VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE PRO DEMONSTRACI SYNCHRONIZAČNÍCH MECHANISMŮ V DISTRIBUOVANÝCH SYSTÉMECH

APPLICATION FOR DEMONSTRATION OF SYNCHRONIZATION MECHANISMS IN DISTRIBUTED SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **LENKA KLIMČÍKOVÁ**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. FRANTIŠEK ZBOŘIL, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání bakalářské práce

Studentka: **Klimčíková Lenka**
Program: Informační technologie
Název: **Aplikace pro demonstraci synchronizačních mechanismů v distribuovaných systémech**
Application for Demonstration of Synchronization Mechanisms in Distributed Systems
Kategorie: Operační systémy

Zadání:

1. Seznamte se s významnými synchronizačními mechanismy pro distribuované systémy komunikujícími předáváním zpráv (Lamport, Raymond, Meakawa, Suzuki-Kasami).
2. Navrhněte aplikaci, která by uživatelsky přívětivým způsobem demonstrovala funkčnost a použití těchto algoritmů.
3. Vytvořte v prostředí JAVA nebo MPI knihovny pro realizaci těchto synchronizačních mechanismů.
4. Realizujte demonstrační aplikaci, která podpoří výuku těchto algoritmů svoji názorností, sadou vhodných příkladů a možností s těmito algoritmy experimentovat.

Literatura:

- A.D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, ISBN: 9780521189842, Cambridge University Press, 2011.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Zbořil František, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať webovú aplikáciu demonštrujúcu vybrané synchronizačné mechanizmy v distribuovaných systémoch. Jedná sa o algoritmy komunikujúce predávaním správ pre zaistenie vzájomného vylúčenia procesov pri snahe o prístup do kritickej sekcie. Implementovanými sú Lamportov algoritmus, Maekawov algoritmus, Raymondov algoritmus a Suzuki-Kasami vysielací algoritmus. Aplikácia je implementovaná v jazyku Java s využitím Spring Boot frameworku pre načúvanie na rôznych koncových bodoch. Ďalej je využitý nástroj Thymeleaf na výmenu dát medzi backendom a frontendom aplikácie a HTML + JavaScript jazyk pre dynamické vykresľovanie posielaných správ na obrazovku. Dané algoritmy sú implementované v jednotlivých knižniciach, ktoré následne využíva finálne vytvorená aplikácia. Aplikácia prehľadne demonštruje ich funkčnosť na rôznych príkladoch pre čo najlepšie porozumenie. Je primárne určená pre podporu výučby študentov Fakulty informačných technológií Vysokého učenia technického v Brne.

Abstract

The aim of this thesis is to design and create a web application for the demonstration of selected synchronization mechanisms in distributed systems. The algorithms communicate by means of message-passing to ensure mutual exclusion of the processes in an effort to access the critical section. Implemented are Lamport's algorithm, Maekawa's algorithm, Raymond's algorithm and Suzuki-Kasami broadcasting algorithm. The application is implemented in Java programming language with the use of Spring Boot framework for listening on different endpoints. Thymeleaf template engine is used to exchange the necessary data between backend and frontend of the application. HTML + JavaScript language are used for dynamic rendering of sent messages to the screen. The algorithms are implemented in separate libraries, which are then used by the web application. The application illustratively demonstrates function of each algorithm with loads of different examples for the best possible understanding. It's primarily intended for students of Faculty of Information Technology, Brno University of Technology.

Kľúčové slová

webová aplikácia, výuková demonštračná aplikácia, distribuované systémy, synchronizácia procesov, komunikácia predávaním správ, algoritmus, vzájomné vylúčenie, kritická sekcia, token, kvórum, uviaznutie, vyhladovenie, Lamport, Maekawa, Raymond, Suzuki-Kasami, Java, Spring Boot, Thymeleaf, Javascript

Keywords

web application, educational demonstrative application, distributed systems, process synchronization, communication by message-passing, algorithm, mutual exclusion, critical section, token, quorum, deadlock, starvation, Lamport, Maekawa, Raymond, Suzuki-Kasami, Java, Spring Boot, Thymeleaf, Javascript

Citácia

KLIMČÍKOVÁ, Lenka. *Aplikace pro demonstraci synchronizačních mechanismů v distribuovaných systémech.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. František Zbořil, Ph.D.

Aplikace pro demonstraci synchronizačních mechanismů v distribuovaných systémech

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána Doc. Ing. Františka Zbořila, Ph.D. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Lenka Klimčíková
8. mája 2022

Poďakovanie

Touto cestou by som sa chcela poďakovať vedúcemu mojej práce pánovi Doc. Ing. Františkovi Zbořilovi Ph.D. za pomoc pri tvorbe bakalárskej práce, za jeho čas, cenné rady a odborné vedenie.

Obsah

Kapitola 1

Úvod

Synchronizácia procesov je jedným z najzásadnejších problémov v distribuovaných systémoch. Jej úlohou je zaručiť vzájomné vylúčenie procesov pri snahe o súčasný prístup k zdieľanému zdroju, inak nazývanému kritická sekcia. To znamená, že iba jeden proces bude mať v danom čase prístup do kritickej sekcie. Keďže distribuované systémy nemôžu na tento účel využiť zdieľané premenné, predávanie správ je jediným spoľahlivým prostriedkom na implementáciu vzájomného vylúčenia. Práve tento spôsob využívajú viaceré existujúce algoritmy, pričom každý z nich rieši danú problematiku mierne odlišným spôsobom.

Táto práca sa zaoberá štyrmi mechanizmami na zaistenie vzájomného vylúčenia - Lamportov, Maekawov, Raymondov a Suzuki-Kasami algoritmus. Žiaci Fakulty informačných technológií Vysokého učenia technického v Brne sa nimi zaoberajú v predmete Paralelní a distribuované systémy. Neexistuje však pre nich v súčasnej dobe nástroj, ktorý by im pomohol lepšie a názornejšie problematiku porozumieť.

Práve za týmto účelom vznikla táto práca, ktorej cieľom je preštudovať uvedené mechanizmy a navrhnúť a zhotoviť webovú aplikáciu. Aplikácia bude názorným spôsobom demonštrovať ich funkcionalitu, princípy, ukáže ich fungovanie na rozličných príkladoch a zefektívni tým ich učenie a pochopenie. Funkcionalita algoritmov bude implementovaná pomocou jazyka Java a webová aplikácia, ktorá bude algoritmy demonštrovať na niekoľkých príkladoch, bude vytvorená príslušnými webovými nástrojmi. Práca má potenciál urýchliť a zjednodušiť učenie a predpokladá sa jej využívanie každým ďalším ročníkom študentov spomínaného predmetu.

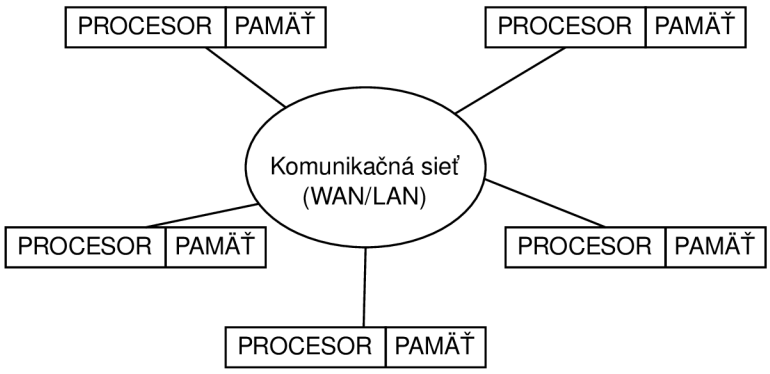
Práca je rozdelená do dvoch častí. Prvá, teoretická časť, sa skladá z dvoch kapitol. Kapitola 2 sa zaoberá obecnou problematikou synchronizácie v distribuovaných systémoch a vysvetľuje mechanizmy, ktoré sú pre jej zaistenie potrebné. V kapitole 3 sú rozobraté jednotlivé algoritmy, spôsob ich práce a požiadavky na ich správne fungovanie. Druhá, praktická časť práce, pojednáva o spôsobe implementácie algoritmov a webovej aplikácie. Ako prvý je v kapitole 4 rozobratý návrh výzoru aplikácie. Ďalšia kapitola 5 už hovorí o konkrétnych technológiách využitých k implementácii a popisuje jej architektonický návrh. V kapitole 6 sú rozobraté podrobnejšie implementačné detaily knižníc daných algoritmov a webovej časti. Spomína tiež niektoré zo vzniknutých problémov a ich riešenie. Posledná kapitola 7 sa zaoberá testovaním vytvorenej aplikácie a užívateľskou spätnou väzbou.

Kapitola 2

Synchronizácia v distribuovaných systémoch

Distribuovaný systém (2.1) je kolekcia nezávislých počítačov (procesov), ktoré komunikujú predávaním správ a spolupracujú na vyriešení problému, ktorý by nemohol byť vyriešený individuálne. Počítače pracujú autonómne, pričom sa však pre užívateľov táto kolekcia javí ako jeden koherentný systém. Rozdiely medzi týmito počítačmi a spôsob ich komunikácie je schovaný od užívateľa. [14]

Jedna z najpopulárnejších definícií distribuovaného systému znie:
Distribuovaný systém je taký systém, v ktorom zlyhanie počítača, o ktorom ste vôbec nemali tušenia, že existuje, učiní váš vlastný počítač nepoužiteľným [8]."



Obr. 2.1: Typický distribuovaný systém. Každý počítač má svoju vlastnú súkromnú pamäť, počítače sú prepojené komunikačnou sieťou a vymieňajú si informácie posielaním správ medzi procesormi. Prevzaté z [6].

Dôležitými vlastnosťami distribuovaných systémov sú: [6]

- Absencia bežných fyzických hodín
- Absencia zdieľanej pamäte - kľúčová vlastnosť požadujúca posielanie správ pre komunikáciu. Z tejto vlastnosti tiež vyplýva absencia spoločných fyzických hodín.

- Geografická separácia - čím viac geograficky ďalej od seba procesory sú, tým reprezentatívnejší systém je. Okrem WAN siete (wide-area network, teda rozsiahla sieť ako napríklad Internet) sú však už používané procesory spojené cez LAN siete (local-area network, ako napríklad v kancelárii či skupine budov) ako malé distribuované systémy.
- Autonómia a heterogenita - rýchlosť procesorov môže byť rôzna a každý môže bežať na inom operačnom systéme.

Kvôli absencii spoločných hodín musí byť pre viacero nezávislých prístrojov, bežiacich súčasne či už v rôznych časových zónach alebo na rôznych kontinentoch, zaistený spôsob synchronizácie, aby boli schopné efektívne spolupracovať. Je totiž dôležité, aby sa viacero procesov nesnažilo súčasne pristúpiť k zdieľanému zdroju (ako napríklad tlačiareň), ale spolupracovalo vo vzájomnom poskytovaní dočasného exkluzívneho prístupu. [14].

V centralizovaných systémoch je čas jednoznačný. Takmer všetky počítače disponujú precízne opracovaným kremenným kryštálom, ktorý slúži ako obvod na sledovanie času. Na základe dobre definovanej frekvenčnej oscilácie tohto kremeňa je operačný systém počítača schopný presne monitorovať vnútorný čas. Hoci je táto frekvencia primerane stabilná, nie je možné zaručiť, že všetky kryštály rôznych počítačov budú pracovať na rovnakej frekvencii, a teda ich hodiny postupne stratia synchronizáciu a ukážu rôzne hodnoty. Na vyriešenie tohto problému je využívaný Network Time Protocol (NTP, Sieťový časový protokol), kde klient kontaktuje server a na základe odchýlky vypočítanej pomocou poslaných správ klient svoje hodiny buď zrýchli alebo spomalí. Ďalšou možnosťou je tiež Berkeley algoritmus, v ktorom naopak server pravidelne kontaktuje klientov s dotazom na ich aktuálny čas a na základe vypočítanej priemernej hodnoty potom jednotlivých klientov informuje, ako si majú upraviť svoj čas. [14].

V centralizovanom systéme sa teda môžu vďaka konkrétnym hodnotám hodín všetky zúčastnené stroje dohodnúť na globálnej časovej osi, v ktorej sa udalosti dejú. Na čom ale skutočne na časovej osi záleží je, v akom poradí sa súvisiace udalosti odohrali. Nie je dôležité vedieť, v akom reálnom čase sa odohrala udalosť a a v akom udalosť b, pokiaľ vieme, že a sa stala pred b. V distribuovanom systéme je niekedy nemožné určiť, že jedna z dvoch udalostí sa udiala skôr. Synchronizácia procesov má teda zaručiť aspoň čiastočné usporiadanie medzi udalosťami. To je zaručené reláciou *happened-before* (stalo sa pred tým, tiež nazývaná relácia kauzality). [7]

2.1 Relácia happened-before (relácia kauzality)

Obecne by bolo možné povedať, že udalosť a sa stala pred udalosťou b, ak a sa stala v skoršom čase ako b. Táto teória by však platila iba v prípade fyzického času, teda v systéme, ktorý disponuje reálnymi fyzickými hodinami. Distribuovaný systém však fyzickými hodinami nedisponuje, preto bude definovaná táto relácia bez použitia fyzických hodín. Najskôr je však potrebné definovať precízne daný systém. Predpokladáme, že systém je tvorený kolekciou procesov a každý proces sa skladá zo sekvencie udalostí. Príkladom udalosti je napríklad odoslanie alebo prijatie správy procesom. Udalosti procesu sú uložené v postupnosti, a to takým spôsobom, že udalosť a je uložená skôr ako udalosť b, ak a sa stala pred b. Jeden proces je teda možné definovať ako postupnosť udalostí s prioritným úplným usporiadaním. Relácia *happened-before* na postupnosti udalostí v systéme, označená ako "$\rightarrow$", je najmenšia relácia splňujúca nasledujúce 3 podmienky:

- Ak a a b sú udalosti v rovnakom procese, a a predchádza b, potom $a \rightarrow b$.
- Ak a je odoslanie správy jedným procesom a b je prijatie tej istej správy iným procesom, potom $a \rightarrow b$. Správa nemôže byť prijatá pred ani v rovnaký čas ako je odoslaná, keďže na jej príchod k prijímateľovi je potrebný konečný, nenulový čas.
- Ak $a \rightarrow b$ a $b \rightarrow c$, potom $a \rightarrow c$ (tranzitivita). Dve nezávislé udalosti (v rôznych procesoch, ktoré si nevymieňajú správy) a a b sa považujú za súbežné, ak $a \nrightarrow b$ a $b \nrightarrow a$. V jednoduchosti to znamená, že nič sa nedá ani nie je potrebné povedať o tom, kedy udalosti nastali alebo ktorá nastala skôr.

Keďže $a \rightarrow a$ nemá zmysel (systémy, v ktorých sa udalosť môže udiať sama pred sebou, nemajú význam), relácia *happened-before* je nereflexívna relácia čiastočného usporiadania. [7]

2.2 Lamportove logické hodiny

Logické hodiny sú funkcia, ktorá pre danú udalosť v procese zobrazí logický čas odpovedajúci tejto udalosti. Nejedná sa nutne o čas odpovedajúci reálnemu času, čo je v takomto systéme dostačujúce. Tieto hodiny sú vlastne len spôsob priradenia čísla udalosti, pričom toto číslo je považované za čas, v ktorom daná udalosť nastala. Každý proces P_i má funkciu C_i, priraďujúcu číslo $C_i(a)$ akejkoľvek udalosti a v danom procese. Na to, aby systém takýchto hodín pracoval správne, je potrebné usporiadať udalosti podľa poradia, v akom sa stali. Na to je potrebná práve relácia *happened-before*, vysvetlená v predchádzajúcej podkapitole. Podmienkou je, že ak sa udalosť a vyskytne pred udalosťou b, potom a sa musí udiať v skoršom čase ako b. Formálne je možné zapísať túto definíciu nasledovne:

- Pre akékoľvek udalosti a a b: ak $a \rightarrow b$, potom $C(a) < C(b)$.

Z definície relácie *happened-before* je zrejmé, že táto podmienka bude splnená, ak sú splnené nasledujúce dve podmienky:

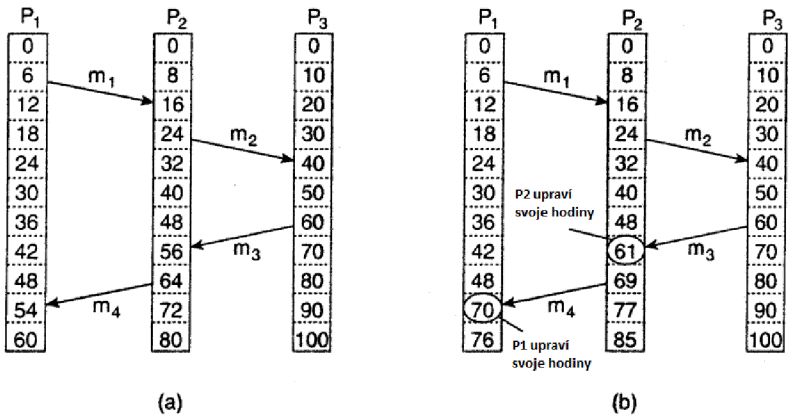
P1. Ak a a b sú udalosti v procese P_i, a a predchádza b, potom $C_i(a) < C_i(b)$.

P2. Ak a je odoslanie správy procesom P_i a b je prijatie tej istej správy procesom P_j, potom $C_i(a) < C_j(b)$.

Pre implementáciu dobre pracujúceho systému logických hodín je potrebné zaistiť platnosť týchto dvoch podmienok. Podmienka **P1** bude implementovaná tak, že každý proces P_i zvýši svoje C_i o 1 medzi akýmikoľvek za sebou idúcimi udalosťami. Na splnenie podmienky **P2** musí každá správa m obsahovať časovú pečiatku T_m, ktorá je rovná času zaslania správy. Ak udalosť a posiela správu m v procese P_i, potom $T_m = C_i(a)$. Pri prijatí správy m potom nastaví proces P_j svoje hodiny C_j na väčšiu z hodnôt: o 1 zvýšený svoj aktuálny čas alebo o 1 zvýšenú časovú pečiatku T_m z prijatej správy: $C_j = \max(C_i + 1, T_m + 1)$. Príklad funkčnosti týchto logických hodín je zobrazený v obrázku (2.2).

V niektorých situáciách môže byť problémom, keď sa 2 udalosti odohrajú v úplne rovnakom čase. Kvôli tomu je pre dosiahnutie úplného usporiadania pridané ľubovoľné označenie procesov - väčšinou podľa ID. Ak by 2 procesy označili správu rovnakým logickým časom, procesu s nižším ID bude daná prednosť. ID bude teda označovať prioritu procesov.

Byť schopný úplne usporiadať udalosti môže byť pri implementácii distribuovaného systému veľmi užitočné. Dôvodom implementácie správneho systému logických hodín je vlastne práve dosiahnutie úplného usporiadania. Jedným z využití tohto úplného usporiadania udalostí je riešenie problému vzájomného vylúčenia. [7]



Obr. 2.2: Logické hodiny. Každý proces má svoje vlastné hodiny bežiace svojou vlastnou konštantnou rýchlosťou. Táto rýchlosť je ale u každého iná z dôvodu rozdielov v kryštáli. Na obrázku (a) je možné vidieť 3 takéto procesy, posielajúce si medzi sebou správy. Časy v správach m1 a m2 sú logicky možné, správa je prijatá neskôr ako bola odoslaná. Správa m3 je odoslaná procesom P3 v čase 60, príde však podľa hodín procesu P2 v čase 56. Podobný problém je možné pozorovať pri správe m4. Tieto hodnoty sú jednoznačne nemožné. Obrázok (b) ukazuje, ako Lamportove logické hodiny upravia tieto hodnoty. Keďže P1 odoslal správu m3 v čase 60, správa musí prísť v čase 61 alebo neskôr. Keď príde správa a hodiny ukazujú hodnotu skoršiu ako čas odoslania správy, prijímateľ nastaví svoje hodiny o 1 vyššie ako čas odoslania správy. Správa m3 teda príde v čase 61 a podobne správa m4 príde v čase 70. Prevzaté a upravené z [14].

Kapitola 3

Mechanizmy vzájomného vylúčenia v distribuovaných systémoch

Jeden z najzásadnejších synchronizačných problémov v distribuovaných systémoch je zaistenie vzájomne výlučného prístupu žiadajúcich procesov ku kritickému zdroju [3]. Vzájomné vylúčenie v distribuovanom systéme teda vyjadruje, že iba jeden proces má povolené vstúpiť do kritickej sekcie (ďalej len KS) v ktoromkoľvek danom čase [6]. Súčasný prístup by mohol danú sekciu narušiť či spraviť ju nekonzistentnou [14]. Keďže nie je možné využiť zdieľané premenné (semafory či monitory) - vysvetlené v sekcii 2 v časti o vlastnostiach distribuovaných systémov, predávanie správ je jediným prostriedkom na implementáciu distribuovaného vzájomného vylúčenia. Rozhodnutie, ktorému procesu bude následne povolené vstúpiť do KS, bude vykonané na základe odoslania správy, z ktorej každý proces zistí stav všetkých ostatných procesov konzistentným spôsobom. Dizajn algoritmov distribuovaného vzájomného vylúčenia je komplexný, pretože tieto algoritmy musia počítať s nepredvídateľnými omeškaniami správ a tiež nekompletnou znalosťou stavu systému. [6]

Nasledujúca časť bola prevzatá z [6].
Existujú 3 hlavné prístupy na implementáciu distribuovaného vzájomného vylúčenia:

- **Token-based (zdieľanie tokenov)**
- **Non-token-based (bez zdieľania tokenov)**
- **Quorum-based (vytváranie kvór)**

Token-based prístup - medzi procesmi (stranami distribuovaného systému) je zdieľaný unikátny token (tiež nazývaný PRIVILEGE správa, teda správa udeľujúca privilégium). Strane je povolené vstúpiť do KS, iba ak aktuálne vlastní token, a bude ho držať, pokiaľ vykonáva svoju KS. Vzájomné vylúčenie je zaručené, pretože takýto token existuje v systéme len jeden. Na zoradenie požiadaviek pre vstup do KS je používané poradové číslo. Algoritmy založené na tomto prístupe sa líšia zásadne v spôsobe, akým strana vykonáva hľadanie tokenu. Príkladom tohto prístupu je algoritmus Suzuki-Kasami.

Non-token-based prístup - medzi stranami sú vymenené 2 alebo viac za sebou idúce kolá správ. Strana potom vstúpi do KS iba vtedy, ak bola splnená podmienka definovaná jej lokálnymi premennými. Vzájomné vylúčenie je zaručené, pretože podmienka bude splnená v danom čase iba v jednej zo strán. Na zoradenie požiadaviek pre vstup do KS sú používané časové značky. Všetky algoritmy využívajúce tento prístup udržiavajú svoje logické hodiny. Príkladom je Lamportov algoritmus a algoritmus Ricart-Agrawala.

Quorum-based prístup - namiesto žiadosti o vstup do KS od všetkých ostatných strán, každá strana žiada povolenie iba od podmnožiny strán. Táto podmnožina sa nazýva kvórum. Akékoľvek dve kvóra pritom musia mať spoločnú aspoň jednu stranu. Keď potom 2 strany súčasne žiadajú o vstup do KS, táto spoločná strana v ich kvórach obdrží obe žiadosti a je zodpovedná za zaručenie vzájomného vylúčenia. Príkladom je Maekawov algoritmus.

3.1 Požiadavky na algoritmy vzájomného vylúčenia

Algoritmus zaručujúci vzájomné vylúčenie procesov by mal splňovať nasledujúce podmienky: [6]

- **Bezpečnosť**
  Táto podmienka zaručuje, že v akomkoľvek čase môže práve jeden proces vykonávať prácu v kritickej sekcii.
- **Živosť**
  Táto podmienka zaručuje neprítomnosť uviaznutia a vyhladovenia. Dva alebo viac procesov by nemalo nekonečne dlho očakávať správy, ktoré nikdy nedorazia. Zároveň by tiež jeden proces nemal neurčito dlho čakať na prístup do kritickej sekcie, zatiaľ čo ostatné procesy získavajú do KS opakovaný prístup. Z toho celého vyplýva, že každá požadujúca strana v systéme by mala dostať možnosť vykonať prácu v KS v konečnom čase.
- **Férovosť**
  V tomto kontexte férovosť znamená, že každý proces by mal dostať spravodlivú šancu na vykonávanie KS. V algoritmoch vzájomného vylúčenia v distribuovaných systémoch vlastnosť férovosti vo všeobecnosti vyjadruje, že žiadosti o prístup do KS by mali byť vyhovené v takom poradí, v akom prišli do systému, pričom čas týchto žiadostí je určený logickými hodinami.

Zatiaľ čo podmienka bezpečnosti je absolútne nevyhnutná, podmienky živosti a férovosti sú považované za dôležité v algoritmoch zaručujúcich vzájomné vylúčenie procesov.

3.2 Lamportov algoritmus

Leslie Lamport, americký matematik a informatik, predstavil algoritmus vzájomného vylúčenia v distribuovaných systémoch ako ilustráciu jeho schémy synchronizácie hodín v článku [7].

Predpokladáme systém zložený z fixného počtu procesov, ktoré zdieľajú jeden zdroj. Iba jeden proces môže využívať v danom čase daný zdroj. Pre zabránenie konfliktu musia byť preto všetky procesy synchronizované.[7] Žiadosti o vstup do KS sú vykonávané podľa vzostupného poradia ich časových pečiatok a čas je aktualizovaný pomocou logických hodín, vysvetlených v podkapitole 2.2. Každý proces udržiava svoj zoznam žiadostí (REQUEST QUEUE), ktorý nikdy nie je videný ostatnými procesmi a ktorý obsahuje žiadosti a vstup do KS prijaté od ostatných procesov a zoradené podľa ich časových pečiatok.[6]

Cieľom algoritmu je zaistenie výlučného prístupu k zdieľanému zdroju procesu pri splnení nasledujúcich podmienok: [7]

1. Proces, ktorému bol pridelený prístup k zdroju, ho musí uvoľniť predtým, ako môže byť pridelený ďalšiemu procesu.

2. Požiadavky procesov na prístup do KS musia byť prideľované v poradí, v ktorom boli podľa logických hodín vyžiadané.

3. Ak každý proces uvoľní zdroj v konečnom čase, potom bude v konečnom čase splnená každá žiadosť o prístup.

Samotný algoritmus je definovaný nasledujúcimi pravidlami: [7]

1. Keď chce proces P_i vstúpiť do KS, odošle správu REQUEST(ts_i, i) všetkým ostatným procesom, čaká na ich reakcie a uloží svoju žiadosť do svojho zoznamu žiadostí (REQUEST QUEUE). ts_i pritom označuje časovú pečiatku odoslania žiadosti.

2. Keď proces P_j obdrží žiadosť o vstup do KS od iného procesu, zaradí ju do svojho zoznamu žiadostí usporiadaného podľa časových pečiatok, alebo v prípade zhodných pečiatok podľa ID žiadateľov (viď podkapitola 2.2). Následne pošle danému procesu odpoveď (REPLY) označenú časovou pečiatkou.

3. Proces P_i vstupuje do KS pokiaľ platia nasledujúce 2 podmienky:

   - P_i obdržal odpoveď s časovou pečiatkou väčšou ako ts_i od všetkých ostatných procesov
   - jeho vlastná žiadosť je na začiatku jeho zoznamu žiadostí

4. V okamihu, keď proces P_i vystupuje z KS, odstráni svoju žiadosť zo začiatku svojho zoznamu žiadostí a odošle správu o uvoľnení KS (RELEASE) označenú časovou pečiatkou všetkým ostatným procesom.

5. Keď proces P_j obdrží správu RELEASE od procesu P_i, odstráni žiadosť procesu P_i zo svojho zoznamu žiadostí. V tomto momente sa jeho vlastná žiadosť môže dostať na začiatok zoznamu, čo mu môže umožniť vstup do KS.

Lamportov algoritmus predpokladá, že posielanie správ je spoľahlivé, teda že žiadna správa nebude stratená. Možným problémom a nevýhodou je nespoľahlivosť, a to v prípade zlyhania jedného z procesov, čo zastaví pokračovanie celého systému. [14]

Pre každý vstup do KS, Lamportov algoritmus vyžaduje $(n-1)$ žiadostí, $(n-1)$ odpovedí a $(n-1)$ uvoľnení. Celkovo je potrebných $3(n-1)$ správ na jeden vstup ľubovoľného procesu do KS.

Možnou optimalizáciou Lamportovho algoritmu je algoritmus Ricart-Agrawala, ktorý zlučuje správy odpovede a uvoľnenia, vďaka čomu bude algoritmus vyžadovať iba $2(n-1)$ vymenených správ na jeden vstup do KS. Funguje takým spôsobom, že ak proces dostal žiadosť o vstup do KS a sám je pritom žiadateľom, počká s odoslaním odpovede v prípade, ak vie, že jeho žiadosť bude mať pred prijatou žiadosťou prednosť (časová pečiatka jeho žiadosti je menšia, alebo v prípade rovnosti pečiatok má jeho ID vyššiu prioritu). [11]

3.3 Maekawov algoritmus

Algoritmy využívajúce vytváranie kvór na zaistenie vzájomného vylúčenia procesov v distribuovanom systéme sa mierne líšia od ostatných prístupov. Najväčším rozdielom je fakt, že strana pri snahe o prístup do KS nežiada povolenie od všetkých ostatných procesov v systéme, ale iba od podmnožiny z nich. Táto podmnožina sa nazýva kvórum. Pre každé dve podmnožiny pritom platí, že majú spoločný aspoň jeden proces, ktorý bude zodpovedný za vyriešenie konfliktu medzi nimi. Druhý rozdiel spočíva v tom, že strana nemôže poslať viac ako jednu správu odpovede (REPLY správu) súčasne. Správu posiela až po tom, ako už dostala správu uvoľnenia (RELEASE) odpovedajúcu na poslednú REPLY správu. Algoritmy využívajúce kvóra na žiadanie o prístup do KS takýmto spôsobom výrazne znižujú počet a komplexnosť posielaných správ. [6]

Dôležitými pojmami na vysvetlenie sú kotérie a kvóra. Kotérium C je definované ako *"množina množín, v ktorom každá množina $g \in C$ sa nazýva kvórum [6]."* Kvórum v kotérii má nasledujúce vlastnosti:

- **Prienik**
  Pre každé kvórum $g, h \in C, g \cap h \neq \emptyset$. Každé dve kvóra teda musia obsahovať minimálne jeden spoločný element. Množiny $\{1, 2, 3\}$ a $\{4, 5, 6\}$ nemôžu byť kvórami v kotérii, pretože nenájdeme ich prienik.

- **Minimalizmus**
  Nemali by existovať žiadne kvóra g, h v kotérii C také, že $g \supseteq h$. Napríklad množiny $\{2, 3, 4\}$ a $\{3, 4\}$ nemôžu byť kvóra v jednej kotérii, pretože druhá množina je podmnožinou prvej.

Každý proces v systéme teda bude mať podmnožinu procesov (kvórum), ktorá bude obsahovať tie procesy, od ktorých bude žiadať povolenie na vstup do KS.

Vytváranie kvór podľa Maekawu

Maekawov algoritmus bol prvým algoritmom využívajúcim vytváranie kvór na zaručenie vzájomného vylúčenia. Kvóra sú v ňom vytvárané tak, aby spĺňali nasledujúce podmienky: [6]

1. Pre každé 2 procesy P_i a P_j, kde $i \neq j$, $1 \leqslant i, j \leqslant N$, kde N je počet procesov v systéme, platí, že kvóra R_i a R_j musia zdieľať nejaký proces ($R_i \cap R_j \neq \emptyset$). Keďže je vždy aspoň jeden spoločný proces medzi každou dvojicou kvór, každá dvojica procesov má vďaka tomu spoločný proces, ktorý bude zodpovedný za zaručenie vzájomného vylúčenia medzi týmito dvomi procesmi. (angl. Intersection property)

2. Každý proces P_i, $1 \leqslant i \leqslant N$, sa nachádza vo svojom kvóre R_i. (angl. Inclusion property)

3. Pre každý proces P_i, $1 \leqslant i \leqslant N$ platí, že $|R_i| = K$. To znamená, že veľkosť K všetkých kvór je rovnaká. Každý proces preto posiela a dostáva pri žiadostiach o vstup do KS rovnaký počet správ, a teda všetky procesy musia vykonať rovnaké množstvo práce na zaručenie vzájomného vylúčenia. (angl. Equal effort property)

4. Každý proces P_i je prítomný celkovo K-krát vo všetkých kvórach. Z toho vyplýva, že od každého procesu bude pýtané povolenie rovnakým počtom procesov, a teda

všetky procesy majú rovnakú zodpovednosť pri zaručovaní povolení pre ostatné procesy. (angl. Equal responsibility property)

Pre správnosť a fungovanie algoritmu musia byť zaručené podmienky 1 a 2. Podmienky 3 a 4 predstavujú iné požadované, ale nepovinné vlastnosti kvór. Podľa podmienky 4 je každý proces prítomný K-krát vo všetkých podmnožinách. Z podmienky 2 je možné vidieť, že každý člen R_i sa teda môže nachádzať $(K-1)$ krát v zoznamoch iných procesov. Preto je maximálny počet podmnožín spĺňujúcich podmienku 1 daný výrazom $N = (K-1)K+1$, čo je možné prepísať na tvar $N = K(K-1)+1$. Problém nájdenia množín R_i, ktoré spĺňajú tieto podmienky, je ekvivalentný s hľadaním konečnej projektívnej roviny N bodov. Je známe, že existuje konečná projektívna rovina rádu k, pokiaľ k je mocninou p^m prvočísla p. Táto rovina má $k(k+1)+1$ bodov. V našom prípade teda množiny R_i existujú, ak $(K-1)$ je mocninou prvočísla. Pre iné hodnoty K musíme upustiť z podmienok 3 a 4. [9] Príklad takýchto množín pre $K = 3$ a $N = 7$:

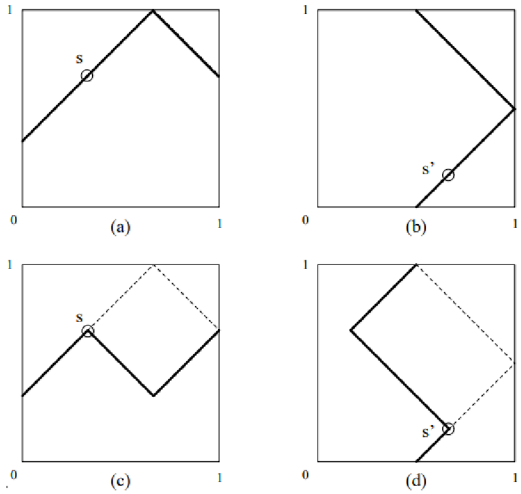
$R_1 = \{1, 2, 3\}$
$R_2 = \{2, 4, 6\}$
$R_3 = \{3, 5, 6\}$
$R_4 = \{1, 4, 5\}$
$R_5 = \{2, 5, 7\}$
$R_6 = \{1, 6, 7\}$
$R_7 = \{3, 4, 7\}$

Ďalšie spôsoby vytvárania kvór

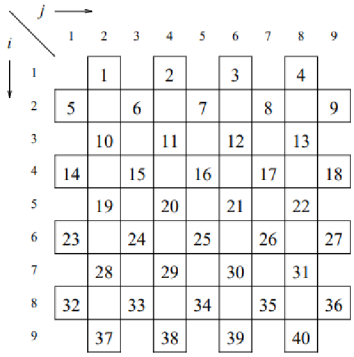
Vytváranie kvór nie je unikátne, existuje viac rôznych spôsobov, ako ich vytvoriť tak, aby spĺňali požadované podmienky. Jedným z týchto spôsobov je generovanie kvór pomocou štvorcovej mriežky. Každý proces v systéme je reprezentovaný jedným bodom v mriežke. Maekawa predstavil myšlienku, že pre daný proces v mriežke by jeho kvórum tvorili všetky procesy na horizontálnej a vertikálnej čiare prechádzajúcej daným procesom. Inou možnosťou je využívanie čiar so sklonom $+-1$. Takýto spôsob sa nazýva biliardová metóda. Jej princíp spočíva v štuchnutí do biliardovej gule takým spôsobom, že začne na hranici $x = 0$ alebo $y = 0$, bude smerovať k danému bodu(procesu) v mriežke a paralelne k čiare $y = x$. Táto cesta bude pre daný proces tvoriť jeho kvórum. Dve takéto cesty je možné vidieť na obrázku 3.1 (a) a (b). Takýmto spôsobom by však dva body ležiace na rovnakej čiare definovali rovnaké cesty. Kvôli tomu je nutné zaviesť zalomené biliardové cesty. Tie sú vytvorené zalomením cesty v bode o 90° a potom ďalším zalomením v takom bode, aby cesta skončila v rovnakom bode, ako by skončila pri nezalomenej ceste. Príklady je možné vidieť na obrázku 3.1 (c) a (d).

Pre zaručenie vyhovenia všetkých podmienok na vytváranie kvór je potrebné procesy v mriežke usporiadať špeciálnym spôsobom. Tento spôsob je možný vidieť na obrázku 3.2. Riadky a stĺpce sú označené i a j. Procesy budú do mriežky vkladané iba na pozície, v ktorých platí, že $i + j$ je nepárne číslo. Vytvorenie kvór v štvorcovej mriežke veľkosti $q * q$ bude možné iba v prípade, že q je nepárne číslo. Príklad vytvorených kvór pomocou biliardovej metódy pre $q = 3$ a $N = 4$:

$R_1 = \{1, 2, 3\}$
$R_2 = \{2, 3, 4\}$
$R_3 = \{1, 3, 4\}$
$R_4 = \{1, 2, 4\}$



Obr. 3.1: Vytváranie kvór biliardovou metódou. Obrázky (a) a (b) predstavujú príklady vytvárania kvór pomocou biliardových ciest. Obrázky (c) a (d) pomocou zalomených biliardových ciest. Prevzaté z [2].



Obr. 3.2: Usporiadanie procesov do mriežky v biliardovej metóde vytvárania kvór, kde veľkosť jednej strany mriežky $q = 9$ a počet všetkých procesov $N = 40$. Prevzaté z [2].

Detailnejšie vysvetlenie všetkých spomenutých metód vytvárania kvór je možné nájsť v článku [2].

Princíp algoritmu

Proces môže udeliť povolenie na vstup do KS, iba ak ešte nepovolil vstup nejakému inému procesu, čo znamená až po prijatí správy uvoľnenia. Vzájomné vylúčenie je teda zaručené. Algoritmus vyžaduje aby boli správy doručované v poradí ich odoslania medzi dvojicou procesov.

Každý proces v systéme vykonáva nasledujúci algoritmus: [6]

- **Žiadosť o vstup do KS**

  a Ak chce proces P_i vstúpiť do KS, zašle správu REQUEST všetkým ostatným procesom v jeho kvóre R_i.

b Keď proces P_j obdrží správu REQUEST, odošle späť procesu P_i správu REPLY, ak ešte neposlal túto správu žiadnemu inému procesu od príchodu poslednej RELEASE správy. Inak si uloží REQUEST do frontu na neskoršie posúdenie.

- **Vykonávanie KS**

  Proces P_i vstupuje do KS iba v tom prípade, že obdržal REPLY správu od všetkých procesov v jeho kvóre R_i.

- **Uvoľnenie KS**

  a Po skončení práce v KS, proces P_i odošle správu RELEASE všetkým procesom z jeho kvóra R_i.

  b Keď proces P_j obdrží správu RELEASE od P_i, zašle správu REPLY nasledujúcemu procesu uloženému v jeho fronte a vymaže ho z nej. Ak je jeho front prázdny, proces si zapamätá, že ešte neodoslal žiadnu REPLY správu od príchodu poslednej RELEASE správy.

Keďže veľkosť každého kvóra je $\sqrt{N}$, na jeden vstup do KS je potrebných $\sqrt{N}$ REQUEST správ, $\sqrt{N}$ REPLY správ a $\sqrt{N}$ RELEASE správ, a teda celkovo $3\sqrt{N}$ správ.

Problém uviaznutia (deadlock)

Najväčším problémom spomenutého algoritmu je možnosť uviaznutia. To môže nastať v prípade, že viac procesov žiada v rovnakom čase o prístup do KS. Keďže procesy nezasielajú správy REQUEST v žiadnom určenom poradí a je možné akékoľvek oneskorenie správ, je možný nasledujúci prípad. [6]

Majme 6 procesov v systéme. Kvóra 3 procesov, P_1, P_2 a P_3, sú nasledovné:

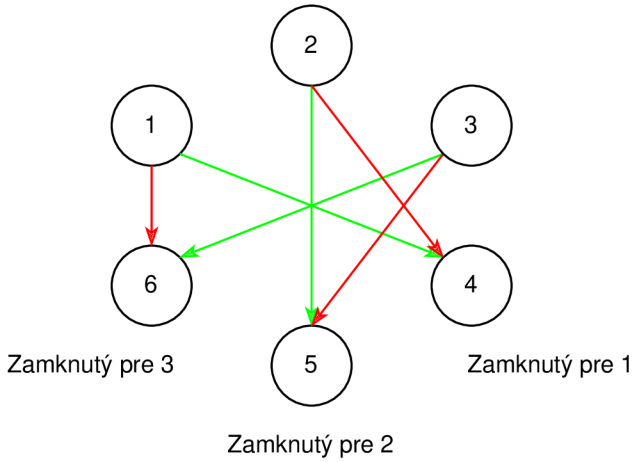
$$R_1 = \{1, 4, 6\}$$
$$R_2 = \{2, 4, 5\}$$
$$R_3 = \{3, 5, 6\}$$

Všetky tieto procesy súčasne žiadajú o prístup do KS. Za zaručenie vzájomného vylúčenia medzi procesmi 1 a 2 je zodpovedný spoločný proces medzi ich kvórami - proces 4. Pre procesy 2 a 3 je to proces 5 a pre procesy 1 a 3 proces 6. V tomto prípade môže nastať situácia na obrázku 3.3. Proces 1 dostal odpoveď od procesu 4, čaká však na proces 6. Proces 2 uzamkol proces 5, čaká ale na proces 4. Proces 3 zas uzamkol proces 6, čaká ale na proces 5. Tým vzniklo kruhové čakanie, ktoré spôsobí uviaznutie všetkých troch procesov a teda nemožnosť pokračovať ďalej vo vykonávaní algoritmu.

Pre vyhnutie sa možným uviaznutiam sú k algoritmu pridané 3 nové typy správ: FAILED, INQUIRE a YIELD. Procesom sa určí priorita na základe časových pečiatok (v prípade zhody podľa ID, tak ako v Lamportovom algoritme - časť 3.2). Algoritmus bude upravený nasledovným spôsobom: [6]

- Keď proces P_j obdrží správu REQUEST od P_i, zistí, akému procesu poslal REPLY správu. Ak to bol proces s vyššou prioritou, pošle správu FAILED procesu P_i a zaradí ho do frontu. Ak však odoslal REPLY správu procesu P_k s nižšou prioritou, odošle INQUIRE správu procesu P_k. Pýta sa ho tým, či sa mu podarilo vstúpiť do KS.

- Keď proces P_k obdrží INQUIRE správu od P_j, odpovie mu správou YIELD v tom prípade, že dostal správu FAILED od niektorého z procesov zo svojho kvóra alebo už predtým poslal inému procesu správu YIELD a nedostal ešte od neho novú REPLY správu.
- Keď proces P_j dostane správu YIELD od procesu P_k, znamená to, že proces P_k nevstúpil do KS a danou správou dal najavo, že umožňuje vstúpiť do nej niekomu inému. Proces P_j si potom uloží REQUEST od P_k do svojho prioritného frontu a pošle REPLY správu inému procesu zo začiatku jeho frontu.



Obr. 3.3: Uviaznutie v Maekawovom algoritme. Procesy 1, 2 a 3 súčasne žiadajú o prístup do KS. Zelené šípky naznačujú žiadosti, na ktoré bola odoslaná odpoveď, a teda žiadaný proces sa uzamkol pre daný žiadajúci proces. Červené šípky naopak znázorňujú neúspešné žiadosti bez odpovede. Každý proces dostal odpoveď iba od jedného z procesov z kvóra a čaká na ďalšiu odpoveď, ktorá však kvôli kruhovému čakaniu nepríde a vznikne uviaznutie (deadlock). Inšpirované z [9].

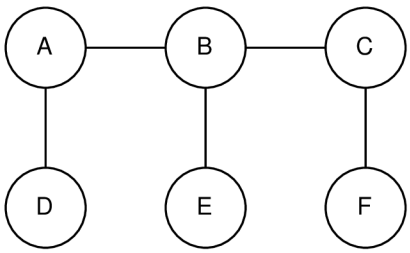
Maekawov algoritmus týmto spôsobom zabraňuje vzniku uviaznutia. Vyžaduje to však väčšie množstvo posielaných správ, a to aj v prípade, že v danej chvíli uviaznutie nehrozilo. Pre každý vstup do KS je s týmto rozšírením potrebných $5\sqrt{N}$ správ.

3.4 Raymondov algoritmus

Ďalším algoritmom pre zaistenie vzájomného vylúčenia v distribuovanom systéme v sieti s N procesmi komunikujúcimi správami je algoritmus založený na stromovej štruktúre predstavený Kerry Raymondom v originálnom článku [10]. Tento algoritmus využíva stromovú štruktúru usporiadania procesov a počet vymenených správ medzi týmito procesmi pre jednu žiadosť o vstup do KS záleží na topológii tohto stromu. Algoritmus predpokladá, že sieť garantuje doručenie správ. Čas ani poradie prijatých správ nie je možné predpovedať. Každý z procesov komunikuje iba s jeho susednými procesmi v stromovej štruktúre a udržiava si informácie týkajúce sa iba práve týchto susedov. [10]

Sieť procesov v tomto algoritme je možné si predstaviť ako graf, stromovú štruktúru bez koreňového uzla, v ktorom uzly sú procesmi v danom systéme a spojnice medzi uzlami predstavujú trajektórie, po ktorých sú posielané správy medzi procesmi. Uzly v strome sú

usporiadané tak, aby počet spojníc v tomto grafe bol čo najmenší, a teda bol potrebný čo najmenší počet posielaných správ. [6] Nie je potrebné, aby všetky uzly poznali strom ako celok. Je postačujúce, že každý uzol vie o svojich priamych susedoch. Takúto stromovú štruktúru je možné vidieť na obrázku 3.4. V tomto príklade uzol A vie, že má dvoch priamych susedov B a D a s nimi si bude posielať správy. O lokácii, ako ani existencii ostatných uzlov nevie a ani vedieť nepotrebuje.



Obr. 3.4: Stromová štruktúra procesov v systéme bez koreňového uzlu. Inšpirované z [6].

Koncept algoritmu je podobný princípu posielania tokenov v algoritme Suzuki-Kasami, ktorý bude vysvetlený v časti 3.5. Medzi uzlami je posielaná správa PRIVILEGE, ktorá signalizuje, ktorý proces má právo vstúpiť do KS. Iba jeden uzol v danom čase vlastní toto právo, okrem prípadu, kedy je v stave posielania z jedného uzlu do druhého. Keď práve žiadny proces nežiada o vstup do KS, posledný proces držiaci toto právo si ho ponechá. [10]

Premenná HOLDER

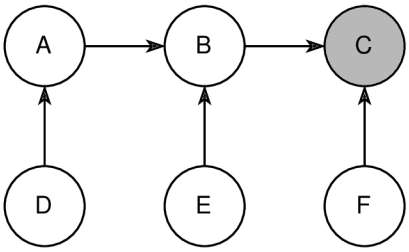
Každý uzol si udržuje premennú HOLDER, ktorá určuje polohu PRIVILEGE relatívne k uzlu samotnému. Uzol má nastavenú svoju premennú HOLDER na ten uzol, o ktorom si myslí, že aktuálne vlastní PRIVILEGE alebo aspoň k nemu vedie. [6] V prípade grafu 3.4, ak by uzol C aktuálne vlastnil PRIVILEGE, premenné HOLDER všetkých uzlov budú vyzerať nasledovne:

$HOLDER_A$ = B,
$HOLDER_B$ = C,
$HOLDER_C$ = C,
$HOLDER_D$ = A,
$HOLDER_E$ = B,
$HOLDER_F$ = C.

Tieto premenné všetkých uzlov potom vytvárajú smerové cesty z každého uzlu do uzlu, ktorý právo vlastní. Reprezentáciu tohto stavu je možné vidieť na orientovanom grafe na obrázku 3.5.

V prípade, že by teraz napríklad uzol D, ktorý nevlastní právo, chcel vstúpiť do KS, pošle správu REQUEST uzlu, ktorý má uložený vo svojej HOLDER premennej, teda procesu A. Proces A túto správu prepošle uzlu ktorý má on vo svojej HOLDER premennej, teda B, a tým istým spôsobom zas pošle uzol B správu procesu C, ktorý právo vlastní. Prebieha teda séria odosielaných správ v smere od žiadajúceho uzlu A k uzlu aktuálne vlastniacemu právo C.

Keď uzol C ukončí svoju prácu v KS a už viac právo nepotrebuje, odošle PRIVILEGE správu susedovi B, ktorý o právo žiadal. Zároveň nastaví svoju premennú HOLDER na B.

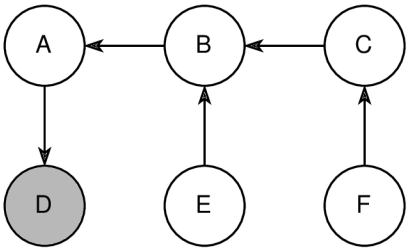


Obr. 3.5: Stromová štruktúra procesov ako orientovaný graf, kde všetky uzly ukazujú smerom k uzlu C, ktorý aktuálne vlastní právo (PRIVILEGE). Inšpirované z [6].

Keďže B nežiadal právo pre seba, ale za uzol A, pošle PRIVILEGE správu uzlu A a nastaví svoju premennú HOLDER na A. Uzol A zas odošle správu uzlu D a obdobne nastaví svoju premennú HOLDER na D. Keďže D je uzol, ktorý žiadal o vstup do KS a práve získal PRIVILEGE, môže vstúpiť do KS. Po vykonaní týchto krokov budú HOLDER premenné všetkých uzlov vyzerať nasledovne:

$HOLDER_A = \text{D},$
$HOLDER_B = \text{A},$
$HOLDER_C = \text{B},$
$HOLDER_D = \text{D},$
$HOLDER_E = \text{B},$
$HOLDER_F = \text{C}.$

Orientácie niektorých hrán sa zmenia tak, ako vidieť na obrázku 3.6.



Obr. 3.6: Zmenená stromová štruktúra procesov zo stavu na obrázku 3.5. Po odoslaní PRIVILEGE správy z uzlu C do žiadajúceho uzlu D všetky uzly ukazujú smerom k uzlu D, ktorý teraz vlastní právo (PRIVILEGE). Inšpirované z [6].

Takýmto spôsobom je zaručené, že v akomkoľvek stave systému (okrem stavu posielania PRIVILEGE správy medzi dvomi uzlami) premenné HOLDER kolektívne zaručujú smer z každého uzlu do uzlu aktuálne vlastniaceho právo. [6]

Využívané dátové štruktúry

Na implementáciu opísaného chovania je potrebné, aby si každý uzol udržiaval niekoľko informácií. Tieto informácie sú uchovávané pomocou nasledujúcich premenných: [10]

- **HOLDER**
  Premenná indikujúca polohu uzlu s PRIVILEGE relatívne k danému uzlu. Obsahuje

názov jedného zo susedných uzlov alebo samého seba, v prípade že aktuálne vlastní právo. Táto premenná bola bližšie vysvetlená v predchádzajúcej časti.

- **USING**
  Premenná indikujúca, či daný uzol aktuálne vykonáva KS. Premenná typu boolean možnosťami true ak vykonáva KS, false ak nie.

- **REQUEST QUEUE**
  FIFO front (prvý dnu, prvý von) obsahujúci mená tých susedných uzlov, ktorí žiadajú o PRIVILEGE ale ešte im nebol odoslaný. Môže obsahovať aj meno samého seba, a to v tom prípade, keď daný uzol žiada PRIVILEGE pre seba. Maximálna veľkosť frontu je počet susedov daného uzlu + 1 (pre samého seba).

- **ASKED**
  Premenná typu boolean, nastavená na true, pokiaľ daný uzol poslal REQUEST správu uzlu, ktorý je aktuálne v jeho HOLDER premennej, inak false. Táto premenná predchádza odosielaniu nepotrebných REQUEST správ a zároveň zaisťuje, že REQUEST QUEUE neobsahuje duplikáty.

Odosielanie správy PRIVILEGE

Uzol vlastniaci PRIVILEGE odošle PRIVILEGE správu inému uzlu, pokiaľ je splnené nasledovné: [10]

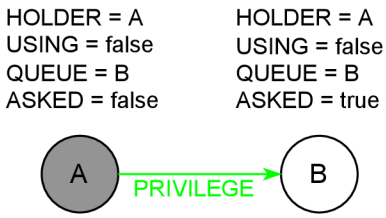
- uzol vlastní PRIVILEGE ale nepoužíva ho (nenachádza sa v KS, jeho USING premenná je false)
- jeho REQUEST QUEUE nie je prázdna
- uzol na začiatku jeho REQUEST QUEUE nie je on samotný, čo znamená, že najstaršia správa REQUEST musela prísť od iného uzlu

Situácia, kedy je daný uzol na začiatku svojej REQUEST QUEUE, sa môže zdať nemožná, avšak môže sa tak stať okamžite po tom, ako tento uzol dostal PRIVILEGE správu. Vtedy uzol odstráni samého seba zo svojej REQUEST QUEUE, vstúpi do KS a nastaví svoju premennú USING na true. Ak je iný uzol na začiatku jeho REQUEST QUEUE, odstráni ho z nej, odošle správu PRIVILEGE tomuto uzlu a nastaví svoju HOLDER premennú na tento uzol (viď obrázok 3.7) Zároveň tiež nastaví svoju premennú ASKED na false, keďže uzol aktuálne vlastniaci PRIVILEGE by ho neposlal inému uzlu, pokiaľ by ho sám aktuálne potreboval.

Odosielanie správy REQUEST

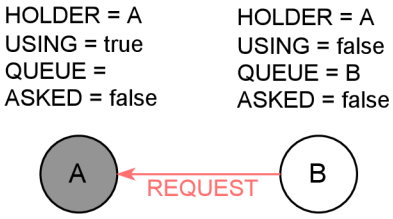
Uzol, ktorý nevlastní PRIVILEGE, odošle správu REQUEST uzlu v jeho premennej HOLDER, pokiaľ: [6]

- nevlastní PRIVILEGE
- jeho REQUEST QUEUE nie je prázdna, čo znamená, že žiada o PRIVILEGE buď pre seba, alebo za jedného z jeho priamych susedov
- ešte neposlal REQUEST správu (jeho premenná ASKED je false)



Obr. 3.7: Odosielanie PRIVILEGE správy. Stav, v ktorom uzol A vlastní PRIVILEGE a nepoužíva ho (USING=false). Jeho REQUEST QUEUE nie je prázdna, ale obsahuje uzol B žiadajúci o PRIVILEGE. V tejto chvíli by teda uzol A odoslal PRIVILEGE správu uzlu B, odstránil B zo svojej REQUEST QUEUE a nastavil svoju premennú HOLDER na B.

Po odoslaní REQUEST správy nastaví uzol svoju premennú ASKED na true. Odosielanie správy REQUEST neovplyvňuje žiadne iné premenné. Príklad je možné vidieť na obrázku 3.8. Premenná ASKED daného uzlu bude true, pokiaľ odoslal REQUEST správu a ešte nedostal odpoveď (v tomto prípade sa tento uzol nachádza v REQUEST QUEUE uzlu, ktorý je v jeho HOLDER premennej, alebo tam bude po príchode REQUEST správy). Inak bude premenná false. Uzol neposiela žiadne REQUEST správy, pokiaľ je jeho premenná ASKED true. Týmto spôsobom zaisťuje premenná ASKED, že nebudú poslané žiadne duplicitné a nepotrebné REQUEST správy a tiež že REQUEST QUEUE jeho susedného uzlu nebude obsahovať duplicitné zápisy žiadajúceho uzlu. Vďaka tomu je veľkosť REQUEST QUEUE akéhokoľvek uzlu ohraničená a nehrozí jej zahltenie ani pri veľkom zaťažení.



Obr. 3.8: Odosielanie REQUEST správy. Stav, v ktorom uzol B žiada o vstup do KS. Keďže aktuálne nevlastní PRIVILEGE, jeho REQUEST QUEUE nie je prázdna (on sám žiada o PRIVILEGE) a ešte neposlal správu (ASKED = false), môže odoslať REQUEST správu. V tejto chvíli by poslal správu REQUEST uzlu A a nastavil by svoju premennú ASKED na true.

Vykonávanie algoritmu

Samotný Raymondov algoritmus pozostáva zo 4 možných udalostí, ktoré sa v uzloch odohrávajú. [6]

- **Uzol chce vstúpiť do KS**
  Ak uzol vlastní PRIVILEGE, môže vstúpiť do KS, viď sekciu o odosielaní správy PRIVILEGE. Ak právo nevlastní, uzol môže poslať správu REQUEST podľa postupu v sekcii o odosielaní správy REQUEST.

- **Uzol dostane správu REQUEST od jedného z jeho susedných uzlov**
  Ak je tento uzol aktuálnym vlastníkom práva, môže ho poslať žiadajúcemu uzlu, viď sekciu o odosielaní správy PRIVILEGE. Ak PRIVILEGE nevlastní, môže preposlať túto žiadosť pomocou postupu v sekcii o odosielaní správy REQUEST.

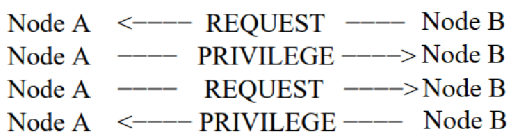
- **Uzol dostane správu PRIVILEGE**
  Prijatie PRIVILEGE správy môže znamenať vstup daného uzlu do KS, alebo preposlanie práva inému uzlu. Po tom, ako je správa preposlaná, môže uzol zažiadať o právo podľa postupu v sekcii o odosielaní správy REQUEST, aby vyhovel zvyšným čakajúcim žiadostiam uloženým v tomto uzle.

- **Uzol vystupuje z KS (ukončuje svoju prácu v KS)**
  Po vystúpení uzlu z KS môže uzol poslať PRIVILEGE inému žiadajúcemu uzlu podľa postupu v sekcii o odosielaní správy PRIVILEGE. Potom môže uzol zažiadať o právo podľa postupu v sekcii o odosielaní správy REQUEST, aby vyhovel zvyšným čakajúcim žiadostiam uloženým v tomto uzle.

Predbiehanie správ

Acyklická stromová štruktúra, v ktorej sú usporiadané uzly v tomto algoritme, predstavuje jednu dôležitú výhodu. Obmedzuje totiž množstvo konfliktných situácií spôsobených rôznymi časmi prenosu správ a predbiehaním správ. Jediný problém môže nastať medzi dvojicou susediacich uzlov, ktorých komunikácia sa riadi akýmsi logickým vzorom (viď obrázok 3.9), a preto v tomto algoritme nie sú potrebné poradové čísla posielaných správ pre zaistenie správneho poradia.

Jediný možný prípad predbehnutia správ sa môže medzi dvomi uzlami A a B odohrať, ak je správa PRIVILEGE, odoslaná z uzlu A do uzlu B, veľmi tesne nasledovaná správou REQUEST z uzlu A do uzlu B (a teda že uzol A odošle právo uzlu B a okamžite ho chce naspäť). Ak by však tieto dve správy prišli v opačnom poradí, a teda že uzol B najprv obdrží správu REQUEST a až potom správu PRIVILEGE od uzlu A, neovplyvní to správne vykonávanie algoritmu. Žiadosť od uzlu A by bola v tomto prípade vložená do frontu REQUEST QUEUE uzlu B. Keďže B nevlastní PRIVILEGE, nie je schopný poslať ho uzlu A. Keď potom uzol B dostane aj správu PRIVILEGE, môže vstúpiť do KS alebo preposlať toto právo žiadajúcemu uzlu, ktorý je na začiatku jeho frontu, čo nebude uzol A, keďže ten bol vložený na koniec tohto frontu. Správanie algoritmu bude teda rovnaké, ako by bolo v prípade správneho poradia prijatých správ. [6]



Obr. 3.9: Logický vzor posielaných správ medzi ľubovoľnými dvomi uzlami A a B. Tento vzor sa počas vykonávania algoritmu niekoľkokrát opakuje. Prevzaté z [6].

Inicializácia algoritmu

Na začiatku algoritmu je jeden uzol vybratý ako privilegovaný. Tento uzol potom pošle INITIALIZE správy všetkým jeho susedným uzlom. Pri prijatí správy INITIALIZE uzol nastaví svoju premennú HOLDER na uzol, ktorý mu INITIALIZE správu odoslal, a následne on samotný pošle túto správu všetkým svojím susedným uzlom. Týmto spôsobom všetky uzly postupne nastavia svoju HOLDER premennú. Po tom čo uzol obdrží správu INITIALIZE, môže začať odosielať žiadosti o PRIVILEGE, hoci ešte celý strom nemusí byť inicializovaný. Všetky uzly si tiež na začiatku nastavia svoje premenné na nasledujúce hodnoty: USING = false, ASKED = false, REQUEST QUEUE - prázdna. [6]

Správnosť algoritmu

Algoritmus zaručuje vzájomné vylúčenie. V akomkoľvek čase môže maximálne jeden uzol vlastniť PRIVILEGE. Keď ho proces vlastní, môže vstúpiť do KS. Keď ho odošle inému uzlu, do KS vstupovať nemôže. V čase odosielania PRIVILEGE správy nemôže žiadny proces vstúpiť do KS, keďže žiadny nevlastní PRIVILEGE. Týmto je zaručené vzájomné vylúčenie procesov. V algoritme nemôže nastať uviaznutie ani vyhladovanie, čo je dokázané v originálnom článku [10] od Kerryho Raymonda.

3.5 Suzuki-kasami vysielací algoritmus

Algoritmus Suzuki-Kasami je založený na zdieľaní tokenov. Na začiatku jeden proces drží token. Ak chce nejaký proces vstúpiť do KS a aktuálne nevlastní token, vyšle správu REQUEST všetkým ostatným procesom. Proces, ktorý drží token, ho po prijatí správy odošle žiadajúcej strane. [6] Ak proces vlastní token, môže do KS vstúpiť opakovane niekoľkokrát, až pokým ho neodošle inému procesu na jeho žiadosť [13]. Keď proces dostane žiadosť o token počas vykonávania KS, odošle token až po ukončení svojej práce v KS. [6]

Základná myšlienka tohto algoritmu je veľmi jednoduchá. Je však potrebné vyriešiť 2 možné problémy: [6]

1. **Ako odlíšiť starú žiadosť od novej**

   Každá poslaná správa môže mať iný čas zdržania. Proces preto môže dostať žiadosť o token od iného procesu po tom, čo už daný proces token obdržal a vykonal KS. Keďže proces nevie zhodnotiť, či už bola požiadavka uspokojená, môže poslať zbytočne token procesu, ktorý ho už nepotrebuje. Síce tým nepoškodí správnosť algoritmu, môže veľmi výrazne znížiť jeho výkonnosť zbytočným odosielaním tokenov a predĺžením čakania ostatných procesov, ktoré skutočne na token čakajú. Nasledujúci mechanizmus tomuto javu predchádza:

   Správa REQUEST procesu P_j má tvar REQUEST(j, n), kde n(n = 1, 2 ...) je poradové číslo označujúce, že proces P_j práve žiada o svoj n-tý vstup do KS. Proces P_i si udržiava vektor hodnôt $RN_i[1...N]$, ktorého veľkosť odpovedá počtu procesov v systéme. $RN_i[j]$ pritom označuje, koľkokrát proces P_j o token žiadal (aké najväčšie poradové číslo od procesu P_j obdržal). Keď proces P_i dostane správu REQUEST(j, n), nastaví $RN_i[j] = max(RN_i[j], n)$. Takýmto spôsobom vie, že správa je stará, ak $RN_i[j] > n$, a tým pádom ju nebude brať do úvahy.

2. **Ako zistiť, ktorý proces má nevybavenú požiadavku o pridelenie tokenu**

   Po tom, ako proces ukončí svoju prácu v KS, musí zistiť, ktorý proces má nevybavenú žiadosť o token, aby mu ho mohol predať. Zistí to takýmto spôsobom:

   Samotný token, ktorý je predávaný medzi procesmi, obsahuje dve položky: front Q žiadajúcich procesov a vektor hodnôt $LN[1...N]$ veľkosti počtu procesov v systéme. $LN[j]$ pritom udáva, koľkokrát bol tento token procesu P_j pridelený (poradové číslo žiadosti, ktorá bola procesu P_j naposledy schválená). Po vykonaní jeho KS proces P_i aktualizuje $LN[i] = RN_i[i]$, čím dá najavo, že jeho žiadosť s poradovým číslom $RN_i[i]$ bola uspokojená. Vektor hodnôt $LN[1...N]$, ktorý token obsahuje, umožňuje procesom týmto spôsobom zistiť, ktoré procesy aktuálne o token žiadajú. Ak totiž pre proces P_i platí, že $RN_i[j] = LN[j] + 1$, znamená to, že proces P_j aktuálne čaká na token (pretože každý proces pri prijatí žiadosti o token od iného procesu nastaví hodnotu na príslušnom indexe svojho vektoru hodnôt na poradové číslo n, ktoré proces v žiadosti zasiela, ako je vysvetlené v predchádzajúcom bode). Po vykonaní KS teda proces P_i skontroluje danú podmienku pre všetky j, aby odhalil všetky procesy čakajúce na token. ID týchto procesov vloží do frontu Q, ak už proces vo fronte nie je. Nakoniec, ak front nie je prázdny, proces odošle token tomu procesu, ktorého ID je na začiatku frontu Q. Ak je Q prázdna, proces P_i si ponechá token, až dokým neobdrží správu REQUEST od iného žiadajúceho procesu [13].

Definícia algoritmu: [6]

- **Žiadosť o vstup do KS**

  a Ak chce proces P_i vstúpiť do KS, ale nevlastní token, zvýši svoje poradové číslo $RN_i[i]$ o 1 a odošle správu REQUEST (i, n) všetkým ostatným procesom v systéme, pričom n značí novú hodnotu $RN_i[i]$.

  b Keď proces P_j obdrží túto správu, nastaví svoje $RN_j[i]$ na max($RN_j[i]$, n). Ak P_j má nevyužívaný token, pošle ho procesu P_i, ak platí $RN_j[i] = LN[i] + 1$.

- **Vykonávanie KS**

  Proces P_i vstupuje do KS po tom, ako obdrží token.

- **Uvoľnenie KS**

  Po skončení práce v KS, proces P_i vykoná nasledovné kroky:

  a Nastaví hodnotu $LN[i]$ na $RN_i[i]$.

  b Pre každý proces P_j, ktorého ID nie je vo fronte Q, vloží jeho ID do frontu, ak $RN_i[j] = LN[j] + 1$.

  c Ak je front Q po vykonaní predchádzajúceho kroku neprázdny, proces P_i vymaže z frontu ID toho procesu, ktorý je na jeho začiatku a pošle tomuto procesu token.

Vzájomné vylúčenie je jednoznačne zaistené, pretože existuje iba jeden token v systéme a proces drží token počas vykonávania KS [6]. V algoritme nenastáva uviaznutie (deadlock) ani vyhladovanie (starvation). Ak proces aktuálne držiaci token žiada o vstup do KS, žiadne správy nie sú potrebné. Ak ho nevlastní, je potrebných N správ na jeden vstup ľubovoľného procesu do KS - $(N-1)$ správ REQUEST + 1 správa zasielajúca token. Algoritmus dosahuje minimálneho možného oneskorenia. [13]

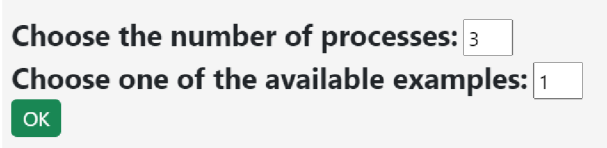
Kapitola 4

Návrh dizajnu aplikácie

Základom tejto práce je navrhnúť užívateľsky prívetivú aplikáciu, ktorá by demonštrovala vysvetlené algoritmy. Aplikácia má primárne slúžiť študentom Fakulty informačných technológií Vysokého učenia technického v Brne (ďalej len FIT VUT), ktorí sa princípy a fungovanie spomenutých algoritmov učia na predmete Paralelní a distribuované algoritmy. Samotné teoretické učenie sa týchto algoritmov môže byť zdĺhavé a ťažké na pochopenie bez vizuálneho zobrazenia. Na internete nie je dostatočné množstvo demonštračných príkladov, ktorí by študentom pomohli lepšie pochopiť funkčnosť týchto algoritmov a umožnili by im sledovať ich na viacerých príkladoch. Aplikácia má teda urýchliť a zefektívniť učenie a pomôcť si naučené princípy lepšie zapamätať.

Prvým bodom činnosti v tejto práci bolo po naštudovaní algoritmov navrhnúť túto aplikáciu. Aplikácia musí mať jednoduchý a rýchlo pochopiteľný dizajn, keďže sa ako najpravdepodobnejší prípad využitia očakáva situácia, že študent si popri učení podľa prezentácie k predmetu aplikáciu zapne a chce aby mu čo najefektívnejšie k učeniu prospela. Zobrazovanie algoritmov by mohlo byť podobné tomu, aké je použité v prezentácii, aby bolo možné sa rýchlo zorientovať a naviazať na už získané znalosti.

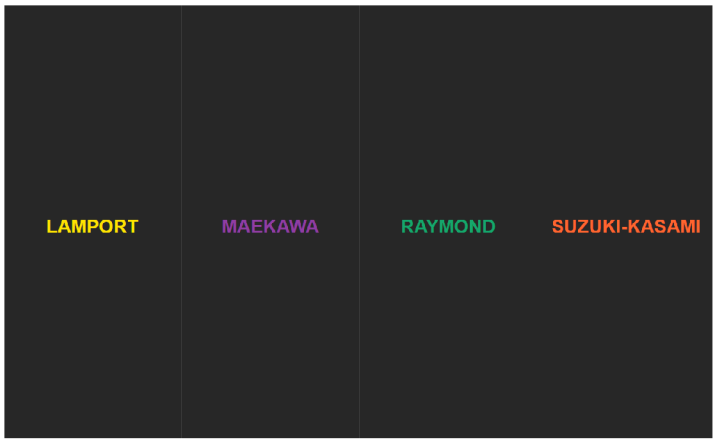
V aplikácii budú demonštrované štyri zo spomenutých algoritmov - Lamportov, Maekawov, Raymondov a Suzuki-Kasami. Každý z týchto algoritmov funguje na trochu inom princípe čo sa týka počtu a typu posielaných správ a komu tieto správy zasiela. Ich cieľ je však rovnaký - zaistiť vzájomné vylúčenie procesov pri snahe o vstup do KS. Na začiatku aplikácie je potrebné jednoduché menu, kde si používateľ vyberie, ktorý algoritmus chce ísť trénovať. Počiatočný návrh[1] tohto menu je možné vidieť na obrázku 4.2. Toto menu bolo nakoniec aj v aplikácii použité.



Obr. 4.1: Výber počtu procesov a čísla príkladu pred samotným spustením algoritmu.

Po výbere algoritmu z menu nasleduje postup podobný pre všetky algoritmy (je možné vidieť na príklade algoritmu Suzuki-Kasami na obrázku 4.1). Ako prvé si užívateľ vyberie, s koľkými procesmi chce v danom príklade pracovať. Bude na výber pre každý algoritmus vždy

[1]Návrh aplikácie bol zhotovený pomocou nástroja na vytváranie prototypov Justinmind - https://www.justinmind.com/



Obr. 4.2: Hlavné menu aplikácie

rozumný počet procesov tak, aby sa ich vykreslenie zmestilo na jednu obrazovku bežného počítača a aby sa vykonávanie algoritmu dalo stíhať sledovať. Ďalšou položkou na výber bude jeden z možných príkladov pre daný počet procesov. Príklady budú navrhnuté tak, aby každý demonštroval trochu iné možné situácie a funkčnosť, ktorá by mohla v danom algoritme prekvapiť či nebyť z teoretického štúdia úplne jasná. Po tomto výbere nasleduje stlačenie tlačidla OK, po ktorom už prebehne samotný výpočet algoritmu a presmerovanie na ďalšiu stránku.

Na obrazovke samotného algoritmu sa objavia vysvetlivky jednotlivých správ pre daný algoritmus. Jednotlivé správy sú farebne rozlíšené pre lepšiu orientáciu v samotnej ukážke. Okrem toho sa na obrazovke vykreslí vybraný počet procesov, a to buď ako čiary v prípade Lamportovho algoritmu, alebo ako kruhy v prípade algoritmov ostatných. Na obrazovke bude tiež tlačidlo využívané na zobrazovanie jednotlivých správ, ktoré si procesy v algoritme posielajú. Stlačením tlačidla sa zobrazí vždy nová správa a zároveň sa môžu meniť niektoré z premenných. Vykreslenie každej správy je sprevádzané zrozumiteľným a stručným textovým popisom, ktorý ma pomôcť k pochopeniu významu danej akcie. Po prejdení všetkých správ a tým ukončení vykonávania daného algoritmu sa zobrazí hláška, že algoritmus skončil, a je možné sa tlačidlom *HOME* dostať naspäť na úvodnú stránku.

Jedinou vecou navyše okrem samotnej demonštrácie algoritmu bude v prípade Maekawovho algoritmu demonštrovanie výberu kvór biliardovou metódou. Užívateľ bude môcť kliknúť do mriežky na proces, ktorého kvórum chce zobraziť, a toto kvórum bude v mriežke zvýraznené.

Kapitola 5

Využité technológie a architektúra aplikácie

V tejto kapitole sú popísané technológie, ktoré boli využité na implementáciu aplikácie. Tiež vysvetľuje, akým spôsobom je pomocou týchto technológií implementovaný architektonický vzor Model-View-Controller (MVC) používaný na vývoj webových aplikácií.

5.1 Spring boot

Spring boot[1] je *open source framework* založený na jazyku Java, ktorý je používaný k vytváraniu mikroslužieb vyvinutých spoločnosťou *Pivotal Team* . Je veľmi dobre použiteľný na vytváranie samostatných aplikácií pripravených do produkcie. Je jedným z najpoužívanejších systémov na tvorbu webových technológií vďaka vytváraniu jednoduchých, flexibilných a rýchlych aplikácií. To je možné vďaka závislosti (angl. dependency) *spring-boot-starter-web*. Tá nám zaisťuje auto-konfiguráciu a automaticky stiahne všetky závislosti súvisiace s vývojom webu. Najdôležitejšími z nich sú Spring MVC, REST a Tomcat server. Poskytuje tiež množstvo *pluginov*, ktoré pomáhajú pri jednoduchom vývoji a zostavovaní aplikácií Spring Boot pomocou nástrojov ako je Gradle a Maven. Pre všetky tieto výhody bol Spring boot vybratý ako framework použitý pri implementácii tejto aplikácie[2].

Spring boot v tejto práci bol potrebný na vytvorenie kontrolera a komunikáciu s rôznymi koncovými bodmi pomocou REST architektúry. REST (Representational state transfer)[3] je rozhranie navrhnuté pre distribuované prostredie. Implementuje základné 4 metódy, ktoré sú známe pod označením CRUD (Create, Retrieve, Update a Delete, teda vytvorenie dát, získanie požadovaných dát, zmena a zmazanie). Tieto metódy sú implementované pomocou odpovedajúcich metód HTTP protokolu, a to GET, POST a PUT. Metódy využité v tejto aplikácii sú GET a POST. Viac v sekcii 5.5.

5.2 Gradle

Gradle[4] je *open-source* nástroj pre automatizáciu zostavovania programov zameraný na flexibilitu a výkon. Vznikol pôvodne pre prostredie Java, ale je navrhnutý tak, aby bol

[1] https://spring.io/projects/spring-boot
[2] https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm
[3] https://zdrojak.cz/clanky/rest-architektura-pro-webove-api/
[4] https://gradle.org/

dostatočne flexibilný na vytvorenie takmer akéhokoľvek typu softvéru. Je veľmi rýchly vďaka tomu, že využíva výstupy z predchádzajúcich spustení, spracovávajúc iba zmenené vstupy a vykonávajúc úlohy paralelne.

V súbore *build.gradle* tohto projektu bolo na jednoduché zostavovanie aplikácie potrebné pridať dôležité potrebné závislosti v časti *dependencies* . Jedná sa o *spring-boot-starter-web*, čo je závislosť potrebná na zostavovanie webových aplikácii v Spring Boot frameworku, a *spring-boot-starter-thymeleaf*, vďaka čomu je možné využívať funkcionalitu nástroju Java šablón Thymeleaf.

Použitím Gradle je možné spustiť aplikáciu z príkazového riadku. Je tiež možné vytvoriť spustiteľný súbor JAR, ktorý bude obsahovať všetky potrebné závislosti, triedy a iné zdroje a tým umožní jednoduché nasadenie či zdieľanie aplikácie medzi rôznymi prostrediami. Po spustení aplikácie stačí navštíviť webovú stránku *http://localhost:8080*, kde sa zobrazí základné menu aplikácie.

5.3 Thymeleaf

Thymeleaf[5] je moderný nástroj Java šablón určený na spracovávanie a vytváranie HTML, XML, JavaScript, CSS a textu. Je často využívaný s frameworkom Spring vo View vrstve Spring MVC aplikácií. Pre auto-konfiguráciu Thymeleaf v Spring Boot je potrebné pridať závislosť *spring-boot-starter-thymeleaf*.

Pomocou nástroja Thymeleaf je v tejto aplikácii vyriešené predávanie objektov z Modelu cez Kontroler do View aplikácie - do html súborov, ktoré to zas predajú ďalej JavaScript funkciám, ktoré sa už starajú o dynamické vykresľovanie elementov na obrazovku.

5.4 HTML, CSS, Bootstrap, JavaScript

HTML (HyperText Markup Language) je základným stavebným kameňom webu. Definuje význam a štruktúru obsahu webu. Okrem neho sú obecne používanými technológiami na prácu s webom CSS (Cascading Style Sheets), ktorý sa používa na popis vzhľadu webovej stránky, a JavaScript, ktorý popisuje funkcionalitu alebo správanie jednotlivých komponentov na webovej stránke.[6] Bootstrap[7] je *open-source* CSS framework, silný nástroj zložený z HTML, CSS a JavaScript dizajnových šablón, určených pre typografiu, formuláre, tlačidlá, navigáciu či iné komponenty front-end rozhrania.

V tejto aplikácii boli využité všetky tieto spomenuté nástroje. Webová stránka každého algoritmu predstavuje samostatný HTML súbor. Základným použitým elementom je *<svg>* element, ktorý slúži ako kontajner pre SVG grafiku. Tento element má množstvo metód na vytváranie čiar, kruhov, textu či iných typov grafiky. Pomocou JavaScriptu sú následne do tohto elementu pridávané či odstraňované vždy najskôr kruhy či čiary reprezentujúce procesy, a potom postupne jednotlivé šípky (čiary), predstavujúce posielané správy medzi procesmi. Každá stránka má svoj CSS súbor definujúci jej vzhľad. Nástroj Bootstrap bol využitý iba definovanie štýlov niektorých tlačidiel.

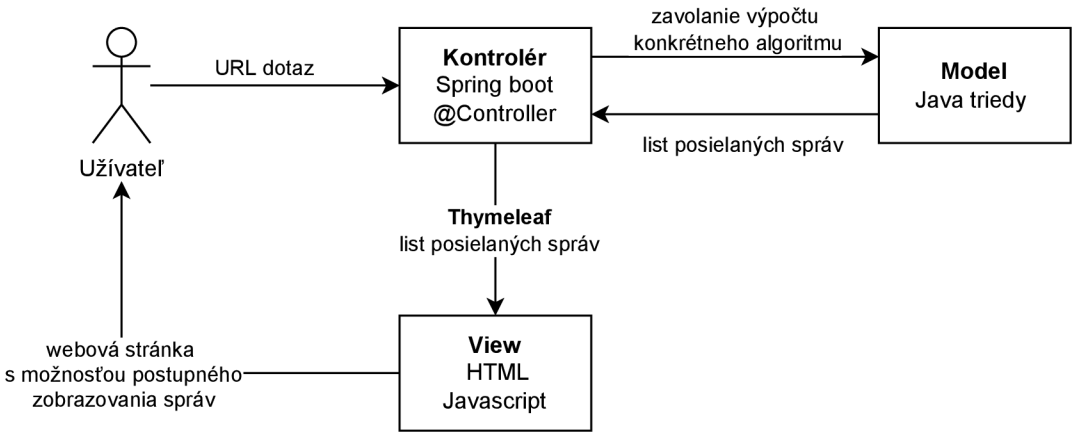
[5] https://www.thymeleaf.org/
[6] https://developer.mozilla.org/en-US/docs/Web/HTML
[7] https://getbootstrap.com/

5.5 MVC architektúra aplikácie

Základnou myšlienkou MVC architektúry je oddelenie logiky od výstupu. Rieši problémy s kódom, v ktorom máme v jednom súbore (triede) logické operácie a súčasne aj renderovanie výstupu. Takýto kód sa zle udržuje a rozširuje. Cieľom takejto architektúry vo webovej aplikácii je, aby zdrojový kód a logika vyzeral ako zdrojový kód v Jave a výstup vyzeral ako HTML stránka s čo najmenšou prímesou ďalšieho kódu. Celá aplikácia bude teda rozdelená do 3 častí: [12]

- Model - obsahuje logiku aplikácie a všetko, čo do nej spadá. Vôbec nevie o výstupe. Jeho funkcia spočíva v prijatí parametrov a vydania dát von.
- View - stará sa o zobrazenie výstupu užívateľovi.
- Controller - prostredník, s ktorým komunikuje užívateľ, Model aj View. Drží celý systém pohromade a jednotlivé komponenty prepojuje.

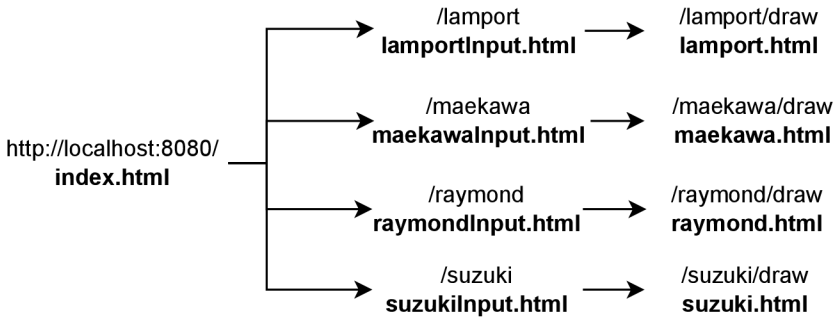
V tejto aplikácii bola MVC architektúra navrhnutá spôsobom, ktorý je možný vidieť na diagrame 5.1.



Obr. 5.1: MVC architektúra tejto aplikácie.

Hlavným komponentom aplikácie je Kontroler - trieda `ProcessController`. Ten je označený Spring anotáciou `@Controller`. Úlohou Kontrolera je odpovedať na žiadosti na jednotlivých definovaných koncových bodoch aplikácie. Typicky je táto anotácia využitá v kombinácii s `@RequestMapping` anotáciou. Tá už slúži priamo na mapovanie žiadosti na danej adrese na konkrétnu metódu triedy.

Keďže sa jedná o webovú aplikáciu, každá z funkcií v Kontroleri tejto aplikácie vracia konkrétny HTML súbor ako odpoveď na žiadosť na danej adrese. Prvotným bodom aplikácie je adresa *http://localhost:8080/*, ktorá vráti *index.html* súbor s úvodným menu aplikácie. Po výbere konkrétneho algoritmu z menu (stlačenie tlačidla) prebehne presmerovanie na odpovedajúcu stránku s výberom vstupných možností - to zaručujú koncové body */lamport, /maekawa, /raymond a /suzuki*. Metódy obsluhujúce tieto koncové body sú označené anotáciou `@GetMapping`. Tá predstavuje REST metódu GET, ktorá je využívaná na získavanie dát. Po stlačení tlačidla OK po vykonaní výberu možností už prebehne zavolanie ďalšieho odpovedajúceho koncového bodu (*/lamport/draw, /maekawa/draw, /raymond/draw či /suzuki/draw*), a to pomocou `<form>` HTML elementu, ktorý pomocou POST metódy pošle



Obr. 5.2: Zoznam jednotlivých koncových bodov aplikácie a ich postupnosť. Z každého koncového bodu je možné dostať sa stlačením tlačidla HOME na počiatočný koncový bod.

vybrané dáta na daný koncový bod. Zoznam všetkých koncových bodov aplikácie a ich následnosť je možné vidieť na obrázku 5.2. Metódy obsluhujúce tieto jednotlivé koncové body (príklad jednej z takýchto metód je možné vidieť na ukážke kódu 5.1) sa po prijatí vybratých vstupných hodnôt (trieda `UserChoice`) starajú o samotné zavolanie výpočtu algoritmu s vybranými hodnotami.

```
1  @RequestMapping(value = "/maekawa/draw", method = RequestMethod.POST)
2  public ModelAndView drawMaekawa(@ModelAttribute UserChoice userChoice) throws
       InterruptedException {
3
4      listOfMaekawaCoordinates = maekawaModel.calculateMaekawa(userChoice.
       getNumberOfProcesses(), userChoice.getExampleNumber());
5
6      ModelAndView model = new ModelAndView();
7
8      model.addObject("listOfCoordinates", listOfMaekawaCoordinates);
9      model.addObject("numberOfProcesses", userChoice.getNumberOfProcesses());
10     model.addObject("exampleNumber", userChoice.getExampleNumber());
11
12     model.setViewName("maekawa");
13     return model;
14 }
```

Výpis 5.1: Príklad jednej z metód triedy `ProcessController`. Jedná sa konkrétne o metódu obsluhujúcu požiadavky na koncový bod `/maekawa/draw`, ktorý je vyvolaný vyplnením formuláru so vstupnými hodnotami v HTML po stlačení tlačidla OK. Trieda `UserChoice` obsahuje 2 užívateľom vyplnené vstupné hodnoty, a to počet procesov (`numberOfProcesses`) a číslo príkladu (`exampleNumber`). Tieto hodnoty sú poslané do Modelu Maekawovho algoritmu, ktorý zavolá metódu `calculateMaekawa`, kde prebehne samotný výpočet algoritmu. Vrátený je list posielaných správ `listOfMaekawaCoordinates` (riadok 4 v kóde). Všetky tieto premenné sa potom posielajú do rozhrania *ModelAndView* (riadok 6), ktoré nám umožňuje odovzdať všetky potrebné dáta do View v Spring MVC architektúre. Tieto dáta sú odovzdávané metódou `addObject` (riadky 8, 9, 10). Ako posledné je potrebné určiť názov HTML súboru, ktorý bude funkciou vrátený (riadok 12). Rovnakým spôsobom fungujú obsluhujúce metódy pre všetky algoritmy aplikácie.

Model aplikácie tvoria Java triedy, ktoré vykonávajú hlavnú funkcionalitu algoritmov. Každý algoritmus má vlastné triedy, ktoré sa starajú o výpočet algoritmu, ukladanie posie-

laných správ do listu a následné predanie tohto listu späť Kontrolera. Adresárová štruktúra tejto časti sa nachádza v prílohe B.

Všetky súbory ***Model.java** obsahujú:

- funkciu na vrátenie konkrétneho preddefinovaného príkladu podľa vstupných parametrov. Uložené sú časy žiadania jednotlivých procesov o vstup do KS. Tieto časy sú uložené v jednoduchých poliach čísel.
- funkciu na vytvorenie jednotlivých vlákien (procesov) a ich spustenie.
- funkciu na namapovanie vrátených správ na koordináty a iné potrebné atribúty grafických objektov, ktoré budú vykresľované.

Tieto súbory sú označené Spring anotáciou `@Component`. Komponent je špeciálny typ triedy, ktorý je auto-detekovaný Springom a využitý na mechanizmus *dependency-injection* (vkladanie závislostí). Keď Spring Boot spustí aplikáciu, nájde všetky triedy s danou anotáciou a vytvorí objekty týchto tried, ktoré budú čakať v akomsi kontajneri na použitie. Vďaka tomu nie je potrebné vytvárať objekty v iných triedach, ale vložiť závislosti na tieto triedy pomocou anotácie `@Autowired`. Závislosť na tieto triedy je vložená do Kontrolera aplikácie, vďaka čomu sú zodpovednosti jednotlivých tried efektívnejšie prideľované.

Všetky súbory ***Process.java** implementujú Java interface `Runnable`, čím je možné tieto triedy spustiť ako vlákna. Týchto vlákien bude toľko, koľko bolo vybratých procesov užívateľom. Každý proces môže posielať a prijímať správy a jedenkrát žiada o vstup do KS. Všetky vlákna skončia, keď už každý proces dostal možnosť vykonávať KS.

Všetky súbory ***Message.java** slúžia na ukladanie jednotlivých správ, ktoré si procesy v snahe o prístup do KS na zaručenie vzájomného vylúčenia posielajú. Pre každý algoritmus je potrebné uložiť rôzne typy správ, ako aj rôzne premenné, ktoré tieto správy ovplyvňujú. Tieto správy, uložené ako Java objekty, je potom potrebné zmeniť na už konkrétne hodnoty, ktoré budú vykresľované na obrazovku pri demonštrácii, ako napríklad x-ové a y-ové súradnice šípok, ktoré budú tieto správy predstavovať. Definície týchto hodnôt sú pre jednotlivé algoritmy uložené v súboroch ***Coordinates.java**. Po vykonaní algoritmu sa teda uložené objekty typu *Message prekonvertujú na objekty typu *Coordinates, ktoré už potom budú posielané späť do Kontrolera a odtiaľ do View programu.

View aplikácie je tvorený html súbormi, ktoré sú vracané užívateľovi po vyvolaní konkrétneho koncového bodu, ako už bolo vysvetlené vyššie v časti o Kontroleri. Do týchto html súborov sú pomocou nástroja Thymeleaf posielané Java *ArrayList*-y, ktoré sú z html preposlané do jednotlivých JavaScript súborov - *lamport-func.js, maekawa-func.js, raymond-func.js a suzuki-func.js*. Funkcia všetkých týchto súborov je veľmi podobná, líši sa len konkrétnymi vykresleniami pre daný algoritmus. Obecne ale tieto súbory obsahujú 3 JavaScript funkcie. Prvá funkcia slúži na vytvorenie statických elementov, ktoré sa už nebudú meniť. Na to je potrebná premenná poslaná z Kontrolera, vyjadrujúca počet procesov v systéme. Funkcia je zavolaná priamo z html súboru, aby bolo tieto elementy vidieť okamžite. Vykresľované sú čiary/kruhy predstavujúce procesy, vytvorené sú textové elementy jednotlivých premenných v algoritmoch, ktorým sa už potom bude meniť iba textový obsah. Druhá funkcia slúži na vykresľovanie dynamických objektov - čiar predstavujúcich správy medzi procesmi. Práve táto funkcia potrebuje pre svoje vykonávanie list získaný z Kontrolera. Volaná je ako *onclick* funkcia tlačidla s názvom *Next message*, ktorým si užívateľ postupne vykresľuje jednotlivé správy, a to takým spôsobom, že je v JavaScript súbore uložený index

listu ako globálna premenná `coordinateIndex`, ktorá sa po každom stlačení tlačidla zvýši o 1, a teda každým kliknutím bude vykreslená ďalšia správa z listu. Tretia funkcia je krátka a slúži na funkčnosť tlačidla *One step back*, pomocou ktorého je možné sa vrátiť o jednu správu späť vo vykonávaní algoritmu. To je možné tak, že na začiatku funkcie vykresľovania správy sa uloží aktuálny stav *div* elementu, do ktorého sú všetky veci vykresľované. V tejto funkcii sa aktuálny stav tohto elementu prepíše na ten predchádzajúci, a zároveň sa zníži premenná `coordinateIndex` o 1, čím je možné potom zas pokračovať vo vykresľovaní správ z listu.

Kapitola 6

Implementácia knižníc a demonštračnej aplikácie

V tejto kapitole budú podrobnejšie popísané konkrétne postupy a riešenia, ktorými bola aplikácia implementovaná. Aplikáciu je možné rozdeliť do dvoch častí - back-end, ktorý tvorí implementácia knižníc jednotlivých algoritmov, teda to, ako procesy v algoritmoch pracujú a posielajú si správy. Front-end aplikácie už tvorí webová stránka, na ktorej sú výpočty konkrétnymi príkladmi demonštrované.

6.1 Implementačné detaily jednotlivých algoritmov

Vykonávanie algoritmov je simulované pomocou Java vlákien, ktoré predstavujú procesy v systéme, ktoré sa majú navzájom synchronizovať v snahe o prístup do KS. Na začiatku každého algoritmu je teda vytvorených a spustených toľko vlákien, koľko procesov v danom príklade má byť. Každé vlákno pritom vykonáva rovnaký algoritmus, definovaný v triede *Process.java. Táto trieda môže byť spustená ako vlákno vďaka jej implementácii triedy `Runnable`. Trieda obsahuje vždy nejaké potrebné statické atribúty, ktoré sú využívané ako premenné, ktoré majú všetky procesy spoločné. Sú nimi napríklad číslo `numberOfProcesses` (počet procesov), list `listOfProcesses` (zoznam všetkých procesov), list `allMessagesToDraw` (zoznam všetkých správ, ktoré boli pri vykonávaní algoritmu posielané a budú vykresľované). Ďalším statickým atribútom je v algoritme Suzuki-Kasami pole čísel `TokenArray`, ktoré predstavuje vektor hodnôt LN tokenu predávaného medzi procesmi (viď 3.5).

Vlákna (procesy mojej aplikácie) musia medzi sebou nejakým spôsobom komunikovať. Na komunikáciu má byť využité posielanie správ. Posielanie správ medzi procesmi je v jazyku Java vykonávané ako posielanie objektu z jedného vlákna do druhého. Na jeho poslanie sú používané synchronizované premenné (listy, fronty). Podľa rôznych zdrojov - napríklad [4], [1] - je jedným možným prístupom používanie takých frontov na posielanie správ, kde keď sa žiadna správa nenachádza vo fronte z ktorého chce proces čítať, proces sa pozastaví a čaká na jej príchod. V mojom prípade ale takéto chovanie nie je žiaduce - proces v mojom systéme potrebuje vykonávať rôzne akcie, môžu mu prichádzať rôzne typy správ, ktoré treba rozlíšiť a až keď mu nejaká správa príde, pristúpi k nej a naviaže na ňu nejakou akciou. Tento prístup teda fungovať nebude.

Keďže však beží viac vlákien súčasne, nemôžu na komunikáciu používať obyčajné premenné, pretože súčasný prístup viacerých vlákien k nim by mohol spôsobiť chyby, problém v

nekonzistencii dát, a v prípade cyklického prechodu frontu jedným vláknom a zároveň zápisom do fronty vláknom druhým by nastala `ConcurrentModificationException` (výnimka, ktorá nastane vo viac-vláknovej aplikácii, keď nastane snaha o súčasnú zmenu objektu keď takáto zmena nie je povolená). Premenné, pri ktorých tieto veci hrozia, pretože k nim bude pristupovať viac procesov naraz, sú v mojej aplikácii teda práve spomínané fronty na vymieňanie jednotlivých druhov správ a list všetkých posielaných správ, pretože dve správy môžu byť dvomi procesmi kľudne poslané v rovnaký čas. Kvôli všetkým možným spomenutým problémom sú na implementáciu týchto štruktúr použité Java premenné typu `synchronizedList` - ukážka kódu 6.1. Tento typ listu je podľa oficiálnej Java dokumentácie synchronizovaný a tzv. *thread-safe* (voľným prekladom bezpečný pre prístup viacerých vlákien). Je tým zabezpečené, že žiadne správy nebudú stratené a tiež že zápis a čítanie z týchto premenných prebehne postupne, nie v úplne rovnaký časový okamih.

```
1 public static List<RaymondMessage> allMessagesToDraw = Collections.synchronizedList(new
      ArrayList<>());
```

Výpis 6.1: List typu `synchronizedList`, používaný ako *thread-safe* premenná. Na ukážke zobrazená premenná `allMessagesToDraw` v Raymondovom algoritme, využitá na uloženie všetkých posielaných správ medzi jednotlivými vláknami (procesmi) algoritmu.

Beh všetkých algoritmov funguje na veľmi podobnom princípe. Všetky procesy v každom z implementovaných algoritmov vykonávajú *while* cyklus, v ktorom buď zisťujú či im prišla nejaká správa od iného z procesov, kedy na danú správu odpovedajú, alebo sami zasielajú správu keď chcú požiadať o vstup do KS.

Lamportov algoritmus

V Lamportovom algoritme sú zasielané 3 typy správ - REQUEST, REPLY a RELEASE. Pre každý typ je zvlášť vytvorený synchronizovaný list, do ktorého správu iný proces vloží. Priebeh cyklu spočíva v tom, že najskôr proces zistí, či nie je aktuálny čas na jeho žiadosť o vstup do KS a potom proces kontroluje, či mu neprišla nejaká správa. Podľa princípu algoritmu pritom proces zvýši svoje logické hodiny o 1 medzi akýmikoľvek dvomi udalosťami. Keď chce proces vstúpiť do KS, pošle správu REQUEST všetkým procesom, ku ktorým má prístup vďaka uloženému listu procesov. Všetky poslané správy sa uložia do listu na vykreslenie. Keď mu nejaká žiadosť príde, uloží do listu na vykreslenie, že správu prijal a odpovie na ňu REPLY správou. Keď procesu príde REPLY správa, uloží si ju, aby vedel, že od daného procesu už REPLY dostal. Keď totiž dostane REPLY od všetkých procesov, vie, že môže vstúpiť do KS v prípade, že je na začiatku svojho frontu žiadostí. Uloží tiež udalosť, ktorá hovorí, že vstupuje a vystupuje z KS. Keď proces z KS vystúpi, pošle RELEASE správu všetkým procesom. Premenné, ktoré je potrebné ukladať, sú časy zasielania a prijatia jednotlivých správ a fronty procesov, ktorých stav sa pri prijatí/odosielaní správ menil.

Maekawov algoritmus

Maekawov algoritmus funguje podobným spôsobom, len s tým, že obsahuje viac typov správ, čo je podrobne vysvetlené v časti 3.3, a preto musí obsahovať viac listov na tieto správy a celková implementácia je dlhšia. Špecifikum u tohto algoritmu je, že neposiela žiadosť všetkým procesom, ale len procesom z jeho kvóra - to má každý proces napevno uložené. U

procesov sú ukladané ich fronty a tiež informácia, ktorému procesu aktuálne pridelili právo na vstup do KS (sú pre neho akoby uzamknuté).

Raymondov algoritmus

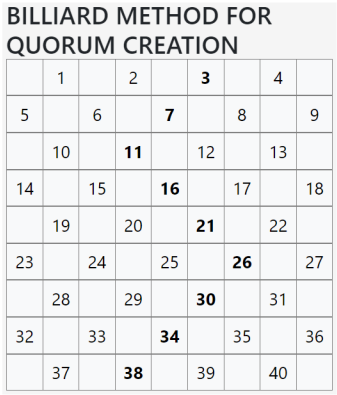
Raymondov algoritmus má menší počet správ (iba REQUEST a poslanie PRIVILEGE), okrem toho však ešte špeciálnu správu INITIALIZE, ktorou sa všetky procesy na začiatku inicializujú, čo znamená, že si dajú navzájom postupne vedieť, kto vlastní PRIVILEGE. Až keď túto správu dostanú a ďalej pošlú svojim susedom, môžu prejsť k vykonávaniu algoritmu. Ten funguje podobne ako všetky doposiaľ, a teda že buď posiela alebo prijíma správy. Tento algoritmus si ukladá väčšie množstvo premenných ako doterajšie algoritmy - na obrazovku bude vypisovať obsah premenných HOLDER, USING, ASKED a front QUEUE.

Algoritmus Suzuki-Kasami

Tento algoritmus má dva typy správ - REQUEST a SEND TOKEN. Má však dôležité premenné, ktoré musí ukladať - vektor hodnôt každého procesu, vektor hodnôt tokenu a front tokenu (vysvetlenie v časti 3.5).

Biliardová metóda vytvárania kvór

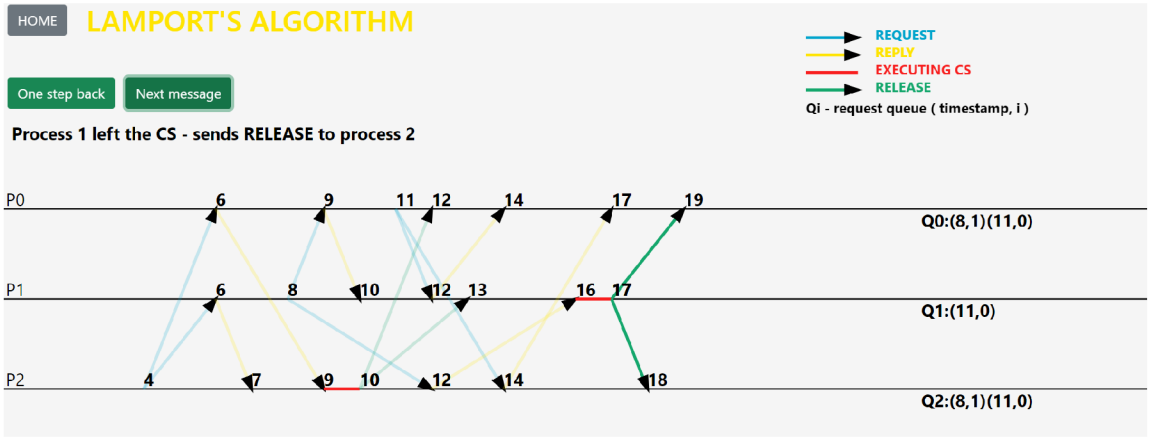
Súčasťou Maekawovho algoritmu je demonštrácia vytvárania kvór biliardovou metódou. Demonštrácia prebieha na jednom príklade kde $N = 40$ (počet procesov) a $q = 9$ (veľkosť kvóra). Algoritmus vytvárania kvór je naprogramovaný pomocou pseudo-algoritmu z [3], strana 7. Študent môže kliknúť na číslo ktoréhokoľvek procesu v mriežke a zvýrazní sa mu kvórum daného procesu, vypočítaný pomocou biliardovej metódy. Ukážka je na obrázku 6.1. Táto demonštrácia je zobrazená na stránke výberu vstupných hodnôt Maekawovho algoritmu. Výpočet algoritmu prebieha v súbore *quorums-func.js*.



Obr. 6.1: Ukážka demonštrácie vytvárania kvór biliardovou metódou v aplikácii. Zvýraznené je kvórum pre proces číslo 26.

6.2 Podrobnosti demonštrácie jednotlivých algoritmov

Každý algoritmus je niekoľkými príkladmi demonštrovaný, pričom táto demonštrácia slúži na sledovanie vykonávania tohto algoritmu, a teda aké správy si medzi sebou procesy posielali. Každé poslanie správy je sprevádzané krátkym výstižným popisom na vrchu obrazovky. Každá šípka predstavujúca správu je vykresľovaná inou farbou pre rozoznanie typov správ (rovnaké farby sú pre rovnaké typy správ udržované u všetkých algoritmov). Na stránke každého algoritmu sú tiež vysvetlivky jednotlivých správ, a teda ktorá farba predstavuje ktorú správu. Ďalej sa už samotná demonštrácia a čo je popri nej na obrazovke vypisované pri každom algoritme mierne líši.



Obr. 6.2: Ukážka vykonávania Lamportovho algoritmu v aplikácii.

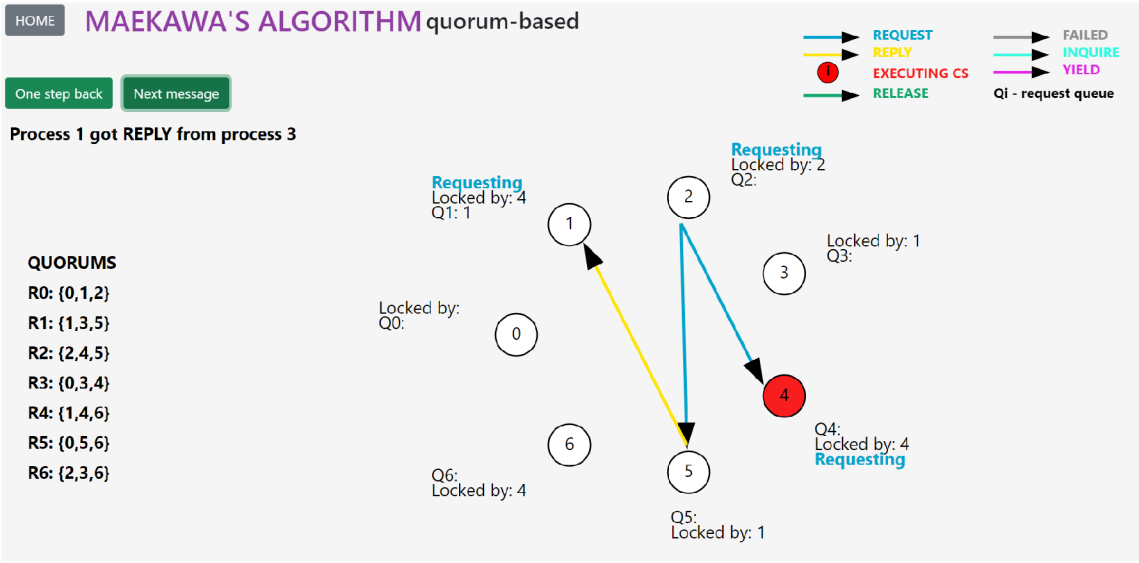
Lamportov algoritmus

V prípade Lamportovho algoritmu sú jednotlivé procesy zobrazené ako čiary. Tie predstavujú akúsi časovú os, na ktorej budú postupne podľa logického času zobrazované odoslané a prijaté správy. Keď sa správa odošle, vykreslí sa šípka medzi osami, keď správu prijímateľ obdrží, šípka zbledne, aby bolo jasne na obrazovke vidieť, ktoré správy ešte neboli prijaté, iba odoslané. Vykonávanie KS je zobrazené krátkou červenou čiarou na ose daného procesu. Na pravom okraji čiary každého procesu je priestor pre vypisovanie frontu daného procesu, ktorý sa bude v priebehu času meniť, ako žiadosti budú pribúdať a ubúdať. Na obrázku 6.2 je možné vidieť snímku z aplikácie.

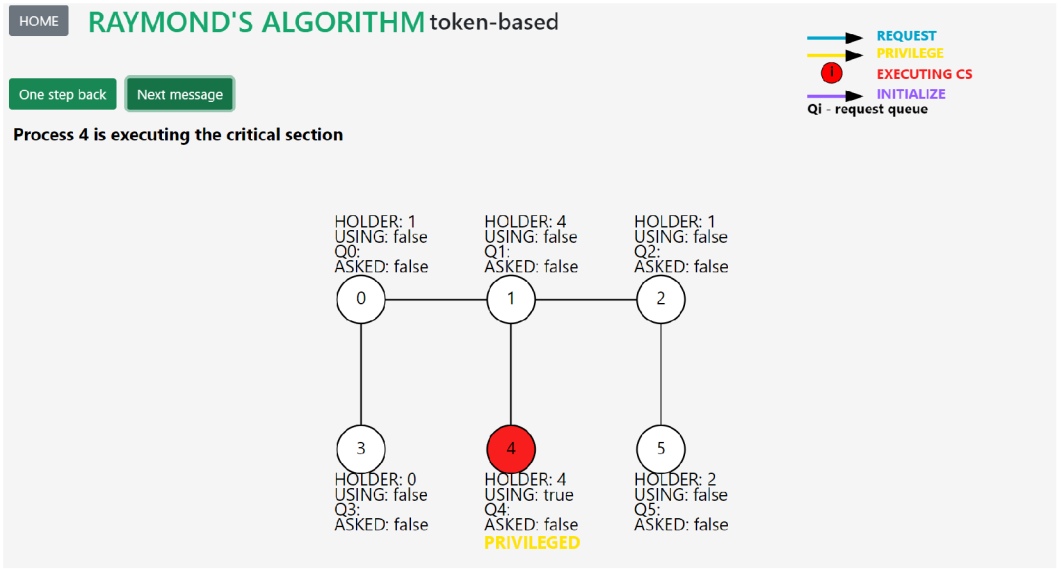
Maekawov algoritmus

Procesy v tomto algoritme sú zobrazené iným spôsobom, a to ako kruhy, ktoré sú usporiadané do kruhu - pre jednoduché zobrazovanie zasielania správ medzi akoukoľvek dvojicou procesov. Keď proces odošle nejakú správu, zobrazí sa šípka, keď ju iný proces príjme, šípka zmizne. Jej prijatie môže byť sprevádzané zmenou premenných, ktoré budú pri každom procese vypisované. Keď proces požiada o vstup do KS, zobrazí sa pri ňom zvýraznený nápis *Requesting*, aby bolo jasné, ktoré všetky procesy požiadali o vstup a ešte im nebolo vyhovené. Nápis zmizne po tom, ako proces opustí KS. Keď proces vstúpi do KS, jeho kruh sa zvýrazní na červeno (na začiatku sú všetky kruhy biele). Keď proces KS opúšťa, kruh

sa opäť sfarbí na bielo. Na obrazovke sú tiež vypísané kvóra všetkých procesov. Ukážku z aplikácie je možné vidieť na obrázku 6.3.



Obr. 6.3: Ukážka vykonávania Maekawovho algoritmu v aplikácii.

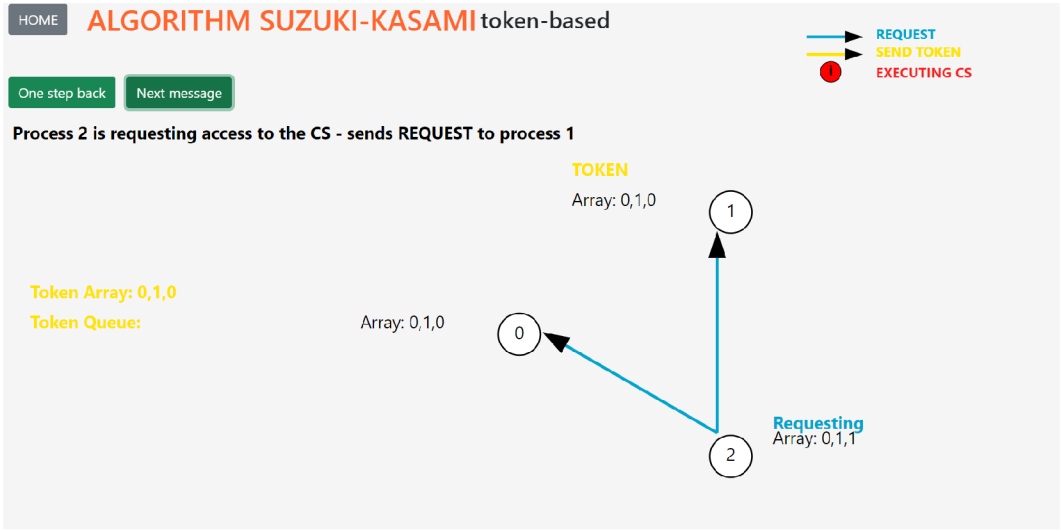


Obr. 6.4: Ukážka vykonávania Raymondovho algoritmu v aplikácii.

Raymondov algoritmus

Raymondov algoritmus je trochu iný, a to v tom, že si užívateľ nevyberá na začiatku počet procesov, ale v každom príklade je ich 6. Je to preto, pretože kvôli stromovej štruktúre usporiadania procesov sa algoritmus zásadne nelíši, keď ich je viac alebo menej. Číslo 6 je vybrané náhodne, predstavuje jednoduchý strom na ktorom sa posielané správy dobre

sledujú. Procesy sú vykreslené ako kruhy, usporiadané do stromu, a susedné uzly, ktoré si budú posielať správy, sú prepojené čiarami. Šípky odosielaných správ tieto čiary prekreslia, príjem správy ich zas naspäť odhalí, keďže farebná šípka pri prijatí správy zmizne. Pri každom procese sú vypísané všetky jeho premenné, ktoré sa vykonávaním algoritmu menia. Zobrazenie vstupu a výstupu z KS je obdobné s Maekawovym algoritmom. Naznačené je tiež, ktorý proces aktuálne vlastní PRIVILEGE, a to nápisom *PRIVILEGED*, ktorý sa vždy zobrazí pri aktuálnom privilegovanom procese. Ukážka je na obrázku 6.4.



Obr. 6.5: Ukážka vykonávania algoritmu Suzuki-Kasami v aplikácii.

Algoritmus Suzuki-Kasami

Tak ako v Maekawovom algoritme, aj v tomto sú procesy zobrazené ako kruhy usporiadané do kruhu, pričom šípky predstavujúce správy sa pri odoslaní vykreslia a pri prijatí zmiznú. Každý proces má pri sebe zobrazený svoj vektor hodnôt (nazvaný ako *Array*). Zobrazenie vstupu a výstupu z KS je obdobné s Maekawovym algoritmom. Takisto je zobrazovaný nápis *Requesting* tak, ako v Maekawovom algoritme. Okrem toho je ešte naznačené, ktorý proces aktuálne vlastní token, a to nápisom *TOKEN* pri kruhu procesu. Na kraji obrazovky sa tiež vypisujú premenné tokenu - jeho vektor hodnôt a front. Snímka z aplikácie je na obrázku 6.5.

6.3 Problémy počas implementácie a ich riešenie

Počas implementácie funkčnosti jednotlivých algoritmov som sa stretla s rôznymi problémami, ktoré bolo potrebné vyriešiť či dať si na nich pozor. Táto sekcia popisuje niektoré z nich.

Jeden z problémov môjho prvotného návrhu pre všetky algoritmy bol, že do poľa všetkých správ, ktoré mali byť následne vykreslené, bolo uložené iba odoslanie správy. Znázornené na obrazovke bolo odoslanie nejakej správy, a pri ďalšej správe mala byť predchádzajúca zmazaná. Premenné, ktoré sa poslaním správy na oboch stranách (prijímateľa i

odosielateľa) zmenili, boli uložené a vykreslené v rámci tejto jednej správy. Vo funkčnosti priebehu algoritmu to nebol žiadny problém, ten sa ale ukázal pri vykresľovaní správ a vypisovaní spomenutých premenných. Ak daná správa prišla príjemcovi neskôr, ako sám odoslal už nejakú inú, ktorá ovplyvnila tie isté premenné, na obrazovke sa prepísal stav týchto premenných na starší. Toto chovanie bolo nežiaduce, keďže hlavným cieľom v tejto aplikácii je, aby študent mohol sledovať správne postupné chovanie algoritmu a zmenu daných premenných počas jeho vykonávania. Po zistení týchto skutočností na základe testovania počas implementácie aplikácie bol vymyslený iný spôsob - do listu sa bude ukladať okrem odoslania správy aj prijatie tej istej správy, keďže tento proces odoslanie-prijatie nemusí nastať hneď za sebou, ale byť prerušený vymieňaním ďalších správ. Odoslanie správy bude pritom meniť iba premenné odosielateľa a šípku danej správy vykreslí. Prijatie správy danú šípku buď vymaže, alebo zbledne (v prípade Lamportovho algoritmu), a bude meniť premenné prijímateľa danej správy. Keď teda nastane situácia, že proces pred prijatím danej správy odoslal nejakú inú, premenné sa vypíšu presne v takom poradí, ako sa vo vykonávaní algoritmu skutočne menili. Tým budú odstránené všetky nekonzistencie a demonštrácia prebehne správne.

Drobným problémom, ktorý bol nakoniec veľmi jednoduchým spôsobom vyriešený vďaka článku [5], bolo vytváranie šípok, ktoré mali predstavovať posielané správy. Vykreslenie čiar prebiehalo pomocou jednoduchého SVG *<line>* elementu. Na koniec tejto čiary bolo následne potrebné pridať šípku, ktorá by ukončovala čiaru a ukazovala v jej smere. Vyriešené to bolo pomocou *<marker>* elementu, ktorý je umiestnený dovnútra *<defs>* elementu, vďaka čomu nebude vykreslený v mieste jeho definície, ale je možné sa naňho odkázať z iného miesta a použiť ho tým. Atribútmi elementu bol v podstate vykreslený malý trojuholník predstavujúci šípku. Vďaka atribútu *orient=auto* sa šípka prispôsobila akémukoľvek smeru čiary. Každej vykreslenej čiare bolo teda definované takéto zakončenie.

Kapitola 7

Testovanie aplikácie

Počas celej tvorby bola aplikácia a jej funkcionalita testovaná. Pri počiatočných fázach vývoja, kedy ešte nebolo vytvorené užívateľské rozhranie, prebiehalo testovanie vypisovaním jednotlivých správ na štandardný výstup. Tým bolo odhalených vždy mnoho chýb, súvisiacich najmä s nesprávnym chodom algoritmu - častými chybami bolo uviaznutie kvôli nedobre definovaným podmienkam, chyby spôsobené prístupom viacerých vlákien k jednej premennej. Po vytvorení webovej demonštrácie daného algoritmu už bola správnosť sledovaná tam - tým sa ukázal napríklad okrem iného aj problém so synchronizáciou, keď sa neukladali všetky posielané správy kvôli nesprávne definovanej premennej na ukladanie správ. Každý algoritmus bol otestovaný na množstve príkladov, z ktorých boli pre výslednú verziu aplikácie vybrané tie, ktoré ukazujú čo najrôznejšie scenáre, ako napríklad žiadanie o prístup do KS viacerými procesmi naraz, v rovnaký čas.

Ďalšou potrebnou časťou testovania bolo užívateľské testovanie - samotnými študentami, ktorí aplikáciu budú používať. To prebiehalo formou premýšľania nahlas, a teda po tom, čo som aplikáciu študentom predstavila, požiadala som ich, aby si aplikáciu spustili a skúsili ju použiť, pričom mi budú nahlas rozprávať, čo sa podľa nich deje a nad čím premýšľajú. Bolo možné sledovať, že interakcia s aplikáciou sa študentom javí jednoduchá, pochopiteľná a intuitívna. Vedeli, kam kliknúť a čo zobrazené elementy na obrazovke predstavovali. Bolo odhalených aj pár drobných chýb - chýbajúce vysvetlivky niektorých skratiek v legende, textový popis nedostatočne vysvetľujúci čo sa deje - všetky tieto chyby boli opravené.

Primárnym cieľom bolo ale zistiť, či sú študenti schopní z aplikácie pochopiť princíp algoritmov. To sa prejavilo takisto ako úspešné. Vzorka študentov, ktorí už daný algoritmus poznali, boli schopní rýchlo si spomenúť pomocou aplikácie na jeho funkcionalitu a pochopiť chovanie v neštandardných demonštračných situáciách (napr. keď viac procesov žiada o vstup do KS súčasne). To bolo možné sledovať, keď počas používania aplikácie najskôr pomaly zobrazovali a čítali si popis k ďalšej a ďalšej správe, ale ku koncu algoritmu už tempo zrýchlili a sami vedeli predpokladať, čo sa môže stať. Keďže má aplikácia slúžiť práve takýmto študentom, ktorí už o algoritmoch počuli a poznajú ich základy z prednášky, výsledok testovania sa dá považovať za úspešný.

Ďalšia vzorka študentov boli študenti bakalárskeho štúdia, ktorí ešte dané algoritmy nepreberali a nikdy o nich nepočuli. Výsledok testovania bol u všetkých veľmi podobný. Študenti boli pomocou aplikácie schopní pochopiť, akým spôsobom algoritmy fungujú, hoci trochu pomalším tempom. Tvrdili, že znázornenie je prevedené dobrou formou, a keď sa človek sústredí a dostatočne nad tým premýšľa, dokáže ich rýchlo pochopiť. Zhodnotili tiež, že by im pomohlo, keby im je najskôr algoritmus teoreticky vysvetlený, aby vedeli, čo od aplikácie majú čakať a rýchlejšie sa do toho dostali. To odpovedá prípadu užitia tejto

aplikácie, keďže práve to sa predpokladá - že ju budú používať študenti, ktorí už prednášku o nich absolvovali, alebo ju aspoň majú k dispozícii a až po jej prečítaní im táto aplikácia prispeje k rýchlejšiemu a lepšiemu zapamätaniu.

Študenti podali aj pár návrhov, ktoré by mohli byť v ďalších vylepšeniach aplikácie implementované. Jednalo sa hlavne o vizuálne vylepšenia - animovanie podávania tokenu medzi procesmi, škrtanie frontov miesto aktualizovania, aby bolo pekne aj na konci algoritmu viditeľné, ako sa algoritmus vyvíjal a bolo možné si ho na konci akoby zrekapitulovať. Celkovo je možné zhodnotiť, že testovanie pomohlo k vylepšeniu aplikácie a je možné predpokladať, že študentom, ktorí ju budú používať, aplikácia naozaj pomôže porozumieť a zapamätať si algoritmy, čo bolo hlavným cieľom tejto práce.

Kapitola 8

Záver

Cieľom tejto bakalárskej práce bolo implementovať demonštračnú aplikáciu, ktorá by pomohla študentom naučiť sa vybrané synchronizačné algoritmy komunikujúce predávaním správ pre zaistenie vzájomného vylúčenia. Po podrobnom naštudovaní funkcionality algoritmov bola navrhnutá architektúra aplikácie a jednotlivé algoritmy naprogramované. Vyvinutá bola webová aplikácia, po ktorej spustení je možné sledovať vykonávanie jednotlivých algoritmov, posielanie správ a je možnosť vybrať si z rôznych príkladov. Študent si môže postupne jednotlivé správy vykresľovať a sledovať priebeh algoritmu. Užívateľské rozhranie pomáha k rýchlej orientácii a pochopeniu algoritmov. Na základe úspešného vytvorenia a otestovania aplikácie potencionálnymi užívateľmi je možné usúdiť, že ciele práce boli naplnené.

Navrhnutá aplikácia demonštruje všetky 4 spomenuté algoritmy - Lamportov, Maekawov, Raymondov a Suzuki-Kasami. Ich funkcionalita bola otestovaná a vyskúšaná na viacerých príkladoch. Aplikácia pracuje spoľahlivo a dostačujúco rýchlo. Práca na tejto aplikácii ma naučila najmä navrhnúť a orientovať sa vo väčšom projekte, vytvoriť jednoduchú webovú aplikáciu pomocou pre mňa nového frameworku a pracovať s vláknami a synchronizáciou v jazyku Java.

Aplikácia je zverejnená na stránkach predmetu Paralelní a distribuované systémy FIT VUT a je dostupná pre študentov ako JAR súbor *distributedSystemsApp.jar*.

Z pohľadu ďalšieho vývoja aplikácie vidím najväčší možný prínos v implementácii mechanizmu, kde by si užívateľ mohol sám zadávať časy žiadostí procesov o vstup do kritickej sekcie. To by bolo možné nejakým pochopiteľným užívateľským rozhraním a následne by prebehol výpočet algoritmu so zadanými hodnotami. Tiež by mohlo byť dobrým rozšírením možnosť zasiahnuť do vykonávania algoritmu - určiť že nejaká správa nebude odoslaná, stratí sa a pod. a sledovať, ako sa algoritmus pri tom zachová. Takisto by bolo možné implementovať nejaké ďalšie existujúce algoritmy, ako napríklad rozšírenie Lamportovho algoritmu - algoritmus Ricard-Agrawala.

Literatúra

[1] Agarwal, N. *Message Passing in Java* [online]. GeeksforGeeks, október 2019 [cit. 2022-04-24]. Dostupné z: https://www.geeksforgeeks.org/message-passing-in-java/.

[2] Agrawal, D., Eğecioğlu Ömer a El Abbadi, A. Billiard quorums on the grid. *Information Processing Letters*. 1997, zv. 64, č. 1, s. 9–16. DOI: https://doi.org/10.1016/S0020-0190(97)00145-2. ISSN 0020-0190.

[3] Agrawal, D., Eğecioğlu, O. a El Abbadi, A. Billiard quorums on the grid. *Information processing letters*. Department of Computer Science, University of California, Santa Barbara: Elsevier B.V. 1997, zv. 64, č. 1, s. 9–16. ISSN 0020-0190.

[4] Amarasinghe, S., Chlipala, A., Devadas, S., Ernst, M., Goldman, M. et al. *Message Passing with Threads and Queues* [online]. Massachusetts Institute Of Technology, 2014 [cit. 2022-04-24]. Dostupné z: https://web.mit.edu/6.005/www/fa14/classes/20-queues-locks/message-passing/.

[5] Bradley, S. *How To Create SVG Arrowheads and Polymarkers — The marker Element* [online]. vanseodesign.com, 04. mája 2015 [cit. 2022-04-23]. Dostupné z: https://vanseodesign.com/web-design/svg-markers/.

[6] Kshemkalyani, A. D. a Singhal, M. *Distributed computing: Principles, Algorithms, and Systems*. 2. vyd. Cambridge ; New York : Cambridge University Press, marec 2011. 736 s. ISBN 978-0-521-18984-2.

[7] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*. New York, NY, USA: Association for Computing Machinery. Júl 1978, zv. 21, č. 7, s. 558–565. ISSN 0001-0782.

[8] Lamport, L. *Distribution email*, 28. mája 1987 12:23:29 PDT [cit. 2021-11-22]. Email poslaný nástenke DEC SRC 12:23:29 PDT 28. mája 1987. Dostupné z: https://www.microsoft.com/en-us/research/publication/distribution/.

[9] Maekawa, M. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Trans. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. Máj 1985, zv. 3, č. 2, s. 145–159. DOI: 10.1145/214438.214445. ISSN 0734-2071.

[10] Raymond, K. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Trans. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. Február 1989, zv. 7, č. 1, s. 61–77. DOI: 10.1145/58564.59295. ISSN 0734-2071.

[11] Ricart, G. a Agrawala, A. K. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*. New York, NY, USA: Association for Computing Machinery. Január 1981, zv. 24, č. 1, s. 9–17. DOI: 10.1145/358527.358537. ISSN 0001-0782.

[12] Střecha, T. *Úvod do MVC architektury ve Spring Bootu* [online]. https://www.itnetwork.cz/ [cit. 2022-04-16]. Dostupné z: https://www.itnetwork.cz/java/spring-boot/spring-boot-zaklady/uvod-do-mvc-architektury-ve-spring-bootu/.

[13] Suzuki, I. a Kasami, T. A Distributed Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. November 1985, zv. 3, č. 4, s. 344–349. DOI: 10.1145/6110.214406. ISSN 0734-2071.

[14] Tanenbaum, A. S. a Steen, M. V. *Distributed Systems: Principles and Paradigms*. 2. vyd. Upper Saddle River : Pearson Education, 2007. 686 s. ISBN 0-13-239227-5.

Príloha A

Obsah priloženého pamäťového média

- **app**
  - **app-src** - zdrojové súbory aplikácie
  - **javadoc** - programová dokumentácia v nástroji Javadoc
  - **distributedSystemsApp.jar** - spustiteľná aplikácia vo formáte JAR
  - **readme.txt** - súbor README, obsahujúci popis aplikácie, spôsob spustenia a použitia
- **thesis**
  - **xklimc03.pdf** - táto písomná správa vo formáte PDF
  - **thesis-src** - zdrojové súbory (LaTeX) písomnej správy

Príloha B

Adresárová štruktúra modelu aplikácie

- **/lamport**
  - **/LamportModel.java** - trieda volaná z Kontrolera po výbere Lamportovho algoritmu
  - **/LamportProcess.java** - trieda predstavujúca proces vykonávajúci Lamportov algoritmus
  - **/LamportMessage.java** - trieda predstavujúca možné správy, ktoré sú vymieňané medzi procesmi v Lamportovom algoritme
  - **/LamportCoordinates.java** - trieda definujúca správy z algoritmu už ako konkrétne hodnoty na vykresľovanie
  - **/RequestComparator.java** - trieda implementujúca komparátor, používaný na zoraďovanie správ v REQUEST QUEUE daného procesu podľa časových pečiatok, prípadne ID
- **/maekawa**
  - **/MaekawaModel.java** - trieda volaná z Kontrolera po výbere Maekawovho algoritmu
  - **/MaekawaProcess.java** - trieda predstavujúca proces vykonávajúci Maekawov algoritmus
  - **/MaekawaMessage.java** - trieda predstavujúca možné správy, ktoré sú vymieňané medzi procesmi v Maekawovom algoritme
  - **/MaekawaCoordinates.java** - trieda definujúca správy z algoritmu už ako konkrétne hodnoty na vykresľovanie
  - **/ProcessRequestComparator.java** - trieda implementujúca komparátor, používaný na zoraďovanie procesov v REQUEST QUEUE daného procesu podľa času odoslania žiadosti, prípadne ID
- **/raymond**
  - **/RaymondModel.java** - trieda volaná z Kontrolera po výbere Raymondovho algoritmu

  - **/RaymondProcess.java** - trieda predstavujúca proces vykonávajúci Raymondov algoritmus
  - **/RaymondMessage.java** - trieda predstavujúca možné správy, ktoré sú vymieňané medzi procesmi v Raymondovom algoritme
  - **/RaymondCoordinates.java** - trieda definujúca správy z algoritmu už ako konkrétne hodnoty na vykresľovanie

- **/suzuki**
  - **/SuzukiModel.java** - trieda volaná z Kontrolera po výbere algoritmu Suzuki-Kasami
  - **/SuzukiProcess.java** - trieda predstavujúca proces vykonávajúci algoritmus Suzuki-Kasami
  - **/SuzukiMessage.java** - trieda predstavujúca možné správy, ktoré sú vymieňané medzi procesmi v algoritme Suzuki-Kasami
  - **/SuzukiCoordinates.java** - trieda definujúca správy z algoritmu už ako konkrétne hodnoty na vykresľovanie