1 zapsal jedničku a potom vytisknul dvě nuly
- 10 znamená, že druhý proces vytisknul jedničku pro proměnnou a a 0 pro proměnnou c
- Čtení c muselo být rozvrženo před zápisem $c = 1$
- 01 znamená, že proces c přečetl 0 z a , 1 z b
- Formálně je dokázáno, že **výkonnost** sekvenční konzistence **není** příliš **vysoká**
 - Je možné optimalizovat buď pro čtení, nebo zápis
 - Vždy ale $r + w \geq t$
 - r ... čas čtení
 - w ... čas zápisu
 - t ... čas přenosu zprávy
 - Obvykle se optimalizuje pro operaci čtení (používá se mnohem více)
- Externí konzistence (externí jako synchronizace s externím časem)
 - Rozšíření sekvenční konzistence
 - Vhodné pro prostředí, kde nechceme obrovské zpoždění instrukcí oproti fyzickému provádění
 - Synchronizace s externím časem
 - Na moderních clusterových systémech, kde synchronizace hodin je přesná, zpoždění nabývá v sekvenčním rozvrhu oproti fyzickému času v řádu milisekund
 - Používá se pro distribuované databáze
 - Pro jakékoliv dvě transakce T_1, T_2
 - Když T_2 začne operaci “dostatečně později” (tj. po tom, co T_1 ukončí operaci commit), tak potom logický čas $T_2 >$ logický čas T_1
- Kauzální konzistence
 - (Víceméně se hledali modely, které jsou výkonnější než sekvenční konzistence)
 - *Kauzálně závislé zápisy musí být viděny všemi uzly ve stejném pořadí. Konkurenční zápisy mohou být viděny v různém pořadí.*
 - Oproti sekvenčnímu modelu není zapotřebí, aby všechny uzly dodržovaly jeden stejný globální rozvrh
 - Všechny rozvrhy ale musí dodržovat kauzalitu
 - **Zápis** hodnoty do paměťového místa ~ **odeslání** zprávy
 - **Čtení** hodnoty z paměťového místa ~ **doručení** zprávy
 - Kauzálně závislé zápisy

- Pokud $W(x)a$ a (druhý proces) $R(x) W(y)b$, pak $W(y)b$ je kauzálně závislý na $W(x)a$ (přes $R(x)$)
- Implementačně složitější
 - Vyžaduje udržování grafu závislostí zápisů na čtení
- Potencionálně mnohem výkonnější → nemáme jeden globální rozvrh

P1: $W(x)1$

P2: $W(x)2$

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

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

kauzálně konzistentní rozvrh

P1: $W(x)1$

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

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

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

není kauzálně konzistentní

- PRAM (Pipelined RAM) konzistence
 - Zápisy prováděné jedním uzlem jsou viděny ostatními uzly v pořadí provádění. Zápisy různých uzlů mohou být viděny různými uzly různě.
 - Ukázalo se, že u kauzální konzistence ten nárůst výkonu není tak drastický a implementace je o hodně složitější
 - Vymýšleli se další modely
 - Jednoduchý na implementaci
 - Žádný globální rozvrh neexistuje
 - Zápisy různých uzlů mohou být viděny v různém pořadí
 - Je nutné dodržovat pořadí zápisů z jednoho zdroje
 - Respektuje se lokální rozvrh instrukcí, ale proložení není synchronizováno
 - Je možné to použít pro **sdílení** paměti ale ne pro synchronizační účely
- Slow Memory
 - Zápisy jedním procesem do jednoho místa musí být viděny ve stejném pořadí
 - Neřeší synchronizace mezi uzly
 - Zajišťuje uspořádání jenom do jednoho paměťového místa
- Existují ještě slabší modely
 - Eventual Consistency
 - "Když je příležitost, tak je to konzistentní"

- Všechny předchozí konzistenční modely jsou určeny pro implementaci na úrovni virtuální paměti, kde framework nemá k dispozici žádné další informace a musí propagovat všechny zápisy všem procesům
- Málo efektivní
 - Ne všechny uzly musí sledovat všechny zápisy (natož jejich pořadí)
- Efektivněji je možné implementovat distribuovanou paměť na vyšší úrovni → knihovny a frameworky
 - Značnou nevýhodou je ztráta transparentnosti narozdíl od distribuovaného stránkování
- Konzistenční modely se synchronizační proměnnou
 - Speciální druh proměnné → synchronizační proměnná
 - Pouze používaná pro synchronizační účely
 - *S* ... synchronizace
 - *Acq* ... vstup do kritické sekce
 - *Rel* ... výstup z kritické sekce
- Weak consistency
 - Sdílená data jsou konzistentní po synchronizaci
 - 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 před dokončením všech předchozích přístupů 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: před čtením dat jsou dokončeny všechny předchozí přístupy k SP

♦ provedením synchronizace před čtením se zajistí aktuální verze dat

- Není nutné propagovat všechny zápisy
- Model neumožňuje rozlišit vstup a výstup z/do kritické sekce
- Release consistency (výstupní konzistence)
 - Sdílená data jsou konzistentní po opuštění kritické sekce
 - 1. *Před přístupem k datům musí být úspěšně ukončeny předchozí *Acq()* procesu*
 - 2. *Před provedením *Rel()* musí být ukončeny všechny předchozí zápisy i čtení prováděné procesem*
 - 3. **Acq()* a *Rel()* **musí být PRAM konzistentní***
 - Po *Acq()* jsou všechny lokální kopie aktuální
 - Po *Rel()* jsou propagovány změny ostatním procesům
 - Při správném párování *Acq()* a *Rel()* je výsledek výpočtu ekvivalentní sekvenčně konzistentní paměti
 - Za cenu ztráty transparentnosti je přístup k distribuované sdílené paměti ekvivalentní k sekvenčně konzistentnímu modelu

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

P2:

Acq R(x)2 **Rel**

P3:

R(x)1

release consistency

bez přístupu k SP
libovolně stará hodnota

- Možnosti implementace
 - Eager release consistency
 - Po *Rel()* se propagují změny všem procesům
 - Optimalizace přístupové doby
 - Náročnější na komunikaci po síti
 - Optimalizace přístupové doby
 - Lazy release consistency
 - Po *Rel()* se nic nepropaguje, až při *Acq()* jiného procesu
 - Optimalizace síťového provozu
 - Pomalejší přístupová doba
 - Entry consistency (vstupní konzistence)
 - Sdílená data vázaná na kritickou sekci jsou konzistentní při vstupu do kritické sekce
 - Zavádí exkluzivní a neexkluzivní přístup k synchronizační proměnné
 - Operace *Acq()* a *Rel()* **nejsou globální ale jsou lokální pro paměťový region**
 - Co je paměťový region je implementačně závislé (stránka, objekt, ...)
 - 1. *Acq()* k SP není povolen, dokud nebyly provedeny všechny aktualizace sdílených dat
 - 2. Exkluzivní přístup k SP (zápis) je povolen pouze pokud 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 předchozího vlastníka SP
- ♦ sdílená data jsou vázána na SP, při přístupu se synchronizují pouze tato data
 - ♦ přístup k datům a SP může být exkluzivní (RW) nebo neexkluzivní (RO)
 - ♦ každá SP má vlastníka - proces, který k ní naposledy přistupoval
 - ♦ vlastník může opakovaně 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

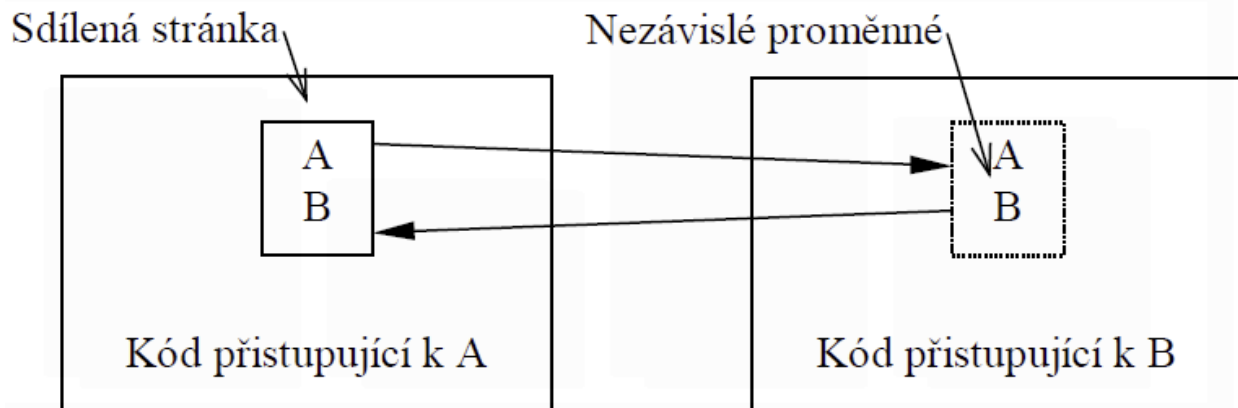
Nezávislá data x a y

- Jednotlivé regiony spolu nijak nekomunikují
- Je možné přistupovat k různým paměťovým oblastem, aniž by se ovlivňovaly
 - Jeden zámek na všechny data = zbytečné prodlevy pro synchronizaci dat, které spolu nesouvisí
- Dáme logicky související data do jednoho paměťového regionu, ten je chráněn pomocí *Acq()* a *Rel()*
- Shrnutí konzistenčních modelů
 - Ukázali jsme si celou řadu modelů
 - 2 třídy
 - Bez SP
 - Pro low level memory management na úrovni virtuální paměti
 - Transparentní
 - S SP
 - Na vyšší úrovni na úrovni middlewaru, frameworků a knihoven
 - Ztratíme transparentnost
 - Vyšší výkonnost

	Konzistence	Vlastnosti
bez SP	Strict	Absolutní časové uspořádání
	Sequential	Všechny události jsou vidět ve stejném pořadí, možná synchronizace s ext. časem
	Causal	Kauzálně vázané události jsou vidět ve stejném pořadí
	PRAM	Události jednoho uzlu jsou vidět ve stejném pořadí
	Slow mem	Zápisy jednoho uzlu na jedno místo jsou vidět ve stejném pořadí
s SP	Weak	Sdílená data jsou konzistentní po synchronizaci
	Release	Sdílená data jsou konzistentní po opuštění kritické sekce
	Entry	Sdílená data vázaná na kritickou sekci jsou konzistentní při vstupu do kritické sekce

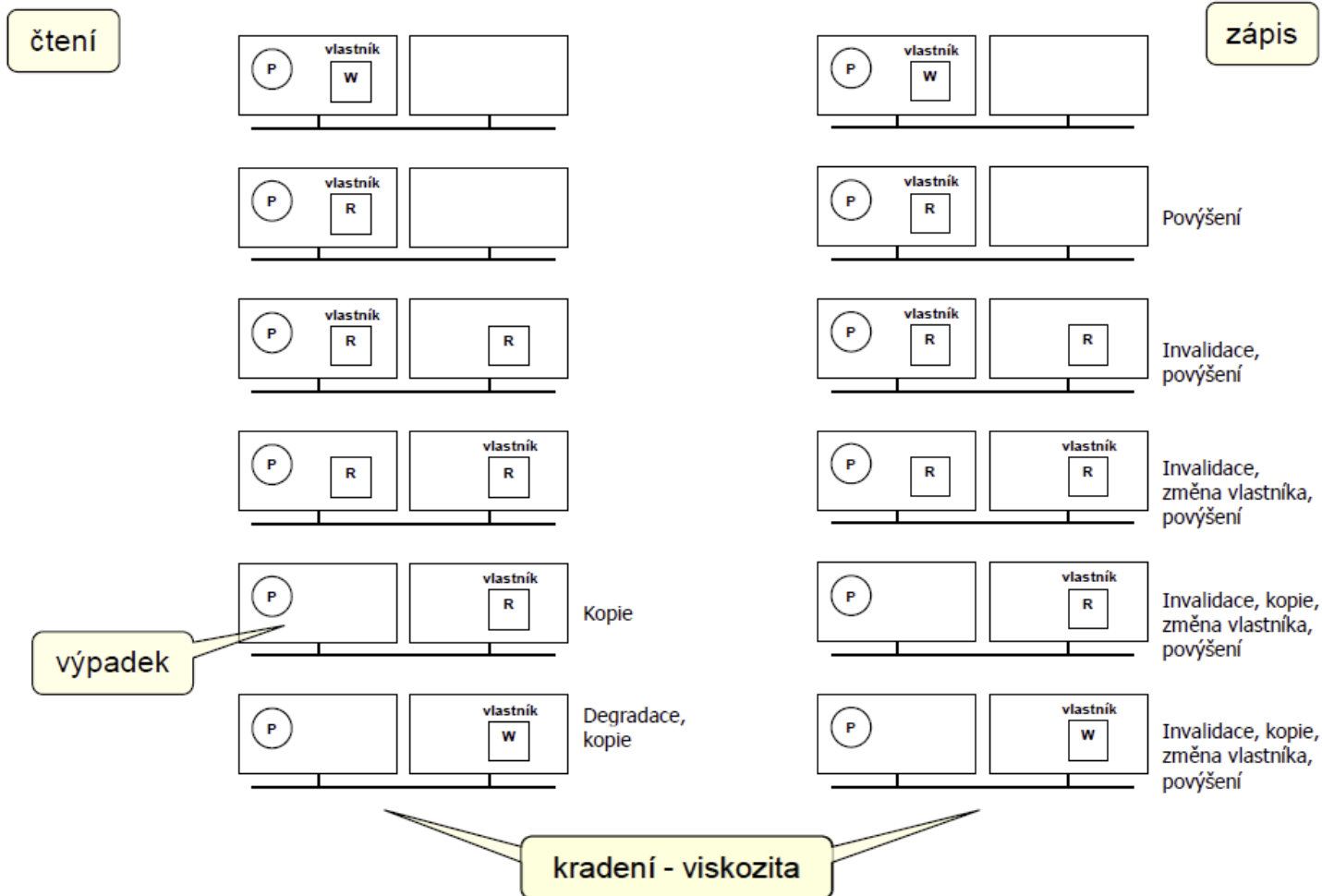
- Nyní se budeme věnovat možnostem implementace distribuované sdílené paměti a distribuovanému stránkování
- Distribuované stránkování
 - Obdoba virtuální paměti
 - Operace čtení a zápisu jsou pro systém transparentní
 - Když proces přistupuje k nenamapované stránce nebo ke stránce, která je pouze read-only a chce do ní psát, tak dojde k přerušení → obsluha
 - V obsluze se děje celá implementace distribuovaného stránkování (synchronizace, načtení a koordinace s ostatními uzly)
 - Problémy
 - Jak nalézt stránku
 - Stránka může migrovat mezi uzly, data se musí nalézt
 - Navíc pro některé konzistenční modely mohou mít některé uzly staré verze této stránky
 - Centralizovaný manager

- Jeden centralizovaný prvek spravuje informace o tom, kde se nachází stránky
- Opět jsme se dostali k centralizovanému prvku → je nutné si zde uvědomit, že sdílené stránkování budeme pravděpodobně požadovat u menšího počtu uzlů, které jsou velmi těsně spřažené
- Nevhodné pro rozsáhlé systémy
- Replikovaný manager
 - Pokud by nám centralizovaný nestačil
 - Je jich více a každá z replik obsahuje jednu část ze sdíleného virtuálního adresového prostoru
 - Jak najít vhodného managera?
 - Typicky jednoduché
 - Např. Spodní 4 bity adresy stránky mohou sloužit jako index manageru, který je za stránku zodpovědný
 - Nebo je možné použít hash stránky
- Broadcast
 - Opět jsme na lokálním malém prostředí
 - Obejdeme se bez managera
- Správa kopií
 - Co dělat s kopiemi při čtení / zápisu
 - Copyset
 - Každý vlastník stránky udržuje množinu uzlů, která má tuto stránky také namapovanou
 - Když je potřeba stránku upgradovat (došlo k zápisu) → všem, co mají kopii pošlu zprávu (typicky invalidační)
 - Broadcast
 - Když není copyset implementovaný
- Uvolňování stránek
 - Jakou stránku uvolnit?
 - Pokud uzel má stránku, která není vlastněná a potřebuje uvolnit místo, tak stránku bez jakékoliv synchronizace a komunikaci ji zahodí
 - Pokud jsme vlastníkem a víme (např. pomocí copysetu), že je stránka replikovaná, tak můžeme přenést vlastnictví na jiný uzel a stránku zahodit
 - LRU strategie
- Falešné sdílení
 - Typický problém v distribuovaném prostředí
 - Dvě různé sady dat jsou čistě náhodou umístěny na stejné stránce



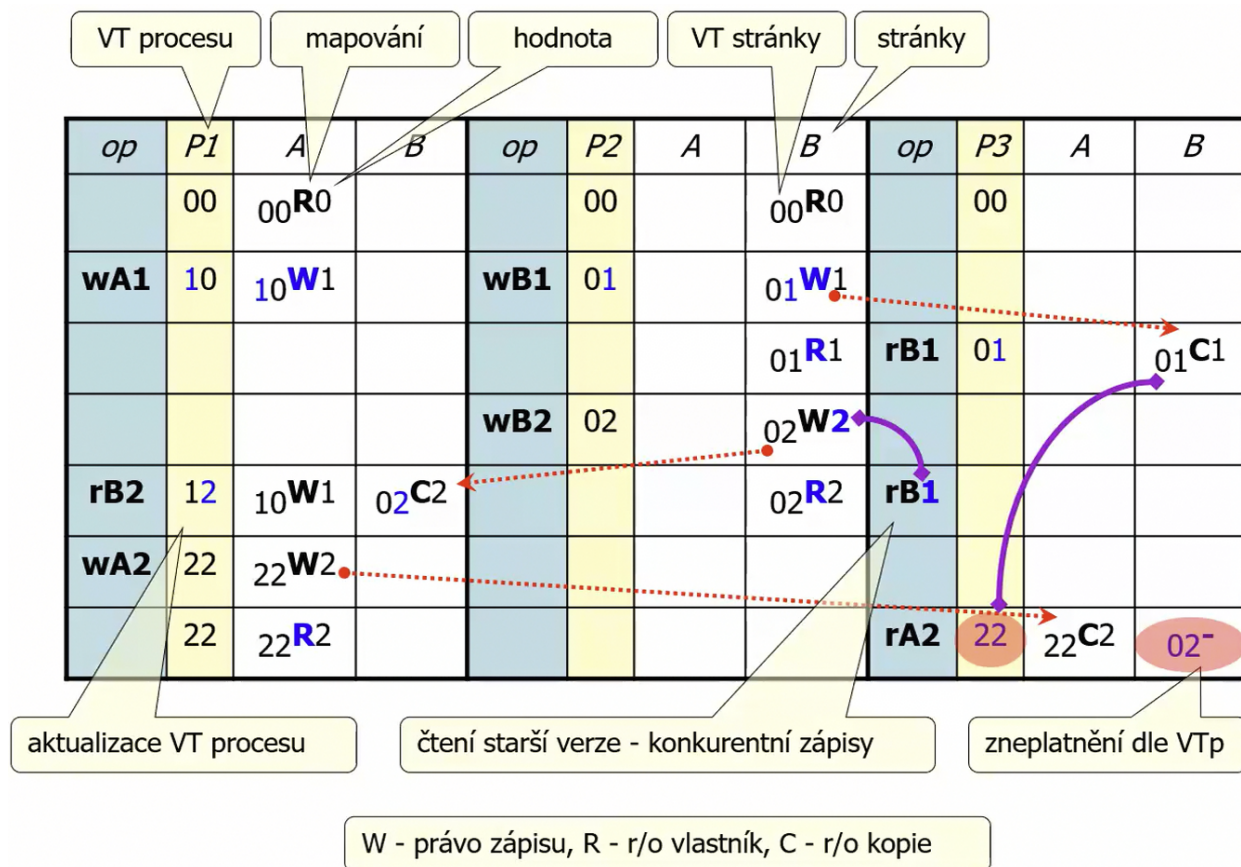
- Jde částečně řešit prevencí
 - Kompilátor a linker mohou umístit data, která spolu nijak nesouvisí na dvou různých stránkách
- Sekvenčně konzistentní distribuované stránkování
 - V rámci jednoho distribuovaného systému je jeden globální rozvrh všech operací čtení a zápisu
 - Stránka je namapována na zápis a vlastník je přímo ten uzel, na kterém běží proces, který chce přistoupit ke stránce
 - Nejjednodušší
 - Proces provede instrukci zápisu
 - Bez interakce s frameworkem
 - Stránka je namapována na čtení
 - Operace čtení proběhne stejně jako v předchozím případě
 - Je jedno, kdo je vlastníkem a kde všude ta stránku je
 - Operace zápisu vyvolá výpadek
 - Pokud jsem vlastníkem
 - Framework povýší stránku na zápis
 - Pokud nejsem vlastníkem stránky → pro zápis potřebuji vlastnictví
 - Vlastnictví se musí přenést
 - Často zahrnuje komunikaci s nějakým managerem stránek
 - Pak zbytek funguje stejně jako zápis do read-only vlastněných stránek
 - Stránka není namapována
 - Synchronizace se ujme framework
 - V okamžiku, kdy najdu vlastníka
 - Pokud vlastník ji má jenom na čtení, tak požádám o kopii
 - Pro čtení ze stránky stačí
 - Read-only kopií může být kolik chceme
 - Pokud vlastník ji má namapovanou na zápis
 - Vlastníkovi se stránka přemapuje na read-only
 - Chceme zajistit, aby další zápisy vyvolaly framework a my jsme se o nich dozvěděli
 - Zbytek stejně jako předchozí

- Případný další zápis od vlastníka vyvolá obsluhu
- Stránka je vlastněna na uzlu, ale je pouze read-only a chci zapisovat, nikdo další tuto stránku nemá
 - Dojde k výpadku a přerušení, ale obsluha pouze povýší tuto stránku na zápis
- Stránka je vlastněna na uzlu, ale je pouze read-only a chci zapisovat, existuje uzel, který má tuto stránku
 - Než je stránka přemapována na zápis, tak si vlastník počká na potvrzení od uzlů majících tuto stránku, že stránku zahodily
 - Dojde k povýšení frameworkem a zápisu
- Je vidět, že pro provedení jedné instrukce, je zapotřebí nákladná analýza
 - Řešení toho problému “kradení stránek” je viskozita
- Viskozita
 - Operace čtení i zápisu jsou velice často prováděné v cyklu
 - Co kdyby jeden uzel chtěl ze stránky číst a druhý do ní zapisovat?
 - Potom výkonost celého systému může jít rapidně dolů → na jednu operaci bude potřeba obrovská režie
 - Viskozita využívá toho, že sekvenčně konzistentní model nemusí přesně odpovídat fyzickému provádění instrukcí
 - V okamžiku, kdy nějaká stránka je na uzlu namapována, tak tam zůstane ještě nějakou dobu navíc (i přes to, že by ji nějaký uzel chtěl odmapovat)



- Kauzálně konzistentní distribuované stránkování
 - Implementace je o něco komplikovanější
 - Je potřeba evidovat **graf kauzálních závislostí** mezi procesy a stránkami
 - Budeme používat vektorové hodiny (jednotka kauzality jsou stránky)
 - VT_S ... vektorové hodiny pro stránky
 - Každá stránka si eviduje na kterých jiných stránkách je kauzálně závislá
 - VT_P ... vektorové hodiny pro procesy
 - Ze kterých verzí stránek znám data
 - $VT[i]$... stránka i
 - Oboje tyto vektorové hodiny slouží k synchronizace verzí stránek, aby kauzálně vázané zápisy byly založené na datech, které jim kauzálně předchází
 - Základem synchronizace je **výpadek stránky**
 - Čtení do (jakkoliv) namapované stránky stejně jako zápis do stránky namapované pro zápis jde mimo framework
 - Výpadek stránky – přenos dat
 - $VT_P = \max(VT_S, VT_P)$

- Zápis do stránky
 - Provedeme zápis do stránky s verzí VT_S
 - Stránka je odmapována a při namapování se inkrementuje odpovídající složka $VT_S = inc(VT_P)$
- Po aktualizaci VT_P se zneplatní stránky i : $VT_{S_i}[i] < VT_P[i]$
 - Když příjemce zjistí, že má u sebe stránku starší než jsou stránky, na kterých kauzálně závisí obsah právě přijaté stránky, tak si tyto stránky zneplatní
- Problémy
 - **Velká prostorová režie**
 - Na 1MB (256 stránek) sdíleného prostoru potřebujeme 512B na jednu stránku
 - Jakým způsobem propagovat konkurenční zápisy
 - Každý uzel si udržuje starou verzi stránky do té doby než je vynucena její invalidace jinou stránkou
 - Řešení pomocí timeoutů nebo explicitních synchronizačních primitiv



- NOTE: Šíleně moc toho říkal k tomu obrázku, ale nedá se to rozumně zapsat
- Distribuované sdílené proměnné
 - Implementace na úrovni knihoven je výrazně jednodušší

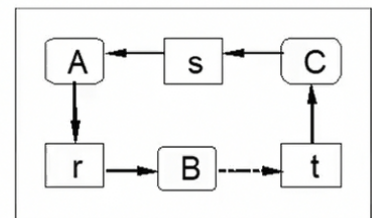
- Na úrovni stránek jsme měli k dispozici pouze výpadek stránky a její obsluhu
 - Výhody
 - Potenciálně lepší výkonnost
 - Framework má přesnější informace o operacích na datech
 - Nevýhoda
 - Menší míra transparentnosti
- Distribuované objekty
 - [NSWI080](#)
 - Enkapsulace objektů

Video 13

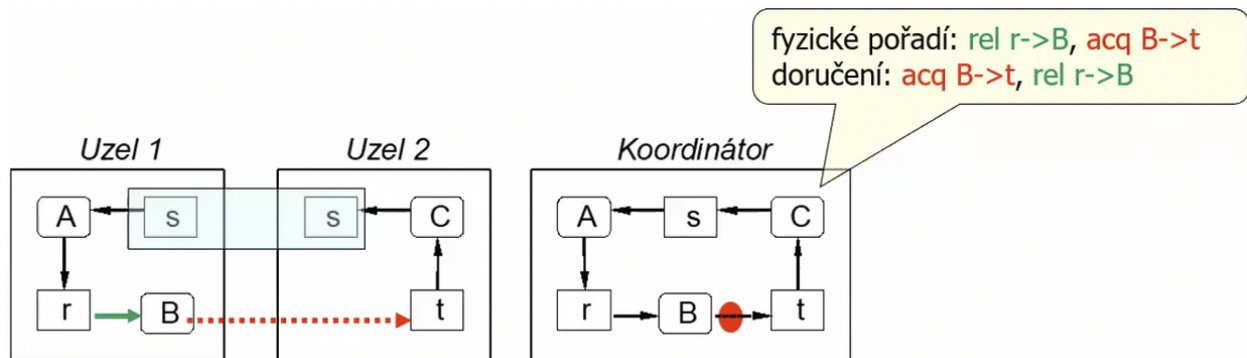
- Dnes se budeme zabývat správou prostředků
 - Konkrétně detekce distribuovaných deadlocků
- Každý prostředek má typicky svého správce
 - Byli teoretické pokusy o distribuovanou správu prostředků → stejný problém jako u distribuovaného vyloučení procesů
- Detekce deadlocků se redukuje na wait-for-grafu
- V distribuovaném systému nikdo nezná přesný aktuální stav → detekce, že došlo k deadlocku je mnohem komplikovanější
- Časté řešení je pštrosí algoritmus
 - Deadlocky ignorujeme
 - Předpokládáme, že případné výskyty daného problému přinesou nižší náklady než snaha problému předejít
- Ukážeme si několik algoritmů na detekci deadlocků, všechny jsou založené na wait-for-grafu
 - Když algoritmus detekuje orientovanou kružnici, tak ohlásí deadlock
- Wait-for-graf
 - Eviduje závislosti mezi procesy

♦ WFG – Wait-For-Graph

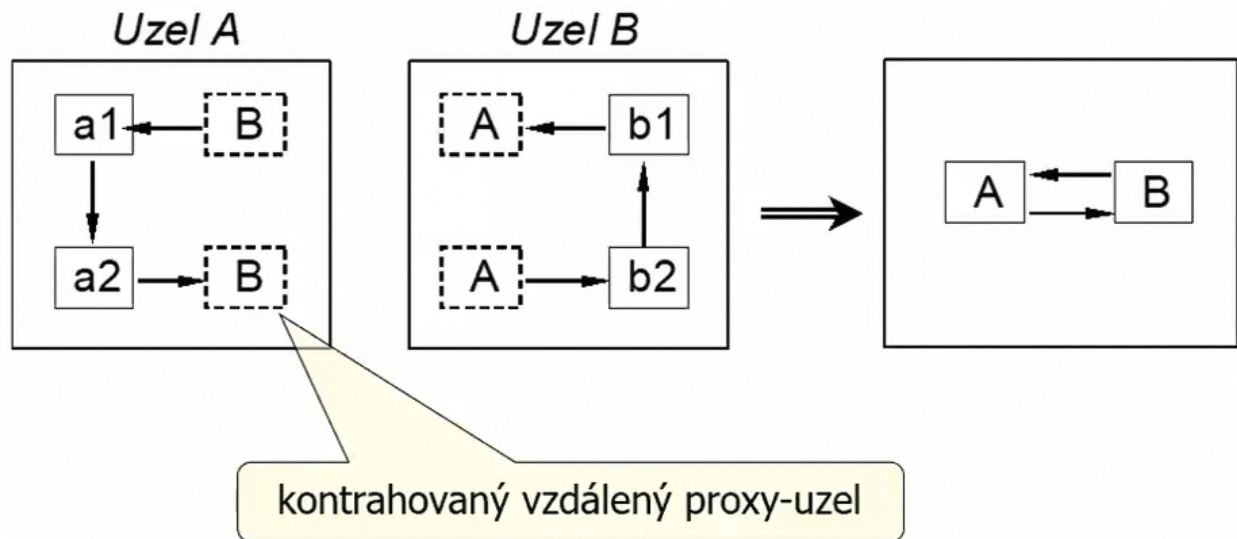
- orientovaný graf (procesů, transakcí)
- $P1 \rightarrow P2$ proces P1 je blokován procesem P2
- $P \rightarrow R$ proces P žádá o prostředek R
- $R \rightarrow P$ proces P drží prostředek R
- orientovaná kružnice – deadlock
- Algoritmy detekce deadlocku (historie)
 - Kdysi vyšla hromada algoritmů, o kterých v dalších článku vyšlo, že nejsou zcela korektní
- Korektnost algoritmu detekce deadlocku
 - 1. každý existující deadlock je v konečném čase detekován
 - 2. detekovaný deadlock musí existovat
- Metody konstrukce WFG



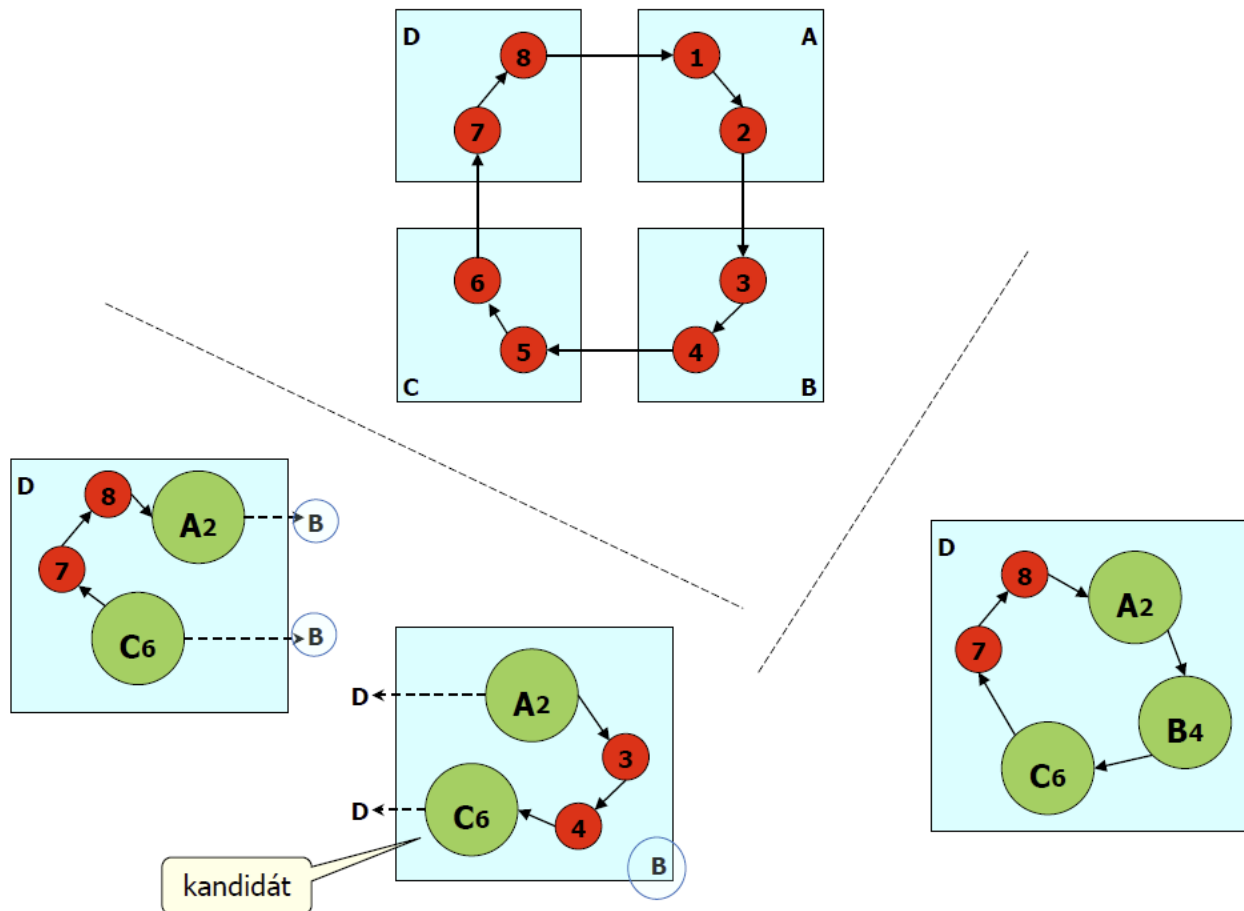
- centralizované řešení
- 4 další způsoby distribuované
 - Path-pushing
 - Edge-chasing
 - Diffusing computation
 - Global state detection
- Centralizovaný algoritmus
 - Máme dedikovaný server, kterému všechny uzly, které žádají o přístup k prostředku nebo uvolnění pošlou zprávu
 - Server tyto zprávy zpracovává a staví si svůj WFG → eviduje si stav celého distribuovaného systému
 - Když ve WFG najde kružnici, tak to řeší
 - Nalezne v kružnici proces, který odpracoval nejmenší čas (nebo jinou heuristikou) a proces zabije
 - Problém
 - Phantom deadlock = falešný deadlock kvůli zpoždění zpráv



- Pokud máme k dispozici logické hodiny a kauzální doručování → řešení zdarma
- Alternativní řešení je při detekci kružnice koordinátorem ji nechávat jako zdetekovaný deadlock ale podezření na deadlock a vykomunikovat to s uzly
- Pokud by tento algoritmus nevyhovoval z pohledu škálovatelnosti → existuje rozšíření hierarchický algoritmus
 - Každému segmentu je přidělen jeden centralizovaný server
 - Množina centralizovaných serverů si vzájemně udržuje vztahy mezi sebou
 - Mohou mít nějakou hierarchii mezi sebou
- Path-pushing
 - Spíš metoda, kterou implementují hromady distribuovaných algoritmů
 - Každý uzel si udržuje svoji vlastní část WFG a vazby na ostatní uzly
 - Ve vhodných okamžicích (periodicky nebo po změně WFG) každý uzel přepośle **externí závislosti** (redukovanou informaci o svém vlastním WFG) sousedním uzlům



- Všechny uzly jsou nějak ohodnoceny (očíslovány)
 - Slouží po případné detekci deadlocku jako cena/náklad procesu pro likvidaci
 - Dále také jako identifikátor toho, co mám zabít, pro případ, že by více procesů detekovalo deadlock, tak aby všichni zabíjeli ten stejný proces

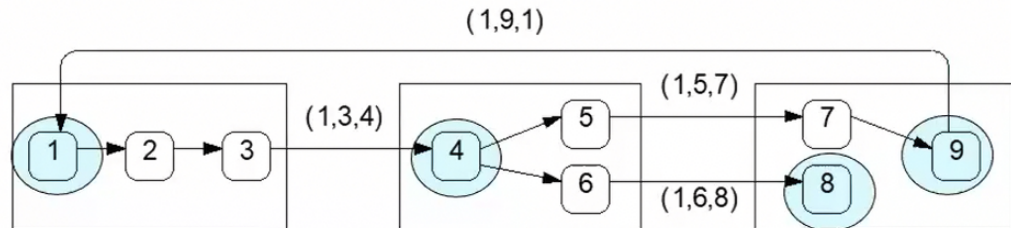


Path-pushing algorithms. Build some simplified form of global wait-for graph at each site. Each site sends its copy to a number of neighbors every time a deadlock detection is performed. Update local copy that is then passed along. This process is repeated until some site has a sufficiently complete picture of the wait-for graph to make a decision (deadlock or no deadlock). Unfortunately, many algorithms of this type are incorrect mainly because the portions of the wait-for graph that are transmitted may not represent a consistent view of the global wait-for graph since each site takes its snapshot asynchronously.

- Edge-chasing
 - Opět spíše metoda
 - WFG je použitý pro zasílání zpráv podél hran
 - Uzel, který má podezření na možnosti deadlocku pošle podél hrany speciální zprávu
 - Pokud proces na nic nečeká a běží → tato zprávu nezajímá a zahodí ji
 - Pokud tato zpráva dojde do počátku (uzlu, který ji vyvolal), tak je detekovaný deadlock
 - Opět během přeposílání zpráv se detekuje nejvhodnější kandidát na zabití
 - Důležité opět pro případ, kdyby více uzlů se rozhodlo detekovat deadlock, tak aby všichni zabíjeli ten stejný proces

Edge-chasing algorithms. The presence of a cycle in a distributed graph structure can be verified by propagating special messages called *probes* along the edges of the graph. When an initiator (there may be several) receives a matching probe, it knows that it is on a cycle in the graph.

- Speciální zprávy (probes) jsou zasílány po WFG
 - ♦ proces rozešle zprávu všem procesům, kterými je blokován
 - ♦ pokud se zpráva vrátí odesílateli – deadlock (orientovaná kružnice)
 - ♦ zpráva zároveň detekuje kandidáta
 - ♦ paralelní spuštění - overkill



- Edge-chasing Chandy-Misra-Haas

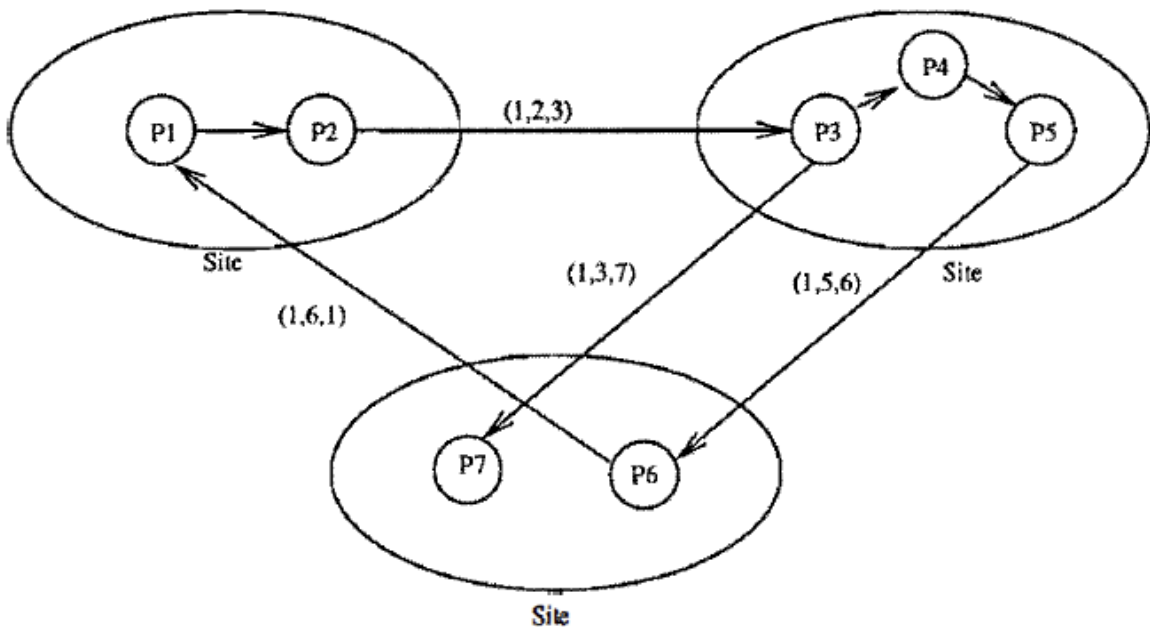


FIGURE 5.10
An example of Chandy et al.'s algorithm.

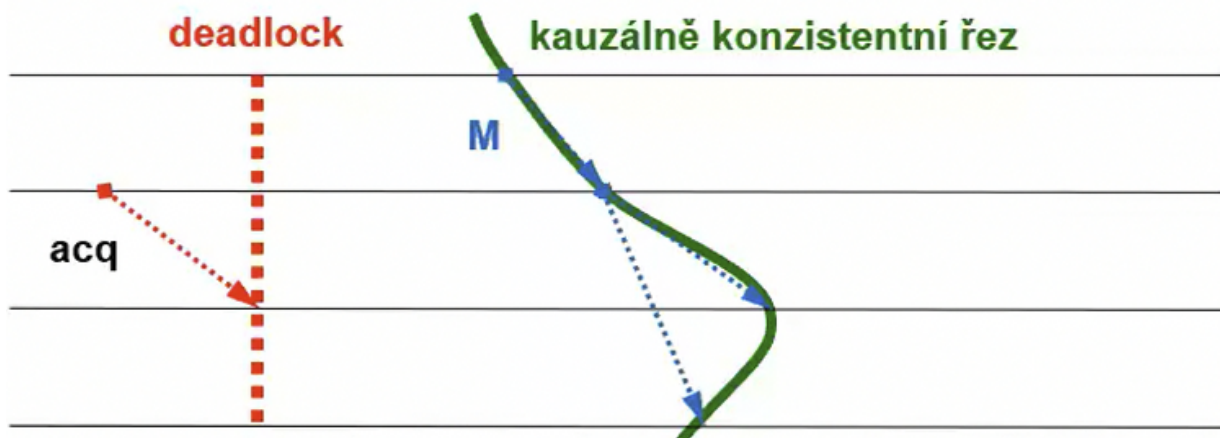
5.7.1 Chandy, Misra, and Hass's algorithm for the AND model

This is an example of edge-chasing algorithms where the distributed deadlock detection algorithm uses a special *probe signal* that is forwarded on the wait-for graph from process to process (site to site). A deadlock is detected if the signal comes back to the initiator. In [3], Chandy et al. uses a signal called a *probe* which is a triplet (i, j, k) denoting that it belongs to a deadlock detection initiated from process P_i and is sent by the home site of P_j (the site where P_j locates) to the home site of P_k . Figure 5.10 shows such an application. If process P_1 initiates deadlock detection, the controller of the home site of P_1 sends probe $(1, 2, 3)$ to the controller at the home site of P_3 . A deadlock is detected if a process receives a signal that was initiated from itself. In Figure 5.10 process P_1 detects a deadlock when it receives signal $(1, 6, 1)$ from process P_6 .

In the above algorithm two or more processes may independently detect the same deadlock. If every process that detects a deadlock resolves it, it will be inefficient because several processes will be aborted to resolve a deadlock. Another problem is that although a process detects a deadlock, it does not have information on which processes are involved. For example, in Figure 5.10 processes $P_1, P_2, P_3, P_4, P_5, P_6$ form a cycle. Process P_1 detects this deadlock but does not have information about the involved processes.

- Diffusing computation
 - Využití značkového algoritmu pro detekci ukončení
 - Příjem první zprávy znamená její propagaci
 - Příjem další zprávy znamená signál zpět
 - Když přijdou všechny signály, tak pošlu signál otci
 - Pokud se zpráva dostane až k iniciátorovi, tak se zpátky vracejí signály
 - Iniciátor se potom může rozhodnout, co s grafem udělá
 - Vhodné pro složitější modely
- Global state detection
 - Je možné přímo aplikovat značkový algoritmus pro detekci globálního stavu
 - (Použití pro detekci deadlocků bylo jednou z motivací pro jeho vytvoření)
 - Deadlock vznikne v konkrétním okamžiku → WFG se jednou konkrétní zprávou uzavře
 - Jsme ale schopni detekovat pomocí značkového algoritmu kauzálně konzistentní řez
 - Když deadlock vznikne, tak na nás počká → na každém uzlu počká na okamžik kauzálně konzistentního řezu

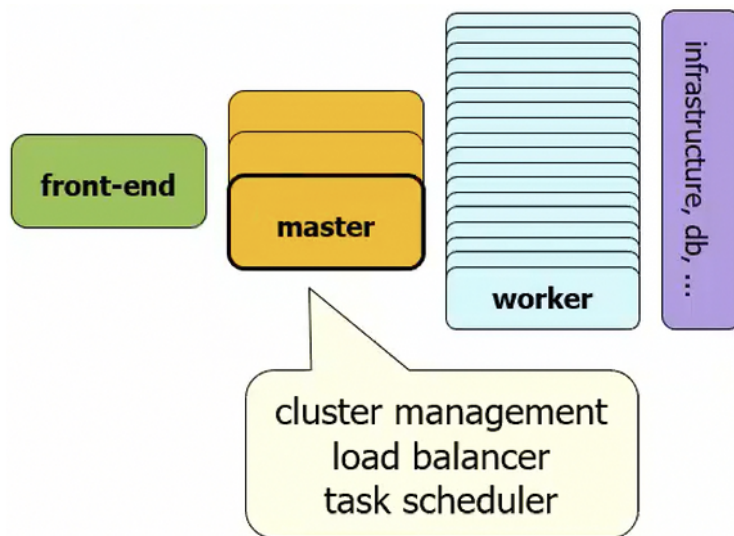
- V okamžiku kauzálně konzistentního řezu je možné detekovat, jaké procesy jsou vázané na které
- Co se s touto informací dá dělat? → varianty
 - Odeslat celý WFG iniciátorovi
 - Kontrakce + zaslání externích závislostí iniciátorovi (path-pushing)



Video 14

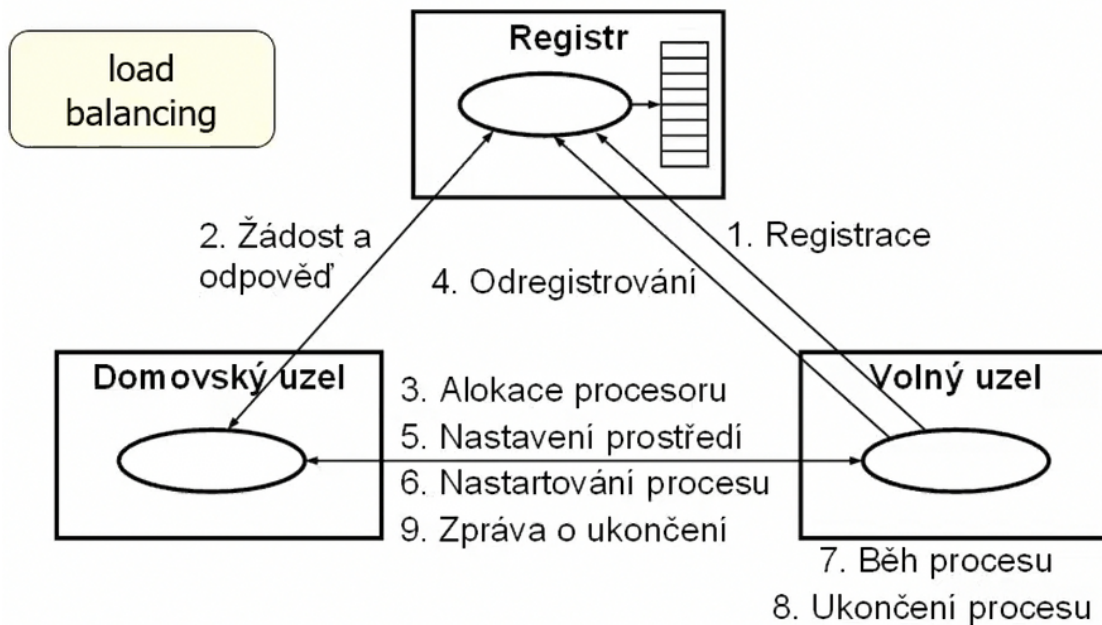
- Správa procesů v distribuovaných systémech se skládá ze 3 úloh
 - Jak sdílet výpočetní sílu
 - Jak rozdělit zátěž na jednotlivé uzly
 - Jak synchronizovat procesy a evidovat jejich stav
- Cíle
 - Cluster – nabízení služeb vnějším klientům
 - Pro krátkodobé úlohy minimalizujeme latenci
 - Pro dlouhodobé úlohy maximalizujeme výkon
 - Typicky se používá centralizovaná, nebo centralizovaně hierarchická správa procesů
 - Kooperativní systémy
 - Typicky rozsáhlejší distribuované výpočty
 - Nenabízíme služby, ale jde nám a nabízení čistého výpočetního výkon pro rozsáhlý distribuovaný výpočet
 - Rovnoměrné sdílení výkonu
 - Správa a řízení je decentralizované / peer-to-peer
- Ukážeme si 3 pohledy na proces management v distribuovaných systémech
 - Load balancing (vyvažování zátěže)
 - Kooperativní systémy
 - Migrace procesů (dneska spíše virtuálních strojů a kontejnerů)
- Load balancing
 - Budeme se bavit o clusterovém load balancingu
 - Vyvažování výkonu v rámci homogenní skupiny uzlů, na které je rozdělována množina úloh
 - Homogenní = uzly jsou stejně výkonné

- Clusterový load balancing
 - Několika málo úrovnový (viz obr)



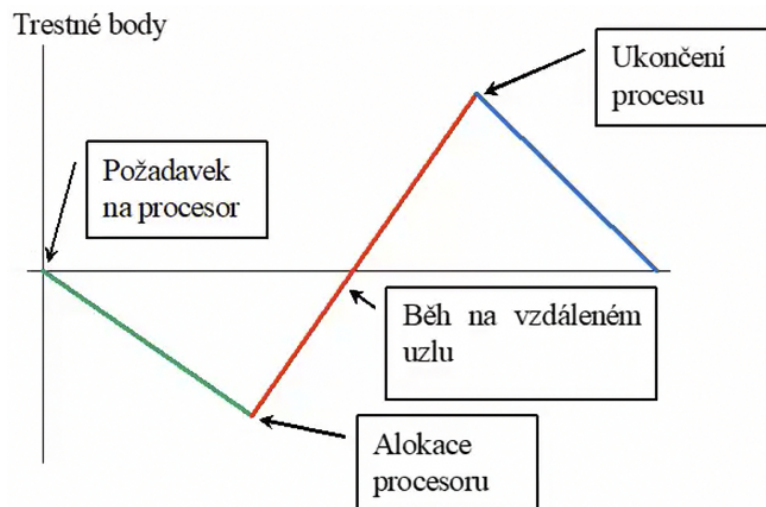
- Cluster management = master je typicky replikovaný
 - Ovládá dost velkou množinu workerů, na které přiděluje úlohy
 - Úlohy spolu zpravidla nekomunikují
- Použití algoritmů pro distribuovaný konsensus
- Vyvažovací strategie
 - Round Robin
 - Každý následující požadavek je postupně přidělován dalšímu workeru v pořadí (až množina dojde tak zase dokola)
 - Jednoduché implementovat
 - Vhodné pro krátké a srovnatelné úlohy
 - Nesmí být rozdíly mezi workery
 - Weighted Round Robin
 - Jednotlivým workerům je přidělen koeficient
 - Dynamic Round Robin
 - Pro dlouhodobější a složitější úlohy
 - Periodické měření aktuální dostupné výkonnosti
 - Master potom vypočítá klouzavý průměr (vyhladí extrémy)
 - Least Connections, Weighted LC
 - Nejmenší počet otevřených úloh
 - Random, Threshold
 - Náhodné střídání (chráněné nějakým limitem)
 - Vhodné pro heterogenní prostředí
 - Master zadá workerovi, ten může odmítnout, když se mu zdá, že je přetížen
 - Agent-Based Adaptive Balancing
 - Chytřejší algoritmy založené na umělé inteligenci...
 - Náročnější implementovat

- Ne vždy efektivní
 - Výrazněji větší režie
- Vzdálené procesy v kooperativních systémech
 - Neřeší umístění nově příchozí úlohy na jeden z workerů
 - Řeší jakým způsobem je možné zapojit volně dostupné uzly do distribuovaného výpočtu
 - Řeší jakým způsobem úlohy alokovat na dostupné uzly
 - Narozdíl od clusterových systémů je řízení a alokace distribuovaná (bez centralizované komponenty)
 - Je ale zapotřebí nějaká evidenční komponenta
 - Registruje jednotlivé volné uzly, jejich vlastnosti a kapacity



- Spuštění vzdáleného procesu
 - Když je nějaký uzel volný, tak se zaregistruje v registru
 - V případě, kdy nějaký jiný uzel, který počítá úlohů, má potřebu svůj výpočet přenést na jiný uzel, tak si může podat do registru žádost a ten mu vrátí referenci na jeden/množinu uzel/uzlů, které splňují jeho požadavky
 - Potom se uzly přímo mezi sebou dohodnou
 - A spustí se proces na volném uzlu
 - Ukončení vzdáleného procesu
 - Doběhnutí (neřešit)
 - Rollback transakce (pokud máme transakce k dispozici)
 - Zabití vzdáleně spuštěného procesu
- Kooperativní load balancing
 - Centralizované / hierarchické řízení přístupu
 - Manažeři skupin
 - Starají se o svůj segment

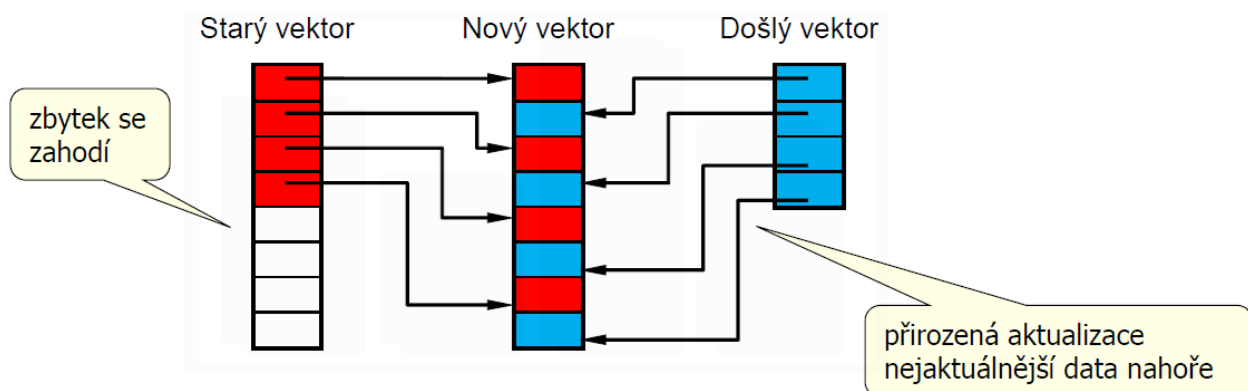
- Při nedostatku výpočetních kapacit propagují žádost nadřazenému manažerovi
- Up-down algoritmus
 - Zajišťuje rovnoměrné sdílení výkonu
 - Založen na koordinátoru
 - Obsahuje pro každý uzel tabulku s trestnými body
 - Podle této tabulky se vybírá uzel, kam se umístí nový proces
 - Tabulka trestných bodů
 - Při každé významnější události uzel pošle informace koordinátoru
 - Za každý proces běžící na jiném uzlu má plus trestné body (vzdáleně spuštěné procesy)
 - Za každý neuspokojený požadavek (touží vyslat nějaký proces někam jinam, ale koordinátor to ještě neumožnil) má minus trestné body
 - Jestliže nic z tohoto tak jeho hodnota se snižuje k směrem k nule



- Distribuovaný heuristický algoritmus
 - Vysílající uzel si vybere k náhodných výběrů
 - Podle odpovědí se rozhodne, kterému uzlu svůj proces svěří
- Deterministický grafový algoritmus
 - Založen na dopředu známých vlastnostech procesů (jak moc spolu komunikují, ...)
 - Procesy jsou rozděleny na podmnožiny
 - Ty, které spolu komunikují hodně jsou na jednom uzlu
 - Aplikace standardních algoritmů pro toky v sítích
 - Optimální deterministický algoritmus
- Migrace procesů
 - Korektní a transparentní přenesení procesu během výpočtu
 - Motivace

- Vyvažování zátěže
- Vyžaduje netriviální integraci s jádrem OS
- Z dnešního pohledu spíše zastaralé (hodně se to zkoumalo v 90. letech)
 - Neděje se to na úrovni procesů, ale virtuálních strojů a kontejnerů
 - Takto je implementace výrazně jednodušší
- Zkušenosti s implementací migrace procesů se zužitkovali pro metody a implementaci migraci virtuálních strojů / kontejnerů
- Ve slidech následuje hromada historických systémů, které Zavoral přeskočil
- MOSIX (**M**ulticomputer **O**perating **S**ystem for **U**nix)
 - Clusterový systém
 - 1977-2017
 - Používal se hrozně dlouho (40 let!)
 - Unikát mezi distribuovanými systémy
- T4
 - MFF UK (Bednárek, Merta, Yaghob, Zavoral) 1994-1997
 - Éra je dnes pryč
 - Zkoušeli na tom distribuovaně spouštět DOOM
- Migrační load balancing
 - Je zapotřebí nějakou migrační jednotku zaživa odmigrovat na jiný uzel
 - Migrační jednotka
 - Původně jeden proces
 - Neukázalo se býti praktické
 - Procesy bývají sdružené
 - Skupina procesů / task
 - Dnes virtuální stroj / kontejner
 - Párový algoritmus
 - V celém distribuovaném systému se náhodně vytváření páry
 - Pro každý pár se spočítá míra vylepšení
 - Za jak rychle by příslušné úlohy doběhli, kdyby byly přesunuty (včetně režie migrace)
 - Významné zlepšení stavu → migrace
 - Vektorový algoritmus
 - Používal MOSIX
 - Elegantní způsob evidence, distribuce a porovnávání zátěží
 - Na základě informací o sobě sama a informací od vybrané množiny ostatních uzlů
 - Každý uzel má nějaký vektor zátěže
 - Na **nulté** položce vektoru **je jeho vlastní zátěž**
 - V pravidelných intervalech si každý uzel
 - Měří vlastní zátěž
 - Zaznamenává si ji do nulté položky vektoru
 - Vezme **první polovinu** svého vektoru a pošle ji vybrané množině uzlů

- Každý uzel, když mu přijde vektor, tak ho spojí se svým vlastním starým vektorem jako zip
 - Došlý vektor se umístí na **liché** pozice
- Podle těchto vektorů se provádí vyvažování výkonu a spouštění procesu
- Pozorování
 - V prvních položkách vektoru jsou nejaktuálnější informace ohledně zátěže
 - Spodní polovina vektoru se zahazuje
 - Odesílatel ji neodesílá
 - Příjemce zipuje nový vektor a zahazuje druhou polovinu svého starého vektoru

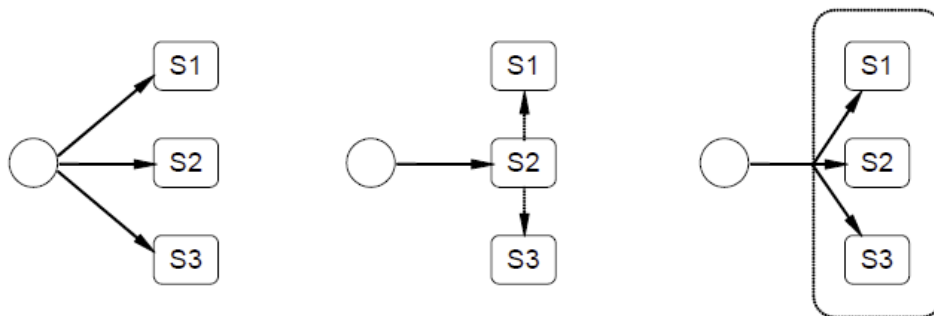


- Existuje celá řada “intelligentních” algoritmů
 - Neprosadili se pro komplikovanost a režii
- Prakticky použitelné vyvažovací algoritmy
 - Prosadili se **centralizované** a **jednoduché** algoritmy (centralizovaný a hierarchický algoritmus)
 - Lokální algoritmy
 - Každý uzel zná jenom svoji hodnoty zátěže, při překročení prahové hodnoty shání ostatní uzly (náhodně) na pomoc

Video 15

- Posledním tématem je replikace
- Replikace dat / souborů = udržování více kopií na více db/fileserverech
- K čemu je replikace dobrá
 - Spolehlivost
 - Dostupnost
 - Výkonnost
- Jak ji implementovat (zajistit)
 - Explicitní replikace
 - Klient se sám stará o udržování konzistence jednotlivých replik
 - Typicky znamená, že je to implementované v knihovnách
 - Odložená replikace

- Přenecháme starost na server
- Existuje jedna tzv. primární replika
- S tou komunikuje klient jakoby to byla jedna jediná replika
- Veškerou synchronizaci má na starost server, který všechny aktualizace posílá dalším replikám
- Ostatní repliky slouží jako záložní
- Skupinová komunikace
 - Množina replik je uzavřena do skupiny a klient komunikuje přímo s tou skupinou
 - Musí to podporovat náš distribuovaný framework



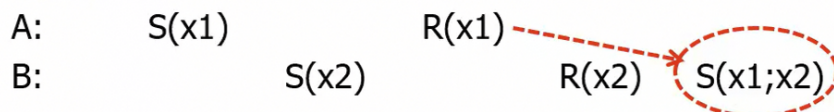
- Aktualizační protokoly
 - Kdy a které kopie budou aktualizovány
 - Primární kopie
 - Primární kopie sdružuje všechny operace a sama je propaguje dalším replikám
 - Většinové hlasování
 - Všechny repliky jsou v množině a na celé množině jsou ustanovená quora, quorum pro čtení a quorum pro zápis
 - Každé quorum je alespoň nadpoloviční
 - Pro čtení je zapotřebí oslovit čtecí quorum replik
 - Pro zápis je potřeba oslovit zápisové quorum replik
 - Teprve po úspěšné operaci na quoru je operace úspěšně přečtena nebo zapsána
 - Vážené hlasování (většinové hlasování bylo speciální případ tohoto)
 - Také zde jsou quora
 - Nemusí být nadpolovičně velké
 - Součet prvků čtecího a zápisového quora musí být větší než celkový počet replik
 - Dá se dokázat, že operace jsou ekvivalentní sekvenčnímu rozvrhu
 - Dá se pomocí velikostí quora optimalizovat celý systém buď na čtení nebo na zápis
- Klientocentrické konzistenční modely
 - <https://www.ctfnote.com/computer-science/distributed-systems/consistency-and-replication/client-centric-consistency>
 - U distribuované sdílené paměti jsme si povídali o konzistenčních modelech
 - Specifikace toho co implementace musí splňovat

- Klienti nezávisle na sobě se připojují k nějaké replikované službě
 - Např. k různé vzhledem ke geolokaci
- Klientocentrický model říká co se má dít za operace vzhledem k jednomu klientovi a příslušným replikám
 - Jaké operace mají být vidět
- Neříkají nic o tom, jaký je vztah mezi klienty → ty jsou nezávislé na sobě
- Jde o vztah **jednoho** klienta k **množině** replik
- Eventuální konzistence
 - Nejjednodušší klientocentrický konzistenční model
 - Nezajišťuje téměř nic
 - *Po ukončení všech zápisů budou všechny repliky v konečného čase aktualizovány*
 - Nevíme: jak bude dlouhý čas, pořadí

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



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

- A a B nejsou dva různí klienti, ale jeden klient, který je střídavě připojován k různým replikám
- Nejdříve nevidí operace, které se provedli na první replice, na druhou se dostanou až se zpožděním
- Monotónní čtení
 - *Po přečtení hodnoty x všechna další čtení vrátí stejnou nebo novější hodnotu*
 - Bez ohledu na kterou repliku jsme připojeni
 - Příklad: Při připojení k jiné replice klient vidí všechny ním dosud přečtené zprávy

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

Vyhovuje monotónnímu čtení

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

Nevyhovuje monotónnímu čtení

○ Monotonní zápis

- Zápis proměnná je proveden před jakýmkoliv následným zápisem této proměnné
- Udržuje se sekvenční posloupnost zápisu respektovaná na všech replikách

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

Vyhovuje monotónnímu zápisu

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

Nevyhovuje monotónnímu zápisu

- Příklad: git commit na různých replikách
 - Jedna replika v Asii, druhá v Evropě
 - Když udělám commit na replice v Asii, tak na replice v Evropě musí být všechny operace z předchozích commitů

○ Čtení vlastních zápisů (Read your writes consistency)

- Kombinuje čtení a zápisy
- Zápis proměnné je proveden před jakýmkoliv následným čtením této proměnné

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

Vyhovuje č.v.z.

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

Nevyhovuje č.v.z.

- Příklad: po aktualizace webové stránky si neprohližím kopii z cache
 - Musí se zařídit, aby aktualizace byla propagována na všechny klienty

○ Zápisy následují čtení (Writes follow reads consistency)

- Kombinuje čtení a zápisy

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

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

Vyhovuje z.n.č.

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

Nevyhovuje z.n.č.

- Příklad: zápis odpovědi do diskuzního fóra se provede tam, kde je příspěvek, na který je reagováno
- Naivní Implementace
 - Máme množinu replik
 - Ke každému zápisu je přiřazen globální jednoznačný identifikátor WID
 - $WID = repl_id + loc_id$
 - $repl_id$... identifikátor repliky
 - loc_id ... lokální identifikátor v rámci repliky (jednoduché sekvenční číslo – čítač)
 - Každý klient si udržuje dvě množiny identifikátorů
 - $read_set$
 - Identifikace čtených dat
 - $write_set$
 - Identifikace zápisu
 - Monotónní čtení
 - Replika serveru při **čtení** ověří podle $read_set$ klienta aktuálnost svých zápisů
 - Při chybějících zápisech provede synchronizace nebo forwarduje čtení
 - Klient si po čtení aktualizuje svůj $read_set$ podle repliky, ze které četl
 - Monotónní zápis
 - Replika serveru při **zápisu** ověří podle $write_set$ aktuálnost svých zápisů
 - Chybějící zápisy si zapíše
 - Klient si po zápisu aktualizuje svůj $write_set$ podle repliky
 - Čtení vlastních zápisů
 - Při **čtení** replika serveru ověří podle $write_set$ aktuálnost svých zápisů
 - Zápisy následují čtení
 - Aktualizace repliky podle $read_set$
 - Aktualizace $read_set$ i $write_set$ klienta
 - **Problém** – neomezený růst $read_set$ a $write_set$

- Pro prakticky použitelnou implementaci je nutné vymyslet efektivní implementaci *read_set* a *write_set*
- Efektivní implementace
 - Jedna z možností je to neřešit technicky ale organizačně
 - Celá aplikace je rozdělena na **krátké relace** (session)
 - Po relaci se množiny **smažou**
 - Reprezentace množin - vektorové hodiny
 - V zásadě vektorové hodiny fungují úplně stejně jak jsme si je ukazovali u jiných použití
 - *read_set* a *write_set* jsou reprezentovány vektorovými hodinami
 - $VT(RS)$... vektorové hodiny klienta pro *read_set*
 - $VT(WS)$... vektorové hodiny klienta pro *write_set*
 - Každá replika S_i si udržuje $RCV(i)$ vektorové hodiny
 - $RCV(i)[j]$... časová značka poslední operace zápisu přijatá S_i od S_j
 - Při přijetí žádosti o **čtení** nebo **zápis** replika vrátí aktuální $RCV(i)$ klientovi
 - Po přijetí $RCV(i)$ si klient aktualizuje *read_set* nebo *write_set*
 - Čtení: $VT(RS)[j] = \max(VT(RS)[j], RCV(i)[j]) \quad \forall j$
 - Zápis: $VT(WS)[j] = \max(VT(WS)[j], RCV(i)[j]) \quad \forall j$
- Epidemické protokoly
 - Slouží k implementaci a optimalizaci komunikace ve

VELMI  rozsáhlých systémech

- Neřeší konflikty
- Při výzkumu synchronizace se začala hodit teorie epidemií
 - Epidemiologové se snaží co nejvíce zabránit rozšíření infekční nákazy
 - Co nejvíce ohniska izolovat
 - Informatičtí poznatky využili v obráceném smyslu
 - Potřebujeme informaci co nejrychleji rozšířit ideálně mezi všechny
- Antientropie
 - Server P náhodně vybere server Q k výměně dat
 - Možné výměny
 - Push $P \rightarrow Q$
 - Nechává aktivitu pro komunikaci na straně uzlu, který se snaží hodnotu propagovat ostatním uzlům
 - Uzel oslovuje další uzly a hodnotu jim předá
 - Dobře funguje v počátcích epidemie
 - Není nejefektivnější na dokončování (doručení všem)

- Uzel Q by musel být náhodně vybraný
- Pull $P \leftarrow Q$
 - Aktivita je na straně uzlu, který potencionálně touží po informaci
 - Aktivně se ptají svých sousedů, jestli pro ně nemají novou informaci
 - Dobře funguje pro dokončování nákazy
- Push/Pull $P \leftrightarrow Q$
 - Kombinovaná metoda
 - Protože předchozí dvě funguje jedna na začátku a druhá na konci
 - Spojíme je dohromady
- Problém (systémů založených na antientropii)
 - Kdy uzel má přestat infikovat? (Kdy už všichni ví informaci?)
- Gossiping
 - Tentokrát sociologie (sociální chování lidí)
 - Při nákaze už jednou infikovaného uzlu se s pravděpodobností $1/k$ (nějakou definovanou) uzel uvede do klidového stavu
 - Někomu povím informaci, kterou už vím
 - Pak mě přestane bavit zkoušet oznamovat informaci, když už ji všichni budou vědět
 - Oblíbená kombinace
 - Gossiping + periodický Pull
- Obecně podporují velice dobře masivní škálovatelnost
- Pro snazší rozšiřování informací je výhodnější strukturovaná topologie
 - Pokud se uzly vybírají zcela náhodně, tak můžou vzniknout oblasti, které jsou přehlčené a oblasti, které jsou neinfikované
 - Když se celá množina hierarchicky rozdělí na domény a každý uzel nejříve infikuje správce domény a ten infikuje další správce domény tak se infekce rozšiřuje mnohem rychleji a rovnoměrněji
- Problém mazání dat
 - Pokud kromě distribuce dat chceme i data smazat
 - Nestačí pouze poslat operaci smazání
 - Zase z jiné strany by se smazaná data přepsali daty propagovanými
 - Recept je z DB
 - Smazaná data se nemažou, ale dá se k nim cedulka, že neplatí → certifikát smrti
 - Certifikáty smrti se propagují na všechny uzly distribuovaného systému
 - Certifikáty smrti mohou růst nekonečně
 - Je to možné řešit mazáním certifikátů za nějaký TTL, který musí být delší než doba rozšiřování informace
- Aplikace - agregace dat
 - Trénujeme mouchy

- Moucha = entita s vlastním procesorem (umí instrukce)
- Mouchy náhodně létají
- Mouchy se mohou náhodně srazit
 - Vždy se srazí dvě mouchy
 - Při srážce si mohou předat celý nebo část svého stavu
- Chceme implementovat
 - *Mouchy, spočítejte se!*
 - Chceme alespoň řádově
 - Nejprve vyřešíme podúlohu → spočítáme průměr
 - Při srážce dvou much si mouchy vymění svoje hodnoty
 - Každá moucha veme průměr dvou much
 - Na každé mouše konverguje hodnota k celkovému průměru celého hejna
 - Vybereme jednu mouchu, té dáme hodnotu 1, ostatním mouchám dáme hodnotu 0
 - Necháme mouchy dlouho létat
 - Potom si najdeme libovolnou mouchu a dostaneme z ní její hodnotu → převrácená hodnota musí být zhruba počet much

♦ 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 $x_i = 0$

- $x_i \rightarrow 1 / N$
 - *Která jste největší?*
 - Každá moucha má na začátku vlastní váhu
 - Při srážce si mouchy vezmou maximum
 - Pak stačí odchytit libovolnou mouchu
 - *Kolik dohromady vážíte?*
 - Máme
 - Průměrná moucha
 - Velikost stáda
 - Vynásobíme
- Poprvé tato přednáška byla v roce 1994
 - Pokud by platil Moorův zákon, tak by se vlastnosti počítačů zvýšili zhruba 4.3 mil krát
 - Realita je asi 200 000 krát
 - 100 MB vs 20 TB
 - HW vývoj je znatelný ve vývoji distribuovaných systémů