Distribuované systémy

poznámky k přednášce
(založeno na Tel G., Introduction to distributed algorithms.)

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Distribuovaný systém

Neformálně

- Množina procesů, které nesdílí paměť (a mohou běžet na různých počítačích, které jsou na různém místě)
- Procesy jsou spojeny komunikačním systémem (například sítí), mohou si posílat zprávy.
- Posílání zpráv je realizováno pomocí procedur **send** a **receive**. O odesílání zpráv budeme předpokládat, že je *asynchronní*, tj. že odesilatel nečeká na přijetí zprávy, ale pokračuje dál. Operace přijetí je naopak *blokující*, proces čeká až nějaká zpráva přijde.
- Odlišnosti v komunikačním systému: topologie, kterým procesům lze posílat zprávu, procesy nemusí znát kompletní topologii, chybovost apod.
- Pro zavedení pojmů a ověřování vlastností si zavedeme formální definici distribuovaného algoritmu a souvisejících pojmů.

Definition (Přechodový systém)

Přechodový systém je trojice $S = (C, \rightarrow, I)$, kde C je množina konfigurací, $\rightarrow$ je binární relace na C, a $I \subseteq C$ je množina počátečních konfigurací.
$(\gamma, \delta) \in \rightarrow$ značíme jako $\gamma \rightarrow \delta$.

Definition

Nechť $S = (C, \rightarrow, I)$ je přechodový systém. Exekuce S je maximální sekvence $E = (\gamma_0, \gamma_1, \gamma_2 \dots)$, kde $\gamma_0 \in I$ a $\gamma_i \rightarrow \gamma_{i+1}$ pro všechna $i \geq 0$.

Pokud je sekvence E maximální, pak je buď nekonečná, nebo končí *terminální konfigurací*, tj. konfigurací γ pro kterou neexistuje δ s $\gamma \rightarrow \delta$.

Definition

Konfigurace δ je dosažitelná z konfigurace γ, označeno $\gamma \rightsquigarrow \delta$, pokud existuje sekvence $\gamma = \gamma_0, \gamma_1 \dots, \gamma_k = \delta$ taková, že $\gamma_i \rightarrow \gamma_{i+1}$ pro $0 \leq i < k$. Konfigurace δ je dosažitelná, pokud je dosažitelná z některé počáteční konfigurace.

Distribuovaný systém

- kolekce procesů a komunikační systém
- v procesu probíhají události, pomocí kterých přechází ze stavu do stavu. Mimo vnitřních událostí může proces posílat zprávy (*send* událost) a přijímat zprávy (*receive* událost). Předpokládáme asynchronní zasílání zpráv.
- send událost vytvoří novou zprávu, receive událost konzumuje jednu zprávu. Množinu všech možných zpráv v systému budeme označovat jako $\mathcal{M}$. Typ zprávy kóduje i příjemce.

Definition (lokální algoritmus)

Lokální algoritmus je čtveřice $(Z, I, \vdash^i, \vdash^s, \vdash^r)$ kde Z je množina stavů, $I \subseteq Z$ je množina počátečních stavů, $\vdash^i \subseteq Z \times Z$ odpovídá vnitřním událostem, $\vdash^s \subseteq Z \times \mathcal{M} \times Z$ událostem send, a $\vdash^r \subseteq Z \times \mathcal{M} \times Z$ událostem receive. Binární relaci $\vdash$ na Z zavedeme jako

$$c \vdash d \text{ p.k. } (c,d) \in \vdash^i \text{ nebo existuje } m \in \mathcal{M} \text{ tak, že } (c,m,d) \in \vdash^s \cup \vdash^r .$$

Exekuce lokálního algoritmu je exekuce přechodového systému $(Z, \vdash, I)$.

Definition (Distribuovaný algoritmus)

Distribuovaný algoritmus pro procesy $\mathbb{P} = \{p_1, p_2, \ldots, p_N\}$ je kolekce lokálních algoritmů, jeden pro každý proces.

Definition

Přechodový systém $S = (C, \rightarrow, I)$ indukovaný distribuovaným algoritmem $\{(Z_j, I_j, \vdash_j^i, \vdash_j^s, \vdash_j^r) \mid j = 1, \ldots, N\}$ je dán následovně:

1. $C = \{(c_1, c_2, \ldots c_N, M) \mid c_j \in z_j$ pro $j = 1 \ldots N, M$ je multimnožina zpráv $\}$
2. $\rightarrow = (\bigcup_{j=1}^{N} \rightarrow_j)$, kde $\rightarrow_j$ je množina dvojic

$$\Big((c_1, \ldots, c_j, \ldots, c_N, M_1), (c_1, \ldots, c'_j, \ldots, c_N, M_2)\Big)$$

takových, že platí jedna z následujících podmínek:

- $(c_j, c'_j) \in \vdash_j^i$,
- pro $m \in \mathcal{M}$, $(c_j, m, c'_j) \in \vdash_j^s$ a $M_2 = M_1 \cup \{m\}$,
- pro $m \in \mathcal{M}$, $(c_j, m, c'_j) \in \vdash_j^r$ a $M_1 = M_2 \cup \{m\}$,

3. $I = \{(c_1, \ldots, c_N, M) \mid c_j \in I_j$ pro $j = 1 \ldots N$ a $M = \emptyset\}$

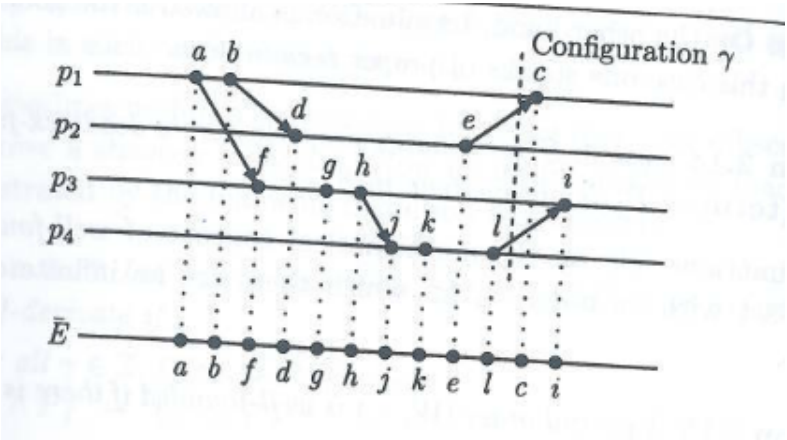
Proveditelnost události

Událost v procesu p je proveditelná z dané konfigurace distribuovaného systému, pokud je proveditelný příslušný přechod v přechodovém systému v předchozí definici. To znamená, že lokální proces musí být v odpovídajícím stavu a pokud je událost receive, odpovídající zpráva musí být v množině odeslaných (a ještě nepřijatých) zpráv.

Kauzální uspořádání událostí

Diagram exekuce

K exekuci $E = \gamma_0, \gamma_1 \ldots$ můžeme přiřadit sekvenci událostí $\overline{E} = e_0, e_1, \ldots,$ které E odpovídají, tj. e_i je událost, která vedla k přechodu z γ_i na γ_{i+1}. Konfiguraci γ_{i+1} pak značíme $e_i(\gamma_i)$.



Theorem (Nezávislost událostí)

Pro konfiguraci γ a události e_1 a e_2 různých procesů takové, že jsou obě proveditelné z konfigurace γ platí, že

1. *e_1 je proveditelná z konfigurace $e_2(\gamma)$, e_2 je proveditelná z konfigurace $e_1(\gamma)$,*
2. *$e_1(e_2(\gamma)) = e_2(e_1(\gamma))$.*

Idea důkazu.

Protože události mění stavy různých procesů, jediný způsob jakým by e_1 mohla interferovat s provedením e_2 je, že přijme zprávu, kterou chce přijímat i e_2. Ale to není možné, protože typ zprávy kóduje i příjemce. Odtud (a ze symetrického argumentu) plyne bod 1.

Pokud jsou obě události receive, přijímají různé zprávy a na jejich pořadí tedy nezáleží. Stejně tak pokud jsou obě události send. Pokud je první událost send a druhá receive, buď pracují s různými zprávami (pak tvrzení platí), nebo pracují se stejnou zprávou. Pak ale tvrzení plyne z toho, že obě události jsou proveditelné z γ, počet zpráv daného typu v M se nezmění, jedna přibude a poté jedna ubude. Tedy platí i bod 2.

□

Definition (Kauzální uspořádání)

Kauzální uspořádání $\prec$ exekuce E je nejmenší binární relace na událostech splňující:

1. Pokud jsou e a f různé události stejného procesu a e nastala dříve než f, pak $e \prec f$.
2. Pokud s je send událost a r je odpovídající receive událost (tj. událost přijetí fyzicky stejné zprávy), pak $s \prec r$
3. $\prec$ je tranzitivní.

Poznámky

Píšeme $a \preceq b$ pokud $a \prec b$ nebo $a = b$. $\preceq$ je částečné uspořádání, mohou existovat neporovnatelné prvky. Takovým dvojicím říkáme, že jsou $konkurentní$ a značíme $a||b$.

$a \prec b$ implikuje existenci kauzálního řetězu, tj. sekvence $a = e_0, e_1, \ldots, e_k = b$ takové, že po sobě jdoucí události odpovídají bodu 1 nebo bodu 2 definice.

Výpočet distribuovaného algoritmu

Implikovaná exekuce

Viděli jsme, že exekuce distribuovaného algoritmu $F = \gamma_0, \gamma_1 \dots$ jednoznačně indukuje sekvenci událostí $f = e_0, e_1, \dots$ (tak, že $e_i(\gamma_i) = \gamma_{i+1}$).

Uvažujme sekvenci událostí $f = e_0, e_1, \dots$. Pokud existuje exekuce $F = \gamma_0, \gamma_1 \dots$ tak, že $e_i(\gamma_i) = \gamma_{i+1}$, říkáme jí *implikovaná exekuce*. Nemusí být unikátní, ale pokud f obsahuje alespoň jednu událost z každého procesu, tak uníkátní je.

Konzistentní uspořádání

Uvažme exekuci E a jí odpovídající sekvenci událostí $f = e_0, e_1, \dots$. Uvažme sekvenci událostí $f' = e'_0, e'_1, \dots$, která vznikne permutací f. Řekneme, že f' je *konzistentní* s kauzalním uspořádáním $\preceq$ exekuce E, pokud $e'_i \preceq e'_j$ implikuje $i \leq j$.

Theorem

Nechť $f = f_0, f_1 \ldots$ je permutace událostí exekuce E, která je s E konzistentní. Potom f implikuje unikátní exekuci F se stejnou počáteční konfigurací jako E. Pokud je E konečná, poslední konfigurace F je stejná, jako poslední konfigurace E.

Důkaz.

Značení: $E = \gamma_0, \gamma_1 \ldots$; $\overline{E}$ je odpovídající sekvence událostí.

Konstrujeme $F = \delta_0, \delta_1 \ldots$, začneme od $\delta_0 = \gamma_0$. Musíme ukázat, že f_i je aplikovatelná v konfiguraci δ_i, pak stačí vzít $\delta_{i+1} = f_i(\delta_i)$. Předpokládejme, že to platí pro $j < i$. Označme $f_i = (c, x, y, d)$ (poč. stav, přijatá, odeslaná, konc. stav) a předpokládejme, že nastala v procesu p; dále označme $\delta_i = (c_1, \ldots, c_N, M)$. f_i je aplikovatelná, pokud $c_p = c$ a $x \in M$.

Dokážeme, že $c_p = c$. Máme dva případy:

- f_i je první událost v p, pak je ale i první událost v p vzhledem k $\overline{E}$ (kvůli kauzalnímu uspořádání), $c = c_p$ je tedy počáteční stav procesu p.
- těsně před f_i je v p event f_i' (f_i pokrývá f_i' v uspořádání $\preceq$), kauzální uspořádání ale implikuje, že f_i' je těsně před f_i i vzhledem k $\overline{E}$. Tudíž stav ve kterém je p je v obou případech stejný, a tedy $c_p = c$.

pokračování.

Ukážeme, že $x \in M$. Pokud f_i není receive událost, tvrzení je triviální. Pokud je f_i receive událost, musí existovat korespondující send událost f_j, a protože $f_j \prec f_i$, máme $j < i$, a tedy $x \in M$.

Pokud jsou E a F konečné, stavy procesů se v terminálních konfiguracích se rovnají, protože jsou to buď počáteční stavy v daných procesech (pokud neproběhla v procesu žádná událost), nebo je ekvivalence stavů způsobena kauzálním uspořádáním událostí v rámci jednoho procesu (které musí být v obou případech stejné). Množina zpráv je u obou konfigurací stejná, protože f a $\overline{E}$ obsahují stejné send a receive události. □

Pokud mají exekuce F a E stejnou množinu událostí a kauzální uspořádání těchto událostí je stejné, opravňuje nás předchozí věta k tomu, abychom prohlásili E a F za *ekvivalentní*. (Z pohledu jednotlivých procesů, nikoliv z pohledu zvenku.)

Definition

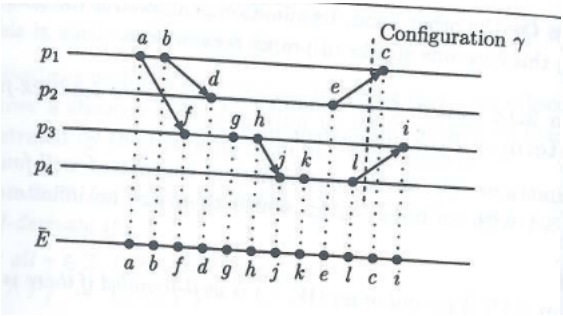
Výpočet distribuovaného algoritmu je ekvivalenční třída exekucí daného algoritmu.

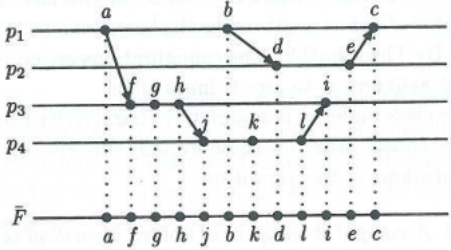
Z předchozí diskuze je zřejmé, pro popis výpočtu algoritmu stačí znát jeho události a jejich kauzální uspořádání.

Lemma

Nechť $(X, <)$ je částečně uspořádaná množina a $b \not< a$. Potom existuje lineární rozšíření $<_l$ uspořádání $<$ takové, že $a <_l b$.

Ekvivalentní exekuce





Logické hodiny

Definition (Logické hodiny)

Logické hodiny jsou funkce Θ zobrazující události na prvky uspořádané množiny $(X, <)$ tak, že

$$\text{jestliže } a \prec b \text{ pak } \Theta(a) < \Theta(b)$$

Jednoduché hodiny

Pro exekuci E implikovanou sekvencí událostí $e_0, e_1, \ldots$ můžeme definovat $\Theta(e_i) = i$. Tento čas je pozorovatelný z vnějšku distribuavaného systému, ale nikoliv zevnitř (viz předchozí slajdy.)

Lamportovy logické hodiny Θ_L

Události a přiřadíme délku k nejdelší posloupnosti událostí takové, že

$$e_1 \prec e_2 \prec e_3 \cdots \prec e_k = a.$$

Pak zcela jistě platí, že pokud $a \prec b$, pak $\Theta_L(a) < \Theta_L(b)$.

Θ_L lze spočítat distribuovaným algoritmem založeným na následujím

1. V každém procesu zavedeme iniciální událost a a řekneme, že $\Theta_L(a) = 0$.
2. Pokud je a vnitřní událost nebo send událost a a' je předchozí událost ve stejném procesu, pak $\Theta_L(a) = \Theta_L(a') + 1$.
3. Pokud je a receive událost, a' je předchozí událost ve stejném procesu a b je send událost odpovídající a, pak $\Theta_L(a) = \max(\Theta_L(b), \Theta_L(a')) + 1$.

```
# lokalni kopie casu
var theta_p : int = 0

# vnitrni event
theta_p := theta_p + 1;
change_state();

# send event
theta_p := theta_p + 1
send (messg, theta_p);
change_state();

# receive event
receive (messg, theta);
theta_p = max(theta_p, theta) + 1;
change_state();
```

Vektorové hodiny

Chceme aby čas zachycoval i paralelismus a paralelní události měly neporovnatelný čas, tj.

$$a \prec b \text{ p.k. } \Theta(a) < \Theta(b)$$

Uvažujeme usp. množinu $(\mathbb{N}^N, \leq_v)$, přičemž $\leq_v$ je definováno jako porovnání po souřadnicích, tj. $(a_1, a_2, \ldots, a_N) \leq_v (b_1, b_2, \ldots, b_N)$ p.k. $a_i \leq b_i$ pro všechna i.

Definice hodin je $\Theta_v(a) = (a_1, a_2, \ldots, a_N)$, kde a_i je počet událostí e v procesu i, pro které $e \prec a$. Hodiny lze počítat úpravou algoritmu, který počítá Lamportovy hodiny (viz následující slajd).

Důkaz zachycení paralelismu vektorovými hodinami je ponechán jako cvičení.

```
# aktualni proces je p
var theta : int[1:N] = ([N] 0);

# vnitrni event
theta[p] := theta[p] + 1
change_state()

# send event
theta[p] := theta[p] + 1
send (messg, theta)
change_state()

# receive event
receive (messg, theta_received);
theta[p] := theta[p] + 1
for i := 1 to N such that i != p
  theta[i] = max(theta[i], theta_received[i])
change_state()
```

Vlnové a průchodové algoritmy

událost decide
je speciální typ vnitřní události procesu

Definition

Vlnový algoritmus je distribuovaný algoritmus, který splňuje tři podmínky

1. Každý výpočet je konečný.
2. Každý výpočet obsahuje alespoň jednu *decide* událost.
3. Pro každý výpočet systému platí, že pro každou *decide* událost existuje v každém procesu událost, která *decide* událost kauzálně předchází.

Poznámky

- Jednomu výpočtu vlnového algoritmu říkáme vlna
- Proces je *iniciátor* vlnového algoritmu, pokud jej začne provádět z vnitřního popudu (tedy ne jako reakci na zprávu). Ostatní procesy jsou *následovníci* (jejich první událost je tedy receive).

Různé algoritmy pro různé systémy

- Centralizované (1 iniciátor) vs. necentralizované (více iniciátorů)
- Mohou být určeny pro speciální topologii komunikační sítě.
- Počáteční znalosti: identita procesu, identita sousedů procesu
- Počet decide událostí (např. 1 celkem vs. 1 v každém procesu apod.)
- Složitost: počet odeslaných zpráv, počet odeslaných bitů, čas jednoho výpočtu

Základní vlastnosti vlnových algoritmů

Lemma

Pro každou událost e ve výpočtu vlnového algoritmu existuje iniciátor p a událost f v p taková, že $f \preceq e$.

Důkaz.

f vybereme jako jeden z minimimálních prvků, které jsou v kauzální uspořádání pod e. Takový prvek určitě existuje, protože každá událost má konečnou historii (= sekvenci událostí, které jsou kauzálně před ní).

Předpokládejme, že f je v procesu p. Musí to být první událost v p, protože jinak by nebyla minimální. Ze stejného důvodu nemůže být f receive událost, odpovídající send událost by totiž byla kauzálně menší. Proces p tedy musí být iniciátor. □

Lemma

Uvažme vlnu s jedním iniciátorem p. Pro každý následnický proces r označme jako father(r) ten proces, který poslal zprávu, která v r vlnu zahájila. Potom je graf definovaný $T = (\mathbb{P}, E_T)$ s hranami

$$E_T = \{(q, r) \mid q \neq p \wedge r = \mathit{father}(q)\}$$

orientovaná kostra směřující k p.

Důkaz.

Počet uzlů v grafu je o 1 vyšší než počet hran (každý proces mimo p obdrží právě jednu *zahajovací* zprávu).

Pokud je e_q první událost v procesu q a $(q, r) \in E_T$, pak pro e_r (první událost v procesu r) určitě platí $e_r \prec e_q$. Protože $\prec$ je částečné uspořádání (= je bez cyklů), neobsahuje ani T cykly. Nejmenší prvek $\prec$ je určitě v procesu p, graf je tedy orientovaný k p. □

Lemma

Uvažme vlnu a decide event d_p v procesu p. Potom v každém procesu q různém od p existuje událost e_q taková, že $e_q \prec d_p$ a e_q je send událost.

Důkaz.

Z definice vlnového algoritmu existuje v procesu q událost e_q tak, že $e_q \prec d_p$. Můžeme předpokládat, že e_q je poslední (= kauzálně největší) událost v procesu q, pro kterou to platí. Uvážíme-li pak kauzální řetěz

$$e_q = f_0, f_1 \dots f_k = d_p,$$

vidíme, že f_1 je událost v jiném procesu než je q, a tudíž e_q musí být send událost. □

Theorem

Nechť C je vlna s jedním iniciátorem p taková, že decide událost d_p nastane také v procesu p. Potom v algoritmu dojde k výměně alespoň N zpráv (= jsou odeslány a přijaty).

Důkaz.

Z předchozích lemmat plyne:

- každá událost v C je kauzálně předejita událostí v p, z kauzálního řetezu (viz předchozí důkaz) vidíme, že jedna taková událost je send.
- decide událost d_p je předejita send událostí ve všech procesech mimo p

To celkem dává N vyměněných zpráv. □

Theorem

Nechť A je vlnový algoritmus pro systém s libovolnou topologií, v němž procesy na začátku neznají identitu svých sousedů. Potom v každém výpočtu A dojde k nejméně $|E|$ výměnám zpráv, kde $|E|$ je počet hran v topologii systému.

Důkaz.

Sporem. Předp. že existuje výpočet C systému A, kde dojde k poslání méně než E zpráv. Potom existuje hrana, po které nejde žádná zpráva, označme ji (p, q). Nyní bychom mohli rozšířit systém o nový proces r, který umístíme tak, že zmizí hrana (p, q) a přibudou hrany (p, r) a (r, q) (procesy na začátku neznají identity svých sousedů). Pokud nyní spustíme C (počáteční stavy všech procesů jsou stejné před a po rozšíření), nastave decide událost, která není kauzálně predejita událostí v procesu r. □

Šíření informací s potvrzováním

Broadcast s potvrzením

- Podmnožina procesů má zprávu m.
- Všechny procesy musí obdržet m.
- Některé procesy musí být notifikovány o tom, že všechny procesy obdrželi m. To znamená, že existuje speciální událost *notify*, kterou dané procesy provedou, až poté (při pohledu zvenku), co všechny procesy obdrží m.

Událost *notify* budeme považovat za událost decide. Potom

Theorem

Každý algoritmus pro broadcast s potvrzením je vlnovým algoritmem.

Důkaz.

Nechť P je algoritmus pro broadcast s potvrzením. Každý výpočet P je konečný (končí po událostech notify) a obsahuje notify událost. Pokud by existovala notify událost v procesu p, před kterou (kauzálně) nenastala žádná událost v procesu q, potom existuje exekuce P, ve které nastane událost notify předtím, než q dostane jakoukoliv zprávu. □

Theorem

Každý vlnový algoritmus lze využít jako algoritmus pro broadcast s potvrzením.

Důkaz.

Uvažme vlnový algoritmus A. K broadcastu jej můžeme využít následovně:

- iniciátoři jsou právě procesy, které mají zprávu m
- m posíláme s každou zprávou, kterou A posílá; toto lze provést, protože iniciátoři m znají a první událost následovníků je přijetí zprávy obsahující m
- decide událost v A interpretujeme jako notify, z definice vlnového algoritmu potom plyne, že tato událost nastane až po přijetí m všemi procesy.

□

Synchronizace

Problém synchronizace

- v každém procesu q musí nastat událost a_q
- v některých procesech p nastane událost b_p
- události b_p nastanou až poté, co nastanou všechny události a_q
- k synchronizaci musí být využito konečné množství zpráv

Theorem

Každý synchronizační algoritmus je vlnovým algoritmem.

Důkaz.

Nechť S je synchronizační algoritmus. Každý výpočet S musí být konečný (konečné množství zpráv, nastanou události b_p). Pokud by existovala událost b_p, která není kauzálně předejita událostí a_q (pro nějaká p a q), pak existuje exekuce S, ve které nastane b_p před a_q. Události b_p jsou pro nás decide událostmi. □

Theorem

Každý vlnový algoritmus lze využít jako algoritmus pro synchronizaci.

Důkaz.

Nechť A je vlnový algoritmus. Abychom ho využili pro synchronizaci:

- Každý proces q musí vykonat a_q předtím, než pošle jakoukoliv zprávu
- Procesy vykonávající b_p ji vykonají, až poté, co vykonají decide událost.

Z předchozích tvrzení plyne, že každá událost b_p je kauzálně předejita všemi událostmi a_q. □

Výpočet infima

Problém výpočtu infima

- každý proces obsahuje prvek částečně uspořádané množiny $(X, \leq)$, ve které existuje infimum pro všechny dvojice prvků (tj. $(X, \leq)$ je průsekový polosvaz).
- chceme spočítat infimum těchto prvků tak, aby bylo známo ve vybrané množině procesů. Tyto procesy musí umět poznat, že výpočet infima je již u konce, a zpracovat jej (například vypsat).

Theorem

Každý algoritmus pro problém výpočtu infima je vlnový algoritmus.

Důkaz.

Za decide událost budeme považovat událost vypsání infima. Uvažme výpočet C algoritmu pro výpočet infima. Předp. (pro účely odvození sporu), že v výpočtu C provede vypsání infima (událost w) proces p, a že v procesu q neexistuje kauzálně menší událost. □

pokračování.

Uvažme situaci, ve které změníme počáteční konfiguraci výpočtu C tak, že proces q obsahuje jinou počáteční hodnotu (ozn. k), která bude menší než infimum spočtené v C (ozn $\inf_C$). Protože žádná událost v q kauzálně nepředchází vypsání decide události, všechny události v p, které předchází w, lze provést ve stejném pořadí jako v C. To znamená, že proces p vypíše $\inf_C$, ale novým infimem je k. □

Theorem

Každý vlnový algoritmus lze použít pro výpočet infima.

Důkaz.

Předpokládejme, že každý proces q má proměnnou i_q obsahující potřebnou hodnotu j_q, a že máme k dispozici vlnový algoritmus A. Infimum je potom $J = \inf_q j_q$. Procesy, které mají J vypsat jsou ty, které provedou decide událost. Pokud proces p během výpočtu A

- pošle zprávu, tak s ní pošle i kopii aktuální hodnoty i_p,
- přijme zprávu od q, která obsahuje i_q, nastaví i_p na $\inf(i_p, i_q)$.

Označme jako v_a hodnotu příslušné proměnné i_- po události a. Očividně platí, že $j_q \geq v_a$, pokud a nastala v procesu q. Z předchozího odstavce plyne, že pokud $a \preceq b$ pak $v_a \geq v_b$ (a, b mohou být v různých procesech) a navíc $J \leq v_a$. Protože A je vlnový algoritmus a v každém procesu existuje událost kauzálně menší než decide událost d, musí být $j_q \geq v_d$ pro každé q, a tedy nutně $v_d = J$. □

Příklady operace infimum

Theorem

Pokud je $\circ$ binární operace na množině X, která je komutativní, asociativní, a idempotentní, pak existuje částečné uspořádání $\leq$ na X takové, že $\circ$ je infimum na X.

Příklady $\circ$

- konjunkce, disjunkce
- maximum, minimum
- gcd, lcm
- průnik, sjednocení

Algoritmus pro kruhovou topologii

- Kruhová topologie: každý proces má jednoho souseda, kterému může posílat zprávy, procesy s orientovanými hranami odpovídajícími směru posílání zpráv tvoří kružnici. (Lze získat například po výpočtu Hamiltonovské kružnice.)
- Centralizovaný algoritmus s jedním iniciátorem, a jednou decide událostí, princip posílání spec. zprávy — *tokenu*.

```
# ALGORITMUS RING
# soused je v promenne next

# iniciator
send(token) to next
receive(token)
decide

# ostatni procesy
receive(token)
send(token) to next
```

Theorem

Algoritmus RING je vlnový algoritmus.

Důkaz.

Označme iniciátora jako p_0. Algoritmus je konečný, skončí v momentě, kdy p_0 přijme zprávu s tokenem (žádný proces ho nedrží).

Z toho, že topologie je kruhová plyne, že každý proces mimo p_0 přijal a odeslal token, který se poté dostal do p_0. Tyto události tedy tvoří kauzální řetěz a decide události kauzálně předchází událost v každém procesu. □

Stromový algoritmus

- Topologie je strom pokrývající všechny procesy (např. kostra v obecné topologii).
- Iniciátoři: listy = procesy s jedním sousedem, decide ve všech procesech

```
# ALGORITMUS STROM
# proces ma S sousedu v mnozine NEIGH

var rec[S] := ([S] false)

while |{q: rec[q] == false}| > 1 do
   q := receive(tok)
   rec[q] := true

send(tok) to q_0 with rec[q] == false
receive(tok) from q_0

decide()
forall q in NEIGH s.t. q != q_0 do
   send(tok) to q
```

Theorem

Algoritmus STROM je vlnový algoritmus.

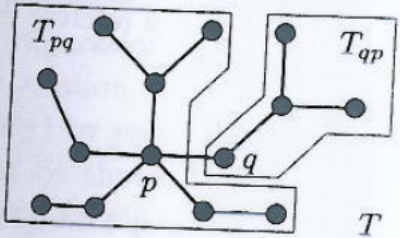
Důkaz.

Označme jako $e_{p\to q}$ událost odeslání tokenu procesu q procesem p, a $f_{p\to q}$ událost přijetí tohoto tokenu. Jako T_{pq} označíme množinu procesů dosažitelných z p pomocí cesty, která neobsahuje hranu (p, q). Potom indukcí přes pořadí receive událostí dokážeme

$$\text{pro každé } r \in T_{pq} \text{ existuje událost } e \text{ procesu } r \text{ tak , že } e \preceq f_{p\to q}.$$

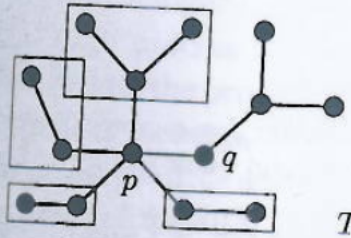
Odtud vyplyne, že před každou decide událostí se stala v každém procesu alespon jedna kauzálně menší událost (viz obrázek na dalším slajdu).
Vidíme, že

- $e_{p\to q} \prec f_{p\to q}$,
- v procesu p pro všechny $r \in \mathrm{NEIGH}$ takové, že $r \neq q$, platí $f_{r\to p} \prec e_{p\to q}$,
- podle indukční hypotézy aplikované na hranu (r, p) pro sousedy $r \neq q$ procesu p dostaneme, že pro každé $s \in T_{r\to p}$ existuje událost e (tohoto procesu) a $e \prec f_{r\to p}$, a tedy $e \prec f_{p\to q}$.



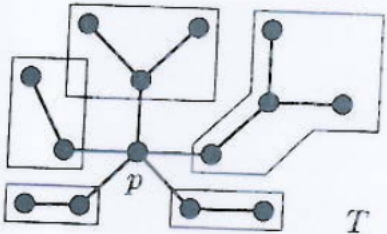
T_{pq} and T_{qp}

T



T

Decomposition of T_{pq}



T

Decomposition of T

pokračování.

Pokud bychom z algoritmu vynechali posledni cyklus, zjistíme, že algoritmus terminuje v konfiguraci, kdy dva procesy provedou decide (dokázat to lze sporem: předp. ze takové procesy neexistují, vybrat si jeden list a jit cestou „sousedu, od kterých daný proces neobdržel token“, jednou bychom museli narazit na situaci, kdy lze provést událost send). Ostatní procesy pak čekají na své poslední události receive. □

Echo algoritmus

```
# ALGORITMUS ECHO
var rec := 0, father := undef

# iniciator
forall q in NEIGH do
   send(tok) to q
while rec < |NEIGH| do
   receive(tok); rec := rec + 1
decide()

# ostatni procesy
receive(tok) from q in NEIGH; father := q; rec := rec + 1

forall q in NEIGH s.t. q != father do
  send(tok) to q
while rec < |NEIGH| do
  receive(tok); rec := rec + 1

send(tok) to father
```

- Centralizovaný algoritmus pro libovolnou topologii.

Theorem

ECHO je vlnový algoritmus

Důkaz.

Každý proces posílá pouze konečný počet zpráv, takže se algoritmus dostane po konečném počtu kroků do koncové konfigurace. Označme iniciátora jako p_0. Pro každý výpočet algoritmu můžeme definovat graf $G = (\mathbb{P}, E)$ pomocí

$$(p, q) \in E \text{ p.k. } \texttt{father}_p = q.$$

Je vidět, že G je kostra. Každý uzel mimo p_0 má jednoho rodiče (proměnná `father`), tedy $|E| = |\mathbb{P}| - 1$. Spojitost G plyne z povahy algoritmu: proces, který má určeného rodiče, posle zprávu všem ostatním sousedům a tedy, každý proces mimo p_0 má proměnnou `father` definovánu. □

pokračování.

G má kořen v p_0. Označme jako T_p podstrom G s kořenem p; jako e_p událost, kdy p pošle token svému rodiči; jako f_p událost, kdy rodič p přijme token od p.
Indukcí přes podstromy v G ukážeme, že

1. výpočet obsahuje e_p pro každé $p \neq p_0$
2. pro všechny $s \in T_p$ existuje událost e v procesu s taková, že $e \prec f_p$.

Máme dva případy:

- p je list v G. Potom p dostal token od všech svých sousedů a provedl e_p, $e_p \prec f_p$ plyne z definice.
- p není list v G. Z indukčního předpokladu pro každého přímého potomka p' (tedy na $T_{p'}$) víme, že určitě nastalo $e_{p'}$ a tedy p obdržel token od všech sousedů mimo předka, a akce e_p je proveditelná. Z toho, že $f_{p'} \prec e_p$ a indukčního předpokladu plyne 2.

□

Fázový algoritmus

- Decentralizovaný
- Libovolná topologie
- Orientovaná topologie: po hrane lze posílat zprávy jenom jedním směrem, IN a OUT množiny sousedu
- Je potřeba znát poloměr grafu D (maximum z nejkratších vzdáleností mezi uzly v grafu), nebo alespoň její horní odhad $D' \geq D$.
- každému OUT sousedovi pošleme D zpráv, i-tou správu pošleme až máme přijatou $i-1$ zprávu od každého IN souseda.

```
# algoritmus PHASE,
# OUT odchozi sousede,
# IN prichozi sousede

const D
var rec[D] := ([|IN|] 0),
sent       := 0

if i_am_initiator then
   forall r in OUT do send(token) to r;
   sent := sent + 1

while min(rec) < D do
   q_0 := receive (token)
   rec[q_0] := rec[q_0] + 1
   if min(rec) > sent and sent < D then
      forall r in OUT do send(token) to r;
      sent := sent + 1

decide()
```

Theorem

PHASE je vlnový algoritmus

kostra.

Každý proces pošle svému OUT sousedu max D zpráv, algoritmus tedy končí.

Každý proces q pošle alespoň jednu zprávu:

- existuje cesta od iniciátora ke q.
- Po této cestě se šíří jeden token, který procesy posílají, když jsou (a) iniciátoři nebo (b) dostali aspoň jeden token od svého IN souseda (v tomto momentě je `sent` rovno 0).

Každý proces provedl *decide*

- řekněmě, že ve finální konfiguraci má proces p minimální hodnotu proměnné `sent`, tedy že $\texttt{sent}_p \leq \texttt{sent}_q$ pro všechna $q \in \mathbb{P}$.
- speciálně pro každého IN souseda q platí, že $\texttt{rec[q]}_p = \texttt{sent}_q$ a tedy $\min \texttt{rec}_p \geq \texttt{sent}_p$. To implikuje, že $\texttt{sent}_p = D$, protože jinak by p posílal zprávy (a algoritmus by nebyl u konce). Odtud plyne, že všechny procesy provedli *decide*.

pokračování.

Před *decide* v procesu p proběhla v každém procesu kauzálně menší událost

- po 1. vlně posílání zpráv proběhla víme, že proběhla kauzálně menší událost v IN sousedech p.
- po 2. vlně víme, že proběhla kauzálně menší událost v IN sousedech IN sousedů p (IN vzdálenost 2).
- po D-té vlně víme, že proběhla kauzálně menší událost ve všech procesech v IN vzdálenosti D.
- maximální IN vzdálenost je omezena právě D.

□

Průchodové algoritmy

Definition

Průchodový algoritmus je vlnový algoritmus, který splňuje následující:

1. Existuje pouze jeden iniciátor, který zahájí algoritmus posláním jedné zprávy.
2. Proces, který přijme zprávu, buď odešle jednu zprávu nebo provede decide.
3. Algoritmus skončí v iniciátorovi, který provede decide. Když se to stane, každý proces poslal zprávu alespoň jednou.

Definition

Algoritmus je f-průchodový algoritmus (pro třídu topologií) pokud

1. je to průchodový algoritmus
2. v každém výpočtu bylo po zaslání $f(x)$ zpráv navštíveno nejméně $\min N, x + 1$ procesů.

Algoritmy

Vlnový algoritmus RING je x-průchodový algoritmus

Algoritmus POLL pro klikovou topologii je x-průchodový algoritmus (detaily na tabuly)

Tarryho algoritmus pro obecnou topologii se řídí pravidly

- Proces nikdy nepošle dvě zprávy stejným kanálem
- Proces pošle token svému rodiči (= prvnímu procesu, od kterého dostal token) pouze když nemá jinou možnost (viz předchozí pravidlo).

```
var used[] := ([N] false), father := udef;

if i_am_initiator then
  father := my_id
  choose q from NEIGH; used[q] := true; send(tok) to q

# reakce procesu pri prijeti zpravy od procesu q0
if father == udef then father := q0

if foreach q in NEIGH : used[q] then
  decide
else if there is q in NEIGH : q != father and not used[q] then
  choose q from NEIGH \ {father} s.t. not used[q]
  used[q] := true
  send(tok) to q
else
  used[father] := true
  send(tok) to father
```

```
# klasicky algoritmus pro pruchod do sirky
var used[] := ([N] false), father := udef;

if i_am_initiator then
  father := my_id
  choose q from NEIGH; used[q] := true; send(tok) to q

# reakce procesu pri prijeti zpravy od procesu q0
if father == udef then father := q0

if foreach q in NEIGH : used[q] then decide
else if there is q in NEIGH : q != father and not used[q] then
  if father != q0 and not used[q0] then
    q := q0
  else
    choose q from NEIGH \ {father} s.t. not used[q]
  used[q] := true
  send(tok) to q
else
  used[father] := true
  send(tok) to father
```

```
# DFS se znalosti identit sousedu
var father := udef

if i_am_initiator then
  father := p;
  choose q from NEIGH
  send ({p}) to q

# po prijeti zpravy od q0 obsahujici mnozinu L
if father = udef then father := q0
if there is q in NEIGH \ L then
  choose q from NEIGH \ L
  send(L ∪ {my_id}) to q
else if i_am_initiator then decide
else send (L ∪ {my_id}) to father
```

```
# volba lidra ve stromu
var ws := false, wr := 0
var rec[] := ([NEIGH] false), v := my_id, state := sleep
# wakeup faze
if i_am_initiator then
  ws := true
  forall q in NEIGH do send(wakeup) to q
while wr < |NEIGH| do
  q := receive(wakeup); wr := wr + 1
  if not ws then
    ws := true;
    forall q in NEIGH do send(wakeup) to q
# vypocet minimalniho id, volba lidra
while |{q : not rec[q]}| > 1 do
  q,r := receive(tok)
  rec[q] := true
  v := min(v, r)
send(tok,v) to q_0 s.t. not rec[q_0]
r := receive(tok) from q_0
v := min(v, r)
if v == my_id then state := leader else state := lost
forall q in NEIGH st. q != q0 do send(tok, v) to q
```

```
#extintion echo algoritmus
var caw := udef, rec := 0, father := udef; lrec := 0, win := udef
# iniciator zacne vlnu se svym id
if i_am_initiator then
  caw := my_id;
  forall q \in NEIGH do send(tok,my_id) to q;

while lrec < |NEIGH| do
  (tag, r) := receive() from q;
# (prvni) obdrzena zprava o leaderovi
  if tag = ldr
    if lrec = 0
      forall q \in NEIGH do send(ldr,r) to q;
      lrec += 1; win := r;
  else
# prejdu na zacatek algoritmu procesu r
      if r < caw then
        caw := r; rec := 0; father := q;
      forall s in NEIGH, s != q do send(tok,r) to s
# pokracuju ve vlnovem algoritmu procesu r
    if caw == r then
      rec := rec + 1;
# vlnovy algoritmus procesu r dokoncen pro my_id
      if rec == |NEIGH| then
        if caw == my_id then forall s \in NEIGH do send(ldr, my_id) to s
        else send (tok,caw) to father
if win = my_id then state := leader lese state := lost
```

```
# the basic distributed algorithm
var state

# S {state = active}
send (<mes>)

# R { A message <mes> has arrived }
receive (<mes>)
state := active

# I { state = active }
state := passive
```

```
# Announce algorithm
var sent_stop := false
var rec_stop := 0

Procedure Announce:
  if not sent_stop then
    sent_stop := true
    forall q in OUTNEIGH do send(stop) to q

# a stop message has arrived
receive (stop);
rec_stop := rec_stop + 1
Announce()
if rec_stop == |INNEIGH| then halt
```

```
# Dijkstra-Scholten algoritmus for termination detection
state  := if my_id == p_0 then active else pasive
sc     := 0
father := if my_id == p_0 then my_id else udef

# S { state == active }
send (mes, p); sc := sc + 1

# R { a message (mes, q) arrived }
receive (mes, q); state := active
if father = udef then father := q else send (sig,q) to q

# I { state == active }
state := passive
if sc == 0 then
  if father == my_id then Announce()
  else send (sig, father) to father
  father := udef

# A { a signal (sig, my_id) arrives }
receive (sig, my_id); sc := sc - 1
if sc == 0 and state == passive then
  if father == my_id then
    Announce()
  else send (sig, father) to (father)
  father = udef
```

```
# Safra's algorithm for termination detection
# RING_NEIGHBOUR je soused v kruhove subtopologii pro vlnovy algoritmus
# iniciatori basic algoritmu maji nastaveno state := active, color := white
# ostatni passive a white
var state
var color
var mc    := 0

# S { state == active }
send(mes); mc := mc + 1;

# R { a message mes has arrived }
receive(mes);
state := active
mc    := mc - 1
color := black

# I { state == active }
state := passive

# T { state == passive, zpracuje token (tok, col, sum) }
if i_am_wave_initiator then
  if col == white and color == white and mc + sum == 0 then Announce()
  else send(tok, white, 0) to RING_NEIGHBOUR
else if (color == white) then send(tok, col, sum + mc) to RING_NEIGHBOUR
     else send(tok, black, sum + mc) to RING_NEIGHBOUR
color := white
```