

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-5958

**URČOVANIE GENETICKÝCH PREDISPOZÍCIÍ
POMOCOU REGULÁRNYCH VÝRAZOV
DIPLOMOVÁ PRÁCA**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

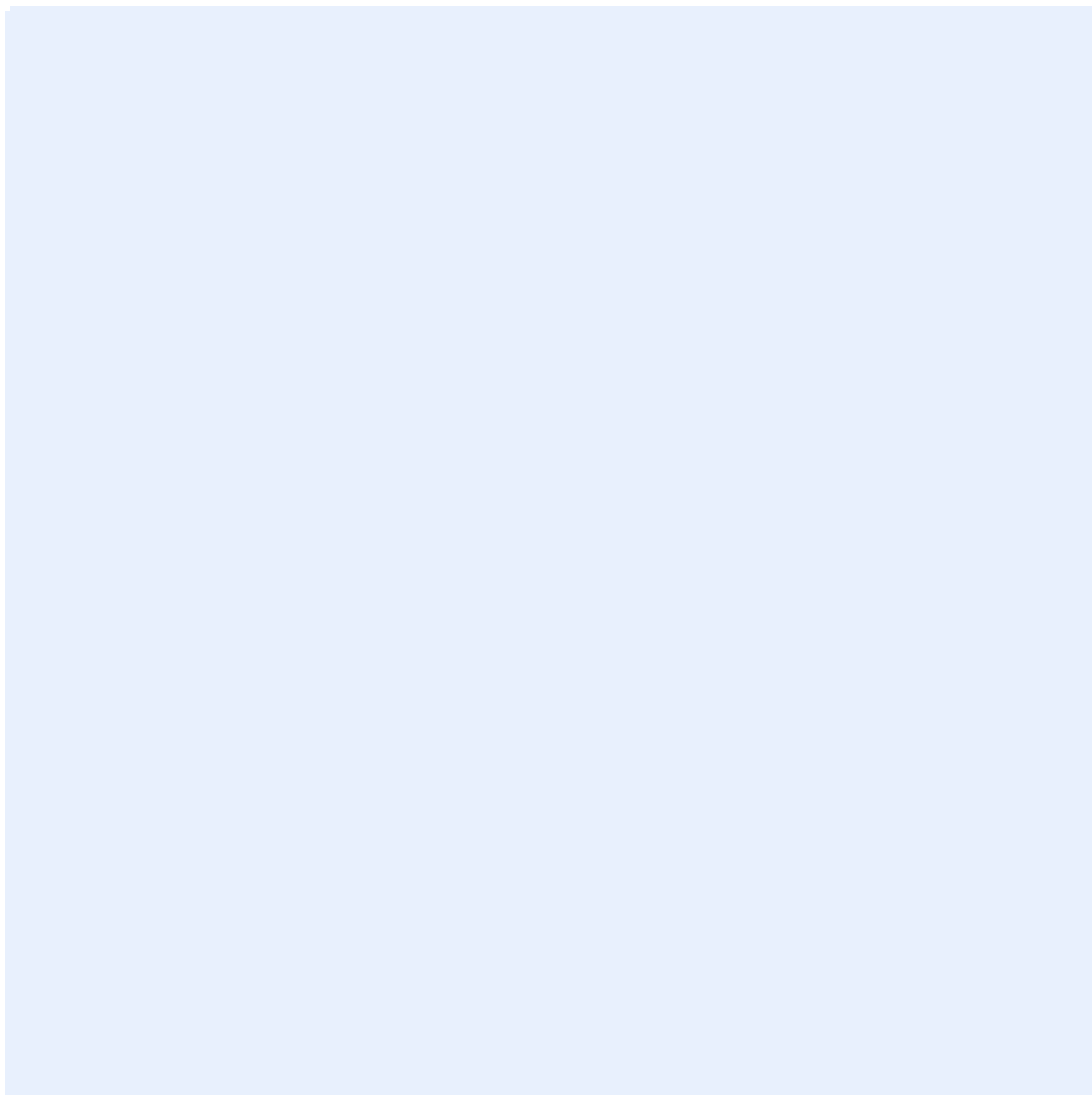
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-5958

URČOVANIE GENETICKÝCH PREDISPOZÍCIÍ POMOCOU REGULÁRNYCH VÝRAZOV DIPLOMOVÁ PRÁCA

Študijný program :	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Stanislav Marček
Konzultant:	Mgr. Zuzana Ševčíková

Sem vložte zadanie z AIS



SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program :	Aplikovaná informatika
Vyberte typ práce	Určovanie genetických predispozícií pomocou regulárnych výrazov
Autor:	Bc. Jakub Kanitra
Vedúci záverečnej práce:	Ing. Stanislav Marček
Konzultant:	Mgr. Zuzana Ševčíková
Miesto a rok predloženia práce:	Bratislava 2015

Práca poskytuje teoretické poznatky potrebné pre pochopenie základov analýzy DNA sekvencie a jej využitia pri diagnostike genetických chorôb. Popisuje návrh a implementáciu distributívneho systému slúžiaceho na analýzu týchto sekvencií. Hlavným výstupom práce je znovu použiteľný model distributívneho systému pracujúcom na základoch webových technológií a jeho vzorová implementácia na big dáta úlohe analýzy genetických sekvencií. V dobe písania práce bol tento model jedným z prvých zdokumentovaných a sprístupnených verejnosti. Práca môže byť použitá začínajúcimi bio-informatikmi ako úvod do problematiky sekvenovania a analýzy genetických sekvencií, ako aj vývojármi zaujímajúcich sa o aplikácie distributívnych systémov.

Kľúčové slová: genetická analýza, distributívny systém, node.js, regulárne výrazy

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION
TECHNOLOGY

Study Programme:	Applied Informatics
Bachelor Thesis:	Genetic predisposition analysis with regular expressions
Autor:	Bc. Jakub Kanitra
Supervisor:	Ing. Stanislav Marček
Consultant:	Mg. Zuzana Ševčíková
Place and year of submission:	Bratislava 2015

The thesis provides theoretical bases required for understanding essentials of DNA sequence analysis and its usage during diagnostic of genetic diseases. It describes design and implementation of distributed system for this type of analysis. Main output of the thesis is re-usable, web-technology based model of distributed system and its implementation. At the time of creation it was one of the first documented and published model of this kind. The thesis can be used as a starting point for bio-informatics and also developers interested in distributed systems.

Key words: genetic analysis, distributed system, node.js, regular expressions

Vyhlásenie autora

Podpísaný Bc. Jakub Kanitra čestne vyhlasujem, že som diplomovú prácu Určovanie genetických predispozícií pomocou regulárnych výrazov vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Uvedenú prácu som vypracoval pod vedením Ing. Stanislava Marčeka a Mgr. Zuzany Ševčíkovej.

V Bratislave dňa 20.05.2015

.....

podpis autora

Pod'akovanie

Ďakujem Mgr. Zuzane Ševčíkovej a Ing. Stanislavovi Marčekovi za poskytnuté rady a trpezlivosť pri tvorbe tejto Diplomovej práce.

Taktiež ďakujem B. Eichovi, J. Resigovi a R. Dahlovi za vytvorenie a rozšírenie JavaScript-u, podľa môjho názoru programovacieho jazyku budúcnosti v ktorom je napísaná celá praktická časť práce.

V neposlednom rade ďakovanie patrí všetkým autorom kníh, internetových článkov, či odpovedí na diskusných fórach, ktoré som využil a pomohli vytvoriť toto dielo.

Obsah

Úvod

1

1	Analýza problému	3
1.1	Genetika.....	3
1.1.1	Biológia bunky	4
1.1.2	DNA	5
1.1.3	Gén a mutácia.....	7
1.1.4	Projekty	7
1.2	Regulárne výrazy	8
1.2.1	Zápis	8
1.2.2	Konečný akceptor.....	10
1.2.3	Thompsonov konštrukčný algoritmus	12
1.2.4	Použitie.....	13
1.3	Distribované systémy.....	14
1.3.1	Základy.....	14
1.3.2	Výzvy	15
1.3.3	Aplikácie	15
2	Opis riešenia 17	
2.1	Špecifikácia výstupnej aplikácie.....	17
2.2	Používané technológie.....	18
2.2.1	NodeJS	19
2.2.2	PostgreSQL	22
2.2.3	Socket.io.....	23
2.3	Implementácia.....	24
2.3.1	Use-case	24
2.3.2	Javascript Distributed System Module (JDSM)	25
2.3.3	Získanie vzorov	30
2.3.4	Formát dát a využitie regulárnych výrazov	31
2.3.5	Algoritmus analýzy	32
2.3.6	Testovanie	38

Záver	40
--------------	-----------

Zoznam použitej literatúry	41
-----------------------------------	-----------

Prílohy	I
----------------	----------

Príloha A: Základný running cyklus JDSM	II
Príloha B: Diagram redistribúcie clustrov.....	II
Príloha C: Inštalačná príručka	III
Príloha D: CD nosič	IV

Zoznam obrázkov a tabuliek

Obrázok 1 Obory potrebné pre riešenie diplomovej práce	3
Obrázok 2 Schéma eukaryotickej bunky. Zdroj: (3)	4
Obrázok 3 Karyotyp človeka. Zdroj: (3)	5
Obrázok 4 Štruktúra DNA makromolekuly	6
Obrázok 5 Nedeterministický akceptor	11
Obrázok 6 Deterministický akceptor	11
Obrázok 7 Nedeterministický akceptor s ϵ -prechodmi	11
Obrázok 8 Diagram príkladu podľa Thompsonovho konštrukčného algoritmu	12
Obrázok 9 Pravidlá Thompsonovho konštrukčného algoritmu	13
Obrázok 10 Use-case diagram projektu	24
Obrázok 11 Aktivita diagram spracovania požiadaviek JDSM	27
Obrázok 12 Aktivita diagram benchmark procesu s pohľadu servera aj uzlu	29
Obrázok 13 ERA diagram výstupnej aplikácie	33
Obrázok 14 Aktivita diagram clustrovacieho algoritmu	34
Obrázok 15 Diagram hlavného behu programu	38
Obrázok 16 Aktivita diagram hlavného cyklu JDSM	II
Obrázok 17 Aktivita diagram redistribúcie clustrov	II
Tabuľka 1 Rozšírené webové technológie	19
Tabuľka 2 Používanosť DBMS k Máj-u 2015, zdroj: http://db-engines.com/en/ranking	22

Zoznam skratiek a značiek

API – Application Programming Interface
BRE – Basic Regular Expression
DBMS – Database management system
DDoS – Distributed Denial of Service
DKA – Deterministický konečný akceptor
DNA – Deoxyribonukleová kyselina
DS – Distributívny systém
ERE – Extended Regular Expression
HGP – Human Genome Project
IEEE – Institute of Electrical and Electronics Engineers
NKA – Nedeterministický konečný akceptor
POSIX – Portable Operating System Interface
RFC – Request For Comments
RNA – Ribonukleová kyselina
TKA – Thompsonov konštrukčný algoritmus
UI – User Interface
UML – Unified Modeling Language
URL – Uniform Resource Locator
VCS – Version Control System

Úvod

Genetickým poruchám a chorobám sa veľmi ťažko dá predísť a v posledných desaťročiach ich rapídne pribúda. Podľa štatistík 3-4% všetkých novorodencov trpí určitým genetickým defektom a spôsobuje 20% všetkých úmrtí novorodencov. O vážnosti týchto ochorení značí aj fakt, že 10% všetkých dospelých a 30% detských hospitalizovaných pacientov má geneticky ovplyvnené choroby. (1)

Diagnostika týchto ochorení nie je jednoduchá, no vďaka rozsiahlemu štúdiu a analýze ľudskej DNA a následnom skúmaní génov vznikajú rozsiahle databázy, ktoré opisujú gény a ich mutácie, na ich základe sa časť z týchto chorôb dá diagnostikovať v počiatočných a teda umožniť včasnú liečbu.

Rozmachu takejto diagnostiky ako jednej zo základných bráni finančná náročnosť a rýchlosť DNA sekvenátora, teda prístroja ktorý zosekvenuje DNA na sekvenciu dusíkových báz ktoré opíšeme v 1.1 . Tento prístroj je v relatívnych počiatočných (prvý vytvoril Lloyd M. Smith v roku 1987) a už teraz vidíme rapídnu evolúciu, a teda sa môže očakávať nárast rýchlosti a dostupnosti tohto prístroja.

Ďalším dôležitým faktom je výpočtová náročnosť analýzy sekvencie, keďže kompletná ľudská DNA sekvencia má približne 3 miliardy nukleotidových párov a 25 000 – 30 000 génov ich určenie vyžaduje značnú výpočtovú silu. Práve na tento fakt sa táto práca zameriava.

Cieľom práce je vytvorenie systému, ktorý by pre danú DNA sekvenciu určil genotyp jedinca a teda určil aj prítomnosť dostupných chorôb. Pre vytvorenie takéhoto systému sú potrebné základné poznatky z molekulárnej biológie a genetiky, ale aj poznatky z teórie regulárnych výrazov a ich spracovávaní napríklad konečnými automatmi, tie sú opísané v 1.2. Táto práca sa neobmedzuje iba na DNA sekvencie, jej variácia sa môže použiť napríklad ako prostriedok big data analýzy napríklad na analyzovanie veľkých dát textov.

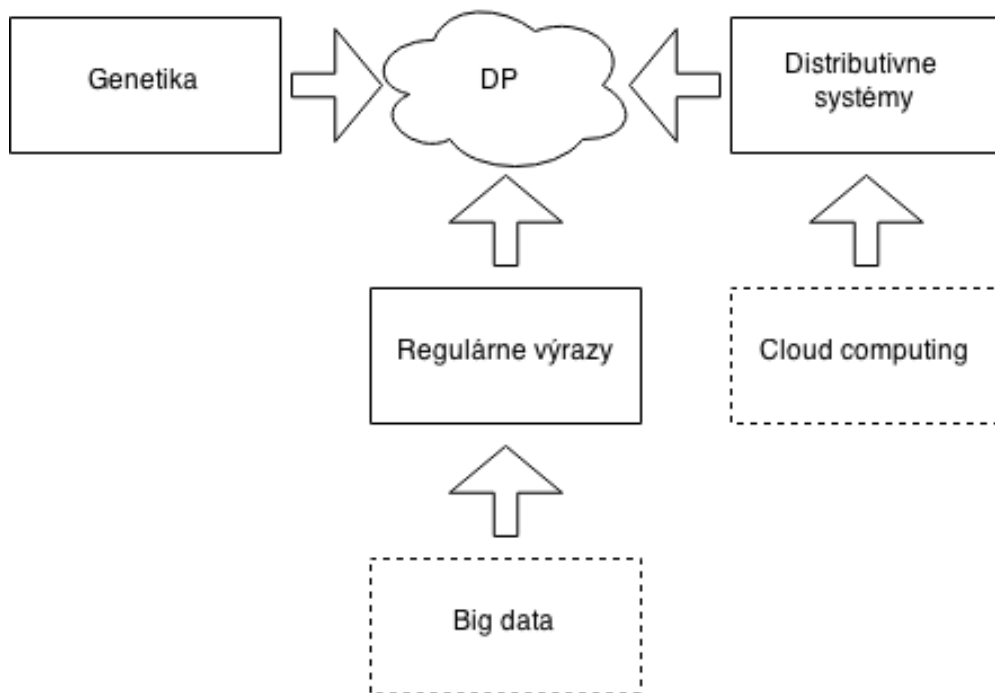
Rýchlosť a efektívnosť systému bude zabezpečovať jeho distribuovaná povaha. Tá zabezpečuje minimálne nároky na výpočtovú silu riadiaceho servera a spolieha sa na pripojené výpočtové zariadenia (uzly). Tie sú platformovo nezávislé, a teda to môžu byť tablety, smartfóny alebo počítače. Bližšie informácie o tomto type systémov je možné nájsť v 1.3. Tento druh bol vybraný pre jeho malé rozšírenie, no enormný potenciál, keďže správne navrhnutý môže byť efektívnejší ako superpočítač a takisto rozšírením dostupných

potenciálnych uzlov vďaka zrýchľovaniu zariadení a zväčšovaniu pokrytia vysokorýchlostným internetom.

Túto tému diplomovej práce sme navrhli pre oboznámenie sa s fascinujúcim a rýchlo sa rozvíjajúcim vedeckým odborom bio-informatiky. Veríme, že koncepty, algoritmy, výsledný systém a jeho podsystemy sa budú môcť uplatniť pri rôznych projektoch v budúcnosti.

1 Analýza problému

Na Obrázok 1 sú v podobe diagramu ukázané potrebné znalosti na vyriešenie problému. V podkapitolách opíšem informácie a teóriu potrebné k splneniu zadania.



Obrázok 1 Obory potrebné pre riešenie diplomovej práce

1.1 Genetika

Táto časť je venovaná ozrejmieniu potrebných znalostí ohľadom genetiky a molekulárnej biológie. Z dôvodu technického zamerania práce tu nebude spomenutá zložitá dedičnosť a replikácia jadra bunky a bunky samotnej. Tieto informácie sa dajú získať z knižných referencií (2) (3).

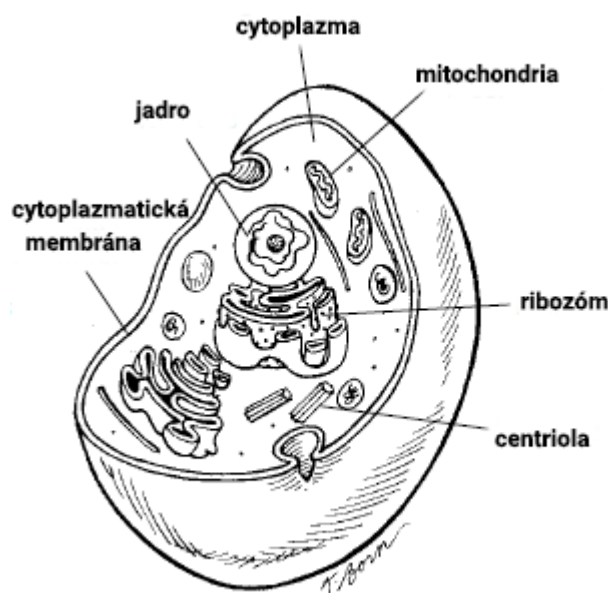
Genetika, veda o dedičnosti, je vo svojom základe štúdium biologickej informácie. Všetky živé organizmy, od jednobunkových baktérií, rastlín a zvierat, musia uchovať, replikovať a preniesť na potomkov mnoho informácií o vývoji, reprodukcii a prežití vo svojom prostredí. Genetici skúmajú ako organizmy odovzdávajú biologické informácie vo forme DNA na svojich potomkov a ako ich využívajú počas života. (2)

1.1.1 Biológia bunky

Aby čitateľ mohol pochopiť súvislosť a neuveriteľná prepracovanosť živých tvorov, je nutné aby bola opísaná základná stavebná jednotka organizmu, bunka.

Existujú 2 typy buniek určené podľa ich zloženia:

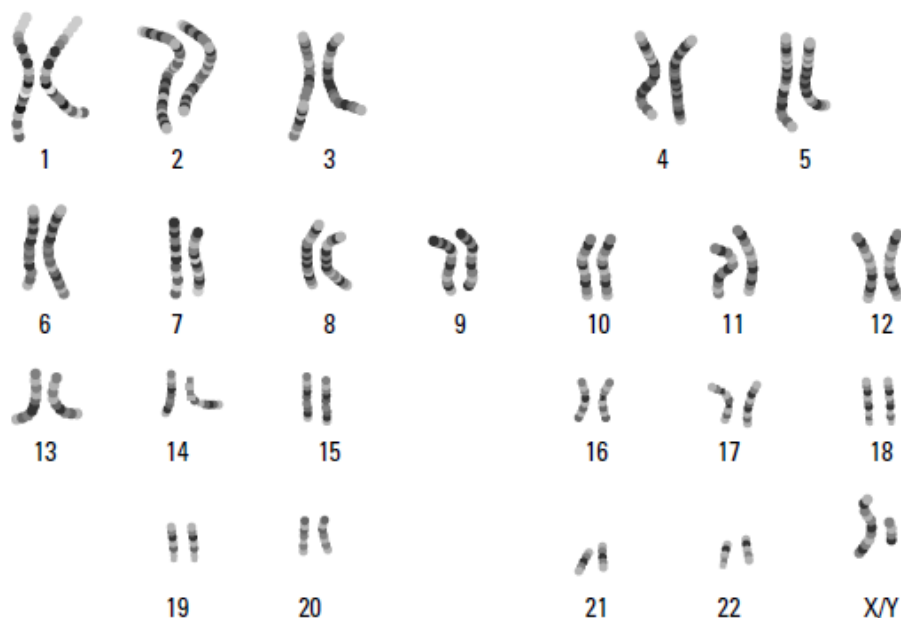
- Prokaryotické – neobsahujúce jadro, a preto sa DNA voľne pohybuje v cytoplazme bunky
- **Eukaryotické** (zobrazená na Obrázok 2) – obsahujúce jadro, DNA je pevne uložené vo vnútri a za žiadnych okolností ho neopúšťa, informácie sa prenášajú iba pomocou RNA, ľudské bunky sú tohto typu, a preto je tento typ v našom centre záujmu



Obrázok 2 Schéma eukaryotickej bunky. Zdroj: (3)

Vo vnútri jadra sa nachádza genetická informácia v podobe chromozómov. Chromozóm je stužkovitý útvar pozostávajúci z DNA a pomocných naviazaných bielkovín. Nieje viditeľný ani pomocou mikroskopu, viditeľným sa stáva iba pri procese delenia bunky (mitózy). Počet chromozómov v jadre sa líši od druhu organizmu, človek má 46 chromozómov v jadre. Tie sa delia na 22 identických párov zhodujúcich sa v tvare a dĺžke a 1 pár pohlavných chromozómov, ktoré môžu byť zhodné (XX pre ženu) alebo rozdielne (XY pre muža). Ich unikátny tvar umožňuje presné zadefinovanie poradia chromozómov, čo je veľmi výhodné, keďže môžeme sekvenciu DNA zapísať ako nepretržitý celok

v definovanom poradí. Toto poradie sa nazýva aj karyotyp organizmu. Ukážka ľudského karyotypu je na Obrázok 3.



Normal Karyotype

Obrázok 3 Karyotyp človeka. Zdroj: (3)

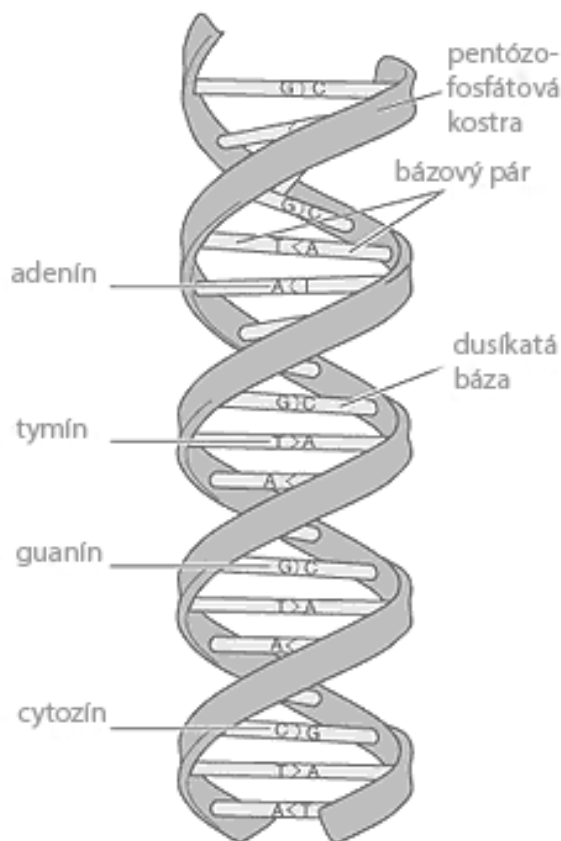
Je nutné poznamenať, že nie každý organizmus obsahuje páry chromozómov, ale iba takzvané diploidné organizmy, napríklad osy sú haploidné čo znamená, že nemajú páry, ale sú organizmy, ktoré obsahujú až šesťnásť kópií toho istého chromozómu.

1.1.2 DNA

Ako bolo spomenuté, chromozómy sa skladajú z molekúl deoxyribonukleovej kyseliny (DNA). Tieto makromolekuly sú veľmi odolné a vďaka tomu je možné ich neporušenú extrakciu zo skamenelých kostí alebo zvierat umrznutých v ľadovcoch.

Chemické zloženie DNA molekúl je pomerne jednoduché. Dusíková báza, deoxyribózový cukor a fosfát sa spojí za vzniku nukleotidu. Nukleotidy sa označujú podľa použitej dusíkovej bázy. Zistilo sa, že v celej DNA sa nachádzajú iba 4 druhy dusíkových báz, konkrétne sú to adenín, guanín, tymín a cytozín. Tie sa komplementárne dopĺňajú, a vznikajú nukleotidové páry adenínu s tymínom a cytozínu s guanínom. Tisíce takýchto

párov sa skladajú do dvojitej závitnice zobrazenej na Obrázok 4. Táto závitnica je kľúčom k ochrane a veľkej odolnosti celej molekuly.



Obrázok 4 Štruktúra DNA makromolekuly

DNA molekuly zo všetkých ľudských chromozómov obsahujú približne 3 miliardy nukleotidových párov a kebyže sa tieto závitnice rozložia do radu, bol by dlhý zhruba 185cm (3). Pre porovnanie baktérie majú zhruba 4 milióny nukleotidových párov.

Zápis celej alebo časti ľudskej DNA ako poradie nukleotidov sa nazýva **sekvenovanie**. Keďže nukleotidový pár je komplementárny stačí zápis iba jednej strany závitnice. Sú vyvinuté automatizované prístroje, ktoré používajú viacero techník na sekvencizáciu DNA, najznámejšia a najpoužívannejšia je Sangerova metóda.

Je nutné poznamenať, že zosekvenovanie celej ľudskej DNA, takzvané sekvenovanie genómu je časovo a finančne náročné. Avšak trend rapídneho klesania ceny a tým súvisiaceho času je vidno napríklad pri porovnaní ceny zosekvenovania celého genómu jednotlivca. V roku 2001 bola cena 100 miliónov dolárov no v roku 2014 to bolo menej ako 10 tisíc dolárov pri zachovaní menej ako 1% miery chybovosti, ktorá je komunitou akceptovaná (4).

1.1.3 Gén a mutácia

Gén je základná fyzická a funkčná jednotka dedičnosti. Gény môžu byť takzvané non-coding RNA a gény ktoré kódujú syntézu určitého druhu bielkoviny. Bielkoviny sú základné stavebné jednotky organizmu a zabezpečujú všetky fyzické a funkčné vlastnosti jedinca, napríklad farbu vlasov, očí, výšku no aj základné fyziologické vlastnosti ako trávenie a dýchanie. Všetky vlastnosti zákódované v DNA sa označujú ako fenotyp jedinca.

Vieme, že ľudský genotyp sa skladá z približne 21 000 génov. Tie sa líšia v dĺžke, a dosahujú až veľkosť 2,3 milióna nukleotidových párov. Niektoré vlastnosti sú ovplyvnené jediným génom (monogenické) a ovplyvnené skupinou génov (polygenické). Vedci odhadujú, že je 10 000 monogenických genetických ochorení, ako príklady uvedieme cystickú fibrózu alebo Huntingtonovu chorobu.

Mutácia je permanentná zmena nukleotidovej sekvencie v DNA. Väčšina mutácií prebehne bez postrehnutia, pretože prebehne v takzvanom „junk DNA“, teda časti sekvencie, ktoré nepatrí do žiadneho génu. No ak prebehne v génovej sekvencii, dôsledky môžu byť fatálne. Napríklad na treťom chromozóme, ktorý nesie 1000-2000 génov sa nachádza jeden ktorý zabezpečuje syntézu rhodopsínu, svetlocitlivej bielkoviny nachádzajúcej sa na sietnici. Je zaznamenaných až 30 mutácií tohto génu, ktoré ovplyvňujú korektné videnie.

Typy mutácií na sekvenčnej úrovni sú (2):

- Substitučné, nukleotidový pár sa posunie v sekvencii alebo sa obrátia jeho strany
- Odstránenie, odstránenie jedného alebo viacerých nukleotidových párov
- Vloženie, vloženie jedného alebo viacerých nukleotidových párov
- Inverzia, obrátenie poradia časti sekvencie

1.1.4 Projekty

Human Genome Project

Najväčšiemu skoku vo výskume genetiky vďačíme práve HGP (5). Začal v roku 1990 a bol označený za úspešne ukončený v roku 2003. Jeho cieľom bolo zosekvencovanie celého ľudského genómu a určenie všetkých génov. Pri začiatkoch sa predpokladalo, že existuje cca. 100 000 génov kódujúcich bielkoviny, no HGP potvrdilo, že ich je cca. 21 000 a zdokumentované ich uložilo vo verejných databázach.

Práve tento projekt odštartoval takzvanú genomickú revolúciu (5). Vytvoril 310 000 pracovných pozícií a považuje sa za jeden z najväčších vedeckých prínosov v histórii

ľudstva. Ovplyvnil mnoho technologických a vedeckých odvetví od zdravotníctva, biotechnológií, poľnohospodárstva, veterinárstva, forénznych vied a mnohých iných.

Genome Browsers

V bioinformatike je nutný rýchly a spoľahlivý prístup k biologickým databázam za účelom získania genomických dát. Za týmto účelom bolo vytvorených viacero „genome browsers“, ktoré poskytujú API alebo UI pre získanie týchto dát. Väčšina obsahuje dáta z tých istých zdrojov a líšia sa iba vo forme.

Práve tieto dáta sú potrebné pre riešenie zadaného problému tejto Diplomovej práce a budú podrobnejšie opísané v 2.3.3.

1.2 Regulárne výrazy

V Kapitole 1.1.2 bolo ukázané, že ľudská DNA a teda všetky vlastnosti jedinca sú zakódované do postupnosti 4 druhov nukleotidov, konkrétne sú to adenín (A), cytozín (C), tymín (T) a guanín (G). Preto sa ľudská DNA dá zapísať ako postupnosť týchto 4 znakov o veľkosti tri miliardy. V kapitole 1.1.3 zas boli opísané gény a mutácie vo vzťahu práve k tejto postupnosti. Z týchto poznatkov môžeme vyvodiť, že určenie génu je vlastne zistenie prítomnosti daného vzoru v reťazci. Tento vzor môže zakomponovávať rôzne variácie reťazca, v tejto implementácii sú to mutácie. Práve na takýto účel boli vytvorené regulárne výrazy.

„Regulárny výraz je zápis popisujúci množinu znakových reťazcov. Keď konkrétny reťazec je v množine popísanej regulárnym výrazom, tak sa hovorí, že reťazec vyhovuje vzoru.“ (6) Je široko používaný v teoretickej počítačovej vede a teórií formálnych výrazov.

1.2.1 Zápis

V najjednoduchšej forme môže regulárny výraz definovať konkrétne slovo alebo znak, no môže definovať aj zložité vzory napríklad validnú e-mailovú adresu, telefónne číslo, dátum alebo aj cystickú fibrózu v ľudskej DNA.

Časť IEEE POSIX štandardu, BRE (basic regular expression) a ERE (extended regular expression) zjednocuje zápis regulárnych výrazov. Využíva konvenčnú znakovú sadu a definuje metaznaky s kontrolnými funkciami.

Takýmito metaznakmi sú:

- . – ľubovoľný znak
- [] - označuje ľubovoľný znak z množiny vo vnútri, napríklad výraz [abc] značí prítomnosť znaku a alebo b alebo c, môže sa použiť aj skrátený zápis [a-c]
- [^] – negácia množiny vo vnútri zátvoriek
- ^ - začiatok textu alebo riadku
- \$ - koniec textu alebo riadku
- () – definovanie podvýrazu
- \n – vloženie n-tého podvýrazu definovaného spôsobom popísaným vyššie
- * - označuje výskyt predchádzajúceho elementu nula alebo viackrát, môže sa použiť s podvýrazom, napríklad (abc)* vyhovuje “”, “abc”, “abcabc” atď.
- {m, n} – označuje výskyt predchádzajúceho elementu minimálne m-krát vrátane a maximálne n-krát vrátane, napríklad a{2,3} vyhovuje slovám “aa” alebo “aaa”
- ? – označuje výskyt predošlého elementu nula alebo jedenkrát
- + – označuje výskyt predošlého elementu jeden alebo viackrát
- | – logické alebo

V praxi sa používajú aj takzvané **znakové triedy**. Existuje viacero znakových tried, napríklad malé písmená, všetky alfanumerické znaky, číselné znaky a ďalšie. Zápis týchto tried nie je ustálený a preto tu spomeniem iba najpoužívanejšiu implementáciu pomocou ASCII znakov najdôležitejších tried. Sú to:

- [a-z] – malé písmená
- [A-Z] – veľké písmená
- [0-9] – číselné znaky
- [A-Za-z0-9] – všetky alfanumerické znaky
- [A-Fa-f0-9] – znaky hexadecimálneho čísla

Je nutné poznamenať, že regulárne výrazy nie sú obmedzené iba na ASCII znaky a môžu sa použiť aj znakové triedy unicode znakov použitím /u a unicode kódu znaku. Napríklad [/u00C1-/u01C4] obsahuje všetky špeciálne slovenské písmená.

Príklad

Demonštrácia zostrojenia regulárneho výrazu je možná na príklade overenia korektnosti zadanej e-mailovej adresy.

Štandardizovaná syntax emailovej adresy je definovaná v RFC 5322. Keďže emailová adresa je case-insensitive, je možné celý vstup pretransformovať na malé písmená. Syntax podľa RFC 5322 sa môže definovať ako *local-part@domain*, kde doménová časť musí obsahovať minimálne dve doménové úrovne a musí sa ukončiť top-level doménom, ktorá sa skladá z dvoch až šiestich ASCII znakov.

Najintuitívnejší výraz, zabezpečujúci validitu e-mailovej adresy by mohol byť:
$$^{[a-z0-9.] +}@[a-z0-9.] +.[a-z]{2,6}\$$$

Tento výraz má zopár nedostatkov, prvým je nepodporovanie všetkých povolených znakov aj keď všetky sa v praxi nepoužívajú. Dokonca emailová služba spoločnosti google nepodporuje všetky znaky a lokálna časť sa môže skladať iba z $[a-z0-9_']$.

Takisto adresa sa nemôže začínať ani končiť bodkov a nemôžu sa v nej nachádzať za sebou stojace bodky.

Kompletný regulárny výraz napísaný v javascriptovej syntaxe (7):
$$^{[\w !\# \$ \% \& ' * + = ? \{ | \} \sim ^ -] + (? : \. [\w !\# \$ \% \& ' * + = ? \{ | \} \sim ^ -] +) * @ (? : [A - Z 0 - 9 -] + \.) + [A - Z] { 2 , 6 } \$$$

1.2.2 Konečný akceptor

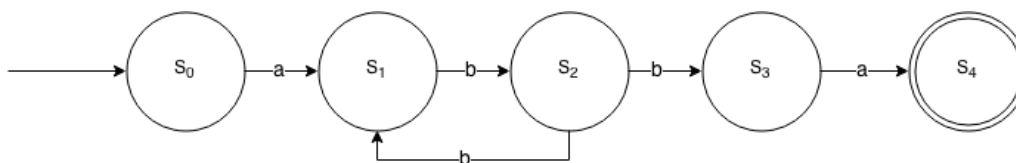
Pre nájdenie najefektívnejšieho spôsobu vytvorenia a vyhodnotenia regulárnych výrazov je nutné oboznámiť sa so základmi teórie automatov. Táto teória je jedna zo základných častí teoretickej počítačovej vedy a diskkrétnej matematiky.

Automat je abstraktný model stroja, ktorý vykonáva spracovanie vstupu pomocou pohybu po stavoch alebo konfiguráciách. V každom stave prechodová funkcia určí nasledujúci stav alebo konfiguráciu z konečnej množiny stavov a konfigurácií. (8)

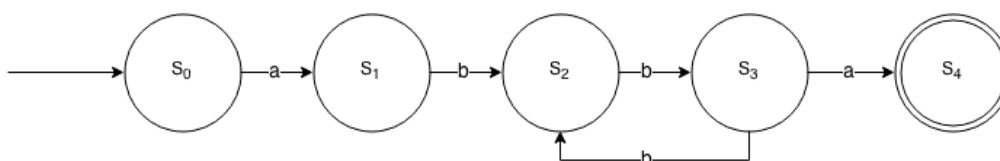
Najkomplexnejšou aplikáciou teórie automatov je Turingov stroj, hypotetický automat manipulujúci so znakovou páskou a množinou pravidiel. Logika ľubovoľného počítačového algoritmu sa dá popísať Turingovým strojom.

Z rozsiahlej oblasti teórie automatov sa oboznámime s konečnými akceptormi. Konečný akceptor sa dá znázorniť ako orientovaný graf, nazývaný stavový diagram, kde miesta znázorňujú stavy a orientované hrany ohodnotené vstupným znakom abecedy definujú

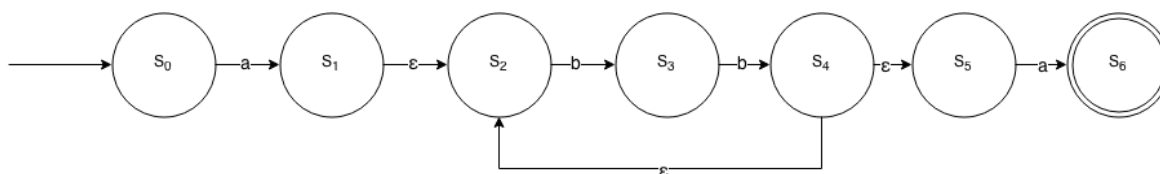
prechodovú funkciu δ . Miesto do ktorého vedie prázdna šípka je počiatočný stav a miesta označené dvojítm krúžkom definujú množinu koncových stavov. Príklady stavových diagramov konečných akceptorov akceptujúcich regulárny výraz $a(bb)^+a$ sú na **Chyba! enašiel sa žiaden zdroj odkazov. , Chyba! Nenašiel sa žiaden zdroj odkazov. a Chyba! Nenašiel sa žiaden zdroj odkazov..**



Obrázok 5 Nedeterministický akceptor



Obrázok 6 Deterministický akceptor



Obrázok 7 Nedeterministický akceptor s ϵ -prechodmi

Keď nový stav je určený súčasným stavom a vstupom, potom je akceptor deterministický (DKA), zo stavového diagramu sa determinizmus dá určiť, že zo žiadneho stavu nevedú dve a viac šípok ohodnotených rovnakým vstupom. Nedeterministický akceptor (NKA) takéto obmedzenie nemá. Existuje forma zápisu nedeterministického automatu pomocou takzvaných ϵ -prechodov. ϵ -prechod dáva ešte väčšiu voľnosť pri zobrazovaní, pretože jeho použitie nieje podmienené vstupom.

Každý NKA sa dá vyjadriť pomocou DKA, ktorý rozpoznáva rovnaký formálny jazyk. DKA je značne rozsiahlejší (ak NKA má n stavov, DKA ekvivalent môže dosahovať až 2^n stavov). Na konverziu sa môže použiť *Rabin-Scott powerset construction* algoritmus.

Testovanie pomocou je DKA je intuitívnejšie, no výpočtové systémy vďaka schopnosti rekurzii dobre pracujú aj s NKA. Bližšie je tento proces popísaný v časti 1.2.3.

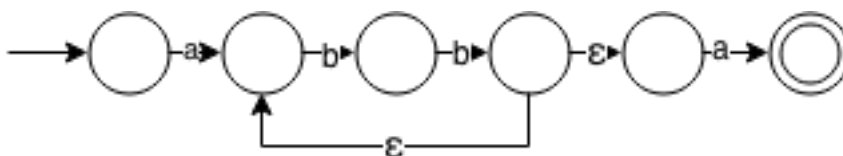
Nedeterministický konečný akceptor (NKA) je konečný automat bez výstupnej funkcie a môže byť popísaný päticou: $SM = (S, E, \delta, s_0, A)$, kde (9):

- S je množina stavov
- E je množina vstupných znakov (vstupná abeceda)
- $\delta: S \times E \rightarrow 2^S$ je prechodová funkcia priradujúca stavu a vstupu množinu nasledujúcich stavov
- $s_0 \in S$ je počiatočný stav
- $A \subseteq S$ je množina koncových stavov

1.2.3 Thompsonov konštrukčný algoritmus

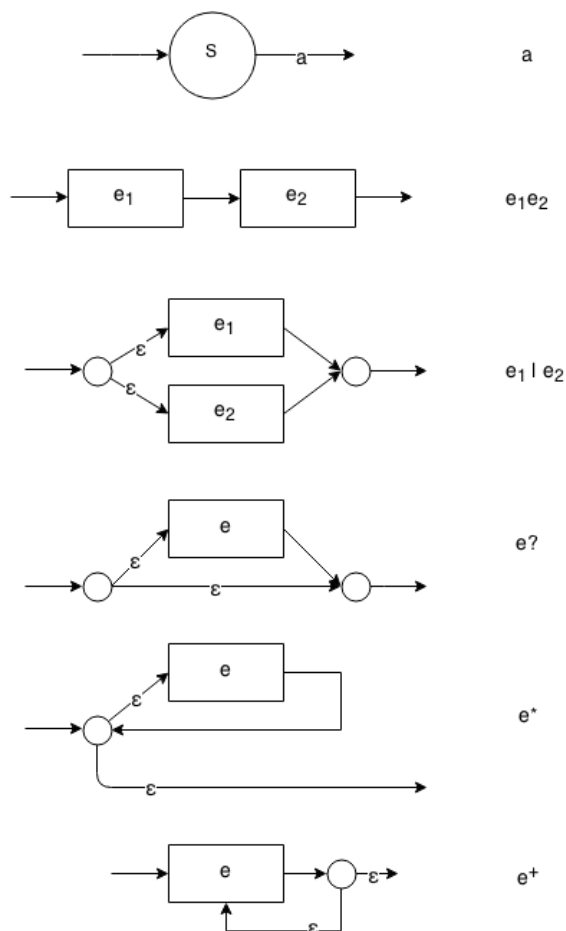
V predošlej časti bolo ukázané, že ľubovoľný regulárny výraz generujúci regulárny jazyk sa dá vyjadriť pomocou nedeterministického konečného akceptora. Existuje viacero algoritmov ako takýto akceptor zostrojiť. Jeden z najpoužívanejších je Thompsonov konštrukčný algoritmus, publikovaný Kenom Thompsonom v roku 1968.

Algoritmus je založený na rekurzívnom rozklade výrazu na jeho podvýrazy až na elementárne výrazy. Tie sa zapisujú pravidlami znázornenými na **Chyba! Nenašiel sa žiaden zdroj odkazov..** Kde e je blok pozostávajúci z ďalších blokov alebo z elementárnych výrazov. Výraz z 1.2.2 po vytvorení pomocou TKA je na **Chyba! Nenašiel sa žiaden zdroj odkazov..**



Obrázok 8 Diagram príkladu podľa Thompsonovho konštrukčného algoritmu

TKA vytvára ϵ -prechody, ktoré sú výhodné pre spracovanie na počítačoch, vďaka novej rekurzii. Vždy keď sa narazí na stav z ktorého vychádzajú ϵ -prechody dôjde k rekurzívnemu rozvetveniu.



Obrázok 9 Pravidlá Thompsonového konštrukčného algoritmu

1.2.4 Použitie

S regulárnymi výrazmi sa človek stretáva každodenne, či je to vyhľadávanie textu na webovej stránke alebo v dokumente, alebo parsovanie html stránky prehľadávačom a mnohých iných.

Regulárne výrazy sa taktiež používajú ako jeden zo základných kameňov big data analýzy, ktorá sa v posledných rokoch stáva dominantným informačným artiklom. Či už je to určovanie trendov zo sociálnych sietí ako twitter alebo facebook, alebo ochrana pred kriminálnymi činnosťami analyzovaním komunikačných sietí bezpečnostnými úradmi. Taktiež väčšina takzvaných NO-SQL databázových systémov využíva prednosti regulárnych výrazov.

1.3 Distribuované systémy

Jedným z výstupov tejto diplomovej práce je vytvorenie rozhrania distribuovaného systému (DS), ktoré je možné implementovať na rôzne výpočtové úlohy a problémy. Aby bolo toto rozhranie možné navrhnuť a implementovať je nutné oboznámiť sa s distribuovanými výpočtami ako oborom počítačovej vedy, ktorý sa zaoberá práve štúdiom DS.

1.3.1 Základy

„Distribuovaný systém je kolekcia nezávislých počítačov, ktoré sa javia používateľom systému ako jeden počítač.“ (10)

Motivácia k vytváraniu a používaniu distribuovaných systémov leží v túžbe zdieľania zdrojov. Zdroj je abstraktný pojem a charakterizuje všetky veci, ktoré môžu byť zdieľané v počítačovom systéme, môžu to byť hardwarové komponenty ako procesor, pamäť, grafická karta, harddisk alebo tlačiareň až po softvérové entity ako súbory, zložky, databázy a iné dátové objekty. Taktiež sa môžu zdieľať dáta z kamier, napríklad dopravných.

S mnohými variáciami týchto systémov sa človek stretáva denne, či už sú to virtuálne decentralizované meny ako bitcoin alebo lifecoin, peer-to-peer aplikácie, multiplayerové hry ale najfundamentálnejším príkladom je internet. Bohužiaľ, výpočtová sila týchto systémov poskytla mocný nástroj záškodníckym činnostiam v podobe DDoS.

DDoS (Distributed Denial of Service) je útok na server alebo cluster serverov pomocou zasielania požiadaviek (request-ov) všetkými uzlami distribuovaného systému. Sila takýchto útokov bola použitá v júli 2009 (11), kedy systém o veľkosti 166 000 počítačov znefunkčnilo viacero systémov, medzi ktorými boli stránky Pentagonu, Bieleho domu a ďalších.

V roku 2017 sa podľa agentúry Gartner (12) predá 2.9 miliárd kusov výpočtovej techniky v podobe PC, notebookov, tabletov a smartfónov. Z toho sa dá dedukovať, že každý obyvateľ technologicky vyspelej krajiny disponuje nejakým výpočtovým zariadením, ktoré po väčšinu dňa nevyužíva. Prečo by sa tieto zariadenia nemohli použiť ako výpočtové jednotky distribuovaných systémov a tým pádom znížili dopyt po stále nových a nových zariadeniach, čo má negatívny dopad na ekonomiku jedinca a taktiež negatívny dopad na životné prostredie? Zvlášť keď na prilákanie a udržanie používateľov sa môže použiť monetárna odmena alebo stále viac a viac rozšírená **gamifikácia**.

Gamifikácia je použitie herných elementov a mechaník na zinteraktívnenie používateľovej návštevy aplikácie. V posledných rokoch sa dostáva do popredia a začína sa študovať na prestížnych univerzitách ako University of Pennsylvania¹ a je používaná na portáloch ako Yahoo! Answers, Stack Overflow, či výukových portáloch ako codeschool, no žiadny distributívny systém túto techniku neimplementoval na získanie a udržanie výpočtových uzlov.

1.3.2 Výzvy

Distribúované systémy sú tak rozsiahle a môžu sa nimi implementovať jednoduché úlohy ako posielanie správ ale aj najkomplexnejšie systémy ako internet alebo výpočet predpovede počasia.

Pri návrhu a implementácii ľubovoľného distribúovaného systému sa podľa Coulourisa (13) naskytuje sedem hlavných výziev, ktoré architekt systému musí brať do úvahy:

- Heterogenita – prístup k systému z rôznych zariadení, výrobcov a pripojení
- Otvorenosť – úroveň možnosti rozšírenia a reimplementácie
- Bezpečnosť – zdieľanie a preposielanie citlivých údajov musí byť chránené pred útokmi, taktiež systém musí byť zabezpečený voči útokom ako DDoS
- Škálovateľnosť – teoretická neobmedzenosť možných používateľov systému
- Spracovanie chýb – je nutné brať do úvahy rôzne chyby, ktoré sa môžu vyskytnúť počas chodu, napríklad odpojenie uzlu od siete, chyba harddisku a mnoho ďalších
- Konkurencia – prístup 2 a viacerých používateľov k rovnakému zdroju nemôže mať za následok nekonzistenciu systému
- Priehl'adnosť – používateľovi sa systém javí ako celok, netuší nič o jeho distribúovanej povahe

Návrhu konkrétneho distribúovaného systému sa budem venovať v 2.3.

1.3.3 Aplikácie

Existuje viacero distribúovaných výpočtových systémov postavených na fakte, že ľudia poskytujú výpočtový výkon svojich počítačov za pocit vedomia, že sa podieľajú na

¹ <https://www.coursera.org/course/gamification>

spoločensko-vedecky prospešných projektoch. Verím, že popularitu takýchto systémov by zvýšil projekt, ktorý by finančne motivoval používateľov k zdieľaniu ich výkonu.

Najúspešnejší projekt je podrobnejšie rozpísaný nižšie, no za zmienku stoja aj **MilkyWay@home** generujúci presný trojrozmerný dynamický model galaxie Mliečnej cesty, ktorý má priemerný výpočtový výkon 573 TFLOPS a 27 000 aktívnych používateľov a **GIMPS** (Great Internet Mersenne Prime Search) hľadajúci Mersenove prvočísla, ktoré sú definované vzťahom $M_n = 2^n - 1$.

Einstein@home

Tento najúspešnejší dobrovoľnícky projekt, ktorý využíva distribuovaný výpočtový systém hľadá slabé astrofyzické signály z rotujúcich neutrónových hviezd (pulzarov) s použitím dát z LIGO gravitačno-vlnových detektorov, rádiového teleskopu Arecibo a satelitu na detekciu gama žiarenia Fermi. Projekt od svojho počiatku v roku 2005 detkoval 36 neutrónových hviezd a jeho cieľom je potvrdenie existencie gravitačných vln emitovaných neutrónovými hviezdami. Tieto vlny predpovedal Albert Einstein, no ešte neboli nikdy priamo detekované.

Priemerná výpočtová sila tohto distribuovaného systému je 470 TFLOPS (14), čím by sa mohol zaradiť medzi prvých 500 superpočítačov na svete.

2 Opis riešenia

Táto kapitola opisuje implementačné a technologické riešenie softvérového výstupu diplomovej práce. V stručnosti odôvodní vybrané technológie, ich charakteristiky, klady a zápory, a objasní základné princípy a algoritmy implementovaného systému. Na ich ilustráciu sú použité UML diagramy.

2.1 Špecifikácia výstupnej aplikácie

Ako bolo spomenuté v 1.1 všetky fyzické vlastnosti jedinca sa ukrývajú v jeho DNA. Človek má 23 chromozómov ukrytých v jadre skoro každej bunky jeho tela. Každý chromozóm sa skladá z dvojzávitnice poskladanej z nukleotidových párov, pozostávajúcich z cytozínu (C), guanínu (G), tymínu (T) a adenínu (A). Dĺžka jednotlivých chromozómov je rôzna, kde najkratším je pohlavný chromozóm Y, ktorý obsahuje cca 50 miliónov nukleotidových párov, naopak najdlhším je chromozóm 1 s 250 miliónmi nukleotidových párov.

Gén je časť DNA sekvencie, ktorá kóduje produkciu konkrétnej bielkoviny a teda fyzického znaku jedinca. Gény sú rôzne, od génu určujúceho počet prstov až po gén krvného typu. Zadefinovanie a popísanie všetkých ľudských génov mal na starosti projekt HUGO prebiehajúci v rokoch 1990 až 2003, jeho výsledkom bolo nájdenie približne 21 000 génov. Nanešťastie niektoré gény spôsobujú genetické choroby ako napríklad rakovinu alebo Huntingtonovu chorobu, práve takéto choroby je potrebné diagnostikovať analýzou genetickej sekvencie jedinca.

Gény (v koncepte aplikácie nazývané vzory) sú voľne dostupné z genómových prehliadačov spomenutých v 1.1.4. V aplikácii môžeme vybrať prehliadač <http://www.ensembl.org/> kvôli dobre zdokumentovanému aplikačnému rozhraniu a REST API servisu. Každý vzor sa skladá z chromozómu na ktorom sa nachádza, počiatočnej nukleotidovej pozície a nukleotidovej sekvencie.

Vzorka, teda zosekvenovaná časť alebo celá DNA, môže obsahovať niekoľko stoviek až miliárd nukleotidových párov uložených v textovom súbore. Pôvod týchto dát nie je v záujme aplikácie, jediná podmienka je splnenie formátu definovaného v 2.3.

Úlohou aplikácie je pre danú vzorku určiť definovateľné (nachádzajúce sa na vzorke) vzory a ich pozitívne alebo negatívne určenie.

Keďže vzorov môže byť niekoľko desiatok tisíc a vzorky môžu dosahovať veľkosti niekoľko GB, táto úloha spadá do big data problematiky. Keď zoberieme 20 000 vzorov s priemernou sekvenčnou veľkosťou 1MB, tak stojíme pred rozhodnutím využiť 20GB operačnej pamäte, ktorá je pri virtuálnych serveroch veľmi nákladná alebo ad hoc načítavaním vzorov z databázy, čo vedie k výraznému zhoršeniu výkonnosti celého systému, pretože I/O operácie sú časovo najnákladnejšie. Alebo využijeme distributívny systém s N pripojenými uzlami a pamäťová náročnosť sa rovnomerne prerozdeli a server bude použitý ako radič. Aj napriek nespornej nevýhode, že pri malom počte vzorov a malom počte pripojených uzlov bude výkon systému výrazne horší ako pri použití klasickej klient-server štruktúry implementujeme systém ako distributívny.

Éra klasických, platformovo závislých aplikácií je vzostupom webových technológií navyď preč a či už je to mobil, tablet, notebook alebo stolový počítač, každý má umožnený prístup na internet a prehliadač schopný behu javascriptových aplikácií, no na trhu nie je distributívny systém založený na tejto technológii, kde sa uzol pripojí jednoduchým otvorením webovej stránky a okamžite poskytne výpočtový výkon. Práve preto, aplikáciu navrhne a implementujeme ako webovú, pomocou webových technológií.

2.2 Použité technológie

Výber správnych technológií je kľúčový pre efektívne navrhnutie a implementáciu systému. V podkapitolách sú opísané hlavné technológie a odôvodnenie ich použitia práve pre charakteristiky a povahu žiadaného systému.

Za zmienku stojí, že pri tvorbe projektu bol použitý voľne dostupný, open-source distribuovaný systém riadenia verzií (VCS) GIT, vytvorený Linus-om Torvalds-om na zefektívnenie vývoja Linux-ového jadra (15). Vďaka nemu je možné bezpečné uchovávanie a spravovanie zmien v projekte, umožňuje kooperáciu stoviek programátorov, a v neposlednom rade, projekty tvorené za pomoci git-u je možné uložiť a sprístupniť na vzdialených serveroch, ktoré môžu byť privátne alebo verejné. Najznámejšie a najpoužívanéjšie verejné git vzdialené cloud služby sú <http://www.bitbucket.org> a <http://www.github.com>.

Dokumentácia zdrojového kódu je jedným z kľúčových faktorov správneho programovania, ktorá uľahčuje kooperáciu programátorov na projekte ako aj zefektívňuje jeho prehľadnosť, čitateľnosť a výrazne redukuje čas strávený pri debugovaní a orientovaní

sa v kóde. Projekt bol dokumentovaný podľa pravidiel YUIDoc syntaxy (yui.github.io/yuidoc/), ktorá umožňuje jednoduché vygenerovanie komplexnej API dokumentácie.

2.2.1 NodeJS

Najpodstatnejším rozhodnutím je zvolenie hlavnej serverovej technológie. V dnešnej dobe je na výber veľké množstvo dostupných serverových riešení, ktoré podporujú vytvorenie komplexných webových aplikácií. Ich súhrn je zobrazený na Tabuľka 1.

Jazyk	Frameworky	Rok vzniku	Rozšírenie
PHP	Laravel, Yii	1995	82.0%
ASP.NET		2007	16.9%
Java	Swing, Spring	1995	2.9%
Ruby	Ruby on Rails	1995	0.6%
Javascript (Node.js)	ExpressJS, SailsJS	2009	0.2%

Tabuľka 1 Rozšírené webové technológie²

Aby sme vybrali správne, je nutné spísať vlastnosti navrhovaného systému. Distribuovaný systém sa spolieha predovšetkým na veľmi aktívnu sieťovú komunikáciu s komunikáciou s množstvom pripojených uzlov a spracovanie big dát DNA sekvencie zas vyžaduje odosielanie veľkých objemov dát. Taktiež systém vyžaduje čítanie veľkých dát z databázy prípadne súborov. Práve z týchto dôvodov môžeme vylúčiť skriptovací jazyk PHP, ktorý vďaka svojej povahe nie je vhodný na náročné input/output operácie.

Ruby bez použitia frameworku Ruby on Rails má veľmi obmedzenú funkčnosť pre tvorenie webového obsahu, no s jeho použitím sa značne degraduje jeho výkon.

Ako hlavnú technológiu som vybral platformu **Node.JS** <https://nodejs.org/>. Aplikácie do Node sa píše v javascripte, no jeho jadro je napísané prevažne v C++ a Javascripte. Je postavený na V8 javascript engine od spoločnosti Google, ktorý používa najrozšírenejší webový prehliadač Chrome.

² Dáta ku 9.5.2015, zdroj: http://w3techs.com/technologies/overview/programming_language/all)

„Aby sme porozumeli čím sa Node odlišuje, mali by sme ho porovnať s Apachom, populárnym webovým serverom. Najprv, Apache spracuje HTTP požiadavky a prenecháva aplikačnú logiku, aby bola implementovaná v jazyku akým je PHP alebo Java. Node odstraňuje túto úroveň komplexnosti skombinovaním serverovej a aplikačnej logiky. Tento prístup dáva Node nevídanú flexibilitu ako serveru.“ (16)

Node, na rozdiel od klasických serverov, ktoré bežia na viacerých vláknach, ktorých správa je zdrojovo náročná, beží iba na jednom vlákne. To bolo dosiahnuteľné využitím neblokovacích I/O požiadaviek, vlákno odošle požiadavku a pokračuje v činnosti a po vykonaní požiadavky sa spustí asynchrónna callback funkcia, ktorá spracuje dáta získané požiadavkou. Príkladom neblokovanej asynchrónnej IO operácie je čítanie súboru, ktorý využije vstavaný node modul fs (file system), pošle požiadavku na prečítanie súboru a po jeho vykonaní, ak nenastala žiadna chyba vypíše obsah. Funkcia require() je štandardnou funkciou na načítanie modulov. V node každý súbor predstavuje jeden modul a vice versa.

```
var fs = require('fs');
fs.readFile('./dummy.txt', function(err, text){
  if(!err)
    console.log(text);
});
```

Neposledná výhoda Node je, že vývojár môže využiť znalosť silného jazyku **javascript**, ktorý je najpoužívanejším jazykom na github.com (ku 10.5.2015 <http://github.info/>). Javascript obsahuje mnoho konceptov, ktoré chýbajú v klasických jazykoch ako napríklad closures, first-class functions, prototypovú dedičnosť a iné. (17).

„Mnoho významných spoločností, vrátane Microsoft-u, LinkedIn-u a Yahoo! si uvedomili výhody Node a začali v ňom implementovať svoje projekty. Napríklad, LinkedIn remigroval svoj celý mobilný stack do Node a prešli z pätnástich serverov s pätnástimi inštanciami na každom na štyri inštancie, ktoré zvládnu dvojnásobnú záťaž.“ (16)

Pre Node bolo napísaných mnoho framework-ov, knižníc a middlewar-u, no základným pilierom je webový framework **Express.js**. Ten poskytuje model-view-controller (MVC) štruktúru, organizáciu smerovania, manažment relácií, extrakciu URL parametrov a mnoho iných základných potrieb, ktoré by sme museli implementovať sami, avšak použitím tohto frameworku máme k dispozícii otestovaný a komunitou spravovaný nástroj, ktorý je možno bezpečne použiť. (18)

Doba, keď spoločnosť alebo sám programátor používal iba svoje knižnice a kódy je našťastie za nami a komunita vývojárov prešla do open-source éry, kde každý zdieľa svoje znalosti a svoje projekty. Vďaka tomuto trendu vznikol **Node package manager (NPM)** <https://www.npmjs.com/>, čo je centralizovaný repozitár, všetkých registrovaných knižníc, frameworkov a middlewar-ov napísaných pre node. Jeho popularitu dokazuje fakt, že sa na ňom nachádza 147 054 projektov a stiahne sa z neho približne 50 miliónov kópií projektov denne.³ Zaregistrovanie projektu do npm je zdarma a neprechádza žiadnou kontrolou, preto sa publikované projekty líšia rozsahom aj kvalitou a treba si dať veľký pozor, ktoré zaintegrujeme do vlastnej aplikácie.

Za zmienku stoja dva npm moduly, ktoré som použil pri tvorbe výslednej aplikácie, oba patria k najšťahovanejším a najpoužívanejším modulom⁴ pre Node platformu a to zabezpečuje ich vysokú kvalitu a aktívnu komunitu stojacu za ich vývojom a testovaním.

Prvým je **browserify**⁵, ako bolo už povedané výhodou Node je použitie programovacieho jazyku JavaScript na serverovej ako aj klientskej strane, tento fakt zaručuje, že moduly nepoužívajúce natívne Node moduly a middleware sú použiteľné v serverovej aj klientskej aplikačnej logike, avšak organizácia a načítanie modulov v node je zabezpečená zavolaním synchrónnej funkcie `require()`, ktorá nie je rozoznateľná prehliadačom, a preto bol vytvorený modul `browserify`, ktorý zabezpečí efektívne načítanie modulov na front-ende. Vďaka tejto skutočnosti môže vývojár používať jednotnú štruktúru aplikácie a nie je potrebné rozlišovať serverové a klientske skripty. Vo výstupnom projekte sa `browserify` využil na zjednotenie napísania klientskeho skriptu, ktorý sa vykonáva na strane jednotlivých uzlov a serverovej strany, ktorá riadi jednotlivé uzly.

Grunt⁶ je najpoužívanejším manažérom úloh pre JavaScript a slovami tvorcov, slúži na automatizáciu, uľahčuje spúšťanie opakujúcich sa úloh ako minifikáciu, kompiláciu a testovanie, v dobe písania práce je dostupných 4662 pluginov, ktoré pokrývajú všetky časti vývoja web aplikácie od základnej kompilácie CoffeeScript-ových a SASS súborov, cez automatické obnovenie prehliadača po zmene súborov až po komplexné skripty pre nasadenie aplikácie na produkčný server. V práci bol využitý grunt na kompiláciu SASS súborov, automatické obnovovanie prehľadovej stránky, v tandeme s `browserify` na

³ Ku 10.5.2015, zdroj: <https://www.npmjs.com/>

⁴ Ku 10.5.2015, zdroj: <https://www.npmjs.com/>

⁵ <https://www.npmjs.com/package/browserify>

⁶ <http://gruntjs.com/>

generovanie klientskeho skriptu pri každej zmene klientských súborov a na vygenerovanie rozsiahlej dokumentácie pomocou YUIDoc. Pre grunt je možné napísanie vlastných úloh, ktoré sú spustiteľné pomocou konzolových príkazov, túto možnosť som použil na napísanie základných úloh uľahčujúcich testovanie aplikácie, akými sú napríklad vygenerovanie dát, vymazanie dát z databázovej tabuľky a iné. Kompletný zoznam dostupných príkazov je spísaný v dokumentácii projektu.

2.2.2 PostgreSQL

Dôležitou súčasťou vybrania kvalitného technologického stacku pre aplikáciu je výber databázového systému (DBMS). V posledných rokoch sa rozšírili NoSQL DBMS ako Redis, MongoDB a iné. Na rozdiel od klasických, relačných systémov neukladajú dáta v tabuľkách ale v dokumentoch alebo jednoduchých key-value pároch. Výhody sú nesporné hlavne v rýchlosti a ľahšiemu rozdeleniu na viacero serverov rozložených po celom svete a väčšina softvérových gigantov sa neubránila kompletnému alebo čiastočnému premigrovaniu ich databázového stacku do NoSQL. Avšak výstupná aplikácia diplomovej práce nevyužíva databázový systém vo veľkom množstve a preto nie je výhodné experimentovanie s novou technológiou s ktorou nemáme skúsenosti.

Najpoužívanejšie DBMS sú zobrazené na Tabuľka 2. Oracle a Microsoft SQL server sú komerčné produkty a nie sú spravované open-source komunitou a teda nie sú vhodné na akademické účely. Zostávajú nám MySQL a PostgreSQL, oba DBMS sú veľmi podobné a výber je uskutočnený podľa používateľských preferencií. My sme vybrali PostgreSQL, no aplikácia by mala fungovať aj pri MySQL, zvlášť pri použití kvalitného modulu objektovo relačného mapovania (ORM).

Poradie	DBMS	Model
1.	Oracle	Relačný
2.	MySQL	Relačný
3.	Microsoft SQL Server	Relačný
4.	MongoDB	NoSQL
5.	PostgreSQL	Relačný

Tabuľka 2 Používanosť DBMS k Máj-u 2015, zdroj: <http://db-engines.com/en/ranking>

ORM zabezpečuje vytvorenie virtuálnej objektovej databázy, kde objekty mapujú tabuľky entitno-relačného modelu a sú nezávislé na konkrétnom DBMS a práca s nimi je zjednotená pomocou ORM aplikačného rozhrania. Vďaka tejto skutočnosti je možné vytvorenie aplikačného kódu, ktorý korektne spolupracuje s rozdielnymi DBMS.

Pre node.js je vytvorených viacero ORM modulov a selekcia kvalitných je kľúčová, lebo neskoré identifikovanie chýbajúcej funkcionality môže viesť k časovo náročnej refaktorizácii celého zdrojového kódu. Našťastie na portáli <http://www.stackoverflow.com> som našiel viacero obdobných diskusií, kde komunita riešila totožnú dilemu.

Ako najlepší ORM modul pre node.js bol vybraný **sequelizejs**⁷, ktorý podporuje PostgreSQL, MySQL, MariaDB, SQLite a MSSQL. Jeho výhodou je kvalitná dokumentácia, veľká komunitná základňa a jeho povaha založená na promis-och (prísľuboch), ktoré výborne spolupracujú s node.js a modulom Q.

2.2.3 Socket.io

Kvôli distributívnej povahe výslednej webovej aplikácie je nutné zabezpečiť duplexnú komunikáciu medzi uzlom (stránkou otvorenou v prehliadači klientského počítača) a serverom.

Avšak od počiatkov webových aplikácií, vývojári pracovali na rôznych spôsoboch duplexnej komunikácie medzi server a prehliadačom, či už použitím technológie Java, Flash, Comet a mnohých iných. No aj napriek tomu, prvá špecifikácia pre tento účel bola schválená až v roku 2011 a to konkrétne v HTML5 WebSocket (RFC 6455⁸), ktorá definuje plne duplexný komunikačný kanál operujúcim cez web cez jeden soket. (19)

Napriek publikovanej špecifikácii, všetky staršie verzie prehliadačov, ktoré sa stále používajú nepodporujú tento štandard. Modul **socket.io**⁹ je abstrakčnou vrstvou pre WebSockets s použitím technológií Flash, XHR, JSONP a HTMLFile. Je open-source komunitou spravovaným projektom pozostávajúcim zo serverovej a klientskej knižnice. Komunikácia prebieha na báze správ, kde klient aj server môžu emitovať a spracovávať správy rôzneho typu.

⁷ <http://docs.sequelizejs.com/en/latest/>

⁸ <https://tools.ietf.org/html/rfc6455>

⁹ <http://socket.io/>

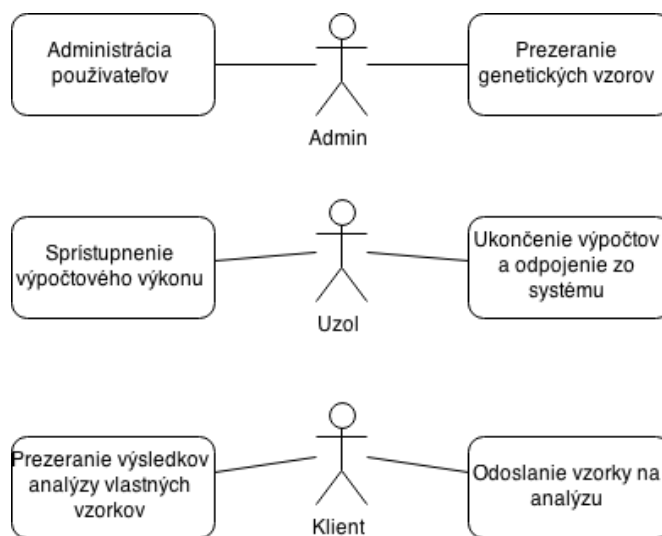
2.3 Implementácia

Táto sekcia je zameraná na priblíženie celkovej štruktúry systému a podrobnému popísaniu algoritmizačných a implementačných výziev, ktoré bolo nutné vyriešiť počas tvorby projektu.

2.3.1 Use-case

Use-case je situácia kedy je systém použitý na vykonanie jednej alebo viacerých používateľských požiadaviek, inými slovami use-case opisuje časť funkcionality, ktorú systém poskytuje. Use-case-y sú v srdci modelu, pretože ovplyvňujú a smerujú všetky ostatné elementy systémového dizajnu. (20)

Vytvorený use-case diagram systému je na Obrázok 10.



Obrázok 10 Use-case diagram projektu

Po analýze zadania a špecifikácie som vyčlenil tri systémové role a tými sú:

- **Admin** – ktorý má na starosti manažment používateľov, môže zaregistrovať klientov a uzly. Za zmienku stojí, že klientov môže registrovať iba admin, uzly sa môžu registrovať aj samé. Taktiež má umožnené prehľadávanie všetkých genetických vzorov, pre kontrolu korektného načítania a korektnosti údajov.
- **Klient** – predstavuje stakeholdera, ktorý má k dispozícii zosekvenovanú vzorku DNA potrebujúcu analýzu. Môže to byť molekulárny biológ, alebo doktor s výsledkom zosekvenovania pacientovej genetickej vzorky. Táto roľa môže

zadat' úlohu na analýzu a môže prezerat' priebeh analýzy a výsledky všetkých vykonaných analýz aj retrospektívne.

- **Uzol** – predstavuje používateľa, ktorý má záujem o poskytnutie svojho výpočtového výkonu do systému. Nemá prístup k dátam a nepredpokladáme, že by robil záškodnícku činnosť. Zneužitie dát nehrozí, pretože sa k nemu dostanú iba diskrétna dáta bez kontextu.

Systém neobsahuje príliš mnoho use-case-ov, to značí, že interakcia používateľov so systémom nie je bohatá, no nižšie bude popísané, že logika stojaca za nimi je zložitejšia a komplexnejšia ako sa na prvý pohľad môže zdať.

2.3.2 Javascript Distributed System Module (JDSM)

Účelom JDSM je separovanie logiky manažmentu a komunikácie uzlov a je tvorený nezávisle od aplikačnej logiky a pomocou jednoduchého aplikačného rozhrania je ho možné implementovať do rôznych projektov, ktoré požadujú spoluprácu pripojených uzlov. Modul je vytvorený štandardizovane a nič mu nebráni k nasadeniu do npm opísaného v 2.2.1. Je postavený na knižnici socket.io približenej v 2.2.3 a pozostáva zo serverovej časti a klientskej časti, ktorú je potrebné prekonvertovať na kód spracovateľný prehliadačom napríklad pomocou modulu browserify spomenutého taktiež v 2.2.1.

Modul zabezpečuje:

- Spracovanie pripojenia uzlu
- Spracovanie odpojenia uzlu
- Zabezpečenie odmerania kvality uzlu pomocou cyklického merania oneskorenia a pomocou jednorazového odmerania bandwidth kapacity uzlu
- Odoslanie a spracovanie asynchrónneho poľa požiadaviek
- Odoslanie a spracovanie synchrónneho reťazca požiadaviek

Hlavný cyklus modulu je zobrazený v podobe aktivity diagramu v Príloha A. Tento cyklus zabezpečuje ošetrovanie pripojenia a odpojenia uzlov pomocou socket.io emitovaných udalostí. Aktívne uzly sú spravované v *Active Node List* (ANL) objekte v podobe *Node* objektov. Podrobná štruktúra tried je dostupná v dokumentácii.

Pri zaregistrovaní uzlu sa automaticky spustí bandwidth tester, ktorý jednorazovo určí rýchlosť odosielania a prijímania dát uzlu. Taktiež sa spustí automatické cyklické testovanie

latencie uzlu, ktoré je použité okrem iného aj na dodatočné testovanie prerušenia spojenia s uzlom a následnom odpojení uzlu zo systému.

Hlavnou funkcionalitou JDMSM je posielanie požiadaviek na spracovanie uzlami pripojenými do systému, aktivita diagram ich spracovania je zobrazený na Obrázok 11. Na tento účel boli vytvorené dve metódy a tými sú:

- JDMSM.sendSyncRequest(reqs, callback)
- JDMSM.sendAsyncRequest(reqs, callback)

Kde parameter *reqs* je pole požiadaviek na jednotlivé uzly, ak uzol nie je špecifikovaný vyberie sa najvýhodnejší uzol. Výber najvhodnejšieho uzlu je zabezpečený funkciou zohľadňujúcou oneskorenie, bandwidth, výpočtovú silu a zaťaženie uzlov, a de facto je zabezpečené rovnomerné rozdelenie výpočtového výkonu.

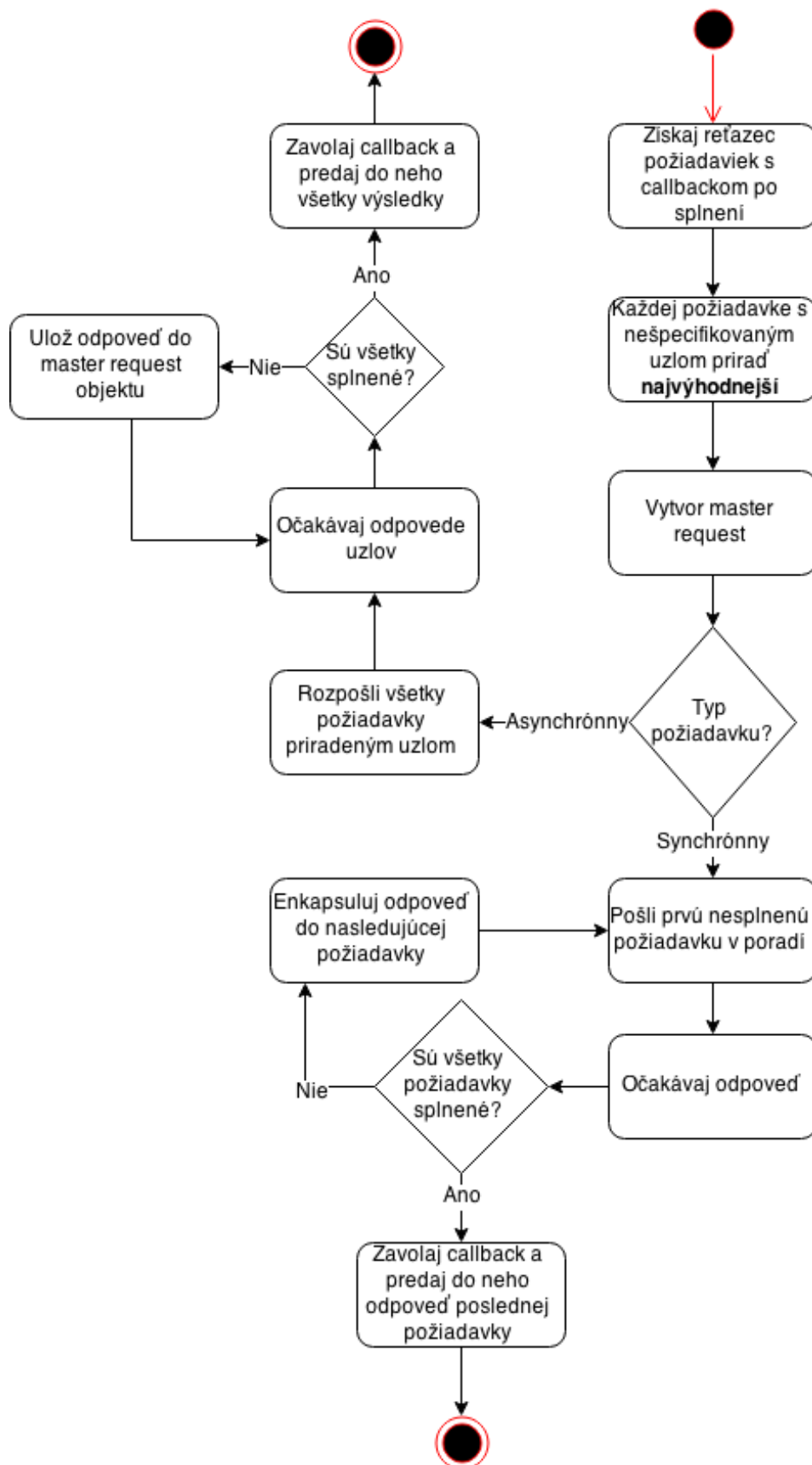
Štruktúru parametru *reqs* v JavaScript syntaxe je možné zapísať:

```
var reqs = [  
  {  
    node: JDMSM.Node object,  //(optional) chosen Node object from ANL  
    requestData: {  
      eventName: 'ping',      //(required) eventName for client side  
      data: {}                 // data for node to process  
    }  
  }  
];
```

Synchrónny typ funkcie zabezpečuje sekvenčné spracovanie požiadaviek, kde výsledok požiadavky je ovplyvnený výsledkom predošlej požiadavky, a preto sa do dát poslaných do uzla pridá hodnota *prerequisites*, kde bude prístupný predošlý výsledok. Po získaní odpovede pre poslednú požiadavku, sa zavolá callback funkcia s vloženými výslednými dátami.

Asynchrónny typ požiadavku odošle všetky požiadavky okamžite a po získaní všetkých odpovedí zavolá callback funkciu do ktorých sa vložia tieto dáta v korektnom poradí, aby bolo možné korektné spárovanie požiadaviek a výsledkov.

Každý používateľskej požiadavke (Master Request) ako aj každej jej požiadavke modul priradí unikátne identifikačné číslo, ktoré zabezpečí jej korektné spracovanie, toto číslo je diskkrétne a v tandeme s klientskou časťou modulu sa zabezpečí jeho anonymita.



Obrázok 11 Aktivita diagram spracovania požiadaviek JDSM

Server registruje pre každý pripojený socket spracovanie udalosti s kľúčovým menom *result*, všetka komunikácia z uzlov prechádza týmto handlerom, ktorý podľa *requestId* určí odpoveď o ktorú požiadavku sa jedná.

Trieda *Client* poskytuje aplikačné rozhranie na implementovanie klientskej časti distributívneho systému, inými slovami logiku vykonávanú na uzle. Z povahy knižnice *socket.io* popísanej v 2.2.3 je komunikácia zabezpečená pomocou správ ľubovoľného typu definovaného ľubovoľným reťazcom nazývaným *eventName*.

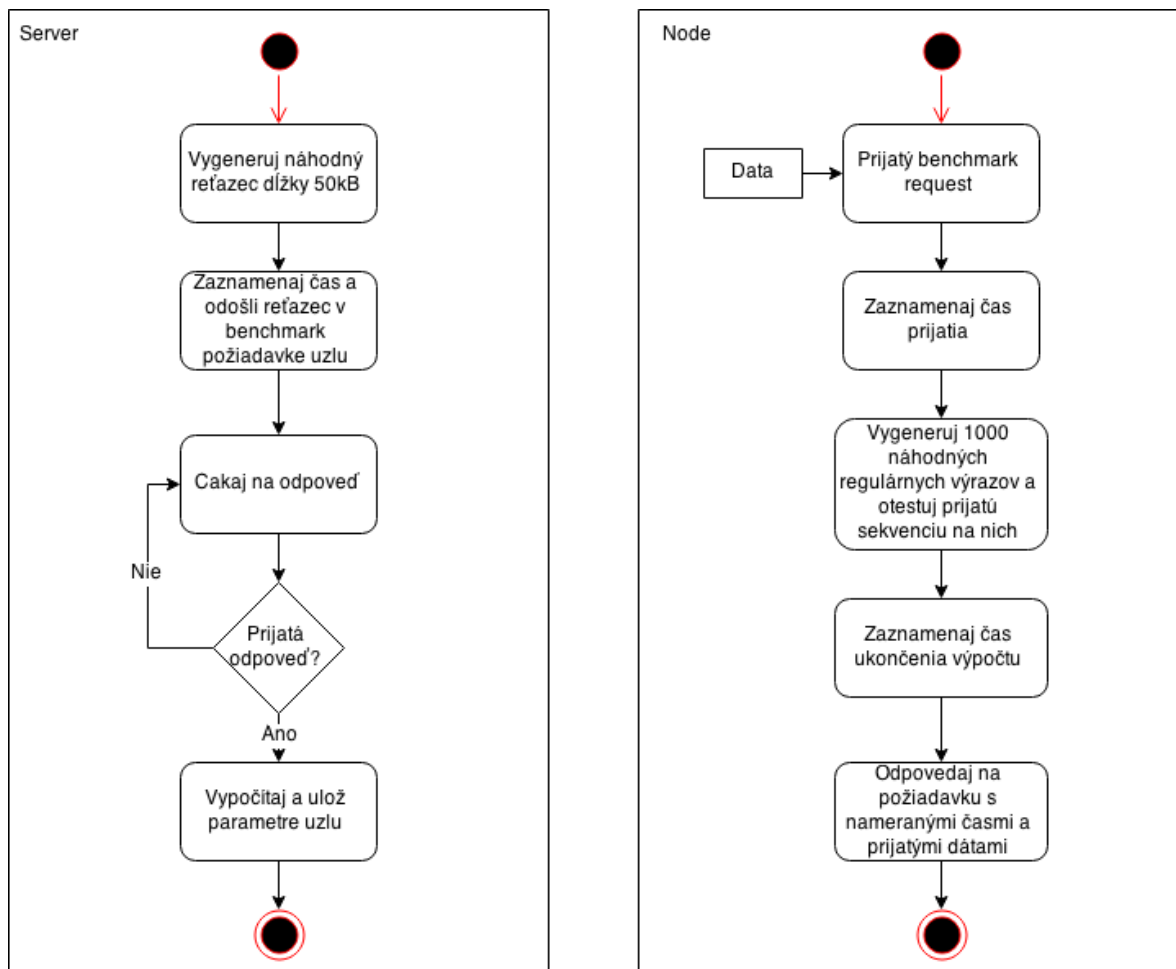
Z dôvodu, že serverová časť JDSDM pracuje s unikátnymi požiadavkami s unikátnymi id, všetky dáta odosielané z uzlu sú emitované s *eventName* = 'result' a sú spracovávané podľa *requestId*. Aby sa predišlo prepisovaniu prípadne manipulácii s týmto unikátnym číslom, tak klientská trieda odstráni všetky kontrolné id a aplikačnej logike sa neposkytnú, pri odoslaní odpovedí sa k nim znova pripoja. Tento spôsob bol inšpirovaným sieťovým modelom ISO/OSI.

JDSDM automaticky registruje a spracováva dva typy správ, konkrétne sa jedná o *ping* a *benchmark*.

Správa typu *ping* slúži na určenie oneskorenia uzlu a jej logika je veľmi jednoduchá, akonáhle zo servera príde emitovaná správa *ping*, uzol na správu odpovie a server určí aktuálnu latenciu.

Správa typu *benchmark* je zložitejšia a je serverom emitovaná okamžite po registrácii uzlu. Slúži na určenie rýchlosti prijímania a odosielania dát uzlom a taktiež na určenie jeho výpočtového výkonu. Všetky tieto parametre sú potom použité na korektné vybranie uzlov do požiadaviek.

Proces vykonania *benchmark* testu je zobrazený na Obrázok 12. Pozostáva z odoslania náhodne vygenerovaného reťazca znakov konštantnej veľkosti a zaznamenania času odoslania. Uzol zaznamená čas prijatia dát podľa ktorého sa neskôr vypočíta rýchlosť sťahovania, vykoná procesne náročnú funkciu, konkrétne vygeneruje tisíc náhodných regulárnych výrazov a otestuje ich na prijatom náhodnom reťazci, po vykonaní zaznamená čas a rozdiel času prijatia a ukončenia manipulácie je index výkonnosti uzlu. Na koniec odošle dáta naspäť a na čase prijatia dát serverom sa určí rýchlosť odosielania dát uzlom.



Obrázok 12 Aktivita diagram benchmark procesu s pohľadu servera aj uzlu

Zaregistrovanie správy na spracovanie uzlom sa vykoná metódou `JDSM.registerTask`, ktorá je približená nižšie.

Najjednoduchšie **použitie JDSM modulu** sa dá ilustrovať krátkym kódom:

```

Client.js
var JDSM = require('./JDSM').Client;
JDSM.registerTask('transform', function(data, respond) {
  var transformed = transformFunc(data);
  respond.respond(transformed);
});

Server.js
var JDSM = require('./JDSM')(app);
JDSM.sendSyncRequest([
  {
    requestData: {
      eventName: 'transform',
      data: 'My data for transformation'
    }
  }, function(respond){
    console.log('Transformed data from node: ', respond);
  }
]);
  
```


Zložitejší príklad aplikačného využitia JDSM je dopodrobna opísaný v 2.3.5.

2.3.3 Získanie vzorov

Aby bolo možné vytvorenie analyzátora vzorky, je potrebné získať reálne vzory génov z dostupných genómových prehliadačov a z nich vytvoriť model s dostupnými atribútmi. Ako bolo spomenuté v 2.2 použijeme voľne prístupný prehliadač ensembl.com.

Tento prehliadač poskytuje voľný priamy prístup k ich MySQL databázam, avšak ich architektúra je veľmi zle navrhnutá a jej časté upravovanie spôsobilo vytvorenie viacerých databázových schém v ktorých sa veľmi ťažko orientuje. Našťastie je sprístupnený aj REST API servis, ktorý je dobre zdokumentovaný a ľahko použiteľný, jedinou nevýhodou je zavedenie limitu požiadaviek na pätnásť za sekundu, čo nespôsobuje príliš veľký problém, keďže načítanie a uloženie vzorov do nášho databázového systému sa vykonáva iba pri prvotnej inštalácii, no je to faktor, ktorý je nutný zohľadniť pri napísaní loadovacieho skriptu.

Skript, ktorý načíta a uloží vzory do lokálnej databázy som nazval Pattern Crawler. Pretože REST servis neposkytuje možnosť získať zoznam všetkých dostupných génov, bol som nútený využiť kombináciu prístupu do MySQL databázy z ktorej som získal unikátne identifikačné reťazce génov a podľa nich som poskladal url na REST požiadavky, ktoré je nutné posilať s oneskorením, aby sa zohľadnilo vyššie spomenuté obmedzenie pätnástich požiadaviek za sekundu.

Z dostupných informácií sme zadefinovali objekt vyjadrujúci vzor s atribútmi:

- **Meno** – označenie génu, považuje sa za unikátny identifikátor génu napr. ENSG00000197386
- **Popis** – detailnejší popis génu, jeho zdroj a možné ďalšie informácie relevantné pre molekulárneho biológa
- **Data** – nukleotidová sekvencia génu
- **Chromozóm** – index chromozómu, kde sa gén nachádza
- **Začiatok sekvencie** – index nukleotidu v chromozóme od ktorého sa začína gén
- **Koniec sekvencie** – index koncového nukleotidu, dá sa vypočítať ako začiatok sekvencie + dĺžka dát
- **Typ závitnice** – keďže DNA je dvojzávitnica a gén sa mohol študovať z jednej z dvojzávitnice, nachádza sa v popise génu aj táto vlastnosť, avšak nukleotidové

páry sú podľa 1.1.2 komplementárne, a preto je možné prekonvertovať gén z jednej zátvorky na druhú a vice versa.

Poznatzky opísané v tejto sekcii sú implementované v grunt úlohe, ktorej spustenie je odporúčané po inštalácii systému pomocou *grunt fetchEnsemblData*, avšak za účelom otestovania je vhodné najprv zavolať úlohu s limitujúcim množstvom génov pridaním číselného parametru.

2.3.4 Formát dát a využitie regulárnych výrazov

V tejto sekcii stručne opíšeme spracovanie dát vzoriek, ich skladovanie, manipuláciu, popíšeme očakávaný formát vzoriek a opíšeme použitie regulárnych výrazov v aplikácii.

V 1.1.2 a 2.1 bolo spomenuté, že sekvencia sa skladá z nukleotidov označenými C,G,T a A, vstupný súbor vzorky sa preto skladá iba z týchto znakov, prípadne ich ekvivalentov c,g,t a a. Navyše je nutné zadefinovanie aj ich pozícií, pretože môže byť zosekvenovaná iba časť alebo časti chromozómu prípadne chromozómov. Z tohto dôvodu sme zaviedli vlastnú syntax popísanú nižšie, no je možné vytvoriť middleware, ktorý by hocikaký proprietárny formát používaný existujúcimi zariadeniami prekonvertoval na náš interný formát.

Používaný formát sa skladá zo sekvenčných znakov a kontrolnej časti určujúcej pozíciu, najlepšie sa demonštruje na príklade.

Obsah súboru *Kanitra_sekvencia.dna* by mohol vyzeráť napríklad:

[1:215]ccgtaccattg[13:487]gggtttcc

Tento súbor definuje zosekvenovanie častí genómu a to konkrétne jedenásť nukleotidov prvého chromozómu od pozície 215 a osem nukleotidov trinásteho chromozómu začínajúcich na pozícii 487. Logické nezrovnalosti ako napríklad dvojité zadefinovanie rovnakej pozície nukleotidu program nerieši.

Program ukladá prijaté súbory so vzorkami do vyhradeného priečinka, avšak pred tým prejdú normalizáciou, ktorá pozostáva z premeny znakov na veľké písmená, aby bola zabezpečená konzistencia dát a nebolo potrebné dodatočné ošetrovanie. Taktiež sa súbory premenujú na formát *UserId_SampleId.dna*, to nám zjednoduší identifikáciu konkrétneho súboru. V súčasnosti sa väčšina textov kóduje vo formáte UTF-8¹⁰, ktorý podporuje kódovanie všetkých dostupných Unicode znakov a jeden znak zaberá osem bajtov. Avšak

¹⁰ <http://en.wikipedia.org/wiki/UTF-8>

z dôvodu, že znaky vzorkových súborov sú limitované a vieme, že všetky sú zakódované vo formáte ASCII, ktorý zaberá iba jeden bajt, výsledný súbor uložíme v tomto kódovaní, táto funkcionálna nám môže zredukovať pamäťové nároky na osminu a takisto aj traffic pri rozdistribuovaní vzoriek jednotlivým uzlom.

Pri tvorení zadania sa predpokladalo rozsiahle použitie **regulárnych výrazov**, avšak prístupné dáta umožňujú iba ich čiastočné použitie. Diskrétna povaha vzorov a nedostupnosť ich variácií, prípadne mutácií zabraňujú vytvoreniu regulárneho výrazu, ktorý by zahŕňal všetky podoby génu, prípadne rodiny génov. Očakával som dostupnosť N variácií génu určitej genetickej choroby, ktoré by sa mohli zlúčiť do jedného regulárneho výrazu definujúceho danú chorobu. Z dôvodov stále pomerne drahej technológie sekvenovania a obmedzenému financovaniu výskumov v tejto doméne tieto informácie nie sú dostupné, ale verím, že v najbližších rokoch sa to zmení.

Jediné využitie regulárnych výrazov, ktoré sme dokázali identifikovať ako použiteľné, je vytvorenie regulárneho výrazu zo sekvencie konkrétneho vzoru za účelom jeho skrátenia (skomprimovania), pretože ak zoberieme napríklad (regulárny výraz je zapísaný vo formáte JavaScript ECMA 262¹¹:

Vzor: ccccccccgtagtagtagta - dĺžka 21B Regulárny výraz: /c{9}gta{4}/ - dĺžka 10B

Vďaka vytvoreniu regulárneho výrazu je možné signifikantné skrátenie výrazu a tým pádom pamäťovej náročnosti a trafficu pri rozdistribuovaní. Taktiež implementácia vlastného testera regulárnych výrazov je veľmi neefektívna, pretože všetky prehliadače implementujú vyhodnocovanie regulárnych výrazov pomocou nedeterministických automatov popísaných v 1.2.2 a 1.2.3, no sú skompilované do natívnych kódov, čo zabezpečuje oveľa väčšiu efektivitu ako hocikako efektívne napísaný klientsky algoritmus.

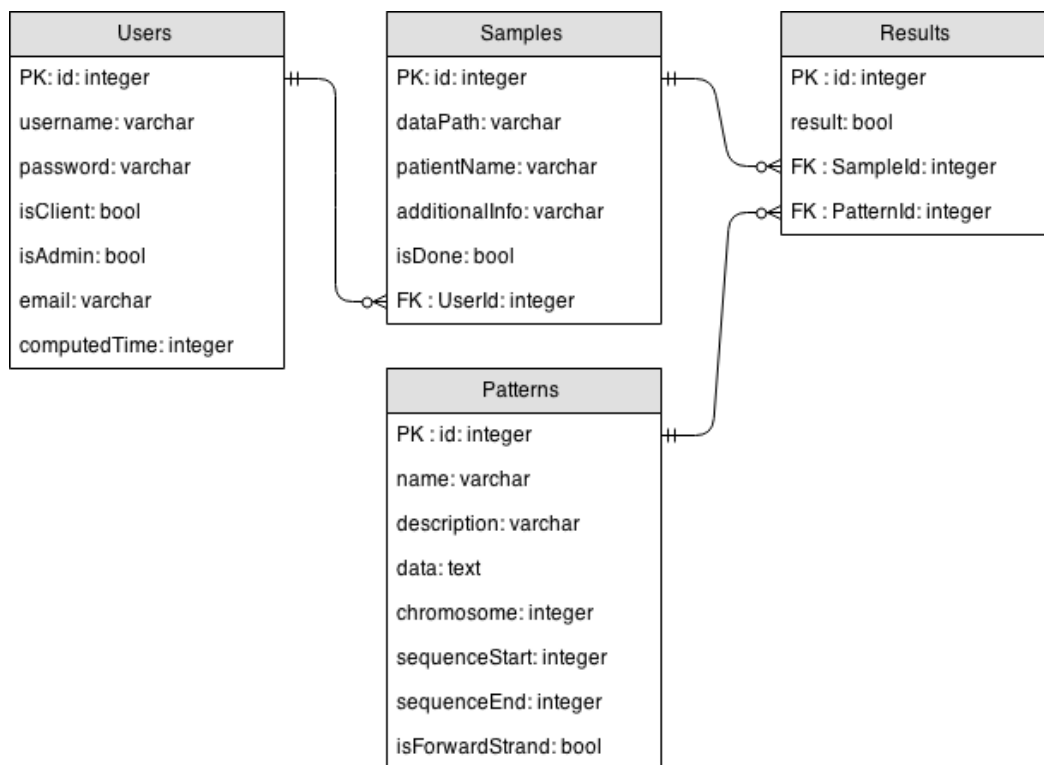
2.3.5 Algoritmus analýzy

V tejto sekcii je dopodrobna opísaný algoritmus vykonávajúci hlavnú funkciu aplikácie, analýzu genetickej sekvencie. Implementuje JDSM modul popísaný v 2.3.2 a očakáva vstupnú sekvenciu vo formáte definovanom v 2.3.4.

Z informácií napísaných v tejto kapitole som navrhol databázovú štruktúru zobrazenú na Obrázok 13 v podobe entitno-relačno atribútového (ERA) diagramu. Databáza je

¹¹ <http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf>

jednoducho porozumiteľná a pozostáva zo prepojených štyroch tabuliek reprezentujúcich používateľov, vzory, vzorky a výsledky analýz.



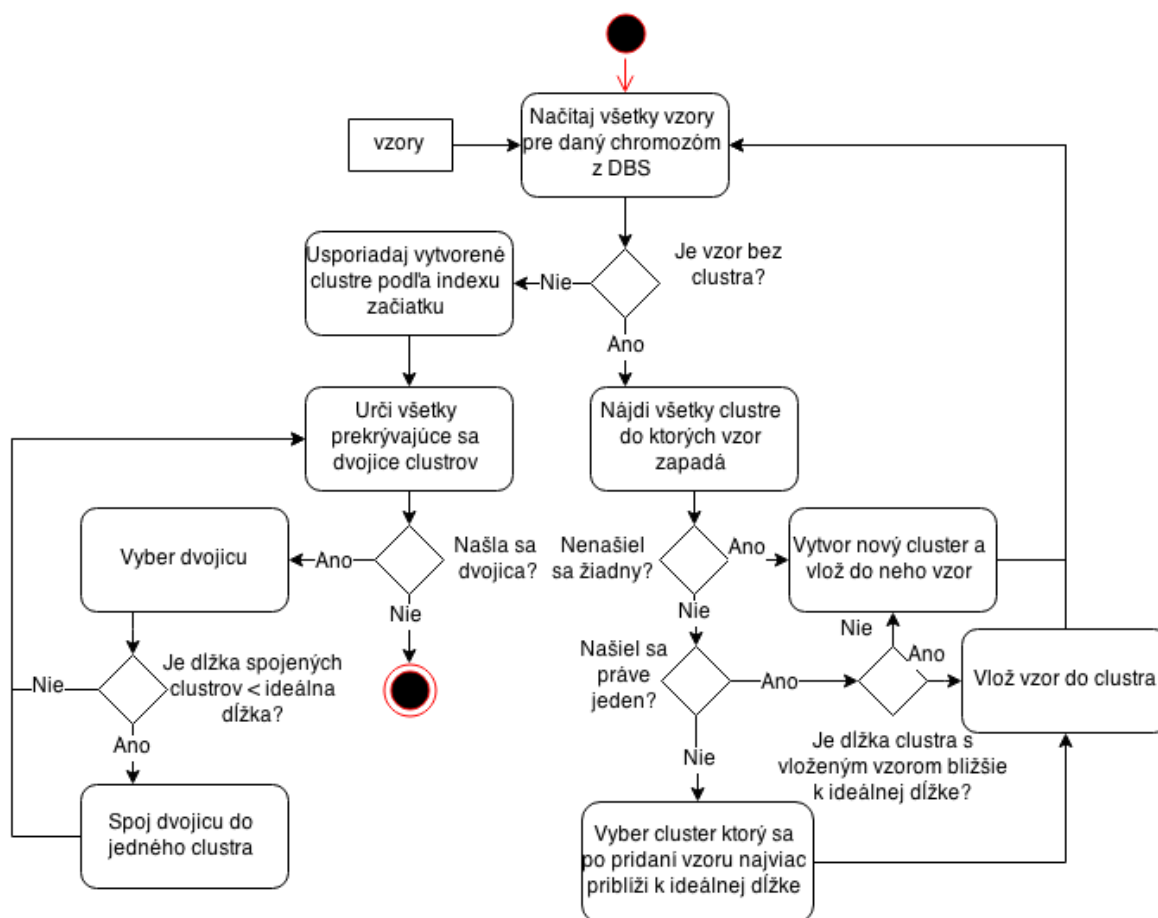
Obrázok 13 ERA diagram výstupnej aplikácie

Pred inicializáciou implementácie JDSM modulu pri spustení aplikácie na serveri prebehne **clustering vzorov** (spájanie vzorov do väčších logických celkov, takzvaných clustrov). Na tento účel som navrhol triviálnu implementáciu hierarchického clustrovania založeného na pozícii vzoru v sekvencii. Vzory sú clustrované osobitne pre každý chromozóm. V aplikácii sú clustre nedeliteľné jednotky a manipuluje sa iba s nimi, to zjednodušuje prácu a zrýchľuje beh programu. Jedným z nastaviteľných atribútov aplikácie je ideálna dĺžka clustra, tá ovplyvňuje množstvo vytvorených clustrov a priemerný počet vzorov patriacich jednému clustru. Ideálna dĺžka je taká, aby počet clustrov bol totožný s počtom pripojených uzlov.

Cluster je definovaný unikátnym číselným identifikátorom a obsahuje atribúty:

- Číslo chromozómu
- Index začiatku sekvencie (aktualizovaný pri každej zmene vzorov)
- Index konca sekvencie (aktualizovaný pri každej zmene vzorov)
- Pole vzorov nachádzajúcich sa vo vyhradenom úseku sekvencie

Diagram aktivít clustrovacieho algoritmu je zobrazený na Obrázok 14.



Obrázok 14 Aktivita diagram clustrovacieho algoritmu

Počas behu programu je nutné zabezpečiť aby v každom momente mali všetky clustre priradené uzly (ak je aspoň jeden uzol pripojený), ktoré ich analyzujú. **Distribúcia clustrov** je možná vďaka možnosti nastavenia funkcie volanej pri každom pripojení, respektíve odpojení uzlu poskytujúcou JDSM modulom popísaným v 2.3.2.

Základnou premisou distribučného algoritmu je rovnomerné rozdelenie clustrov medzi uzly všetky pripojené uzly. Experimentálnym testovaním je možné dosiahnuť lepšie rozdelenie s uvažovaním parametrov uzlov (výkon, oneskorenie, rýchlosť prijímania/odosielania), avšak vývoj takéhoto algoritmu by bol časovo náročný s potrebou reálneho testovania a bohužiaľ táto funkcionálna sa neurčila ako prioritná, a preto je použité rovnomerné rozdelenie.

Po pripojení prvého uzlu do systému, priradíme všetky clustre tomuto uzlu a spustíme hlavný beh programu popísaný nižšie.

Po pripojení n-tého uzlu odstránime C clustrov z každého uzlu, kde C je definované vzťahom:

$$C = \frac{C_c}{N_c} - \frac{C_c}{N_c + 1}$$

,kde C_c je počet clustrov a N_c počet uzlov.

Pri odpojení uzlu priradíme aktívnym uzlom clustre odpojeného uzlu. Algoritmus redistribúcie clustrov je zobrazený v Príloha B.

Klientská časť má dva možné módy operovania:

- Cachovanie clustrov
- Necachovanie clustrov

Pri necachovaní clustrov sa v pamäti uzlov neuchovávajú žiadne dáta a všetky zasielané požiadavky od servera sú sebestačné, to znamená, že každá požiadavka obsahuje vzory aj dáta vzorky a je ju možné vyhodnotiť na ľubovoľnom uzle.

Pri cachovaní clustrov sa po priradení alebo odstránení clustra z uzlu odošle požiadavka na pridanie, respektíve odstránenie clustra z cache clustrov uchováanej v pamäti uzlu. Výhodou tohto princípu je, že vzory clustrov nie je nutné posielat' pri každej požiadavke na analýzu, tento spôsob je rýchlejší a efektívnejší, keďže sa odosiela iba časť sekvencie vzorky určenej pre cluster už uchovaný v pamäti.

Aj keď metóda s cachovaním clustrov má nesporné výhody, konkrétne signifikantné obmedzenie trafficu a lepšie využitie pamäte uzlu, koncepčne zaostáva pri vysokom pomere vzoriek k počtu aktívnych uzlov. Ak zoberieme hraničný prípad, že je definovaných tisíc vzorov s priemernou dĺžkou 1MB a pripojený je iba jeden pracovný uzol, znamenalo by to uloženie 1GB dát do operačnej pamäte uzlu, čo by mohlo viesť k výraznému spomaleniu používateľovho zariadenia alebo až ku pádu prehliadača. Práve preto, je nastaviteľná hranica pomeru vzoriek a aktívnych uzlov a ak je aktuálny pomer pod ňou, používa sa pri výkone programu princíp necachovania, ktorý nie je obmedzovaný a nehrozí zahltenie uzlu a ak sa aktuálny pomer dostane nad hranicu odošlú sa požiadavky na cachovanie clustrov a prejde sa na tento spôsob.

Výpočtový algoritmus vykonávaný na uzloch je priamočiary. Pomocou JDSM modulu zaregistrujeme spracovávanie štyroch typov správ:

- *addClusters* – pridá clustre do pamäte
- *freeClusters* – odstráni clustre z pamäte
- *analyze* – analyzuje vzorku s použitím cachovaných clustrov, obsahuje subsekvenciu vzorky, id vzorky a id požadovaného clustra
- *analyzeNoCache* – analyzuje vzorku s použitím clustrov v požiadavke, obsahuje subsekvenciu vzorky, id vzorky a cluster pozostávajúci z id a poľa vzorov s vlastnými id

Odpoveďou uzlu na *analyze* a *analyzeNoCache* požiadavky je objekt s id vzorky pre ktorú sú výsledky a pole výsledkov definované id vzoru a boolovej hodnoty značiacej pozitívny alebo negatívny test daného vzoru na poskytnutú sekvenciu vzorky.

Po zadaní clustrov vzorov a ich distribúciou medzi aktívne uzly je na rade **analýza nahranej vzorky** sekvencie DNA.

Ako bolo opísané v 2.3.4 vzorka sa po nahratí do systému normalizuje a uchová v štandardizovanom tvare so štandardizovaným menom. Po tomto procese nastane čas spustenia analýzy.

Ako bolo viackrát spomenuté sekvencia vzorky a teda čítaný súbor môže mať veľkosť niekoľkých bajtov až niekoľkých gigabajtov, keďže za žiadnych okolností nie je potrebné pracovať s celou sekvenciou naraz je veľmi výhodné a žiadané využiť dostupnú možnosť `node.js` a tou je vytvorenie čítacieho prúdu údajov (`read stream`), ktorý číta súbor po častiach variabilnej dĺžky, to zabezpečí menšie nároky na pamäť a rýchlejší začiatok spracovania.

Pre potreby načítania je vytvorená trieda *SampleReader (SR)*, ktorá má na starosti skladanie sekvencie po častiach získaných z dátového prúdu a určovanie clustrov, ktoré patria do načítanej vzorky. Uchováva v sebe takzvanú internú sekvenciu, ktorú sa snaží minimalizovať pre ušetrenie pamäťovej náročnosti. Akonáhle sa po pridaní načítanej časti nájde cluster plne pokrytý internou sekvenciou, je vytvorená požiadavka na analýzu clustra do ktorej sa vloží subsekvencia tejto sekvencie potrebná k analýze a odošle uzlu, následne sa interná sekvencia uchovávaná skráti na najbližší neanalyzovaný cluster. Toto skrátenie je

dosiahnuteľné uchovávaním clustrov usporiadaných vzostupne podľa začiatkovej pozície na chromozóme.

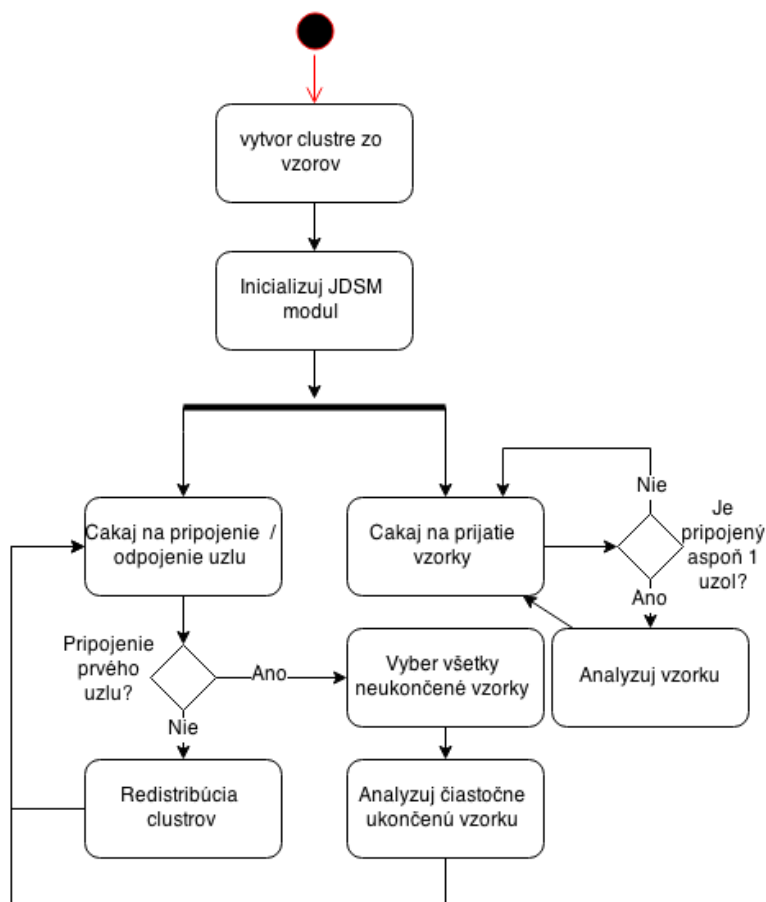
SR taktiež identifikuje kontrolné sekvencie vzorku, ktoré ako je popísané v 2.3.4 určujú pozičné časti sekvencie. Ak sa ukončí pozícia, SR nájde všetky clustre čiastočne pokryté internou sekvenciou a odošle ich na analýzu. Toto opatrenie je nutné z dôvodu nedeliteľnosti clustrov, avšak určitý vzor clustra sa môže nachádzať na načítanej časti.

Po načítaní celého vzorku, následnom identifikovaní, rozposlaní požiadaviek, získaní odpovedí na požiadavky a ich perzistencie v databázovej tabuľke Results sa uloží databázový atribút *isDone* konkrétnej vzorky (tabuľka Samples) na true.

S popísanými procesmi je čas na navrhnutie celkového behu programu. Diagram tohto procesu je na Obrázok 15.

Všetky aktivity diagramu boli popísané okrem aktivity *Analyzuj čiastočne ukončenú vzorku*. Tá sa volá pri pripojení prvého uzlu do systému, inými slovami pri zapnutí výpočtov programu. Vtedy je nutné vybrať všetky neukončené vzorky, ktoré mohli nastať z dôvodu nahrania pri žiadnom pripojenom uzle, alebo pri odpojení posledného aktívneho uzlu (a tým pádom zastavenia výpočtov), ktorý mal rozpracované požiadavky a tieto vzorky znova analyzovať. Avšak pre zlepšenie výkonu je nutné rozposlať požiadavky iba na clustre, ktoré obsahujú neanalyzovaný vzor, inak by sa mohlo stať, že k ukončeniu vzorky chýba iba výsledok jedného vzoru, no duplicitne by sa analyzovali všetky vzory. Ako bolo spomenuté cluster je nedeliteľná časť logiky programu, preto by sa teoreticky mohli ošetrovať iba vyhodnotené celé clustre, avšak to by mohlo viesť k nekonzistencii lebo pri každom zapnutí programu sa clustering vykoná nanovo a výsledné clustre môžu mať iný tvar, z dôvodu pridania nových vzorov alebo zmeny parametru programu určujúcej ideálnu dĺžku clustra.

Diagram neobsahuje terminálny stav, pretože hlavná slučka programu sa nedá ukončiť okrem tvrdého vypnutia serverového procesu.



Obrázok 15 Diagram hlavného behu programu

2.3.6 Testovanie

Táto sekcia popisuje použité spôsoby testovania korektnosti aplikácie.

Napriek našim snahám o čo najzrozumiteľnejšie vysvetlenie našich zámerov a otázok ohľadom ich vedného odboru, čoho dôkazom je dokument *OvereniePostupuDP.doc* priložený v Príloha D, mi **odborníci nevedeli odpovedať**. Dôvodom je veľmi obmedzený počet molekulárnych biológov s hlbokými poznatkami v danej doméne a ešte menší počet ľudí zaoberajúcich sa analýzou DNA sekvencií. Oslovili sme troch vyštudovaných odborníkov, no bohužiaľ ani jeden mi nebol schopný jednoznačne odpovedať na položené otázky a ani potvrdiť alebo vyvrátiť korektnosť môjho navrhovaného riešenia. Z tohto dôvodu sme systém navrhli podľa svojho najlepšieho svedomia a vedomia podľa mnou naštudovanej literatúry.

Za účelom testovania kritických metód a algoritmov som využil JavaScript testovací framework Mocha¹², ktorý je postavený na platforme node.js a vstavaný štandardný assert modul, ktorý je súčasťou node.js jadra. Všetky napísané testovacie skripty sú v priečinku test, avšak aj keď Mocha výborne podporuje asynchrónne testy, nenašiel som spôsob ako nasimulovať a otestovať pripájanie a interakciu uzlov.

Z toho dôvodu som sa spoliehal na časovo náročné **manuálne testovanie** a ad hoc debugovanie.

Pre uľahčenie manuálneho testovania boli vytvorené grunt úlohy spustiteľné konzolovými príkazmi. Všetky úlohy sú zdokumentované v projektovej dokumentácii a zahŕňajú:

- vygenerovanie náhodného vzorku pozitívnych pre zadané vzory
- vygenerovanie viacerých náhodných vzoriek s náhodnými pozitívnymi a negatívnymi vzormi uložených v *additionalInfo* vzorku
- vytvorenie testovacieho vzoru
- vygenerovanie súboru so simulovanou zosekvenovanou ľudskou DNA (3 miliardy nukleotidov rozdelených korektne do chromozómov)
- odstránenie všetkých dát z tabuľky

Výsledný kód bol otestovaný podľa dostupných schopností a možností, no vzhľadom na povahu systému je veľmi ťažké nasimulovanie a odladenie stoviek pripojených, pracujúcich uzlov na desiatkach niekoľko gigabajtových súboroch, a preto intenzívne testovacie skripty sú možné až pri spustenej beta prevádzke. Z toho istého dôvodu nie je v dobe písania diplomovej práce možné poskytnutie časových a výkonnostných benchmarkov, no po spustení verejnej beta verzie sa dáta pridajú na stránku projektu.

¹² <http://mochajs.org/>

Záver

Teoretická časť diplomovej práce v stručnosti opisuje poznatky z molekulárnej biológie a genetiky potrebné k pochopeniu komplexnosti analýzy DNA sekvencií živých organizmov. Snaží sa základné genetické pojmy ako gén, genetická porucha, fenotyp a iné vložiť do kontextu fyzickej stavby bunky v podobe chromozómov, DNA molekúl až po ich najzákladnejšiu jednotku, nukleotid, ktorého štyri typy (cytozín (C), guanín (G), adenín (A) a tymín (T)) definujú analyzovanú sekvenciu. Taktiež približuje základné teoretické základy stojacimi za teóriou regulárnych výrazov a distributívnych systémov. Ich využitie pri big data analýzach, ktoré sú stále viac a viac v centre pozornosti a záujmu dátových gigantov ako google, IBM a iné.

Praktická časť sa zaoberá návrhom a implementáciou distributívneho systému založeného na webových technológiách. Takýto model donedávna nebol zrealizovateľný, pretože JavaScriptové skripty v prehliadačoch boli obmedzované a veľmi pomalé, avšak v posledných rokoch sa z nich stali mocné nástroje konkurujúce zaužívaným platformám ako Java alebo .NET. Práve z tohto dôvodu, je tento druh systému jedným z prvých a jeho využitie rozmanité. Diplomová práca má dva výstupy, jedným je Node.JS modul, ktorý sa stará o komunikáciu medzi uzlami systému a poskytuje jednoduché aplikačné rozhranie na implementovanie ľubovoľného distributívneho systému. Druhým výstupom je implementácia modulu do webovej aplikácie slúžiacej na analyzovanie DNA sekvencie a určenie nachádzajúcich sa génov v nej.

Zistili sme, že využitie regulárnych výrazov pri analýze je značne obmedzené, z dôvodu nedostatku dostupných dát. Očakávali sme, že sekvencia génu rakoviny prsníka bude zdokumentovaná desiatkami variácií sekvencie podľa ktorých by sme vytvorili regulárny výraz, no tieto dáta sú nedostupné. Fakt, že molekulárnej biológii a konkrétne sekvenovaniu genetického kódu sa venuje malé množstvo ľudí a nami opýtaní nevedeli alebo nemali čas a vôľu odpovedať na nezrovnalosti značí, že táto doména je zložitá a je tu veľa miesta na zlepšenie.

Zoznam použitej literatúry

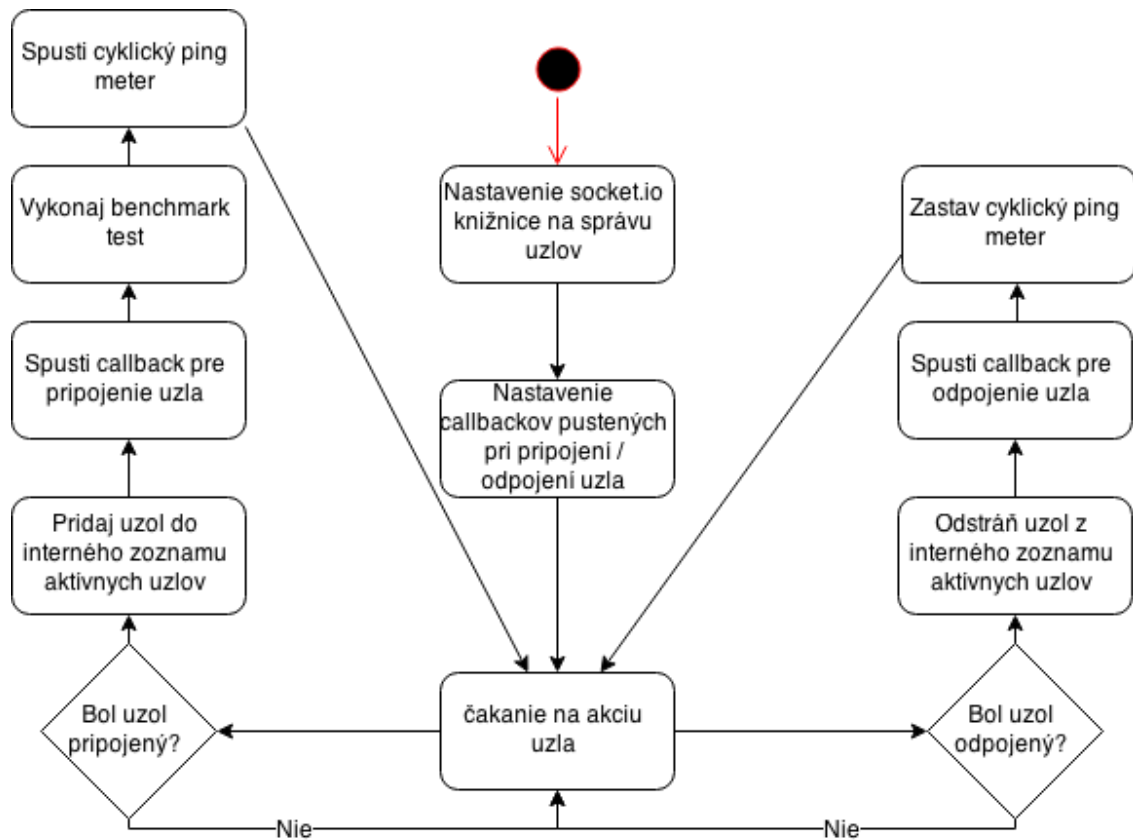
1. **Anne Matthews, RN, PhD.** <http://www.netwellness.org/>. [Online] 28. 6 2010.
<http://www.netwellness.org/healthtopics/idbd/2.cfm>.
2. **Hartwell, Leland, et al.** *Genetics: from genes to genomes*. 4th. New York : McGraw-Hill, 2011. ISBN 978-0-07-352526-6.
3. **Robinson, Tara ,PhD.** *Genetics for dummies*. 1st. Hoboken : Wiley Publishing, Inc., 2005. ISBN 978-0-7645-9554-7.
4. **Wetterstrand, Kris, M.S.** www.genome.gov. [Online] 31. 10 2014.
<http://www.genome.gov/sequencingcosts/>.
5. **Tripp, Simon und Grueber, Simon.** Economic Impact of the Human Genome Project. *battelle.org*. [Online] http://battelle.org/docs/default-document-library/economic_impact_of_the_human_genome_project.pdf.
6. **Cox, Russ.** Regular Expression Matching Can Be Simple And Fast . *swtch.com*. [Online] 1 2007. <http://swtch.com/~rsc/regexp/regexp1.html>.
7. **Goyvaerts, Jan und Levithan, Steven.** *Regular Expressions Cookbook, Second Edition*. s.l. : O'Reilly Media, 2012.
8. **Roberts, Eric.** *Automata Theory*. [Online] 1 2004.
<http://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html>.
9. **Juhás, Gabriel.** *Modelovacie formalizmy udalostných systémov*. Bratislava : RT systems s.r.o, 2011.
10. **Tanenbaum, Andrew S.** *Distributed Operating Systems*. s.l. : Prentice Hall, 1994. ISBN-13: 9780132199087.
11. **The Daily Beast.** Hackers' Most Destructive Attacks. [Online] 12. 11 2010.
<http://www.thedailybeast.com/articles/2010/12/11/hackers-10-most-famous-attacks-worms-and-ddos-takedowns.html>.
12. **Gartner.** *Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013*. 4 2013.
13. **Coulouris, George, Dollimore, Jean und Kindberg, Tim.** *Distributed Systems: Concepts and Design*. 3rd. s.l. : Addison-Wesley, 2001. ISBN: 0-201-61918-0.
14. **wikipedia.** *List of distributed computing projects*. [Online] 11 2014.
http://en.wikipedia.org/wiki/List_of_distributed_computing_projects.

15. **Loeliger, Jon und McCullough, Matthew.** *Version Control with Git.* s.l. : O'Reilly Media, 2012. 978-1-4493-1638-9.
16. **Ihrig, Colin.** *Pro Node.js for Developers.* s.l. : Apress, 2013. 978-1-4302-5860-5.
17. **Elliott, Eric.** *Programming JavaScript Applications.* s.l. : O'Reilly Media, 2014. 978-1-49195-029-6.
18. **Mardan, Azat.** *Pro Express.js.* s.l. : Apress, 2014. 978-1-484200-38-4.
19. **Rai, Rohit.** *Socket.IO Real-time Web Application Development.* s.l. : Packt Publishing, 2013. 978-1-78216-078-6.
20. **Miles, Russ und Hamilton, Kim.** *Learning UML 2.0.* s.l. : O'Reilly Media, 2006. 978-0-596-00982-3.

Prílohy

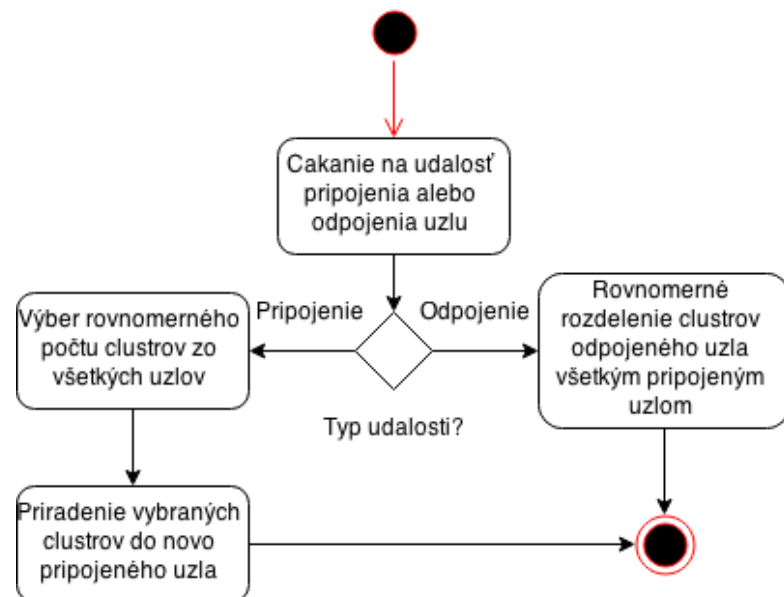
Príloha A: Základný running cyklus JDSM.	II
Príloha B: Diagram redistribúcie clustrov.	II
Príloha C: Inštalačná príručka.	III
Príloha D: CD nosič.	IV

Príloha A: Základný running cyklus JDSM



Obrázok 16 Aktivita diagram hlavného cyklu JDSM

Príloha B: Diagram redistribúcie clustrov



Obrázok 17 Aktivita diagram redistribúcie clustrov

Príloha C: Inštalačná príručka

Za účelom čo najjednoduchšej inštalácie systému na ľubovoľnom počítači je nutné vykonať tieto úkony:

1. Nainštalovanie node.js – <http://www.nodejs.org>
2. Nainštalovanie PostgreSQL databázy – <http://www.postgresql.org/>
3. Spustenie: `npm install sequelize-cli -g`
4. Prekopírovanie súborov z Príloha D alebo naklonovať repozitár spustením:
`git clone git@bitbucket.org:KandoSVK/diplomka.git`
5. Presun do novovytvoreného priečinku
6. Spustenie: `npm install`
7. Spustenie: `npm install bower -g`
8. Spustenie: `bower install`
9. Vytvorenie databázy a prihlasovacie údaje zapísať do: `config/config.json`
10. Spustenie: `sequelize db:migrate`
11. Spustenie: `npm install grunt -g`
12. Spustenie: `grunt createData`
13. Spustenie: `grunt fetchEnsemblData:100`
14. Spustenie: `sudo grunt` (iba grunt na Windows platforme)
15. Otvorenie prehliadača na url: `localhost:3000`
16. Prihlásenie pomocou údajov:

Login	Heslo	Roľa
admin	admin	admin
client	client	client
node	node	node

Príloha D: CD nosič

Elektronický nosič CD obsahujúci elektronickú formu tejto práce spolu s implementačnou časťou, ktorá obsahuje zdrojové kódy, všetky potrebné inšalačné súbory a návod na inštaláciu.