Vizualizácia distribuovaných algoritmov

Bc. JURAJ PORUBSKÝ

2009

Vizualizácia distribuovaných algoritmov

DIPLOMOVÁ PRÁCA

Bc. Juraj Porubský

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Študijný odbor: 9.2.1 INFORMATIKA

Vedúci diplomovej práce:
doc. RNDr. Rastislav Královič, PhD.

BRATISLAVA 2009

Abstrakt

PORUBSKÝ, Juraj: Vizualizácia distribuovaných algoritmov. [diplomová práca] – Fakulta matematiky, fyziky a informatiky Univerzity Komenského v Bratislave; Katedra informatiky. – Školiteľ: doc. RNDr. Rastislav Královič, PhD.– Bratislava 2009. 56 strán.

Práca sa primárne zaoberá špecifikáciou, popisom a implementáciou systému, ktorý podporuje vizualizáciu distribuovaných algoritmov v reálnom čase. Vizualizácia v reálnom čase kladie veľké nároky na výpočtovú silu procesora, preto bola snaha realizovať implementáciu čo najefektívnejšie. Sekundárnym zameraním práce je implementácia vybraných distribuovaných algoritmov v tomto systéme a popis ich funkčnosti.

Kľúčové slová: distribuované algoritmy, distribuované programovanie, vizualizácia, simulácia, voľba šéfa, výmena správ

Predhovor

V súčasnosti nenapreduje vývoj frekvencie procesorov takou rýchlosťou, ako tomu bolo napr. pred desiatimi rokmi. V tej dobe sa frekvencia procesora dokázala zdvojnásobiť v priebehu dvoch rokov, teraz sa ustáľuje viacmenej na tej istej hodnote. Z toho dôvodu sa sekvenčné riešenie problémov nedá časovo zlepšiť. Urýchlenie sa dá dosiahnuť, ak sa problém rozdelí na viacero podproblémov, ktoré sa dajú spracovať paralelne, resp. distribuovane na viacerých nezávislých procesoroch. Pri tomto type výpočtu je analýza algoritmu zložitejšia, preto som sa rozhodol spraviť systém, ktorý by užívateľovi dokázal algoritmus lepšie objasniť.

Mnou navrhnutý systém simuluje a vizuálne zobrazuje priebeh distribuovaných algoritmov v rôznych situáciách, preto môže užívateľ nadobudnúť lepšiu predstavu o tom, ako algoritmus funguje. Systém umožňuje vizualizovať ľubovoľný distribuovaný algoritmus. Vizualizácia je robená v reálnom čase, aby čo najpresnejšie odrážala realitu. Na zachytenie rôznych situácií priebehu algoritmu je užívateľovi dovolené parametrizovať nastavenia siete, v ktorej je distribuovaný algoritmus spustený.

Po dokončení implementácie systému je mojou ambíciou vytvoriť zbierku vybraných distribuovaných algoritmov, ktoré môžu byť vhodné na vzdelávacie účely.

Ďakujem svojmu vedúcemu diplomovej práce, doc. RNDr. Rastislavovi Královičovi, PhD. za cenné rady a pripomienky pri písaní tejto práce.

Ďakujem aj celej mojej rodine za podporu počas celého štúdia.

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry.

V Bratislave 4.mája 2009 _____

Obsah

Úvod

Vizualizácia sa ukazuje ako dobrý edukačný prostriedok, vďaka ktorému vie človek získať lepší prehľad o tom, ako veci fungujú. Téma vizualizácie distribuovaných algoritmov už bola doposiaľ spracovaná viacerými spôsobmi.

- **ConcurrentMentor** – aplikácia zobrazujúca informácie po skončení distribuovaných algoritmov programovaných v ThreadMentore, čo je knižnica podporujúca distribuované programovanie pomocou vlákien. Vizualizácia staticky zobrazuje rôzne štatistické údaje po skončení algoritmu ako napr. správy, ktoré sa posielali.

- **LYDIAN** – aplikácia, ktorá vizualizuje distribuované algoritmy. Je naprogramovaná v C a užívateľovi dovoľuje napísanie aj vlastného distribuovaného algoritmu, ktorého správanie si následne môže nechať odvizualizovať. Používa zobrazovanie objektov definovaných pomocou externej knižnice POLKA.

  Nevýhody:

  - rozštiepenie aplikácie – užívateľ si najprv namodeluje sieť v programe GraphWin a ručne si skompiluje svoj zdrojový kód distribuovaného algoritmu. Model siete a skompilovaný kód si následne otvorí v aplikácii LYDIAN, kde sa spustí vizualizácia.
  - vizualizácia nie je v reálnom čase – priebeh distribuovaného algoritmu napreduje pomocou stláčania tlačidla, kedy sa správy v jednom kroku presunú po linkách. Takouto vizualizáciou sa stráca dojem autonómie a samostatnosti procesov, a preto je pochopenie algoritmu pre užívateľa zložitejšie.

Doterajšie projekty uskutočňujú vizualizáciu viacerými spôsobmi. Niektoré iba staticky znázorňujú rôzne informácie po skončení distribuovaného algoritmu, iné sa snažia zachytiť aj samotný priebeh algoritmu. Žiadna vizualizácia však nebola robená v

reálnom čase. Preto som sa rozhodol implementovať systém, ktorý by znázorňoval reálne správanie distribuovaného algoritmu.

V prvej kapitole sa nachádza funkčná špecifikácia systému.

V druhej kapitole je rozpracovaný podrobný popis systému a možnosti jeho použitia.

Tretia kapitola rieši problémy, ktoré vznikli pri implementácii systému. Ponúka analýzu možných riešení, ich výhod a nedostatkov.

Štvrtá kapitola je venovaná implementácii vybraných distribuovaných algoritmov v navrhnutom systéme.

1. Špecifikácia systému

1.1 Multiplatformovosť a voľba programovacieho jazyka

Jednou zo základných požiadaviek bola multiplatformovosť systému, tzn. aby sa dal program spustiť na viacerých platformách, minimálne vo Windows a v Linux. Táto požiadavka obmedzuje programátora vo výbere programovacích prostriedkov. Najčastejšie používanou voľbou pre multiplatformové programovanie je JAVA. Keďže som sa nechcel rozhodnúť hneď pre prvú nájdenú možnosť, skúšal som nájsť aj alternatívne programovacie jazyky ponúkajúce určitý druh multiplatformovosti.

Jednou z nich je Lazarus, čo je IDE pre freepascal ponúkajúce multiplatformovosť na úrovni kompilácie programu. Lazarus sa snaží o abstrakciu programovacích príkazov závislých od platformy, preto sa dá program naprogramovaný v Lazarovi skompilovať pre rôzne operačné systémy. Keďže je Lazarus viac závislý od operačného systému ako JAVA, spravil som testovaciu aplikáciu. Kompilácia prebehla na oboch platformách úspešne, avšak vizualizácia sa správala odlišne. Problémom je práve už spomínaná abstrakcia nad príkazmi. Každý operačný systém obsahuje svoje vlastné ovládacie prvky a tie majú rôzne prednastavené vlastnosti, ktoré ovplyvňujú správanie aplikácie. Vzhľadom na rozsah diplomovej práce som si nemohol dovoliť testovať každú jednu časť aplikácie a upravovať ju pre jednotlivé operačné systémy osobitne. Preto voľba pre Lazarus nebola možná.

Ďalšou možnosťou by bolo pri programovaní využívať externé ovládacie prvky, akým je napríklad Qt, a programovať napr. v C++ alebo inom platformovo závislom programovacom jazyku. Pri tejto voľbe by nevznikli problémy pri vizualizácii, problémom by bolo udržať absolútnu abstrakciu nad operačným systémom.

Po analýze všetkých možných komplikácií som sa rozhodol pre jazyk JAVA, ktorá v sebe zahŕňa rôzne grafické sady ovládacích prvkov (SWT, AWT, SWING). SWING obsahuje práve tie prvky, ktoré sú väčšinou platformovo nezávislé. Ich grafické zobrazenie je naprogramované v JAVA, teda sa zobrazujú nezávisle od platformy, a preto programovanie nebude vyžadovať kontrolu, testovanie a upravovanie aplikácie v jednotlivých operačných systémoch.

1.2 Spôsob vizualizácie

Algoritmy riešiace určitú úlohu distribuovane sú spúšťané vrámci sietí, preto vizualizácia musí obsahovať vizualizáciu procesov, ich prepojení a správ, ktoré si medzi sebou posielajú. Najčastejšie používaným spôsobom reprezentácie siete je graf, kde vrcholy reprezentujú procesy a hrany prepojenia medzi nimi. Vďaka prehľadnosti som sa rozhodol použiť tento spôsob vizualizácie.

Keďže cieľom práce je vizualizáciou čo najlepšie vysvetliť užívateľovi priebeh distribuovaných algoritmov, je potrebné, aby sa dal ten istý algoritmus vizualizovať z rôznych uhlov pohľadu. Ďalšou požiadavkou bolo, aby vizualizácia prebiehala v reálnom čase.

1.3 Interaktivita systému

Spôsoby interaktivity systému:

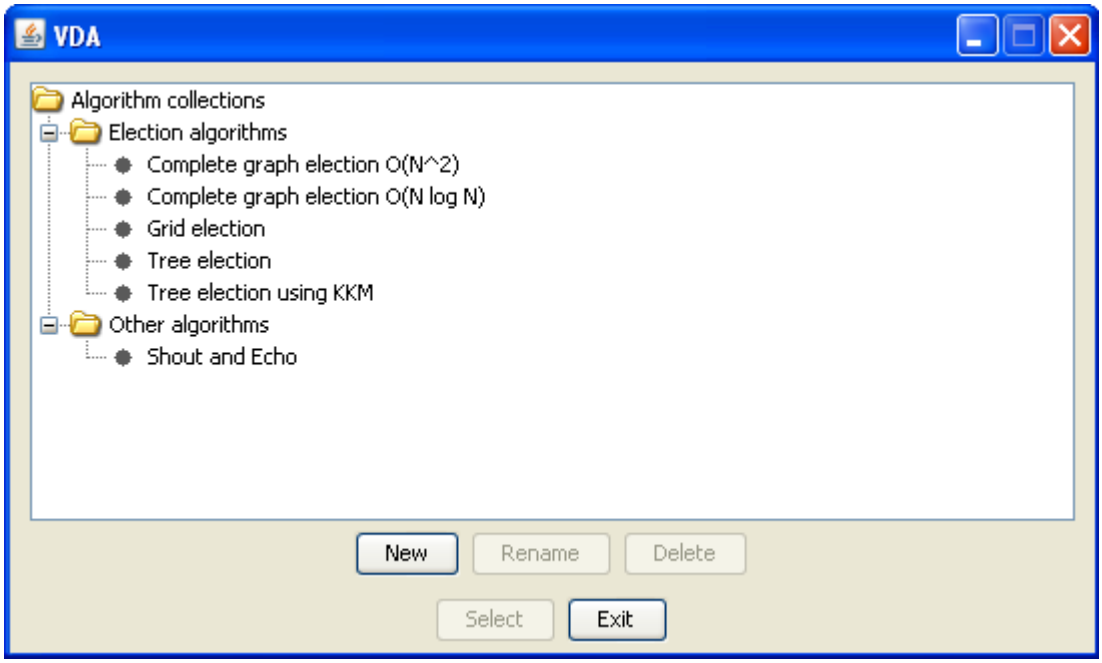
- **1. Interaktivita so zdrojovým kódom distribuovaného algoritmu.**
  Užívateľ má možnosť si pozrieť zdrojový kód distribuovaného algoritmu, prípadne ho pozmeniť a pozmenený aj spustiť. Zdrojový kód je pre jednoduchosť a kompatibilitu celého systému tiež písaný v jazyku JAVA a ponúka užívateľovi príkazy pre jednoduché distribuované programovanie.
- **2. Interaktivita so štruktúrou siete a s časovaním správ.**
  Každý distribuovaný algoritmus obsahuje zoznam predpripravených štruktúr sietí, ktoré užívateľ môže meniť, resp. pridať vlastné alebo mazať už existujúce. Tvorba a modifikácia štruktúry siete je umožnená pomocou myši. Na urýchlenie tvorby siete je užívateľovi ponúknutá voľba generovania vybraných typov sietí. Pre správne pochopenie algoritmu je často potrebné ukázať aj správanie v okrajových situáciách. Tie sa dosiahnu časovou následnosťou správ, preto je užívateľovi umožnené nastavovať obmedzenia na linky a tým ovplyvniť zdržanie správ.
- **3. Interaktivita so spôsobom vizualizácie.**
  Správanie distribuovaného algoritmu je zaujímavé sledovať z viacerých uhlov pohľadu. Z toho dôvodu sa k algoritmu viaže aj zoznam predefinovaných vizualizácií, z ktorých každá v sebe nesie informácie o tom, ktoré údaje sa majú vizualizovať. Zoznam vizualizácií si môže užívateľ upraviť sám.

Z dôvodu prehľadnosti bola vyvíjaná snaha všetky tri spôsoby interaktivity čo najviac od seba oddeliť, napr. ak si chce užívateľ nechať odvizualizovať iné údaje o priebehu distribuovaného algoritmu, nemusí pritom meniť zdrojový kód.

2. Popis systému

Systém bol vyvíjaný v jazyku JAVA a kompilovaný v JDK verzii 1.6.06, preto je na spustenie potrebné JRE minimálne tej istej verzie. Keďže sa distribuovaný algoritmus pred vizualizáciou kompiluje, je potrebné mať nainštalované JDK najlepšie v horeuvedenej verzii. Nižšia verzia JDK môže spôsobiť problémy pri kompilácii, vyššia zase problém pri načítaní skompilovanej triedy. Ak nastanú komplikácie a JDK by nebolo dostupné v spomínanej verzii, treba celú aplikáciu skompilovať odznova v dostupnej verzii.

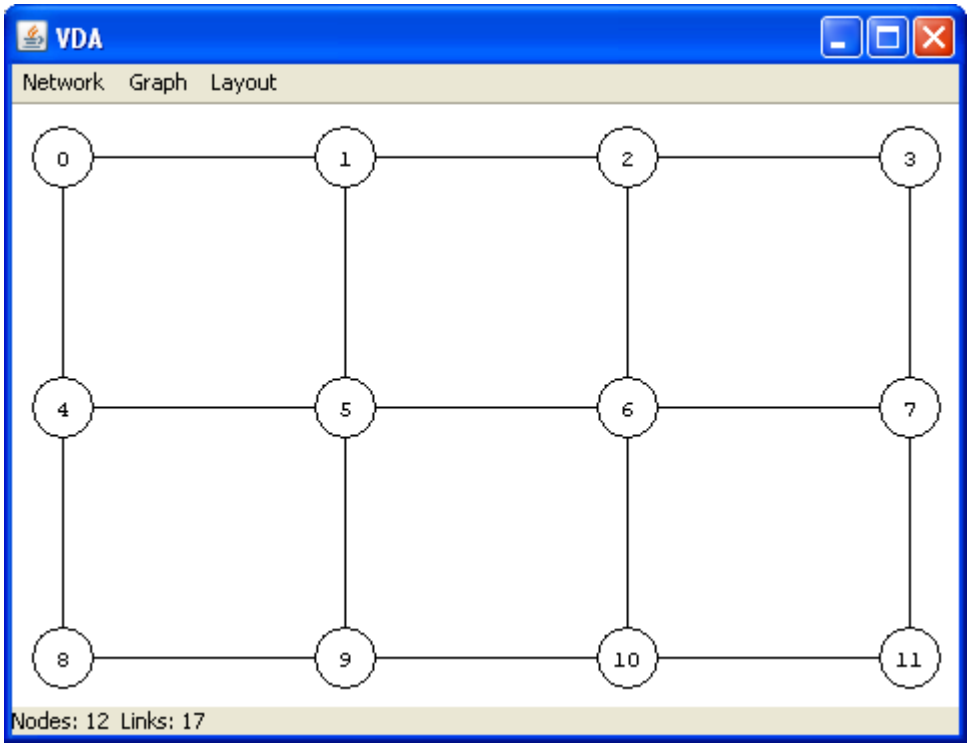
2.1 Vytvorenie distribuovaného algoritmu



Obr. 2.1 Vytvorenie distribuovaného algoritmu

Štruktúrovanie algoritmov je kvôli prehľadnosti dvojúrovňové, každý algoritmus patrí do nejakej zbierky algoritmov. V okne, ktoré je zobrazené na obrázku 2.1, vidno operácie umožňujúce vytvoriť, premenovať a vymazať algoritmus, resp. zbierku algoritmov. Po vybratí konkrétneho algoritmu z ponuky sa zjaví okno so zoznamom sietí, so zdrojovým kódom algoritmu a so zoznamom vizualizácií.

2.2 Editor siete



Obr. 2.2 Editor siete

Editor siete umožňuje vytváranie a modifikáciu siete a jej rozvrhnutie do 2D priestoru. Sieť sa dá vytvoriť ručne, teda pomocou myši, alebo sa dá vygenerovať konkrétny typ siete. Všeobecne platí pravidlo, že na vytvorenie nového elementu (procesu, linky) sa používa ľavé tlačidlo myši a na premiestnenie už existujúceho elementu sa používa pravé tlačidlo myši. Keďže je sieť zobrazená ako graf, označenie graf bude ďalej reprezentovať sieť.

Generovateľné typy grafov sú:

- **Náhodný graf** – procesy sú náhodne rozmiestnené v 2D priestore.
- **Kompletný graf –** procesy sú cyklicky usporiadané, pričom linky sa vytvoria pre každú dvojicu procesov.
- **Bipartitný graf –** procesy sú rozdelené do dvoch skupín, pričom platí, že žiadna linka nespája procesy jednej skupiny.

- **Kompletný bipartitný graf –** špeciálny typ bipartitného grafu, avšak linka existuje medzi každou dvojicou procesov, v ktorej jeden pochádza z prvej skupiny a druhý z druhej.
- **Cyklický graf –** rozloženie procesov je cyklické, pričom každý obsahuje práve 2 linky.
- **Strom –** vytvorí sa acyklický a súvislý graf.
- **Mriežka**

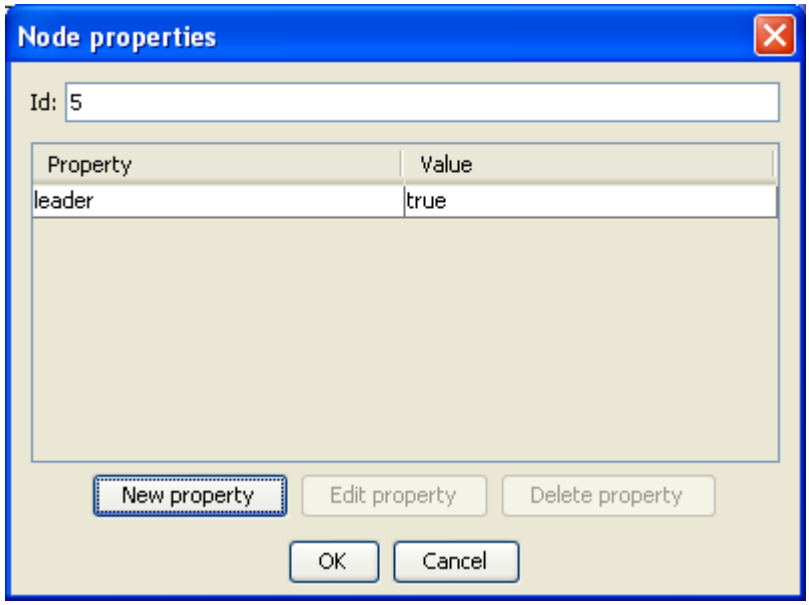
Okrem funkcií generujúcich siete sú k dispozícii aj funkcie, ktoré menia súradnice jednotlivých vrcholov, teda vrcholy rozložia do 2D priestoru.

Funkcie meniace súradnice vrcholov:

- **Náhodné rozloženie** - vrcholy rozloží v okne náhodne.
- **Cyklické rozloženie -** usporiada vrcholy do elipsy podľa veľkosti okna.
- **Bipartitné rozloženie** - pre jeho použitie musí graf spĺňať bipartitnosť, čo znamená, že vrcholy sa musia dať rozdeliť na 2 skupiny, v ktorých žiadna linka nespája vrcholy tej istej skupiny. Ak táto podmienka platí, vrcholy patriace do prvej skupiny sa zobrazia v okne naľavo, vrcholy druhej skupiny napravo.
- **Centrované rozloženie** - presunie celý graf do stredu. Dĺžky liniek sa zachovávajú rovnako ako aj navzájom relatívne vzdialenosti vrcholov.
- **Roztiahnuté rozloženie -** priblíži graf na maximum. Dĺžky všetkých liniek sa predĺžia alebo skrátia rovnakým pomerom.
- **Rozloženie podľa mriežky -** vytvorí imaginárnu mriežku, ktorej vzdialenosť dvoch susedných vrcholov je určená hodnotou Dist. Následne sa každý vrchol v grafe presunie na pozíciu najbližšieho vrchola v mriežke.

Každý element v sieti (proces, linka) má svoje nastavenia, ktoré sa dajú určiť jednotlivo alebo globálne.

2.2.1 Vlastnosti procesov



Obr. 2.2.1 Vlastnosti procesu

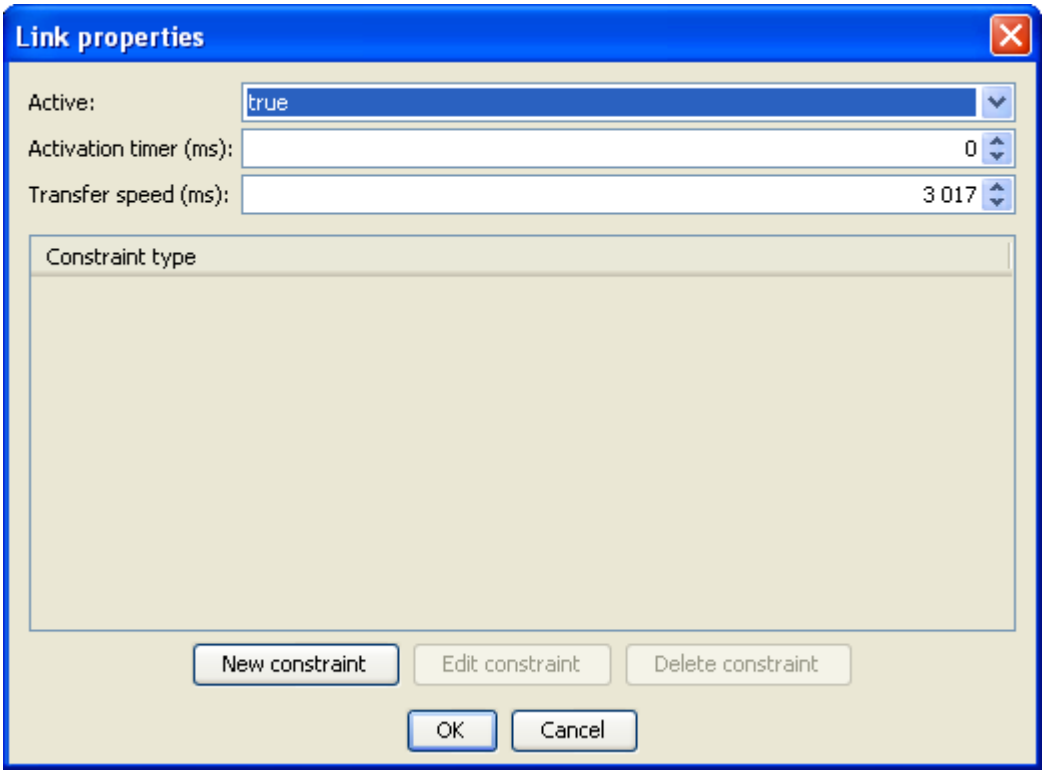
Každý proces má unikátny identifikátor – id. Okrem neho obsahuje niekoľko vlastností a k nim počiatočné hodnoty. Počas samotnej vizualizácie ich distribuovaný algoritmus môže čítať pomocou príkazu GetProperty, prípadne prepisovať príkazom SetProperty. Všetky hodnoty vlastností zadané cez tabuľku (obr. 2.2.1) sú uložené ako String. Distribuovaný algoritmus môže ukladať objekty ľubovoľných typov.

2.2.2 Vlastnosti liniek

Posielanie správ po linke prebieha asynchrónne, tzn. že proces po odoslaní správy nevie, kedy ju cieľový proces prijme. Linka môže byť v dvoch stavoch: v aktívnom alebo v neaktívnom. Ak proces posiela správu po neaktívnej linke, správa je stratená a nikdy nebude doručená cieľovému procesu. Odosielajúci proces však nevie identifikovať, či sa daná správa stratila alebo nie. Táto možnosť dovoľuje testovať správanie distribuovaných algoritmov v systéme so stratami správ. Ďalšou vlastnosťou liniek je prenosová rýchlosť, ktorá sa udáva v milisekundách a určuje dobu prechodu každej správy po linke. Keďže aj v reálnom svete je rýchlosť prenosu neznáma, tento parameter umožňuje parametrizovať vlastnosti siete.

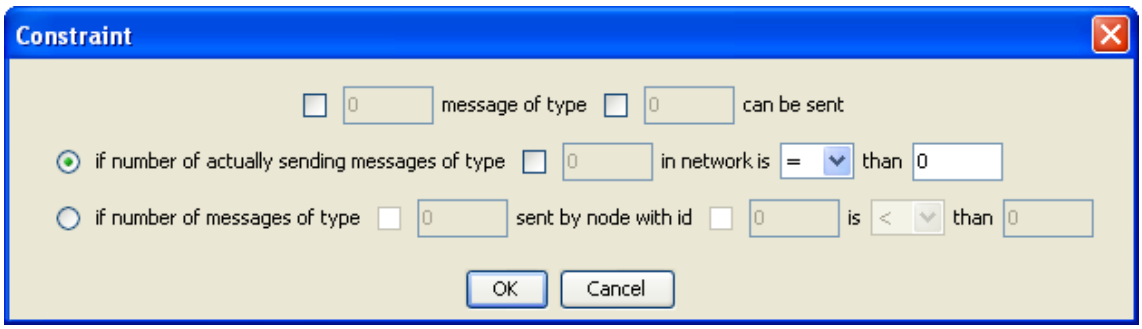
Pre lepšie zachytenie okrajových situácií môžu byť definované obmedzenia na linke. Ak ich obsahuje, tak pri každom posielaní správy sa kontrolujú všetky

obmedzenia viazané na linku, po ktorej správa prechádza. Ak sa nájde aspoň jedno obmedzenie, ktoré nie je splnené, tak je správa zadržaná až do momentu, keď budú všetky obmedzenia zachované. Až potom sa začne správa po linke posielať. Pri použití obmedzení, linky naďalej zostávajú FIFO. Preto aj v prípade, že proces poslal po tej istej linke dve správy, pričom iba na prvú z nich sa viaže obmedzenie, nemôže druhá správa predbehnúť prvú. Musí preto počkať na splnenie obmedzení viazaných na prvú správu.



Obr. 2.2.2 Vlastnosti liniek

2.2.3 Obmedzenia na linkách



Obr. 2.2.3 Obmedzenia liniek

Pre lepšie korigovanie posielania správ v sieti slúžia obmedzenia, ktoré v čase vizualizácie nadobúdajú hodnotu TRUE alebo FALSE. Správa posielaná po linke musí spĺňať všetky obmedzenia. Vo formulári je každé zaškrtávacie políčko pridružené k textovému poľu. Pokiaľ nie je políčko zaškrtnuté, sémanticky platí pre všetky.

Jednotlivé typy obmedzení:

- **Obmedzenie viazané na sieť:**
  - Umožňuje definovať obmedzenie, ktorého pravdivosť závisí od aktuálneho počtu správ určitého typu (alebo bez ohľadu na typ správy) v sieti.
  - Znenie: [prvá, druhá,..., každá] správa typu [číslo, nezáleží] môže byť poslaná, ak počet práve posielaných správ typu [číslo, nezáleží] v sieti je [<,<=,=,>=,>] ako [číslo]
  - Príklad: Obmedzenie, ktoré bude čakať, až kým počet posielaných správ v sieti bude rovný 1. Tzn. že okrem čakajúcej správy sa už žiadna iná správa v sieti neposiela.
- **Obmedzenie viazané na proces:**
  - Definuje obmedzenie, ktoré kontroluje počet poslaných správ nejakým procesom (alebo všetkými).
  - Znenie: [prvá, druhá,..., každá] správa typu [číslo, nezáleží] môže byť poslaná, ak počet správ typu [číslo, nezáleží] poslaných vrcholom [id_vrchola, všetkými] je [<,<=,=,>=,>] ako [číslo]
  - Príklad: Obmedzenie zadržujúce správu, až kým počet správ typu 1 poslaných procesom s id rovným 5 je väčšie ako 0. Tzn. že správa čaká až kým proces s id 5 pošle aspoň 1 správu typu 1.

Pomocou obmedzení sa dá jednoduchým spôsobom definovať zdržanie správy, ktoré sa viaže na nejaký jav v sieti. Nie je pritom nutné meniť zdrojový kód distribuovaného algoritmu. Pri používaní obmedzení si treba dať pozor na deadlock, teda na situáciu, v ktorej budú správy čakať na splnenie obmedzení, ktoré sa navzájom vylučujú a správy tak nikdy nebudú k cieľovým procesom doručené. Napr.: v sieti sú dve správy, z ktorých jedna čaká na to, kým bude druhá doručená a naopak druhá čaká

na doručenie prvej správy. Táto situácia nemá východisko, treba aspoň jedno obmedzenie odstrániť.

2.3 Distribuovaný algoritmus

Zdrojový kód distribuovaného algoritmu sa píše ako samostatná trieda v jazyku JAVA. Jediná podmienka je, že trieda musí dediť z triedy algorithm.DistributedAlgorithm. Tá poskytuje aj niekoľko metód potrebných pre distribuované programovanie. Pri vizualizácii sa pre každý proces v sieti vytvorí samostatné vlákno, ktoré zavolá metódu Run(). Preto pri definovaní správania vlastného distribuovaného algoritmu je potrebné, aby sa predefinovala metóda Run(). Po skončení vykonávania Run() metódy sa vlákno stane neaktívnym a od toho momentu už nie je schopné vykonávať žiadnu činnosť ako napr. prijímanie správ, posielanie správ, atď.

Príklad:

```
import algorithm.DistributedAlgorithm;
import network.Message;

public class MyAlgorithm extends DistributedAlgorithm
{

  public void Run()
  {
    if (LinkExists(0))
      SendMessageByLink(1,"hello",0);
    while (true)
    {
      Message mes=ReceiveMessage();
      SendMessageByLink(mes.GetType(),mes.GetObject(),mes.GetLinkNum());
    }
  }

}
```

Popis uvedeného algoritmu:

Na začiatku každý proces zistí, či obsahuje aspoň jednu linku. Ak áno, pošle po prvej linke správu „hello“ typu 1. Ak proces obdrží nejakú správu, odošle ju po tej istej linke naspäť.

Trieda network.Message:

Každá správa posielaná po linke nesie okrem posielaného objektu aj informáciu o type správy. Na zaobalenie týchto, ale aj iných, informácií sa používa objekt triedy network.Message, ktorý ponúka niekoľko metód:

- int **GetType**() - vráti typ správy
- Object **GetObject**() - vráti posielaný objekt
- int **GetLinkNum**() - vráti číslo linky, po ktorej proces správu prijal

Metódy triedy algorithm.DistributedAlgorithm:

- int **GetId**()

  Vráti identifikátor procesu, ktorý je v celej sieti unikátny.
- void **SendMessageByLink**(int type, Object mes, int linkNum)

  Pošle správu mes typu type po linke číslo linkNum. Pokiaľ také číslo linky neexistuje, správa nie je poslaná. Posielanie správy je neblokujúca operácia, preto proces volajúci metódu SendMessageByLink alebo SendMessageToNode pokračuje vo vykonávaní nasledujúcich inštrukcií bez toho, aby čakal, kým cieľový proces správu prijme.
- void **SendMessageToNode**(int type, Object mes, int nodeId)

  Pošle správu mes typu type procesu s id rovným nodeId. Pokiaľ taký proces neexistuje, výnimka nevznikne, avšak žiadna správa nie je z odosielajúceho procesu poslaná. Odosielajúci proces nemusí byť priamo prepojený s prijímajúcim procesom. V takom prípade sa vyberie najkratšia trasa medzi procesmi a správa je po nej poslaná. Procesy, cez ktoré správa prechádza, ju však nikdy neprijmú cez volanie metódy ReceiveMessage, prijme ju len koncový proces. Algoritmus hľadajúci najkratšiu trasu medzi procesmi využíva prehľadávanie do šírky. Za najkratšiu trasu sa považuje tá, ktorá obsahuje najmenší počet liniek, a nie tá, po ktorej sa správa najrýchlejšie dopraví cieľovému procesu.
- int **GetLinksCount**()

  Vráti počet liniek vychádzajúcich z procesu. Linky sú očíslované od 0 po GetLinksCount()-1 vrátane.

- boolean **LinkExists**(int linkNum)

  Vráti TRUE ak existuje linka s číslom linkNum. Inými slovami, ak platí, že linkNum>=0 a zároveň linkNum<GetLinksCount().
- Message **ReceiveMessage**()

  Správy doručené procesu sa ukladajú do fronty (FIFO). Proces má k nim prístup cez metódu ReceiveMessage. Ak bola procesu doručená nejaká správa, vráti správu ako objekt typu network.Message. Ak je však fronta s prijatými správami prázdna, vlákno sa pozastaví, až kým nejakú správu neprijme. Ide teda o blokované prijatie správy.
- Message **ReceiveMessage**(int linkNum)

  Podobne ako ReceiveMessage(), ale s tou podmienkou, že sa správa prijíma iba na konkrétnej linke. Správy prichádzajúce z ostatných liniek sa ukladajú do fronty, takže sa žiadna správa nestratí a môže byť procesom obdržaná pri ďalšom volaní metódy ReceiveMessage.
- Message **ReceiveMessage**(Set<Integer> linkSet)

  Parameter linkSet je inštancia triedy, ktorá implementuje interface java.util.Set, čiže množinu. Najčastejšie používanými triedami, ktoré implementujú interface Set, sú HashSet, LinkedHashSet a TreeSet. Parameter linkSet v sebe obsahuje čísla liniek, na ktorých sa má čakať na prijatie správy. Prvá správa prijatá procesom po linke, ktorá sa nachádza v množine linkSet, je vrátená volaním tejto metódy. Podobne ako u ReceiveMessage(int linkNum) sa všetky správy prichádzajúce z ostatných liniek ukladajú do fronty a preto môžu byť procesom prijaté po zavolaní metódy ReceiveMessage s adekvátnym filtrom.
- void **StopProcess**()

  Volanie metódy zastaví vykonávanie aktuálneho procesu vzniknutím výnimky StopException, ktorá signalizuje, že proces žiada o skončenie. Pre správnu funkčnosť by sa metóda StopProcess nemala volať z try-catch bloku, lebo sa výnimka zachytí a signál o ukončení procesu sa stratí.
- Object **GetProperty**(Object key)

  Vráti hodnotu vlastnosti procesu. Inicializačné hodnoty vlastností sa určujú v editore siete, avšak počas behu distribuovaného algoritmu sa môžu prepisovať,

alebo sa môžu definovať úplne nové vlastnosti, ktoré neboli definované v editore siete.

- void **SetProperty**(Object key, Object value)
  Nastaví hodnotu vlastnosti. Key označuje názov vlastnosti a value jej hodnotu.
- int **GetNodesCount**()
  Vráti počet procesov v sieti.
- boolean **NodeExists**(int nodeId)
  Výsledkom je true práve vtedy, keď sa v sieti nachádza proces s identifikátorom rovným nodeId. Inak vráti false.
- int[] **GetNodesIds**()
  Vráti pole, ktorého hodnoty obsahujú identifikátory všetkých procesov v sieti. Pole nemusí byť utriedené.
- void **Assert**(boolean cond, String text)
  Metóda Assert sa všeobecne používa pri ladení algoritmov. Nakoľko je debugovanie viacerých vlákien dosť náročné na implementáciu, aplikácia nepodporuje debugovanie distribuovaného algoritmu krok za krokom. Preto práve používanie Assert metódy môže užívateľovi pomôcť nájsť v algoritme chybu. Volaním metódy sa skontroluje parameter cond a pokiaľ platí, že je cond rovné false, tak sa vpíše hodnota parametra text do Assert logu. Assert log zdieľajú všetky simulované procesy a je možné si ho prečítať pri pozastavení alebo stopnutí vizualizácie.
- void **SetMessageReceivedListener**(int type, MessageReceivedListener listener)
  Pomocou tejto metódy sa dajú definovať vlastné obsluhovače udalostí po prijatí konkrétneho typu správy. Každý typ správy môže mať definovaný iba jeden obsluhovač, a preto ak sa zavolá táto metóda viackrát na rovnaký typ, pamätá si iba posledný z nich. Podrobnejšie informácie o používaní obsluhovačov sú uvedené nižšie.

Niektoré vyššie spomenuté metódy ponúkajú informácie, ktoré nie sú bežne dostupné v reálnom distribuovanom programovaní (napr: GetNodesCount a GetNodesIds) a ktorých znalosť môže výpočet značne zjednodušiť. Preto sa tieto

metódy pri štandardnom riešení úloh zväčša nepoužívajú, môžu však poslúžiť na kontrolu správnosti výpočtu.

2.3.1 Používanie obsluhovačov udalostí MessageReceivedListener

Dôvodom ich zavedenia je snaha implementačne oddeliť kombinovanie viacerých distribuovaných algoritmov. Pokiaľ je teda súčasťou distribuovaného algoritmu aj iný algoritmus, ktorý používa nezávislý mechanizmus posielania správ pre špeciálne typy správ, môže sa tento mechanizmus definovať spomínanými obsluhovačmi udalostí algorithm.MessageReceivedListener.

Interface algorithm.MessageReceivedListener:

- public void **OnMessageReceive**(Message mes) – vykoná sa po prijatí takého typu správy, ktorý bol asociovaný cez volanie metódy SetMessageReceivedListener. Parametrom mes je prijatá správa, ktorá spĺňa vyššie spomenuté kritériá.

Pokiaľ sa nastaví obsluhovač udalostí na konkrétny typ správy, správa tohto typu nemôže byť prijatá cez volanie metódy ReceiveMessage. Jej prijatie sa zaznamenáva iba cez obsluhovač udalostí. Vykonávanie OnMessageReceive neprebieha paralelne s hlavným telom metódy Run() z dôvodu nutnosti synchronizácie prístupu ku každému objektu. Pod paralelným priebehom sa myslí existencia samostatného vlákna na obsluhovanie udalostí. Prijímanie správ cez obsluhovač udalostí sa vykonáva vždy, ako sa zavolá metóda ReceiveMessage. Z toho dôvodu nemožno prijímať správy iba cez obsluhovače udalostí a bez použitia volania metódy ReceiveMessage.

Príklad:

```
import algorithm.DistributedAlgorithm;
import algorithm.MessageReceivedListener;
import network.Message;

public class Algorithm extends DistributedAlgorithm
{
  @Override
  public void Run()
  {
    //define listener
    MessageReceivedListener listener=new MessageReceivedListener()
    {
      public void OnMessageReceive(Message mes)
      {
        SendMessageByLink(mes.GetType(), mes.GetObject(), mes.GetLinkNum());
      }
    };
    SetMessageReceivedListener(1, listener);

    //algorithm
    if (LinkExists(0))
      SendMessageByLink(1, "message", 0);
    Message mes=ReceiveMessage();
  }
}
```

Popis uvedeného algoritmu:

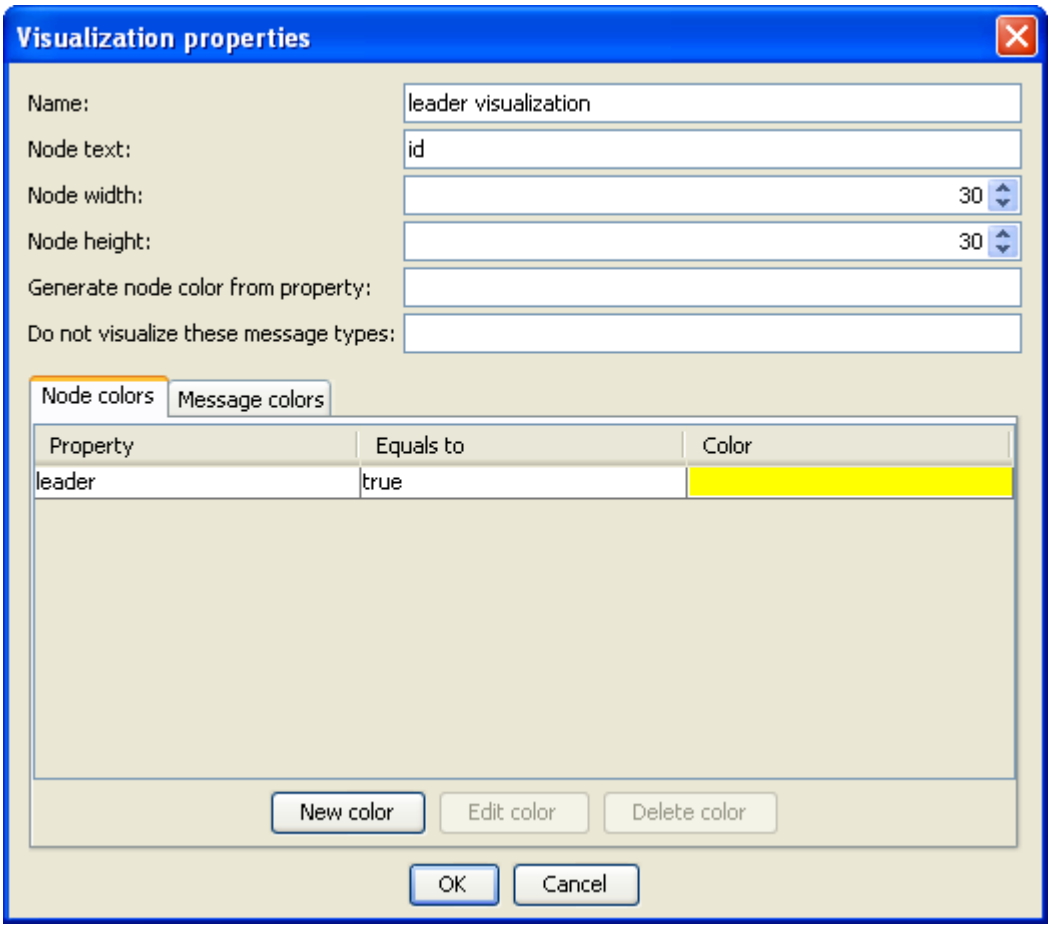
Na začiatku sa definuje obsluhovač udalostí, ktorý sa priradí k správam typu 1. Ten zabezpečuje, že ak proces prijme správu typu 1, tak ju pošle po tej istej linke naspäť. Po definovaní obsluhovača udalostí sa z každého vrchola po prvej linke pošle správa typu 1 a následne prejde proces do stavu čakania na správu zavolaním metódy ReceiveMessage, ktorá však nikdy žiadnu správu nevráti. Dôvodom je, že sa v sieti neposielajú správy iných typov ako 1 a správa typu 1 sa prijíma iba cez definovaný obsluhovač udalostí. Príkaz ReceiveMessage sa však nemôže odstrániť. Ten zabezpečuje, že sa vykonávanie metódy Run() nikdy neskončí, čo znamená, že proces zostane aktívnym a je schopný reagovať na prijatie správy cez definovaný obsluhovač udalostí.

2.4 Nastavenie vizualizácie

Priebeh distribuovaného algoritmu môže byť, hlavne pri zložitejších algoritmoch, zaujímavý z rôznych uhlov pohľadu. Preto si každý distribuovaný algoritmus pamätá aj zoznam vizualizácii, ktoré v sebe obsahujú popis ako sa má sieť zobrazovať. Na spustenie vizualizácie alebo otvorenie editora siete je potrebné, aby bola práve jedna vizualizácia aktivovaná.

Nastaviteľné atribúty vizualizácie:

- Rozmery vrcholov v grafe. Šírka vrchola nemusí byť totožná s výškou vrchola, preto sa vrchol vykresľuje ako elipsa.
- Definuje sa názov vlastnosti procesu, ktorej hodnota sa vpíše do vrcholov. K hodnotám vlastností môže distribuovaný algoritmus pristupovať cez metódy SetProperty a GetProperty.
- Určia sa typy správ, ktoré sa nebudú vizualizovať. Platia pre ne rovnaké pravidlá ako pre tie, ktoré sa graficky zobrazujú. Pokiaľ sa nemá zobrazovať viacero typov správ, treba ich oddeliť čiarkou.
- Farby vrcholov sa dajú určiť dvoma spôsobmi. Prvý z nich definuje názov vlastnosti, jej hodnotu a farbu. Ak sa hodnota vlastnosti procesu rovná zvolenej hodnote, vrchol sa vyfarbí zvolenou farbou. Pokiaľ distribuovaný algoritmus prepíše hodnotu vlastnosti objektom, ktorý nie je String, spomínaný mechanizmus vyhodnotí, že sa hodnota vlastnosti nerovná zvolenej hodnote. Pri druhom spôsobe farbenia vrcholov sa farba generuje automaticky podľa hodnoty vlastnosti. Keď sa hodnoty zvolenej vlastnosti oboch procesov rovnajú, vrcholom sú pridelené rovnaké farby. Druhý spôsob farbenia vrcholov má prednosť pred prvým, a preto sa farbenie vrcholov prvým spôsobom vykonáva iba v prípade, že sa vlastnosť v druhom spôsobe neurčí.
- Farby posielaných správ. Pre každý typ správ sa dá definovať farba, ktorou sa vyfarbia.



Obr. 2.4 Nastavenia vizualizácie

2.5 Priebeh vizualizácie

Sieť sa vykresľuje na základe nastavenia aktivovanej vizualizácie. Samotný priebeh vizualizácie je v reálnom čase, tzn. že sa posielanie správ deje plynulo a vizualizácia sa dá v akomkoľvek okamžiku pozastaviť. V tomto stave je tak možné získať rôzne informácie o priebehu algoritmu alebo o stave simulovaných procesov, prípadne je možné ich pozmeniť.

Pomocou pravého tlačidla na myši sa po kliknutí na proces, linku alebo správu dajú získať detailnejšie informácie. Pri procese sa zobrazí počet odoslaných správ (pre každý typ správy je počet uvedený osobitne) a hodnoty vlastností procesu. Linka ponúka zoznam správ čakajúcich na splnenie obmedzení. Správa obsahuje svoj typ, text správy a identifikátory odosielateľa a príjemcu. Nastavenia procesov a liniek sa dajú v čase pozastavenia vizualizácie upraviť rovnako ako v editore siete. Prepísanie hodnoty vlastnosti procesu ovplyvní jej aktuálnu hodnotu.

Okrem informácií viazaných na konkrétny element (proces, linku alebo správu) je možné získať aj celkové informácie o priebehu distribuovaného algoritmu, medzi ktoré patria štatistické údaje a chybové správy. Štatistické údaje informujú o tom, koľko správ bolo v sieti poslaných a koľko správ sa aktuálne posiela. Počty sú uvedené pre každý typ správy osobitne. Keďže predmetom analýzy distribuovaného algoritmu je aj počet vykomunikovaných správ, štatistické údaje sú v tomto nápomocné. Ak je distribuovaný algoritmus chybne naprogramovaný a za jeho behu dochádza k vyvolaniu výnimiek, ich popisy si užívateľ môže prečítať po kliknutí na tlačidlo Exceptions. Popis obsahuje identifikátor procesu, ktorý výnimku vyvolal, slovné pomenovanie výnimky a číslo riadku, na ktorom výnimka nastala. Keď proces vyvolá výnimku, ktorá nie je programátorsky zachytená v bloku try-catch, tak sa skončí vykonávanie metódy Run a proces sa stane neaktívnym.

3. Implementácia

3.1 Simulácia procesu

Posielanie správ prebieha v sieti asynchrónne, a preto proces posielajúci správu nevie určiť, kedy bude správa cieľovému procesu doručená. Každý proces je samostatná výpočtová jednotka, ktorá sa správa autonómne a neberie ohľad na žiaden synchronizačný mechanizmus s inými procesmi.

Na prvý pohľad črtá implementačné riešenie, v ktorom bude každý simulovaný proces definovaný ako samostatný proces vrámci operačného systému. Riešenie by ponúklo okrem schopnosti samostatného výpočtu aj ochranu prístupu k dátam na úrovni operačného systému. Pri tejto voľbe sa vyskytol problém, lebo JAVA nepozná pojem proces. Vrámci jednej JVM (Java Virtual Machine) môže existovať iba jeden proces, ktorý je automaticky vytvorený pri štarte aplikácie. Riešenie by muselo pre každý simulovaný proces spustiť samostatnú inštanciu JVM, čo je z pamäťového hľadiska neprípustné. Spravil som skúšobný test, v ktorom sa ukázalo, že každé spustenie jednoduchého programu s bežnými nastaveniami JVM potrebuje približne 12 MB pamäte. Pri simulácii tridsiatich procesov by sa tak množstvo potrebnej pamäte zvýšilo na 360 MB.

Druhé riešenie simuluje procesy vláknami. Oproti predchádzajúcemu riešeniu sa síce stráca ochrana prístupu k dátam, ale výhodou sú oveľa nižšie pamäťové nároky. Nakoľko je aplikácia určená hlavne na edukačné účely, ochrana prístupu k dátam nie je taká dôležitá. Pre porovnanie, pamäťové nároky tohto riešenia pri simulácii 30 procesov sú spolu okolo 2 MB, čo je oproti prvému riešeniu značný rozdiel. A preto som sa rozhodol implementovať druhé riešenie simulácie procesu.

Ako tretím riešením je možnosť realizácie výpočtu všetkých simulovaných procesov iba jedným vláknom. Pamäťová náročnosť oproti druhému riešeniu by bola pravdepodobne ešte nižšia, ale nastali by problémy, pri ručnom prepínaní simulovaných procesov. Operačný systém prideľuje procesorový čas procesom a vláknam. A keďže podľa tretieho riešenia existuje iba jedno vlákno, ktoré sa môže vykonávať iba na jednom jadre, tak ostatné jadrá zostanú nevyužité. V prípade, že sa aplikácia spustí na viacjadrovom procesore, je druhé riešenie efektívnejšie. Kvôli vyššie spomenutým nevýhodám som zavrhol aj tretiu možnosť riešenia.

3.2 Posielanie správ

Okrem simulácie procesov, je potrebné mať aj mechanizmus, ktorý by zabezpečoval posielanie správ. A to spôsobom, že by z aktuálne posielaných správ v sieti kontroloval, ktoré z nich môžu byť k cieľovým procesom doručené a tie im následne aj doručil. Do úvahy musí brať rýchlosť prenosu linky, obmedzenia viazané na linku a musí zachovávať vlastnosť FIFO pre jednotlivé linky. Vyžaduje sa efektívna implementácia, aby sa predišlo činnému čakaniu, ktoré zbytočne zaťažuje procesor. Mechanizmus posielania správ je implementovaný ako samostatné vlákno.

Priebeh posielania a prijímania správ:

1) Vlákno simulovaného procesu volaním metód SendMessageByLink alebo SendMessageToNode vytvorí správu, ktorú uloží do zoznamu buď aktívnych alebo čakajúcich správ. Medzi aktívne sú zaradené tie správy, ktorých všetky obmedzenia sú splnené a žiadna iná správa na linke v tom istom smere nečaká. Všetky ostatné sú zaradené do zoznamu čakajúcich správ.
2) Po uložení správy do správneho zoznamu je potrebné zobudiť vlákno obsluhujúce posielanie správ. To najprv prejde zoznam čakajúcich správ a správy, ktoré spĺňajú všetky obmedzenia presunie do zoznamu aktívnych správ. V druhom kroku prejde zoznam aktívnych správ a tie správy, ktoré v ňom už zotrvali dobu prechodu správy po linke, vyberie a uloží ich do zoznamu prijatých správ procesu. Následne informuje vlákno simulujúce proces o doručení správy a stane sa nečinným. Zo stavu nečinnosti ho zobudí buď vlákno simulujúce proces poslaním novej správy, alebo sa zobudí samé po uplynutí časového kvanta, po ktorom je potrebné doručiť ďalšiu správu.
3) Prijímanie správ sa uskutočňuje volaním metódy ReceiveMessage. Po jej zavolaní vlákno simulovaného procesu zistí, či mu bola doručená nová správa. Ak áno, správu obdrží okamžite. Ak nie, tak sa vlákno uvedie do nečinnosti, až kým ho nezobudí vlákno obsluhujúce posielanie správ. Keď sa tak stane, znamená to, že sa prijala nová správa.

Efektívnosť posielania správ:

Z implementácie vlákna obsluhujúceho posielanie správ vidno, že je vlákno v činnosti, iba v prípade, že je potrebné niečo zariadiť. Nevzniká žiaden polling, ktorého podstata spočíva v neefektívnom využívaní procesorového času. Na zvýšenie efektívnosti v prehľadávaní správ sa zoznamy aktívnych a čakajúcich správ udržiavajú vo vhodnej dátovej štruktúre.

Pri aktívnych správach je zaujímavý čas, kedy môžu byť správy doručené, preto je vhodné ich udržiavať v zotriedenom poradí podľa času ich doručenia. Keďže sa jedná o dynamickú štruktúru, čo znamená, že správy môžu priebežne pribúdať a ubúdať, rozhodol som sa použiť červeno-čierne stromy. Tie zaručujú, že uloženie, vyhľadanie a vymazanie správy zo zoznamu sa vykoná v čase O(log N), kde N predstavuje počet správ v zozname. Správy sa navyše nachádzajú v zotriedenom poradí, preto pri zisťovaní, ktoré správy môžu byť procesom doručené, stačí prechádzať správy postupne od prvej a zastaviť sa pri správe, ktorej čas doručenia ešte nenastal.

V zozname čakajúcich správ nie je dôležité, aby boli správy utriedené, lebo sa nevie, ako dlho bude správa čakať na splnenie obmedzení. Podstatou je efektívne ukladať, vyhľadávať a mazať správy na základe linky, ku ktorej patria. Pre zachovanie FIFO vlastnosti pri linkách sa správy ukladajú do spájaného zoznamu zahašovaného podľa identifikátorov odosielajúceho a prijímajúceho procesu. To znamená, že každá linka má spolu dva spájané zoznamy čakajúcich správ, pre každý smer jeden. Vďaka tomu sa dá zabezpečiť FIFO odosielanie a prijímanie správ. Vyhľadanie správneho zoznamu správ pre konkrétnu linku spočíva v tom, že sa najprv podľa identifikátora odosielajúceho procesu v prvej hašovacej tabuľke vyhľadá druhá hašovacia tabuľka. Tá obsahuje k identifikátorom prijímajúcich procesov zahašované zoznamy čakajúcich správ. To znamená, že nájdenie správneho zoznamu čakajúcich správ sa realizuje pomocou vyhľadávania v dvoch hašovacích tabuľkách. Keďže časová efektívnosť vyhľadania záznamu v jednej hašovacej tabuľke je v priemernom prípade O(1), výsledná časová zložitosť pre dve prehľadania je v priemere takisto O(1).

3.3 Zamykanie objektov a problém uviaznutia

V aplikáciách využívajúcich viacero vlákien je potrebné riešiť zamykanie objektov a tým vylúčiť simultánne vykonávanie kritického úseku inštrukcií. Zamykanie objektov sa vykonáva pomocou príkazu synchronized, ktorý nie je reentrantný. Preto vlákno, ktoré vlastní zámok na konkrétnom objekte, nesmie zámok znova požadovať, lebo dôjde k zablokovaniu vlákna. Objekty, ktoré vyžadujú zamykanie, sú v systéme zväčša zoznamy správ (zoznam aktívnych správ, zoznam čakajúcich správ a zoznam doručených správ). Takmer všetky operácie na zoznamoch sa týkajú jednoduchého vyhľadávania a vloženia nového objektu, preto čas vykonávania kritických úsekov je relatívne krátky. Z toho dôvodu nerozlišujem úrovne zamykania objektov ako napríklad zámok na čítanie alebo zámok na zápis. Implementačne by sa tak zamykanie objektov veľmi skomplikovalo a výsledné zlepšenie by bolo minimálne.

Ak vlákna zamykajú viacero objektov, môže dôjsť k ich uviaznutiu (deadlock). Táto situácia vzniká, ak sa viacero vlákien navzájom čaká, kým sa uvoľnia prostriedky alebo ak vlákna čakajú na uvoľnenie prostriedkov v cykle.

Nutné podmienky uviaznutia:

- **Vzájomné vylúčenie** – prostriedok môže byť v tom istom čase používaný iba jedným vláknom.
- **Postupné získavanie zdrojov s čakaním** – získavanie prostriedkov prebieha postupne. Ak vlákno žiada o prostriedok, ktorý je používaný iným vláknom, tak čaká, kým sa prostriedok neuvoľní.
- **Nemožnosť prerozdelenia prostriedkov** – ak vlákno získa prostriedok, žiadne iné vlákno nemá právo mu ho odobrať. O uvoľňovanie prostriedkov sa stará iba vlákno, ktoré prostriedky získalo.
- **Cyklické čakanie** – existuje množina vlákien, v ktorej prvé vlákno čaká na prostriedok držaný druhým vláknom, ..., n-té vlákno čaká na prostriedok držaný prvým vláknom.

Existuje viacero spôsobov ako riešiť problém uviaznutia. Prvým spôsobom je detekcia uviaznutia a jeho odstránenie. Toto riešenie periodicky vykonáva algoritmus,

ktorý kontroluje, či nastala podmienka cyklického čakania. Frekventovaná kontrola však míňa veľa času procesora, preto som toto riešenie zamietol.

Druhým spôsobom je prevencia pred uviaznutím. Tento prístup sa snaží zabrániť platnosti niektorej z nutných podmienok uviaznutia. Ako sa ukázalo, implementačne sa najľahšie zamedzí platnosti cyklického čakania. Prostriedky sa očíslujú navzájom rôznymi číslami a ak vlákno potrebuje uzamknúť viacero prostriedkov, robí to vo vzostupnom poradí. Keďže objektov vyžadujúcich zamykanie je v mojej aplikácii relatívne málo, dá sa implementácia tohto riešenia dobre ustriehnuť. Táto implementácia nezvyšuje časové nároky aplikácie a tiež vďaka jej jednoduchosti som zvolil toto riešenie.

Tretím riešením je dynamické vyhýbanie sa uviaznutiu. Každá podmienka z nutných podmienok uviaznutia môže byť splnená, avšak sa zamedzí ich súčasná platnosť. Táto metóda menej obmedzuje vlákna pri získavaní prostriedkov ako preventívne metódy, je však algoritmicky zložitejšia a vyžaduje viac procesorového času. Zo spomínaných dôvodov som sa rozhodol neimplementovať tretí spôsob riešenia problému uviaznutia.

3.4 Kompilácia distribuovaného algoritmu

Jednou z prvotných požiadaviek na aplikáciu bola možnosť meniť zdrojový kód distribuovaného algoritmu. Vynárajú sa minimálne dve riešenia ako to dosiahnuť.

Prvým riešením je, že by sa distribuovaný algoritmus písal v nejakom pseudojazyku a jeho vykonávanie by zabezpečoval interpreter. Vykonávanie algoritmu interpreterom by bolo určite pomalšie ako vykonávanie natívneho kódu. Ďalšou nevýhodou by bola ohraničenosť jazyka, čo znamená, že by pseudojazyk podporoval iba určitú (veľmi malú) množinu príkazov. Tá by ani zďaleka nepokryla možnosti, ktoré ponúkajú rozšírené programovacie jazyky ako napr. JAVA.

JAVA v sebe obsahuje classloader, ktorý podporuje dynamické nahrávanie tried do pamäte. Druhým riešením je implementácia distribuovaného algoritmu v JAVA. Pred vizualizáciou je možné ho pomocou JAVA kompilátora skompilovať a následne nahrať do pamäte. Tento spôsob neobsahuje spomenuté nevýhody prvého riešenia, preto som sa rozhodol ho použiť v mojej aplikácii. Pri takto zvolenom postupe nemusí pri kompilácii zdrojového súboru .java vzniknúť iba jeden skompilovaný súbor .class, napr.

keď sa v zdrojovom kóde používajú vnútorné alebo anonymné triedy. Pri vizualizácii je potrebné načítať všetky skompilované súbory, a preto je pre ne vyhradený adresár tmp (temporary files=dočasné súbory). Jeho predchádzajúci obsah sa pred vizualizáciou vymaže a všetky skompilované súbory sa do neho uložia. Následne sa načítajú do pamäte, aby boli dostupné v čase vizualizácie a v prípade potreby sa použijú.

3.5 Zastavenie simulovaného procesu

Príkaz, ktorý zastaví vykonávanie procesu, je nesporne nevyhnutnou súčasťou distribuovaného programovania. V aplikácii sa nachádza pod názvom StopProcess(). Simulácia procesov sa deje na úrovni vlákien, preto bolo potrebné nájsť spôsob ako zastaviť vykonávanie vlákna. Metóda stop() triedy Thread bola globálne zavrhnutá z dôvodu narušenia konzistencie dát. Ak vlákno vlastnilo zámky na objektoch, po volaní stop() sa zámky uvoľnia a objekty sa tak môžu zachovať v nekonzistentnom stave.

Vlákno distribuovaného procesu sa automaticky ukončí po tom, ako sa ukončí vykonávanie metódy Run() triedy DistributedAlgorithm. To sa docieli tým, že po zavolaní metódy StopProcess() nastane výnimka (špeciálne StopException). Výnimka tohto typu by nemala byť v celom distribuovanom algoritme odchytená, aby sa zabezpečilo, že sa výnimka v konečnom dôsledku vynorí aj z metódy Run(). Trieda StopException je nasledovníkom triedy RuntimeException, a preto má vlastnosť, že sa na miestach jej použitia nemusí definovať try-catch blok.

3.6 Generovanie náhodných farieb

Z hľadiska vizualizácie distribuovaného algoritmu je niekedy žiadúce farebné rozlíšenie procesov na základe rozdielnosti hodnôt niektorej vlastnosti. Užívateľ tak nadobudne prehľad o tom, ktoré procesy majú hodnotu rovnakú a ktoré rôznu. Bolo preto potrebné navrhnúť mechanizmus, ktorý by objektom (hodnotám vlastnosti) priraďoval farbu. Navyše musí platiť, že rovnakým hodnotám sa vždy pridelí rovnaká farba a pre rôzne hodnoty by mal byť farebný rozptyl čo najväčší, a to z dôvodu, aby sa rozdielnosť farieb ľahko určovala. Snaha maximalizovať farebný rozptyl sa týka aj príbuzných objektov (napr. po sebe idúce čísla). Keďže počet zobraziteľných farieb je štandardne 2^{24} a počet možných objektov je teoreticky neohraničený, nie je možné

prideliť tú istú farbu iba jednému objektu. Z toho dôvodu musia existovať dva rôzne objekty, ktorým sa pridelí rovnaká farba.

Prvým riešením je možnosť užívateľa si navoliť paletu farieb, z ktorej by sa objektom prideľovali farby. Takto zvolené riešenie má dve nevýhody. Prvou nevýhodou je pracný postup pri definovaní palety farieb a môže to užívateľa od tohto úkonu odradiť. Druhou nevýhodou je nedostačujúca veľkosť palety farieb, ktorá spôsobí väčšiu pravdepodobnosť, že sa rôznym objektom pridelí rovnaká farba. Z dôvodu spomenutých nevýhod som sa rozhodol zavrhnúť prvý spôsob riešenia.

Druhé riešenie sa snaží potlačiť predchádzajúce nevýhody tým spôsobom, že využije všetkých 2^{24} farieb, ktoré sú k dispozícii. V prvom riešení sa predpokladalo, že si užívateľ navolí rozlíšiteľné farby, preto nebolo potrebné maximalizovať farebný rozptyl z hľadiska rozlíšiteľnosti farieb. V druhom riešení to však urobiť treba. Viacero známych hašovacích funkcií (napr. MD5, SHA-1, SHA-2) spĺňajú vyššie spomenuté kritériá a preto som sa rozhodol použiť jednu z nich. JAVA v sebe priamo obsahuje implementáciu MD5 a SHA-1, preto som sa rozhodol zúžiť výber na tieto dve. Obe produkujú haš väčší ako 24 bitov, preto sa vhodnou projekciou vyberie 24 bitov, ktoré budú definovať farbu. Kritériom na výber hašovacej funkcie a vhodnej projekcie bol farebný rozptyl, ktorý vznikol pri číselných objektoch. Testovalo sa na číslach od 0 po 100, lebo sa predpokladá, že práve tieto čísla budú najčastejšími objektami pri generovaní farieb. Subjektívnym hodnotením sa ukázala SHA-1 s projekciou 5-28 bitu ako najlepšia voľba.

4. Príklady distribuovaných algoritmov

V tejto kapitole sa rozoberajú konkrétne distribuované algoritmy, ktoré sú na ukážku implementované v mojej aplikácii. Premenné potrebné na uskutočnenie výpočtu algoritmu sa zväčša ukladajú ako vlastnosti procesu. Je to z dôvodu ich čitateľnosti v popise procesov počas pozastavenia vizualizácie.

4.1 Voľba šéfa na kompletnom grafe so správami $O(N^2)$

Na kompletnom grafe, v ktorom má každý proces priame prepojenie so všetkými ostatnými procesmi, sa črtá jednoduché riešenie. Na začiatku každý proces pošle správu všetkým ostatným procesom, že chce byť šéf a čaká, že mu to ostatní dovolia. To sa dovolí iba procesu s najvyšším identifikátorom.

Vlastnosti procesu:

- **level** – určuje počet prijatých správ typu MESSAGE_ACCEPT.
- **leader** – ak sa rovná true, tak je proces šéfom, inak nie je.
- **leader id** – definuje identifikátor procesu, ktorý je šéfom. Pokiaľ šéf nie je zvolený, jeho hodnota je unknown.

Typy správ:

- **MESSAGE_CAPTURE** – používa sa na začiatku algoritmu, keď každý proces informuje všetky ostatné procesy o svojom identifikátore.
- **MESSAGE_ACCEPT** – posiela sa ako odpoveď na prijatie správy MESSAGE_CAPTURE, keď navyše platí, že identifikátor prijímajúceho procesu je menší ako odosielajúceho procesu.
- **MESSAGE_LEADER** – ak proces prijal od všetkých ostatných procesov správu MESSAGE_ACCEPT, tak vie, že jeho identifikátor je v sieti najväčší a preto sa môže považovať za šéfa. Následne pošle ostatným procesom správu typu MESSAGE_LEADER, v ktorej ich informuje o svojom identifikátore.

Algoritmus 4.1:

```
import algorithm.DistributedAlgorithm;
import network.Message;

public class CompleteGraphElection extends DistributedAlgorithm
{
  private static final int MESSAGE_CAPTURE=1;
  private static final int MESSAGE_ACCEPT=2;
  private static final int MESSAGE_LEADER=3;

  @Override
  public void Run()
  {
    SetProperty("level","0");
    SetProperty("leader", "false");
    SetProperty("leader id", "unknown");
    for (int i=0;i<GetLinksCount();i++)
    {
      SendMessageByLink(MESSAGE_CAPTURE, GetId(), i);
    }
    while (Integer.parseInt(GetProperty("level").toString())<
      GetLinksCount())
    {
      Message mes=ReceiveMessage();
      switch (mes.GetType())
      {
        case MESSAGE_CAPTURE:
          int id=Integer.parseInt(mes.GetObject().toString());
          if (id>GetId())
          {
            SendMessageByLink(MESSAGE_ACCEPT, "accept",
              mes.GetLinkNum());
          }
          break;
        case MESSAGE_ACCEPT:
          SetProperty("level",
             Integer.parseInt(GetProperty("level").toString())+1);
          break;
        case MESSAGE_LEADER:
          SetProperty("leader id", mes.GetObject());
          return;
      }
    }
    SetProperty("leader", "true");
    SetProperty("leader id", GetId());
    for (int i=0;i<GetLinksCount();i++)
    {
      SendMessageByLink(MESSAGE_LEADER, GetId(), i);
    }
  }
}
```

Analýza počtu správ:

Nech N označuje počet procesov v sieti, potom počet liniek je v kompletnom grafe $\frac{N\cdot(N-1)}{2}$. Podľa princípu fungovania algoritmu sa dá zistiť, že správy typu MESSAGE_CAPTURE sa pošlú po každej linke práve dvakrát. Pre každý smer práve

raz. Správy typu MESSAGE_ACCEPT sa po každej linke pošlú iba raz, lebo iba raz platí podmienka, že identifikátor prijímajúceho procesu je menší ako identifikátor odosielajúceho procesu. Správy typu MESSAGE_LEADER sa pošlú práve (N-1)-krát.

Výsledný počet správ je preto $\frac{3N^2 - N - 2}{2}$, asymptoticky $O(N^2)$.

4.2 Voľba šéfa na kompletnom grafe so správami O(N log N)

Problémom predchádzajúceho algoritmu bolo veľké množstvo vykomunikovaných správ, ktoré sa snaží znížiť tento algoritmus. Povolenia od ostatných procesov sa nebudú získavať paralelne ale sekvenčne. Pri porovnaní sa neberie do úvahy len identifikátor procesu, ale aj level (počet porazených procesov). Proces posiela správy MESSAGE_CAPTURE a vždy čaká na odpoveď MESSAGE_ACCEPT. Tá však príde iba vtedy, ak vyhral. Odpoveď na správu MESSAGE_CAPTURE závisí od toho, či už bol proces porazený alebo nie. Ak nie, rozhoduje veľkosť [level, id] lexikograficky. Ak áno, porovnanie nerobí on, ale rodič (proces, ktorý ho posledný krát zajal).

Vlastnosti procesu:

- **state** – definuje stav procesu. Hodnotami môže byť active, captured, killed. Každý proces je na začiatku active. Do stavu captured sa active alebo killed proces dostane prijatím správy MESSAGE_CAPTURE, keď navyše platí, že [level, id] posielajúceho procesu je väčšie ako [level, id] prijímajúceho procesu. Active proces prejde do stavu killed, keď obdrží správu MESSAGE_HELP a platí, že [level, id] posielajúceho procesu je väčšie ako [level, id] prijímajúceho procesu.
- **level** – určuje počet prijatých správ typu MESSAGE_ACCEPT.
- **leader** – nadobúda hodnoty true alebo false. Ak sa rovná true, tak je proces šéfom, inak šéfom nie je.
- **leader id** – definuje identifikátor procesu, ktorý je šéfom. Pokiaľ šéf nie je zvolený, jeho hodnota je unknown.

Typy správ:

- **MESSAGE_CAPTURE** – obdobne ako v algoritme 4.1 sa používa na informovanie ostatných procesov o [level, id].
- **MESSAGE_ACCEPT** – posiela sa ako odpoveď na prijatie správy MESSAGE_CAPTURE, keď navyše platí, že [level, id] prijímajúceho procesu je menší ako [level, id] odosielajúceho procesu.
- **MESSAGE_HELP** – ak proces obdrží správu MESSAGE_CAPTURE a už bol zajatý iným procesom (je v stave captured), porovnávanie [level, id] nerobí on, ale proces, ktorý ho poslednýkrát zajal. Na to sa používajú správy typu MESSAGE_HELP.
- **MESSAGE_VICTORY a MESSAGE DEFEAT** – proces obdrží správu MESSAGE_HELP od procesu, ktorého zajal. Za neho spraví porovnanie [level, id] a ak platí, že [level, id] prijímajúceho procesu je väčšie ako [level, id] prijaté v správe MESSAGE_HELP, tak ako odpoveď pošle MESSAGE_VICTORY. Inak pošle MESSAGE_DEFEAT.
- **MESSAGE_LEADER** – ak proces prijal od všetkých ostatných procesov správu MESSAGE_ACCEPT, tak sa vyhlási za šéfa a pomocou správy MESSAGE_LEADER informuje ostatné procesy o svojom identifikátore.

Algoritmus 4.2:

```
import algorithm.DistributedAlgorithm;
import network.Message;

public class CompleteGraphElection extends DistributedAlgorithm
{
  private static final int MESSAGE_CAPTURE=1;
  private static final int MESSAGE_ACCEPT=2;
  private static final int MESSAGE_HELP=3;
  private static final int MESSAGE_VICTORY=4;
  private static final int MESSAGE_DEFEAT=5;
  private static final int MESSAGE_LEADER=6;

  private int parent, actualLink;

  @Override
  public void Run()
  {
    SetProperty("state","active");
    SetProperty("level","0");
    SetProperty("leader", "false");
    SetProperty("leader id", "unknown");
    parent=-1;actualLink=0;
    Capture();
    while (true)
    {
      Message mes=ReceiveMessage();
      switch (mes.GetType())
      {
        case MESSAGE_CAPTURE:
          int[] myScore=new
             int[]{Integer.parseInt(GetProperty("level").toString()), GetId()};
          int[] hisScore=(int[])mes.GetObject();
          if ((GetProperty("state").equals("active") ||
            GetProperty("state").equals("killed"))
            && IsGreaterScore(hisScore, myScore))
          {
            SetProperty("state", "captured");
            parent=mes.GetLinkNum();
            SendMessageByLink(MESSAGE_ACCEPT, "", parent);
          }
          else if (GetProperty("state").equals("captured"))
          {
            SendMessageByLink(MESSAGE_HELP, mes.GetObject(), parent);
            Message mes2=ReceiveMessage(parent);
            if (mes2.GetType()==MESSAGE_DEFEAT)
            {
              SendMessageByLink(MESSAGE_ACCEPT, "", mes.GetLinkNum());
              parent=mes.GetLinkNum();
            }
          }
          break;
        case MESSAGE_ACCEPT:
          if (GetProperty("state").equals("active"))
          {
            SetProperty("level",
              Integer.parseInt(GetProperty("level").toString())+1);
            Capture();
          }
          break;
        case MESSAGE_HELP:
          myScore=new
            int[]{Integer.parseInt(GetProperty("level").toString()),GetId()};
          hisScore=(int[])mes.GetObject();
```

```
          if (IsGreaterScore(myScore, hisScore))
          {
            SendMessageByLink(MESSAGE_VICTORY, "", mes.GetLinkNum());
          }
          else
          {
            SendMessageByLink(MESSAGE_DEFEAT, "", mes.GetLinkNum());
            if (GetProperty("state").equals("active"))
            {
              SetProperty("state", "killed");
            }
          }
          break;
        case MESSAGE_LEADER:
          SetProperty("leader id", mes.GetObject());
          return;
      }
    }
  }

  private boolean IsGreaterScore(int[] score1, int[] score2)
  {
    return ((score1[0]>score2[0]) || ((score1[0]==score2[0]) &&
     (score1[1]>score2[1])));
  }

  private void Capture()
  {
    if (actualLink<GetLinksCount())
    {
      int[] mes=new
        int[]{Integer.parseInt(GetProperty("level").toString()),GetId()};
      SendMessageByLink(MESSAGE_CAPTURE, mes, actualLink);
      actualLink++;
    }
    else
    {
      SetProperty("leader id", GetId());
      SetProperty("leader", "true");
      for (int i=0;i<GetLinksCount();i++)
      {
        SendMessageByLink(MESSAGE_LEADER, GetId(), i);
      }
    }
  }
}
```

Popis metód:

- **boolean IsGreaterScore(int[] score1, int[] score2)**

  Keďže sa v algoritme vyžaduje posielanie dvoch objektov (identifikátor procesu a level procesu) a metóda SendMessageByLink podporuje odosielanie iba jedného objektu, oba objekty sa uložia do poľa a volaním SendMessageByLink sa pošle celé pole. Metóda IsGreaterScore lexikograficky porovnáva dve takéto polia.

- **void Capture()**

  Metóda sa volá, keď chce proces získať povolenie od ďalšieho procesu. Pokiaľ už získal všetky povolenia, znamená to, že sa proces stal šéfom. V tom prípade metóda Capture() informuje ostatné procesy o výsledku volieb.

4.3 Voľba šéfa na mriežke

Prípady, že mriežka môže mať aj tvar spájaného zoznamu, sa neberú do úvahy, a preto sa procesy na mriežke dajú rozdeliť do troch skupín podľa toho, koľko liniek z nich vychádza (2, 3 a 4 linkové). Z dôvodu nevykomunikovania veľkého množstva správ sa šéf bude voliť iba zo skupiny procesov, ktoré obsahujú dve linky a komunikácia bude prebiehať po obvode mriežky. Na začiatku algoritmu sa každý dvojlinkový proces považuje za šéfa a začne po oboch linkách posielať svoj identifikátor. Trojlinkové procesy slúžia na preposielanie správ, ktoré príjmu. Aby sa zabezpečilo, že správy nebudú zasahovať ďaleko od okraja mriežky, štvor-linkové procesy nebudú správy preposielať. Ak sa dvojlinkový proces považuje za šéfa a príde mu správa s väčším identifikátorom ako má, za šéfa sa už ďalej nepovažuje a správu prepošle po inej linke, ako ju prijal. Ak dvoj-linkový proces obdrží správu s menším identifikátorom ako má, správu ďalej nepreposiela. Práve jeden dvojlinkový proces obdrží správu so svojim identifikátorom. Vtedy sa vie, že je právoplatným šéfom a preto začne všetkým procesom oznamovať svoj identifikátor.

Vlastnosti procesu:

- **leader** – ak je hodnota rovná true, znamená to, že je proces šéf.
- **leader id** – určuje identifikátor šéfa. Ak nie je známe, kto je definitívnym šéfom, hodnota sa rovná unknown.

Typy správ:

- **MESSAGE_ELECT** – používa sa pri voľbe šéfa. Obsahuje v sebe identifikátor procesu, ktorý súťaží o post šéfa.
- **MESSAGE_BROADCAST** – po zvolení šéfa sa jeho identifikátor rozpošle týmito správami do všetkých procesov.

Algoritmus 4.3:

```
import algorithm.DistributedAlgorithm;
import network.Message;

public class GridElection extends DistributedAlgorithm
{
  private static final int MESSAGE_ELECT=1;
  private static final int MESSAGE_BROADCAST=2;

  @Override
  public void Run()
  {
    SetProperty("leader", "false");
    SetProperty("leader id", "unknown");
    if (GetLinksCount()==2)
    {
      SetProperty("leader", "true");
      for (int i=0;i<2;i++)
      {
        SendMessageByLink(MESSAGE_ELECT, GetId(), i);
      }
    }
    while (true)
    {
      Message mes=ReceiveMessage();
      switch (mes.GetType())
      {
        case MESSAGE_ELECT:
          if (GetLinksCount()==3)
          {
            for (int i=0;i<GetLinksCount();i++)
            {
              if (i!=mes.GetLinkNum())
                SendMessageByLink(mes.GetType(), mes.GetObject(), i);
            }
          }
          else if (GetLinksCount()==2)
          {
            int id=Integer.parseInt(mes.GetObject().toString());
            if (GetId()<id)
            {
              SetProperty("leader", "false");
              for (int i=0;i<GetLinksCount();i++)
              {
                if (i!=mes.GetLinkNum())
                  SendMessageByLink(mes.GetType(), mes.GetObject(), i);
              }
            }
            else if (GetId()==id)
            {
              for (int i=0;i<GetLinksCount();i++)
              {
                SendMessageByLink(MESSAGE_BROADCAST, GetId(), i);
              }
              return;
            }
          }
          break;
        case MESSAGE_BROADCAST:
          SetProperty("leader id", mes.GetObject());
          for (int i=0;i<GetLinksCount();i++)
          {
            if (i!=mes.GetLinkNum())
              SendMessageByLink(mes.GetType(), mes.GetObject(), i);
```

```
            }
            return;
        }
    }
  }
}
```

Analýza počtu správ:

Nech má mriežka rozmery M x N. Na začiatku sú štyri rožné procesy aktívne a posielajú správy typu MESSAGE_ELECT oboma smermi. Každá z týchto správ prejde maximálne po celom obvode a keďže na prejdenie jednej správy po celom obvode sa spotrebuje maximálne 4M+4N preposielaní, tak výsledný počet poslaných správ typu MESSAGE_ELECT je maximálne 32(M+N) = O(M+N).

Po zvolení šéfa je potrebné upovedomiť každý proces o jeho identifikátore (pomocou MESSAGE_BROADCAST). Každý proces má maximálne štyri linky a po každej z nich mu môže prísť maximálne jedna správa typu MESSAGE_BROADCAST. Teda výsledný počet poslaných správ typu MESSAGE_BROADCAST je maximálne 4MN = O(MN).

4.4 Voľba šéfa na strome

Strom je súvislý acyklický graf, v ktorom sa dajú rozlíšiť dva typy vrcholov/procesov. Tie, ktoré sú listami (majú jednu linku), a tie, ktoré listami nie sú (majú aspoň dve linky). Na začiatku algoritmu sú iniciátormi listy, ktoré po linke pošlú svoj identifikátor. Procesy, ktoré nie sú listami, po linkách postupne prijímajú identifikátory iných procesov. Každý identifikátor príde po samostatnej linke a ak ich proces prijme L-1 (L označuje počet liniek vychádzajúcich z procesu), tak po linke, po ktorej žiadnu správu neprijal, pošle maximálny identifikátor, o ktorom sa doposiaľ dozvedel. Následne si na tej istej linke počká na odpoveď, ktorá v sebe prinesie informáciu o najväčšom identifikátore, o ktorom sa dozvedel proces na druhej strane linky. Proces, ktorý po spomínanej linke prijme správu, už pozná maximálny identifikátor z celého stromu. Preto ho prepošle po ostatných linkách, aby o výsledku informoval aj ostatné procesy.

Vlastnosti procesu:

- **leader** – ak sa hodnota rovná true, znamená to, že proces je šéf.
- **leader id** – určuje identifikátor šéfa. Ak nie je známe, kto je šéfom, hodnota sa rovná unknown.

Algoritmus 4.4:

```
import algorithm.DistributedAlgorithm;
import network.Message;

public class TreeElection extends DistributedAlgorithm
{
  private static final int MESSAGE_TOK=1;

  @Override
  public void Run()
  {
    SetProperty("leader", "false");
    SetProperty("leader id", "unknown");
    boolean[] received=new boolean[GetLinksCount()];
    int numReceived=0;
    for (int i=0;i<GetLinksCount();i++)
      received[i]=false;
    int maxId=GetId();
    while (numReceived<(GetLinksCount()-1))
    {
      Message mes=ReceiveMessage();
      if (!received[mes.GetLinkNum()])
      {
        received[mes.GetLinkNum()]=true;
        numReceived++;
      }
      int id=((Integer)mes.GetObject()).intValue();
      if (id>maxId)
        maxId=id;
    }
    int unreceivedMessage=0;
    while (received[unreceivedMessage])
      unreceivedMessage++;
    SendMessageByLink(MESSAGE_TOK, maxId, unreceivedMessage);
    Message mes=ReceiveMessage(unreceivedMessage);
    int id=((Integer)mes.GetObject()).intValue();
    if (id>maxId)
      maxId=id;
    if (maxId==GetId())
      SetProperty("leader", "true");
    SetProperty("leader id", maxId);
    for (int i=0;i<GetLinksCount();i++)
    {
      if (i!=unreceivedMessage)
        SendMessageByLink(MESSAGE_TOK, maxId, i);
    }
  }
}
```

Analýza počtu správ:

Na začiatku algoritmu listy pošlú po jedinej linke, ktorú majú, jednu správu. Na konci algoritmu po nej obdržia druhú. Proces, ktorý nie je list, čaká na L-1 správ (L označuje počet liniek vychádzajúcich z procesu). Keď ich obdrží, po zvyšnej linke pošle jednu správu a jednu po nej príjme. Následne odošle L-1 správ. Celkovo sa v sieti odošle dvakrát toľko správ ako je počet liniek. Počet liniek je v strome N-1, kde N označuje počet procesov, a preto sa správ vykomunikuje 2N-2 = O(N).

4.5 Korach-Kutten-Moran

Korach, Kutten a Moran ukázali, že existuje blízka spojitosť medzi voľbou šéfa a traverzovaním. Ich hlavným výsledkom je všeobecná konštrukcia efektívneho algoritmu na voľbu šéfa pre triedu sietí, pre ktoré existuje traverzovanie. Traverzovanie je proces systematického navštevovania vrcholov v grafe, pričom sa každý vrchol navštívi práve raz.

Pre správne fungovanie algoritmu sa v sieti musí nachádzať aspoň jeden iniciátor (vlastnosť initiator sa rovná true). Na začiatku algoritmu sa z každého iniciátora spustí traverzovanie siete s jeho identifikátorom. Ak traverzovanie skončí, iniciátor traverzovania sa stane šéfom. Algoritmus garantuje, že sa to stane pre práve jedno traverzovanie. V sieti môže vzniknúť viacero traverzovaní z toho istého procesu, preto aby sa navzájom rozlíšili, algoritmus pracuje v leveloch. Ak sa v sieti začali aspoň dve traverzovania, správy dorazia do procesu, ktorý bol navštívený inou správou. Ak táto situácia nastane, traverzovanie je zrušené. Novým cieľom sa stane priniesť obe správy do toho istého procesu, v ktorom začne nové traverzovanie. Dve správy vyvolajú nové traverzovanie, iba ak majú rovnaký level. Novo vygenerované traverzovanie má level o jedno väčší. Ak sa správa stretne s druhou správou, ktorej level je väčší, alebo dorazí do procesu, ktorý už bol navštívený správou s vyšším levelom, prichádzajúca správa je jednoducho zrušená bez toho, aby ovplyvnila správu s vyšším levelom. Za účelom priniesť dve správy rovnakého levelu do toho istého procesu, sa každá správa nachádza v troch stavoch: annexing, chasing alebo waiting. Nakoľko sa správa v stave waiting po linkách neposiela, nie je pre tento stav definovaný typ správy. Správy typu MESSAGE_ANNEX a MESSAGE_CHASE sa dajú reprezentovať ako (q, l), kde q určuje identifikátor iniciátora a l jeho level.

Správa (q, l) sa v stave annexing posiela v sieti po linkách, ktoré určuje traverzovacia funkcia Trav, až kým nenastane jedna z nasledovných situácií:

- Traverzovací algoritmus sa skončí: q sa vyhlási za šéfa (Case IV).
- Správa dorazí do procesu, ktorého level > l: Správa je zrušená.
- Správa dorazí do procesu, v ktorom čaká správa toho istého levelu: Obe správy sú zrušené a z procesu začne nové traverzovanie (Case II).
- Správa dorazí do procesu, ktorého level = l a bol nedávno navštívený správou s identifikátorom cat > q (Case VI), alebo správou v stave chasing (Case III): Správa prejde do stavu waiting.
- Správa dorazí do procesu, ktorého level = l a bol nedávno navštívený správou s identifikátorom cat < q: Správa prejde do stavu chasing a je poslaná po tej istej linke ako predchádzajúca správa (Case V).

Správa (q, l) sa v stave chasing posiela v sieti po linkách, po ktorých bola naposledy prechádzajúca správa poslaná, až kým nenastane jedna z nasledovných situácií:

- Správa dorazí do procesu levelu lev > l: Správa sa zruší.
- Správa dorazí do procesu, v ktorom sa nachádza čakajúca správa s levelom rovným l: Obe správy sa zrušia a z procesu začne nové traverzovanie s levelom o 1 vyšším (Case II).
- Správa dorazí do procesu s levelom l, v ktorom naposledy prechádzajúca správa bola v stave chasing: Správa prejde do stavu waiting (Case III).

Správa v stave waiting čaká v procese, až kým nenastane jedna z nasledovných situácií:

- Správa s vyšším levelom dorazí do rovnakého procesu: Čakajúca správa je zrušená (Case I).
- Do procesu dorazí správa s rovnakým levelom: Obe správy sa odstránia a z procesu začne nové traverzovanie s levelom o 1 vyšším (Case II).

Vlastnosti procesu:

- **leader** – ak sa hodnota rovná true, znamená to, že je proces šéf.
- **leader id** – určuje identifikátor šéfa. Ak šéf nie je zvolený, hodnota sa rovná unknown.
- **lev** – udáva level procesu. Ten sa rovná maximálnemu levelu správy, ktorá sa nachádzala v procese.
- **cat** – obsahuje identifikátor iniciátora poslednej správy typu MESSAGE_ANNEX, ktorá bola procesom poslaná. Ak taký identifikátor neexistuje, hodnota je undefined.
- **wait** – je rovná undefined, ak žiadna správa nečaká v procese. Inak určuje identifikátor iniciátora čakajúcej správy.
- **last** – je potrebná pre správy typu MESSAGE_CHASE. Určuje číslo linky, po ktorej sa poslala posledná správa typu MESSAGE_ANNEX.

Algoritmus 4.5:

```
import algorithm.DistributedAlgorithm;
import algorithm.MessageReceivedListener;
import network.Message;

public class KKM extends DistributedAlgorithm
{
  private static final int MESSAGE_ANNEX=1;
  private static final int MESSAGE_CHASE=2;
  private static final int MESSAGE_LEADER=3;

  @Override
  public void Run()
  {
    MessageReceivedListener leaderListener=new MessageReceivedListener()
    {
      public void OnMessageReceive(Message mes)
      {
        if (GetProperty("leader id").equals("unknown"))
          for (int i=0;i<GetLinksCount();i++)
            if (i!=mes.GetLinkNum())
              SendMessageByLink(MESSAGE_LEADER, mes.GetObject(), i);
        SetProperty("leader", "false");
        SetProperty("leader id", mes.GetObject());
        StopProcess();
      }
    };
    SetMessageReceivedListener(MESSAGE_LEADER, leaderListener);

    SetProperty("leader", "false");SetProperty("leader id", "unknown");
    SetProperty("lev", "-1");SetProperty("cat", "undefined");
    SetProperty("wait", "undefined");SetProperty("last", "undefined");
    if ((GetProperty("initiator")!=null) &&
      (GetProperty("initiator").equals("true")))
    {
      SetProperty("lev", Integer.parseInt(GetProperty("lev").toString())+1);
      SetProperty("last",
        Trav(GetId(),Integer.parseInt(GetProperty("lev").toString())));
      SetProperty("cat", GetId());
      SendMessageByLink(MESSAGE_ANNEX, ImplodeData(GetId(),
        GetProperty("lev")),Integer.parseInt(GetProperty("last").toString()));
    }
    while (true)
    {
      Message mes=ReceiveMessage();
      int q=Integer.parseInt(((Object[])mes.GetObject())[0].toString());
      int l=Integer.parseInt(((Object[])mes.GetObject())[1].toString());
      if (l>Integer.parseInt(GetProperty("lev").toString()))
      {
        //Case I.
        SetProperty("lev", l);SetProperty("cat", q);
        SetProperty("wait", "undefined");SetProperty("last", Trav(q, l));
        SendMessageByLink(MESSAGE_ANNEX, ImplodeData(q, l),
          Integer.parseInt(GetProperty("last").toString()));
      }
      else if ((l==Integer.parseInt(GetProperty("lev").toString())) &&
        (!GetProperty("wait").equals("undefined")))
      {
        //Case II.
        SetProperty("wait", "undefined");SetProperty("cat", GetId());
        SetProperty("lev", Integer.parseInt(GetProperty("lev").toString())+1);
        SetProperty("last", Trav(GetId(),
          Integer.parseInt(GetProperty("lev").toString())));
        SendMessageByLink(MESSAGE_ANNEX, ImplodeData(GetId(),
```

```
            GetProperty("lev")),
            Integer.parseInt(GetProperty("last").toString()));
        }
        else if ((l==Integer.parseInt(GetProperty("lev").toString())) &&
          (GetProperty("last").equals("undefined")))
        {
          //Case III.
          SetProperty("wait", q);
        }
        else if ((l==Integer.parseInt(GetProperty("lev").toString())) &&
          (mes.GetType()==MESSAGE_ANNEX) &&
          (q==Integer.parseInt(GetProperty("cat").toString())))
        {
          //Case IV.
          SetProperty("last", Trav(q, l));
          if (Integer.parseInt(GetProperty("last").toString())==-1)
            SetLeader();
          else
            SendMessageByLink(MESSAGE_ANNEX, ImplodeData(q, l),
              Integer.parseInt(GetProperty("last").toString()));
        }
        else if ((l==Integer.parseInt(GetProperty("lev").toString())) &&
          (((mes.GetType()==MESSAGE_ANNEX) &&
          (q>Integer.parseInt(GetProperty("cat").toString()))) ||
          (mes.GetType()==MESSAGE_CHASE)))
        {
          //Case V.
          SendMessageByLink(MESSAGE_CHASE, ImplodeData(q, l),
            Integer.parseInt(GetProperty("last").toString()));
          SetProperty("last", "undefined");
        }
        else if (l==Integer.parseInt(GetProperty("lev").toString()))
        {
          //Case VI.
          SetProperty("wait", q);
        }
      }
    }

    private int Trav(int p, int lev)
    {
      //Traversing function
    }

    private Object ImplodeData(Object p, Object lev)
    {
      return new Object[]{p, lev};
    }

    private void SetLeader()
    {
      SetProperty("leader", "true");
      SetProperty("leader id", GetId());
      for (int i=0;i<GetLinksCount();i++)
      {
        SendMessageByLink(MESSAGE_LEADER, GetId(), i);
      }
      StopProcess();
    }
}
```

Popis metód:

- **int Trav(int p, int lev)**

  Slúži na traverzovanie siete. Argumentami metódy sú: p – identifikátor procesu, v ktorom traverzovanie začalo; lev – level traverzovania. Výstupom metódy je číslo linky, po ktorej sa má správa poslať.

- **Object ImplodeData(Object p, Object lev)**

  Keďže metóda SendMessageByLink podporuje posielanie iba jedného objektu a algoritmus vyžaduje zasielanie dvoch objektov (číslo procesu, v ktorom traverzovanie vzniklo a level traverzovania), metóda ImplodeData vytvorí z dvoch objektov jeden tým spôsobom, že ich uloží do poľa.

- **void SetLeader()**

  Ak proces zistí, že sa traverzovanie správy skončilo, značí to, že sa proces stal šéfom. Metóda SetLeader slúži na oboznámenie ostatných procesov o svojom identifikátore.

Záver

V diplomovej práci som sa zaoberal návrhom a implementáciou systému, ktorý podporuje vizuálnu simuláciu správania distribuovaných algoritmov. Primárnym cieľom bolo vytvoriť systém, ktorý je pre distribuované programovanie flexibilný, efektívne implementovaný a nezávislý na platforme. Sekundárnym cieľom bolo v systéme implementovať štandardné distribuované algoritmy, ktoré môžu byť vďaka vizualizácii v reálnom čase dobrou učebnou pomôckou.

Flexibilita distribuovaného programovania spočíva v čo najlepšom oddelení troch interaktívnych zložiek: zdrojového kódu algoritmu, spôsobu vizualizácie a časovania správ. Tento cieľ sa mi podarilo naplniť, preto napr. pri zmene časovania správ alebo spôsobu vizualizácie nie je potrebné meniť zdrojový kód algoritmu.

Pri implementácii systému som riešil viacero problémov, ktoré zväčša vznikali z podstaty programovania pomocou vlákien. Najväčšia výpočtová záťaž procesora vzniká v čase vizualizácie, preto som kládol dôraz na efektívnosť jej implementácie. Pre tento účel som využíval rôzne dátové štruktúry, ktoré umožňovali efektívny prístup k požadovaným objektom.

Java sa ukázala ako dobrý prostriedok vedúci k multiplatformovosti, preto som vývoj systému robil v nej. Využíval som SWING sadu grafických komponentov, ktorá zaručuje, že sa aplikácia na platformách podporujúcich Java zobrazí bez veľkých rozdielov.

Zdrojový kód distribuovaného algoritmu sa z dôvodu konzistencie celého systému programuje v jazyku Java. Užívateľovi sú sprístupnené funkcie, ktoré ponúkajú možnosť distribuovaného programovania. Na ukážku som implementoval zopár štandardných distribuovaných algoritmov, ktoré sú zväčša zamerané na voľbu šéfa. Nakoľko distribuovaný algoritmus neprebieha sekvenčne ale paralelne, ťažko sa predstavujú okrajové situácie priebehu algoritmu. Vďaka môjmu systému sa však môže stať programovanie distribuovaných algoritmov lepšie pochopiteľné.

V budúcnosti by sa systém mohol prerobiť do webovej verzie. Výhodou by bolo zdieľanie algoritmov medzi viacerými užívateľmi a kompilácia distribuovaného algoritmu na serveri. Užívateľ by potom nemusel mať nainštalovaný Java kompilátor na

svojom počítači. Prevádzkovanie kompilácie na serveri však nesie so sebou riziko narušenia bezpečnosti, ktoré by muselo byť odstránené.

Zoznam použitej literatúry

[1] MATCHA, K.: A Look At The Java Class Loader. http://www.javalobby.org/java/forums/t18345.html [2009-01-25]

[2] KORACH, E., KUTTEN, S. a MORAN, S.: A modular technique for the design of efficient distributed leader finding algorithms. ACM Transactions on Programming Languages and Systems, vol. 12. 1990. New York. s. 84-101. ISSN 0164-0925

[3] KATUŠČÁK, D.: Ako písať vysokoškolské a kvalifikačné práce. 2. vyd. Bratislava. 1998. 119 s. ISBN 80-85697-82-3

[4] ATTIYA, H.: Constructing efficient election algorithms from efficient traversal algorithms. 2nd International Workshop on Distributed Algorithms, J. Van Leeuwen, vol. 312 of Lecture notes in Computer Science, Springer Berlin/Heidelberg. 1988. s. 337-344. ISBN 978-3-540-19366-1

[5] Deadlock. http://en.wikipedia.org/wiki/Deadlock [2009-04-14]

[6] MULLIGAN, T.: GUI Building in NetBeans IDE 5.5. http://www.netbeans.org/kb/55/quickstart-gui.html [2009-01-25]

[7] TEL, G.: Introduction to distributed algorithms. Glasgow. 1994. 119 s. ISBN 0-521-47069-2

[8] Java. http://java.sun.com/ [2008-11-07]

[9] Lazarus. http://www.lazarus.freepascal.org/ [2008-11-05]

[10] SINGH, G.: Leader election in complete networks. Proceedings of the eleventh annual ACM symposium on Principles of distributed computing. New York. 1992. s. 179-190. ISBN 0-89791-495-3

[11] AVONDOLIO, D., MITCHELL, M. W., RICHARDSON, W. C., SCANLON, J. a SCHRAGER, S.: Professional Java, JDK 6 Edition. Indianapolis. 2007. ISBN 0-471-77710-2

[12] SHA hash functions. http://en.wikipedia.org/wiki/SHA_hash_functions [2009-03-26]

[13] KORACH, E., MORAN, S. a ZAKS, S.: Tight lower and upper bounds for some distributed algorithms for a complete network of processors. Proceedings of the third annual symposium on Principles of distributed computing. New York. 1984. s. 199-207. ISBN 0-89791-143-1

Slovník termínov

- **asynchrónny systém** – systém, v ktorom sa jednotlivé procesy správajú autonómne a ich vykonávanie nie je ovládané žiadnym synchronizačným mechanizmom.
- **deadlock** – je situácia, ktorá vzniká, ak sa viacero procesov čaká navzájom, kým sa uvoľnia prostriedky alebo ak procesy čakajú na prostriedky v cykle. Takisto sa používa označenie uviaznutie.
- **distribuovaný algoritmus** – algoritmus spustený na viacerých počítačoch, resp. procesoch, ktoré pri výpočte medzi sebou komunikujú.
- **FIFO** – skratka pre first-in-first-out. Termín označuje spôsob ukladania a čítania objektov, pri ktorom sa prvky, ktoré sa uložili najprv, aj ako prvé z pamäte vyberú.
- **IDE** – skratka z *Integrated development environment* označujúca vývojové prostredie, ktoré programátorom uľahčuje prácu. Obsahuje editor zdrojového kódu, kompilátor, resp. interpreter a väčšinou aj debugger.
- **JDK** – *Java Development Kit* je súbor základných nástrojov určených pre vývoj aplikácií na platforme Java.
- **JRE** – *Java Runtime Environment* obsahuje implementáciu JVM.
- **JVM** – *Java Virtual Machine* je sada programov a dátových štruktúr, ktoré umožňujú vykonávanie programov písaných v Java.
- **linka** – komunikačné prepojenie dvoch procesov. Vďaka nej si môžu prepojené procesy posielať správy.
- **multiplatformový softvér** – softvér spustiteľný na viacerých platformách (kombináciách software a hardware).
- **polling** – inými slovami činné čakanie. Označuje stav, v ktorom proces čaká na splnenie určitej podmienky tým spôsobom, že jej splniteľnosť pravidelne kontroluje.
- **proces** – samostatná výpočtová jednotka. V distribuovaných algoritmoch sa ním označuje počítač, na ktorom je spustený distribuovaný algoritmus.
- **sieť** – systém procesov a liniek.

- **synchrónny systém** – systém, v ktorom medzi elementami existuje synchronizačný mechanizmus zabezpečujúci ich upovedomenie v rovnakých časových okamihoch.
- **vizualizácia v reálnom čase** – vizualizácia, ktorá plynulo zobrazuje zmeny stavu vizualizovaného prostredia.

Prílohy

Na webovej stránke Fakulty matematiky, fyziky a informatiky Univerzity Komenského sú dostupné zdrojové kódy, skompilovaná verzia systému a elektronická verzia tejto práce.