FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Framework pro distribuované výpočty v prostředí webu
Student:	Bc. Jakub Šiller
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

S rozvojem moderních technologií webu je možné využít výpočetní kapacitu klientů v tzv. oblasti volunteer computing. Cílem práce je navrhnout, implementovat a otestovat framework pro distribuované výpočty v prostředí webu s využitím prohlížečů či strojů uživatelů jako výpočetní uzly.

- Proveďte rešerši v oblasti tzv. volunteer computing. Zaměřte se zejména na možnosti využití strojů návštěvníků webu.
- Analyzujte vlastnosti prostředí webu a s využitím rešerše:
  - navrhněte řešení problematiky spojené s prostředím webu jako je distribuce dat či výpadky uzlů apod.
  - navrhněte framework zahrnující klientskou a serverovou stranu.
  - vyberte vhodné typy úloh.
- Zvolte vhodné nástroje, technologie a implementujte:
  - serverovou stranu sloužící zejména k zadání, distribuci a získání výsledků úloh.
  - klientskou stranu pro získání, provedení a zaslání výsledků úloh.
- Otestujte framework na modelových příkladech.
- Analyzujte výsledky a navrhněte případné zlepšení.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 29. ledna 2018

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práca

Framework pro distribuované výpočty v prostředí webu

Bc. Jakub Šiller

Vedúci práce: Ing. Jaroslav Kuchař, Ph.D.

2. mája 2018

Poďakovanie

Týmto by som chcel poďakovať Ing. Jaroslavovi Kuchařovi, Ph.D. za odborné vedenie pri tvorbe tejto práce. Chcel by som poďakovať aj vývojárom, ktorí prispievajú do fór, ako je napríklad Stack Overflow. Nakoniec by som chcel poďakovať mojej rodine, priateľke a kamarátom za podporu počas celého magisterského štúdia.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 2. mája 2018 .....

Odkaz na túto prácu

Šiller, Jakub. *Framework pro distribuované výpočty v prostředí webu.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Táto diplomová práca sa zaoberá návrhom a implementáciou frameworku, ktorý prostredníctvom webových prehliadačov využíva počítače návštevníkov webovej stránky ako výpočtové uzly. V práci sa nachádza analýza prostredia webu, súhrn doterajších prístupov a projektov, návrh vlastného riešenia a jeho implementácia. Práca popisuje riešenie pri výpadku výpočtového uzla, reakciu na pomalý výpočtový uzol, možnosti regulovania záťaže počítača návštevníka stránky frameworkom, možnosti využitia webových technológií asm.js a WebAssembly, ako aj stratégie distribúcie práce a bezpečnostné opatrenia frameworku. Na záver práca obsahuje vyhodnotenie experimentov a návrhy na zlepšenie frameworku.

Kľúčové slová Distribuované výpočty, výpočty vo webových prehliadačoch, výpočty na webe, volunteer computing, distribúcia práce, vlastnosti webu, asm.js, WebAssembly, wasm, Emscripten, TypeScript, Node.js

Abstract

This master thesis focuses on design and implementation of a framework that uses computers of website visitors as computing nodes through web browsers. It contains an analysis of the Web environment, summarization of previous approaches and projects, design and implementation of the framework. The work describes the solution of computing node failure, reaction to slow computing node, possibilities of controlling the load of the framework on a website visitor's computer, options of asm.js and WebAssembly technology utilization, strategies for work distribution and security of the framework. At the end of the thesis, the experiment results and proposal of improvements are listed.

Keywords Distributed computing, computation in Web browsers, computations in Web, volunteer computing, distribution of work, Web characteristics, asm.js, WebAssembly, wasm, Emscripten, TypeScript, Node.js

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

V dnešnej dobe je počítač, tablet, či smartphone neoddeliteľnou súčasťou života väčšiny ľudí. Tieto zariadenia sú čím ďalej výkonnejšie, avšak množstvo ľudí len málokedy využíva ich výpočtovú silu naplno. Vo svete tým pádom existuje veľký nevyužitý výpočtový potenciál.

Majitelia zariadení sú rôzni a aplikácie, ktoré majú nainštalované tiež. Jedna aplikácia ale nechýba takmer v žiadnom z nich. Je ňou webový prehliadač. Vďaka webovým štandardom, ktoré prehliadače dodržiavajú, nemusí webový vývojár takmer vôbec riešiť aký prehliadač užívateľ používa. Existujúce webové technológie sa neustále zdokonaľujú a zároveň vznikajú ďalšie, ktoré otvárajú nové možnosti. Tieto, ako aj ďalšie aspekty a trendy viedli k tomu, že webový prehliadač, ktorý pôvodne slúžil na statické zobrazovanie a prechádzanie webu sa zmenil na platformu pre aplikácie. Webový prehliadač je dnes jednou z najpoužívanejších a najuniverzálnejších aplikácií. Robíme pomocou neho každodenné aktivity osobného aj pracovného života. Používame ho na komunikáciu prostredníctvom emailov, chatov, či videohovorov, kancelársku činnosť ako je práca s textovým a tabuľkovým procesorom, sledovanie videí a filmov, hranie hier, čítanie novín, nakupovanie a množstvo ďalších aktivít.

Webový prehliadač sa javí ako ideálna platforma pre systém, ktorý by dokázal využiť nevyužitý potenciál užívateľských zariadení. V rámci svojej diplomovej práce som sa rozhodol takýto systém navrhnúť a implementovať.

Práca je rozdelená do niekoľkých kapitol. Ciele mojej diplomovej práce sú obsahom prvej kapitoly. V druhej kapitole popisujem predchádzajúce práce a projekty v oblasti využívania počítačov návštevníkov webovej stránky. Tretia kapitola obsahuje analýzu prostredia webu a návrh frameworku. V nej najskôr popíšem výpočtový model a skupiny užívateľov frameworku. Potom vysvetlím jeho časti a funkcie. V tejto kapitole vysvetlím tiež bezpečnostné opatrenia frameworku. Zvolené technológie a implementačné detaily sú obsahom štvrtej kapitoly. Piata kapitola je venovaná experimentom, ich výsledkom a návrhmi na zlepšenie.

Kapitola 1

Cieľ práce

Cieľom mojej diplomovej práce je na základe analýzy prostredia webu a rešerše v oblasti takzvaného *volunteer computing*, so zameraním hlavne na práce využívajúce počítače návštevníkov webových stránok, navrhnúť a implementovať prototyp frameworku, ktorý bude schopný pre výpočty využívať počítače návštevníkov webových stránok. Mojim cieľom je navrhnúť framework tak, aby čo najviac spĺňal požiadavky pre nasadenie v praxi. Framework preto musí riešiť problematiky spojené s prostredím webu ako sú distribúcia práce, výpadok výpočtového uzla, pomalý výpočtový uzol a podobne. Framework by mal tiež obsahovať základné bezpečnostné mechanizmy.

Framework som obmedzil na typy úloh, v ktorých počítač návštevníka webovej stránky dostane úlohu na spracovanie, prevedie výpočet a vráti výsledok, pričom počas výpočtu pridelenej úlohy neprebieha komunikácia (týkajúca sa výpočtu) medzi počítačmi návštevníkov stránok a ani so serverom. Užívateľ frameworku môže okrem výpočtu, ktorý sa má vykonať, definovať aj takzvanú deliacu a zlučovaciu funkciu. Pomocou nich má byť framework schopný automaticky rozdeliť úlohu na podúlohy a výsledky podúloh zlúčiť do výsledku pôvodnej úlohy.

Dôležitým cieľom práce je aj experimentálne vyhodnotenie implementovaného frameworku na modelovej úlohe a návrh možností na zlepšenie.

Kapitola 2

Rešerš

2.1 Definícia distribuovaného výpočtu a volunteer computing

2.1.1 Distribuovaný výpočet

Autori práce [1] uvádzajú, že distribuovaný výpočet spočíva v riešení problému, ktorý je rozdelený na niekoľko častí, ktoré sú zvané podproblémy. Podproblémy sú vyriešené osobitne, zvyčajne na nezávislých procesoroch či počítačoch. Z výsledkov podproblémov sa potom vytvorí riešenie pôvodného problému.

V [2] sa popisuje distribuovaný systém ako sieť autonómnych počítačov, ktoré medzi sebou komunikujú pre dosiahnutie cieľa. Počítače sú nezávislé a fyzicky nezdieľajú procesor a pamäť. Medzi sebou komunikujú pomocou správ. Počítače v distribuovanom systéme môžu mať rôzne role.

2.1.2 Volunteer computing

Autori práce [3] popisujú takzvaný *volunteer computing* ako systém, ktorý využíva infraštruktúru, v ktorej rôzni ľudia poskytujú procesor pre zapojenie sa do spoločného výpočtu. Uvádzajú, že *volunteer computing* musí riešiť niekoľko problémov. Jedným z nich je to, že nie je možné predpovedať výkon systému, pretože nie je známy počet ani výkonnosť procesorov, ktoré sa do výpočtu zapoja. Ďalším problémom, ktorý je spojený s *volunteer computing* je, že je potrebné zabezpečiť, aby do výpočtu bolo zapojených dostatočne veľa počítačov.

V práci [4] sa uvádza, že prostredníctvom *volunteer computing* užívatelia môžu prispieť inak nevyužitými cyklami procesora k vyriešeniu výpočtovo náročnej úlohy. Distribuovaním práce do veľkej siete užívateľov môžu *volunteer computing* projekty využiť viac výpočtového výkonu, ako by sa dalo zhromaždiť lokálne vzhľadom na finančné a priestorové obmedzenia.

2.2 Predchádzajúce práce

Využívanie nevyužitého výpočtového výkonu nie je novinka. Podľa [5] prvý projekt, ktorý sa touto témou zaoberal bol projekt s názvom *Great Internet Mersenne Prime Search* [6] a bolo to už v roku 1996. Projekt sa zaoberá hľadaním prvočísel. V máji 1999 bol spustený projekt *SETI@Home* [7], ktorý sa zaoberá hľadaním mimozemskej inteligencie. O rok neskôr sa spustil aj projekt *Folding@home* [8] [9], ktorý využíva počítače dobrovoľníkov pre výskum v oblasti chorôb. Neskôr vznikol nástroj *BOINC (Berkeley Open Infrastructure for Network Computing)* [10]. *BOINC* je softvérová platforma pre *volunteer computing*. Následne vzniklo veľa projektov, ktoré *BOINC* využívajú [11]. Na zapojenie počítača do výpočtu v rámci spomínaných projektov je naň treba nainštalovať a spustiť klientský program. Všetky uvedené nástroje a projekty fungujú dodnes.

Využitím počítačov návštevníkov webových stránok sa ľudia zaoberali minimálne od roku 2007. Autori článku *Distributed Computing Through Web Browser* [1] z tohto roku uvádzajú, že ich práca je prvá štúdia o realizácii nového riešenia distribuovaných výpočtov s webom ako infraštruktúrou. Myšlienkou bolo realizovať distribuovaný výpočet v prostredí internetu s využitím len schopností webového prehliadača bez inštalácie ďalšieho softvéru na strane klienta. Cieľom práce bolo zistiť realizovateľnosť takéhoto riešenia distribuovaného výpočtu. Architektúra ich riešenia je klient-server a funguje tak, že klient si vyžiada od servera úlohu, vypočíta ju a odošle výsledok výpočtu. Tieto kroky potom dookola opakuje. Klientská časť je napísaná v jazyku JavaScript [12]. Aby zabránili blokovaniu práce užívateľa s prehliadačom, používajú autori namiesto klasických cyklov funkcie, ktoré emulujú cykly. Jednotlivé iterácie cyklu nie sú vykonané okamžite, ale až po určitej, parametrizovateľnej, dobe. Užívateľské rozhranie prehliadača potom môže využívať procesor v čase medzi iteráciami a teda nie je blokované výpočtom. Komunikácia so serverom prebieha pomocou technológie AJAX [13] a dáta sa posielajú vo formáte XML [14]. Server riadi rozdelenie problému na úlohy, ich distribúciu a prijímanie výsledkov. Server je implementovaný v jazyku Java [15]. Server si uchováva záznamy o prebiehajúcich výpočtoch, vďaka čomu môže prideliť stratenú úlohu novému klientovi. Za stratenú úlohu sa považuje úloha, od ktorej neprišiel výsledok do určeného časového limitu. Záver autorov je, že výsledky ukazujú, že takýto systém môže byť realizovaný s dobrou výkonnosťou, jednoduchou architektúrou, s použitím štandardných technológií a dosiahnuť celosvetové zapojenie. Upozorňujú však, že v práci neriešili viaceré aspekty, ako je napríklad bezpečnosť.

Cieľom práce s názvom *Browser-based Distributed Evolutionary Computation: Performance and Scaling Behavior* [3] bolo vytvorenie, prezentovanie a testovanie frameworku na distribuované evolučné výpočty s využitím webových prehliadačov. Framework má serverovú časť implementovanú pomocou Ruby on Rails [16] a klientskú časť implementovanú v jazyku JavaScript [12]. Z pohľadu klientskej časti framework funguje nasledovne. Klient si od servera vyžiada jedincov na spracovanie. Server pošle klientovi prednastavený počet jedincov. Klient vypočíta fitness funkciu pre každého jedinca a výsledky odošle na server. Tieto kroky sa opakujú, kým nie je splnená podmienka ukončenia. Zvyšok výpočtu prebieha na serveri. Framework rieši bezpečnosť povolením konkrétnych IP adries, ktoré sa môžu pripojiť. IP adresy sú dopredu známe, pretože testy frameworku prebiehajú v laboratórnych podmienkach. Autori článku testovali výkonnosť výpočtov v rozdielnych webových prehliadačoch a zistili, že výkony v rôznych prehliadačoch sa výrazne líšili. Urobili tiež experiment na vplyv veľkosti paketov na výkon frameworku. Z experimentov vyšiel výsledok, že čím použili väčšiu veľkosť paketu, tým lepší mal framework výkon. Výkon frameworku je v práci porovnaný aj s výkonom lokálneho výpočtu implementovanom na jednom počítači. Autori uvádzajú, že po odladení parametrov výpočtu a nastavení klientov a servera sú výkony podobné.

Distribuovanými evolučnými výpočtami, konkrétne genetickým programovaním, sa zaoberal aj článok *Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML* [17]. Pri implementácií využívali autori jazyk PUSH [18], ktorý bol vymyslený primárne pre využitie pri evolučných výpočtoch. V klientskej časti je využitý jazyk PushScript – implementácia jazyka PUSH v jazyku JavaScript. Systém funguje tak, že klient si vyžiada testy fitness a programy, vykoná ich a na server odošle výsledky. Tieto kroky sa dookola opakujú. Zvyšok funkcionality je implementovaný na serveri. Server využíva okrem PUSH implementovaného v C++ [19] jazyk PHP [20] pre komunikáciu s klientom. Autori uvádzajú, že v dobe písania článku (2007) JavaScript nemá dobrú povesť, čo sa výkonu týka a tiež uvádzajú veľké rozdiely vo výkone medzi prehliadačmi. Pri ich testoch na privátnom serveri vychádzali výsledky vytvoreného systému rádovo pomalšie, ako lokálny výpočet implementovaný v C++. Autori ale tvrdia, že by mohli existovať úlohy, pri ktorých by bol systém prakticky využiteľný. Hovoria napríklad o serveroch s vysokou premávkou (anglicky traffic), o výpočtovo náročných úlohách a o otvorených úlohách s neznámym riešením.

V roku 2008 vyšiel článok *Asynchronous Distributed Genetic Algorithms with Javascript and JSON* [21]. Oproti prechádzajúcim dvom prácam autori tohto článku vytvorili systém, ktorý veľkú časť práce preniesol zo servera na klientov. A to tak, že u každého klienta beží kompletný a nezávislý genetický algoritmus. Po definovanom počte generácii klient odošle najlepšieho jedinca z poslednej generácie na server. Server si ho uloží do databázy a klientovi odošle najlepšieho jedinca, ktorý je v databázy uložený. Tento jedinec sa začlení do populácie a klient pokračuje v algoritme. Server slúži na výmenu najlepších jedincov, nastavovanie parametrov a zber štatistík. Vďaka tomu, že server ukladá do databázy jedincov, ktorí prichádzajú od klientov, je zaručené, že najlepší jedinec nájdený v niektorom z klientov sa uchová na serveri. Server bol implementovaný v jazyku Perl [22] a klient v jazyku JavaScript [12]. Nábor užívateľov, ktorí poskytli počítať pre experimenty prebiehal pomocou emailov, aplikácie Twitter [23] a pomocou jedného španielskeho blogu. Do experimentov sa zapojilo niekoľko stoviek počítačov. Počet zapojených počítačov bol pri každom experimente iný. Väčšina experimentov mala päť až desať klientov. Autori uvádzajú, že experimenty ukázali, že pomocou *volunteer computing* sa dá dosiahnuť vysoký a do istej miery spoľahlivý výkon.

Práca *The Online Community Grid – Volunteer Grid Computing with the Web Browser* [5] sa venuje možnosti distribuovaného výpočtu na internete s využitím Adobe Flash [24]. Autor vytvoril pre klientskú stranu Adobe Flash doplnok (anglicky widget) s grafickým rozhraním. Doplnok slúži ako obal (anglicky wrapper), pre ktorý autor vytvoril API, ktoré majú k dispozícií programátori, ktorí chcú nástroj využívať. Autor však v práci API takmer nepopisuje a odkazuje na stránku *OnlineCommunityGrid.com*, ktorá je v dobe písania tejto práce nedostupná. Doplnok je vytvorený tak, že návštevník webovej stránky sa môže rozhodnúť, či sa zapojí do výpočtu a môže zvoliť projekt, do ktorého sa chce zapojiť. Môže tiež nastaviť koľko z výkonu procesora môže aplikácia využívať. Po zvolení projektu doplnok zo serveru stiahne kód jeho programu a program spustí. Počas výpočtu doplnok priebežne ukladá stav výpočtu do pamäte počítača návštevníka stránky. Ak návštevník odíde zo stránky, vo výpočte sa vďaka priebežnému ukladaniu môže pokračovať pri opätovnom navštívení stránky daným návštevníkom. Do grafického rozhrania autor pridal zobrazovanie noviniek z *Yahoo Top Stories*, aby motivoval správcov stránok umiestniť doplnok na stránku a aby zabavil návštevníkov a predĺžil tak ich pobyt na stránke. Autor práce nástroj netestuje na žiadnom ukážkovom príklade. Píše len o porovnaní výkonu jazyka Action Script [25] pre Adobe Flash v porovnani s výkonom nízko-úrovňových jazykov.

Webové prehliadače pre distribuované výpočty využíva aj framework predstavený v práci *MRJS: A JavaScript MapReduce Framework for Web Browsers* [26], ktorý ako výpočtový model zvolil MapReduce [27]. Vstupné dáta pre úlohu sú rozdelené na rovnako veľké časti. Počet týchto častí určuje zadávateľ úlohy. Autori ich nazývajú *WorkChunk*. *WorkChunk* je základná jednotka práce, ktorá môže byť pridelená klientovi. Obsahuje kód v jazyku JavaScript, index začiatku a konca dát, s ktorými sa má pracovať a URL adresu, z ktorej sa dajú dáta získať. Keď je *WorkChunk* pridelený klientovi na vykonanie, vytvorí sa takzvaný *ClientTask*. Do *ClientTask* sa potom uloží výsledok z vykonania *WorkChunk* klientom. Pre jeden *WorkChunk* sa môže vytvoriť niekoľko *ClientTask*. Vďaka tomu sa potom dajú detekovať falošné výsledky. Správny výsledok sa určí pomocou majoritného hlasovania.

Server úloh plánuje prideľovanie *WorkChunk* klientom. Pre každú úlohu sleduje, ktoré *WorkChunk* sú nekompletné. Server monitoruje počet *ClientTask* pre každý *WorkChunk* a keď si klient vyžiada prácu, pridelí mu *WorkChunk* s najmenším počtom *ClientTask*. Výsledok takéhoto plánovania je, že ak jeden klient zlyhá, výsledok môže prísť od iného klienta. Zároveň to zmierňuje negatívny vplyv málo výkonných klientov a s využitím majoritného hlasovania zvyšuje integritu výsledkov. Nevýhoda tohoto prístupu je, že úsilie vynaložené na výpočet jedného *WorkChunk* by sa mohlo použiť na výpočet viacerých *WorkChunk*. Tvorca úlohy môže určiť, koľko *WorkTask* sa má vytvoriť pre jeden *WorkChunk*. Východiskové nastavenie je jeden. Dizajn systému je navrhnutý tak, aby bolo možné načítavať dáta z externého zdroja.

Vykonávanie úloh má tri fázy. V prvej fáze server distribuuje *WorkChunk* s mapovacou funkciou a zbiera výsledky v podobe párov <kľúč, hodnota>. Keď server získa výsledky všetkých mapovacích *WorkChunk*, zoskupí páry podľa kľúča. Potom distribuuje *WorkChunk* s redukčnou funkciou. Keď získa výsledky zo všetkých redukčných *WorkChunk*, zapíše výsledky do súboru, ktorý si potom môže tvorca úlohy stiahnuť. Klient na vykonanie úlohy využíva technológiu Web Worker [28] a so serverom komunikuje pomocou technológie AJAX [13].

Autori uvádzajú, že výkonnosť frameworku závisí od typu úlohy. Výpočtovo náročné úlohy zvláda framework dobre na rozdiel od dátovo náročných úloh.

Ostrovný model zvolili pri tvorbe nástroja na distribuované evolučné výpočty autori článku *Distributed Evolutionary Computing System Based on Web Browsers with Javascript* [29]. Systém navrhli nasledovne. Klient sa pripojí na server otvorením webovej stránky, ktorá obsahuje vložený JavaScriptový kód na vykonanie genetického algoritmu, alebo algoritmu na lokálne hľadanie. Ak kli-

ent pri vykonávaní genetického algoritmu nájde lepšie riešenie ako dovtedy najlepšie, odošle ho na server. Ten ho uloží na vrch zásobníka nájdených najlepších riešení a potom toto riešenie distribuuje ostatným klientom. Distribúcia prebieha pomocou takzvanej *push* (tlačiacej) technológie, pri ktorej dáta klientovi server posiela bez toho, aby si ich klient musel vyžiadať. Autori ju implementovali pomocou takzvaného *long polling*. V pravidelných intervaloch posiela klient na pozadí na server požiadavku o dáta. Ak mu server dáta pošle, považuje sa to, akoby server poslal dáta sám od seba. Klient, ktorý má za úlohu vykonať lokálne hľadanie si najskôr vyžiada riešenie, ktoré má vylepšiť (zo zásobníka najlepších nájdených riešení). Server mu pošle jedno z riešení a u klienta prebehne lokálne hľadanie. Ak je riešenie vylepšené, odošle sa späť na server. Klientská aj serverová strana boli implementované v JavaScripte. Kód pre klienta bol predkompilovaný pre rôzne webové prehliadače. Ako server použili Node.js. Na testovanie bolo použitých 10 strojov. Na záver autori uvádzajú, že distribuované výpočty využívajúce JavaScript sú lacné, jednoduché na implementovanie a zároveň môžu byť dosť efektívne.

V článku *Web Browser-Based Social Distributed Computing Platform Applied to Image Analysis* [30] autori predstavili platformu pre spracovanie obrazu na veľkých dátach (anglicky big data). Architektúra platformy je klient-server. Ako prvé sa po pripojení klienta prevedie test výkonu zariadenia. Výsledok testu sa odošle na server, ktorý potom rozhodne, či klientovi priradiť úlohu. Od tohto momentu, ak je výkon zariadenia dostatočný, začína prenos dát medzi serverom a klientom. Server pošle klientovi úlohu, ktorá pozostáva z dát, algoritmu, formátu výsledku a podobne. Klient úlohy vykoná a výsledok odošle na server. Klient vykonáva výpočet vo vlákne na pozadí, čím neblokujú hlavné vlákno. Jedným z komponentov klientskej časti je kontrola výkonu. Tento komponent sleduje zaťaženie výkonu zariadenia a ak je príliš zaťažené, ostatné komponenty obmedzia svoju činnosť. Server je zodpovedný za distribúciu úloh, monitoring a zber výsledkov. Ak sa výsledok z úlohy správne neprijal, úloha sa odošle znova inému klientovi. Autori v závere článku tvrdia, že napriek nižšej efektivite v porovnaní s inými riešeniami (napríklad lokálny výpočet) dokáže takýto prístup riešiť big data problémy, ktorých riešenie bolo doposiaľ nedosiahnuteľné. A to vďaka obrovskému množstvu zariadení, ktoré je možné do systému zapojiť.

Myšlienkou využitia *volunteer computing* pri vyhľadávaní na webe na vyžiadanie (anglicky on-demand Web searching) sa zaoberajú autori článku *Web Pages Content Analysis Using Browser-Based Volunteer Computing* [31]. V prezentovanom riešení sa využíva veľké množstvo počítačov používateľov webu na

rýchle spracovanie webových stránok. Časť procesu spracovania stránok (spracovanie obsahu, detekcia URL a ďalšie) je vykonávaná v prehliadači užívateľa. Sever je zodpovedný za plánovanie úloh, rozosielanie stránok na spracovanie a zber výsledkov. Klient po pripojení dostane skript s definíciou vyhľadávacích pravidiel a implementáciou algoritmu na spracovanie textu. Potom od servera vyžiada stránku na spracovanie. Spracuje ju a vráti výsledky. Server bol implementovaný v jazyku PHP a klient v jazyku JavaScript. Autori uvádzajú, že výkon systému závisel najmä od počtu pripojených klientov a že pri dostatočnom počte klientov toto riešenie vie dosiahnuť vysoký výkon.

CrowdCL: Web-Based Volunteer Computing with WebCL [4] je názov článku, v ktorom autori predstavujú framework pre rýchly vývoj aplikácií na webe využívajúcich *volunteer computing* a OpenCL [32]. Vývojár využívajúci tento framework vytvorí JavaScriptovú triedu, ktorá reprezentuje inštanciu ľubovolného výpočtového problému. Konštruktor môže prijať ľubovolný počet parametrov. Trieda musí definovať minimálne metódu run. Táto metóda obsahuje logiku na získanie jedného riešenia problému. V kontexte optimalizačného problému metóda run reprezentuje jeden krok riešiteľa (anglicky solver). Framework potom spúšťa metódu run u klienta a výsledky posiela na server pomocou AJAX. Server frameworku je implementovaný ako Node.js aplikácia. Server je zodpovedný za zber výsledkov od klientov. Aby sa predchádzalo prijímaniu a ukladaniu falošných výsledkov môže vývojár definovať funkciu pre verifikáciu výsledku. Funkcia má na vstupe jeden výsledok a vracia pravdivostnú hodnotu. Ak vráti *false*, daný výsledok sa odstráni. Autori v článku predstavujú aj knižnicu *KernelContext*, ktorá má slúžiť pre jednoduché využitie grafickej karty na výpočty v prehliadači. Avšak táto knižnica stavia na technológií WebCL [33], ktorú prehliadače natívne nepodporujú.

Autori článku *Distributed Computing on an Ensemble of Browsers* [34] napísali hypotézu, že prehliadače sú budúcou platformou pre výpočty, ktorá by mohla pretransformovať internet na pravý distribuovaný počítač, ktorý by využíval nevyužívaný výpočtový výkon. Na dôkaz konceptu (anglicky proof-of-concept) implementovali prototyp frameworku. Klient pomocou technológie XHR [35] periodicky žiada server o prácu. Ak je práca k dispozícii, server odpovie popisom úlohy v XML. Popis úlohy obsahuje URL s JavaScriptovým kódom danej úlohy. Klient vytvorí nový objekt *Web Worker*, ktorý načíta a spustí kód z URL z popisu úlohy. Parametre z popisu úlohy sa odovzdajú do vytvoreného *Web Worker* objektu a spustí sa výpočet úlohy. Vypočítané výsledky odošle klient späť na server a okamžite si vyžiada ďalšiu úlohu. Ak nie je žiadna úloha k dispozícii, klient chvíľu čaká a potom znovu začne žiadať

server o prácu. Autori v práci píšu, že vďaka tomu, že väčšina prehliadačov si ukladá stiahnuté skripty do cache, vykonávanie rovnakých úloh viackrát s rôznymi vstupmi zníži záťaž siete vďaka tomu, že kód pre úlohu sa stiahne len raz. Autori uvádzajú, že neimplementujú bezpečnostné prvky, avšak JavaScript je vykonávaný v bezpečnom prostredí (anglicky sandbox), ktoré zabezpečuje prehliadač, takže bezpečnosť na strane klienta je taká dobrá ako ju zabezpečuje prehliadač. Navyše JavaScript spustený v objekte *Web Worker* nemá prístup k DOM [36]. Pomocou nastaviteľne dlhej pauzy medzi jednotlivými hlavnými cyklami frameworku je možné znížiť zaťaženie procesora frameworkom. Server v databáze drží zoznam úloh na spustenie u klienta. Pri odosielaní úloh na server môže užívateľ špecifikovať viacero hodnôt pre každý parameter. Server potom vytvorí karteziánsky súčin týchto hodnôt a pre každú n-ticu parametrov vytvorí novú úlohu.

QMachine je názov výpočtovej platformy poskytovanej ako webová služba, ktorú autori predstavili v článku *QMachine: commodity supercomputing in web browsers* [37]. Autori píšu, že architektúra *QMachine* nasleduje vzor Web 3.0 tak, že server slúži ako perzistentné úložisko a zvyšok logiky beží u klienta. *QMachine* pozostáva z troch hlavných komponentov: API server, web server a webová stránka. API server je implementovaný ako Node.js program a poskytuje funkcie pre pridanie úlohy, získanie úlohy pre výpočet a získanie stavu úlohy. Pri vytvorení úlohy sa vytvorí identifikátor úlohy. Klient, ktorý sa chce zapojiť do výpočtu úlohy musí tento identifikátor poznať a potom ho požíva ako API kľúč. Tým je možné zvýšiť bezpečnosť systému. Úloha pre *QMachine* pozostáva zo vstupných dát a transformácie, ktorá sa má vykonať. Framework umožňuje jednoduché transformácie a paradigmu MapReduce [27]. Web server je tiež implementovaný ako Node.js program. Jeho jedinou úlohou je poskytovať prezentačnú a analytickú vrstvu pre klientské počítače. Webová stránka slúži ako klient pre *QMachine* API, ktorý so serverom komunikuje pomocou XHR. Platforma umožňuje zadať pole adries s knižnicami, ktoré je treba pre vykonanie výpočtu úlohy. Framework zabezpečí, že uvedené knižnice sa u klienta pred výpočtom načítajú.

Ako názov napovedá, autori článku *MLitB: MACHINE LEARNING in the BROWSER* [38] sa zaoberali strojovým učením vo webových prehliadačoch. Systém je navrhnutý tak, aby umožňoval výskumníkovi vytvoriť projekt z oblasti strojového učenia vo webovom prehliadači. Potom sa do výpočtu riešenia problému môžu cez internet zapojiť rôzne zariadenia. Popisovaný framework sa riadi architektúrou master-slave a má niekoľko komponentov. Master server udržuje databázu problémov/projektov, spravuje pripojenia klientských za-

riadení a riadi hlavný cyklus udalostí (anglicky event loop). Nezávislý dátový server spravuje dáta a slúži na to, aby master server nebol zablokovaný pokiaľ by odosielal a prijímal dáta. Keď sa prehliadač pripojí k master serveru, spustí sa užívateľské rozhranie nazývané *boss*. Pomocou neho môže užívateľ vytvoriť nový projekt, načítať projekt zo súboru, nahrať dáta pre projekt, alebo pridať takzvaný *slave worker*. *Slave worker* môže slúžiť pre rôzne účely. Najdôležitejší účel je trénovanie, čo zahŕňa pripojenie sa do cyklu udalostí projektu a prispievanie do výpočtu. Užívateľ môže kedykoľvek odstrániť slave. Ďalšou možnou úlohou pre slave je monitorovanie. Slave si od master serveru vyžiada parametre modelu, čo umožní užívateľovi sledovať štatistiky modelu, alebo spustiť model. Komunikácia s master serverom prebieha pomocou technológie WebSocket [39] a komunikácia s dátovým serverom prebieha pomocou XHR. Celý framework bol implementovaný kompletne v jazyku JavaScript. Experiment škálovateľnosti demonštruje, že implementovaný prototyp zvládne obsluhovať do 64 klientov bez toho, aby latencia výrazne znížila výkonnosť. Najväčšia strata výkonnosti je spôsobená synchronizáciou v hlavnom cykle udalostí.

Článok *Gray Computing: An Analysis of Computing with Background JavaScript Tasks* [40] sa zaoberá oblasťou distribuovaných výpočtov v prostredí webu z pohľadu efektívnosti nákladov. Dizajn navrhnutého systému je zameraný na prostredie cloud. Berie do úvahy to, že náklady výpočtového času nie sú fixné, ale závisia od záťaže. Pri navrhovaní systému ďalej využívajú asymetriu v cenách medzi nahrávaním a sťahovaním. Záťaž na serveroch pre distribúciu úloh znižujú využitím siete obsahu – CDN [41]. Systém je zameraný na model MapReduce [27].

Server pre distribúciu úloh rozdelí dáta na malé kúsky a priraďuje ich klientom na spracovanie. Autori zvolili AmazonS3 ako dátový server na uchovávanie vstupných dát pre klientov a prijímanie výsledkov. Amazon EC2 server používajú pre rozdeľovanie dát a správu radu úloh. V predstavovanom riešení sa úlohy distribuujú nasledovne. Klient si od servera vyžiada HTML stránku. Server vloží do stránky prídavný JavaScriptový súbor, ktorý obsahuje úlohu. Klient si pomocou technológie AJAX vyžiada dáta z CDN. Keď prijme dáta vykoná mapovaciu a/alebo redukčnú funkciu. Následne vykoná výpočet a výsledok odošle na dátový server. Klient si potom znovu vyžiada ďalšie dáta z CDN na spracovanie.

Práca sa zaoberá aj tým, ako úlohy prideľovať klientom. Autori práce navrhli a implementovali adaptívny plánovač. Využívajú to, že podľa [42] majú dĺžky trvania návštevy stránok Weibull distribúciu. A pravdepodobnosť, že návštevník odíde sa časom zmenšuje. Plánovač navrhli tak, že čím je dĺžka návštevy dlhšia, tým pošlú klientovi viac dát na spracovanie, až kým veľkosť dát nenarazí na definovanú hranicu.

Autori skúmali aj výkonnosť JavaScriptu. Pomocou nástroja Emscripten [43] transformovali kód z jazyka C++ na kód v jazyku asm.js [44], čo je vysoko efektívna podmnožina JavaScriptu. Testovali rýchlosť výpočtu implementovaného pomocou klasického JavaScriptu, C++ a asm.js. Autori uvádzajú, že preložený kód z C++ do asm.js je výrazne rýchlejší ako implementácia v klasickom JavaScripte a je dvakrát pomalší ako implementácia v C++. Autori sa zaoberali aj otázkou spracovania falošných výsledkov a tvrdia, že efektívny prístup je replikácia a majoritné hlasovanie.

Využívaním prehliadačov pri distribuovaných výpočtoch sa zaoberá aj práca *Browser-Based Harnessing Of Voluntary Computational Power* [45]. Autori v práci popisujú framework s názvom *PovoCop*. Uvádzajú päť základných entít frameworku. Majiteľ úlohy (1) je osoba, ktorá vytvára JavaScriptový kód, ktorý sa vykoná u klientov. Majiteľ úlohy vytvorí úlohu, ktorej dáta môže framework rozdeliť a vytvoriť tak podúlohy. Každá podúloha sa spustí v jednom objekte *Web Worker* v klientskom prehliadači. Počet úloh, ktoré sa spustia v klientskom prehliadači, závisí od počtu jadier procesora daného počítača. Náborár (2) je server s vlastným obsahom a s vloženým skriptom poskytnutým frameworkom. Prehliadače, ktoré sa pripoja k tomuto serveru, potom začnú načítavať a vykonávať podúlohy a stanú sa takzvanými plantážami (3). Plantáž prijíma podúlohy, vykonáva ich a odosiela výsledky. Adresárová služba (4) je server s databázou úloh, ktoré čakajú na vykonanie. Úloha obsahuje kód, adresu plánovača a ďalšie informácie o úlohe, ale neobsahuje dáta. Plánovač (5) je server, ktorý distribuuje podúlohy respektíve ich vstupné dáta. Okrem toho monitoruje výpočty.

Framework využíva rýchly test na zistenie počtu jadier procesora a rýchlosť jedného procesora. Na komunikáciu medzi klientom a servermi zvolil autori technológiu WebSocket [39].

Autori sa v práci zaoberajú aj bezpečnosťou systému. V rámci toho píšu, že aby sa znížilo riziko zneužívania platformy, môžu byť úlohy kontrolované manuálne a/alebo testované predtým, ako sa akceptujú k distribúcii. Uvádzajú príklad, kde XtremWeb [46] umožňoval pracovať len s databázou dôveryhodných úloh. Úlohy boli akceptované len od dôveryhodných inštitúcií.

Autori po experimentoch konštatujú, že prototyp dobre zvládal algoritmus, ktorý nemá veľké množstvo dát na vstupe. Ďalej tvrdia, že podúlohy by mali byť čo najkratšie, pretože server tak získa výsledky viacerých výpočtov v prípade, že stratí spojenie s prehliadačom.

V oblasti využívania počítačov návštevníkov webových stránok vznikli aj ďalšie práce. Napríklad *Client-Side Processing Environment Based on Compo-*

nent Platforms and Web Browsers [47] je práca, ktorá používa na serveri Javu a u klienta Java applet. Juan-J. Merelo spolu s ďalšími autormi publikovali okrem [3] a [21] aj články [48] a [49] o využití webových prehliadačov pri distribuovaných evolučných algoritmoch. Využitie webových prehliadačov pre distribuované výpočty spomína aj práca [50], ktorá sa ale najmä venuje predstaveniu knižnice pre prácu s maticami a takzvaným *deep learning*.

Na internete sa dá taktiež nájsť niekoľko knižníc, ktoré majú slúžiť na distribuované výpočty v prehliadačoch. Sú to napríklad *deThread* [51], *dis.io* [52], *BoomerangJS* [53], *comp-pool* [54], *js-spark* [55], *Workhorse* [56] a ďalšie. Jedná sa však zväčša o projekty, ktoré sa už (podľa ich repozitárov na portály GitHub [57]) nevyvíjajú niekoľko rokov a ich čísla verzií sú väčšinou typu 0.x.y. Uvedené knižnice zväčša neriešia výpadky uzlov, bezpečnosť, verifikáciu výsledkov a podobne.

2.3 Dolovanie kryptomien

S nárastom popularity kryptomien a vývojom webových technológií začali vznikať komerčné nástroje na dolovanie kryptomien v prehliadačoch návštevníkov webových stránok. Pravdepodobne najrozšírenejším takýmto nástrojom je *Coinhive* [58]. Pomocou neho môže správca webového portálu využívať počítače návštevníkov na ťažbu kryptomeny Monero [59]. *Coinhive* umožňuje správcovi nastaviť počet jadier, ktoré má u návštevníka využívať a tiež podiel času medzi výpočtami a pauzy medzi jednotlivými výpočtami. Ďalej sa dá nastaviť, či sa má výpočet spustiť ak v inom okne alebo karte už výpočet beží. Podľa dokumentácie tento nástroj v závislosti od nastavenia využíva technológiu WebAssembly [60] alebo asm.js. Serverová časť systému je spravovaná na strane *Coinhive*. Aby mohol správca webu *Coinhive* využívať, musí sa zaregistrovať, čím získa API kľúč. Podľa [61], webový portál, ktorý má 10–20 stálych návštevníkov, môže zarobiť okolo 30 dolárov mesačne.

Podobne ako *Coinhive* fungujú aj ďalšie nástroje ako napríklad `webmine.cz`, `pop-coin.co`, či `jsecoin.com`.

Okrem komerčných nástrojov existuje aj nástroj *CryptoNoter* [62], ktorý je open source. Na komunikáciu medzi serverom a klientmi využíva WebSocket a na výpočty WebAssembly.

Taktiež vznikli webové portály, ktoré sú zamerané na to, že ich návštevník sa môže vedome a explicitne zapojiť do ťaženia kryptomien, z čoho má potom priamo profit. Príkladom takého portálu je napríklad `brominer.com`.

S príchodom týchto nástrojov však prichádzajú aj doplnky pre prehliadače, ktoré sa snažia tieto výpočty blokovať. Tvorcovia nástrojov na ťaženie preto ponúkajú nastavenia správcom webových portálov, ktoré podmieňujú výpočty tým, že návštevník portálu musí explicitne odsúhlasiť používanie. Takto odsúhlasené výpočty potom už blokovacie nástroje často ponechajú bežať.

2.4 Projekt Computes

Iný prístup ako doteraz popísané práce zvolili autori projektu *Computes* [63]. Namiesto klasickej centralizovanej klient-server architektúry vyvíjajú nástroj s decentralizovanou distribuovanou architektúrou. Snažia sa o vytvorenie super počítača s využitím čo najväčšieho spektra zariadení a platforiem od malých zariadení cez webové prehliadače až po servery. Na svojom blogu [63] uvádzajú, že vývoj tohto nástroja začal pred dva a pol rokmi. Verziu 1.0 opisujú ako centralizovanú distribuovanú platformu pre paralelné výpočty, ktorá bola navrhnutá tak, aby bežala na rôznych platformách: Windows, Mac, Linux, Android, iOS a vo webových prehliadačoch. Vo verzií 2.0 zlúčili funkcionalitu distribúcie úloh a výpočtu úloh tak, aby ktorýkoľvek uzol platformy mohol distribuovať úlohy a/alebo prevádzať výpočet. Uzly, ktoré distribuovali úlohy, museli byť ale stále aktívne a online pre distribúciu dát a zber výsledkov. Aktuálnu verziu 3.0 tvorcovia popisujú ako plne decentralizovanú a distribuovanú výpočtovú architektúru. *Computes* nevyužíva protokol HTTP [64], ale je postavené na technológií IPFS [65] a píšu, že vďaka tomu vie byť ich distribuovaný stroj vysoko výkonný. Na projekte pracuje niekoľko vývojárov a aktuálne prebieha beta testovanie.

2.5 Zhrnutie

Uvedené práce sa zameriavali na rôzne typy úloh distribuovaných výpočtov s využitím prehliadačov. Napríklad práca [3] sa zaoberala evolučnými výpočtami. Na spracovanie obrazu sa zamerali autori [30]. Strojovému učeniu sa venovala práca [38]. V [31] vytvorili nástroj na prehľadávanie webu. MapReduce framework bol predstavený v [26]. A sú aj práce, ktoré neboli zamerané na špecifický druh úloh napríklad [45].

Okrem tvorcov projektu *Computes* [63], ktorí zvolili decentralizovanú distribuovanú architektúru používali práce architektúru klient-server.

Pre implementáciu klientskej časti používala väčšina autorov jazyk JavaScript, ale boli aj práce, ktoré zvolili inú technológiu. Napríklad Adobe Flash v práci [5], či Java applety v [47].

Pre serverovú časť boli zvolené technológie rôznorodejšie. Jazyk Java využívali autori [1], Ruby bol použitý v [3], práca [31] zase popisovala riešenie za použitia PHP, autori [21] zvolili Perl a viackrát bol použitý JavaScript aj na strane servera. A to napríklad v práci [29], [4], či [37].

Pre komunikáciu medzi klientom a serverom používali práce XHR, napríklad v [34], alebo WebSocket, ten bol použitý v [38].

Pre overenie výsledkov použili práce [26] a [40] majoritné hlasovanie. Práca popísaná v [4] umožňovala zadať funkciu na verifikáciu výsledkov.

Aby neovplyvňovali činnosť užívateľa, autori práce [1] používali emulované cykly, ktoré zabezpečovali, že stránka sa nezablokuje. Autori novších prác, ako

napríklad [34] a [38] používali technológiu Web Worker.

V prácach autori riešili rôzne problémy a situácie, ktoré môžu nastať, ako napríklad výpadok uzlov, falošný výsledok, bezpečnosť systému a podobne, ale vo väčšine prác autori skôr skúšali či je možne takýto framework vytvoriť a nevenovali sa komplexnému riešeniu frameworku, ktorý by sa mohol nasadiť produkčne

Kapitola 3

Analýza a návrh

3.1 Vlastnosti prostredia webu

Prostredie webu má niekoľko výrazných charakteristík. Jednou z nich je *rôznorodosť*. Užívatelia používajú rôzne verzie rôznych webových prehliadačov. Štatistiky z rozličných zdrojov ([66], [67], [68], [69]) sa síce líšia v konkrétnych číslach, ale uvádzajú 5–10 majoritných prehliadačov. Menovite sú to Chrome [70], Firefox [71], Internet Explorer[72], Edge [73], Opera [74], prehliadač pre Android [75], Safari [76], Samsung Internet [77], UC Browser [78]. Rôzne sú aj výkony počítačov a rýchlosti pripojenia, ktoré závisia od krajiny, v ktorej sa zariadenie nachádza [79] [80], ako aj od typu pripojenia.

Ďalšou charakteristikou, ktorú môžme v prostredí webu pozorovať sú ale aj *špecifikácie a štandardy*. Vytváranie a dodržiavanie štandardov najmä zo strany tvorcov webových prehliadačov uľahčuje prácu vývojárom, ktorí potom nemusia vyvíjať pre každý prehliadač zvlášť.

Dôležitou vlastnosťou webu je aj to, že je *verejný* – každý sa môže pripojiť, získavať a publikovať dáta.

Web a jeho prostredie sú z viacerých pohľadov veľmi *dynamické*. Technológie webu sa intenzívne vyvíjajú a priebežne vznikajú nové. Vzniká aj veľa technológií pre podporu vývoja. Napríklad vzniklo viacero jazykov určených pre vývoj, ktoré sa následne preložia do jazyka JavaScript. Medzi najznámejšie patria Elm [81], TypeScript [82], ClojureScript [83], CoffeeScript [84], PureScript [85], či Kotlin [86]. Web je dynamický aj z pohľadu chovania užívateľov. Jakob Nielsen v [87] uvádza, že návštevníci webovej stránky často odchádzajú po 10–20 sekundách a priemerná návšteva stránky trvá menej ako minútu. Dynamickosť webu sa prejavuje aj v samotnom obsahu webových stránok, ktorý je na množstve stránok tvorený veľkým počtom užívateľov. Takými stránkami sú napríklad sociálne siete ako Facebook [88], ale aj rôzne komunitné stránky a fóra ako napríklad Stack Overflow [89]. Okrem toho exitujú ďalšie druhy stránok, ktorých obsah primárne tvoria určení autori, ktorí cielene často menia, respektíve vytvárajú nový obsah stránok. Takýmito stránkami sú aj

spravodajské portály, pri ktorých sa dá všimnúť trend veľmi častého publikovania krátkych správ. Napríklad Denník N [90] publikuje niekoľko správ alebo článkov za hodinu.

V neposlednom rade, jeden z najvýraznejších aspektov webu je jeho *veľkosť*. Podľa [91] a [92] je užívateľov webu viac ako 3,7 miliárd a toto číslo každoročne rastie.

3.2 Dôsledky vlastnosti webu na framework

Uvedené charakteristiky treba brať do úvahy pri návrhu frameworku pre lepšiu identifikáciu, riešenie, či predchádzanie možným problémom, ale aj pre efektívne využitie webu a jeho technológií. Vlastnosti webu poukazujú na niekoľko potencionálnych problémov alebo situácií, ktoré by framework mal byť schopný riešiť. Z uvedených vlastností vyplývajú nasledovné požiadavky na framework:

- framework by mal byť schopný bežať v čo najväčšom počte používaných prehliadačov vrátane ich rôznych verzií, preto by mal používať štandardy implementované v týchto prehliadačoch
- kvôli krátkym dobám návštev webových stránok môže často dochádzať k výpadkom výpočtových uzlov, framework s tým musí počítať a vedieť takéto situácie riešiť
- framework by mal brať ohľad na to, že výkony počítačov a rýchlosti pripojenia užívateľov webu sa môžu výrazne líšiť
- vzhľadom na otvorenosť a verejnosť webu by framework mal riešiť bezpečnosť na strane serveru aj na strane klienta
- framework by mal zvládnuť obsluhovať veľký počet požiadaviek
- framework by mal zabezpečiť dôveryhodnosť a správnosť výsledkov

Nakoniec framework by mal pracovať tak, aby neobmedzil užívateľa webu v práci, ideálne aby z užívateľského pohľadu nebolo možné určiť, či framework pracuje alebo nie.

3.3 Návrh frameworku

3.3.1 Výpočtový model frameworku

Distribuované výpočty sa môžu využívať na riešenie veľkej škály problémov rozličnými spôsobmi. Systém, ktorý slúži na distribuované výpočty môže mať

centralizovanú, alebo decentralizovanú architektúru. Môže poskytovať relatívne nízko-úrovňové funkcie ako napríklad Open MPI [93], alebo vysoko-úrovňové funkcie ako napríklad MapReduce frameworky [27].

Po diskusii s vedúcim práce a na základe vyššie popísaných prác a vlastností prostredia webu som sa rozhodol použiť centralizovanú architektúru klient-server a obmedziť framework na typy úloh, v ktorých klient dostane úlohu na spracovanie, prevedie výpočet a vráti výsledok, pričom počas výpočtu pridelenej úlohy neprebieha komunikácia (týkajúca sa výpočtu) medzi klientmi ani medzi klientom a serverom. Programátor ale môže okrem výpočtu, ktorý sa má vykonať, definovať aj takzvanú deliacu a zlučovaciu funkciu. Pomocou deliacej funkcie framework automaticky rekurzívne rozdelí veľké dáta na menšie a vytvorí tak podúlohy. Tieto podúlohy potom rozdistribuuje medzi klientov a pomocou zlučovacej funkcie zlúči čiastočné výsledky podúloh do výsledku pôvodnej úlohy. Tento model výpočtu úlohy je na obrázku 3.1.

3.3.1.1 Typy úloh vhodné pre framework

Z výpočtového modelu vyplýva, že framework je vhodný pre rekurzívne úlohy alebo úlohy iného typu, ktoré sa dajú rozdeliť na podúlohy, a zlúčením výsledkov podúloh sa vytvorí výsledok pôvodnej úlohy.

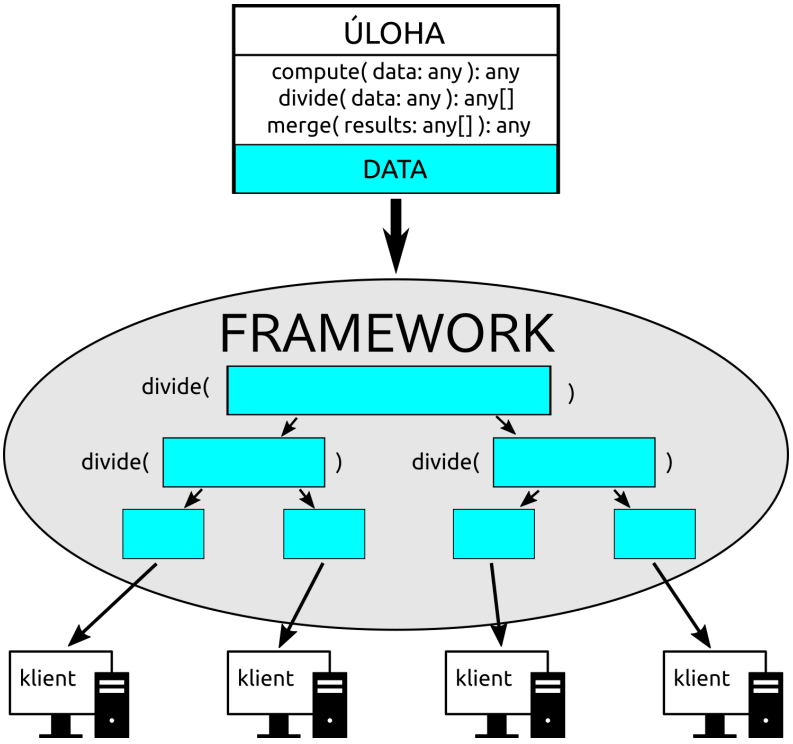
Framework za účelom výpočtu výsledku úlohy bude často posielať dáta medzi serverom a klientom. Prenos dát po sieti závisí od pripojenia servera aj klienta a môže byť časovo náročný. Z toho vyplýva, že framework je vhodný pre výpočtovo náročné úlohy. A teda na úlohy, ktorých výpočet má vysoký podiel operácií na jednotku dát. Inak povedané, framework je vhodný pre úlohy, ktorých časová náročnosť výpočtu je výrazne vyššia ako časová náročnosť prenosu dát. Pritom ale výpočet úlohy, prípadne jej podúloh, ktoré sa už ďalej nedajú rozdeliť na menšie, nemôže trvať príliš dlho, pretože by sa nemuseli stihnúť vypočítať počas žiadnej z náštev webovej stránky, počas ktorých budú výpočty prebiehať.

Použitie frameworku môže byť výhodnejšie oproti lokálnemu výpočtu aj pri výpočte veľkého množstva malých úloh, ktoré sa vďaka frameworku budú vykonávať súčasne na viacerých klientských počítačoch.

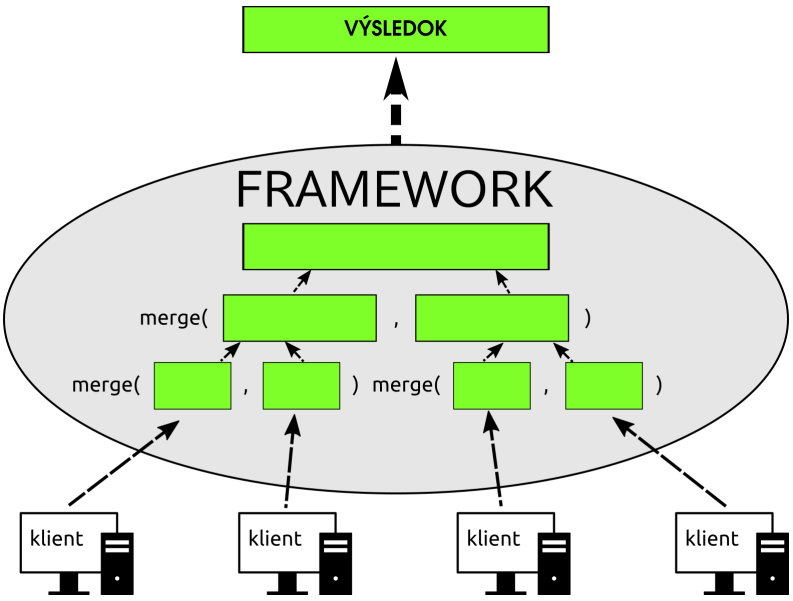
3.3.2 Užívatelia frameworku

Užívateľov frameworku som rozdelil na dve skupiny. Jednu skupinu tvoria tvorcovia prototypov úloh. O nich sa predpokladá, že dobre poznajú framework, jeho silné a slabé stránky, použité technológie a podobne. Ich úlohou je vytváranie prototypov úloh. Prototypom úlohy myslím definíciu (kód) funkcie na spracovanie dát u klienta, prípadne definície (kód) voliteľnej deliacej a zlučovacej funkcie.

Druhou, hlavnou skupinou sú bežní užívatelia, ktorí využívajú framework pre spracovanie dát. Úlohu na spracovanie vytvoria tak, že na server odošlú



(a) Vizuálna ukážka prvej fázy výpočtového modelu frameworku. V nej sa dáta úlohy rozdelia na menšie podúlohy a tie sa distribuujú klientom pre výpočet. Funkcia výpočtu je definovaná užívateľom (funkcia *compute*). Delenie prebieha automaticky pomocou užívateľom definovanej funkcie (funkcia *divide*).



(b) Vizuálna ukážka druhej fázy výpočtového modelu frameworku. V nej framework prijme výsledky výpočtov od klientov a zlúči ich do finálneho výsledku úlohy. Zlučovanie prebieha automaticky pomocou užívateľom definovanej funkcie (funkcia *merge*).

Obr. 3.1: Vizuálna ukážka výpočtového modelu frameworku.

dáta spolu s identifikátorom prototypu úlohy, ktorú vytvoril užívateľ z prvej skupiny. Títo užívatelia nepotrebujú vedieť o frameworku skoro nič. Stačí im zoznam prototypov úloh, ktoré môžu využiť.

Takéto rozdelenie som zvolil z niekoľkých dôvodov. Prvým z nich je bezpečnosť. Dá sa predpokladať, že tvorcov prototypov úloh bude rádovo menej ako ostatných užívateľov. Preto ich nebude priveľmi ťažké preveriť z pohľadu bezpečnosti (overenie totožnosti; overenie, či sú súčasťou dôveryhodnej inštitúcie; a podobne). Po takejto „bezpečnostnej previerke“ môžem do istej miery považovať tvorcov úloh a ich kód odoslaný na server za dôveryhodný. Ak by bolo treba ešte vyšší stupeň ochrany, mohli by sa prototypy úloh pred uložením do databázy aj manuálne overovať. Princíp databázy dôveryhodných úloh sa spomína aj v [45].

Ďalším dôvodom pre toto rozdelenie je praktickosť využitia frameworku. Pri takomto modele nemusí bežný užívateľ vedieť o frameworku skoro nič a dokonca mu na jeho využívanie stačia minimálne znalosti programovania. Zároveň sa tým zjednodušuje využitie frameworku z rôznych programovacích jazykov, pretože jediné čo treba na server odoslať pre výpočet úlohy sú dáta a identifikátor prototypu úlohy.

Efektivita, (bez)chybnosť a vhodnosť úlohy sú tiež dôvodmi pre takéto rozdelenie. Predpokladám, že pri takomto rozdelení, kedy skupina užívateľov dobre pozná detaily systému a zameriava sa len na tvorbu prototypov úloh a teda na tvorbu kódu, ktorý sa vykoná, bude vytvorený kód efektívny, dobre otestovaný, odladený, s malým počtom chýb a nebudú sa vytvárať prototypy úloh, ktoré sú svojou povahou nevhodné pre framework.

Podobné rozdelenie užívateľov uvádzajú aj autori práce *Browser-Based Harnessing Of Voluntary Computational Power*.

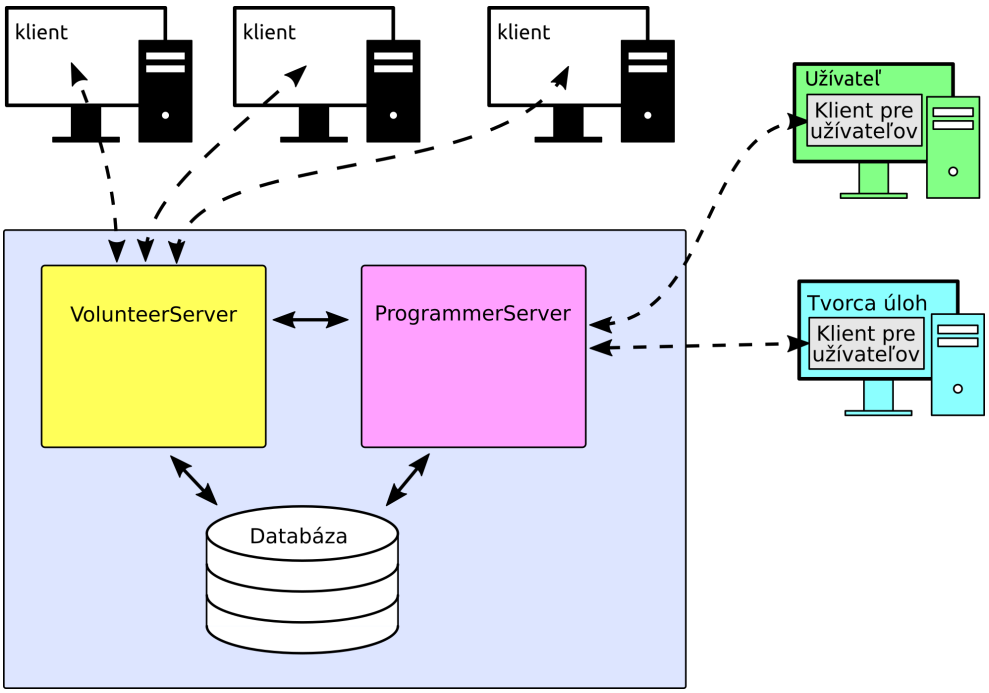
Aby boli tvorcovia úloh motivovaní vytvárať čo najkvalitnejšie prototypy úloh, mohol by sa okolo systému postaviť ekonomický model podobný trhu s aplikáciami ako je napríklad Google Play [94]. Tvorca úloh by potom získaval odmenu za používanie jeho prototypu.

3.4 Základná architektúra systému

Základná architektúra frameworku je na obrázku 3.2. Skladá sa z dvoch hlavných komponentov – klientskej časti a serverovej časti. Okrem toho framework obsahuje malú knižnicu pre užívateľov frameworku.

3.4.1 Klient pre užívateľov frameworku

Klient pre užívateľov frameworku je malá knižnica, ktorá zabezpečuje komunikáciu so serverom. Umožňuje získať zoznam prototypov úloh uložených na



Obr. 3.2: Základná architektúra frameworku.

Serverová časť pozostáva z dvoch hlavných komponentov, ktoré spolu zdieľajú databázu. *ProgrammerServer* sa stará o komunikáciu s užívateľmi frameworku, zatiaľ čo *VolunteerServer* je zodpovedný za komunikáciu s výpočtovými klientmi a výpočet úloh.

serveri, odoslať prototyp úlohy, odoslať úlohu, získať informácie o odoslanej úlohe a prevziať výsledok úlohy.

3.4.2 Výpočtový klient

Výpočtový klient je program, ktorý sa spustí na navštívenej webovej stránke a zmení kartu webového prehliadača na výpočtový uzol frameworku. Zabezpečuje komunikáciu so serverom, prijatie, výpočet a odoslanie výsledku úloh.

3.4.3 Server

Serverová časť je rozdelená na dve časti, ktoré zdieľajú databázu. Jedna časť sa nazýva *ProgrammerServer* a slúži ako prístupový bod pre užívateľov frameworku. Umožňuje prijať a spracovať prototypy úloh, prijať úlohy a odoslať výsledky. Ďalej poskytuje informácie o prototypoch úloh a o stave úloh.

Druhá časť sa nazýva *VolunteerServer*. Ten je srdcom frameworku. Je zodpovedný za prípravu úloh na distribúciu, distribúciu úloh a prijímanie a spracovanie výsledkov.

3.5 Význam slova klient

V nasledujúcich sekciách a kapitolách budem opisovať funkcie frameworku. Počas toho budem používať slovo *klient*. To môže mať niekoľko významov, ktoré sa prelínajú. Prvým významom je klient ako jedna z dvoch hlavných častí architektúry klient-server. V druhom význame ide o spustenú inštanciu programu klientskej časti v rámci architektúry klient-server. Pre zvýraznenie toho, že ide o tento význam, explicitne píšem *výpočtový klient*. Slovo klient môže pomenovávať aj všetky inštancie programu klientskej časti v rámci architektúry klient-server spustené v rôznych oknách či kartách jedného webového prehliadača. A nakoniec slovo klient môže označovať webový prehliadač, v ktorom bežia inštancie klientskej časti v rámci architektúry klient-server spustené.

Verím, že na základe kontextu bude čitateľ vždy intuitívne rozumieť v akom význame je slovo *klient* použité.

3.6 Princíp fungovania frameworku v skratke

V tejto sekcii v skratke popíšem fungovanie frameworku pre ilustráciu celkového obrazu frameworku. Čitateľ tak bude mať lepšiu predstavu o kontexte pri popisovaní jednotlivých častí a funkcií frameworku.

Pre výpočet úlohy odošle užívateľ na server dáta spolu s identifikátorom prototypu úlohy, ktorý na server v minulosti odoslal tvorca prototypov úloh. Server požiadavku prijme a zaradí úlohu do kolekcie úloh. Ak v kolekcii, ktorá udržiava úlohy, ktoré sa práve distribuujú je dostatok miesta, úloha sa pripraví na distribúciu. Príprava na distribúciu spočíva najmä v rekurzívnom rozdelení úlohy na podúlohy nazývané práce. Keď od klienta príde požiadavka o pridelenie práce, server vyberie niekoľko prác a pridelí a odošle ich klientovi. Jednotlivé výsledky prijaté od klienta server nemôže považovať za dôveryhodné, respektíve správne a preto sa každá práca nechá vypočítať viackrát. Z prijatých výsledkov sa pomocou majoritného hlasovania určí správny výsledok. Majoritným hlasovaním myslím: Ak existuje výsledok, ktorý sa medzi prijatými výsledkami vyskytuje v počte, ktorý je väčší ako polovica počtu výsledkov, je tento výsledok prehlásený za výsledok práce. Ak taký výsledok neexistuje (napríklad, ak je každý prijatý výsledok iný), je treba znovu vykonať prácu a získať ďalší výsledok. Ak všetky práce úlohy, ktoré už nie sú ďalej rozdelené (listy v strome prác) majú majoritným hlasovaním určený výsledok, zlúčia sa výsledky jednotlivých čiastkových prác do výsledku úlohy, ktorá sa potom prehlási za dokončenú.

Klient frameworku v pravidelných intervaloch žiada o pridelenie práce. Ak mu práca bola pridelená, vykoná ju, odošle výsledok a znovu začne žiadať prácu.

3.7 Základný dátový model

Predtým ako začnem detailnejšie popisovať ako framework funguje, uvediem najdôležitejšie časti dátového modelu, aby som ich potom mohol používať pri vysvetľovaní.

3.7.1 Client

Táto trieda reprezentuje klienta v zmysle webového prehliadača návštevníka webovej stránky.

Medzi jej atribúty patria:

- *id* – identifikátor klienta
- *activeSessionId* – identifikátor relácie, v ktorej aktuálne prebieha výpočet
- *scheduledAt* – časová značka, kedy systém naposledy odoslal klientovi, respektíve niektorej z jeho relácií prácu
- *scheduleAfter* – časová značka, kedy najskôr má systém znovu odoslať klientovi, respektíve niektorej z jeho relácií prácu

3.7.2 ClientSession

Táto trieda reprezentuje reláciu, čiže návštevu stránky vo webovom prehliadači. Relácia začína načítaním webovej stránky a končí jej opustením (prechod na ďalšiu stránku, ukončenie prehliadača a podobne).

Medzi jej atribúty patria:

- *id* – identifikátor relácie
- *clientId* – identifikátor klienta (webového prehliadača), ktorého je súčasťou
- *lastAccess* – časová značka posledného prístupu, respektíve poslednej požiadavky na server
- *sessionStart* – časová značka začiatku relácie
- *currentWorks* – zoznam prác, ktoré relácia práve vykonáva alebo by mala vykonávať
- *wasmSupport* – informácia o tom či webový prehliadač, v ktorom beží relácia podporuje technológiu WebAssembly (skrátene wasm). Túto technológiu popisujem v sekcii 4.1.4

3.7.3 TaskPrototype

Trieda reprezentujúca prototyp úlohy.

Medzi jej atribúty patria:

- *id* – identifikátor prototypu úlohy
- *workingFunctions* – kód výpočtu, ktorý sa má u klienta vykonať, uložený v rôznych jazykoch spustiteľných u klienta
- *cplusplusWorkingFunction* – voliteľný kód výpočtu, ktorý sa má u klienta vykonať. Kód je v jazyku C++ a je pripravený na kompiláciu do jazykov asm.js a wasm
- *emscriptenFlags* – voliteľné parametre pre kompilátor Emscripten z C++ do asm.js a wasm
- *dividingFunction* – JavaScriptový kód voliteľnej deliacej funkcie
- *mergingFunction* – JavaScriptový kód voliteľnej zlučovacej funkcie
- *suggestedMaxDataSizeInBytes* – odporúčaná veľkosť dát pre jednu prácu odosielanú klientovi
- *postedBy* – identifikátor užívateľa, ktorý prototyp vytvoril

3.7.4 WorkingFunctions

Táto trieda slúži ako obaľovacia trieda pre kód výpočtu, ktorý sa má vykonať u klienta. Kód je uložený v rôznych jazykoch pre rôzne webové technológie.

Medzi jej atribúty patria:

- *jsWorkingFunction* – kód výpočtu v jazyku JavaScript, túto funkciu definuje užívateľ
- *asmJSWorkingFunction* – kód funkcie v jazyku JavaScript, ktorá slúži ako rozhranie medzi frameworkom a skompilovaným C++ kódom, túto funkciu tiež definuje užívateľ
- *asmJSCompiled* – skompilovaný C++ kód do jazyka asm.js
- *asmJSMemFile* – súbor vygenerovaný nástrojom Emscripten, ktorý využíva asm.js modul pri behu
- *wasmJSCompiled* – skompilovaný C++ kód do jazyka wasm
- *wasmJSCompiledRuntime* – kód behového prostredia pre wasm modul vygenerovaný nástrojom Emscripten

3.7.5 Task

Táto trieda reprezentuje úlohu zadanú užívateľom frameworku.

Medzi jej atribúty patria:

- *id* – identifikátor úlohy
- *taskPrototypeId* – identifikátor prototypu úlohy
- *state* – stav úlohy, ktorý môže byť: „čaká na distribuovanie“, „pripravuje sa k distribuovaniu“, „hotová“
- *data* – dáta úlohy
- *defaultReplicationFactor* – východiskový počet výsledkov práce, z ktorých má framework určiť správny výsledok, voliteľne ho môže užívateľ nastaviť
- *postedAt* – časová značka vytvorenia úlohy
- *resultHash* – haš výsledku úlohy
- *postedBy* – identifikátor užívateľa, ktorý úlohu odoslal

3.7.6 TaskUnderScheduling

Táto trieda reprezentuje úlohu, ktorej práce sa práve distribuujú medzi klientov. Túto triedu a jej vzťah k ďalším triedam zobrazuje obrázok 3.3.

Medzi jej atribúty patria:

- *taskId* – identifikátor úlohy
- *taskPrototypeId* – identifikátor prototypu úlohy
- *workTreeRoot* – koreň stromu prác, ktoré vznikli rekurzívnym delením úlohy na podúlohy
- *defaultWorkingFunctions* – kód výpočtu, ktorý sa má u klienta vykonať, uložený v rôznych jazykoch
- *created* – časová značka, kedy sa úloha pridala medzi úlohy, ktoré sa distribuujú

3.7.7 Work

Táto trieda reprezentuje prácu. Úloha môže obsahovať jednu prácu, alebo môže rekurzívnym delením dát úlohy vzniknúť viac prác. Tie potom reprezentujú čiastkové podúlohy. Túto triedu a jej vzťah k ďalším triedam zobrazuje obrázok 3.3. Medzi jej atribúty patria:

- *id* – identifikátor práce
- *taskId* – identifikátor úlohy
- *workUnit* – objekt, ktorý sa odošle klientovi na vykonanie výpočtu práce (viď. trieda *WorkUnit*)
- *result* – výsledok práce určený majoritným hlasovaním
- *resultHash* – haš výsledku
- *defaultReplicationFactor* – východiskový počet výsledkov práce, z ktorých má framework určiť správny výsledok majoritným hlasovaním
- *currentReplicationFactor* – počet výsledkov práce, z ktorých má framework určiť správny výsledok majoritným hlasovaním. Tento atribút je po vytvorení nastavený na hodnotu atribútu *defaultReplicationFactor*, ale počas výpočtu sa jeho hodnota môže zvyšovať
- *hasChildren* – informácia o tom, či práca je ďalej rozdelená alebo nie
- *remainingReplicasCount* – počet relácií, ktorým ešte treba prideliť prácu
- *tracks* – pole záznamov o pridelení prác (viď. trieda *Track*)
- *directResults* – pole výsledkov získaných od klientov obohatených o metadáta (viď. trieda *DirectResult*)
- *serverSideMetadata* – objekt, do ktorého môže deliaca funkcia uložiť metadáta, ktoré potom framework poskytne zlučovacej funkcií. Tieto metadáta ostanú uložené na serveri
- *addedAt* – časová značka vytvorenia práce na serveri

3.7.8 WorkTreeNode

Táto trieda slúži na usporiadanie prác úlohy do stromovej štruktúry. Reprezentuje jeden uzol stromu.

Medzi jej atribúty patria:

- *children* – potomkovia uzla
- *work* – práca aktuálneho uzla

3.7.9 WorkUnit

Trieda reprezentujúca objekt reprezentujúci prácu, ktorý je odoslaný klientovi.
Medzi jej atribúty patria:

- *workingFunctions* – kód výpočtu, ktorý sa má u klienta vykonať
- *workingFunctionsReqId* – identifikátor kódu výpočtu, ktorý sa ma u klienta vykonať
- *data* – dáta práce
- *workId* – identifikátor práce

3.7.10 Track

Táto trieda reprezentuje záznam o pridelení práce.
Medzi jej atribúty patria:

- *toSessionId* – identifikátor relácie, ktorej bola práca pridelená
- *at* – časová značka, kedy bola práca pridelená

3.7.11 WorkUnitResult

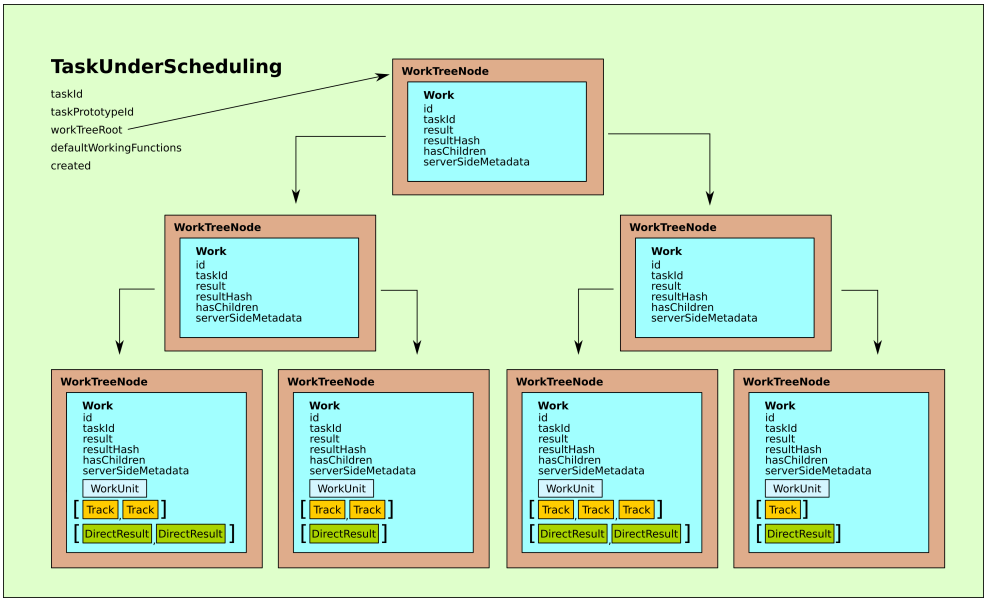
Táto trieda reprezentuje výsledok, ktorý klient odoslal na server.
Medzi jej atribúty patria:

- *result* – výsledok výpočtu práce
- *error* – chyba, ktorá prípadne nastala pri výpočte
- *workId* – identifikátor práce, ktorej je toto výsledok

3.7.12 DirectResult

Táto trieda reprezentuje výsledok prijatý od klienta obohatený o metadáta.
Medzi jej atribúty patria:

- *result* – výsledok práce
- *resultHash* – haš výsledku
- *fromSessionId* – identifikátor relácie, ktorá výsledok poslala
- *workId* – identifikátor práce, ktorej je toto výsledok
- *at* – časová značka prijatia výsledku



Obr. 3.3: Výňatok z dátového modelu frameworku

Diagram zachycuje vzťah tried *TaskUnderScheduling*, *WorkTreeNode*, *Work*, *WorkUnit*, *Track* a *DirectResult*. Každá *TaskUnderScheduling* obsahuje objekty *Work* reprezentujúce prácu. Pomocou objektov *WorkTreeNode* sú usporiadané v stromovej štruktúre tak, ako vznikli delením. Každý *Work* objekt obsahuje práve jeden objekt *WorkUnit*, ktorý obsahuje dáta a funkcie, ktoré sa odošlú klientovi. *Work* objekty si tiež držia pole záznamov o pridelení práce vo forme objektov *Track* a pole prijatých výsledkov obohatených o metadáta, ktoré reprezentujú objekty *DirectResult*.

3.8 Funkcie modulu ProgrammerServer

3.8.1 Prijatie a spracovanie prototypu úlohy

Keď *ProgrammerServer* prijme prototyp úlohy, skontroluje ho najskôr po formálnej stránke. To znamená, že overí, či je definovaná funkcia výpočtu, či už v databáze nie je prototyp s daným identifikátorom a podobne. Ak prototyp obsahuje kód výpočtu v jazyku C++, skompiluje ho pomocou nástroja Emscripten do jazykov wasm a asm.js a vygenerované kódy uloží do prototypu. Ak počas spracovania prototypu nastala chyba, informácia o nej sa zašle v odpovedi pre užívateľa.

3.8.2 Prijatie úlohy

Pri prijímaní úlohy *ProgrammerServer* skontroluje, či existuje prototyp, ktorý úloha vyžaduje. Ak neexistuje, odpovie užívateľovi, že zadaný prototyp neexistuje. Inak vytvorí novú úlohu, ktorej stav nastaví na „čaká na distribuovanie“

a uloží ju do databázy úloh. Užívateľovi vráti identifikátor, ktorý bol pre úlohu vygenerovaný. Pomocou neho potom užívateľ môže získavať informácie o úlohe a prevziať výsledok. *ProgrammerServer* po úspešnom prijatí úlohy upozorní *VolunteerServer*, že prijal úlohu.

3.9 Funkcie modulu VolunteerServer

3.9.1 Príprava úlohy na distribúciu

VolunteerServer udržiava kolekciu objektov *TaskUnderScheduling*. Tieto objekty reprezentujú úlohy, na ktorých výpočte framework aktuálne pracuje, čo znamená, že ich práce distribuuje klientom. Jedna *TaskUnderScheduling* môže obsahovať viacero prác, ktoré vznikli rekurzívnym delením dát úlohy. Práca teda môže byť rozdelená na menšie, čiastkové práce. Klientovi sa budú distribuovať len tie práce, ktoré už nie sú ďalej rozdelené. Inak povedané, distribuovať sa budú len práce z listov stromu prác. Kolekcia, ktorú server udržiava, obsahuje istý počet prác pre distribuovanie. V určitom momente niektoré z týchto prác už majú výsledok určený majoritným hlasovaním a už sa nebudú znovu distribuovať medzi klientov. V konfigurácií frameworku je možné nastaviť hranicu s názvom *worksMaxCount*. Ak počet prác, ktoré sa ešte budú klientom distribuovať, je pod touto hranicou, môže sa pre distribuovanie pripraviť ďalšia úloha.

O prípravu úloh sa stará modul s názvom *TasksPreparationManager*. Tento modul dostáva požiadavky typu „ak treba, priprav ďalšie úlohy na distribúciu“. „Ak treba“ znamená, že existuje úloha, ktorej práce sa ešte nedistribuujú (jej stav je „čaká na distribuovanie“) a zároveň počet prác, ktoré sa distribuujú medzi klientov je menší ako *worksMaxCount*. Ak toto platí, modul začne pripravovať úlohu na distribúciu. Ak je úloh čakajúcich na distribuovanie viac, modul pripravuje najskôr úlohy, ktoré server prijal skôr.

Pri začatí prípravy úlohy pre distribúciu *TasksPreparationManager* nastaví jej stav na „pripravuje sa na distribúciu“. Následne vytvorí nový objekt *TaskUnderScheduling*, ktorého základné atribúty nastaví podľa atribútov úlohy. Potom vytvorí strom prác nasledovne. Z úlohy najskôr vytvorí prácu úlohy – koreň stromu prác. Ak prototyp úlohy obsahuje deliacu funkciu, modul začne rekurzívne deliť dáta práce a vytvárať strom prác. Delenie sa ukončí, ak deliaca funkcia už nemá ako dáta rozdeliť, alebo ak veľkosť dát pre prácu je dostatočne malá. Hranicu pre veľkosť dát, ktorá sa považuje za dostatočne malú označujem *maxDataSizeInBytes* a je nastaviteľná v konfigurácii. Tvorca prototypu úlohy však môže túto hranicu upraviť pomocou atribútu prototypu *suggestedMaxDataSizeInBytes*. Nakoniec modul nastaví stav úlohy na „distribuuje sa“ a uloží ju do kolekcie objektov *TaskUnderScheduling*.

Modul pripravuje ďalšie úlohy kým platia predpoklady pre prípravu úlohy – existencia čakajúcej úlohy a počet prác pre distribuovanie je menší ako definovaná hranica.

3.9.2 Distribúcia prác

Práce, ktoré už nie sú ďalej rozdelené je treba distribuovať medzi klientov. O to sa stará takzvaný *Scheduler*. Keď od relácie príde na server požiadavka o pridelenie práce, server najskôr skontroluje, či relácia nemá pridelené práce, ktorých výsledok server ešte neprijal. Ak nemá, skontroluje či klient relácie má aktívnu reláciu. Záznam o aktívnej relácii slúži na to, aby sa u klienta vykonával výpočet vždy len v jednej relácii. Bližšie jej funkciu vysvetlím v sekcii 3.9.7. Ak klient nemá aktívnu reláciu, skúsi server nastaviť reláciu požiadavky ako aktívnu reláciu klienta. „Skúsi nastaviť“ a nie „nastaví“ preto, pretože súčasne môžu od jedného klienta prísť požiadavky z viacerých relácií, ktoré sa budú spracovávať naraz a je dôležité, aby sa nastavila len jedna z nich ako aktívna a aby bolo jasné, ktorá to je. Preto nastavenie aktívnej relácie musí byť atomická operácia typu otestuj-a-nastav. Ak sa nepodarilo nastaviť reláciu na aktívnu, alebo ak už server mal aktívnu reláciu, alebo relácia mala pridelenú prácu, ktorej výsledok ešte server neprijal, nepridelí sa relácii žiadna práca. Žiadna práca sa nepridelí relácii ani v prípade, že od posledného pridelenia prác jej klientovi, respektíve niektorej z jeho relácií neubehlo dostatočne veľa času. Toto opatrenie slúži na regulovanie výkonu počítača. Bližšie ho vysvetlím v sekcií 3.9.7.

Ak požiadavka o prácu prešla všetkými testami, pridelí jej *Scheduler* niekoľko prác. *Scheduler* najskôr určí, ktoré práce sa pridelia relácii. Prideľuje len tie práce, ktoré už nie sú ďalej rozdelené. Keď niektorú prácu *Scheduler* pridelí relácii, uloží o tom záznam v podobe objektu triedy *Track*. Do záznamu uloží kedy a komu sa práca pridelila. Pre každú distribuovanú prácu potrebuje framework získať výsledky z niekoľkých rôznych relácií, z ktorých potom majoritné hlasovanie určí správny výsledok. Treba preto prácu prideliť viacerým reláciám. Údaj o tom, koľkým reláciám ešte treba prácu prideliť sa udržiava v atribúte práce *remainingReplicasCount*. Pri každom pridelení práce relácii sa tento atribút dekrementuje. Situácie, v ktorých sa zvyšuje hodnota *remainingReplicasCount* popisujem v nasledujúcich sekciach.

Server do odpovede pre klienta pridá *WorkUnit* objekt každej pridelenej práce. Objekt *WorkUnit* obsahuje dáta a kód výpočtu práce v rôznych jazykoch. Podľa nastavenia frameworku a údaji o tom, aké technológie klient podporuje, sa zvolí technológia, ktorej kód sa ponechá a kódy pre ostatné technológie sa do odpovede nedostanú, aby sa neprenášali nevyužité dáta po sieti. Tento kód sa posiela s každým *WorkUnit*. Framework je možné nastaviť tak, že namiesto kódu sa pošle klientovi len identifikátor pre získanie kódu. Klient si potom pomocou tohoto identifikátora od servera vyžiada kód na vykonanie práce. Server do odpovede, v ktorej zašle kód pridá HTTP hlavičky s údajmi pre ukladanie odpovede do cache. Hlavičky bližšie popisujem v sekcii 4.2.6. Takéto nastavenie je výhodné, ak sa predpokladá, že jedna relácia bude často žiadať rovnaký kód. To nastane v prípade, ak práce prideľované reláciii sú z úloh, ktoré majú rovnaký prototyp, alebo je počet prototypov malý.

Toto nastavenie môže zvýšiť výkon frameworku aj v prípade, že je pre to vhodná topológia a vlastnosti siete, v ktorej je možné využiť cache medziľahlých sieťových prvkov. Požiadavka o kód na vykonanie potom často neopustí prehliadač, ale odpoveď sa načíta z cache prehliadača prípadne odpoveď vráti sieťový prvok zo svojej cache pamäti. Zníži sa tak záťaž serveru a množstvo dát, ktoré treba preniesť medzi serverom a klientom. Toto nastavenie môže byť ale aj kontraproduktívne a to v tom prípade, ak relácia bude vždy vykonávať iný kód, ktorý nebude v cache žiadneho sieťového prvku ani prehliadača. Klient teda bude musieť poslať pre každú pridelenú prácu požiadavku až na server a réžia okolo každej požiadavky bude u klienta aj na serveri trvať nejaký čas.

Ako a koľko prác sa relácii pridelí záleží od zvolenej stratégie a jej parametrov. Pre framework som navrhol niekoľko na seba naväzujúcich stratégií, ktoré teraz popíšem.

3.9.2.1 SuperSimpleScheduler

SuperSimpleScheduler je podtrieda triedy *Scheduler*, ktorá na pridelenie prác relácii využíva veľmi jednoduchú stratégiu. Najskôr si vyžiada práce pre distribúciu, ktorých *remainingReplicasCount* je viac ako 0 a ktoré ešte neboli priradené relácii, ktorá žiada o prácu. Práce zoradí podľa časovej značky vytvorenia, aby úlohy, ktoré sa začali distribuovať ako prvé boli aj vyriešené ako prvé. Nakoniec vyberie prvých n prác (ak toľko existuje), ktoré pridelí relácii. Parameter n je nastaviteľný v konfigurácii frameworku v atribúte s názvom *worksUnitsToClientCount*.

3.9.2.2 SimpleSizeBasedScheduler

Veľmi podobnú stratégiu som navrhol pre triedu *SimpleSizeBasedScheduler* s tým rozdielom, že nie je pevne stanovený počet prác, ktoré sa majú poslať ako odpoveď na jednu žiadosť o prácu, ale súčet ich veľkostí. Túto hodnotu som označil *defaultRecomendedWorkSizeInBytes*. Server zbiera štatistiky o veľkosti prác, ktoré sa klientom posielajú a na základe priemernej veľkosti správy a *defaultRecomendedWorkSizeInBytes* vypočíta, koľko prác predpokladá, že by mal odoslať relácii. Predpokladaný počet prác je určený vzťahom:

$$n = \lceil \frac{defaultRecomendedWorkSizeInBytes}{avgWorkSize} \rceil$$

kde $avgWorkSize$ je priemerná veľkosť prác, ktoré sa už distribuovali a n je počet prác, o ktorých sa predpokladá, že sa pošlú relácii. Môže sa stať, že v skutočnosti súčet veľkosti prác, ktorý sa bude *SimpleSizeBasedScheduler* chystať prideliť relácii bude väčší ako *defaultRecomendedWorkSizeInBytes*. Preto ešte pred odoslaním prác prebehne odstránenie niektorých prác. Práce sú stále zoradené podľa času vytvorenia. Odstraňujú sa práce od konca zoznamu (a teda

tie, ktoré boli vytvorené neskôr) a odstraňujú sa pokiaľ súčet veľkostí prác nie je menší ako *defaultRecomendedWorkSizeInBytes*, alebo kým nezostane len jedna práca.

3.9.2.3 SimpleAdaptiveScheduler

Stratégia, ktorú som navrhol pre túto triedu berie do úvahy čas, ako dlho už relácia beží. Podľa [87] a [42] pravdepodobnosť odchodu návštevníka zo stránky má takzvané Weibull rozdelenie, ktoré má dva parametre označované ako miera a tvar (anglicky scale a shape). Pri zafixovaní parametra tvaru na 1 odpovedá rozdelenie exponenciálnemu rozdeleniu. Autori uvádzajú, že pravdepodobnosť odchodu návštevníka webovej stránky vo väčšine prípadov klesá s časom stráveným na stránke. Čo sa dá interpretovať tak, že najskôr návštevník rýchlo zhodnotí, či ho stránka zaujíma a chce na nej zostať a potom, ak sa rozhodne zostať, tak s ňou nejaký čas interaguje. Poznatok, že pravdepodobnosť ukončenia relácie s postupom času klesá, využíva stratégia pre *SimpleAdaptiveScheduler*.

Táto stratégia prideľovania prác je založená na stratégií, ktorú používa *SimpleSizeBasedScheduler*. Veľkosť súčtu prác pre klienta však nie je pevne daná, ale počíta sa na základe času, ktorý už relácia beží podľa vzťahu:

$$\begin{aligned} computed &= func(duration) \cdot a \ + b \\ worksSize &= min(computed, worksSizeThreshold) \end{aligned} \tag{3.1}$$

kde $worksSize$ je veľkosť súčtu prác pre reláciu, $worksSizeThreshold$ je konfigurovateľná hranica pre maximálnu veľkosť súčtu prác, a a b sú konfigurovateľné parametre a $func$ je funkcia času, ktorá je spolu s jej parametrami v konfigurácií frameworku nastaviteľná na lineárnu funkciu, exponenciálnu funkciu, alebo funkciu, ktorá odpovedá distribučnej funkcii exponenciálneho rozdelenia.

3.9.3 Prijatie a spracovanie výsledkov prác

Pri prijímaní výsledku od klienta v podobe objektu *WorkUnitResult* framework najskôr skontroluje, či požiadavka obsahuje validný identifikátor relácie. Ak neobsahuje, požiadavka sa ignoruje. Aké identifikátory framework považuje za validné popisujem v sekcii 3.11. Následne skontroluje, či objekt obsahuje validný identifikátor práce, výsledok výpočtu a či neobsahuje informáciu o chybe, ktorá mohla nastať u klienta. Ak je všetko v poriadku, výsledok sa spracuje. Ak nie, objekt sa spracuje ako chybný a to nasledovne. Najskôr sa do logu zapíše informácia o chybe. Od relácie, v ktorej nastala chyba už nepríde výsledok výpočtu a preto je treba, aby sa práca pridelila ďalšej relácii. Skontroluje sa preto, či relácia má pridelenú uvedenú prácu (či nejde o podvrh, alebo či framework už raz neprijal *WorkUnitResult* od danej relácie) a ak

áno, inkrementuje sa počet relácií, ktorým ešte treba danú prácu distribuovať (atribút *remainingReplicasCount*).

Spracovanie výsledku bez chyby prebieha nasledovne. Najskôr sa skontroluje či existuje práca, ktorej výsledok sa spracováva. Môže nastať situácia, keď príde výsledok práce úlohy, ktorá už bola kompletne vyriešená. To môže nastať, ak je jedna relácia výrazne pomalá a kým vráti výsledok, stihne sa práca prideliť ďalšej relácii, ktorá prácu vypočíta, vráti výsledok a úloha sa kompletne vyrieši. Potom sa skontroluje, či práca bola danej relácii pridelená. To sa skontroluje podľa záznamov o pridelení práce (atribút *tracks*). Ak práca pridelená nebola, výsledok je pravdepodobne podvrhnutý a preto sa ďalej nespracováva. Nakoniec sa skontroluje, či od danej relácie už raz výsledok neprišiel. To sa skontroluje podľa prijatých a uložených výsledkov (atribút *directResults*). Táto situácia by bežne nemala nastať, ale môže sa stať, že u klienta nastane chyba pri odoslaní výsledku, odošle ho znova a nakoniec prídu oba výsledky. Alebo ak by chcel niekto výsledok podvrhnúť a opakovane ho zasiela. Ak všetky testy dopadnú úspešne, k výsledku výpočtu sa pridajú metadáta a v podobe objektu *DirectResult* sa výsledok uloží. Metadáta, ktoré sa k výsledku od klienta pridávajú, sú: trvanie výpočtu z pohľadu servera, časová značka prijatia výsledku, identifikátor relácie, klienta a práce a haš výsledku.

Po úspešnom spracovaní výsledku od klienta framework skontroluje, či práca obsahuje výsledky od dostatočného počtu relácií. To znamená, či počet prijatých výsledkov práce sa rovná, alebo je väčší ako atribút s názvom *currentReplicationFactor*. Ak áno, vykoná sa majoritné hlasovanie. Vzhľadom na to, že výsledkom práce môže byť štrukturovaný objekt a porovnanie by mohlo byť časovo náročné, porovnávajú sa haše výsledkov a nie priamo výsledky. Majoritné hlasovanie môže, ale nemusí určiť výsledok. Ak výsledok nejde určiť (napríklad ak je každý prijatý výsledok iný), je treba, aby sa práca pridelila ďalším reláciám. Minimálny počet ďalších relácií, ktorý je nutný k tomu, aby majoritné hlasovanie mohlo určiť výsledok je daný nasledujúcim vzťahom:

$$m + x \geq \lfloor \frac{c + x}{2} \rfloor + 1$$

kde m je počet najpočetnejšieho výsledku, c je *currentReplicationFactor* a x je to čo hľadáme, čiže minimálny počet relácií, ktorým ešte treba prideliť danú prácu, aby majoritné hlasovanie bolo schopné určiť výsledok. Nová hodnota atribútu *currentReplicationFactor* je potom daná vzťahom:

$$currentReplicationFactor = currentReplicationFactor + x$$

Ak sa výsledok majoritným hlasovaním neurčí, *currentReplicationFactor* sa zvýši podľa uvedeného vzťahu a atribút *remainingReplicasCount* sa zvýši o x, pretože je treba prácu prideliť ešte minimálne x reláciám. Situácia, kedy majoritné hlasovanie neurčí výsledok môže nastať pri chybe v kóde prototypu úlohy, alebo pri podvrhnutí výsledkov.

Ak majoritné hlasovanie určí výsledok, haš tohto výsledku sa priradí práci (do atribútu *resultHash*) a práca sa považuje za dokončenú. Už sa viac nebude distribuovať medzi klientov. Následne sa skontroluje, či všetky práce v listoch stromu prác danej *TaskUnderScheduling* sú dokončené. Ak áno, framework zlúči výsledky prác do celkového výsledku úlohy. Zlučovanie prebieha obdobne ako delenie s využitím zlučovacej funkcie z prototypu. Ak strom prác má len jeden uzol, nezlučuje sa, lebo sa nemá zlúčiť s čím. Po zlúčení sa vytvorený finálny výsledok a jeho haš uloží do danej úlohy, stav úlohy sa nastaví na „hotová“, objekt *TaskUnderScheduling* sa vymaže a pošle sa požiadavka na modul *TasksPreparationManager*, aby pripravil na distribúciu ďalšie úlohy.

3.9.4 Detekcia a reakcia na výpadok výpočtového uzla

Ako som už spomínal v sekcii o vlastnostiach webu 3.1, návštevníci webových stránok často opustia stránku po veľmi krátkej dobe. Pre framework to znamená, že relácie budú často veľmi krátke a ak sa im pridelí práca, nemusí sa stihnúť vykonať a výsledok odoslať na server.

Pre detekciu a riešenie týchto situácií som navrhol nasledujúce riešenie. *VolunteerServer* udržiava kolekciu bežiacich relácií. Keď na server príde nová požiadavka o prácu, ktorá neobsahuje identifikátor relácie, ide o novú reláciu. Server vygeneruje identifikátor relácie a do kolekcie bežiacich relácií pridá nový objekt *ClientSession* reprezentujúci túto reláciu. Do tohto objektu server uloží čas kedy začala, čas posledného prístupu, identifikátor klienta v ktorom beží a ďalšie metadáta. Identifikátor relácie pridá server do odpovede. Relácia tento identifikátor potom posiela pri každej požiadavke na server. Pri každej požiadavke server aktualizuje čas posledného prístupu relácie. V pravidelných intervaloch modul s názvom *DeadSessionCollector* kontroluje časy posledných prístupov relácii. Ak dĺžka doby od posledného prístupu relácie po moment kontroly je väčšia ako stanovená hranica, relácia sa prehlási za ukončenú/mŕtvu a vymaže sa z kolekcie bežiacich relácií. Ak boli relácii pridelené práce, ktorých výsledky nestihla odoslať, inkrementuje sa im atribút *remainingReplicasCount*, aby sa práce pridelili ďalšej relácii. Kým relácia prevádza výpočet práce, v pravidelných intervaloch posiela prázdnu požiadavku, takzvaný *heartBeat* na server, aby server bol informovaný o tom, že relácia stále beží. Podobne na zánik relácie, ktorá mala pridelenú prácu, reagoval aj systém popisovaný v [1].

Interval kontrol, interval posielania *heartBeat* aj doba, po ktorej sa relácia pokladá za ukončenú je konfigurovateľný.

Môže nastať situácia, keď požiadavka obsahuje validný identifikátor, ale v kolekcií sa nenachádza objekt s týmto identifikátorom. V tom prípade server vytvorí nový objekt relácie s daným identifikátorom a vloží ho do kolekcie. Takáto situácia môže nastať napríklad ak sa za mŕtvu prehlási relácia, ktorá ešte beží a príde od nej požiadavka.

3.9.5 Regulovanie využitia výkonu počítača klienta serverom

Regulovanie využitia výkonu počítača klienta frameworkom je dôležité minimálne z dvoch dôvodov. Po prvé, v jednom klientovi môže bežať viac relácií. Ak by sa naraz vykonávali výpočty vo viacerých reláciách, mohlo by dochádzať k predlžovaniu času čakania relácie na procesor, čím by sa predlžoval čas výpočtu práce. Ďalším dôvodom je to, že framework by nemal obmedziť ostatnú činnosť klientského počítača, prehliadača, alebo interakciu so stránkou. V ideálnom prípade by návštevník webovej stránky nemal na prvý pohľad vedieť rozlíšiť, či sa u neho vykonáva výpočet alebo nie.

3.9.6 Práca len v jednej relácii

Aby sa zabezpečilo vykonávanie výpočtov u klienta vždy len v jednej relácii, navrhol som nasledujúce riešenie. Server si udržiava v objekte reprezentujúcom klienta v zmysle webového prehliadača (trieda *Client*) identifikátor takzvanej aktívnej relácie (atribút *activeSessionId*). Aktívna relácia je relácia klienta, v ktorej práve prebieha výpočet práce. Relácia sa stáva aktívnou, keď sa jej pridelia práce, a prestáva byť aktívnou, keď vráti výsledok poslednej práce, alebo keď ju *DeadSessionCollector* prehlási za mŕtvu. Kým má klient nejakú reláciu aktívnu, žiadnej inej relácii sa nepridelí práca. Tým je zabezpečené, že sa vždy pracuje len v jednej relácii.

Môže sa stať, že na jednom počítači beží viac klientov. Nastane to napríklad vtedy, keď sa používajú rôzne prehliadače, alebo sa používa inkognito okno a podobne. Tieto situácie framework nedokáže rozpoznať a môže sa teda za istých okolností stať, že na jednom počítači bude bežať viac aktívnych relácií, pričom každá bude v inom klientovi.

3.9.7 Regulovanie záťaže procesora klientského počítača

Nepodarilo sa mi nájsť spôsob ako priamo regulovať využívanie procesora u klienta pomocou klientského kódu. Predpokladám teda, že taká možnosť nie je (aj vzhľadom na to, že kód je vykonávaný v bezpečnom prostredí [anglicky sandbox] prehliadača) a ak by aj bola, tak pravdepodobne nie je štandardizovaná. Napriek tomu, že framework nedokáže regulovať momentálne zaťaženie procesora, vie ho regulovať z dlhodobého pohľadu a to tak, že istú dobu zaťažuje procesor naplno, istú dobu čaká a potom zase zaťažuje procesor. Z dlhodobého hľadiska je potom procesor priemerne vyťažený menej ako naplno. Napríklad, ak by sa procesor zaťažil naplno na x sekúnd a potom by sa x sekúnd nezaťažoval vôbec, dalo by sa povedať, že priemerne počas $2x$ sekúnd bol procesor zaťažený na 50%. Rôzne relácie v rámci jedného klienta nemajú informácie jedna o druhej, navyše môžu hocikedy vzniknúť a zaniknúť a informácie by sa mohli stratiť. Toto čakanie medzi zaťažovaním procesora preto rieši server. U každého klienta si ukladá, kedy naposledy bola niektorej z jeho

relácií pridelená práca (atribút *scheduledAt*). Keď aktívna relácia vráti výsledky všetkých pridelených prác, server vypočíta ako dlho relácia pracovala a zaťažovala procesor a vypočíta kedy najbližšie sa môže klientovi, respektíve niektorej z jeho relácií opäť prideliť práca. Tento údaj sa uloží do atribútu *scheduleAfter*. Hodnota atribútu *scheduleAfter* sa vypočíta nasledovne:

$$\begin{aligned} duration &= now - scheduledAt \\ scheduleAfter &= now + duration \cdot throttleFactor \end{aligned} \tag{3.2}$$

kde *throttleFactor* je konfigurovateľný parameter a *now* je aktuálny čas. Podobný princíp regulovania zaťaženia procesora spomínajú autori práce [34].

V sekcií 4.2.7 popisujem ako framework môže využiť viac jadier procesora na vykonávanie výpočtov.

3.9.8 Detekcia a reakcia na málo výkonný uzol

Výpočet úlohy nemôže skončiť skôr, ako sa dopočíta posledná práca. Zároveň framework nemôže určiť výsledok práce, kým neprijme dostatočný počet výsledkov práce. Môže sa stať, že výpočet práce v jednej relácii trvá výrazne dlhšie ako v ostatných, čo môže spôsobiť značné predĺženie času výpočtu úlohy. Takéto situácie je treba detekovať a riešiť. O to sa stará modul, ktorý som nazval *LongRunningSessionCollector*. Ten v pravidelných intervaloch kontroluje, či opísaná situácia nenastala. Modul najskôr získa pole prác, ktoré ešte nie sú dokončené, ale už sú pre každú prijaté aspoň dva výsledky. Pre každú získanú prácu potom vypočíta priemernú dĺžku výpočtu práce (z pohľadu servera) a skontroluje, či relácia, od ktorej ešte nebol prijatý výsledok, pracuje na práci priveľmi dlho oproti ostatným reláciám, ktoré už prácu vypočítali a vrátili výsledok. Ak pracuje príliš dlho, je výhodné prideliť prácu ďalšej relácii, lebo je šanca, že ďalšia relácia vráti výsledok práce skôr ako tá pôvodná. Framework preto inkrementuje atribút *remainingReplicasCount* danej práce. Toto opatrenie funguje rovnako aj v prípade, že relácia síce v skutočnosti rýchlo vypočíta prácu, ale má pomalé pripojenie a dlho trvá prenos dát medzi serverom a reláciou. Relácia sa prehlási za príliš dlho pracujúcu ak platí:

$$\begin{aligned} maxDuration &= factor \cdot avgDuration \\ duration &> min(maxDuration, maxRunningTime) \end{aligned} \tag{3.3}$$

kde *avgDuration* je priemerná doba trvania výpočtu práce (a prenosu dát) reláciami od ktorých už bol prijatý výsledok, $factor$ je konfigurovateľný parameter, $maxRunningTime$ je konfigurovateľná maximálna doba trvania výpočtu práce a *duration* je doteraz uplynulý čas odkedy bola práca pridelená relácii.

3.10 Funkcie klientskej časti

Výpočtový klient frameworku funguje podobne ako väčšina klientov v prácach popísaných v kapitole 2.2. V každej relácii sa spustí nová inštancia výpočtového klienta frameworku. V pravidelných intervaloch žiada od servera pridelenie práce. V požiadavke o prácu zasiela informáciu o tom, či prehliadač podporuje technológiu WebAssembly. Pridelené práce obsahujú okrem dát aj kód, ktorý sa má vykonať, alebo identifikátor kódu. Ak obsahujú len identifikátor, klient si pomocou neho vyžiada od servera kód, ktorý sa má vykonať. Ak vyžiadaný kód existuje v cache pamäti prehliadača alebo medziľahlého sieťového prvku, použije sa bez toho, aby požiadavka prišla až na server. Následne sa kód spustí, spracujú sa dáta a výsledok sa pošle na server v podobe objektu *WorkUnitResult*. Ak nastala chyba počas spracovávania práce, uloží sa o nej informácia do atribútu *error*. Kým sa spracovávajú pridelené práce, klient v pravidelných intervaloch zasiela na server prázdnu požiadavku, takzvaný *heartBeat*, aby informoval server, že relácia stále beží. Akonáhle výpočtový klient vypočíta výsledok práce, odošle ho na server, nečaká na dokončenie všetkých prác. Keď sa spracujú a odošlú výsledky všetkých prác, ktoré relácia získala začne opäť žiadať od servera prácu. Tento cyklus beží kým sa relácia neukončí. V každej požiadavke na server sa zasiela identifikátor relácie a klienta (webového prehliadača).

Server môže do odpovede pridať konfiguračné nastavenia a nastaviť tak interval v akom má klient žiadať o prácu a interval v akom má klient zasielať *heartHeat*.

3.11 Bezpečnosť

Pre framework som navrhol niekoľko základných bezpečnostných opatrení.

Prvým opatrením je, že *ProgrammerServer* spracováva len požiadavky, ktoré obsahujú API kľúč, ktorý sa nachádza v databáze API kľúčov, ktorú si server uchováva. Ostatné požiadavky ignoruje a HTTP kód odpovede nastaví na 403. HTTP kód 403 indikuje, že server požiadavke porozumel, ale odmieta ju autorizovať [95]. Zoznam API kľúčov a subjektov, ktorým sú pridelené sú napevno vypísané v konfigurácií frameworku.

Framework spúšťa na strane servera aj na strane klienta kód od užívateľov. Napriek tomu, že tvorcovia prototypov úloh môžu byť do istej miery považovaní za overených, spúšťaný kód treba stále brať ako nedôveryhodný. Je teda dôležité, aby spustený kód nemohol spôsobiť škody na serveri ani u klienta. Užívateľský kód sa preto aj na serveri aj u klienta spúšťa v bezpečnom prostredí, v takzvanom *sandbox* (slovensky pieskovisko).

Ďalším bezpečnostným opatrením je to, že komunikácia medzi serverom a klientom je šifrovaná. To znižuje riziko, že do komunikácie vstúpi tretia strana, ktorá by neoprávnene menila posielané dáta alebo kód.

Aby útočník nemohol podvrhovať identifikátor relácie, klienta či práce, sú ich hodnoty pred odoslaním klientovi zašifrované. Klient potom posiela zašifrované identifikátory, ktoré sa na serveri naspäť dešifrujú. Navyše tieto identifikátory majú špecifickú štruktúru, ktorá sa pri spracovaní požiadavky kontroluje. Ak má dešifrovaný identifikátor správnu štruktúru, považuje sa za validný.

Pre zníženie rizika, že výsledky sú falošné, framework využíva spomínané majoritné hlasovanie, ktoré je jednoduché, rýchle a autori [40] uvádzajú, že pri dostatočnom počte klientov je aj vysoko spoľahlivé. Užívateľ má možnosť definovať minimálny počet prijatých výsledkov, z ktorých sa má majoritným hlasovaním určiť výsledok, ktorý sa bude považovať za správny. Mal by si byť však vedomý toho, že pravdepodobnosť, že výsledok práce, určený majoritným hlasovaním, je naozaj správny nikdy nebude 100%.

Téma zabezpečenia dát je komplikovanejšia. Webový prehliadač zo svojej podstaty pred užívateľom neskrýva dáta. Naopak, umožňuje mu nahliadať do zdrojových kódov, zobrazovať preberané dáta a podobne. Z princípu sa teda dáta v prehliadači nedajú skryť pred užívateľom. Mohli by sa na serveri zašifrovať a u klienta dešifrovať. U klienta by sa ale aj tak v istom momente nachádzali nezašifrované dáta. Útočník, ktorý by implementoval vlastný webový prehliadač by nemal problém dostať sa k nim. Na druhej strane, ak sa úloha rozdelí na viac podúloh (prác) tak s veľkou pravdepodobnosťou sa k útočníkovi dostane len časť dát, ktoré samé o sebe nemusia dávať zmysel. V konečnom dôsledku existujú úlohy, ktorých dáta bez kontextu nedávajú žiadny zmysel. Ďalšia vec je, že z pohľadu užívateľa frameworku nemusí byť ani správca frameworku dôveryhodný na to, aby mu zaslal svoje dáta na výpočet. Pre užívateľa by bolo teda najbezpečnejšie zašifrovať dáta lokálne a nechať spracovať zašifrované dáta, ktoré by sa znovu dešifrovali až zase lokálne u užívateľa. Existujú matematické modely, ktoré prácu nad zašifrovanými dátami umožňujú. O týchto modeloch sa píše napríklad v [96]. Na to, aby sa pri výpočte počítalo nad zašifrovanými dátami, nemusia byť vo frameworku urobené žiadne zmeny. Stačí ak tvorca prototypu úlohy implementuje algoritmus, ktorý to dokáže a dá inštrukcie užívateľom ako treba dáta pripraviť pred odoslaním a ako potom prijatý výsledok dešifrovať. Použitie takejto metódy ochrany dát so sebou však prináša aj nevýhody ako je napríklad nižšia rýchlosť výpočtu a podobne. Užívateľ frameworku by si mal byť vedomý rizík úniku dát a počítať nad dátami, ktorých únik (alebo únik časti dát) mu nespôsobí škodu, alebo využiť bezpečnejší prístup, ako je výpočet nad zašifrovanými dátami spojený zase s inými nevýhodami.

Popísané opatrenia zvyšujú bezpečnosť frameworku, no stále je zraniteľný. Napríklad je možné automaticky vytvárať nových klientov a relácie a zahltiť tak databázu frameworku. Zahltiť je možné framework aj veľkým množstvom požiadaviek o prácu.

Kapitola 4

Implementácia

4.1 Zvolené technológie

4.1.1 Jazyk implementácie

4.1.1.1 JavaScript

JavaScript je jedným z hlavných jazykov webu a podľa w3schools.com [97] patrí medzi 3 jazyky, ktoré musí poznať každý webový vývojár. Jeho oficiálny názov je ECMAScript. Je to odľahčený (anglicky lightweight) interpretovaný alebo JIT-kompilovaný programovací jazyk s first-class funkciami [12]. *JIT* je skratka pre *just-in-time*, čo by sa dalo preložiť ako *práve-včas*. JIT-kompilovanie odkazuje na správanie moderných JavaScriptových strojov, ktoré sledujú vykonávanie kódu a pre zrýchlenie sa môžu rozhodnúť niektoré funkcie skompilovať. Potom, ak je to možné vzhľadom na typy vstupných parametrov a podobne, používajú tieto skompilované varianty funkcií [98]. To, že sa povie, že programovací jazyk má *first-class* funkcie znamená, že s funkciami sa zaobchádza ako s premennými. Napríklad, môžu sa odovzdávať ako argument funkcie, môžu byť návratovou hodnotou funkcie, môžu byť priradené do inej premennej a podobne.

Ako sa ďalej píše v [12] JavaScript je známy najmä ako skriptovací jazyk pre webové stránky, ale využíva ho aj veľa iných prostredí ako napríklad spomínaný Node.js, alebo Adobe Acrobat [99], či Apache CouchDB [100].

4.1.1.2 Typescript

Hoci skriptovacím jazykom pre web stránky je JavaScript, webový vývojár nemusí programovať priamo v tomto jazyku. Ako som spomínal v sekcii 3.1, existuje dnes viacero jazykov pre podporu vývoja webových aplikácií. Vývojár si môže teda vybrať jazyk, ktorý mu najviac vyhovuje, programovať v ňom a následne kód preložiť do JavaScriptu.

Jeden z najznámejší jazykov, ktoré vznikli pre účely vývoja webových aplikácií je TypeScript. Podľa oficiálneho webu [82] je TypeScript nadmnožina JavaScriptu, ktorá sa kompiluje na čistý JavaScript. Najdôležitejšou vlastnosťou TypeScriptu, pre ktorú som sa rozhodol implementovať framework pomocou neho je podpora dátových typov. TypeScript umožňuje do kódu zaviesť statické typy pomocou anotácií. Vývojár je potom nútený dodržiavať typy pri priraďovaní do premennej, odovzdávaní argumentov do funkcie a podobne tak, ako pri staticky typovaných jazykoch. Vývojové prostredia ako WebStorm [101], Visual Studio Code [102] a ďalšie integrovali podporu TypeScriptu. Vývojára potom upozorňujú na nedodržanie dátových typov a zároveň vďaka informáciám o typoch môžu poskytovať lepšiu nápovedu ako bez tejto informácie. Týmto sa môže predchádzať chybám v kóde a zároveň zrýchliť jeho vývoj. Zároveň používateľ TypeScriptu môže využívať jeho vlastnosti a konštrukty, ktoré nie sú súčasťou JavaScriptu, alebo sú to vlastnosti, ktoré sa práve zavádzajú (anglicky bleeding-edge features) do JavaScriptu. Napríklad triedy boli súčasťou TypeScriptu skôr ako v JavaScripte, ktorý ich zaviedol vo verzií ECMAScript 2015 [103] [104]. TypeScript sa ale nakoniec vždy preloží do čistého čitateľného JavaScriptu a to vo verzií akú zvolí vývojár, minimálne však ECMAScript 3 [82]. TypeScript je síce produkt firmy Microsoft, no napriek tomu je otvorený a s veľkou podporou komunity.

4.1.2 Node.js

Oficiálny web [105] uvádza, že Node.js je behové prostredie (anglicky runtime) pre JavaScript postavené na JavaScriptovom stroji V8 pre Chrome [106], ktoré je navrhnuté pre vývoj škálovateľných sieťových aplikácií. Vykonávanie JavaScriptu v Node.js je jednovláknové a využíva udalosťami riadený (anglicky event-driven) asynchrónny neblokujúci model. Vďaka tomu je dobre škálovateľný, efektívny a odľahčený (anglicky lightweight). Web w3schools [107] uvádza príklad spracovania požiadavky Node.js serverom na príklade získania súboru. Pre získanie súboru Node.js server

1. odošle úlohu pre súborový systém
2. vzápätí je server pripravený obsluhovať ďalšiu požiadavku
3. keď súborový systém otvorí a prečíta súbor, server vráti obsah súboru klientovi.

Takýmto spôsobom Node.js eliminuje blokovanie obsluhy požiadaviek čakaním na operácie, ktoré vykonáva operačný systém alebo iný proces.

Napriek tomu, že Node.js je jednovláknový, umožňuje využívať výhody viacjadrových procesorov. Napríklad tak, že poskytuje API pre vytvorenie procesu potomka (anglicky child process), s ktorým vie potom rodič jednoducho komunikovať [105].

Node.js sa teší veľkej popularite a komunite. Podľa údajov na [108] má 9 miliónov inštancií, celosvetovo sa usporiadalo viac ako 1500 stretnutí Node.js vývojárov a jeho register balíčkov npm je podľa [109] najväčší svetový register softvéru s približne tromi miliardami balíčkov. Npm zároveň označuje web registra a nástroj pre príkazový riadok. Npm slúži na inštaláciu, zdieľanie a distribúciu kódu, spravuje závislosti projektu, slúži aj na zdieľanie a prijímanie spätnej väzby od ostatných vývojárov a ďalšie [110].

Popísané vlastnosti indikujú že, Node.js je vhodný nástroj pre framework, o ktorom je táto práca. Taktiež ho využívali autori prác [29], [4], či [37], ktoré som popisoval v kapitole 2.2. Zároveň vzhľadom na to, že Node.js používa JavaScript, môže byť serverová aj klientská časť implementovaná v jednom jazyku. Pre tieto dôvody som sa rozhodol použiť tento nástroj.

4.1.3 Asm.js

Špecifikácia na oficiálnom webe [44] definuje asm.js ako striktnú podmnožinu JavaScriptu, ktorá môže byť použitá ako nízko-úrovňový cieľový jazyk pre kompilátory. Jazyk asm.js poskytuje podľa špecifikácie abstrakciu virtuálneho stroja veľkou binárnou haldou s efektívnym zápisom a čítaním, aritmetikou celočíselného typu a typu pohyblivej rádovej čiarky, funkciami a ukazovateľmi na funkcie.

Asm.js je staticky typovaný jazyk, ktorý môže byť skontrolovaný počas parsovania. Explicitná direktíva

```
1 "use asm";
```

umožní JavaScriptovému stroju, aby sa vyhol zbytočnej a potencionálne náročnej validácii na ostatnom JavaScriptovom kóde [44].

Asm.js umožňuje kompiláciu vopred (anglicky ahead-of-time). Kód vygenerovaný vopred môže byť veľmi efektívny napríklad vďaka absencií kontroly typov za behu, efektívnemu zápisu a čítaniu z haldy a podobne.

Predbežné testy programov napísaných v jazyku C porovnávali beh programov po kompilácií pomocou nástroja clang [111] a beh po skompilovaní do asm.js. Faktor spomalenia asm.js bol zvyčajne menší ako 2. K rovnakým výsledkom sa dostali autori vyššie popísanej práce [40].

4.1.4 WebAssembly

V roku 2015 sa začala oficiálna história technológie WebAssembly. Oficiálna stránka [60] uvádza, že WebAssembly, skrátene wasm, je nový prenosný, veľkosťou aj časom pre načítanie efektívny formát, vhodný na kompiláciu pre web. Autori stránky dodávajú, že WebAssembly popisuje pamäťovo bezpečné prostredie bežiace v sandboxe, ktoré môže byť implementované v existujúcom virtuálnom stroji pre JavaScript.

Mozilla na svojom webe pre vývojárov [112] popisuje WebAssembly ako nový druh kódu, ktorý môže bežať v moderných prehliadačoch. Je to nízko-

úrovňový jazyk s kompaktným binárnym formátom, ktorý beží s výkonom takmer ako natívne aplikácie a slúži ako cieľ kompilácie (anglicky compilation target) pre jazyky ako C/C++, ktoré vďaka tomu môžu bežať na webe. Je navrhnutý tak, aby bežal popri JavaScripte a aby mohli spolupracovať. Článok [112] ďalej tvrdí, že WebAssembly má veľký dopad na webovú platformu, pretože poskytuje možnosť ako spúšťať kód napísaný v rôznych jazykoch na webe s takmer natívnym výkonom. Vďaka spolupráci s JavaScriptom, ktorý má dopĺňať a nie nahradiť, môže programátor využiť vysoký výkon WebAssembly a vyjadrovaciu silu a flexibilitu JavaScriptu v jednej aplikácií.

WebAssembly už teraz podporujú Firefox, Edge, Chrome, Chrome pre Android a Safari. Podľa [113] 72,75% užívateľov používa prehliadač, ktorý podporuje WebAssembly. Na štandarde pracuje W3C Community Group [114] a W3C Working group [115] s aktívnou spoluprácou s majoritnými prehliadačmi [112].

Osobne si myslím, že WebAssembly je revolučná technológia, ktorá zmení pohľad na webové aplikácie vďaka tomu, že umožňuje, aby sa jazyky, ktoré pôvodne neboli určené na vývoj webových aplikácií skompilovali na kód, ktorý je určený pre web a bežali pri takmer natívnom výkone. Zároveň okrem nových projektov sa môžu presunúť na web aj vyspelé, dlho vyvíjané aplikácie či knižnice, čo spomína aj Alex Danilo na konferencí Google I/O '17 [116]. Napríklad knižnica pre spracovanie obrazu OpenCV [117] ponúka pripravený skript pre skompilovanie časti knižnice do asm.js alebo wasm.

WebAssembly sa intenzívne vyvíja a v budúcnosti sa plánujú pridať ďalšie vlastnosti, napríklad podpora vlákien a má sa zjednodušiť jeho používanie.

4.1.5 Emscripten

V predchádzajúcich sekciách som popisoval technológie WebAssembly a asm.js, ktoré sú určené pre web a majú vysoký výkon. Obe tieto technológie sú cieľovým jazykom pre kompiláciu a nie sú určené pre vývoj. Nepredpokladá sa, že by programátor písal kód v asm.js, alebo WebAssembly, hoci možné to je.

Pre využitie týchto technológií je teda potrebný kompilátor. Väčšina oficiálnych zdrojov ([118], [119], [60]) o asm.js a WebAssembly odkazuje na nástroj Emscripten [43].

Emscripten je sada nástrojov pre kompiláciu na asm.js a WebAssembly. Pomocou Emscripten sa dá skompilovať C, C++ ale aj ďalšie jazyky. Nie každý kód sa však dá skompilovať, alebo po kompilácií môže bežať pomaly. Je to napríklad kód, ktorý využíva nízko-úrovňové volania natívneho prostredia, alebo kód využívajúci C++ výnimky a podobne. Viac o týchto obmedzeniach sa píše v [43]. Taktiež niekedy je treba zasiahnúť do originálneho kódu a spraviť v ňom jemné úpravy pred kompiláciou pomocou Emscripten.

Emscripten emuluje funkcionalitu, ktorá sa bežne využíva pri vývoji mimo webovej platformy ako je napríklad súborový systém. Okrem asm.js respektíve wasm kódu vygeneruje Emscripten aj JavaScriptový súbor, v ktorom je

zahrnutá spomínaná emulácia niektorých funkcionalít, ktoré nie sú poskytnuté webovým prostredím. Do tohto JavaScriptového kódu sa vygenerujú aj funkcie na uľahčenie interakcie medzi JavaScriptom a asm.js, respektíve wasm modulom. Pri malých a jednoduchých C/C++ kódoch niekedy Emscripten vygeneruje zbytočne veľký sprievodný JavaScriptový kód. Tohto problému sú si vývojári vedomí a snažia sa ho odstrániť. Hlavný dôvod, pre ktorý tento problém vznikol je ten, že Emscripten vzniklo so zameraním na čo najväčšiu podporu už existujúceho kódu, čo viedlo ku generovaniu veľkého sprievodného JavaScriptového kódu [120].

Nástroj Emscripten umožňuje rôzne nastavenia vrátane úrovne optimalizácie. Z osobnej skúsenosti mi jeho použitie príde trochu komplikované. Predpokladám, že čoskoro pribudnú ďalšie podobné nástroje, ktoré budú jednoduché a na rozdiel od Emscripten budú zamerané na nové projekty od začiatku vyvíjané pre web. Emscripten je zatiaľ ale najrozvinutejší nástroj pre kompilovanie do wasm a asm.js a všetky oficiálne návody odkazujú naň. Preto som sa rozhodol ho využívať.

4.1.6 Web Worker

Web Worker je technológia, ktorá umožňuje vykonávanie JavaScriptu vo vlákne na pozadí nezávisle na ostatných skriptoch bez ovplyvňovania interakcie so stránkou. Výhoda technológie Web Worker spočíva v tom, že náročný výpočet môže bežať v samostatnom vlákne, čo umožňuje hlavnému vláknu bežať bez blokovania či spomalenia [121], [28].

Web Worker je objekt, v ktorom beží pomenovaný JavaScriptový súbor. Objekty *Web Worker* pracujú v kontexte, ktorý je iný ako aktuálne okno a preto nie sú k dispozícií metódy a premenné a atribúty objektu okna. Komunikácia a prenos dát medzi *Web Worker* objektami a hlavným vláknom prebieha posielaním správ, pričom sa dáta nezdieľajú, ale kopírujú [28].

Autori článku [40] zisťovali, či použitie *Web Worker* objektov na náročný výpočet ovplyvní užívateľa. Testovalo sa, či výpočtovo náročné úlohy z oblasti spracovania obrazu prebiehajúce vo *Web Worker* objektoch ovplyvnia automatické doplňovanie pri vyhľadávaní. Ukázalo sa, že z užívateľského pohľadu je takmer nerozlíšiteľné, či na pozadí beží náročný výpočet v objekte *Web Worker* alebo nie.

4.1.7 MongoDB

MongoDB je dokumentová databáza s dobrou škálovatelnosťou a flexibilitou [122]. Pracuje s akoby JSON (anglicky JSON-like) objektami, nazývanými dokumenty, ktoré ukladá vo formáte BSON. BSON je binárna reprezentácia pre JSON [123]. Špecifikácia BSON obsahuje aj ďalšie dátové typy, ktoré nie sú súčasťou JSON [124]. MongoDB nepoužíva schému a štruktúra dokumentov

môže byť rôzna. Databáza v MongoDB pozostáva z kolekcií, ktoré obsahujú dokumenty.

JavaScript Object Notation, skrátene JSON, je založený na podmnožine jazyka JavaScript [125] a vďaka tomu je práca s MongoDB respektíve jeho dokumentami prirodzená a jednoduchá. Preto som sa rozhodol použiť MongoDB pri tvorbe frameworku.

4.1.8 HTTP/2

HTTP/2 je nová verzia protokolu HTTP [64]. Autori článku [126] uvádzajú, že pomocou tohto protokolu môžu byť aplikácie rýchlejšie, jednoduchšie a robustnejšie. Ďalej píšu, že medzi jeho hlavné ciele patria zníženie latencie a efektívna kompresia hlavičiek. Dodávajú tiež, že všetky základné koncepty ako sú HTTP metódy, kódy, či hlavičky sa zachovávajú tak ako sú a že nová verzia protokolu nenahrádza, ale rozširuje predchádzajúce štandardy.

Oficiálna stránka [127] tiež uvádza, že zameranie nového protokolu je na výkonnosť. Oproti predchádzajúcim verziám protokolu má HTTP/2 binárny formát. To má za následok, že je kompaktnejší a jeho parsovanie je efektívnejšie. V [127] sa tiež uvádza, že je menej náchylný na chyby.

Podľa [113] používa viac ako 84% užívateľov internetu webový prehliadač, ktorý HTTP/2 podporuje.

Vzhľadom na to, že podľa uvedených zdrojov je nový protokol efektívnejší ako prechádzajúce verzie a zároveň ho dokáže využívať relatívne veľa užívateľov internetu, rozhodol som sa využiť tento protokol. Pre používanie protokolu HTTP/2 využívam modul *spdy* [128], ktorý zabezpečí, že ak webový prehliadač nepodporuje tento protokol, použije server protokol predchádzajúcej verzie.

4.2 Implementačné detaily vybraných častí a funkcií frameworku

4.2.1 Procesy modulu VolunteerServer

Napriek tomu, že Node.js je jednovláknový proces, viacjadrový procesor počítača serveru sa dá využiť tak, že hlavný proces vytvorí viac podprocesov, s ktorými potom môže komunikovať pomocou správ. Hlavný proces modulu *VolunteerServer* zabezpečuje najmä obsluhu požiadaviek a distribúciu úloh. Server pomocou Node.js funkcie *fork* vytvorí ďalšie procesy. Samostatný proces vytvorí server pre modul *DeadSessionCollector* a pre modul *LongRunningSessionCollector*. V týchto podprocesoch prebiehajú kontroly, či niektorú reláciu prehlásiť za mŕtvu, alebo dlhotrvajúcu pre niektorú prácu. Ak reláciu prehlásia za mŕtvu, alebo dlhotrvajúcu, pošlú jej identifikátor (v prípade dlhotrvajúcej relácie aj identifikátor práce) hlavnému procesu, ktorý reláciu spracuje.

Server ďalej vytvorí niekoľko ďalších procesov na spracovanie úloh. Koľko procesov na spracovanie úloh sa vytvorí je možné nastaviť v konfigurácií. Tieto procesy majú na starosti prípravu úloh a zlučovanie výsledkov prác. Hlavný proces posiela podprocesu správy, ktoré obsahujú informáciu o tom, ktorú činnosť má podproces vykonať – pripraviť práce, alebo zlúčiť výsledky. Ak hlavný proces žiada zlúčenie výsledkov, zašle v správe aj identifikátor úlohy, ktorej výsledky sa majú zlúčiť. Ak je to možné, podproces zlúči výsledky a hlavnému procesu vráti objekt *TaskUnderScheduling* so zlúčenými výsledkami. Ak počas spracovania nastala chyba, alebo výsledky nie je možné zlúčiť (napríklad, keď chýba niektorý z výsledkov prác), vráti podproces hlavnému procesu chybu, ktorá nastala. Činnosti sú podprocesom prideľované cyklicky. Funkcia, ktorá určí proces, ktorému hlavný proces pošle správu s požiadavkou o vykonanie činnosti môže byť pseudokódom zapísaná nasledovne:

```
1  function getSubProcessId(){
2    id = counter;
3    counter = ( counter + 1 ) % n;
4    return id;
5  }
```

kde n je počet procesov na spracovanie úloh a *counter* je globálny čítač.

Aby sa komunikácia pomocou správ medzi hlavným procesom a procesmi pre spracovanie úloh javila pre moduly hlavného procesu ako volanie asynchrónnych funkcií, implementoval som nasledujúci mechanizmus. Modul *TaskUnderSchedulingManager*, ktorý má na starosti komunikáciu s týmito podprocesmi, poskytuje ostatným modulom funkcie pre prípravu úloh na distribúciu a pre zlúčenie výsledkov úlohy. Vnútri týchto funkcií zasiela správy podprocesom. Každej správe pridá unikátny identifikátor. Modul si udržiava mapu volaní, ktorej kľúče sú identifikátory správ a hodnotami sú dvojice funkcií. Prvá funkcia s názvom *resolve* (slovensky vyrieš) je pre spracovanie bezchybnej odpovede, druhá funkcia s názvom *reject* (slovensky zamietni) je na spracovanie chyby. Správa prijatá od podprocesu tiež obsahuje identifikátor správy. Podľa neho sa z mapy volaní vyberie funkcia, ktorá sa má zavolať na zvyšok správy. Zavolaním týchto funkcií sa spôsobí, že funkcia poskytovaná modulom *TaskUnderSchedulingManager* vráti výsledok modulu, ktorý ju zavolal. Tento mechanizmus pracuje pomocou objektu *Promise* [129] (slovensky prísľub), ktorý umožňuje, aby sa asynchrónny kód javil ako synchrónny.

Pseudokód funkcií modulu *TaskUnderSchedulingManager*:

```
1 //funkcia, ktorá sa zavolá, keď hlavný proces prijme správu od podprocesu
2 function onMessageFromSubProcess (message){
3   let promiseFunctions = promisesMap.get(message.id);
4   if( message.error ) {
5     promiseFunctions.reject(message.error);
6   }else{
7     promiseFunctions.resolve(message);
8   }
9 }
```

```
10
11 //funkcia pre prípravu úloh na distribuovanie
12 function prepareWorksForSchedulingIfNeeded() {
13   return  new Promise( ( resolve, reject ) => {
14     let id = getUniqueMessageId();
15     let message = {
16       id:id,
17       operation:"prepare"
18     };
19     promisesMap.set(id,{resolve:resolve,reject:reject});
20     sendMessage(message);
21   })
22 }
23
24 //funkcia pre zlúčenie výsledkov úlohy
25 function mergeTask( taskId ) {
26   return new Promise( ( resolve, reject ) => {
27     let id = getUniqueMessageId();
28     let message = {
29       id:id,
30       operation:"merge",
31       taskId: taskId
32     };
33     let resolveFunction = (message) => {
34       taskUnderScheduling = message.taskUnderScheduling
35       resolve(taskUnderScheduling);
36     };
37     promisesMap.set(id,{resolve:resolveFunction,reject:reject});
38   })
39 }
```

Použitie funkcií modulu *TaskUnderSchedulingManager* potom môže vyzerať napríklad takto:

```
1 taskUnderScheduling = await taskUnderSchedulingManager
2   .mergeTask(work)
3     .catch(
4       (error) => {
5         console.log("error", error);
6         return null;
7       });
```

kde kľúčové slovo *await* spôsobí, že vykonávanie kódu sa preruší, kým asynchrónna funkcia nevráti výsledok, respektíve kým sa nezavolá funkcia *resolve*, alebo *reject* objektu *Promise*. Ak nastala chyba a zavolala sa funkcia *reject*, zavolá sa funkcia, ktorá je argumentom funkcie *catch* a jej návratová hodnota, v tomto prípade *null* sa priradí do premennej *taskUnderScheduling*.

4.2.2 Perzistencia dát

Databázový model odpovedá dátovému modelu, ktorý som popísal v sekcii 3.7. Databáza frameworku pozostáva z niekoľkých kolekcií. Najdôležitejšie z nich sú kolekcie pre objekty tried *TaskUnderScheduling*, *Task*, *TaskPrototype*, *Client*

a *ClientSession*. Každá z vymenovaných tried má vlastnú kolekciu. Práce, ktoré sa pre úlohu vytvorili, sú uložené v dokumente *TaskUnderScheduling*. Výhoda tohoto modelu je v tom, že objekty triedy *work* ani *workUnit* si nemusia znovu ukladať kódy výpočtu práce a netreba ani znovu pýtať od databázy tieto kódy výpočtu, pretože sa vrátia vo výsledku vyhľadávania spolu s prácou. Veľkosť dokumentu je minimálne taká veľká ako je veľkosť dát úlohy. MongoDB má však obmedzenie 16 MB na jeden dokument, čo nie je veľa. Preto som implementoval framework tak, že dáta prác a ich výsledky sa ukladajú na disk a do databázy sa ukladá len ich haš.

Alternatívne by databázový model mohol vyzerať tak, že by bola osobitná kolekcia pre práce, ktoré by buď mali uložené kódy výpočtu, alebo by bolo treba pre každú prácu pred pridelením klientovi získať kódy z databázy z prototypu úlohy.

4.2.3 Kompresia dát

Dáta, ktoré framework ukladá na disk pred uložením, skomprimuje pomocou Node.js modulu *zlib* [130]. Takéto opatrenie šetrí miesto na disku. Na druhú stranu kompresia a dekompresia dát trvá istý čas.

Objekt reprezentujúci prácu a objekt reprezentujúci jej výsledok sa po sieti tiež prenášajú komprimované. Pre komprimáciu dát v klientskej časti využívam modul s názvom *pako* [131]. Napriek tomu, že komprimácia dát pred odoslaním po sieti vyžaduje nejaký čas, doba, za ktorú server prijme výsledok odoslanej práce sa však paradoxne môže skrátiť. A to preto, že prenos nekomprimovaných dát po sieti môže trvať dlhšie ako kompresia, prenos komprimovaných dát a následná dekompresia.

4.2.4 Sandbox na serveri

V sekcii 3.11 som písal o tom, že je treba zabezpečiť, aby kód, ktorý napísali užívatelia a ktorý sa spúšťa na serveri nebol schopný spôsobiť škodu. Užívateľský kód by nemal mať prístup k súborovému systému, nemal by mať možnosť meniť chovanie procesu, v ktorom beží a podobne. Delenie dát úlohy a zlučovanie výsledkov pomocou užívateľom definovaných funkcií preto prebieha v bezpečnom behovom prostredí (anglicky sandbox). Využívam na to modul VM2 [132]. Vďaka tomuto modulu môže framework regulovať, k čomu má užívateľský kód prístup. Autori modulu uvádzajú, že ich sandbox je imúnny voči všetkým známym metódam útoku. Modul pomáha chrániť framework aj pred zahltením procesov nikdy nekončiacimi funkciami a to tak, že je možné nastaviť maximálnu dobu behu funkcie. Táto doba je nastaviteľná v konfigurácií frameworku.

4.2.5 Sandbox u klienta

Webové prehliadače spúšťajú JavaScriptový kód v bezpečnom prostredí, ktoré neumožňuje prístup k súborovému systému, procesoru a ďalším prostriedkom v počítači klienta. To, že framework spúšťa výpočet práce v objekte *Web Worker* zároveň zabraňuje spúšťanému kódu manipulovať s komponentami stránky a užívateľským rozhraním. To ale nie je dostatočne bezpečné pre nedôveryhodný kód, ktorý aj z prostredia *Web Worker* môže napríklad komunikovať s ľubovolným serverom pomocou objektu *XMLHttpRequest*. Kód na výpočet práce je preto spúšťaný v bezpečnom prostredí, ktoré som implementoval na základe [133]. Implementácia je jednoduchá a spočíva v tom, že sa vytvorí *Web Worker*, v ktorom sa zakáže používať všetko okrem frameworkom určených objektov. Zákaz používania objektu, atribútu, alebo funkcie som implementoval tak, že framework nastaví hodnotu premennej, v ktorej mal byť objekt na nedefinovanú hodnotu, alebo objekt vymaže. Prípadne hodnotu nastaví na funkciu, ktorá jediné čo spraví je, že vyhodí výnimku.

4.2.6 Caching

Ak objekt *WorkUnit* neobsahuje kód výpočtu práce, ale len jeho identifikátor, vyžiada si relácia kód výpočtu od servera pomocou tohto identifikátora. Server do odpovede pridá nasledujúce HTTP hlavičky:

- hlavička *Cache-Control: public* indikuje, že odpoveď si môže do cache pamäti uložiť akýkoľvek sieťový prvok, prehliadač a podobne [134]
- hlavička *Cache-Control: max-age = <workingFunctionCacheMaxAge>* udáva maximálnu dobu (v sekundách), po akú sa odpoveď v cache pamäti bude pokladať za relevantnú [134], *workingFunctionCacheMaxAge* je parameter nastaviteľný v konfigurácií frameworku
- hlavička *Last-Modified: <createdAt>* hovorí o tom, kedy bol kód (a teda obsah odpovede) naposledy zmenený [135], parameter *createdAt* je časová značka vytvorenia prototypu
- hlavička *ETag: <etag>* je identifikátor pre špecifickú verziu odpovede obsahu odpovede (a teda výpočtového kódu). Prehliadače ju využívajú na to, aby zistili, či odpoveď, ktorú majú uloženú v cache sa líši od tej, ktorú by poslal server. Ak sa obsah odpovede nezmenil, server nemusí posielať celú odpoveď znovu [136]. Parameter *etag* sa vypočíta ako haš daného kódu

4.2.7 Využitie viacerých procesorov u klienta

Vďaka využívaniu technológie *Web Worker* môže framework využiť viacero jadier procesorov klientského počítača. Niektoré webové prehliadače pre každý

WebWorker objekt vytvoria vlákno na úrovni operačného systému [28]. *Web Worker* objekt na spracovanie prác pre túto sekciu nazvem *pracovník*. Výpočtový klient vytvorí toľko pracovníkov, koľko je počet jadier procesora mínus jeden. Mínus jeden preto, aby aspoň jedno jadro ostalo voľné pre ostatné činnosti frameworku, ako je posielanie *heartBeat* alebo odosielanie výsledku a ostatné činnosti webového prehliadača a ďalších spustených programov v počítači klienta. Ak má klientský počítač len jeden procesor s jedným jadrom, vytvorí sa jeden pracovník. Server má možnosť zaslať klientovi konfiguráciu, podľa ktorej výpočtový klient určí, koľko pracovníkov sa má na spracovanie prác využívať. V konfigurácii sa neposiela presné číslo, ale údaj o tom, či sa majú využiť všetci vytvorení pracovníci, polovica, alebo len jeden.

Práce, ktoré výpočtový klient získa, pridá do zoznamu prác na vykonanie. Hlavné vlákno výpočtového klienta potom postupne posiela jednotlivé práce na výpočet nakonfigurovanému počtu pracovníkov. O prideľovanie prác pracovníkom sa stará modul s názvom *WorkManager*. Na začiatku spracovania prác sú všetci pracovníci k dispozícií pre spracovanie práce. Každému využívanému pracovníkovi sa pošle jedna práca a odstráni sa zo zoznamu prác na spracovanie. Keď pracovník vráti výsledok, odošle sa na server a ak zoznam prác na spracovanie nie je prázdny, pridelí sa mu ďalšia práca. Toto sa opakuje, kým nie je zoznam prác prázdny.

4.2.8 Implementácia prototypu úlohy a používanie funkcií prototypu frameworkom

V tejto sekcií popíšem ako má vyzerať implementácia prototypu úlohy a ako framework využíva implementované funkcie prototypu úlohy. Konkrétny príklad implementácie je v prílohe A. Tvorca prototypu úlohy musí implementovať funkciu, prípadne funkcie, pre výpočet práce a ich kód uložiť do objektu atribútu *workingFunctions*. Tieto funkcie potom framework využije na výpočet práce. Tvorca prototypu musí implementovať funkciu *jsWorkingFunction*, alebo funkcie *asmJSWorkingFunction* a *cplusplusWorkingFunction*. Funkciu *jsWorkingFunction* framework použije v prípade, keď sa rozhodne využiť na výpočet len JavaScript. Ako tvorca frameworku odporúčam, aby tvorca úloh vždy túto funkciu implementoval. Užívateľ môže implementovať funkciu pre výpočet aj v jazyku C++ a uložiť ju do *cplusplusWorkingFunction*. Server túto funkciu potom pomocou nástroja *Emscripten* skompiluje do modulov v jazykoch asm.js a wasm. Framework priamo nevyužíva skompilované moduly. Pre výpočet pomocou skompilovaných modulov framework najskôr inicializuje modul a v kontexte behového prostredia, ktoré vygeneroval nástroj Emscripten, zavolá funkciu *asmJSWorkingFunction*. Táto funkcia slúži hlavne ako rozhranie medzi frameworkom a skompilovanými modulmi. Framework poskytuje užívateľovi podporné funkcie pre prácu so skompilovanými modulmi. Funkcie *jsWorkingFunction* a *asmJSWorkingFunction* implementuje užívateľ v jazyku JavaScript. Obom funkciám framework odovzdá jediný objekt, v ktorom

sú dáta pre výpočet. Framework očakáva ľubovolne štrukturovanú návratovú hodnotu. Objekt vstupných dát môže byť ľubovolne štrukturovaný. Štruktúru objektu dát udáva tvorca prototypu úloh. Návratovou hodnotou môže byť hodnota *null*, avšak hodnota *undefined* sa nepovažuje za validný výsledok. Funkcie môžu vyhodiť výnimku, ktorú framework odchytí a odošle chybu na server.

Do atribútu *emscriptenFlags* môže tvorca prototypu uložiť niektoré z príznakov pre kompilátor Emscripten. Príznaky, ktoré môže užívateľ určiť sú úroveň optimalizácie a názvy funkcií, ktoré má Emscripten exportovať.

Tvorca prototypu môže voliteľne implementovať zlučovaciu a deliacu funkciu (v jazyku JavaScript). Server pomocou deliacej funkcie automaticky rozdelí veľké dáta úlohy na menšie a vytvorí tak čiastkové práce. Deliaca funkcia dostane od frameworku jediný argument nasledujúcej štruktúry:

```
{
  data: workData,
  serverSideMetadata: metadata
}
```

kde *workData* sú dáta rozdeľovanej úlohy a *metadata* sú metadáta, ktoré vytvorila deliaca funkcia v predchádzajúcom kroku pre práve rozdeľovanú úlohu.

Ako návratovú hodnotu očakáva framework pole objektov s rovnakou štruktúrou ako má objekt na vstupe. Pričom *workData* reprezentuje jednu časť rozdelených dát a *metadata* je ľubovolný objekt, ktorý zostane uložený na serveri a odovzdá sa do ďalšieho kroku deliacej funkcie a neskôr sa odovzdá aj zlučovacej funkcii. Ak sú dáta po rozdelení stále príliš veľké, postupne sa na ne znovu zavolá deliaca funkcia. Ak sa už dáta nemajú ako rozdeliť, framework očakáva návratovú hodnotu *null*. Tvorca prototypu úlohy môže definovať odporúčanú veľkosť dát (atribút *suggestedMaxDataSizeInBytes*), ktorá sa bude považovať za dostatočne malú pre odoslanie klientovi.

Pri zlučovaní výsledkov framework predá zlučovacej funkcií pole objektov nasledujúcej štruktúry:

```
{
  result: result,
  serverSideMetadata: metadata
}
```

kde *result* je výsledok práce a *metadata* je objekt, ktorý uložila deliaca funkcia k dátam, z ktorých vznikol daný výsledok. Framework do zlučovacej funkcie v jednom kroku odovzdáva objekty s výsledkami, ktoré vznikli z dát, ktoré vznikli rozdelením v jednom kroku. Pole objektov výsledkov zachováva poradie rovnaké ako malo pole dát, z ktorých výsledky vznikli. Inak povedané framework zachováva štruktúru rozdelenia dát a zlučovanie výsledkov prebieha presne v tej štruktúre ako sa delilo, len opačne. Príklad: deliaca funkcia rozdelí dáta na časti a, b, c a dáta a potom rozdelí na dáta $a1$, $a2$, $a3$. Z dát $a1$, $a2$, $a3$, b, c vzniknú výsledky $A1$, $A2$, $A3$, B, C. Zlučovacia funkcia sa najskôr zavolá na pole s výsledkami $A1$, $A2$, $A3$, ktoré sú v poli usporiadané v tomto poradí.

Výsledok zlučovania A potom spolu s výsledkami B a C vytvoria vstupné pole pre ďalší krok zlučovacej funkcie. Ako návratovú hodnotu framework očakáva ľubovolný objekt reprezentujúci výsledok zlúčenia.

Pre názornú ukážku implementácie prototypu úlohy, v prílohe A uvádzam ukážkové kódy pre úlohu výpočtu determinantu matice naivným algoritmom.

4.3 API pre užívateľa

Knižnica pre užívateľa implementovaná pre jazyk JavaScript poskytuje nasledujúce API:

- *getTaskPrototypesListAsync()* – táto funkcia slúži na získanie zoznamu prototypov, ktoré sú uložené na serveri.
- *postTaskPrototypeAsync(taskPrototype)* – táto funkcia slúži na odoslanie prototypu úlohy. Funkcii sa odovzdá prototyp úlohy v podobe objektu typu *TaskPrototype*. Návratovou hodnotou je odpoveď servera. Ak nastala chyba pri spracovaní prototypu, obsahuje odpoveď informáciu o nej.
- *postTaskAsync(task)* – táto funkcia slúži na manuálne odoslanie úlohy. Funkcii sa odovzdá úloha v podobe objektu typu *Task*. Návratovou hodnotou je odpoveď servera. Odpoveď obsahuje identifikátor úlohy, ktorý server úlohe priradil. Ak nastala chyba pri spracovaní úlohy, obsahuje odpoveď informáciu o nej.
- *getTaskAsync(taskId)* – táto funkcia slúži na manuálne získanie objektu úlohy zo serveru. Funkcii užívateľ odovzdá identifikátor, ktorý vrátila funkcia *postTaskAsync*. Pomocou tejto funkcie môže užívateľ manuálne zistiť aktuálny stav úlohy.
- *downloadResultToFileAsync(taskId, fileDirectoryPath, fileName)* – pomocou tejto funkcie môže užívateľ manuálne stiahnuť výsledok úlohy. Užívateľ odovzdá funkcii identifikátor úlohy, cestu do zložky a meno súboru, kam sa má výsledok stiahnuť.
- *deleteTaskAsync(taskId)* – táto funkcia slúži na vymazanie dokončenej úlohy.
- *computeAsync(taskPrototypeId, data, directoryPath, fileName, defaultReplicationFactor = 3, pollingIntervalMs = 1000)* – táto funkcia zjednodušuje využívanie frameworku a umožňuje, aby sa výpočet úlohy pomocou frameworku javil ako lokálny výpočet. Predpokladá sa, že užívateľ bude bežne používať túto funkciu a funkcie *postTaskAsync*, *getTaskAsync*, *downloadResultToFileAsync* a *deleteTaskAsync* len ojedinele. Užívateľ odovzdá funkcii identifikátor prototypu úlohy, dáta a zložku

a meno súboru kam sa výsledok stiahne. Voliteľne môže zadať minimálny počet výsledkov, z ktorých má majoritné hlasovanie určiť výsledok. Východiskový počet je 3. Voliteľne môže zadať aj dĺžku intervalu v milisekundách, v ktorom sa má kontrolovať, či je úloha už dokončená. Východisková hodnota je jedna sekunda. Funkcia zabezpečí odoslanie úlohy, priebežne kontroluje, či sa úloha dokončila a nakoniec stiahne výsledok, ktorý odovzdá ako návratovú hodnotu.

4.4 HTTP API

4.4.1 ProgrammerServer

ProgrammerServer poskytuje nasledujúce HTTP API, ktoré využíva aj vyššie uvedená knižnica:

- *GET: /task-prototypes* – ako odpoveď na požiadavku na túto URL server vráti pole objektov *TaskPrototype*, ktoré reprezentujú prototypy úloh. Požiadavka môže v rámci *query string* obsahovať voliteľne parametere:
  - *sortKey* – reťazec, ktorý určuje, podľa ktorého parametra sa má zoznam prototypov úloh zoradiť
  - *sort* – určuje, či má byť pole utriedené vzostupne alebo zostupne. Môže mať hodnotu 1 alebo −1
  - *limit* – určuje maximálny počet objektov, ktoré má odpoveď obsahovať
  - *skip* – určuje počet objektov na začiatku poľa, ktoré sa majú preskočiť a nepridať do výsledku

  Odpoveď má nasledujúcu štruktúru:

```
1 {taskPrototypes: prototypesArray}
```

  kde *prototypesArray* je pole objektov prototypov úloh.

- *GET: /task-prototypes/:taskPrototypeId/* – táto URL slúži na získanie konkrétneho objektu *TaskPrototype*, pričom *:taskPrototypeId* treba nahradiť identifikátorom prototypu úlohy. Ak prototyp úlohy so zadaným identifikátorom neexistuje, vráti server hodnotu *null* a HTTP kód odpovede nastaví na 404. HTTP kód 404 indikuje, že server nevie nájsť požadovaný zdroj [137]. Odpoveď má nasledujúcu štruktúru:

```
1 {taskPrototype: prototype}
```

  kde *prototype* je objekt prototypu úlohy alebo hodnota *null*.

- *GET: /tasks/:taskId* – pomocou tejto URL môže užívateľ získať objekt *Task*, ktorý reprezentuje odoslanú úlohu. Ak úloha neexistuje, vráti server odpoveď s HTTP kódom 404. Server tiež kontroluje, či požiadavka prichádza od užívateľa, ktorý úlohu zadal. Ak nie, vráti HTTP kód 403 a namiesto úlohy vráti hodnotu *null*. Odpoveď má nasledujúcu štruktúru:

```
{task: task}
```

  kde *task* je objekt reprezentujúci úlohu alebo hodnota *null*.

- *GET: /tasks/:taskId/result* – táto URL slúži na prevzatie výsledku úlohy, pričom *:taskId* treba nahradiť identifikátorom úlohy. Ak úloha s daným identifikátorom neexistuje, vráti server odpoveď s HTTP kódom 404. Aj tu server kontroluje, či požiadavka prichádza od užívateľa, ktorý úlohu zadal. Ak nie, vráti HTTP kód 403. Ak úloha existuje, ale ešte nie je dokončená, HTTP kód odpovede je 204. Ten indikuje, že server požiadavku spracoval úspešne, ale nevracia žiadny obsah [138]. Ak je všetko v poriadku, odpoveďou servera je objekt výsledku úlohy.

- *POST: /task-prototypes/* – pre uloženie nového prototypu úlohy na server je treba poslať HTTP POST požiadavku na túto URL. Telo požiadavky by malo obsahovať objekt *TaskPrototype*. Ak objekt prototypu úlohy neobsahuje všetky potrebné náležitosti, server vráti informácie o chybe a HTTP kód odpovede nastaví na 400, ktorý indikuje, že server nerozumie požiadavke [139]. V odpovedi server uvedie informácie o chybe. V prípade, že nastane chyba pri spracovaní, server nastaví HTTP kód odpovede na 500, čo indikuje internú chybu na serveri [140]. V prípade, že žiadna chyba nenastala a prototyp úlohy sa úspešne uložil, vráti server odpoveď s HTTP kódom 201, ktorý indikuje úspech pri vytvorení zdroja [141]. Odpoveď má nasledujúcu štruktúru:

```
{error: err}
```

  kde *err* obsahuje informácie o chybe, ktorá nastala, alebo hodnotu *null*, ak žiadna chyba nenastala.

- *POST: /tasks/* – pomocou HTTP POST požiadavky na túto URL server umožňuje vytvorenie úlohy. Server v tele požiadavky očakáva objekt štruktúry:

```
{
  data: data,
  taskPrototypeId: taskPrototypeId,
  defaultReplicationFactor: defaultReplicationFactor
}
```

kde *data* sú dáta úlohy, *taskPrototypeId* je identifikátor prototypu úlohy a *defaultReplicationFactor* je voliteľný parameter pre určenie minimálneho počtu výsledkov od klientov, z ktorých má framework pomocou majoritného hlasovania určiť výsledok. Server odpovedá obdobne ako na požiadavku o vytvorenie prototypu úlohy, pričom do odpovede vkladá identifikátor vytvorenej úlohy. Odpoveď má nasledujúcu štruktúru:

```
1 { taskId :taskId, error:err }
```

- *DELETE: /tasks/:taskId* – pomocou HTTP DELETE požiadavky na túto URL server vymaže dokončenú úlohu, pričom *:taskId* treba nahradiť identifikátorom úlohy na odstránenie. Ak úloha neexituje, server nastaví HTTP kód odpovede na 404. Server kontroluje, či požiadavka prichádza od užívateľa, ktorý úlohu zadal. Ak nie, vráti HTTP kód 403. Ak úloha nie je dokončená, server ju nevymaže a HTTP kód odpovede nastaví na 409, čo indikuje konflikt so stavom serveru [142]. V prípade chyby na serveri je HTTP kód odpovede 500.

Každá požiadavka na *ProgrammerServer* musí v cookies obsahovať parameter *apiKey*, s hodnotou API kľúča užívateľa. V opačnom prípade sa požiadavka ignoruje a HTTP kód odpovede je 401, ktorý indikuje, že požiadavka je neautorizovaná [143]. Ak nie je povedané inak, server vracia HTTP kód 200, ktorý indikuje úspech pri spracovaní požiadavky [144].

4.4.2 VolunteerServer

VolunteerServer komunikuje s výpočtovým klientom pomocou nasledujúceho HTTP API:

- *GET: /work* – pri prístupe na túto URL získa klient pole *workUnit* objektov reprezentujúce práce
- *GET: /workingfunction/:taskPrototypeId/:type* - na tejto URL klient získa kód pre výpočet práce, pričom parameter *:taskPrototypeId* je uvedený v objekte práce a parameter *:type* určuje technológiu, ktorej kód, prípadne kódy klient žiada. Parameter *:type* môže mať nasledujúce hodnoty:
  - *wasm->asm->js* – táto hodnota indikuje, že klient chce použiť technológiu WebAssembly. Ak nie sú kódy pre túto technológiu k dispozícií, žiada kódy pre technológiu asm.js a ak ani tie nie sú k dispozícií, tak žiada kód pre vykonanie výpočtu v čistom JavaScripte
  - *asm->js* – táto hodnota indikuje, že klient chce použiť technológiu asm.js. Ak nie sú kódy pre túto technológiu k dispozícií, tak žiada kód pre vykonanie výpočtu v čistom JavaScripte

  - *only-wasm* – touto hodnotou klient indikuje, že žiada o kódy pre použitie technológie WebAssembly. Ak nie sú k dispozícií, nedostane žiadny kódy
  - *only-asm* – funguje tak isto ako *only-wasm*, ale pre technológiu asm.js
  - *only-js* – funguje tak isto ako *only-wasm*, ale klient žiada kódy pre vykonanie výpočtu v čistom JavaScripte

- *POST: /result* – klient odosiela výsledok na túto URL, pričom odpoveď obsahuje ďalšie pridelené práce
- *POST: /onlyresult* – klient odosiela výsledok na túto URL, pričom odpoveď je prázdna
- */heartbeat* – táto URL slúži pre zasielanie *heartBeat* požiadavky, pri ktorej server aktualizuje čas posledného prístupu relácie, pričom nezáleží na HTTP metóde požiadavky. Implementácia klienta používa HTTP metódu OPTIONS

Výpočtový klient pri každej požiadavke zasiela v *cookies* identifikátor klienta v atribúte *clientId* a v *query string* zasiela identifikátor relácie ako parameter *sessionId*.

4.5 Zapojenie webovej stránky do frameworku

Pre zapojenie webovej stránky do frameworku stačí, keď správca stránky vloží do zdrojového kódu stránky nasledujúci kód:

```
1 <script src="https://<host>/client.js"></script>
```

kde *host* je adresa, na ktorej je dostupný *VolunteerServer*.

Po načítaní stránky s týmto kódom sa načíta kód programu výpočtového klienta, ktorý sa v zápätí spustí. Klientský kód frameworku sa však nespustí na mobilných zariadeniach kvôli tomu, že dáta prijaté mobilným zariadením môžu majiteľa zariadenia stáť nemalé peniaze.

4.6 Obmedzenie veľkosti úlohy a jej výsledku

Z implementácie frameworku vyplýva obmedzenie na veľkosť úlohy a jej výsledku. Pri delení pomocou deliacej funkcie sa v prvom kroku odovzdajú celé dáta úlohy. Z toho vyplýva, že dáta sa musia zmestiť do operačnej pamäte servera. Rovnako v poslednom kroku zlučovania výsledkov prác do výsledku úlohy je návratovou hodnotou funkcie výsledok úlohy, ktorý sa opäť musí zmestiť do operačnej pamäte servera.

Kapitola 5

Experimenty a vyhodnotenie

Po implementovaní prototypu som spravil niekoľko experimentov, v ktorých som testoval vplyv parametrov frameworku a jeho prostredia na rôzne ukazovatele. Experimenty a ich výsledky popíšem v tejto kapitole.

5.1 Prostredie a základné nastavenie frameworku

Pre experimenty som vytvoril testovací server, ktorý poskytoval jedinú webovú stránku, ktorá mala minimálny obsah. Obsahovala však kód, ktorý načítal a spustil program výpočtového klienta frameworku. Stránka obsahovala aj skript, ktorý stránku čas od času obnovil. Obnovením stránky z pohľadu frameworku zaniká relácia a vzniká nová. To, po akom čase od načítania sa stránka obnoví je dané nasledujúcim vzťahom:

$$refreshAfter = random() \cdot maxDuration$$

kde $refreshAfter$ je počet sekúnd, po ktorých sa stránka obnoví, $random$ je funkcia, ktorá vracia náhodné čísla od 0 po 1 a $maxDuration$ je parameter, ktorý udáva maximálnu dobu od načítania, po ktorej sa stránka obnoví.

Experimenty som robil v laboratórnych podmienkach na mojej fakulte. Zapojené počítače sa nachádzali v učebniach T9-350, T9-351 a T9-303. Jeden test prebiehal v učebni T9-349. Konfigurácie počítačov v učebniach sú v tabuľke 5.1. Na každom počítači bol spustený len webový prehliadač. Na polovici počítačov bol spustený Chrome [70] verzie 64.0 a na polovici Firefox [71] vo verzií 52.6. Testy prebiehali spolu na 60 počítačoch.

Verzia prehliadača Firefox nainštalovaná na počítačoch v učebniach nepodporovala technológiu WebAssembly a tak som sa rozhodol robiť experimenty s využitím funkcie *jsWorkingFunction*, ktorá využíva iba JavaScript.

Úloha, ktorú som používal pri experimentoch bola výpočet determinantu matice naivným algoritmom. Východisková veľkosť matice pri testoch bola 13×13. Na výpočet sa klientovi posielala práca obsahujúca maticu veľkosti

Učebňa	Model procesora	Veľkosť RAM v GB
T9-350	Intel® Core™ i5-6500 CPU @ 3.20GHz	16
T9-351	Intel® Core™ i5-3470 CPU @ 3.20GHz	8
T9-303	Intel® Core™ i5-3470 CPU @ 3.20GHz	8
T9-349	Intel® Core™ i5-4570S CPU @ 2.90GHz	8

Tabuľka 5.1: Konfigurácia testovacích počítačov

maximálne 11×11 a *defaultReplicationFactor* bol nastavený na hodnotu 3. Z úlohy sa vytvorilo 156 prác pre distribúciu, pričom každá sa musela vypočítať minimálne trikrát. Z toho vyplýva, že pre výpočet úlohy muselo prebehnúť minimálne 468 výpočtov na strane klientov. Implementácia prototypu úlohy je v prílohe A. Stratégiu distribúcie prác som pre testovanie zvolil tú, ktorú som navrhol pre *SuperSimpleScheduler* a počet prác, ktoré sa majú relácii poslať v jednej odpovedi na požiadavku o prácu (atribút *worksUnitsToClientCount*) som nastavil na 4.

Všetky serverové časti, vrátane databázového stroja a testovacieho servera, bežali na jednom notebooku. Model procesora počítača servera bola Intel® Core™ i5-2410M CPU @ 2.30GHz, veľkosti RAM boli 2048 MB a 4096 MB, obe RAM boli typu DDR3 s rýchlosťou 1333 MHz. Pevný disk serveru bol typu SSD.

Východiskové nastavenia vybraných parametrov frameworku pre experimenty sú v tabuľke 5.2.

Parameter	Hodnota	Popis
deadSessionCollector:		
interval	2 000	Dĺžka intervalu v milisekundách, v ktorom sa vykonávajú kontroly mŕtvych relácií
deadTime	10 000	Určuje, po akej dobe (v milisekundách) od poslednej požiadavky sa má relácia prehlásiť za mŕtvu
LongRunningSessionCollector:		
interval	5 000	Dĺžka intervalu v milisekundách, v ktorom sa vykonávajú kontroly relácií, ktoré príliš dlho vykonávajú výpočet práce
factor	5	Koľkokrát dlhšie musí trvať výpočet práce v relácii oproti priemernej dobe výpočtu, aby bola relácia prehlásená za dlhotrvajúcu pri danej práci
maxRunningTime	360 000	Maximálna doba (v milisekundách) výpočtu práce v relácii, po ktorej je relácia prehlásená za dlhotrvajúcu pri danej práci
Ďalšie nastavenia:		
maxDuration	960	Maximálna doba v sekundách, po ktorej sa obnoví testovacia stránka
throttleFactor	0	Koľkonásobok doby výpočtu pridelených prác sa má čakať, kým framework môže niektorej z klientových relácií zase prideliť práce
getWorkRetryInterval	1 000	Dĺžka doby v milisekundách, po ktorej má relácia znovu žiadať prácu, ak jej naposledy server žiadnu nepridelil.
heartBeatInterval	5000	Dĺžka intervalu, v ktorom má relácia zasielať *heartBeat* požiadavky
useCPUs	all	Indikuje, že výpočtový klient má využiť všetky vytvorené *Web Worker* objekty pre spracovanie prác
separateDataAndFunctions	true	Indikuje, že server má v objektoch *WorkUnit* zasielať klientovi len identifikátory kódu.
CacheMaxAge	180	Doba v sekundách, počas ktorej sa odpoveď zo servera uložená v cache pamäti považuje za relevantnú.

Tabuľka 5.2: Východiskové nastavenia vybraných parametrov frameworku pre experimenty

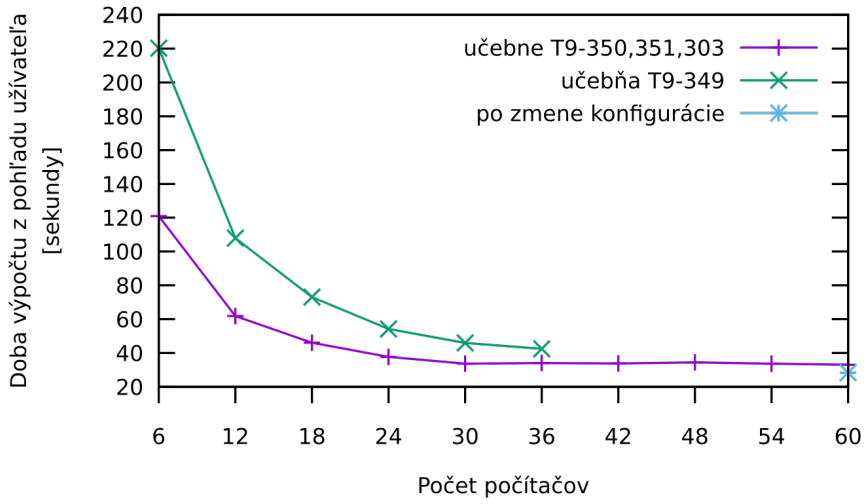
5.2 Experimenty

Každý experiment som spravil päťkrát a za výsledok testu som prehlásil priemer jednotlivých behov. Nová úloha na spracovanie sa frameworku poslala vždy až po tom, ako framework vrátil výsledok predchádzajúcej úlohy. Z toho plynie, že framework vždy pracoval na výpočte jedinej úlohy.

5.2.1 Vplyv počtu počítačov

Prvým experimentom bol vplyv počtu počítačov, respektíve klientov. Pred týmto experimentom bol parameter konfigurácie *DeadSessionCollector.interval* nastavený na 1 000 milisekúnd, *DeadSessionCollector.deadTime* na 5 000 milisekúnd a *LongRunningSessionCollector.factor* na hodnotu 3. Experiment som urobil v učebni T9-349 a následne v učebniach T9-350, T9-351 a T9-303.

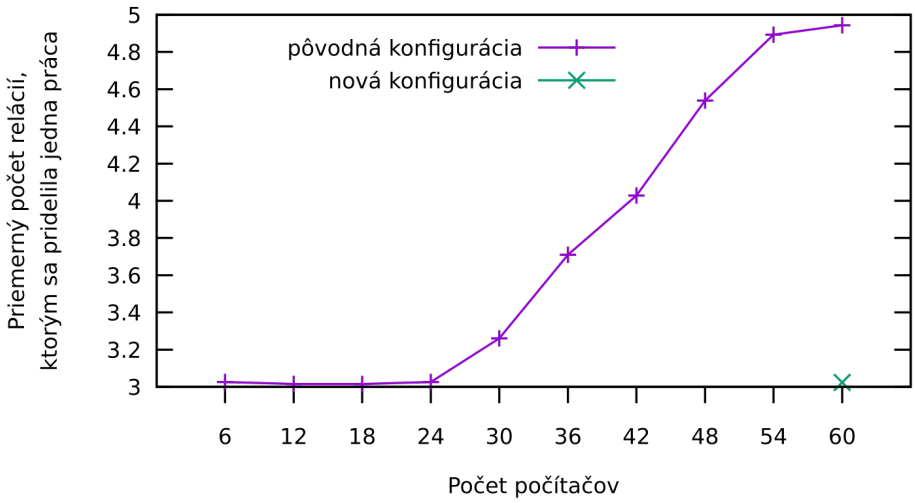
Graf 5.1 zobrazuje vplyv počtu počítačov na dobu výpočtu úlohy z pohľadu užívateľa, čiže dobu od odoslania dát na server po prijatie výsledku. Z grafu vidno, že výpočet na počítačoch v učebni T9-349 prebiehal dlhšie ako v ostatných učebniach, ale zvyšovaním počtu počítačov tento rozdiel postupne zanikal, pretože každému klientovi bolo prideľovaných menej prác a zároveň sa práce vypočítavali súčasne. V súvislosti s veľkosťou webu tento výsledok predpovedá, že framework je schopný niektoré typy úloh vyriešiť veľmi rýchlo a to aj v prípade, že sa do výpočtu zapoja menej výkonné počítače.



Obr. 5.1: Vplyv počtu klientských počítačov zapojených do frameworku na čas výpočtu úlohy z pohľadu užívateľa frameworku

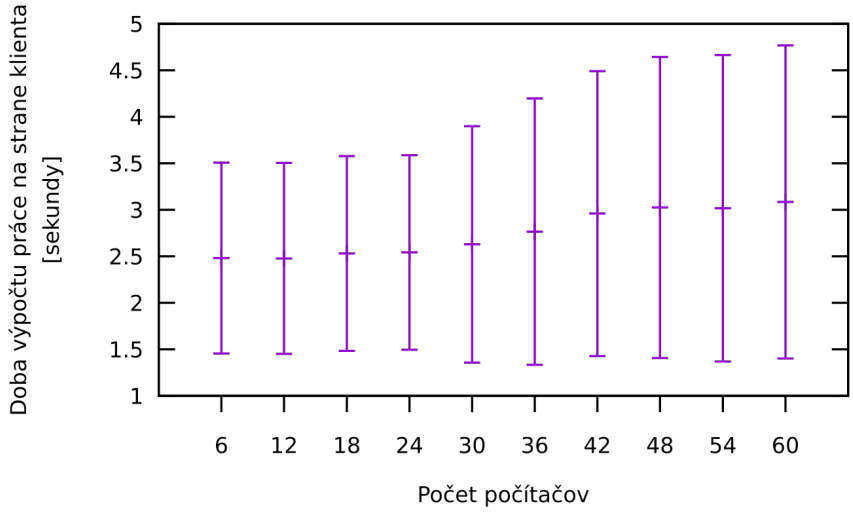
Pridávaním počítačov pri testovaní v učebniach T9-350, T9-351, T9-303 prestalo zrýchľovanie rásť pri počte 30. To mohlo byť spôsobené najmä tým, že framework začal jednu prácu priraďovať viacerým reláciám (zobrazuje graf 5.2),

čo sa dialo pravdepodobne preto, že počítače v učebni T9-350 sú výkonnejšie ako v zvyšných učebniach (zobrazuje graf 5.3) a preto framework mohol niekoľko relácií prehlásiť za pomalé. Spomalenie, respektíve zastavenie zrýchľovania mohlo byť do istej miery spôsobené aj tým, že pri viacerých klientoch je pravdepodobnosť obnovy stránky na niektorom z nich väčšia a istú dobu frameworku trvá kým to rozpozná a zareaguje. Zároveň interval posielania *heartBeat* som nesprávne nastavil na rovnakú hodnotu ako dobu, po ktorej sa relácia prehlási za mŕtvu a preto, ak výpočet práce (spolu s prenosom dát) na pomalších počítačoch trval dlhšie ako daná doba, prehlásila sa relácia za mŕtvu. Preto som potom nastavenia pre moduly *DeadSessionCollector* a *LongRunningSessionCollector* zmenil na tie, ktoré sú uvedené v tabuľke 5.2. V grafe 5.1 vidno, že po zmene konfigurácie sa výpočet zrýchlil. Znížil sa aj počet relácií, ktorým bola pridelená jedna práca, čo ukazuje graf 5.2.

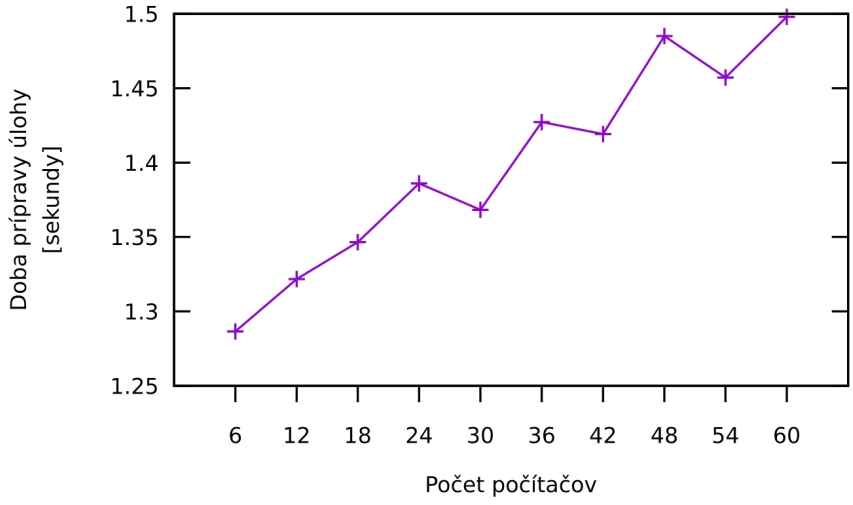


Obr. 5.2: Vplyv počtu klientských počítačov zapojených do frameworku na priemerný počet relácií, ktorým framework pridelil jednu prácu

Znižovanie zrýchľovania mohlo byť spôsobené do istej miery aj tým, že pri väčšom počte klientov sa zvyšuje počet požiadaviek, ktoré musí server, ale aj databázový stroj obslúžiť. Počas experimentov som pozoroval, že spustená MongoDB databáza v niektorých momentoch značne zaťažovala procesor počítača serveru. To malo pravdepodobne dopad aj na dĺžku prípravy úlohy na distribúciu (zobrazuje graf 5.4). Modul *TasksPreparationManager* bežal vo vlastnom procese, preto si myslím, že predlžovanie času prípravy úlohy na distribuovanie bolo spôsobené tým, že modul musel čím ďalej tým dlhšie čakať na procesor, ktorý bol čím ďalej tým viac zaťažovaný najmä databázou. Zvýšiť výkon frameworku by sa mohol po optimalizovaní volaní do databázy, prípadne nasadením databázy na iný stroj, aby si navzájom s procesmi serveru nebránili vo využívaní procesora.



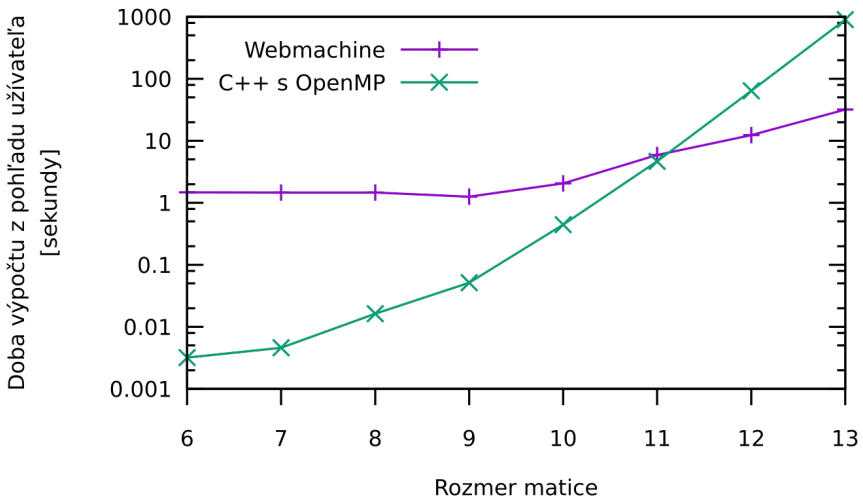
Obr. 5.3: Štatistiky doby výpočtu jednej práce u klienta pri rôznom počte počítačov zapojených do výpočtu úlohy.
Pre každý počet počítačov na osi X je na osi Y znázornený priemer (prostredná z hodnôt) a priemer ± štandardná odchýlka doby výpočtu jednej práce na strane klienta



Obr. 5.4: Vplyv počtu klientských počítačov zapojených do frameworku na dobu prípravy úlohy

5.2.2 Vplyv veľkosti matice

Ďalší experiment, ktorý som urobil, bol na zistenie vplyvu veľkosti dát, v tomto prípade veľkosti matice na dobu výpočtu z pohľadu užívateľa. Zároveň som v tomto experimente testoval význam frameworku a to tak, že som výpočet matice implementoval v jazyku C++ s využitím knižnice OpenMP [145] tak, aby výpočet využíval všetky jadrá procesora. Následne som výpočet spustil lokálne na počítači, na ktorom bežal server frameworku, ktorý počas výpočtu spustený nebol. Výsledky testu zobrazuje graf 5.5. Z grafu vidno, že pri malých dátach je lokálny výpočet rádovo rýchlejší, ale postupne začína byť efektívnejší výpočet pomocou frameworku. Doba výpočtu matice veľkosti 13×13 sa oproti dĺžke výpočtu matice veľkosti 6×6 pri lokálnom výpočte zväčšila o 5 rádov, zatiaľ čo pri výpočte pomocou frameworku sa nezväčšila ani o 2 rády.



Obr. 5.5: Vplyv veľkosti matice na čas výpočtu úlohy z pohľadu užívateľa. Matice sú štvorcové, preto na osi X je len jeden rozmer.

Zaujímavosťou je, že výpočet pomocou frameworku a lokálny výpočet trvali pri veľkosti matice 11×11 približne rovnako dlho. Zaujímavé je to preto, že framework maticu veľkosti 11×11 nerozdelil na menšie matice, ale klientom ju distribuoval v tejto veľkosti. Z čoho vyplýva, že klientské počítače boli o toľko výkonnejšie, že lokálny výpočet s využitím viacerých jadier bol len o málo rýchlejší ako odoslanie úlohy, distribúcia prác, prijatie a spracovanie výsledkov a odoslanie výsledku užívateľovi. Je pravda, že odoslanie úlohy a prijatie výsledku úlohy trvalo minimálny čas, pretože prenos dát po sieti reálne neprebiehal a že prenos dát medzi serverom a klientmi bol rýchly, pretože boli blízko pri sebe. Na druhej strane klientské počítače na výpočet matice mohli na rozdiel od lokálneho výpočtu využiť len jedno jadro procesora. Z toho vyplýva, že ak majú návštevníci webovej stránky, na ktorej je kód,

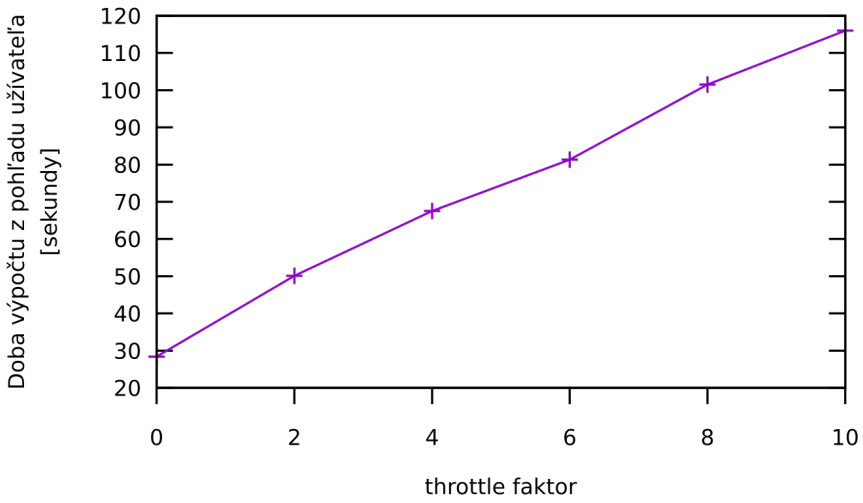
ktorý spustí výpočtového klienta frameworku výkonné počítače, dá sa využiť menej výkonný počítač ako server frameworku a pomocou neho vykonávať výpočty za kratší čas ako by prebehol lokálny výpočet na danom počítači.

Tento experiment, tak ako aj nasledujúce, prebiehal na 60 počítačoch. Pri pohľade na grafy 5.1 a 5.5 však vidno, že výpočet matice veľkosti 13×13 pomocou frameworku bol oproti lokálnemu výpočtu rýchlejší už pri šiestich zapojených počítačoch a to aj keď počítače boli z učebne T9-349.

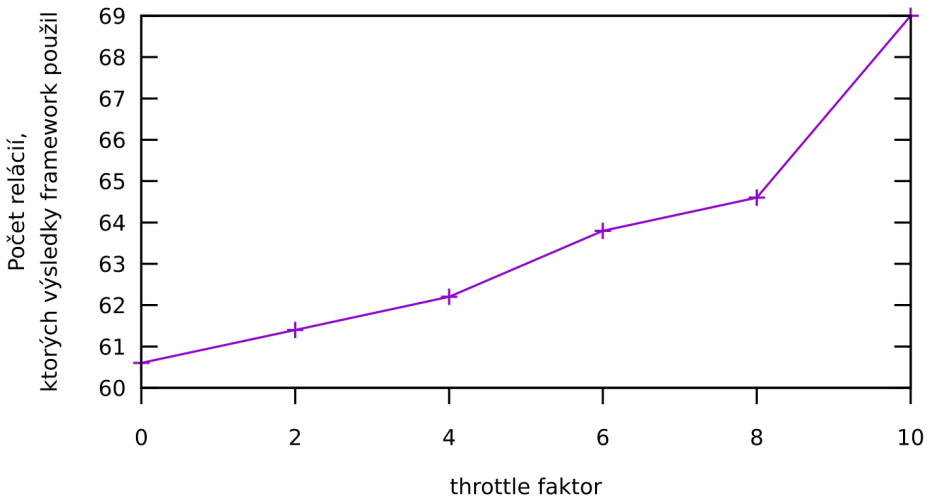
Pri pokuse o výpočet matice veľkosti 14×14 však databázový stroj veľa času zaťažoval všetky jadrá procesora a jedna požiadavka na server bola zodpovedaná približne po minúte. Výpočet som preto ani nenechal dobehnúť. Tento problém by sa mohol dať odstrániť optimalizáciou volaní do databázy, nasadením databázy na iný počítač a pravdepodobne by mohlo pomôcť, keby sa zmenil dátový model pre databázu, napríklad na ten, ktorý v sekcii 4.2.2 popisujem ako alternatívnu možnosť k aktuálnemu modelu. V aktuálnom modele sa všetky práce vytvorené pre úlohy ukladajú do databázy v rámci jedného objektu *TaskUnderScheduling*. Je možné, že efektívnejšie by bolo, ak by sa práce ukladali do osobitnej kolekcie. To by však prinášalo iné nevýhody a preto by bolo treba otestovať, aký spôsob je efektívnejší.

5.2.3 Vplyv parametru throttleFactor

V tomto experimente som testoval, aký vplyv má atribút *throttleFactor* na čas výpočtu úlohy z pohľadu užívateľa. So zvyšovaním atribútu *throttleFactor* lineárne rastie doba, počas ktorej framework nepriradí žiadnej klientovej relácii prácu. Dalo by sa očakávať, že preto bude lineárne rásť aj čas výpočtu úlohy. Z grafu 5.6 vidno, že atribút *throttleFactor* má na dobu výpočtu očakávaný vplyv. Z grafu 5.7 vidno, že zvyšovaním atribútu *throttleFactor* sa zvyšuje aj počet relácií, od ktorých prijatý výsledok sa použil pre určenie výsledku úlohy. To je spôsobené tým, že čím viac sa výpočet práce predlžuje, tým je väčšia šanca, že sa stránka obnoví, čím relácia zanikne a vznikne nová. Toto môže o niečo predĺžiť dobu výpočtu, ale nie výrazne, pretože čím väčší je atribút *throttleFactor*, tým je väčšia šanca, že relácia zanikne v dobe, keď nemá pridelené práce. Pri pohľade na grafy 5.5 a 5.6 vidno, že aj pri nastavení atribútu *throttleFactor* na hodnotu 10, čím sa z dlhodobého hľadiska obmedzí využitie procesora klientského počítača frameworkom na menej ako 10% je výpočet pomocou frameworku rýchlejší ako lokálny výpočet.



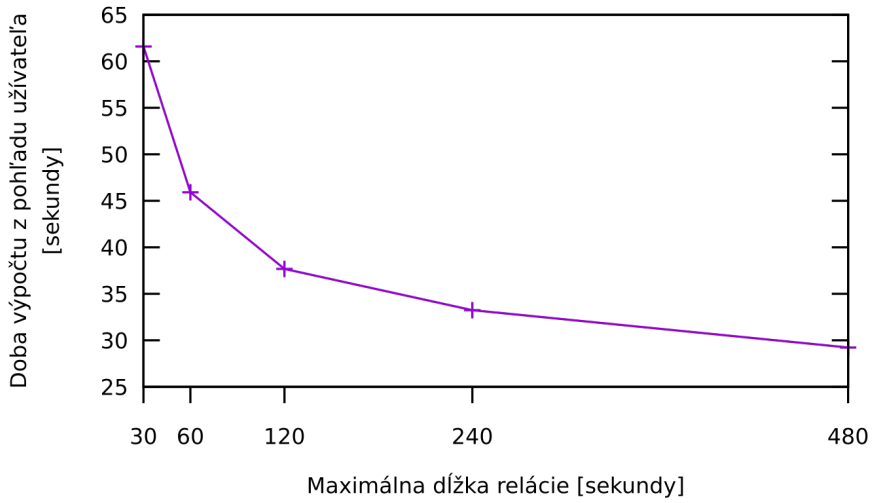
Obr. 5.6: Vplyv atribútu *throttleFactor* na čas výpočtu úlohy z pohľadu užívateľa frameworku



Obr. 5.7: Vplyv atribútu *throttleFactor* na počet relácií, ktorých výsledok sa použil pri výpočte úlohy.

5.2.4 Vplyv maximálnej doby trvania relácie

Nakoniec som testoval vplyv maximálnej doby trvania relácie na výpočet úlohy. Z grafu 5.8 vidno, že zvyšovaním maximálnej doby trvania relácie sa výpočet úlohy zrýchľuje. To sa deje preto, že so zvyšovaním doby trvania relácie klesá počet relácií, ktoré umrú a pritom mali pridelené práce, ktoré nestihli dopočítať. Tieto práce sa potom musia prideliť ďalšej relácii. Graf 5.9 znázorňuje, ako sa mení priemerný počet relácií, ktorým sa pridelí jedna práca, keď sa mení maximálna doba trvania relácie.

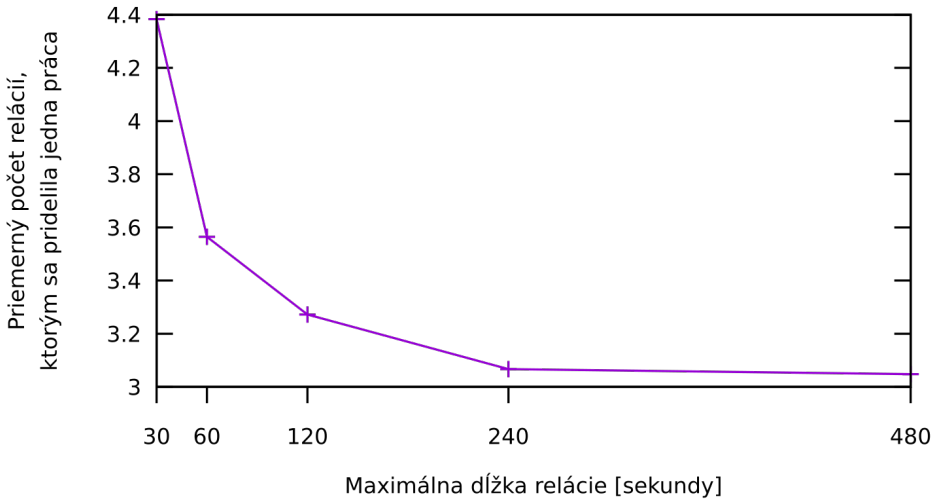


Obr. 5.8: Vplyv maximálnej dĺžky relácie na čas výpočtu úlohy z pohľadu užívateľa frameworku

5.3 Zhodnotenie experimentov

Experimenty potvrdili, že jednotlivé parametre frameworku a jeho prostredia majú očakávateľný vplyv na výpočet úlohy. Experimenty tiež ukázali, že framework so 60 klientmi dokáže niektoré typy úloh vypočítať rádovo rýchlejšie ako lokálny výpočet, pričom záťaž na procesor serveru aj klientských počítačov je menšia ako pri lokálnom viacvláknovom výpočte. Tieto výsledky naznačujú, že využívanie frameworku v produkcii môže viesť k výraznému zvýšeniu rýchlosti výpočtov bez toho, aby bolo treba kupovať nové a výkonnejšie stroje.

Na druhej strane počas experimentov sa objavili aj limity implementácie a nasadenia frameworku. Ukázalo sa, že pre reálne využitie frameworku v praxi je treba optimalizovať volania do databázy, ako aj dátový model databázy. Počas experimentov sa ukázalo aj to, že pre vyšší výkon frameworku



Obr. 5.9: Vplyv maximálnej dĺžky relácie na priemerný počet relácií, ktorým framework pridelil jednu prácu

by pravdepodobne bolo lepšie, keby databázový stroj bežal na inom počítači ako server frameworku, aby si navzájom neblokovali procesor.

Vykonané experimenty testovali vplyv základných parametrov frameworku a jeho prostredia. Pre zhodnotenie výkonu a možností využitia frameworku v praxi je treba vykonať ďalšie, komplexnejšie testy.

5.4 Návrhy na zlepšenie

Framework sa ďalej dá zlepšovať v rôznych smeroch. Ako testy ukázali, pre reálne nasadenie frameworku je nutné ho optimalizovať na viacerých miestach. To sa týka databázového modelu, volaní do databázy, ale možnosti na optimalizovanie by sa dali nájsť aj v implementácií samotného frameworku.

Zlepšiť by sa mala aj bezpečnosť systému. Framework má síce implementované viaceré bezpečnostné opatrenia, avšak stále sa nevie brániť voči všetkým známym útokom. Napríklad, server nemá mechanizmus proti útoku, ktorý by mal za cieľ zahltiť server alebo jeho databázu. Tento útok je navyše ľahko zrealizovateľný.

Framework by sa ešte mal otestovať ďalšími jednoduchými, ale aj komplexnými testami pre odhalenie ďalších možností na zlepšenie, ale aj pre analýzu jeho silných a slabých stránok a pre analýzu optimálneho nastavenia parametrov frameworku.

Stratégie pre distribúciu úloh, ktoré som pre framework navrhol sú jednoduché a základné. Zvýšiť efektivitu frameworku by sa mohlo dať aj navrhnutím novej stratégie. Tá by mohla do úvahy brať aj rýchlosť pripojenia klienta.

Pre lepšie praktické využitie by framework mal poskytovať nástroje na monitorovanie frameworku, ale aj na monitorovanie výpočtu úlohy pre užívateľa.

Rýchle a jednoduché vylepšenie frameworku môže byť aj implementovanie knižnice pre užívateľov do ďalších programovacích jazykov.

Užitočným rozšírením frameworku by mohlo byť aj implementovanie mechanizmu, ktorý by umožňoval regulovanie záťaže klientského počítača frameworkom užívateľovi klientského počítača.

Vylepšením frameworku by mohla byť aj implementácia deliaceho a zlučovacieho mechanizmu tak, aby veľkosť dát a výsledku úlohy nebola obmedzená operačnou pamäťou serveru.

Komplexnejším vylepšením frameworku môže byť správa užívateľov frameworku, webových portálov zapojených do frameworku a ľudí, ktorí chcú svoj počítač cielene poskytnúť pre výpočty. Aktuálne framework nemá takmer žiadnu správu užívateľov. Framework by do budúcna mohol umožňovať vytvárať a spravovať užívateľov. Správa užívateľov by pre praktické využitie mala byť dostupná cez grafické užívateľské rozhranie, čím môže byť webová stránka. Správa užívateľov a monitorovanie ich činnosti v rámci frameworku by potom mohla viesť k vytvoreniu trhu s prototypmi úloh, ktorý spomínam v sekcii 3.3.2. Framework by mohol tiež umožňovať správu webových portálov, ktoré ich správcovia zapojili do frameworku. To spolu so správou užívateľov môže viesť k zavedeniu poplatku za využívanie frameworku na výpočty a naopak odmeňovanie správcov portálov, prípadne priamo ich návštevníkov za vykonanie výpočtov. Framework by tiež mohol umožňovať registráciu priamo ľuďom, ktorí chcú poskytnúť svoj počítač pre výpočty, za ktoré by získavali odmenu.

Záver

V rámci tejto diplomovej práce som najskôr vytvoril rešerš z prác za posledných 11 rokov a zanalyzoval vlastnosti webu. Na základe toho som navrhol framework, ktorý je schopný pre výpočty používať počítače návštevníkov webových stránok bez nutnosti inštalácie programu či doplnku webového prehliadača na strane návštevníka. Framework som navrhol tak, aby dokázal riešiť výpadky výpočtových uzlov a reagovať na pomalý uzol. Navrhol a implementoval som bezpečnostné opatrenia na strane servera, klienta a komunikácie medzi nimi. V práci sa venujem aj bezpečnosti dát, nad ktorými framework počíta. Pre framework som tiež navrhol niekoľko stratégií distribúcie úloh.

Server frameworku som implementoval v jazyku JavaScript ako Node.js aplikáciu, ktorá dokáže využiť viaceré jadrá procesora servera. Ako databázový stroj som zvolil MongoDB. Klientskú časť frameworku som tiež implementoval v jazyku JavaScript. Pre zrýchlenie výpočtov umožňuje framework užívateľovi definovať výpočet aj v jazyku C++, ktorý sa následne preloží do modulov v jazykoch asm.js a wasm. Vďaka technológií Web Worker výpočty, ktoré framework vykonáva neobmedzujú používanie webovej stránky, v rámci ktorej je framework spustený. Navyše je možné z dlhodobého hľadiska regulovať priemerné zaťaženie procesora návštevníka webovej stránky frameworkom. Dá sa tiež nastaviť, koľko jadier návštevnikovho počítača má framework využívať.

Implementáciu som otestoval na úlohe výpočtu determinantu matice naivným algoritmom. Experimenty prebiehali v laboratórnych podmienkach spolu na 60 počítačoch. Ukázalo sa, že výpočet pomocou frameworku dokáže byť výrazne rýchlejší ako lokálny viacvláknový výpočet a to aj pri vyťažení procesorov počítačov návštevníkov stránky, z dlhodobého pohľadu na menej ako 10%. Experimenty odhalili aj limity implementácie frameworku, ktoré pre nasadenie frameworku v praxi treba odstrániť. Celkovo však experimenty dopadli očakávane a ich výsledky predpovedajú sľubné využitie frameworku.

Nakoniec som navrhol možné vylepšenia, ktoré môžu viesť k väčšiemu výkonu frameworku a rozšírenia, ktoré by z frameworku mohli vytvoriť výpočtovú platformu.

Literatúra

[1] Boldrin, F.; Taddia, C.; Mazzini, G.: Distributed Computing Through Web Browser. In *2007 IEEE 66th Vehicular Technology Conference*, Sept 2007, ISSN 1090-3038, s. 2020–2024, doi:10.1109/VETECF.2007.424.

[2] Chapter 4: Distributed and Parallel Computing. [cit. 30.4.2018]. Dostupné z: `http://wla.berkeley.edu/~cs61a/fa11/lectures/communication.html#distributed-computing`

[3] Merelo, J. J.; García, A. M.; Laredo, J. L. J.; aj.: Browser-based Distributed Evolutionary Computation: Performance and Scaling Behavior. In *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-698-1, s. 2851–2858, doi: 10.1145/1274000.1274083. Dostupné z: `http://doi.acm.org/10.1145/1274000.1274083`

[4] MacWilliam, T.; Cecka, C.: CrowdCL: Web-based volunteer computing with WebCL. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2013, s. 1–6, doi:10.1109/HPEC.2013.6670348.

[5] Miller, D. M.: The Online Community Grid Volunteer Grid Computing with the Web Browser. Georgia Institute of Technology, 2008. Dostupné z: `https://smartech.gatech.edu/handle/1853/33477`

[6] Mersenne Research, Inc.: Great Internet Mersenne Prime Search. [cit. 30.4.2018]. Dostupné z: `https://www.mersenne.org`

[7] University of California : About SETI@home. [cit. 30.4.2018]. Dostupné z: `https://setiathome.berkeley.edu/sah_about.php`

[8] Pande Lab: Folding@home. [cit. 30.4.2018]. Dostupné z: `http://folding.stanford.edu`

[9] Beberg, A. L.; Ensign, D. L.; Jayachandran, G.; aj.: Folding@home: Lessons from eight years of volunteer distributed computing. In *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, ISSN 1530-2075, s. 1–8, doi:10.1109/IPDPS.2009.5160922.

[10] Anderson, D. P.: BOINC: a system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, Nov 2004, ISSN 1550-5510, s. 4–10, doi:10.1109/GRID.2004.14.

[11] University of California: BOINC, Open-source software for volunteer computing. [cit. 30.4.2018]. Dostupné z: `https://boinc.berkeley.edu/`

[12] Mozilla and individual contributors: MDN web docs: JavaScript reference. [cit. 30.4.2018]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference`

[13] Mozilla and individual contributors: MDN web docs: Ajax. [cit. 30.4.2018]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX`

[14] Extensible Markup Language (XML). [cit. 30.4.2018]. Dostupné z: `https://www.w3.org/XML/`

[15] Oracle Corporation: Java. [cit. 30.4.2018]. Dostupné z: `https://java.com/en/download/`

[16] Ruby on Rails. [cit. 30.4.2018]. Dostupné z: `http://rubyonrails.org/`

[17] Klein, J.; Spector, L.: Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-697-4, s. 1628–1635, doi:10.1145/1276958.1277282. Dostupné z: `http://doi.acm.org/10.1145/1276958.1277282`

[18] Spector, L.; Perry, C.; Klein, J.; aj.: Push 3.0 Programming Language Description. [cit. 30.4.2018]. Dostupné z: `http://faculty.hampshire.edu/lspector/push3-description.html`

[19] C++ Language. [cit. 30.4.2018]. Dostupné z: `http://www.cplusplus.com`

[20] PHP Group: PHP. [cit. 30.4.2018]. Dostupné z: `http://php.net/`

[21] Merelo-Guervos, J. J.; Castillo, P. A.; Laredo, J. L. J.; aj.: Asynchronous distributed genetic algorithms with Javascript and JSON. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, June 2008, ISSN 1089-778X, s. 1372–1379, doi:10.1109/CEC.2008.4630973.

[22] Perl. [cit. 30.4.2018]. Dostupné z: https://www.perl.org/

[23] Twitter, Inc.: Twitter. [cit. 30.4.2018]. Dostupné z: https://twitter.com/

[24] Adobe Systems Inc.: Adobe Flash Player. [cit. 30.4.2018]. Dostupné z: https://get.adobe.com/flashplayer/

[25] Adobe Systems Inc.: Learning ActionScript 3. [cit. 30.4.2018]. Dostupné z: https://www.adobe.com/devnet/actionscript/learning.html

[26] Ryza, S.; Wall, T.: MRJS: A JavaScript MapReduce Framework for Web Browsers. 2010. Dostupné z: http://static.cs.brown.edu/courses/csci2950-u/f11/papers/mrjs.pdf

[27] Stuchlík, O.; Bartoň, T.; Tvrdík, P.: MapReduce frameworks (prednáška). 2017, [cit. 30.4.2018]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-PDP.16/_media/lectures/mi-pdplecture12-mapreduce.pdf

[28] Mozilla and individual contributors: MDN web docs: Using Web Workers. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

[29] Duda, J.; Dlubacz, W.: Distributed Evolutionary Computing System Based on Web Browsers with Javascript. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, PARA'12, Berlin, Heidelberg: Springer-Verlag, 2013, ISBN 978-3-642-36802-8, s. 183–191, doi:10.1007/978-3-642-36803-5_13. Dostupné z: http://dx.doi.org/10.1007/978-3-642-36803-5_13

[30] Zorrilla, M.; Martin, A.; Tamayo, I.; aj.: Web Browser-Based Social Distributed Computing Platform Applied to Image Analysis. In *2013 International Conference on Cloud and Green Computing*, Sept 2013, s. 389–396, doi:10.1109/CGC.2013.68.

[31] Turek, W.; Nawarecki, E.; Dobrowolski, G.; aj.: WEB PAGES CONTENT ANALYSIS USING BROWSER-BASED VOLUNTEER COMPUTING. *Computer Science*, ročník 14, č. 2, 2013: str. 215, ISSN 2300-7036. Dostupné z: https://journals.agh.edu.pl/csci/article/view/278

[32] The Khronos™ Group Inc.: OpenCL. [cit. 30.4.2018]. Dostupné z: https://www.khronos.org/opencl/

[33] The Khronos™ Group Inc.: WebCL. [cit. 30.4.2018]. Dostupné z: https://www.khronos.org/webcl/

[34] Cushing, R.; Putra, G. H. H.; Koulouzis, S.; aj.: Distributed Computing on an Ensemble of Browsers. *IEEE Internet Computing*, ročník 17, č. 5, Sept 2013: s. 54–61, ISSN 1089-7801, doi:10.1109/MIC.2013.3.

[35] Mozilla and individual contributors: MDN web docs: XMLHttpRequest. [cit. 30.4.2018]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest`

[36] Mozilla and individual contributors: MDN web docs: DOM. [cit. 30.4.2018]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Glossary/DOM`

[37] Wilkinson, S. R.; Almeida, J. S.: QMachine: commodity supercomputing in web browsers. *BMC Bioinformatics*, ročník 15, č. 1, Jun 2014: str. 176, ISSN 1471-2105, doi:10.1186/1471-2105-15-176. Dostupné z: `https://doi.org/10.1186/1471-2105-15-176`

[38] Meeds, E.; Hendriks, R.; al Faraby, S.; aj.: MLitB: Machine Learning in the Browser. *CoRR*, ročník abs/1412.2432, 2014, `1412.2432`. Dostupné z: `http://arxiv.org/abs/1412.2432`

[39] Mozilla and individual contributors: MDN web docs: WebSockets. [cit. 30.4.2018]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`

[40] Pan, Y.; White, J.; Sun, Y.; aj.: Gray Computing: An Analysis of Computing with Background JavaScript Tasks. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, ročník 1, May 2015, ISSN 0270-5257, s. 167–177, doi:10.1109/ICSE.2015.38.

[41] CDN.net: What is a CDN? [cit. 30.4.2018]. Dostupné z: `https://cdn.net/what-is-a-cdn/`

[42] Liu, C.; White, R. W.; Dumais, S.: Understanding Web Browsing Behaviors Through Weibull Analysis of Dwell Time. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0153-4, s. 379–386, doi:10.1145/1835449.1835513, [cit. 30.4.2018]. Dostupné z: `http://doi.acm.org/10.1145/1835449.1835513`

[43] Emscripten Contributors: Emscripten. [cit. 30.4.2018]. Dostupné z: `https://kripken.github.io/emscripten-site/`

[44] Herman, D.; Wagner, L.; Zakai, A.: asm.js: Working Draft. [cit. 30.4.2018]. Dostupné z: `http://asmjs.org/spec/latest/`

[45] Fabisiak, T.; Danilecki, A.: Browser-based Harnessing of Voluntary Computational Power. ročník 42, 03 2017. Dostupné z: https://www.degruyter.com/downloadpdf/j/fcds.2017.42.issue-1/fcds-2017-0001/fcds-2017-0001.pdf

[46] Fedak, G.; Germain, C.; Neri, V.; aj.: XtremWeb: A generic global computing system. 02 2001.

[47] Piórkowski, A.; Szemla, P.: Client-Side Processing Environment Based on Component Platforms and Web Browsers. In *Computer Networks*, editace A. Kwiecień; P. Gaj; P. Stera, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-38865-1, s. 21–30.

[48] Merelo Guervós, J. J.; García-Sánchez, P.: Modeling browser-based distributed evolutionary computation systems. *CoRR*, ročník abs/1503.06424, 2015, 1503.06424. Dostupné z: http://arxiv.org/abs/1503.06424

[49] Merelo Guervós, J. J.; Valdez, M. G.; Valdivieso, P. Á. C.; aj.: NodIO, a JavaScript framework for volunteer-based evolutionary algorithms : first results. *CoRR*, ročník abs/1601.01607, 2016, 1601.01607. Dostupné z: http://arxiv.org/abs/1601.01607

[50] Hidaka, M.; Miura, K.; Harada, T.: Development of JavaScript-based deep learning platform and application to distributed training. *CoRR*, ročník abs/1702.01846, 2017, 1702.01846. Dostupné z: http://arxiv.org/abs/1702.01846

[51] bryanyee, s., dlaosb: deThread. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/dethread

[52] Gallacher, T.: dis.io. [cit. 30.4.2018]. Dostupné z: https://github.com/tomgco/dis.io

[53] Bergwinkl, T.: BoomerangJS. [cit. 30.4.2018]. Dostupné z: http://www.boomerangjs.org/

[54] mcomghall: comp-pool. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/comp-pool

[55] syzer: js-spark. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/js-spark

[56] yoni: Workhorse. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/workhorse

[57] GitHub Inc.: GitHub. [cit. 30.4.2018]. Dostupné z: https://github.com/

[58] Badges2Go UG: Coinhive: A Crypto Miner for your Website. [cit. 30.4.2018]. Dostupné z: https://coinhive.com/

[59] Monero: Private Digital Currency. [cit. 30.4.2018]. Dostupné z: https://getmonero.org/

[60] WebAssembly. [cit. 30.4.2018]. Dostupné z: http://webassembly.org/

[61] Hay, S.: Web Mining – Monetize Your Website through User Browsers. [cit. 30.4.2018]. Dostupné z: https://99bitcoins.com/webmining-monetize-your-website-through-user-browsers/

[62] CryptoNoter: Great things happen when we work together. [cit. 30.4.2018]. Dostupné z: https://www.cryptonoter.computing

[63] Computes Inc.: Computes: A secure decentralized mesh computing platform designed for businesses & the greater good. [cit. 30.4.2018]. Dostupné z: http://computes.io

[64] Mozilla and individual contributors: MDN web docs: HTTP. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP

[65] Protocol Labs: InterPlanetary File System. [cit. 30.4.2018]. Dostupné z: https://ipfs.io/

[66] Refsnes Data: Browser Statistics. [cit. 30.4.2018]. Dostupné z: https://www.w3schools.com/browsers/default.asp

[67] StatCounter: Browser Market Share Worldwide. [cit. 30.4.2018]. Dostupné z: http://gs.statcounter.com/

[68] Awio Web Services LLC: Web Browser Usage Trends. [cit. 30.4.2018]. Dostupné z: https://www.w3counter.com/trends

[69] Global market share held by leading internet browsers from January 2012 to February 2018. [cit. 30.4.2018]. Dostupné z: https://www.statista.com/statistics/268254/market-share-of-internet-browsers-worldwide-since-2009/

[70] Google LLC: Chrome Web Browser. [cit. 30.4.2018]. Dostupné z: https://www.google.com/chrome/

[71] Mozilla Foundation: Firefox. [cit. 30.4.2018]. Dostupné z: https://www.mozilla.org

[72] Microsoft Corporation: Internet Explorer. [cit. 30.4.2018]. Dostupné z: https://www.microsoft.com/cs-cz/download/internet-explorer.aspx?SearchType=0

[73] Microsoft Corporation: Microsoft Edge. [cit. 30.4.2018]. Dostupné z: https://www.microsoft.com/sk-sk/windows/microsoft-edge

[74] Opera Software : Opera. [cit. 30.4.2018]. Dostupné z: https://www.opera.com

[75] Google LLC: Android. [cit. 30.4.2018]. Dostupné z: https://www.android.com/

[76] Apple Inc.: Safari. [cit. 30.4.2018]. Dostupné z: https://www.apple.com/safari/

[77] Samsung Electronics Co., Ltd.: Samsung Internet Browser. [cit. 30.4.2018]. Dostupné z: https://play.google.com/store/apps/details?id=com.sec.android.app.sbrowser

[78] UCWeb Inc. : UC Browser. [cit. 30.4.2018]. Dostupné z: http://www.ucweb.com/

[79] Average Internet Speeds By Country. [cit. 30.4.2018]. Dostupné z: https://www.fastmetrics.com/internet-connection-speed-by-country.php

[80] Countries with the highest average internet connection speed as of 1st quarter 2017 (in Mbps). [cit. 30.4.2018]. Dostupné z: https://www.statista.com/statistics/204952/average-internet-connection-speed-by-country/

[81] Czaplicki, E.: elm. [cit. 30.4.2018]. Dostupné z: http://elm-lang.org/

[82] Microsoft Corporation: TypeScript. [cit. 30.4.2018]. Dostupné z: https://www.typescriptlang.org/

[83] Hickey, R.: ClojureScript. [cit. 30.4.2018]. Dostupné z: https://clojurescript.org/

[84] Ashkenas, J.: CoffeeScript. [cit. 30.4.2018]. Dostupné z: http://coffeescript.org/

[85] Freeman, P.; Burgess, G.; other contributors: PureScript. [cit. 30.4.2018]. Dostupné z: http://www.purescript.org/

[86] JetBrains s.r.o.: Kotlin. [cit. 30.4.2018]. Dostupné z: https://kotlinlang.org/

[87] Nielsen, J.: How Long Do Users Stay on Web Pages? [cit. 30.4.2018]. Dostupné z: https://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/

[88] Facebook, Inc.: Facebook. [cit. 30.4.2018]. Dostupné z: https://facebook.com/

[89] Stack Exchange Inc.: Stack Overflow. [cit. 30.4.2018]. Dostupné z: https://stackoverflow.com/

[90] N Press s.r.o.: Denník N. [cit. 30.4.2018]. Dostupné z: https://dennikn.sk/

[91] Liedke, L.: 100+ Internet Stats and Facts for 2018. [cit. 30.4.2018]. Dostupné z: https://www.websitehostingrating.com/internet-statistics-facts-2018/

[92] Stevens, J.: Internet Stats & Facts for 2017. [cit. 30.4.2018]. Dostupné z: https://hostingfacts.com/internet-facts-stats-2016/

[93] Software in the Public Interest : Open MPI. Dostupné z: https://www.open-mpi.org/

[94] Google LLC: Google Play. [cit. 30.4.2018]. Dostupné z: https://play.google.com/store

[95] Mozilla and individual contributors: MDN web docs: 403 Not Found. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403

[96] Vaikuntanathan, V.: How to Compute on Encrypted Data. In *Progress in Cryptology - INDOCRYPT 2012*, editace S. Galbraith; M. Nandi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-34931-7, s. 1–15.

[97] Refsnes Data: JavaScript Versions. [cit. 30.4.2018]. Dostupné z: https://www.w3schools.com/js/js_versions.asp

[98] Clark, L.: A crash course in just-in-time (JIT) compilers. [cit. 30.4.2018]. Dostupné z: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

[99] Adobe Systems Inc.: Adobe Acrobat Reader DC. [cit. 30.4.2018]. Dostupné z: https://get.adobe.com/sk/reader/otherversions/

[100] The Apache Software Foundation: CouchDB. [cit. 30.4.2018]. Dostupné z: http://couchdb.apache.org/

[101] JetBrains s.r.o.: WebStorm. [cit. 30.4.2018]. Dostupné z: https://www.jetbrains.com/webstorm/

[102] Microsoft Corporation: Visual Studio Code. [cit. 30.4.2018]. Dostupné z: https://code.visualstudio.com/

[103] Turner, J.: TypeScript. [cit. 30.4.2018]. Dostupné z: https://blogs.msdn.microsoft.com/typescript/2013/06/18/announcing-typescript-0-9/

[104] Mozilla and individual contributors: MDN web docs: Classes. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

[105] Node.js Foundation: Node.js. [cit. 30.4.2018]. Dostupné z: https://nodejs.org

[106] Google LLC: Chrome V8. [cit. 30.4.2018]. Dostupné z: https://developers.google.com/v8/

[107] Refsnes Data: Node.js Introduction. [cit. 30.4.2018]. Dostupné z: https://www.w3schools.com/nodejs/nodejs_intro.asp

[108] The Linux Foundation®: Node.js Foundation. [cit. 30.4.2018]. Dostupné z: https://foundation.nodejs.org/

[109] npm, Inc. and Contributors: npm. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/

[110] npm, Inc. and Contributors: What is npm? [cit. 30.4.2018]. Dostupné z: https://docs.npmjs.com/getting-started/what-is-npm

[111] University of Illinois at Urbana-Champaign: clang: a C language family frontend for LLVM. [cit. 30.4.2018]. Dostupné z: https://clang.llvm.org/

[112] Mozilla and individual contributors: MDN web docs: WebAssembly. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly

[113] Deveria, A.; Schoors, L.: Can I use ___ ? [cit. 30.4.2018]. Dostupné z: https://caniuse.com/

[114] WebAssembly Community Group. [cit. 30.4.2018]. Dostupné z: https://www.w3.org/community/webassembly/

[115] WebAssembly Working Group. [cit. 30.4.2018]. Dostupné z: https://www.w3.org/wasm/

[116] Google I/O 2017. [cit. 30.4.2018]. Dostupné z: https://events.google.com/io2017/

[117] OpenCV team: Open Source Computer Vision Library. [cit. 30.4.2018]. Dostupné z: https://opencv.org/

[118] asm.js - frequently asked questions. [cit. 30.4.2018]. Dostupné z: http://asmjs.org/faq.html

[119] Mozilla and individual contributors: MDN web docs: Compiling a New C/C++ Module to WebAssembly. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm

[120] Emscripten: Better JS size for small programs #5794. [cit. 30.4.2018]. Dostupné z: https://github.com/kripken/emscripten/issues/5794

[121] HTML5 Web Workers. [cit. 30.4.2018]. Dostupné z: https://www.w3schools.com/html/html5_webworkers.asp

[122] MongoDB Inc. : MongoDB. [cit. 30.4.2018]. Dostupné z: https://www.mongodb.com/

[123] Holubová, I.; Svoboda, M.: Document Databases, JSON, MongoDB (prednáška). [cit. 30.4.2018]. Dostupné z: http://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/lectures/Lecture-13-Document-Databases-JSON-MongoDB.pdf

[124] BSON. [cit. 30.4.2018]. Dostupné z: http://bsonspec.org/

[125] Introducing JSON. [cit. 30.4.2018]. Dostupné z: http://json.org/

[126] Grigorik, I.; Surma: Introduction to HTTP/2. [cit. 30.4.2018]. Dostupné z: https://developers.google.com/web/fundamentals/performance/http2/

[127] Group, T. I. H. W.: HTTP/2. [cit. 30.4.2018]. Dostupné z: https://http2.github.io/

[128] Dias, D.; Indutny, F.; rauchg: SPDY Server for node.js. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/spdy

[129] Mozilla and individual contributors: MDN web docs: Promise. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[130] Node.js Foundation: Zlib. [cit. 30.4.2018]. Dostupné z: https://nodejs.org/api/zlib.html

[131] Puzrin, V.: pako. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/pako

[132] Šimek, P.: VM2. [cit. 30.4.2018]. Dostupné z: https://www.npmjs.com/package/vm2

[133] Barré, G.: Executing untrusted JavaScript code in a browser. [cit. 30.4.2018]. Dostupné z: https://www.softfluent.com/blog/dev/Executing-untrusted-JavaScript-code-in-a-browser

[134] Mozilla and individual contributors: MDN web docs: Cache-Control. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

[135] Mozilla and individual contributors: MDN web docs: Last-Modified. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Last-Modified

[136] Mozilla and individual contributors: MDN web docs: Etag. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag

[137] Mozilla and individual contributors: MDN web docs: 404 Forbidden. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404

[138] Mozilla and individual contributors: MDN web docs: 204 No Content. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204

[139] Mozilla and individual contributors: MDN web docs: 400 Bad Request. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400

[140] Mozilla and individual contributors: MDN web docs: 500 Internal Server Error. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500

[141] Mozilla and individual contributors: MDN web docs: 201 Created. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201

[142] Mozilla and individual contributors: MDN web docs: 409 Conflict. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409

[143] Mozilla and individual contributors: MDN web docs: 401 Unauthorized. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401

[144] Mozilla and individual contributors: MDN web docs: 200 OK. [cit. 30.4.2018]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200

[145] OpenMP ARB: OpenMP. [cit. 30.4.2018]. Dostupné z: http://www.openmp.org/

Dodatok A

Príklad použitia frameworku

V tejto prílohe uvádzam príklad implementácie funkcií prototypu úlohy pre výpočet determinantu matice naivným algoritmom. Následne uvádzam ich použitie pri vytvorení a odoslaní prototypu úlohy na server. Nakoniec táto príloha obsahuje ukážku použitia frameworku pre výpočet úlohy.

```
1  //kód pre jsWorkingFunction (JavaScript)
2  let jsWorkingFunction = function(data){
3    let sign = function(p){
4    return p%2 == 0 ? 1 : -1;
5    }
6
7    let determ = function(matrix){
8      let a = matrix;
9      let n = matrix.length;
10     let det = 0,
11       p, subi, subj, i, j;
12     let temp;
13     if (n == 1) {
14       return a[0][0];
15     } else if (n === 2) {
16       det = (a[0][0] * a[1][1] - a[0][1] * a[1][0]);
17       return det;
18     } else {
19       for (p = 0; p < n; p++) {
20         subi = 0;
21         subj = 0;
22         temp = [];
23         for (i = 1; i < n; i++) { //rozvoj podla 0-teho riadku
24           temp.push([]);
25           for (j = 0; j < n; j++) {
26             if (j === p) {
27               continue;
28             }
29             temp[subi].push(a[i][j]);
30             subj++;
31             if (subj === n - 1) {
32               subi++;
```

```
33                subj = 0;
34              }
35            }
36          }
37          det = det + a[0][p] * sign(p) * determ(temp, n - 1);
38        }
39        return det;
40      }
41    }
42
43    if( Array.isArray(data) &&
44      data.length > 0 &&
45      Array.isArray(data[0])
46    ){
47      return determ(data);
48    }
49
50    throw "Wrong data";
51  }
```

Kód A.1: Ukážka implementácie funkcie *jsWorkingFunction*, pomocou ktorej sa môže vykonávať výpočet práce u klienta.

```
1  let cplusplusWorkingFunctionString = `
2  //kód pre cplusplusWorkingFunction (C++)
3  #include <emscripten.h>
4
5  extern "C"
6  {
7    EMSCRIPTEN_KEEPALIVE
8    float ** getEmptyFloatMatrix(int rows, int cols){
9      float ** matrix = new float*[rows];
10     for(int i = 0; i< rows; i++){
11     matrix[i] = new float[cols];
12     }
13     return matrix;
14   }
15
16
17   EMSCRIPTEN_KEEPALIVE
18   int sign(int p){
19     return p%2 == 0 ? 1 : -1;
20   }
21
22   EMSCRIPTEN_KEEPALIVE
23   float ** getEmptyFloatMatrix(int rows, int cols){
24     float ** matrix = new float*[rows];
25     for(int i = 0; i< rows; i++){
26       matrix[i] = new float[cols];
27     }
28     return matrix;
29   }
30
31   EMSCRIPTEN_KEEPALIVE
```

```
32   float determ(float **a,int n) {
33     float det=0;
34     int p, subi, subj, i, j;
35     float **temp;
36     if(n==1) {
37       return a[0][0];
38     } else if(n==2) {
39       return a[0][0]*a[1][1]-a[0][1]*a[1][0];
40     } else {
41       temp = getEmptyFloatMatrix(n-1,n-1);
42       for(p=0;p<n;p++) {
43         subi = 0;
44         subj = 0;
45         for(i=1;i<n;i++) { // rozvoj podla 0-teho riadku
46           for( j=0;j<n;j++) {
47             if(j==p) {
48               continue;
49             }
50             temp[subi][subj] = a[i][j];
51             subj++;
52             if(subj==n-1) {
53               subi++;
54               subj = 0;
55             }
56           }
57         }
58         det=det+a[0][p]*sign(p)*determ(temp,n-1);
59       }
60       for(i = 0; i < n-1 ;i++){
61         delete [] temp[i];
62       }
63       delete[] temp;
64       return det;
65     }
66   }
67
68   EMSCRIPTEN_KEEPALIVE
69   float determFlat(float* flat, int n){
70     float **a = new float*[n];
71     for( int i = 0; i < n; i++){
72       a[i] = new float[n];
73       for(int j = 0; j < n; j++){
74         a[i][j]=flat[i*n+j];
75       }
76     }
77     float resultT = determ(a,n);
78     return resultT;
79   }
80 }';
```

Kód A.2: Ukážka implementácie funkcie v jazyku C++, ktorá sa skompiluje do asm.js a wasm modulov. Pomocou nich sa potom môže vykonávať výpočet práce u klienta. Funckia v jazyku C++ je v ukážke vložená ako textový reťazec do premennej v skripte v jazyku JavaScript.

```
//kód pre asmJSWorkingFunction (JavaScript)
let asmJSWorkingFunction = function(data){
  if(!Array.isArray(data) || !data.length >0 || !data[0].length >
      0 || data.length !== data[0].length){
    throw "wrong data";
  }

  let n = data.length;

  let flat = [];
  for(let i = 0; i< n; i++){
    flat = flat.concat(data[i]);
  }

  // vytvorenie typovaného poľa
  let flatTypedArray = new Float32Array(flat);
  //skopírovanie poľa do haldy asm.js/wasm modulu
  let heapBytes = lib.arrayToHeap(module,flatTypedArray);

  // pomocné priradenie
  module["_determFlat"] = module["asm"]["_determFlat"];
  let result = module.ccall(  //zavolanie funkcie modulu
   'determFlat',  // názov funkcie
   'number', // typ návratovej hodnoty
   ['number','number'], // typy argumentov
   [heapBytes.byteOffset, n ] // argumenty
  );

  return result;

}
```

Kód A.3: Ukážka implementácie funkcie *asmJSWorkingFunction*, ktorá slúži ako rozhranie medzi asm.js a wasm modulmi a frameworkom pri výpočte práce u klienta

```
// kód pre deliacu funkciu (JavaScript)
let dividingFunction = function ( d ) {
  let matrix = d.data;
  let n = matrix.length;
  if(n <= 11)
    return null;
  let subi,subj;
  let subData = [];
  let temp;
  for(let p = 0; p < n; p++ ) {
    subi = 0;
    subj = 0;
    temp = [] ;
    for(let i = 1; i < n; i++) {//rozvoj podla 0teho riadku
      temp.push([]);
      for(let j = 0; j < n; j++ ) {
        if( j === p ) {
          continue;
```

```
        }
        temp[subi].push(matrix[i][j]);
        subj++;
        if(subj === n-1) {
          subi ++;
          subj = 0;
        }
      }
    }
    subData.push({
      data: temp,
      serverSideMetadata: {
        a0p: matrix[0][p]
      }
    });
  }
  return subData;
}
```

Kód A.4: Ukážka implementácie deliacej funkcie, pomocou ktorej framework automaticky rozdelí veľké dáta úlohy a vytvorí tak čiastkové práce

```
// kód pre zlučovaciu funkciu (JavaScript)
let mergingFunction = function (resultArr){
  let sign = function(p){
    return p%2 == 0 ? 1 : -1;
  }
  let n = resultArr.length;
  let a0p;
  let det = 0;
  for( let p=0; p < n; p++ ) {
    a0p = resultArr[p].serverSideMetadata.a0p;
    det = det + a0p * sign(p) * resultArr[p].result;
  }
  return det;
}
```

Kód A.5: Ukážka implementácie zlučovacej funkcie, pomocou ktorej framework automaticky zlúči výsledky čiastkových prác do výsledku úlohy

```
let taskPrototype = new TaskPrototype();
taskPrototype.workingFunctions = new WorkingFunctions();
taskPrototype.workingFunctions
  .jsWorkingFunction = jsWorkingFunction.toString();
taskPrototype.workingFunctions
  .asmJSWorkingFunction = asmJSWorkingFunction.toString();
taskPrototype
  .cplusplusWorkingFunction = cplusplusWorkingFunctionString;
taskPrototype.emscriptenFlags =  {
  Owasm:'3', // úroveň optimalizácie pri kompilácií do wasm modulu
  Oasm:'s', // úroveň optimalizácie pri kompilácií do asm.js modulu
  exportedFunctions:  // funkcie, ktoré má Emscripten exportovať
  [
    'ccall',
```

```
    'allocate'
  ]
};
taskPrototype.mergingFunction = mergingFunction.toString();
taskPrototype.dividingFunction = dividingFunction.toString();
taskPrototype.suggestedMaxDataSizeInBytes = 1;
taskPrototype.id = "determTaskPrototype_n_11"
```

Kód A.6: Ukážka vytvorenia prototypu úlohy s využitím vyššie uvedených funkcií.

```
let client = new WebMachineClient();
client.postTaskPrototypeAsync(taskPrototype)
  .then( async ()=>{
    let data = [
        [0,2,1,2,1],
        [1,0,1,2,1],
        [1,2,0,2,1],
        [1,2,1,0,1],
        [1,2,1,2,0]
    ];

    let result = await client.computeAsync (
      taskPrototype.id,
      data,
      "/home/jakub/results",
      "determ.txt"
    );
    console.log(result);
  })
```

Kód A.7: Ukážka odoslania uvedeného prototypu úlohy a jeho využitia pre výpočet úlohy

Dodatok **B**

Zoznam použitých skratiek

AJAX Asynchronous JavaScript + XML

API Application programming interface

CDN content delivery network

DOM Document Object Model

HTTP Hypertext Transfer Protocol

JIT Just-in-time

JSON JavaScript Object Notation

URL Uniform Resource Locator

XML Extensible markup language

XHR XML HttpRequest

Dodatok C

Obsah priloženého média

- `readme.txt` ..... stručný popis obsahu média
- `code` ..... adresár so zdrojovými kódmi
  - `server` ..... zdrojové kódy pre server
  - `client` ..... zdrojové kódy výpočtového klienta
  - `client-programmer` ..... zdrojové kódy knižnice pre užívateľa frameworku
  - `fooWebsiteServer` ..... zdrojové kódy testovacej webovej stránky
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF
  - `thesis.tex` ..... zdrojová forma práce vo formáte LaTeX