Distrib. syst. ≡ Systém propojení množiny nezáv. uzlů, který poskytuje uživateli dojem jednotného systému

- mají něco jako procesor, paměť a připojení na nějakou kom. linku
- obyč. počítače v běžné síti

80. léta – distrib. protok. a alg. – teorie

90. léta – distrib. OS, např. systém Amoeba, T4
→ neměl být rozdíl mezi lok. a distrib. aplikacemi
– ale FUJ – moc komplikované

0. léta – lokOS+ middleware – distrib. mezivrstva poskytující těm aplikacím, které chtějí, to distrib. prostř.
⇒ rozdíl mezi lok. a distr. apl.

201x – cloud computing, *aaS
↑ Software Infrastructure …

– např. Gmail, mapy.cz … běží nad tímhle

202x – trend decentralizace (web 3.0)

Cluster computing ✗✗ Grid computing

- na 1 fyz. místě je víc racků (např. na HPC MS) → Decodex, Gitlab, …
- HPC – většinou 1 master uzel a víc těch výkonných
- většinou jednotný OS

- rozsáhlejší výpočty třeba v celosvět. měřítku – např. nějaké vědecké výpočty
- volnější vazba – např. se nepředpokládá jednotný OS
- např. OSGA

Distrib. Informační systémy – distrib. databáze, ...

Pervazivní systémy, IoT – velké množství malých zařízení
~> nejde o výkon, ale tu spolupráci, distribuovatelnost

Peer-to-peer, decentral. systémy

Motivace vývoje distr. syst.

- Ekonomika – spojením více běžných počítačů docílíme podobného výkonu drahých superpočítačů
- Rozšiřitelnost – přidání uzlu je levnější a jednodušší než upgrade jedné centralizované komponenty
- Spolehlivost – výpadek jednoho uzlu není tak velký problém
  XX výpadek centralizovaného systému
- Výkon
- Distribuovatelnost – inherentní

Transparentnost

- vyšší vrstva nemusí řešit, jak to dělá nižší vrstva
- paralelismová transparentnost – teď je otevřené, jak udělat obecně automatické rozdělení problému na podproblémy pro paralelizaci

Přizpůsobivost

- autonomie – uzly nezáv. na ostatních + samostatná funkčnost
- otevřenost – obecná rozhraní

Škálovatelnost

servery, služby, tabulky, ...?

- chceme se vyhnout čemukoliv centralizovanému
- uzly se rozhodují na základě lok. dostupných informací
- nelze se spoléhat na existenci přesných glob. hodin

Komunikace

- není sdílená paměť ⇒ komunikace pomocí zasílání zpráv
  ‖
  ☺ nejde třeba udělat semafor

- z pohledu klienta je transp., zda je to lok., nebo vzdálená kom.

TCP

- moc chytrý ⇒ nevhodný pro impl. clusterového systému (malá prav. ztráty zpráv)
  (navržen pro celosvět. kom.)
- vyšší latence - handshake

TCP/FO$^{act}_{pas}$, FLIP, - -.

- specializované pro spolehlivé lok. sítě

—

Sémantika služeb

idempotentní služby - nevadí opakované provedení

- **exactly once** sémantika (ale ne vždy jde - např. tiskárna)
  nebo je to moc náročné

- at least **once** sémantika - stačí u idempotentních služeb ☺

- at most once sémantika - musí se dořešit na aplik. úrovni

Spolehlivost klienta

havárie klienta = dá požadavek a než přijde odpověď, tak umře
→ často se vůbec neřeší - jen třeba když je ten výpočet drahý či dlouhotrvající

řešení

- reinkarnace - výpočty jsou rozděleny na epochy, klient po znovuzrození nastartuje novou epochu, přičemž server zruší výpočty z předešlých epoch
- exterminace - klient po zrození sám zruší tu službu ⇒ je to jeho zodpovědnost
- expirace - úloha má přidělené kvantum času, po překročení si server klientovi řekne o další

Vzdálené volání

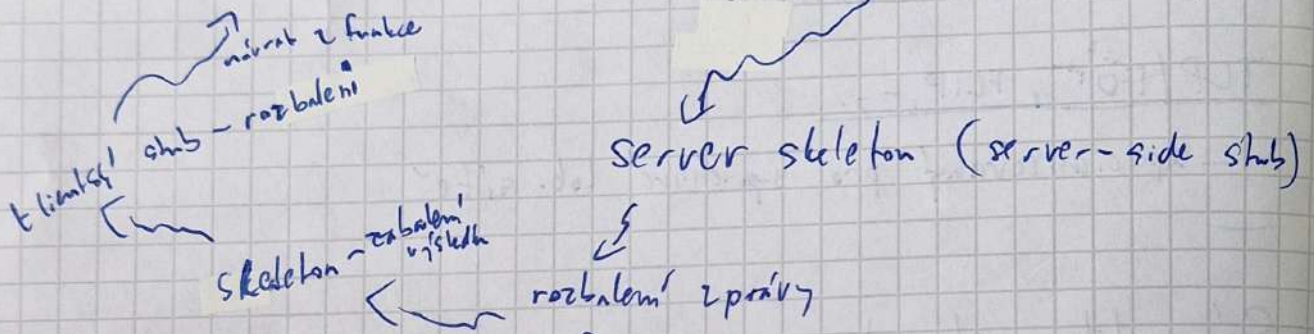
kontrola!

- server – parametry, syntaxe, práva, ...
- klient – return value - jestli se to povedlo

↳ nízkoúrovňové !!

RPC – vypadá to jako lok. volání

→ klient zavolá funkci ~> klientský stub ~> zpráva



mezi kompilací a vyvoláním musí být ještě fáze spojení

běží i třeba load-balancing

directory server

=> na dobře def. místo se zanesou údaje toho serveru (kde běží, ...)

-> klient se spojí s tím directory serverem

rozhraní specifikováno v interface definition file

↓

to se předá compileru

↓

client stub, server stub

problémy RPC

není zcela transparentní:

- není tam společná paměť – těžko se dělá práce s rozsáhlejšími dat. strukturami
- reprezentace dat – např. endianita, floaty, ... → musí být specifikováno
- komunikační chyby – výpadky ⇒ musí se řešit odlišně
- bezpečnost

příklady:

- gRPC
- Google Protobufs

Skupinová komunikace

vlastnosti:

- atomicita – zpráva je buď doručena všem, nebo nikomu
  ↳ mechanismus pozdržování
- synchronizace – aby uzly dostaly zprávy v nějakém dobře def. pořadí

uzavřená skupina – jen mezi členy
– vhodné pro koop. alg.

otevřená skupina – zasílat zprávy může kdokoliv
– spolehlivé, distribuované služby

překrývající se skupiny – může vadit, když není synchronizované
⇒ doručovací protokoly

Synchronizační algoritmy

- Synchronizace hodin mezi sebou – mezi všemi uzly
- synchronizace s nějakým centrálním časem

C := vnitřní hw hodiny, fyzický čas t (UTC)

hodiny C na uzlu p v čase t: $C_p(t)$

↪ kdyby přesné, tak $\forall t: C_p(t) = t$, $\frac{dC}{dt} = 1$

↓

ale nejsou: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$

pro dvoje hodiny: max. odchylka $2\rho\Delta t$

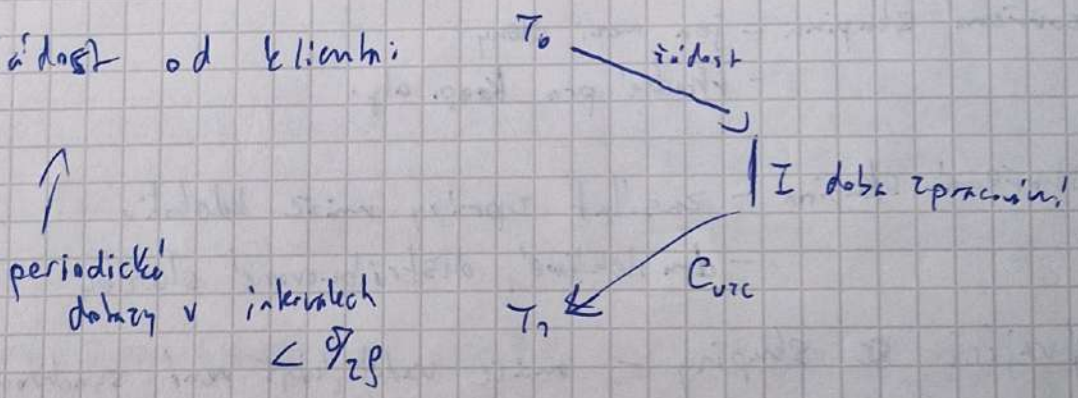
a my požadujeme odchylku δ: $\frac{\delta}{2\rho}$ synchronizační intervaly

Fyzické hodiny

Cristianův alg.

– 1 pasivní časový server

žádost od klienta:



$$T = T_{src} + (T1 - T0 - I)/2$$

– nepřeřídit najednou

– nesmí být skoky (dozadu!! – pak by se něco dělo vícekrát)

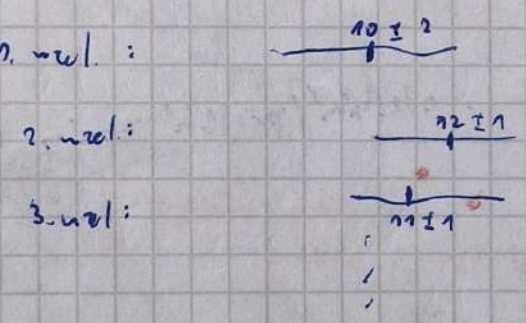
→ dělat postupným zrychlováním/zpomalováním (např. poslíček s pizzou)

Berkeley algoritmus

- 1 aktivní time server se periodicky ptá všech klientů na odchylku od času (svého mu řekne)
  - => zahodí extrémy, spočítá průměr
    - => všem pošle, o kolik si mají přeřídit hodiny
      - zase ne skokově, ale zpomalením / zrychlením tikání

Intersection algorithm

- není potřeba centrální time server
- uzly v daných intervalech broadcastují svůj čas



  - => zahození extrémů, spočítání průměru a odchylky

-> v upravené verzi používá NTP

Intervalový čas - čas je interval now ± ε

- DCE
- Google TrueTime
  - např. při nepřesnosti 200 μs/s a synchronizaci každých 30 s je odchylka ±6 ms

Logické hodiny

- fyz. hodiny nejde sesynchronizovat dostatečně přesně pro účely nějakých alg. – např. na úrovni instrukcí
- ukazuje se, že je hlavně důležité **pořadí** událostí, ne přesný čas
  + procesy, které spolu nekomunikují, synchronizaci nepotřebují

Kauzální závislost

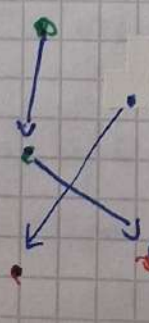
zn.: $e1 \rightarrow^p e2$ uspořádání událostí v rámci procesu/uzlu p

říkají: **jedny** hodiny

send(m)
rcv(m) . odešlání/přijetí zprávy m

DF.:
- Pokud $\exists p: e1 \rightarrow^p e2$, pak $e1 \rightarrow e2$; "e1 kauzálně předchází e2", "e2 je kauzálně vázaná událostí e1"
- $\forall m$: send(m) $\rightarrow$ rcv(m)
- $e1 \rightarrow e2 \wedge e2 \rightarrow e3 \Rightarrow e1 \rightarrow e3$

DF.: Události e1, e2 jsou **konkurentní** $\equiv e1 \not\rightarrow e2 \wedge e2 \not\rightarrow e1$



→ evidují kauzalitu: události a, b ⇒ časové značky C(a), C(b)

jestliže a → b, pak C(a) < C(b)

ale pozor: když C(a) < C(b), tak nemusí platit a → b

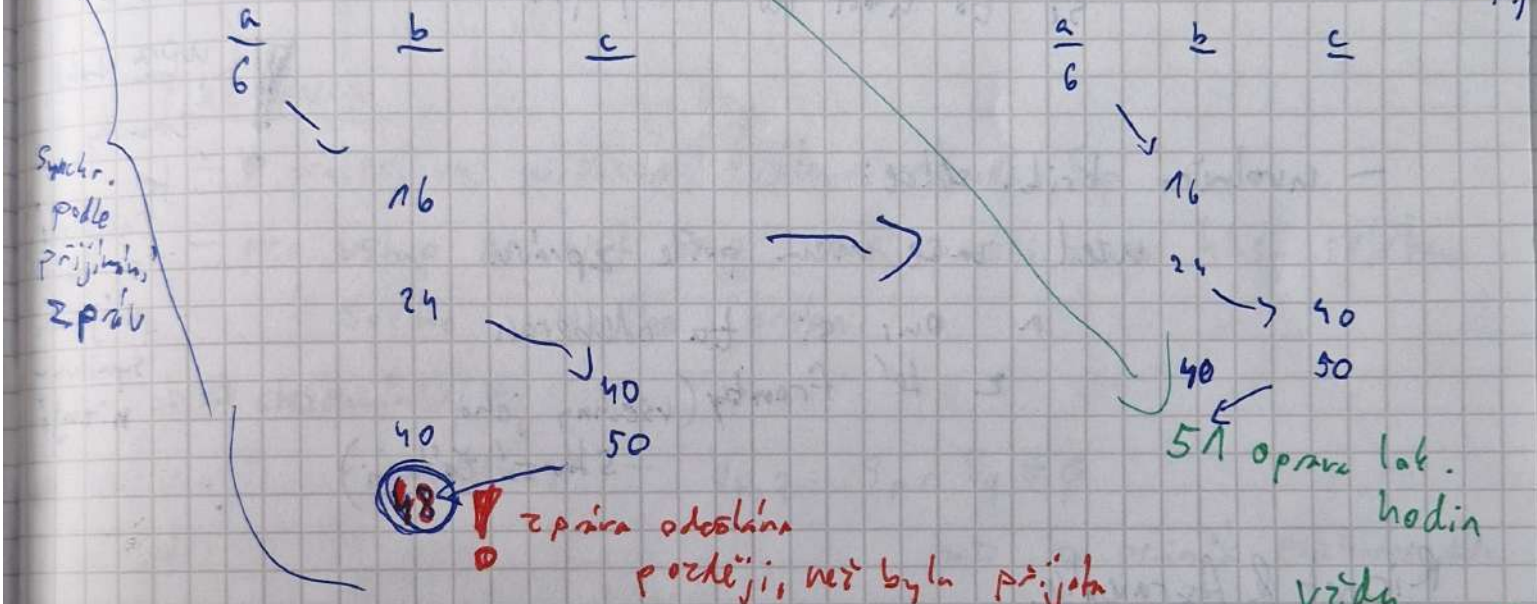
Synchronizace log. hodin

proces i vysílá v čase $C_i(a)$ zprávu m → $T_m = C_i(a)$

proces j přijme zprávu m v čase $C_j(b)$

→ pak $C_j = \max(C_j(b), T_m + 1)$

def. na tě DS, která reprezentuje by značky (číslo, vektor, ..)



úplnější – aby jakékoliv 2 události měly rozdílná čas. raz.

$C(a) = C(b) \wedge P_i < P_j \Rightarrow C'(a) < C'(b)$

(událost v proc. i; událost v proc. j; nějaké označení procesů i, j)

= "byrokratické usp."

Distrib. synchr. alg.: --> mutual exclusion

Centralizovaný alg.

- je jeden server s frontou, kam mu ostatní posílají žádost a až na ně přijde řada, tak jim server řekne
- FUJ – centralizovaná komponenta

při výpadku serveru musí všichni klienti server kontaktovat znova atd...

Lamportův alg.

- když proces chce vejít do krit. sekce, tak pošle žádost všem – ti si to uloží do Fronty žádostí a zpět mu pošlou potvrzení
- do krit. sekce může vstoupit ten proces, jehož žádost má nejstarší čas. značku a má už od všech potvrzení (tedy ten uzel u sebe nevidí starší žádost od nějakého jiného)
- když proces přijme potvrzení, tak si to uloží do Fronty potvrzení s upravenou čas. značkou (podle čas. značky té žádosti)
- uvolnění krit. sekce: uzel zašle všem pošle zprávu a oni si ho odeberou z té fronty (všechny jeho starší žádosti)

si to přijemci synchronizují

Ricart & Agravala

- jako Lamport, ale ACK má jiný význam: Souhlasím s tím, abys vstoupil do krit. sekce
- když chce proces vejít do krit. sekce, tak všem pošle žádost (se svou TS)
- když proces přijme žádost:
  - když ho nezajímá krit. sekce → ACK
  - když je v krit. sekci → dokončí svou práci tam a až pak pošle ACK všem z fronty
  - když není v krit. sekci, ale chce tam taky:
    - když jeho žádost má nižší TS, tak
    - jinak ACK a čeká

žádost si dá do fronty,

Princip voleb

- proces sbírá hlasy od ostatních
  —> když dostane víc hlasů než jakýkoliv jiný, tak může vstoupit do krit. sekce
- ∀ proces má 1 hlas

Naivní volby

- prostě mu ten hlas dáme (pokud jsme ještě nehlasovali)
- deadlock!, složitost $O(n)$ ☹
- ale odolné vůči výpadkům ☺

Maekawa

- ∀ proces má přiřazený volební okrsek S_p
- pro vstup do krit. sekce musí proces získat hlasy celého svého volebního okrsku
- podmínky:
  - korektnost – $\forall p,q: S_p \cap S_q \neq \emptyset$
    => ten průnik rozhodne, kdo tam z těchto dvou jako další může vstoupit
  - spravedlnost: $\forall p,q: |S_p| = |S_q| \geq k$ (± 1)
  - rozpovědnost: $\forall p,q: |\{S_i : p \in S_i\}| = |\{S_j : q \in S_j\}| = D$

=> kom. složitost $O(|S_p|)$ ☺

N := # okrsků
k := velikost okrsku
M := # procesů
D := # okrsků, ve kterých je každý proces

$N \cdot k = D \cdot M$

pro proces ≈ okrsek: $N = M \Rightarrow k = D$

max. # okrsků je $(D-1)k+1$

∀ proces q z okrsku S_p je obsažen v $D-1$ jiných okrscích
$\Rightarrow M = k \cdot (k-1) + 1 \Rightarrow O(\sqrt{M}) = k$

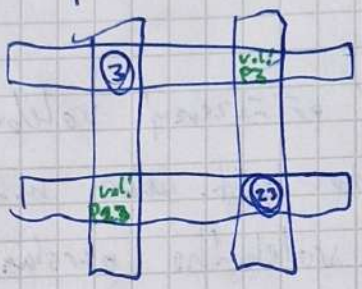
Rozdělování do okrsků

- opt. alg. FUJ v přímém členství vyžaduje restrukturalizaci
- subopt. pro $k = O(\sqrt{n})$

=> dáme si tyto procesy do množiny

a procesu přidělíme okrsek z jeho řádku a sloupce

↳ ale pozor na deadlock!



Soutěží P_3 a P_{23}

=> nikdo z nich nedostane všechny hlasy ze svého okrsku

řešení!

pomocí log. hodin

– když už proces P hlasoval, tak si porovná TS_q s TS_r tentokrát dal hlas

tabulka:

$TS_r > TS_q$ → dá si žádost do fronty

$TS_r < TS_q$ => pošle q zprávu REJECT

q už je v krit. sekci => dokončí a pak odpoví

q ještě nemá všechny hlasy => vrátí hlas procesu P a ten ho dá procesu r

race condition, ale nevadí – neřešíme výpočet, ale obranu proti deadlocku

Stromové alg.

Agrawal & El Abbadi – $O(\log n)$

- úplný Bin. strom – proces musí získat hlasy všech na lib. cestě, která jde od kořene do listu přes ten proces

- Fault tolerance – havarovaný uzel lze nahradit cestou v jeho levém a pravém podstromu

- asymetrický !! – ten v kořeni musí furt dělat nějaké hlasování

⇒ řešení: nějak se to rabuje
→ ale tím se to zesložitňuje a vyjde to horší než ten odmocninový protokol

Token-based alg.

Raymond

- 1 token v krit. sekci
- uzly organizovány do or. stromu, na vrcholu je vždy uzel s tokenem
- uzel chce do krit. sekce: ve směru or. pošle žádost
  → ten, kdo má ten token a už není v krit. sekci pošle proti směru ten token (a mění se orientace)

Suzuki & Kasami

- hodí se pro koop. distr. výpočty (přímá komunikace všech se všemi + máme broadcast)
- žádost o vstup se pošle broadcastem jen tomu, který drží token, ho přidá do fronty

Token ring

- stačí jen lokální znalost
- uzel si token nechá, když chce vstoupit do krit. sekce, a pak ho pošle dál

- různé varianty - když nikdo nechce do té krit. sekce, tak se zbytečně vytěžuje síť
  → takže např. má čítač a když uzlu přijde ten token moc rychle od minula, tak ho trochu pozdrží

všechny ty distr. alg. mají velké problémy při výpadcích

→ snaží se řešit centralizovaný problém (vstup do jedné krit. sekce) distribuovaným způsobem → FUJ

takže pro menší sítě se vyplatí ten centralizovaný alg.
a u větších se chceme potřebě vzájemného vyloučení vyhnout

Volba koordinátora (leader election)

→ aby měl nějakou roli

– nejdůležitější je, aby nemohli zároveň 2 koordinátoři

přístupy:

- detekce extrému
  - předp.: každý uzel je identifikovatelný nějakým řetězcem id
  - Bully, Invitation, kruhové alg.
- race condition (+ randomizace)
  - necháme ty procesy něco dělat a který náhodou vyhraje, tak se stane koordinátorem
  - ale ta podmínka výhry nemusí být jednoznačná!! ^
  - ⇒ randomizace ☺

Detekce extrému

Bully alg.

– předp.: homogenní uzly, hodně spolehlivá síť "

⇒ spolehlivá detekce havárie

(omezená doba přenosu zprávy T_m + omezená doba zpracování zprávy a odpovědi) T_p

$2T_m + T_p + \varepsilon$

ne třeba TCP

1. kolo: osloví se všichni vrstevní kandidáti

2. kolo: vítěz všem oznámí, že je novým koordinátorem

volby

– ∀ proces, který chce volit, pošle ∀ uzlům s vyšším id zprávu

- přijde odpověď: hotovo
- nepřijde: stane se koordinátorem

– proces dostane zprávu o volbě: odpoví a pošle žádosti ∀ vyšším procesům

nelze použít v async. systému ^

byly by falešné detekce havárií

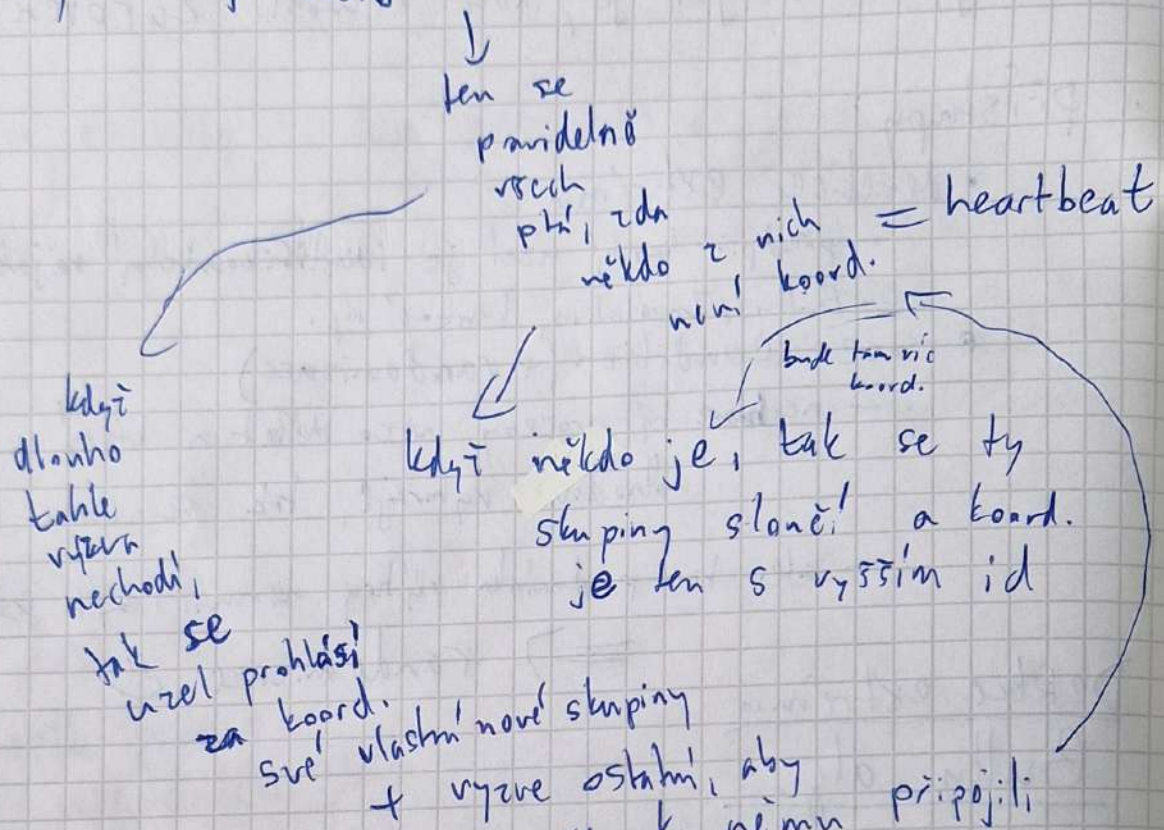
BLE

víc koord.

Invitation alg.

- už nepředpokládá, že absence odpovědi = havárie

- Skupiny - mají svého koord.



Kruhové alg.

- pro rozsáhlejší prostředí, jednotlivé uzly nemají kompletní znalost

Le Lann, Chang, Roberts

- proces pošle zprávu podél orient. hrany

s id odesilatele
+
s id uzlu s nejvyšším id, kterému to už přišlo

--> sbírá se seznam živých uzlů

—> když už je to nejvyšší, tak všem oznámí, že je koordinátor

- složitost $O(n^2)$ ↳ může začít volit líb. proces i všechny najednou

HS alg. – optimalizace

– orientace hran je obousměrná

– v několika kolech, postupně se zdvojnásobuje okolí, ve kterém se volí a určí se účastní jen ten vítěz z předchozího kola

$O(n \log n)$

Randomizované protokoly

- úplně za začátku jsou všichni **followeři**; každý po rand. timeoutu přejde do role **kandidáta** a pošle ostatním žádost o hlas pro tuto **epochu**
  => příjemce má novější epochu => nedá mu hlas
  => příjemce má aktuální epochu a ještě nehlasoval (ani sám není kandidát) => dá mu hlas
  => příjemce je kandidát v >= epoše => nedá mu hlas
  => příjemce má starší epochu => dá mu hlas a pokud je kandidátem, tak se kandidatury vzdá
- když se dlouho nedří získat nadpoloviční většinu hlasů, tak každý uzel počká náhodný timeout a zkusí se to celé znovu
- funguje dobře, když T>>doba přenosu zprávy
- když followerovi dlouho nechodí heartbeat, tak se stane kandidátem a zahájí novou epochu

– v clusterech, cloudech apod.

↳ např. v RAFTu (viz později)

– jiný přístup: uzly náhodně vygenerují číslo -> pošlou ostatním

2-fázový commit

příjem zprávy – v bufferu ✗ doručení zprávy – přijatá + splněny podmínky

Doručovací protokoly

Když mi někdo lepšího kandidáta, tak to nepotvrdí

Globální usp.

-> zprávy doručovány v pořadí odeslání

– NEJDE 🙁 – nemáme dost přesné glob. hodiny

kauzalita -> obsah vázané zprávy může být ovlivněn

Sekv. usp. (totální usp.)

-> Události mají čísla a podle nich jsou usp.

– bez ohledu na skut. čas odeslání lze podle těch čísel určit jejich pořadí

sekvencer – centralizovaná komponenta přiřazuje ta čísla

dvoufázový distr. alg.

Atomický multicast

– sekv. + zaručuje atomicitu (buď doručeno všem, nebo nikomu)

Kauzální usp.

-> kauz. vázané zprávy v pořadí kauzality, konkurentní v lib. pořadí

– když přijde zpráva, ale my k ní kauzálně vázané ještě ne všechny, tak se ještě nesmí doručit

Jak příjemce zjistí, že už může tu zprávu zpracovat

$dest(m)$:= množina uzlů, kterým je zaslána zpráva m

$deliver_p(m)$:= událost doručení zprávy m uzlu p

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

chceme ↑

vektorové hodiny

n := # procesů ve skupině

$VT(p)$, $VT(m)$ – čas. značky procesu / zprávy (vektory!)

↓

na začátku $VT(p_i) = 0$

při odeslání: zvýšíme si VT a to bude $VT(m)$ (svou složku)

při doručení: upravíme si podle zprávy (ne přijetí!) své $VT(p)$ - po složkách na max.

kauzální doruč. protokol

odeslání zprávy m uzlem p_i:

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

⇒ pošle $VT(m)$ a tu m

přijetí uzlem p_j: doručení se pozdrží, dokud: (= musí to být první zpráva od toho odesílatele)

- $VT(m)[k] = VT(p_j)[k] + 1$ pro $k = i$
- $VT(m)[k] \leq VT(p_j)[k]$ jinak

→ = musí být doručeny všechny kauz. vázané předchozí zprávy

doručení: p_j si upraví VT: $\forall k = 1, \dots, n: VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k])$

Kauzální doručování pro překrývající se skupiny

→ maticové hodiny – uzel s_i udržuje ty skupiny, jejichž je sám částí

odesílání v uzlu p_i do skup. g_a:

- $VT_a(p_i)[i]$ ++
- $VT(m) := \bigcup_{b:\, p_i \in g_b} VT_b(p_i)$

přijetí zprávy v uzlu p_j (od p_i, s $VT(m)$, do g_a)

→ doručení se pozdrží, dokud neplatí:

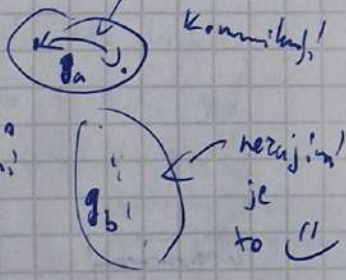
jako u vekt. {
- $VT_a(m)[i] = VT_a(p_j)[i] + 1$
- $\forall k: (p_k \in g_a \wedge k \neq i): VT_a(m)[k] \leq VT_a(p_j)[k]$

kauzalita na průniku všech skupin { • $\forall b: (p_j \in g_b): VT_b(m) \leq VT_b(p_j)$

průnik odesílatele a příjemce

~~vektorové hodiny přes všechny skupiny~~

→ mat. hodiny umožňují izolaci lokálního chování uzlů



→ u vekt. bychom synuli, že by se musely všechny zprávy posílat všem !! (protože na ně musí čekat)

nořeší se fyz. usp., ale dodržuje jednu posl. událost na všech příjemcích

Distribuovaný total-order prot. => sekv. doručování

=> všechny uzlům doručeny ve stejném pořadí → stačí skalární hodiny

příjem zprávy: synchronizace log. hodin s $TS(m)$ + pošle se potvrzení s $TS(p_j)$ → při obdržení všech potvrzení odesílatel pošle finální zprávu s $TSF := \max(TS(p_k))$ (je jednoduché uspořádání těch událostí)

Centralizovaný total-order prot.

sekvencer = proces, který synchronizuje pořadí událostí

↳ jemu se posílají ty zprávy a on je pak přeposílá

(příjemci ještě musí zajistit, aby se ty zprávy nepředbíhaly + pomocí nějakého čítače)

↳ a příjemci mu pošlou potvrzení

- zase je to centralizovaný problém, tak nemá smysl to řešit distribuovaným algo.

Spolehlivé doručování

↳ ∀ členům skupiny bude doručena zpráva

↳ spolehlivý kanál? → ALE:

- výpadky příjemce
- výpadky odesílatelů

↳ řešení:

- transakce
  - třífázové
  - příliš striktní
- záplavový alg.
  - při příjmu dosud nepřijaté zprávy ji uzel přeposle ∀ členům skupiny
  - → spolehlivé, ale neefektivní ($O(n^2)$)
- alg. s potvrzováním
  - stabilní zpráva = byla doručena ∀ nezhavarovaným uzlům
  - jako záplavový, ale nepřeposílá se všem, ale jen potřebným
    - ↳ když uzel odešle zprávu, tak si ji příjemci nechají, dokud nedostanou od všech ostatních potvrzení, že ji přijali (nebo do timeoutu)

ještě HORŠÍ

zhavarovaný uzel = po nějakou dobu nebyl schopný na nic odpovídat

↳ příjemci si ji taky po tuhle dobu nechají (a pošlou Acks)

↳ pak uzly tu zprávu přeposlou všem, kdo ji nepotvrdili

jak zjistit:

Trans alg.

↳ transitive ACKs

-> ack(m) se posílá, když přijmeme m a ∀ kauzální vázané předcházející zprávy

↳ nemusí se potvrzovat ty předchozí

- ∀ uzly si pamatují graf kauzality skupiny (DAG)

- m' potvrzuje m => $(m', m) \in G$

- 3 komponenty: • příjem ack a výpočet vlastních ack viz slide

G_p := ∀ zprávy, které p přijal a ještě nejsou stabilní

když ale nějaký uzel trvale zhavaruje, tak alg. do nekonečna čeká!! ^

Jak p_j zjistí, které uzly zprávu přijaly:

zpráva obsahuje i seznam všech zpráv, které už byly kauz. potvrzeny

∀ uzel má: • ack_list = seznam zpráv pro potvrzení

• nak_list = nepřijaté zprávy

• undelivered_list = přijaté, ale nedoručené

(pošle s každou zprávou)

-> když přijme zprávu:

1) podívá se na její nak_list a pokud něco z toho má, tak to VŠEM pošle (stačilo by jenom tomu jednomu uzlu)

protože když je problém na jednom uzlu, tak je typicky i na dalších

↳ optimalizace - snížení latence řešení problému

2) když už ji dostal ($m \in G_p$), tak konec

3) když je v jeho vlastním nak_listu, tak ji z toho vyndá

4) podívá se, jestli v ack_listu té zprávy není nějaká, kterou nemá

5) tu zprávu si dá do undelivered_listu

6) prohrabe se undelivered_listem, jestli nějaká zpráva nemá splněné ty podmínky (jsou doručeni její kauzální předchůdci)

a když tak ji doručí

a když tak ji si dá do nak listu

↳ dá ji do ack_listu

7) z G_p si odebere ∀ stabilní zprávy

Virtuální synchronie

--> koncept řešení problému konzistence při selhání
- při havárii nastane změna pohledu na nový (L^x --> L^{x+1})

group view - množina uzlů ve skupině

značení: L, L_p, L^x, L_p^x ← verze pohledu

↑ jak proces p vidí tu množinu

procesy p, q sdílejí L^x a L^{x+1}

změna verze pohledu

$\Rightarrow$ install L_p^x < $deliver_p(m)$ < install L_p^{x+1} $\Rightarrow$ install L_q^x

↑ deliver(m)

↑ install L_q^{x-1}

pohledově-synchronní komunikace

$\Rightarrow$ $p \in L_q \Rightarrow q \in L_p$ (= podmínka vzájemné konzistence)

implementuje

- když je zpráva doručena nějakému uzlu a odesilatel zhavaruje, tak musí protokol zajistit, aby byla doručena i ostatním
- nesmí se doručovat do jiných pohledů

Transis alg. – rozšíření Trans

- paranoia = stačí, aby jeden uzel měl podezření, že jiný uzel je zhavarovaný, a už ho ten uzel označí za zhavarovaného a všechny uzly z toho pohledu to musí respektovat

všem pošle zprávu FAULT(q) a ti ji zase všem přepošlou se svou verzí

- jednosměrnost = když je někdo označen za zhavarovaného, tak je ze skupiny vyloučen (a nic proti tomu nemůže dělat)

→ jenom pak znovu požádat o členství

bordel, protože v dist. prostředí nelze při nespolehlivosti dosáhnout spolehlivého konsenzu

Last[i] := uzly označené proc. i za zhavarované
J := nově přidávané uzly
J Last[i] := uzly naposledy označené proc. i za přidávané

kauzálně vázané zprávy si počkají (blocked)

v každém uzlu se individuálně nastaví kauzální hranice instalace nového pohledu a

ISIS/VSync protokol

- používá maticové hodiny (MT)

$MT_{P_i}[j][k]$ – co uzel P_i ví o doručení zpráv uzlu P_j od P_k

- příjem zprávy od P_j:

$$MT_{P_i}[i][j] = VT_m[j]$$
$$MT_{P_i}[j][*] = VT_m[*]$$

- $\forall$ proces udržuje zprávy, dokud nejsou stabilní
- při instalaci nového pohledu:
  - přepošle $\forall$ nestabilní zprávy
  - ostatní mu pošlou flush zpráv – potvrzení instalace
  - když jich má dost, tak nainstaluje ten nový pohled (od všech, kteří nejsou v množině zhavarovaných)

Globální stav a Konsensus

Diffusing Computation

= uzly tvoří orientovaný graf, dynamická kostra, v opačném směru kom. kanálů jsou signální kanály

otec je ten, od koho přišla první zpráva v tom výpočtu

Ukončení distr. procesů

stav uzlu < aktivní / pasivní (může přijímat zprávy a přejít při tom do akt. stavu)

konec výpočtu → všechny uzly pasivní

Dijkstra-Scholten (DS) algoritmy

Strom

- když se list dostane do pas. stavu, pošle signál otci
- vnitřní uzel když dostane signály od všech synů, tak pošle signál svému otci

DAB

- v kterém je čítač = deficit – rozdíl mezi aplik. zprávami poslanými tam a signálními zprávami poslanými zpět
- končící proces počká na nulový deficit dat. kanálů a pak každým signálním kanálem pošle tolik signálů, aby v nich vynuloval deficit

Obecný graf

- dynamicky se vytváří kostra
- proces $\forall$ signálními kanály kromě toho do otce pošle signál, čeká na zpětné signály a pak pošle signál otci

Huangův alg.

V proces a zpráva na váhu

W = váha iniciátora, ostatní mají váhu 0

- při odeslání zprávy uzel má část své váhy té zprávě
- při příjmu si proces váhu zprávy vezme
- při ukončení proces pošle zprávu s celou svou váhou
  → konec výpočtu: váha iniciátora = W

problémy:
- dělitelnost vah
- havárie procesů
- ztráty vah

Značkové alg. (ZM)

- značka = speciální zpráva
- když iniciační uzel má pocit, že by měl výpočet už být ukončen, tak všem výst. kanály pošle značku
- při příjmu značky:
  - uzel aktivní – nack
  - uzel pasivní – propaguje značku výst. kanály

→ **detekce glob. stavu**

- množina událostí E
- řez c ≡ disj. rozdělení E na P_c a F_c (past, future)
  ↳ je kauzálně konz. ≡ $a \to b \wedge a \in F_c \Rightarrow b \in F_c$
  (minulost nelze měnit v budoucnosti)
- stav ≡ množina událostí, co už proběhly
  ↳ konzistentní ≡ $S = P_c$ pro c kauz. konz. řez

- stav uzlu = množina přijatých a odeslaných zpráv + dodatečné informace specifické pro to, co řešíme
- stav kanálu = množina zpráv, které byly kanálem odeslány, ale ještě nebyly doručeny

→ iniciátor všem výst. kanály pošle značku
- při příjmu první značky uzlem: = okamžik lokálního řezu
  - zapamatuje si svůj stav, vynuluje stav všech vstupních kanálů
  - všem výst. kanály pošle značku
- příjem zpráv kanály, ze kterých ještě nepřišla značka
  → uzel si zapamatuje jejich čísla
- příjem dalších značek → stav kanálu = zprávy došlé mezi příjmem první a této značky

→ konec po přijmutí všech značek

konzistentní globální stav = zaznamenáváme stavy uzlů a kanálů

Distribuovaný konsensus

- → Byzantský = 1 → 1
- → konsensus = n → 1
- → Interaktivní konzistence = n → n

Problém dvou armád – koordinace entit – spolehlivé uzly, nespolehlivé kanály

B_1 → W ← B_2 musí zaútočit najednou

poslové, špehové => nespolehlivý kom. kanál

použití např.:
- replikace serverů
- replikace logů
- synchronizace
- bariérová orchestrace
- distrib. key-val. databáze
- obecně cloudové aplikace

=> obě entity si musí být jisty, že ví, co mají dělat

Praktické řešení

- agresivní strategie – vyšle se dostatečné množství zpráv
- pravděpod. strategie – když se ty charakteristiky kanálu mění
  - pošle se víc zpráv a k tomu i jejich počet
  - → ta druhá strana si porovná, kolik toho přišlo, s tím počtem

Problém Byzantských generálů

- uzly mohou havarovat a chovat se zákeřně
- spolehlivá komunikace →
  - někteří generálové zrádci
  - všichni se musí shodnout
  - rozhodují se na základě informací od ostatních gen.

BÚNO 1 generál, ostatní důstojníci (předají rozkaz od generála), rozkaz bude vydán na základě většiny

chceme:
- $\forall$ uzly se shodnou na hodnotě
- Pokud hodnotu navrhl nehavarovaný uzel, tak se uzly na ní shodnou

=> pro f zrádců $\nexists$ řešení pro $n \leq 3f$ uzlů, je potřeba $n \geq 3f+1$ uzlů → exp. alg. vzhledem k f :(

Praktické řešení

- PBFT → 4 fáze:
  - request (od klienta) změny stavu – řekne nějakému nodu → ten to přepošle všem + zahashuje
  - pre-prepare – nody ověří tu zprávu a hashem a všem přepošlou
  - prepare – po obdržení 2f+1 prepare zpráv včetně svojí => ověříme a pošleme COMMIT
  - commit – po obdržení 2f+1 COMMIT zpráv → ověříme a provedeme
- kryptografie, podpisy

+ timeouty, checkpointy

polynomiální :) ale malá škálovatelnost!! a citlivé na Sybil attack → útočník si vygeneruje víc těch identit a tím přehlasuje ty korektní uzly

Paxos protokoly

Fault-tolerance – není to výjimečná událost, ale počítá se s tím

- nespolehlivá síť, nespolehlivé uzly, ale nejsou zákeřné (když nefungují, tak neškodí)
- použití: Fault-tolerant replikovaný automat → $\forall$ uzly provádí stejné změny na stejném automatu
- Part-time parliament
  - poslanci, poslíčci > nespolehliví, asynchronní systém (bez timeoutů) + není potřeba exactly-once sémantika
  - ↳ každý poslanec má zákoník a nesmazatelný inkoust (disk) – info tam zůstane zaznamenané, i když si poslanec odskočí!

- role:
  - Client
  - Proposer – navrhuje nový stav
  - Acceptor v Quoru – schvaluje návrhy
  - Learner – zapisuje si akceptované výsledky

  } v každém uzlu – moduly

- pro log. usp. se používají log. hodiny s byrokrat. usp. = ballot numbers
  → akceptoři akceptují jen zprávy s nejvyšším Ballot number
  → pamatují si:
  - LastBallot
  - AcceptBallot
  - AcceptValue

- protokol (základní verze):
  1. fáze: volba leadera, ještě se vůbec neřeší ta hodnota
  2. fáze: akceptace hodnoty

  1. fáze: cílem je potvrdit proposerovi, že má dostatek hlasů na navrhování
  2. fáze: leader navrhuje

  Když umře starý leader, ale ještě se nestihla dosáhnout konsenzu, tak aby to nový leader ještě dodělal

  ale nepoužívá se – neefektivní

  Korektnost není deterministicky zaručena
  → v praxi se tam při konfliktech lávají randomizované timeouty

- Multi-Paxos
  → 1. fáze jen jednou a pak už jen 2. (dokud nedojde ke konfliktu)
  → zprávy mají ještě jednu položku – číslo instance
- Cheap Paxos → jsou tam záložní uzly pro výpadky
- Fast Paxos → client rovnou posílá acceptorům, při konfliktu to vyřeší leader (určí pořadí a udělá to v tom pořadí jako v basic verzi)
- Byzantský Paxos → ještě verifikace, ale moc se nepoužívá

- impl. a aplikace:
  - Google Chubby – distrib. zamykání
  - Google Borg / Omega / kubernetes
  - Derecho

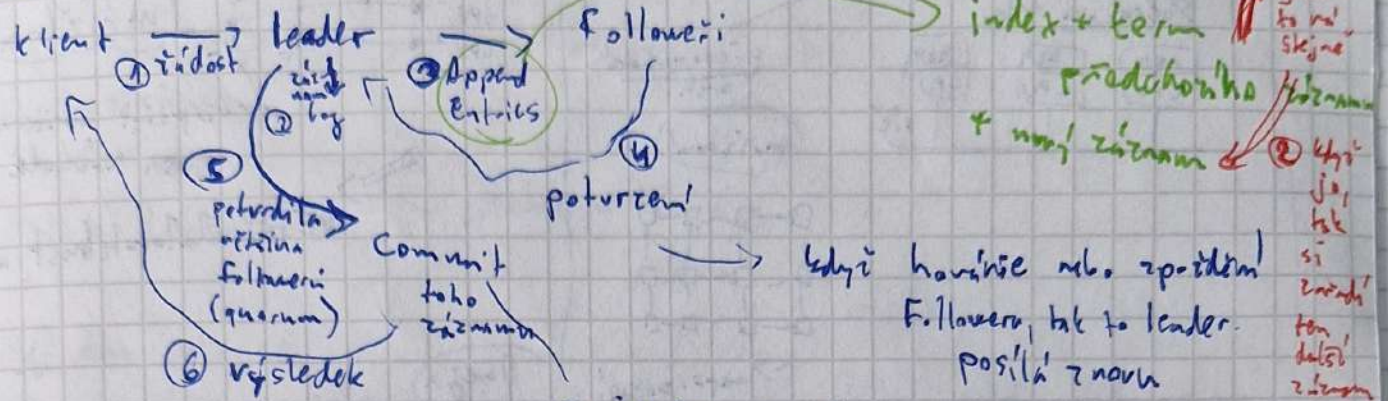
- ale je to složité a těžko pochopitelné !!

RAFT

- navržen, aby byl snadno pochopitelný a implementovatelný
- role:
  - leader – navrhuje
  - follower – akceptuje
  - candidate – dočasná role při volbě leadera
- komponenty:
  - volba leadera – on pravidelně posílá heartbeat a když to nedojde tak je to důvod pro volbu nového leadra (každý pál sekundu)
  - replikace logu – když leaderovi přijde nějaká zpráva od klienta, tak to řekne followerům
  - udržování konzistence záznamů

- term = zákl. jednotka provádění operace
  - na začátku volba leadera termu
  - identifikátor akt. termu se s každou zprávou všem posílá:
    - starý ⇒ reject
    - novější ⇒ uzel si to aktualizuje

- průběh:



V záznamu (indexované) obsahuje: • term • příkaz

V uzel má svou repliku logu záznamů

↳ průnik replik s akt. termem je ten commitovaný stav (ten z nich velikostí quora)

leader má pak postupně posílá rozšířené AppendEntries s dalšími předchozími záznamy, dokud se nestrefnou → follower si to odsud celé přepíše na to od leadera

slide RAFT-kmz. modrý do 5

- volba leaderů:
  - kdo nedostane heartbeat, tak se prohlásí za kandidáta a V pošle request vote
    ⇒ když dostane většinový # hlasů, tak se stane leaderem
  - konflikt - víc kandidátů: pomoc! randomizovaného timeoutu
    → do určitého timeoutu od začátku volby se nezvolí leader

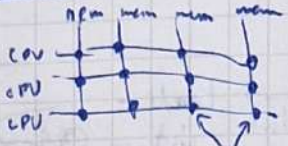
ZAB

- současná Yahoo Zookeeper framework
- účel: backupy systémů → záznamy se nesmí proházet

Distribuovaná sdílená paměť

Paral. arch.
- multiproc.
  - sdílená různá
  - sběrnicová arch.
  - přepínačová arch.
- multicomp. → uzel = vlastní paměť
  - sběrnicová arch.
  - přepínačová arch.

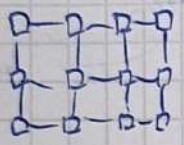
nízká škálovatelnost



mřížková

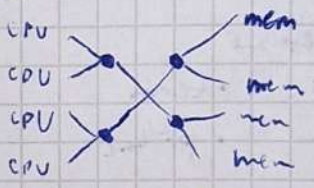
crosspoint switche
- na základě adresy přepíná

⇒ větší škálovatelnost, ale hw náročné!!
(závisí # přepínačů)



→ např. počasí

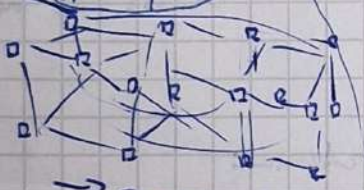
Omega



⇒ menší hw náročnost, ale stejně

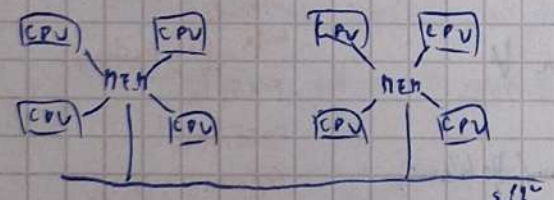
je potřeba dobře optimalizovat SW, aby se to rozprostřelo přes hw bloky paměti

hyperkrychle



→ superpočítače

NUMA



HW

GPGPU

CPU + GPU

RDMA

speciální síť. karta ⇒ přímé sdílení paměti ‼
s velmi nízkou latencí ‼‼‼

Zápis:
- W(x)a - zápis a do x
- R(x)a - přečtení z x
- S - synchronizace
  - Acq - vstup do kritické sekce
  - Rel - výstup z kritické sekce
- P_i - proces i

Signatura = výstupy jsou seřazeny podle uzlů

Konzistenční modely

- pro virt. paměť - bez přímé interakce s vnějšími procesy - transparentní
- pro sdílená data - pomocí frameworků → efektivnější

- striktní/atomická konzistence
  - všechny zápisy okamžitě všude viditelné
  - v distrib. systému nedosažitelné - musí být globální přesný čas

• sekvenční konzistence
- výsledek je ekvivalentní s tím, že by z jeden rozvrh, pořadí respektuje lok. pořadí operací na uzlu
→ ale zápis klidně může trvat dlouho, než se propíše
→ na V uzlu vidíme stejné změny proměnných, jen trošku v jiném čase
→ není determ. – opakovaně spuštěný stejný program může dát jiné výsledky
- nejsilnější model, který je rozumně možné implementovat v distrib. prostředí
- ale pomalý :(read ≥ t (čas přenosu zprávy)
↳ jde optimalizovat jen čtení, nebo zápis

• externí konzistence
- sekv. + čas propíšnutí omezen
- v distrib. databázích "citlivost lidského oka"

• kauzální konzistence
- zápisy, které se na sebe kauzálně váží, musí být na všech uzlech ve stejném pořadí
odeslání, čtení ≈ přijetí
$W(x)a \xrightarrow{P_1} R(x) \xrightarrow{P_2} W(y)b$
kaus. záv.
- složitější impl. – musíme mít ten graf závislostí

• PRAM (pipelined) konzistence
- zápisy jedním uzlem jsou na V uzly doručeny v tomhle lok. pořadí, ale zápisy různých uzlů mohou být viděny různě
- V uzel má vlastní rozvrh (X sekvenční konz.)

• Slow memory
- synchronizují se jen zápisy do jednoho místa

na úrovni správy virt. paměti – nepoužitelný! :(
:) podpora od jazyka atd. – jen propagaci zápisů; transparentní :), ale pomalejší :(
jednotkem je stránka

propagace až při opuštění :)

Konz. modely se synchr. proměnnou

- proces to přímo musí ovládat – třeba vstup do krit. sekce
- ale ztráta transparentnosti – musíme vědět, že pracujeme s distrib. pamětí, jak se ovládá ten framework, ...

Značí.: synchronizační proměnná → S, Acq, Rel

• Weak consistency
- přístupy v blocích ohraničených přístupy k synchr. proměnným → sekv. konzistentní → propagace hodnot

- prakticky nepoužitelné – nerozeznává se Acq a Rel – vždy se dělají synchronizace pro obě případy

- release consistency
  - Acq a Rel přísná konzistentní
    - ↳ dokončení zápisů i čtení
    - ↳ čtení
  - impl.:
    - eager rel. cons. – při Rel se všem rozešlou všechny aktuální verze dat
      ⇒ ostatní to dostanou hned ☺, ale zbytečně to zatěžuje síť !!
    - lazy rel. cons. – propaguje se až při Acq procesu
      ⇒ pošle se to jen těm, kteří to chtějí ☺, ale vyšší režie u Acq !!

- entry consistency
  - Acq a Rel jsou vázané na nějaké množiny (mohou se prolínat)
  - exkluzivní a neexkluzivní přístup k synchr. proměnné
    ↳ zápis ↳ čtení ⇒ může být jeden zapisovatel / více čtenářů

Distrib. stránkování

- po dobu, kdy je stránka načtená, nejde nijak detekovat čtení/zápis do ní
  ⇐⇒ to my ale potřebujeme kvůli té synchronizaci ⇒ dělá se read-only mapování ⇒ při zápisu dojde k přerušení

Nalezení stránky

- centralizovaný manager – eviduje, kdo má jaké stránky
  - v nějakých malých clusterech
- replikovaný manager
  - různé skupiny stránek spravovány různými managery (třeba podle spodních 4b)
  - už i pro větší clustery ☺
- broadcast (hw) – a majitel té stránky se sám přihlásí a pošle nám
  - menší clustery

Správa kopií

- broadcast – všichni, kdo má verzi 57 a nižší, ať si to zinvalidnjí
- copyset
  - zodpovědnost na to, kdo všechno má kopie té stránky, je přenesena na vlastníka stránky – a děláme to přes něj

Uvolňování stránek – Kleron vyhodit

- nevlastní read-only kopie – máme jistotu, že vlastník ji má, tak je to nikdo nepřijde
- vlastněná replikovaná kopie (v copysetu) → přeneseme vlastnictví na někoho, kdo má tu kopii
- lokální heuristika

False sharing

- prevence – hinty překladači, aby to řekl linkeru ⇒ ztráta transparentnosti !!
- hystereze – když máme stránku jenom chvilku a někdo ji chce, tak mu ji nedáme

Sekv. konz. stránkování

- při zápisu musí být stránka jen na jednom uzlu
- když se normálně čtou/zapisují namapované stránky, tak to nejde nijak zdetekovat
  - → akce až při výpadku stránky

my-think	mapovaná u vlastníka	
R	R	✓ ani nepoznáme
R	W	vlastník si ze stránky udělá R-O
W	W	my = vlastník
W	R (a copyset = ∅)	povýšíme na W
W	R	všem z copysetu řekneme, ať si to zinvalidují
W, mine R	R	vlastník nám předal vlastnictví, invaliduje si to
W	W	

Kauz. konz. stránkování

- vektorové hodiny – VT_S, VT_P, složky jsou stránky
  - VT_S → verze stránek v procesu P
  - na kterých stránkách stránka kauzálně závisí

Distrib. sdílené proměnné – na úrovni Frameworku – není abstrakce (ale efektivnější) (X stránkování)

- distrib. objekty ← základní kámen v těch Frameworcích – viz předmět Middleware

Epidemické protokoly – ve VELKÝCH rozsáhlých systémech

→ takže třeba broadcast by byl náročný – FUJ

→ fungují nějaké seznamy uzlů (ne moc velké)

→ eventuální konzistence

- antientropie – když má uzel nějakou informaci, tak ji pošle i na nějaký další (náhodný)
  - push – funguje dobře na začátku, když jich je ještě málo nakažených
  - pull – uzly si o ty informace samy říkají → na začátku pomalé rozšiřování
  - push+pull

- kdy ale přestat infikovat nové uzly?
  - → gossip protokol – když uzel natrefí na nějaký už infikovaný, tak se s nějakou pravděpod. přepne do klid. stavu

- problém mazání dat – jak zrušit nějakou informaci
  - naivní – prostě smažeme, ale z jiných uzlů se ta informace zase vrátí 😱
  - certifikát smrti – nová informace o zrušení té staré 🙂

- trénování much
  - spočítání jedna vyvolená, které se pak zeptáme, počítají si priméry
  - nejvěrnější moucha
  - celková váha

Správa prostředků a procesů

Detekce deadlocků

(kružnice ve Wait-for grafu (WFG))

- korektnost ⇔ 1) ∀ deadlock v kon. čase detekován
  2) ∀ det. deadlock opravdu ∃

Centralizovaný alg.

- do jednoho uzlu se reportují události o změnách WFG
- je potřeba mít (strikt. přísl.) prot. pro doručování zpráv (→ kauzálně kons.) ⇒ jinak falešné detekce
  nebo při podezření na deadlock nejdřív kontrola

Path-pushing

- v uzel si udržuje část WFG, která se ho týká
- vyměňují si externí závislosti + kandidáty na zabití (heuristicky nejčastěji nejmladší proces (s nejvyšší hodnotou log. hodin))
- průběžně se kontroluje, jestli není deadlock → když ano, pošle informace tomu uzlu, kde je ten kandidát na zabití

Edge-chasing

- podle WFG se routuje speciální značka, ve větvení se replikuje na ∀ větve (proces sbírá kandidáty na zabití)
  → když se dostane zpátky k iniciátorovi, tak on usoudí, že došlo k deadlocku

Diffusing computation – podobně jako alg. na detekci zastavení výpočtu (viz dříve)

- distrib. výpočet po WFG
- v praxi moc ne

Detekce glob. stavu

- "deadlock na nás počká" – dokud nerozbijeme tu kružnici
- najde se kauz. konz. řez

Distrib. správa procesů

- dělení dle toho, kde jsou ty procesy spuštěny: (služby klientům)
  - krátkodobé úlohy ⇒ chceme min. latenci
  - dlouhodobé ⇒ max. výkon
  - distrib. výpočty → chceme rovnoměrné rozložení zátěže

Load balancing / task scheduling

- replikovaný master (např. pomocí Paxosu) → rozděluje ty úlohy
  - round robin
    - na krátké / srovnatelně náročné úlohy
    - jednoduché ☺
  - weighted round robin – např. ½ serverovny jsou starší stroje, zbytek novější ⇒ vyšší váha ☺
  - dynamic round robin
    - měří se, jak moc jsou uzly vytížené – pravidelně posílají – a podle toho se master zařídí
  - least connections – podobné dynamic – ale master si jenom prostě pamatuje, kdo má kolik spuštěných procesů
  - random, threshold – možnost i toho, že uzly sám odmítnout, když už toho mají moc

• agent-based, ..., AI – spíš jenom aby dědci měli o čem psát

Koop. systémy – uzly po celém světě, volně organizovány
– když uzel není co dělat, tak si řekne centrálnímu registru o job (↑ autorované tím, co třeba potřebuji a tak..)

Koop. load balancing
• distrib. heuristický alg.
• det. graf. alg. – data-flow driven computing
  → toky v sítích
• bidding alg. – procesory nabízejí výp. sílu a procesy si ji kupují
  – nepoužívá se
• centralizované řízení ~ UP-down alg.
  – koordinátor si evidnuje ∀ uzel trestné body < procesy na jiném uzlu +, nenaplněné požadavky –
  → když se uvolní kap. uzlu, tak se vyberou ty s nejmenším # trestných bodů
  (požadavky z uzlu)

Migrace procesů/virt. strojů → přenášení procesů během výpočtu (↳ korektní, transparentní)
↳ dnes kontejnerů a virt. strojů

MOSIX - distrib. OS
– byl na něm implementován vekt. alg. pro řízení zátěže

T4
– z MFF

Migrační load balancing

Párový alg.
– náhodně se vytváří páry, účastník má seznam svých významných procesů a počítá se, o kolik by se zlepšilo to zatížení, kdyby se nějaké procesy přenesly na druhý uzel z páru

Vektorový alg.
– ∀ uzel si evidnuje vektor (veľ. menší než # uzlů) s vytížením uzlů
  ↳ 0. prvek je vlastní vytížení
– v nějaké periodě si ∀ uzel vybere náhodně # uzlů odpovídající půlce vektoru a od nich si vyžádá jejich horní polovinu vektoru (zbytek zahodí)
– v případě potřeby vyslání procesu se uzel podívá na ten vektor

z jiným spojí

Bidding alg. a další
– FUJ

Centralizovaný/hierarchický alg.

⇓

Lok. rand. alg.

⇓ ⇓

Distrib. dat. struktury

CRDT – Conflict-free Replicated Data Type ↙ chceme

– updaty se asynchronně přenáší mezi replikami

→ při konfliktech se udělá merge ⟶ konvergence ke stejným výsledkům ve všech replikách

State-based replication

– lokální úprava + async. poslání stavu ostatním – gossip prot.

– merge: komutativní, asoc., idempotentní, monotónní vzhledem k část. usp.

DF.: kauzální historie C replik x_i objektu x:

1) na začátku: $C(x_i) = \emptyset$

2) po op. update-f: $C(f(x_i)) = C(x_i) \cup \{f\}$

3) po op. merge stavů x_i, x_j: $C(merge(x_i, x_j)) = C(x_i) \cup C(x_j)$

ekvivalentní

Operation-based objekty

– nepřenáší se stavy, ale operace – není potřeba řídící merge operace

– operace jsou kom. a asoc., ne idemp.

– doručování v kauz. usp.! , je potřeba spolehlivý broadcast

Konvergence (eventual convergence)

$C(x_i) = C(x_j) \Rightarrow$ stavy i a j jsou ekvivalentní

$f \in C(x_i) \Rightarrow$ někdy nastane i $f \in C(x_j)$

Last-Writer-Wins registr

– každý update má svůj timestamp a ID

Množiny

– add a remove nejsou komutativní :(

řešení: • pomníčky (ať prvek patrně zmizí)
• prvky + ID