Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Optimalizácia architektúry distribuovaného systému pre výpočty simulácii v heliosfére

Diplomová práca

2024 **Bc. Daniel Slanina**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Optimalizácia architektúry distribuovaného systému pre výpočty simulácii v heliosfére

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Michal Solanik, PhD.
Konzultant:	doc. Ing. Ján Genči, PhD.

Košice 2024 **Bc. Daniel Slanina**

Abstrakt v SJ

V predchádzajúcej bakalárskej práci bol navrhnutý a implementovaný prototyp distribuovaného systému pre výpočty simulácii v heliosfére. Kvôli novým požiadavkám na systém a postupnej evolúcii jednotlivých podsystémov, ktoré tento systém využíva, sa táto práca venuje optimalizácii architektúry spomínaného distribuovaného systému. Na základe analýzy existujúcich systémov z podobnej oblasti, ktoré sú dlhoročne nasadené v produkčnom prostredí, je vyhotovený návrh optimalizovanej architektúry pre predmetný distribuovaný systém. Pomocou nového návrhu je implementovaná nová verzia distribuovaného systému, ktorá zahŕňa najmä pridanie podpory ďalších typov distribuovaných úloh, podporu vlastných simulačných modelov v systéme, zlepšenie bezpečnosti a odolnosti systému, aktualizáciu používateľského rozhrania a potenciálne zrýchlenie systému na úrovni výpočtov simulácií. V práci je navrhnutý aj mechanizmus automatizovaného nasadenia centrálnej vrstvy tohto systému formou automatizovaného generovania manifestov nasadenia pre platformu Kubernetes. Na základe verifikácie distribuovaného systému princípom testovania v simulovanom produkčnom prostredí bolo dokázané, že navrhnuté a implementované mechanizmy zlepšili celkovú bezpečnosť a integritu systému, odolnosť systému voči zlyhaniam a zotavenie z chýb a zrýchlili celý distribuovaný systém z hľadiska rýchlosti výpočtu simulácií. Výpočet rovnakej simulácie je na optimalizovanom distribuovanom systéme rýchlejší o 7.2% v porovnaní s prototypom z bakalárskej práce, respektíve o 25.3% v porovnaní s úplne pôvodným prototypom distribuovaného systému.

Kľúčové slová v SJ

distribuovaný systém, distribuované plánovanie úloh, heliosféra, plánovanie riadené dopytom

Abstrakt v AJ

In a previous bachelor thesis, a prototype distributed system for Distributed System for Cosmic Ray Simulations in the Heliosphere was designed and implemented. Due to the newly defined requirements for the system and the continuous evolution of various subsystems used by this system, this thesis is devoted to the optimization of the architecture of the above-mentioned distributed system. Based on the analysis of existing systems from a similar field of domain that have been used in production for years, an optimized architecture for the distributed system is proposed. Following the new design, a new version of the

distributed system is implemented, which includes new support for additional types of distributed tasks, support for custom simulation models in the system, improvements in areas of security and resilience of the system, an updated user interface with new functionalities and a potential speed-up of the system at the level of simulation computations. The thesis also proposes a mechanism for automated deployment of the central layer of this system in the form of automated generation of deployment manifests for the Kubernetes platform. Based on the verification of the distributed system by testing in a simulated production environment, it has been proved that the proposed and implemented mechanisms have improved the overall security and integrity of the system, the resilience of the system to failures and recovery from errors, and have accelerated the entire distributed system in terms of the speed of computation of simulations. The computation of the same simulation on the optimized distributed system is faster by 7.2%, when compared to the prototype from the bachelor's thesis, and by 25.3%, when compared to the very first prototype of this distributed system.

Kľúčové slová v AJ

demand-driven scheduling, distributed system, distributed task scheduling, heliosphere

Bibliografická citácia

SLANINA, Daniel. *Optimalizácia architektúry distribuovaného systému pre výpočty simulácii v heliosfére*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2024. 73s. Vedúci práce: Ing. Michal Solanik, PhD.

71151

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

Optimalizácia architektúry distribuovaného systému pre výpočty simulácii v heliosfére

Optimization of Distributed System Architecture for Simulations in the Heliosphere

Študent: **Bc. Daniel Slanina**
Školiteľ: **Ing. Michal Solaník, PhD.**
Školiace pracovisko:
Konzultant práce: **doc. Ing. Ján Genči, PhD.**
Pracovisko konzultanta: **Katedra počítačov a informatiky**

Pokyny na vypracovanie diplomovej práce:
1. Analyzovať architektúru distribuovaného systému pre heliosferické výpočty s ohľadom na ďalšie existujúce systémy.
2. Navrhnúť novú optimalizovanú architektúru s ohľadom na budúci rozvoj distribuovaného systému pre heliosferické výpočty.
3. Implementovať navrhnuté riešenie.
4. Otestovať implementované riešenie s ohľadom na vykonávaciu rýchlosť jednotlivých modelov a odolnosti systému voči chybám.
5. Vypracovať dokumentáciu podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 19.04.2024
Dátum zadania diplomovej práce: 31.10.2023

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -11-

v.z.

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 19.4.2024 .....

Vlastnoručný podpis

Poďakovanie

Rád by som sa poďakoval svojmu vedúcemu práce Ing. Michalovi Solanikovi, PhD., za celkovú spoluprácu a jeho čas. Jeho odborné skúsenosti v predmetnej oblasti dopomohli v značnej miere vypracovať túto prácu.

Rovnako by som sa rád poďakoval svojmu konzultantovi práce doc. Ing. Jánovi Genčimu, PhD., za jeho cenné rady k písaniu a úprave práce.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Simulácie v heliosfére, podobne ako iné vedecké simulácie, vyžadujú obrovské množstvo výpočtov. Pre automatizáciu a zrýchlenie samotných výpočtov simulácií sa používajú moderné výpočtové systémy, najmä vďaka neustálemu pokroku v oblasti zrýchľovania výpočtových systémov. Kvôli náročnosti a veľmi dlhému vykonávaciemu času výpočtov simulácií, je pre urýchlenie celkového času výpočtu vhodné použiť mechanizmy **paralelizácie**.

Škálovanie a paralelizácia na úrovni jedného počítačového systému je však obmedzená hardvérom, resp. architektúrou. Pre akékoľvek ďalšie škálovanie systému je potrebné jednotlivé výpočty distribuovať medzi nezávislé výpočtové stanice, čím sa z pôvodného systému, ako opisujú autori v [1] a [2], stáva **distribuovaný systém**. Tento postup bol zvolený aj pre predmetný systém pre výpočty simulácii v heliosfére. Jeho pôvodným autorom je Michal Solanik, ktorý vo svojej práci [3] navrhol a implementoval prvý prototyp. Tento prototyp docielil paralelné spracovanie na úrovni niekoľkých nezávislých pracovných staníc. Nakoľko sa jednalo o prvotný prototyp, jeho implementácia bola vo viacerých oblastiach nedostatočná na produkčné použitie. Hlavnými nedostatkami systému boli použitie *hrubozrnného* distribučného mechanizmu a zlá škálovateľnosť pre rôzne simulačné modely.

Problémy prvého prototypu systému boli adresované v bakalárskej práci [4]. Hlavnou témou práce bola úprava pôvodného prototypu, hlavne úprava algoritmu distribúcie na **jemnozrnný**. Výsledná úprava distribučného mechanizmu priniesla zlepšenia v rýchlosti, odolnosti voči zlyhaniam a celkovej flexibilite systému. Testovaním nového prototypu[4] sa potvrdila odolnosť systému voči chybám, ktoré môžu nastať v bežnej prevádzke. Taktiež sa testovaním ukázalo, že implementácia *jemnozrnného distribučného* algoritmu zrýchlila systém o **21.3%** oproti pôvodnému, za použitia rovnakej testovacej sady ako pri pôvodnom prototype [3].

Aj napriek implementovaným vylepšeniam, prototyp stále nie je úplne pripravený na produkčné použitie. Niektoré zásadné časti systému boli zjednodušené,

resp. úplne vynechané pre účely prototypovania. Pre uvedenie systému do reálneho produkčného prostredia je najprv potrebné dokončiť všetky detaily, ktoré boli vynechané v prototypoch. Systém pritom musí poskytovať existujúce funkcionality bez zmeny.

Okrem optimalizácie existujúceho systému táto práca obsahuje aj návrh a implementáciu nových požiadaviek, ktoré vznikli od času písania bakalárskej práce [4]. Jednou takouto je **podpora vlastných simulačných modelov**, definovaných priamo používateľmi systému. Pre splnenie tejto požiadavky, Solanik navrhol a implementoval [5] program *AutoSphere*, ktorý umožňuje na základe jednoduchého zdrojového kódu generovať optimalizovanú implementáciu simulačného modelu. Implementácia modelu je následne používaná programom Geliosphere, ktorý sa v systéme používa na samotné výpočty. Spomínaný zdrojový kód je písaný v doménovo-špecifickom jazyku, vytvoreného špecificky pre tento účel[5]. Aktuálna verzia prototypu však dokáže pracovať len so základnými simulačnými modelmi, ktoré sú k dispozícii v programe Geliosphere. Táto požiadavka je kľúčová pre budúci rozvoj a použiteľnosť celého systému, a preto je nutné ju implementovať v rámci distribuovaného systému.

Formulácia úlohy

Hlavným cieľom práce je celková optimalizácia distribuovaného systému pre výpočty simulácii v heliosfére a jeho príprava pre produkčné nasadenie a používanie. Nakoľko sa jedná o rozsiahly systém, optimalizácia bude prebiehať na viacerých úrovniach a komponentoch systému.

Ciele práce sa dajú rozdeliť do nasledujúcich krokov:

- **Analýza aktuálneho stavu distribuovaného systému pre výpočty simulácií v heliosfére.**

  Analýzou je priblížený aktuálny stav distribuovaného systému vytvoreného v bakalárskej práci[4] a preukážu sa jeho nedostatky a príznaky pre optimalizáciu. Porovnávané budú pritom aj iné existujúce systémy, ktoré využívajú podobné mechanizmy distribúcie. Dôraz sa bude klásť hlavne na systémy nasadené v produkčných prostrediach s veľmi vysokou úrovňou prevádzky.

- **Špecifikácia nových požiadaviek systému, ktoré zabezpečia budúci rozvoj distribuovaného systému.**

  Na základe analýzy momentálneho stavu distribuovaného systému a súvisiacich podsystémov sa špecifikujú nové požiadavky, ktoré je nutné zapra-

covať do existujúceho systému. Tieto požiadavky budú aplikované v návrhu a implementácii na základe relevantných vzorov a zdrojov.

- **Návrh a implementácia optimalizovaného distribuovaného systému.**

  Na základe analýzy a definovanej špecifikácie požiadaviek sa navrhne optimalizovaná architektúra systému. Návrh sa bude opierať o poznatky získané pri analyzovaní iných, podobných systémov. Tento návrh je potrebné implementovať, a teda optimalizovať existujúci distribuovaný systém.

- **Aktualizácia používateľského rozhrania systému.**

  Pôvodné používateľské rozhranie je neaktuálne a nepodporuje nové funkcionality systému. Po optimalizovaní interných komponentov distribuovaného systému je nutné taktiež aktualizovať, resp. implementovať nové používateľské rozhranie, ktoré bude reflektovať nový stav distribuovaného systému. Implementované používateľské rozhranie však kvôli náročnosti práce bude slúžiť len ako *dočasné*, určené hlavne pre testovacie účely.

- **Príprava a nasadenie distribuovaného systému do produkčného prostredia.**

  Pre optimalizovaný systém je nutné pripraviť automatizované mechanizmy pre nasadenie do produkčného prostredia. Táto požiadavka sa vzťahuje ako aj pre centrálnu vrstvu systému, tak aj pre jeho distribuované časti. Následne sa systém nasadí na určené, pripravené prostredie.

- **Verifikácia distribuovaného systému.**

  Optimalizovaný systém je potrebné otestovať v reálnej záťaži v prostredí, ktoré je podobné produkčnému. Systém sa nasadí do predpripraveného testovacieho prostredia a vykoná sa viacero testov. Najprv sa použije rovnaká testovacia sada ako v bakalárskej práci [4], čím sa dokáže korektnosť systému po jeho optimalizácii. Napokon sa otestujú nové funkcionality, ktoré sú implementované na základe špecifikácie požiadaviek.

1 Distribuovaný systém pre výpočty simulácií v heliosfére

V úplne pôvodnom prototype distribuovaného systému z [3] sa osvedčil koncept distribuovaných výpočtov simulácií, najmä kvôli zrýchleniu celkového výpočtu simulácie. Na tento prototyp nadväzovala bakalárska práca [4], ktorej hlavnou témou bola úprava distribučného modelu jednotlivých výpočtov simulácií na **jemnozrnný**. Prechod na jemnozrnnú distribúciu zrýchlil celkový čas výpočtu jednej simulácie o **21.356%**. Okrem toho sa systém dokázal zotaviť z neplánovaných zlyhaní jednotlivých distribuovaných staníc a flexibilne na ne reagoval. Prínosy bakalárskej práce sú zhrnuté v tabuľke 1.1, ktorá porovnáva prototyp z bakalárskej práce a pôvodný prototyp.

Je teda možné konštatovať, že vytvorený prototyp v bakalárskej práci priniesol určité výhody systému a pripravil priestor pre jeho budúci rozvoj. Pre nasadenie do produkčného prostredia a reálne používanie je však prototyp stále nedostatočný. Pre účely tohto prototypu boli vynechané, resp. zjednodušené niektoré aspekty systému, ktoré sú v plnej verzii systému nevyhnutné. Jedná sa napríklad o používateľské rozhranie, niektoré bezpečnostné mechanizmy, a podobne. Okrem toho pribudli do systému nové požiadavky, ktoré najprv vyžadujú optimalizáciu samotnej architektúry systému, a neskôr aj rozšírenie podpory v systéme pre tieto požiadavky. V nasledujúcich podkapitolách sú identifikované jednotlivé nedostatky a príznaky na optimalizáciu, resp. zmenu v systéme.

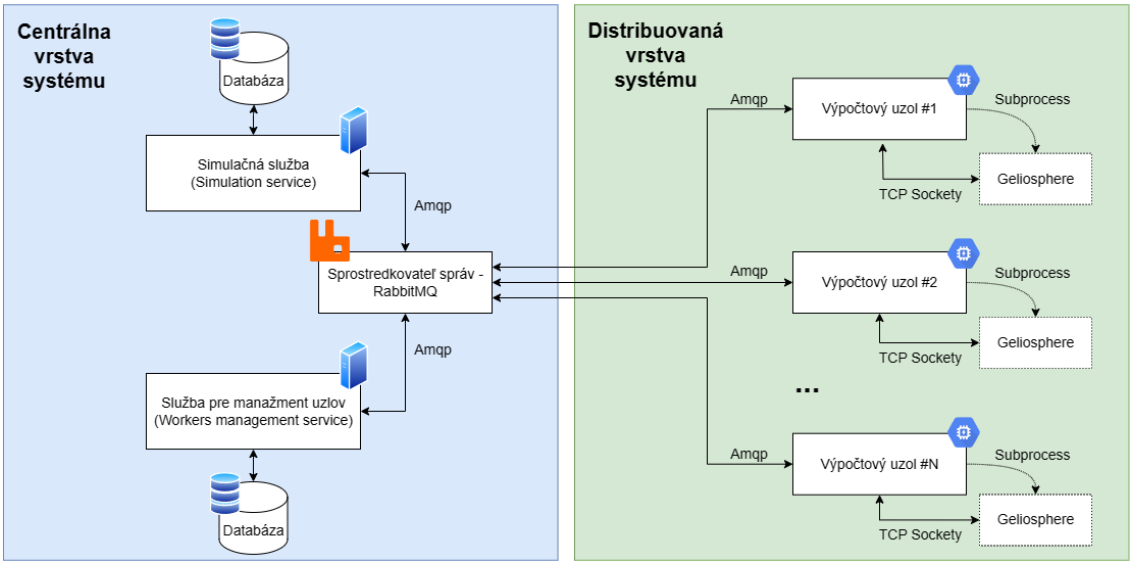
1.1 Architektúra prototypu systému

Ako je spomínané v tabuľke 1.1, systémová architektúra sa skladá z viacerých mikroslužieb. Každá mikroslužba má na starosti špecifickú oblasť domény a počas behu systému komunikuje s ostatnými komponentami systému. Komunikácia medzi jednotlivými službami prebieha výlučne vymieňaním správ. Komunikačnú vrstvu v prototype zastrešuje služba *RabbitMQ*, ktorá funguje ako systém

	Pôvodný prototyp systému	**Prototyp systému z bakalárskej práce**
Distribučný model	Hrubozrnný	Jemnozrnný
Architektúra	Monolit	Mikroslužby
Plánovanie (Scheduling)	Úlohy sú fixne priradené pri vytvorení simulácie	Plánovanie je riadené dopytom, úlohy sú priradené dynamicky
Výpadok distribuovanej stanice	Výpočet zostáva priradený stanici. Po uplynutí určenej doby je daná časť simulácie označená ako *zlyhaná*	Scheduler automaticky preplánuje výpočet na inú stanicu. Simulácia tým pádom **nie je** označená ako *zlyhaná*
Hardvérová podpora distribuovaných staníc	Len GPU	Aj GPU aj CPU
Trvanie výpočtu simulácie pri testovaní	1 hodina, 10 minút a 12 sekúnd	56 minút a 32 sekúnd

Tabuľka 1.1: Porovnanie pôvodného prototypu distribuovaného systému pre výpočty simulácií v heliosfére a prototypu z bakalárskej práce

pre manažovanie správ (angl. message broker) [6] [7]. Celkový prehľad systémovej architektúry je znázornený na obrázku 1.1.



Obr. 1.1: Prehľad architektúry prototypu systému z bakalárskej práce

Nakoľko bakalárska práca sa nezaoberala používateľským rozhraním, je z prehľadu vynechané. Interakcia s prototypom systému je realizovaná pomocou REST API jednotlivých služieb. Pre použiteľnosť systému v praxi je nutné implementovať aj používateľské rozhranie. Tým sa zabezpečí jednoduchá ovládateľnosť a prehľad stavu systému. Pôvodný prototyp obsahuje jednoduché používateľské rozhranie[3] v podobe webovej aplikácie, avšak toto rozhranie nie je aktuálne, obsahuje veľmi obmedzené množstvo funkcionalít a nie je kompatibilné s novým prototypom.

1.1.1 Komunikácia medzi centrálnou a distribuovanou vrstvou

Distribuované stanice komunikujú s centrálnou vrstvou, ktorá riadi distribúciu jednotlivých výpočtov. Z prehľadu architektúry v obrázku 1.1 vyplýva, že jednotlivé distribuované stanice komunikujú s centrálnou vrstvou pomocou sprostredkovateľa správ - RabbitMQ [6]. Výhodou takejto komunikácie je schopnosť distribuovaných staníc takmer okamžite reagovať na zmeny v systéme, vďaka architektúre doručovania v reálnom čase (angl. push-based) [8], ktorú RabbitMQ využíva. V prototype je použitý rovnaký sprostredkovateľ správ aj na internú komunikáciu medzi jednotlivými službami centrálnej vrstvy.

RabbitMQ však nie je ideálnym kandidátom na komunikáciu medzi centralizovanou a distribuovanou vrstvou. Viaceré zdroje [9] [10] [11] uvádzajú, že RabbitMQ, podobne, ako aj iní sprostredkovatelia správ, kladie dôraz skôr na spoľahlivosť a flexibilitu asynchrónnej komunikácie ako na jej rýchlosť. Taktiež uvádzajú, že existujú iné, vhodnejšie protokoly a rámce na takýto typ komunikácie [10] [11], medzi ktoré patrí napríklad **gRPC**. V prototype je taktiež inštancia RabbitMQ zabezpečená len jednoducho, mechanizmom meno + heslo. Tieto údaje musia byť distribuované na všetky používané distribuované stanice, čo prináša **značné bezpečnostné riziko**. Nakoľko distribuované stanice nemusia byť dôveryhodné, je nepriaznivé im zdieľať prístupové údaje do komunikačnej služby. Taktiež po získaní prístupu do služby RabbitMQ, má klient prístup ku všetkým správam v systéme, ktoré dokáže odpočúvať, resp. v horšom prípade aj meniť. Pre správne zabezpečenie by bolo nutné minimálne nastaviť správne autorizačné mechanizmy, napríklad RBAC, ideálne pre každého klienta (distribuovanú stanicu) zvlášť.

Na základe vyššie spomenutých nedostatkov je komunikácia medzi centralizovanou a distribuovanou vrstvou systému prvou a zároveň kľúčovou témou na optimalizáciu. Pre správne a **bezpečné** fungovanie systému je potrebné ďalej analyzovať a navrhnúť riešenie, ktoré spĺňa kritéria spomenuté v tejto sekcii. V na-

sledujúcich kapitolách tejto práce budú analyzované komunikačné mechanizmy podobných distribuovaných systémov, ktoré pomôžu pri návrhu a implementácii nového, optimálneho komunikačného mechanizmu pre tento systém.

1.2 Nové požiadavky systému

Od poslednej iterácie vývoja distribuovaného systému v bakalárskej práci pribudli nové požiadavky, ktoré je potrebné do systému zapracovať. Pre optimalizovanú verziu systému taktiež platia všetky existujúce požiadavky, ktoré boli popísané v bakalárskej práci[4]. Pri testovaní systému treba dbať dôraz nielen na nové požiadavky, ale aj na existujúce, čím sa overí jeho korektnosť.

Nové požiadavky vznikli z viacerých dôvodov. Jedným z nich je postupná evolúcia podsystémov, z ktorých sa tento distribuovaný systém skladá. Príkladom takéhoto podsystému je aplikácia Geliosphere. Systém sa musí adaptovať a prispôsobiť novým verziám použitých podsystémov. Druhým dôvodom je zlepšenie celkovej kontroly nad systémom. Pre účely prototypu boli niektoré kontrolné funkcionality obmedzené, resp. vynechané. V neposlednej rade je dôvodom pre vznik niektorých nových požiadaviek zlepšenie **bezpečnosti** systému.

Pre nasledujúcu iteráciu distribuovaného systému preto boli definované nasledujúce požiadavky:

- **Podpora vlastných simulačných modelov.**
- **Autentifikácia a autorizácia jednotlivých distribuovaných výpočtových staníc.**
- **Administrácia a kontrola distribuovaných výpočtov.**
- **Aktualizácia používateľského rozhrania.**
- **Integrácia s novou verziou programu Geliosphere.**

Každá z vyššie vymenovaných požiadaviek je špecifikovaná v samostatnej pod-sekcii. Tieto požiadavky sú adresované aj v [5], kde sú detailne rozanalyzované a pre niektoré sú aj navrhnuté možné teoretické riešenia.

1.2.1 Podpora vlastných simulačných modelov

Podpora vlastných simulačných modelov je jednou z kľúčových tém tejto práce. Fyzikálne modely pre výpočty rôznych simulácií nie sú fixné, ale časom vznikajú, zanikajú, rozvíjajú sa. V praxi to znamená, že systém Geliosphere sa musí upraviť

vždy, keď vyjde nová verzia fyzikálneho modelu. Táto úprava je ale **časovo a finančne náročná**, nakoľko si vyžaduje zásah programátora. Modely sa však môžu meniť aj denne, na základe rôznych vedeckých výskumov a objavov, preto je pre budúci rozvoj distribuovaného systému takéto fungovanie **nereálne**.

Pre tento problém Solanik navrhol a implementoval prototyp systému *Autosphere*[5], ktorý dokáže generovať optimalizovaný kód na výpočet simulácie zo zadaného vstupu od používateľa. Očakávaný vstup od používateľa je zdrojový kód v jednoduchom, *doménovo-špecifickom jazyku*, ktorý je navrhnutý tak, aby ho dokázali pripraviť aj používatelia bez skúseností s paralelným programovaním, resp. s paralelnými rámcami, prípadne programovaním ako takým (nakoľko cieľovou skupinou tohto systému sú poväčšine vedeckí pracovníci rôznych fyzikálnych ústavov). Výsledný optimalizovaný kód je kompatibilný so systémom Geliosphere, do ktorého bola pridaná podpora dynamického načítavania vygenerovaných knižníc s daným modelom.

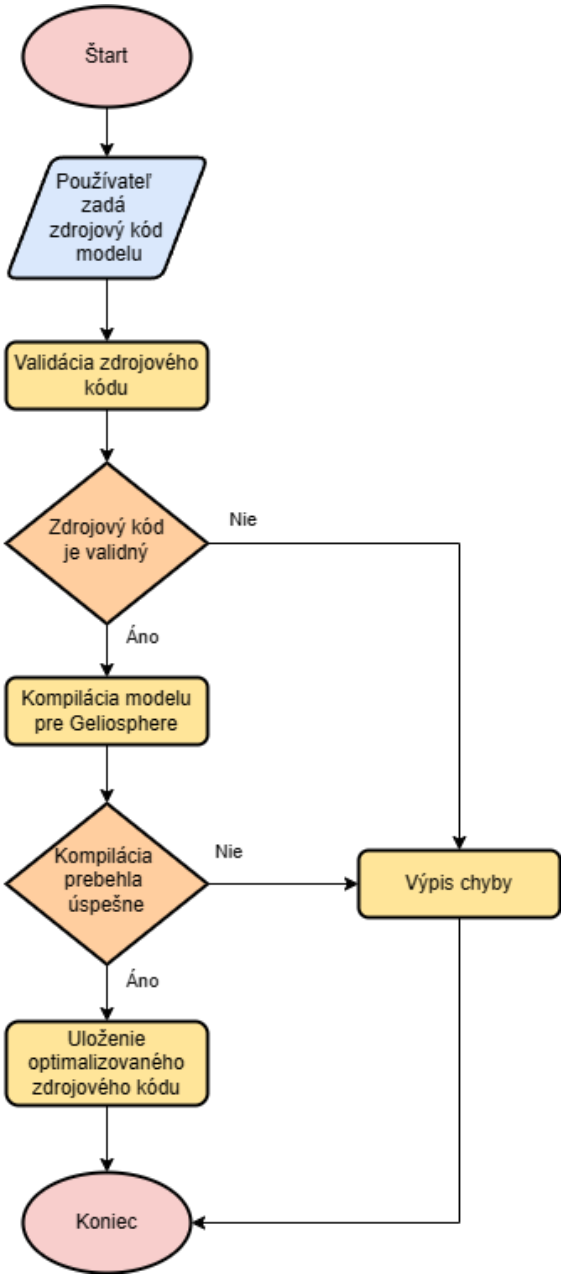
Vývojový diagram na obrázku 1.2 znázorňuje zjednodušený postup interakcie s programom *Autosphere*, ktorý je detailne popísaný v [5]. Podľa diagramu je zjavné, že mechanizmus interakcie sa dá rozdeliť na 2 hlavné kroky, a to **validácia** a **kompilácia**. Z hľadiska distribuovaného systému je nutné tieto 2 kroky kategorizovať, podľa vrstvy, v ktorej je možné daný krok realizovať.

Validácia zdrojového kódu nevyžaduje žiadne špecifické prostredie ani podporu hardvéru (napríklad GPU). Validácia zdrojového kódu, podľa [5], pozostáva len zo statickej analýzy vstupného zdrojového kódu, medzi ktoré patria napríklad kontrola syntaxe alebo validácia parametrov. Tento krok je navyše veľmi rýchly a jeho vykonávanie je možné realizovať v rámci kontextu jednej HTTP požiadavky. Pre poskytnutie čo najrýchlejšej spätnej väzby používateľovi je aj vhodnejšie vykonávať validáciu hneď, ako ju zbytočne distribuovať a spracovávať asynchrónne. Vymenované vlastnosti potvrdili, že validačnú časť je prirodzené integrovať do **centrálnej vrstvy** distribuovaného systému. Vďaka flexibilnej architektúre distribuovaného systému je možné pre túto časť navrhnúť napríklad novú mikroslužbu, ktorej doménou budú fyzikálne model pre následné simulácie. Do zvyšnej časti systému je potom nutné pridať integráciu s touto službou.

Kompilácia, na rozdiel od validačnej časti, si už vyžaduje určité prostredie, resp. podporu hardvéru. Výsledný skompilovaný model je optimalizovaný pre určitý hardvér, na ktorom prebiehala kompilácia[5]. Pokiaľ je model určený aj pre počítanie na grafických kartách, je pri kompilácii taktiež nutná prítomnosť GPU. Kompilačná časť preto musí byť vykonávaná na **distribuovanej** úrovni systému, konkrétne na každej výpočtovej stanici. Pre zabezpečenie správnej koordinácie

je v distribuovanom systéme potrebné rozšíriť distribučný mechanizmus o nový typ distribučnej jednotky, v tomto prípade *kompilácie*. Distribuované výpočtové stanice musia mať prehľad o stave svojich modelov. Pri obdržaní výpočtovej jednotky musí stanica správne zareagovať, ak model ešte doposiaľ neexistuje v jej lokálnom úložisku.

Pre distribuovaný systém táto požiadavka znamená zmenu, resp. úpravu na všetkých úrovniach. Výhodou tejto úpravy je však automatizácia systému, lepší budúci rozvoj a hlavne spravovanie modelov priamo používateľmi, bez nutnosti zásahov programátorov.



Obr. 1.2: Vývojový diagram interakcie s programom Autosphere

1.2.2 Autentifikácia a autorizácia distribuovaných výpočtových staníc

Druhou požiadavkou je zlepšenie bezpečnosti systému, hlavne pri komunikácii distribuovaných uzlov systému s centralizovanou vrstvou. V prototype z bakalárskej práce je komunikácia medzi jednotlivými vrstvami systému realizovaná pomocou služby *RabbitMQ*. Jedná sa teda o komunikáciu pomocou vymieňania správ. V prieskume o bezpečnej komunikácii [12] autori spomínajú rôzne bezpečnostné mechanizmy pri komunikácii pomocou *RabbitMQ*. Možno konštatovať, že väčšinu spomínaných bezpečnostných mechanizmov prototyp už obsahuje. Medzi nich radíme napríklad šifrovanie komunikácie pomocou TLS alebo autentifikáciu pomocou jedinečného mena a hesla. Okrem toho sa však v prototype na tejto vrstve nenachádza žiadna autorizácia. To otvára vektor útoku, v rámci ktorého škodlivý, nebezpečný uzol dokáže manipulovať so systémom nedovoleným spôsobom. Medzi hrozby, na ktoré je prototyp náchylný radíme:

- **Nedovolené odpočúvanie komunikácie**

  Jednotlivé výpočtové jednotky sú v prototype medzi vrstvami vymieňané pomocou RabbitMQ exchange typu *direct* [1]. Direct exchange využíva trasovanie na základe kľúča. V prípade predmetného distribuovaného systému je kľúčom unikátny identifikátor výpočtovej stanice. Pokiaľ škodlivá stanica získa identifikátor inej stanice, dokáže komunikáciu odpočúvať a taktiež kradnúť výpočty, ktoré boli pridelené iným staniciam.

- **Vyčerpanie fronty výpočtových jednotiek**

  Prototyp nie je ošetrený na prípadný útok rýchlym vyčerpaním výpočtovej fronty. Škodlivá stanica dokáže požiadať o vysoký počet výpočtových jednotiek naraz, čím jej budú pridelené a rezervované. Tieto výpočtové jednotky si môže udržiavať aj donekonečna, pokiaľ bude hlásiť, že sú stále v stave počítania. V najhoršom prípade môže nastať až úplné uviaznutie (angl. deadlock) simulácií.

- **Vydávanie sa za inú výpočtovú stanicu**

  Pokiaľ škodlivá stanica získa prístup k výpočtovým jednotkám cudzích staníc, napríklad ako je to uvedené v prvej spomenutej hrozbe, dokáže odoslať nepravé, zmanipulované výsledky centralizovanej vrstve. Overenie totož-

[1] `https://www.rabbitmq.com/tutorials/amqp-concepts#exchange-direct`

nosti stanice sa realizuje len na základe jej identifikátora, preto je maskovanie za inú výpočtovú stanicu veľmi jednoduché.

Spomínané hrozby je pred uvedením systému do produkčného prostredia nutné zmierniť, resp. zamedziť úplne. Minimálnym riešením je zlepšiť zabezpečenie priamo v *RabbitMQ* tak, ako odporúčajú autori v [12]. Pre podporu dynamickej autentifikácie, autorizácie a kontroly nad výpočtovými stanicami je možné prepojiť akcie v systéme s riadiacim API služby *RabbitMQ*. Na *RabbitMQ* je v prototype kladených až príliš mnoho zodpovedností. Oveľa lepším riešením je vyňať tieto zodpovednosti z komunikačnej vrstvy napríklad do jednej z backendových služieb systému. Výhodou je dosiahnutie plnej kontroly nad správou staníc a zlepšenie bezpečnosti odstránením vyššie spomínaných hrozieb.

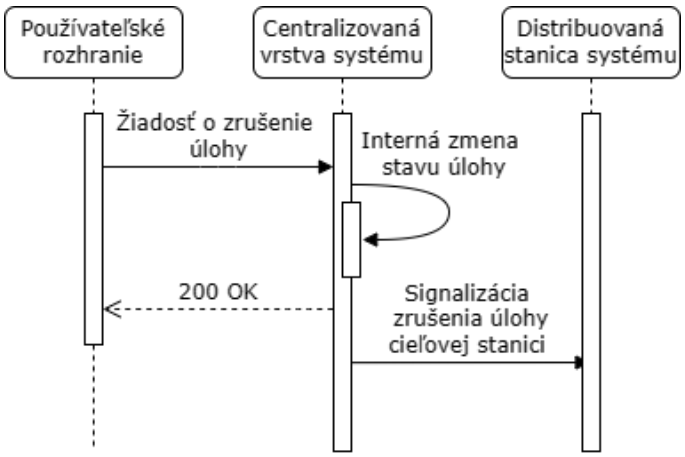
1.2.3 Administrácia distribuovaných výpočtov

Témou práce je taktiež zabezpečiť možnosť administrácie distribuovaných výpočtov (privilegovaným) používateľom a správcom systému. V existujúcom prototype je kontrola a administrácia samotných výpočtov obmedzená. Kontrolovať výpočty je možné len na úrovni simulácií, a teda jej pridružených, menších výpočtov. Distribuované výpočty momentálne nie je možné administrovať zo strany používateľa. Administrácia by mala podporovať minimálne tieto úkony:

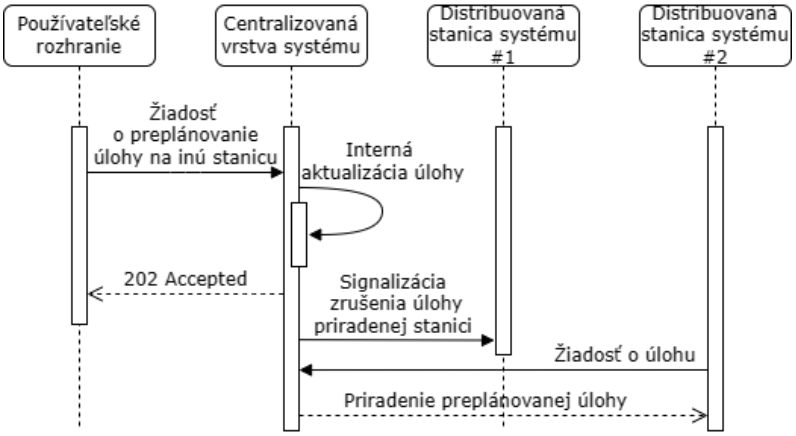
- **Detailný prehľad o jednotlivých distribuovaných úlohách**. Prehľad musí zahŕňať minimálne momentálny stav úlohy, históriu vykonávania a pod.
- **Možnosť predčasne zrušiť úlohu (angl. cancelling)**.
- **Možnosť preplánovať úlohu na inú stanicu (angl. rescheduling)**.
- **Možnosť opakovať úlohu (angl. retry)**.

Celá administrácia a prehľad distribuovaných úloh musí byť pravidelne, v reálnom čase, aktualizovaná. Systém musí flexibilne reagovať na akcie. Napríklad, po obdržaní žiadosti o zrušenie úlohy musí systém čo v najkratšom čase akciu vykonať až na úrovni distribuovanej stanice, ktorá práve počíta danú úlohu. Sekvenčné diagramy na obrázkoch 1.3, 1.4 a 1.5 popisujú očakávané správanie distribuovaného systému pri jednotlivých administračných akciách.

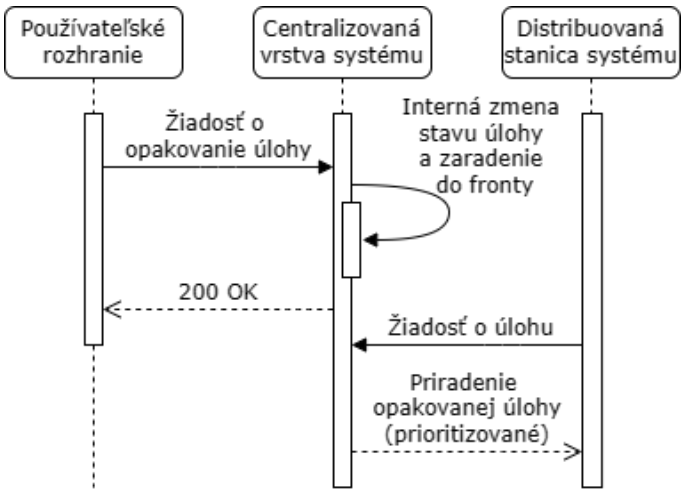
Z analýzy požiadavky v sekcii 1.2.1, o podpore vlastných simulačných modelov definovaných používateľom vyplýva, že systém musí podporovať viacero **rozličných typov distribuovaných úloh**, nie len samotné úlohy pre výpočty simulácií. V budúcnosti do systému môžu pribudnúť ešte ďalšie typy úloh, napríklad analýza modelov, ktorá využíva umelú inteligenciu pre lepšiu optimalizáciu



Obr. 1.3: Očakávaná sekvencia používateľskej akcie rušenia distribuovanej úlohy



Obr. 1.4: Očakávaná sekvencia používateľskej akcie preplánovania distribuovanej úlohy na inú stanicu



Obr. 1.5: Očakávaná sekvencia používateľskej akcie opakovania distribuovanej úlohy

a iné. Návrh a implementácia distribučného mechanizmu musí byť preto **generická**, bez ohľadu na typ úlohy. Mechanizmus nesmie byť viazaný na žiadnu doménovú logiku jednotlivých úloh, aby nebola poškodená generickosť riešenia a systém dokázal integrovať nové typy úloh bez úprav distribučného mechanizmu. Značná časť optimalizácie architektúry systému sa musí odrážať od tejto skutočnosti. Správnym návrhom nového distribučného mechanizmu sa docieli podpora administrácie jednotlivých distribuovaných úloh zo strany používateľa.

Keďže je táto požiadavka úzko prepojená aj s inými spomínanými požiadavkami, a taktiež predstavuje najväčší zásah do existujúcej architektúry systému, je pre správnu optimalizáciu dôležité zanalyzovať podobné systémy, ktoré využívajú distribuované úlohy. V kapitole 2 je detailne rozpracovaný koncept distribuovaného plánovania spoločne s priblížením niekoľkých produkčných systémov, ktoré používajú distribuované počítanie a plánovanie úloh.

1.2.4 Aktualizácia používateľského rozhrania

Menšou požiadavkou je taktiež aj aktualizácia používateľského rozhrania. V bakalárskej práci používateľské rozhranie aktualizované nebolo. Ako bolo uvedené v úvode tejto kapitoly, komunikácia so systémom prebieha pomocou REST API jednotlivých služieb. Pred uvedením do produkčného prostredia je vhodné aktualizovať webové používateľské rozhranie, pre zjednodušenie prístupu a manipulácie so systémom.

Pôvodné používateľské rozhranie je popísané vo viacerých prácach [3][4][5]. Toto používateľské rozhranie bolo vytvorené pre potreby prvého prototypu systému[3]. Obsahuje však veľmi obmedzený set funkcionalít, len prehľad simulácií a vytvorenie nových. Od doby vytvorenia spomínaného používateľského rozhrania prešiel distribuovaný systém veľkými zmenami a rozšíreniami. Pôvodné rozhranie je preto neaktuálne z hľadiska použitých technológií a balíčkov, ale aj z hľadiska obsiahnutých funkcionalít.

Okrem aktualizácie technológií musí používateľské rozhranie zahrnúť aj nové požiadavky, ktoré sú definované v tejto kapitole. Používatelia a správcovia systému musia mať možnosť plne ovládať systém, bez nutnosti zásahov programátorov. Aktualizované používateľské rozhranie musí obsahovať minimálne tieto funkcionality, vyplývajúce z bakalárskej, ako aj tejto práce:

- Správa a prehľad distribuovaných staníc.
- Správa a prehľad simulačných modelov. K tejto funkcionalite patrí aj vytváranie vlastných modelov, opísaných v požiadavke v sekcii 1.2.1.

- Prehľad existujúcich simulácií, resp. výsledkov dokončených simulácií.
- Vytváranie nových simulácií. Pri vytváraní musí byť možné zvoliť simulačný model, ktorý bude použitý na výpočet samotnej simulácie.

Pri lokálnom skúmaní používateľského rozhrania, jeho technológií a použitých balíčkov bolo zistené, že niektoré použité balíčky už neexistujú, resp. ich podpora a aktívny vývoj skončili. Pri aktualizácií používateľského rozhrania je teda nutné minimálne nahradiť takéto balíčky za iné, podporované, ktoré neobsahujú zraniteľnosti v bezpečnosti. Druhou možnosťou je nepokračovať v existujúcej webovej aplikácií, ale **vytvoriť nové** používateľské rozhranie, pravdepodobne taktiež vo forme webovej aplikácie. Tejto možnosti nahráva aj fakt, že existujúce funkcionality v pôvodnom používateľskom rozhraní sú neaktuálne, najmä kvôli miernej zmene API jednotlivých služieb.

Pre zvýšenie celkovej bezpečnosti systému by bolo vhodné nepublikovať interné služby centralizovanej vrstvy do verejnej siete. Prístup k interným službám je realizovaný pomocou API brány, špecifickej pre jedno používateľské rozhranie. Takýto vzor sa nazýva Backend for Frontend (BFF). Autori v [13] a [14] popisujú výhody použitia BFF v distribuovaných systémoch, špecificky pri použití architektúry mikroslužieb. Podľa spomínaných článkov medzi najvýraznejšie výhody patria **zlepšená bezpečnosť** systému a **možnosť prispôsobenia žiadostí** a dát pre stránky webovej aplikácie. BFF pomocou agregácie dát z viacerých interných služieb systému dokáže pripraviť špecifické dáta potrebné pre zobrazenie na stránkach webovej aplikácie. Zvýšenie bezpečnosti je dosiahnuté vystavením len jedného komponentu - BFF do verejnej siete. Iba BFF má prístup do internej siete, kde dokáže komunikovať s internými službami systému.

1.2.5 Integrácia s novou verziou programu Geliosphere

V starších verziách programu Geliosphere zaberala značnú časť výpočtu simulácie inicializácia samotnej aplikácie. Pri spustení Geliosphere sa najprv alokovala pamäť na GPU, potrebná pre výpočet simulácie, inicializovali sa generátory náhodných čísel, a podobne. Každý výpočet simulácie, resp. každý štart aplikácie vyžadoval **4 až 9 minút** na inicializáciu [3].

Pri jemnozrnnom distribučnom modeli, ktorý bol implementovaný v bakalárskej práci [4], dĺžka inicializácie predstavovala výrazné časové straty, dokonca spomalila celý systém (výpočet celej simulácie) oproti pôvodnej implementácii. Pre podporu jemnozrnného modelu, bol do Geliosphere pridaný tzv. **interaktívny mód**[4]. V interaktívnom móde, sa program Geliosphere spustil a iniciali-

zoval len raz, pri štarte distribuovanej stanice. Následne kontrolný mechanizmus sledoval proces s Geliosphere a komunikácia medzi nimi prebiehala pomocou TCP socketov a UNIX signálov. Vďaka tejto funkcionalite sa systém zrýchlil, pretože vynechal inicializačnú časť pri každom výpočte.

Od času písania bakalárskej práce vyšlo niekoľko nových verzií programu Geliosphere. Vo verzii *1.1.0*[2] bol značne zmenený inicializačný mechanizmus. Čas potrebný na inicializáciu klesol z niekoľkých minút na **zopár sekúnd**. Interaktívny mód sa týmto stáva nepotrebným. Komunikácia, udržiavanie a sledovanie procesu pri interaktívnom móde sú náročnejšie a náchylnejšie na zlyhania. Je vhodné upraviť implementáciu z bakalárskej práce integrovaním najnovšej verzie Geliosphere. Výhodou je hlavne zjednodušenie komunikácie medzi procesmi, kontrola nad dcérskym procesom a taktiež lepšia odolnosť distribuovanej stanice voči zlyhaniam.

Okrem vyššie spomínaných optimalizácií, ktoré boli implementované v nových verziách Geliosphere, bolo taktiež zjednodušené CLI rozhranie programu. Nadbytočné konfiguračné parametre a prepínače boli odstránené. Rovnako boli pridané niektoré nové parametre, ktoré boli pridané spolu s novými funkcionalitami, ako napríklad podpora vlastných modelov. Tabuľka 1.2 obsahuje všetky konfiguračné parametre programu Geliosphere, ktoré sú k dispozícii v najnovšej verzii programu[3].

Jednou z požiadaviek tejto práce je taktiež integrovať novú verziu Geliosphere s distribuovaným systémom, pričom je nutné upraviť spúšťanie výpočtov podľa podporovaných parametrov z tabuľky 1.2. Menšie úpravy sú nutné aj pri čítaní výsledkov z výsledkových súborov, nakoľko ich štruktúra sa mierne zmenila. Ukážka kódu 1.1 obsahuje premennú s regex vzorom slúžiacim na extrakciu výsledkov. Táto časť kódu pochádza z prototypu z bakalárskej práce [4].

Zdrojový kód 1.1: Premenná s regex vzorom na extrakciu výsledkov

```
private readonly Regex _extractionRegex =
    new Regex("(\\S*) (\\S*) (\\S*)", RegexOptions.Compiled);
```

Menším experimentom bolo zistené, že pôvodný regex vzor už nedokáže správne vyextrahovať výsledky, čím sa znehodnotia výsledky celej simulácie v rámci distribuovaného systému. Vyšetrením sa ukázalo, že pôvodný vzor dokáže extrahovať výsledky 1D simulácií, avšak nie 2D simulácií. Pre správne fungovanie systému je nutné opraviť túto nezrovnalosť a podporovať všetky typy simulácií, 1D aj 2D.

[2] Detaily verzie: `https://github.com/msolanik/Geliosphere/releases/tag/1.1.0`

[3] V čase písania práce je to verzia *1.1.1*

Parameter	**Typ hodnoty**	**Popis**
-F	/	Simulácia typu Forward-in-time
-B	/	Simulácia typu Backward-in-time
-E	/	Simulácia typu SOLARPROPLike 2D Backward-in-time
-T	/	Simulácia typu Geliosphere 2D Backward-in-time
-d	float	Časový krok (5.0s)
-K	float	K0 ($5 * 10^{22} cm^2/s$)
-V	float	Solárna rýchlosť vetra (400 km/s)
-N	int	Počet testovaných častíc - v miliónoch
-m	int	Načítaj K0 a V pre daný mesiac z Usoskinovych tabuliek pre 1D a K0 a uhol sklonu pre 2D
-y	int	Načítaj K0 a V pre daný rok z Usoskinových tabuliek pre 1D a K0 a uhol sklonu pre 2D
--custom-model	string	Simulácia na základe vlastného modelu
--cpu-only	/	Simulácia bude počítaná len na CPU
-s	string	Cesta ku konfiguračnému súboru vo formáte *TOML* (Settings.tml v aktuálnom priečinku)
-p	string	Vlastná cesta pre výsledky v priečinku s výsledkami
-c	/	Výstup vo formáte .csv
-h	/	Pomoc pre Geliosphere

Tabuľka 1.2: Zoznam konfiguračných parametrov programu Geliosphere vo verzii 1.1.1

1.3 Nasadenie systému - Deployment

Už v bakalárskej práci [4] boli všetky aplikácie centralizovaných služieb, ako aj aplikácia distribuovanej stanice kontajnerizované. Každá aplikácia obsahuje vlastnú definíciu Dockerfile, ktorá ju a všetky jej potrebné závislosti zabalí do obrazu (image) a je pripravená na spustenie. V bakalárskej práci [4] bola pre účely vývoja a testovania prototypu systému vytvorená definícia Docker Compose, pomocou ktorej bolo možné celý systém jednoducho nasadiť na testovacie prostredie.

Docker Compose však nie je vhodný na produkčné nasadenie. Toto tvrdenie podporili aj autori Ľuboš Mercl a Jakub Pavlík v [15], kde porovnávali najznámejších kontajner ochestrátorov. Podľa nich v Docker Compose chýbajú kľúčové funkcionality, ktoré produkčne pripravený orchestrátor musí obsahovať, ako napríklad automatizované škálovanie, priebežné aktualizácie (angl. rolling updates) alebo vyvažovanie záťaže (angl. load balancing). Primárne je centralizovaná vrstva určená na nasadenie do cloudového prostredia. Podpora Docker Compose od poskytovateľov cloudových riešení je tiež obmedzená.

Jednou z možností je pripraviť skripty nasadenia pre každú službu, podľa zvolenej cloudovej platformy. Cloudové platformy poskytujú manažované prostredia pre kontajnerizované aplikácie. Príkladmi takýchto prostredí sú napríklad *Azure Container Apps*[4] alebo *Amazon Elastic Container Service*[5]. Nevýhodou tohto riešenia je však tzv. **vendor locking**, čo v praxi znamená orientáciu len na jedného špecifického poskytovateľa cloudového riešenia. Potenciálny zákazníci systému si tak nemôžu nasadiť systém na ich preferované prostredia, bez vytvárania nových, vlastných definícií nasadenia.

Druhou možnosťou, ktorá nespôsobí prípadný vendor locking, je orientácia na všeobecnú platformu, ako napríklad **Kubernetes**. Podľa dokumentácie [16] a autorov [15], Kubernetes spĺňa všetky štandardy a požiadavky produkčného orchestrátora. Navyše takmer každý poskytovateľ cloudových riešení má podporu pre Kubernetes. Nasadenie do prostredia Kubernetes sa realizuje pomocou manifestov, písaných v jazyku YAML. Pre budúci rozvoj systému je ideálna práve táto druhá možnosť. Príprava manifestov pre nasadenie systému bude tiež súčasťou tejto práce.

[4] `https://azure.microsoft.com/en-us/products/container-apps`
[5] `https://aws.amazon.com/ecs/`

2 Distribuované plánovanie úloh

Pinedo, vo svojej knihe [17], definuje plánovanie ako *rozhodovací proces*, ktorý je pravidelne využívaný v mnohých odvetviach. Zaoberá sa **prideľovaním dostupných zdrojov úlohám** v daných časových obdobiach, pričom jeho úlohou je optimalizácia jedného alebo viacerých cieľov. V bakalárskej práci [4] bol navrhnutý a implementovaný jednoduchý plánovač, ktorý mal na starosti distribúciu úloh medzi dostupné distribuované stanice. V prototype existuje len jeden typ distribuovanej úlohy, a to výpočet časti simulácie. Tento plánovač [4] bol implementovaný ako súčasť simulačnej mikroslužby. Niektoré požiadavky z podkapitoly 1.2 vyžadujú podporu ďalších typov distribuovaných úloh, ako napríklad kompiláciu vlastného simulačného modelu, a podobne. Existujúci plánovač je úzko prepojený s doménovou logikou výpočtov simulácií, preto ho nie je možné jednoducho rozšíriť o podporu ďalších typov úloh. Najlepším riešením je abstrahovať distribuované plánovanie úloh do generického riešenia - vlastný plánovač (angl. scheduler). Plánovač je potrebné navrhnúť tak, aby bol schopný pracovať s akýmkoľvek typom distribuovaných úloh a **nesmie byť závislý** na doménových detailoch úlohy, aby sa neporušila generickosť riešenia a zabezpečila sa jednoduchá rozšíriteľnosť do budúcnosti.

V tejto kapitole sú porovnávané existujúce systémy, ktoré využívajú distribuované plánovanie. Typickými príkladmi takýchto systémov, ktoré využívajú distribuované plánovanie v ich jadre sú platformy CI/CD. Konkrétne sa táto kapitola zaoberá systémami **GitLab CI** [18] a **GitHub Actions** [19]. Oba systémy implementujú *generické* distribuované plánovače, ktoré podporujú rôzne typy úloh. Do tejto kategórie je ešte vhodné spomenúť aj samotný Kubernetes, konkrétne jeho **Kube-scheduler** [16], ktorý je taktiež implementáciou distribuovaného plánovača. Tento plánovač však plánuje iný typ "úloh", ktoré sa v zásade líšia od úloh v predmetnom distribuovanom systéme, preto nebude podrobnejšie analyzovaný.

Detailná analýza a experimenty s vyššie spomenutými systémami pomôžu pri návrhu a implementácii plánovača pre predmetný distribuovaný systém pre výpočty simulácií v heliosfére.

2.1 GitLab CI

V dokumentácii [18] je GitLab CI definovaný ako systém, ktorý zabezpečuje metódu kontinuity vývoja [1]. Je súčasťou platformy *GitLab*. Systém sa skladá z dvoch častí - **centralizovanej služby**, ktorá riadi úlohy (tu sa nachádza aj implementácia plánovača), a **distribuovaných agentov** - GitLab Runners, na ktorých sú úlohy vykonávané. GitLab Runners komunikujú s centralizovanou vrstvou pomocou HTTP API.

Úlohy sú plne definované používateľom v jazyku YAML. Keďže GitLab je platforma pre (najmä) správu Git repozitárov, definície úloh sú uložené priamo v Git repozitári, v súbore *.gitlab-ci.yml*. Pri výskyte akejkoľvek udalosti spojenej s repozitárom, ako napríklad `git push`, sú jednotlivé úlohy, definované vo vyššie spomenutom súbore, naplánované a vykonané na jednotlivých distribuovaných agentoch. Touto architektúrou je možné automatizovať väčšinu opakujúcich sa akcií a zároveň paralelizovať vykonávanie, ako spomínajú v [20], [21] a [22]. Vďaka tomuto predpokladu vzniklo niekoľko rámcov [21] [22], ktoré využívajú architektúru tejto platformy a integrujú systém *GitLab CI*. Nasledujúce sekcie približujú, ako je vyriešené zabezpečenie a samotné plánovanie a vykonávanie úloh.

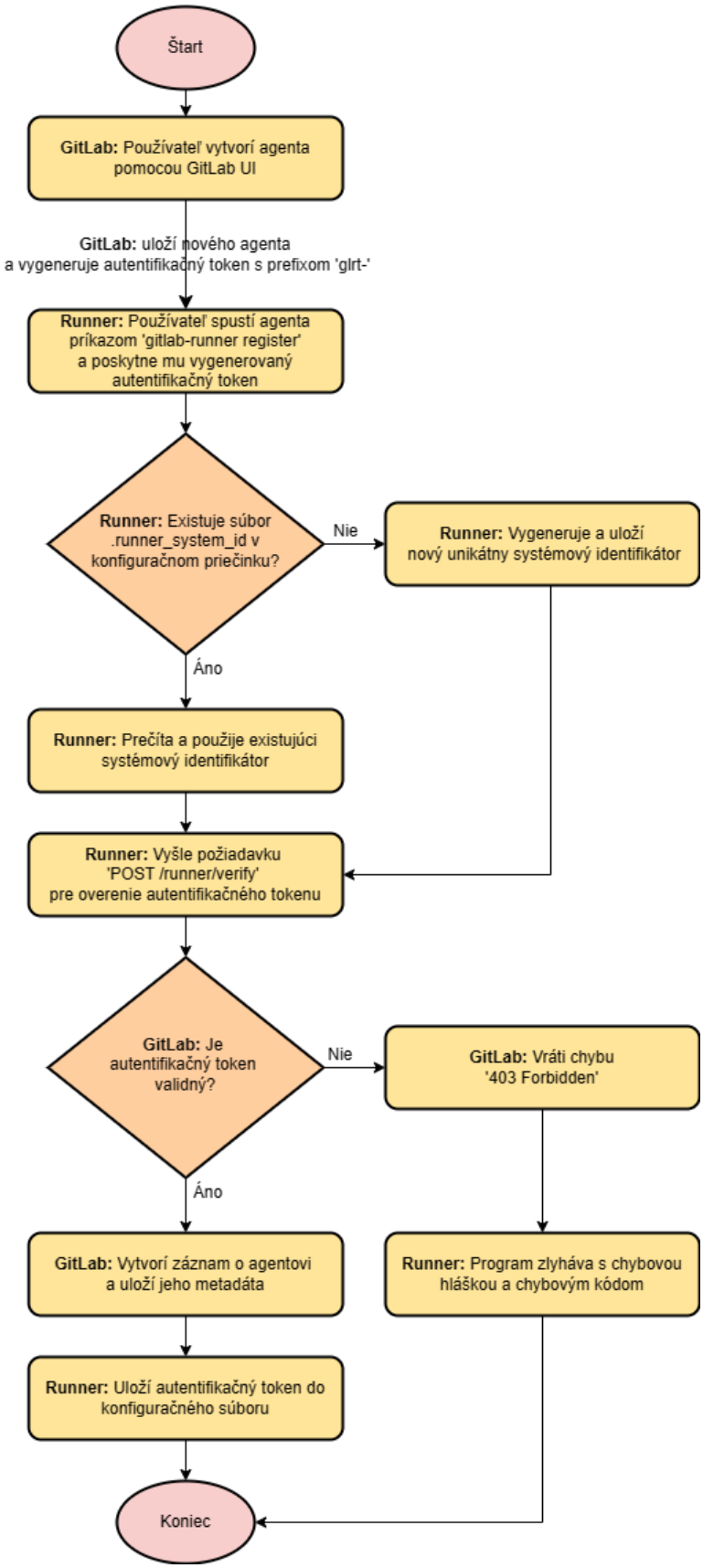
2.1.1 Autentifikácia agentov - GitLab Runners

V prípade platformy *GitLab* je registrácia a autentifikácia agentov dvojfázová. V prvom kroku je potrebné vytvoriť zaregistrovať, resp. vytvoriť agenta pomocou používateľského rozhrania, resp. pomocou HTTP API. Používateľ môže pri vytváraní pridať agentovi rôzne metadáta, ako napríklad názov, krátky popis alebo tagy. Po úspešnom vytvorení sa zobrazí **autentifikačný token**[2] a inštrukcie na spustenie agenta. V druhom kroku správca pomocou tohto tokenu spustí agenta príkazom `gitlab-runner register`, čím sa overí platnosť tokenu a inicializuje sa prostredie pre agenta. Následne, po overení autentifikačného tokenu, je možné agenta spustiť príkazom `gitlab-runner run`. Pri následnej komunikácii s centralizovanou vrstvou sa používa autentifikačný token na overenie identity a autorizáciu. Na obrázku 2.1 je znázornený vývojový diagram celého procesu dvojfázovej registrácie distribuovaného agenta.

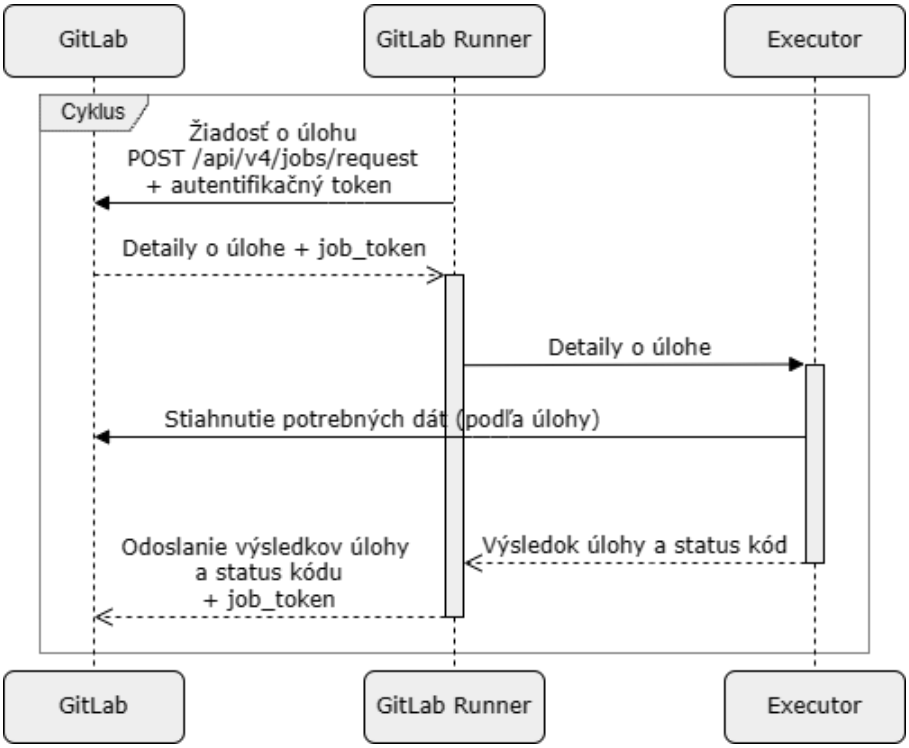
Takýto autentifikačný mechanizmus je jednoducho implementovateľný, pričom stále zachováva bezpečnosť a dovoľuje jednoznačnú identifikáciu distribu-

[1] metóda vývoja softvéru, kde sústavne vytvárate, testujete, nasadzujete a monitorujete iteračné zmeny kódu

[2] autentifikačný token začína prefixom *glrt-*



Obr. 2.1: Vývojový diagram procesu registrácie distribuovaného agenta na platforme GitLab[18]



Obr. 2.2: Sekvencia hlavného vykonávacieho cyklu distribuovaného agenta na platforme GitLab[18]

ovaných agentov. Vďaka týmto vlastnostiam dokáže pokryť jednu z požiadaviek tejto práce, definovanú v sekcii 1.2.2.

2.1.2 Plánovanie a vykonávanie distribuovaných úloh

Rovnako ako prototyp z bakalárskej práce [4], aj systém *GitLab CI* využíva pri plánovaní techniku **plánovania riadeného dopytom**. Jedná sa o architektúru, pri ktorej si distribuovaný agent aktívne pýta prácu (angl. pull-based). V systéme *GitLab CI* je táto technika implementovaná pomocou mechanizmu pravidelne sa opakujúcich HTTP volaní (angl. short-polling). Podľa dokumentácie [18] je časový interval medzi jednotlivými žiadosťami o prácu nastavený na **3 sekundy**. Akonáhle dostane server žiadosť od agenta, naplánuje a priradí mu úlohu na spracovanie, za predpokladu, že nejaká úloha vhodná pre žiadajúceho agenta existuje. Priradená úloha je na serveri **uzamknutá**, pokiaľ ju nedokončí agent, ktorému bola priradená. Pokiaľ agent neodpovedá dlhší čas, resp. nestihne vykonať úlohu do maximálneho povoleného času, úloha je **odomknutá** a pripravená na preplánovanie na iného agenta. Sekvenčný diagram na obrázku 2.2 popisuje vykonávací cyklus (main loop) jedného agenta.

Generickosť riešenia je zabezpečená používateľskými definíciami. Samotný

systém *GitLab CI* nepozná doménovú logiku jednotlivých úloh, ktoré definoval používateľ. Plánovač pracuje s úlohami ako s abstraktným prvkom. Dokáže ich spravovať a plánovať. Doménovú logiku pre každý typ distribuovanej úlohy obsahuje implementácia agenta, konkrétnejšie jeho časť s názvom *Executor* [18].

Toto riešenie plánovania je vhodné pre predmetný distribuovaný systém pre výpočty simulácií v heliosfére, nakoľko využíva rovnakú techniku plánovania - *plánovanie riadené dopytom* a navyše je dostatočne generické pre tento systém.

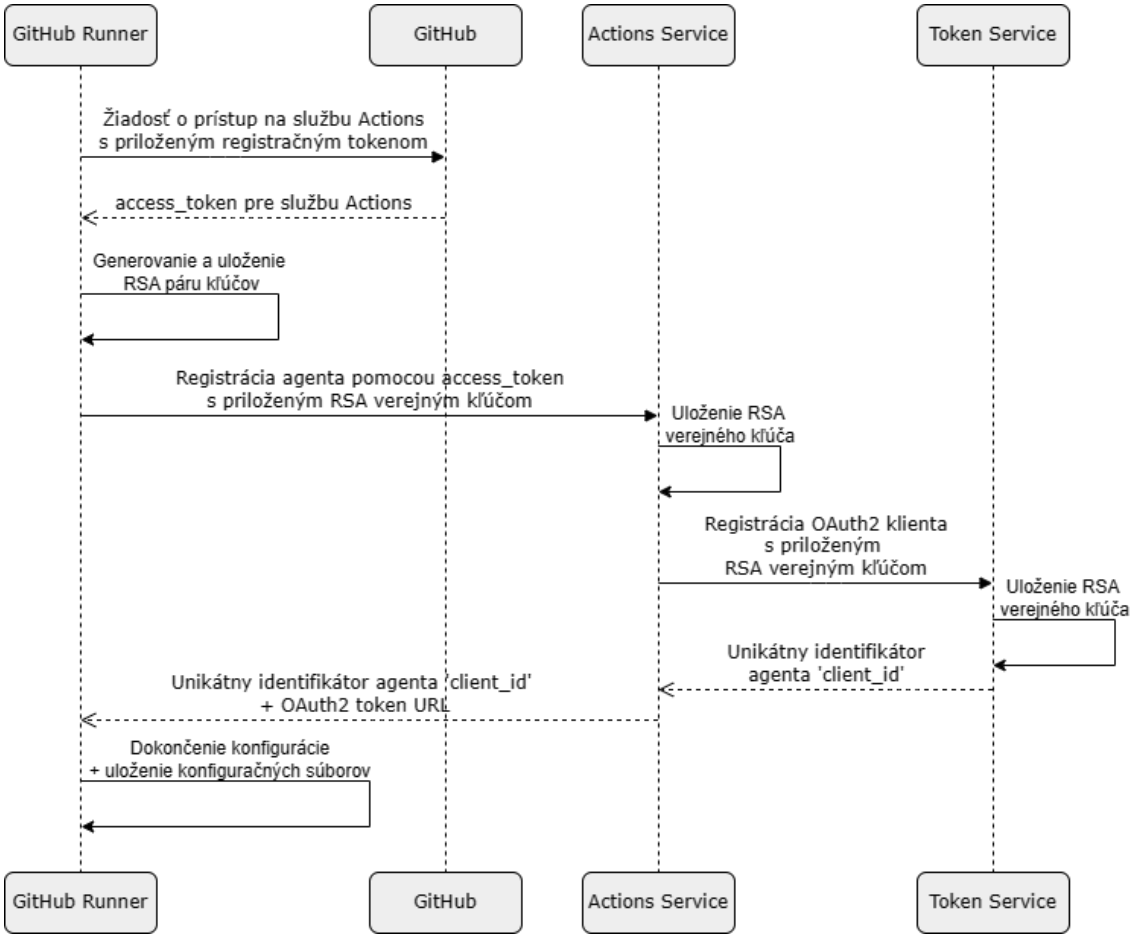
2.2 GitHub Actions

GitHub Actions je v oficiálnej dokumentácii [19] definovaná ako platforma poskytujúca Continuous integration a continuous delivery (CI/CD), ktorá umožňuje automatizovať kompiláciu, testovanie a nasadzovanie softvérových projektov. Je súčasťou platformy *GitHub*. Tento subsystém sa skladá z 3 hlavných častí - **Action mikroslužby**, **Token mikroslužby** a **distribuovaných agentov** - GitHub Runners. Táto platforma je rozšírená aj vedeckou prácou [23], v ktorej sa autori venujú integráciou tejto platformy so zdrojmi High-performance computing (HPC).

Podobne ako na platforme *GitLab*, aj tu sú úlohy definované používateľom v jazyku YAML. Rozdielom však je, že tu nie je používateľ obmedzený na jeden hlavný súbor, ako to je na platforme *GitLab*, ale definície úloh môžu byť vo viacerých súboroch. Jedinou podmienkou je, že všetky definície musia byť k dispozícii v repozitári, v priečinku *.github/workflows*. Porovnávané 2 platformy zdieľajú rovnakú doménu, avšak sa značne líšia jej návrhom a implementáciou. V nasledujúcich sekciách je priblížený systém **GitHub Actions** z hľadiska zabezpečenia, plánovania a vykonávania úloh. Rôzne prístupy oboch platforiem sú taktiež porovnané, s pridaným komentárom.

2.2.1 Autentifikácia agentov - GitHub Runners

Rovnako ako na platforme *GitLab*, aj tu je realizovaná registrácia a autentifikácia agentov dvojfázovo. Agent sa však nevytvára pomocou používateľského rozhrania, resp. pomocou HTTP API, ako to bolo v prípade systému *GitLab CI*. Pomocou používateľského rozhrania používateľ získa vygenerovaný **registračný token**. Následne je možné po stiahnutí programu agenta ho vytvoriť a nakonfigurovať, pomocou priloženého skriptu `config.sh`. Skript nakonfiguruje agenta, zaregistruje ho pomocou registračného tokenu a následne vytvorí **RSA kľúčový pár**, ktorý slúži na šifrovanie komunikácie s centralizovanými službami. Sekvenčný diagram na obrázku 2.3 popisuje akcie vykonávané pri konfigurácií agenta.



Obr. 2.3: Sekvencia registrácie a konfigurácie distribuovaného agenta na platforme GitHub[19]

Registračný token sa stáva **neplatným** ihneď po prvom použití - konfigurácii agenta. Pre autentifikáciu a autorizáciu komunikácie, agenti používajú **OAuth tokeny**, ktoré si obnovujú každých 50 minút.

GitHub Actions využíva násobne komplexnejší mechanizmus autentifikácie v porovnaní so systémom *GitLab CI*. Autentifikačné tokeny s obmedzenou dobou platnosti, implementujúce štandard *OAuth2*, spoločne s asymetricky šifrovanou komunikáciou pomocou *RSA šifrovania*, poskytujú zvýšenú bezpečnosť a ochranu vymieňaných dát. Dáta prenášane v týchto platformách často obsahujú kritické údaje, ako napríklad prístupové kľúče, napríklad na produkčné prostredia, a podobne. Predmetný distribuovaný systém pre výpočty simulácií v heliosfére ale neobsahuje žiadne kritické dáta, ktoré sú prenášané v rámci komunikácie s distribuovanými stanicami. Tento bezpečnostný mechanizmus preto neprináša ďalšie výhody, v porovnaní s mechanizmom platformy *GitLab*. Nevýhodou je zbytočný nárast náročnosti systému, ktorý takýto mechanizmus so sebou prináša. Preto je lepším kandidátom pre tento distribuovaný systém skôr mechanizmus platformy

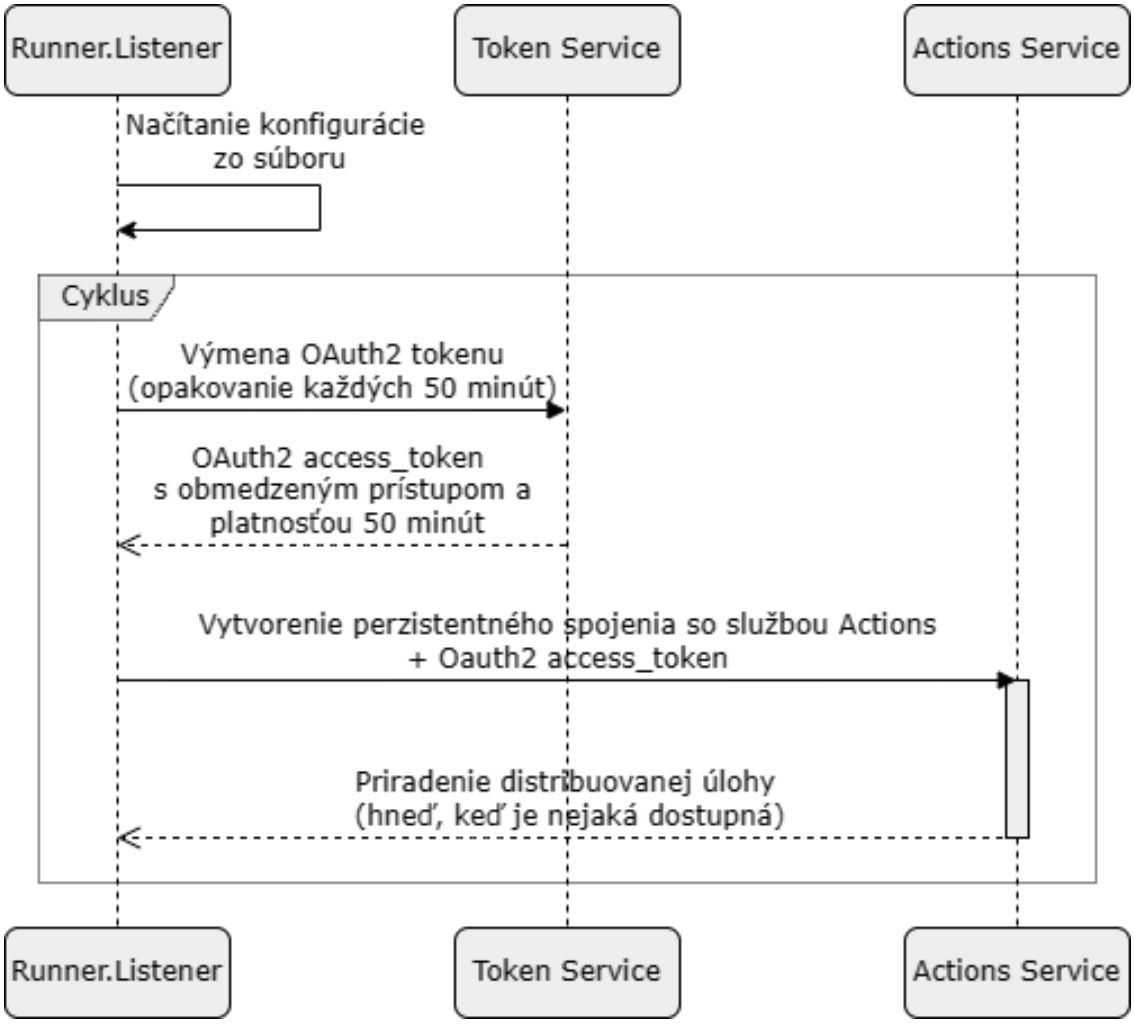
GitLab, opísaný v sekcii 2.1.1.

2.2.2 Plánovanie a vykonávanie distribuovaných úloh

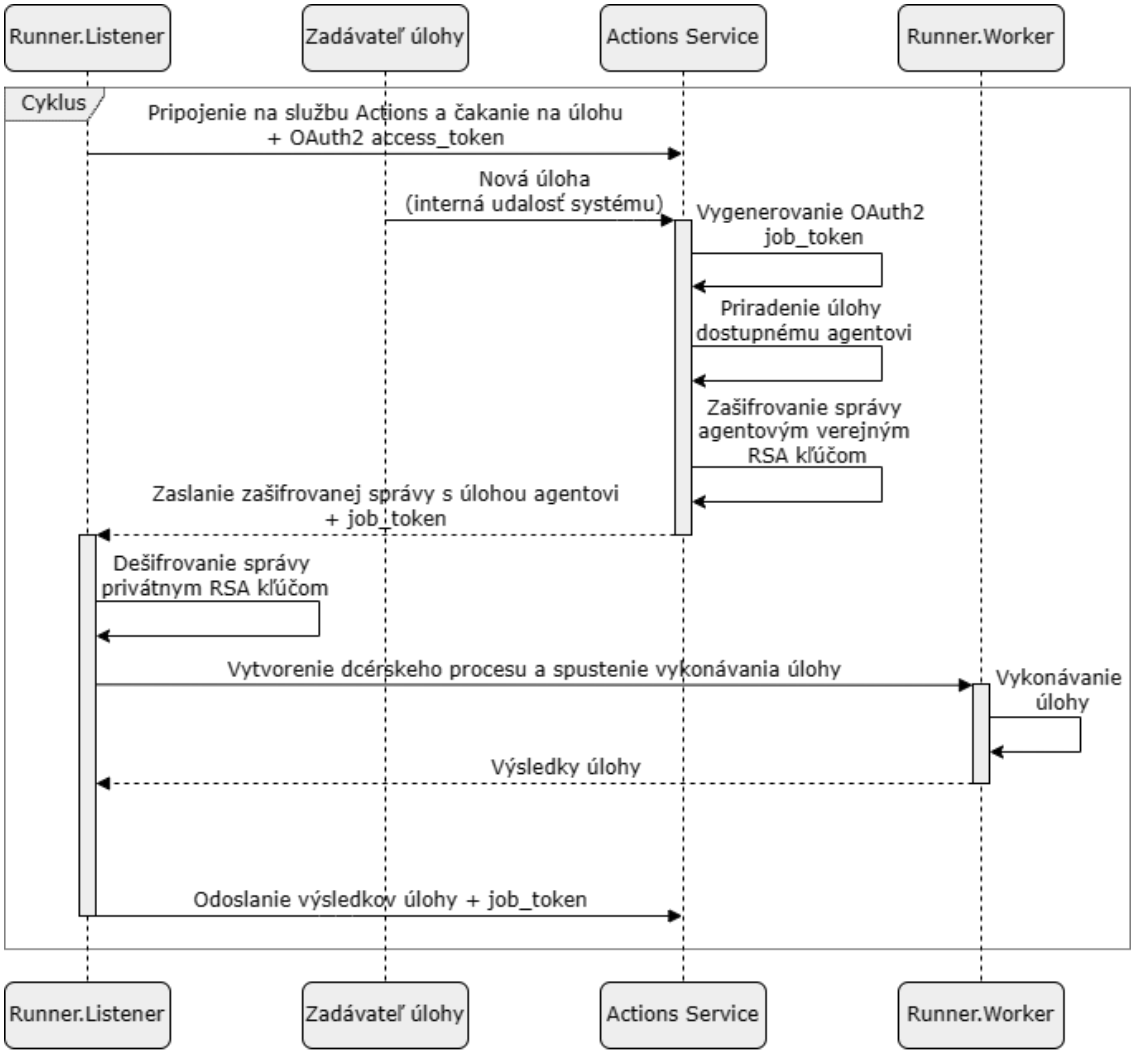
Na rozdiel od systému *GitLab CI*, ktorý využíva *plánovanie riadené dopytom*, sa v tomto systéme rozhodli implementovať plánovač, ktorý využíva **plánovanie v reálnom čase**. Charakterizuje sa ako *push-base* architektúra, kde má vždy plánovač k dispozícii aktuálny stav všetkých agentov, ktorým priamo priraďuje prácu a v reálnom čase ich notifikuje. Keďže sú úlohy priamo delené a priraďované, mechanizmus **uzamykania úloh** je preto veľmi jednoducho implementovateľný. Taktiež aj následne **odomknutie úloh**, v prípade zlyhania distribuovaného agenta, je jednoduchšie, nakoľko má plánovač vždy prístup k aktuálnemu stavu agentov. Technika plánovania v reálnom čase je implementovaná pomocou perzistentého HTTP spojenia, ktoré je vďaka *keep-alive* pingom pravidelne udržiavané. V prípade straty spojenia, môže plánovač takmer ihneď reagovať. Sekvenčný diagram na obrázku 2.4 zobrazuje akcie agenta po jeho spustení. Cyklus vytyčuje akcie, ktoré sú opakované vždy, keď vyprší platnosť OAuth2 autentifikačného tokenu - každých 50 minút. Druhý sekvenčný diagram na obrázku 2.5 popisuje akcie vykonávania pridelenej úlohy agentovi.

Rovnako ako v systéme *GitLab CI*, aj tu je generickosť riešenia zabezpečená používateľskými definíciami úloh. Obe riešenia ponúkajú takmer rovnaké funkcionality z hľadiska plánovania a správy distribuovaných úloh. Doménovú logiku pre každý typ distribuovanej úlohy v tomto prípade obsahuje časť implementácie agenta s názvom *Runner.Worker* [19].

Implementovať tento typ plánovania by bolo pre predmetný distribuovaný systém pre výpočty simulácií v heliosfére náročnejšie a vyžadovalo by si väčšie zmeny v jadre systému, nakoľko tento systém momentálne využíva inú techniku plánovania. Na základe toho sa zdá byť lepšou možnosťou pre tento systém zachovať pôvodný plánovací mechanizmus a využiť poznatky z analýzy platformy *GitLab*, zo sekcie 2.1.2, na jeho vylepšenie.



Obr. 2.4: Sekvencia akcií autentifikácie a pripojenia distribuovaného agenta na platforme GitHub[19]



Obr. 2.5: Sekvencia procesu vykonávania úlohy pridelenej distribuovanému agentovi na platforme GitHub[19]

3 Návrh optimalizovanej architektúry distribuovaného systému

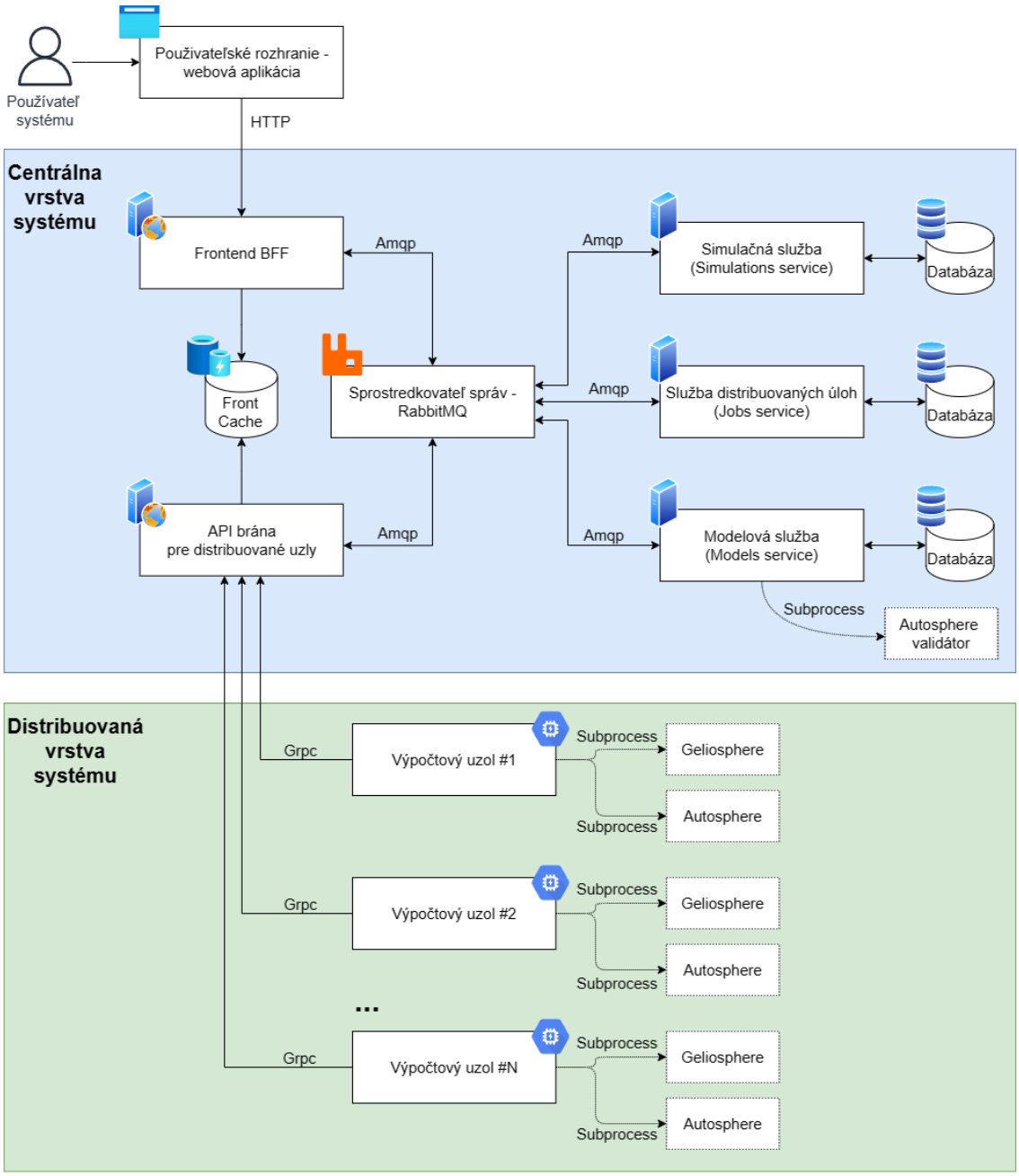
Táto kapitola opisuje návrh optimalizovanej architektúry pre distribuovaný systém pre výpočty simulácii v heliosfére. Návrh je inšpirovaný hlavne podobnými systémami, ktoré využívajú distribuované plánovanie úloh, sú úspešne nasadené v produkčnom prostredí a denne vykonávajú obrovské množstvo distribuovaných úloh. Tieto systémy boli analyzované v kapitole 2. Nový návrh však dbá na novo definované požiadavky systému, definované v podkapitole 1.2, ako aj na zachovanie všetkých existujúcich funkcionalít a požiadaviek systému z bakalárskej práce[4].

Optimalizovaná architektúra v návrhu obsahuje zmeny v takmer všetkých komponentoch systému. Niektoré nové komponenty sú do systému **pridané**. Pôvodné, existujúce komponenty sú **čiastočne pozmenené**, aby zapadli do novej architektúry. Nasledujúce podkapitoly popisujú novú architektúru a zmeny v oblastiach distribuovaného systému z nej vyplývajúce.

3.1 Celkový prehľad architektúry

V kapitole 1 sú popísané viaceré možnosti rozšírenia systému, vrátane splnenia nových požiadaviek. Nový návrh architektúry systému sa opiera o tieto tvrdenia a prináša rozšírenie systému, s dôrazom na zachovanie čo najviac pôvodných, existujúcich komponentov. Finálne zvolený návrh celkovej architektúry systému je znázornený na obrázku 3.1.

Finálny návrh spĺňa všetky požiadavky, definované v kapitole 1. Obsahuje **4 nové komponenty** systému, zobrazuje zmeny v existujúcich komponentoch a mierne pozmeňuje komunikačnú vrstvu a mechanizmy komunikácie v systéme. V ďalších podkapitolách sú detailne rozpracované zmeny, ktoré tento návrh architektúry prináša.



Obr. 3.1: Návrh celkovej optimalizovanej architektúry distribuovaného systému

3.2 Nové komponenty systému

Z celkového prehľadu návrhu architektúry z podkapitoly 3.1, medzi nové komponenty v systéme patria:

- Služba distribuovaných úloh - Jobs service
- Modelová služba - Models service
- API brána pre distribuované uzly
- Frontend BFF pre webové používateľské rozhranie

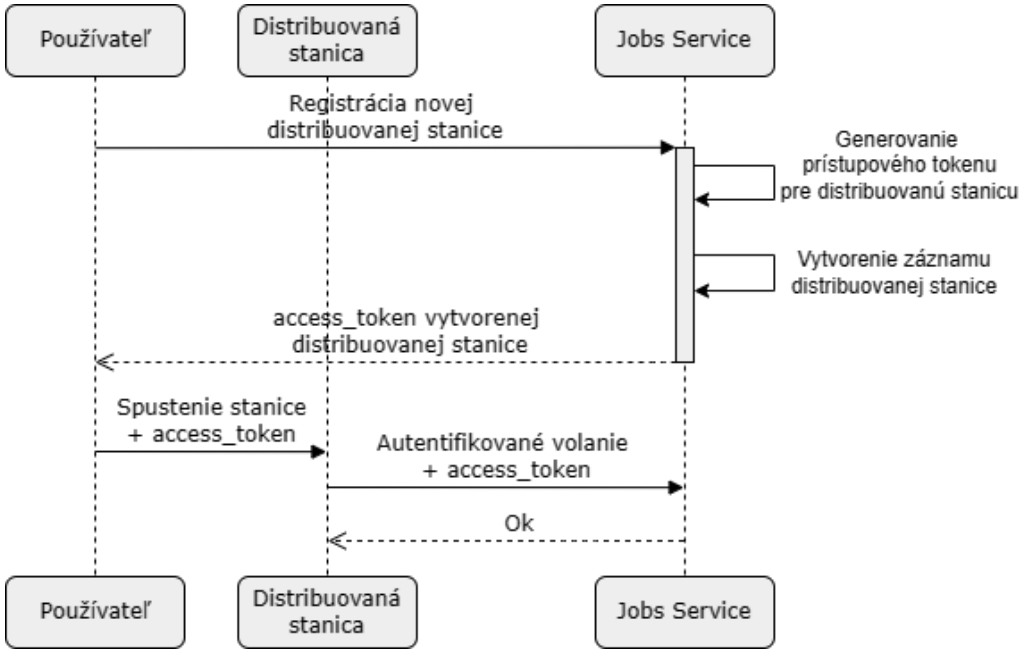
Každý z týchto nových komponentov je opísaný v nasledujúcich podkapitolách. V porovnaní s pôvodnou architektúrou z podkapitoly 1.1 si je možné všimnúť, že komponent *Služba pre manažment uzlov* (*Workers management service*) sa v novom návrhu architektúry nenachádza. Táto služba je **spojená** so *Službou distribuovaných úloh* (*Jobs service*), nakoľko doména pôvodnej služby je sub-doménou novej.

3.2.1 Služba distribuovaných úloh - Jobs service

Najväčšou zmenou v dizajne systému je práve táto mikroslužba. Ide o nový komponent v systéme, ktorého úlohou je **plánovanie**, **distribúcia** a **správa** distribuovaných úloh a distribuovaných staníc systému. Ako je zadefinované v sekciách 1.2.1, 1.2.2 a 1.2.3, táto služba musí poskytovať **generické riešenie** pre manipuláciu s distribuovanými úlohami a jednotlivé stanice musia spadať pod autorizačný mechanizmus, ktorý je v správe tejto služby.

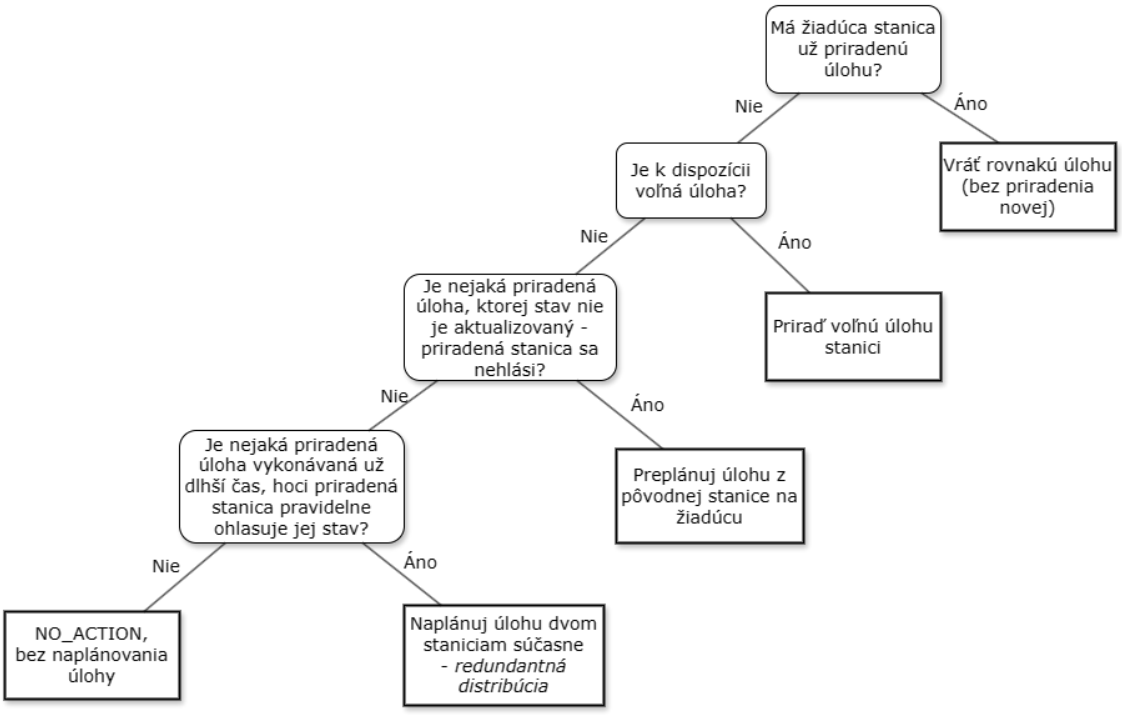
Z hľadiska funkcionalít správy distribuovaných staníc, táto služba ponúka prehľad aktuálneho stavu staníc a možnosť zaregistrovania nových staníc. Sekvenčný diagram na obrázku 3.2, znázorňuje registráciu novej distribuovanej stanice do systému a získanie autentifikačného tokenu, ktorý je neskôr priamo použitý na autentifikáciu distribuovanej stanice. Ide o rovnaký dvojfázový mechanizmus, aký používa systém *GitLab CI*, popísaný v sekcii 2.1.1. Tento mechanizmus pokrýva bezpečnostnú požiadavku, opísanú v sekcii 1.2.2.

Táto služba obsahuje aj implementáciu generického plánovača úloh. Je zodpovedná za správne naplánovanie a distribuovanie úloh medzi distribuované stanice systému. Úlohy sú zadávané inými komponentmi systému, a jednoznačne identifikované **unikátnym identifikátorom** a kategorizované podľa **typu úlohy**. Typ úlohy je dôležitý kvôli viacerým dôvodom. Napríklad, na strane zadávateľa úlohy je typ úlohy dôležitý pre správne filtrovanie udalostí o úlohách, ktoré táto



Obr. 3.2: Sekvencia procesu registrácie distribuovanej stanice v novom návrhu distribuovaného systému

služba generuje a publikuje. Na strane distribuovanej stanice je opäť typ úlohy dôležitý pre voľbu správneho exekútora úlohy. Keďže navrhnutý plánovač používa techniku *plánovania riadeného dopytom*, ktorá už bola čiastočne implementovaná v bakalárskej práci [4], plánovač vykonáva rozhodnutia o distribúcii vždy pri obdržaní žiadosti o úlohu od niektorej z distribuovaných staníc. Na obrázku 3.3 je zobrazený rozhodovací strom, ktorým sa riadi plánovač pri plánovaní a distribúcii úloh. Z dôvodu rozdielneho výkonu jednotlivých distribuovaných staníc poskytuje plánovač aj funkcionalitu **redundantných distribúcií**. V prípade, že jedna distribuovaná stanica je pomalšia a vykonáva úlohu príliš dlho, plánovač dokáže priradiť vykonávanie tej istej úlohy aj inej distribuovanej stanici. V takomto prípade vykonávajú úlohu 2 stanice súbežne, pričom vyhráva rýchlejšia. Služba taktiež notifikuje pomalšiu stanicu a určí jej predčasné ukončenie vykonávania, na základe určitého dôvodu - napríklad, že ju predbehla s vykonávaním iná distribuovaná stanica. Sekvencia akcií zrušenia vykonávania úlohy je rovnaká ako v špecifikácií požiadavky zo sekcie 1.2.3. Očakávanú sekvenciu tiež popisuje v danej kapitole obrázok 1.3. Obe prípady zadávania úlohy a oznamovanie výsledkov sú realizované pomocou internej komunikačnej vrstvy - v tomto prípade pomocou vymieňania správ cez službu *RabbitMQ*. Oznamovanie výsledkov úlohy riadi plánovač po obdržaní správy o úspešnom alebo neúspešnom vykonaní úlohy od priradenej stanice. Autorizačný mechanizmus zamedzuje nedovolenú manipuláciu iným staniciam, pokiaľ im úloha nie je explicitne priradená plánovačom.



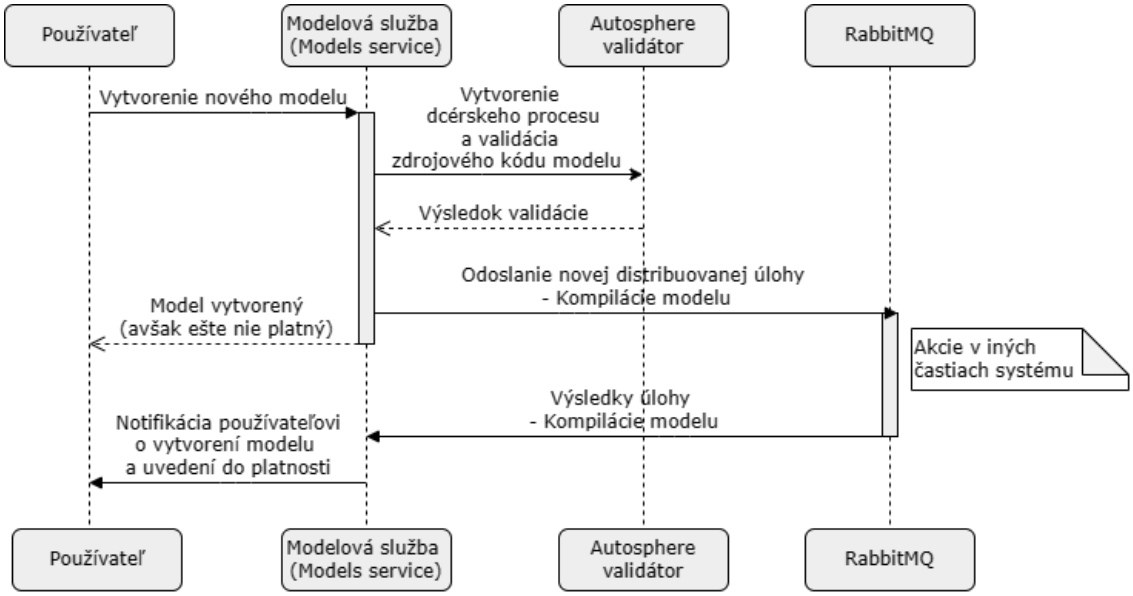
Obr. 3.3: Rozhodovací strom distribúcie úloh na úrovni distribuovaného plánovača úloh

3.2.2 Modelová služba - Models service

Druhým novým komponentom v tomto návrhu je *Modelová služba* (*Models service*). Jej doménou je **správa fyzikálnych modelov**, ktoré slúžia pre simulácie ako predloha - *blueprint*. Kvôli požiadavke o vlastných modeloch, zo sekcie 1.2.1, už systém nepracuje len s fixnou množinou modelov, ktoré sú priamo dostupné v Geliosphere. Architektúra mikroslužieb, na ktorej je systém postavený ešte z bakalárskej práce [4], umožňuje takéto rozšírenie veľmi jednoducho. Z analýzy požiadavky v sekcii 1.2.1 vyplýva, že navrhnutá služba musí poskytovať používateľom možnosť spravovať vlastné modely a používať ich pre ich simulácie. Okrem toho musí služba poskytovať **validačné mechanizmy**, a to hneď pre 2 typy operácií:

1. **Validácia modelu samotného**

   Pri vytváraní modelu je model overený dvoma krokmi, ako je to opísané v sekcii 1.2.1. Jeden z krokov, **validáciu**, dokáže služba vykonať ihneď, v kontexte používateľovej žiadosti. Druhý krok, **kompilácia**, je však asynchrónny, nakoľko operácia má vyššiu časovú náročnosť a vyžaduje distribuovanú stanicu. Tento krok musí byť teda vykonaný v rámci distribuovanej úlohy na jednej z distribuovaných staníc systému. Sekvenčný diagram na obrázku 3.4



Obr. 3.4: Sekvencia akcií vytvorenia vlastného modelu a validácie modelu v Modelovej službe

zobrazuje akcie vytvorenia modelu.

2. **Validácia parametrov simulácie podľa zvoleného modelu**

   Vlastné modely si definujú vlastné vstupne parametre pre simuláciu. V modeloch si používateľ vie definovať aj nejaké validačné podmienky pre každý parameter, ako napríklad najmenšia povolená hodnota, a podobne. Doménou tejto služby je aj poskytnúť vzdialenú validáciu parametrov pre zvolený model. Hneď ako je model vytvorený, je transformovaný do interných dátových štruktúr, takže prístup k definícii parametrov a ich validačným podmienkam je v rámci jedného procesu. Tento krok je tak veľmi rýchly a nevyžaduje si vytvárať distribuované úlohy. Táto operácia je zvyčajne volaná *Simulačnou službou* (*Simulations service*) pri vytvorení simulácie.

Nakoľko táto služba definuje a vytvára nový typ distribuovanej úlohy - **Kompilácia modelu**, je nutné ju integrovať so *službou pre distribuované úlohy*, zo sekcie 3.2.1. Vďaka generickému návrhu *služby pre distribuované úlohy* je integrácia jednoduchá. Pri vytváraní novej úlohy služba uvedie vyššie spomínaný typ. Takisto, pre získanie výsledkov zadaných úloh, služba aktívne počúva na udalosti pre svoj definovaný typ úlohy.

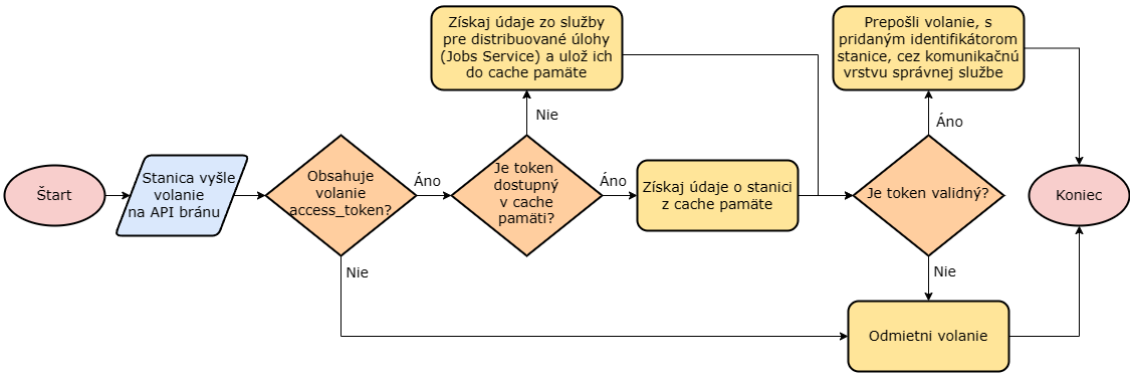
3.2.3 API brána pre distribuované uzly

Tento komponent je v návrhu obsiahnutý hlavne kvôli zlepšeniu **bezpečnosti systému**. Podľa analýzy požiadavky o bezpečnosti systému, zo sekcie 1.1.1, je vhodné

rozdeliť komunikačnú vrstvu systému na internú komunikáciu medzi jednotlivými centralizovanými komponentami a externú komunikáciu s distribuovanými stanicami. Služba *RabbitMQ* je preto v tomto návrhu použitá len na internú komunikáciu v centralizovanej vrstve systému. Distribuované stanice systému tak strácajú prístup ku komunikačnej vrstve a nedokážu komunikovať so zvyškom systému.

Pre tento prípad je navrhnutý komponent *API brány pre distribuované uzly*, ktorý poskytuje jediný prístupový bod systému pre komunikáciu distribuovaných staníc so zvyškom systému. API brána obsahuje bezpečnostné mechanizmy, ktoré pomáhajú chrániť systém a udržiavajú ho svižným. Jedná sa napríklad o mechanizmus **rate limiting**, ktorý je v článku [24] opísaný ako zásadný prvok ochrany voči viacerým typom útokov, ku ktorým patrí, napríklad, útok typu Denial of Service (DoS).

Pre odľahčenie práce *služby pre distribuované úlohy*, je na úrovni tohto komponentu realizovaná autentifikácia distribuovaných staníc. Pri neautorizovanom volaní brány je tak žiadosť zastavená hneď na prvej úrovni systému, ostatné backend služby a interná komunikačná vrstva nie sú zbytočne zaťažované. Autorizačné dáta si brána vyžiada zo *služby pre distribuované úlohy*, ktorá spravuje dáta o distribuovaných staniciach systému. Cez bránu prechádza veľké množstvo volaní zo všetkých distribuovaných staníc. Preto brána musí obsahovať efektívny cache mechanizmus, vďaka ktorému si (nielen) autentifikačné dáta dokáže dočasne uložiť lokálne. Vývojový diagram na obrázku 3.5 popisuje proces autentifikácie na úrovni API brány.



Obr. 3.5: Vývojový diagram procesu autentifikácie distribuovanej stanice na úrovni API brány

3.2.4 Frontend BFF pre webové používateľské rozhranie

Tento nový komponent zabezpečuje používateľom prístup k dátam systému. Je súčasťou vrchnej vrstvy, kde poskytuje špecializovaný prístup k dátam systému pre účely používateľského rozhrania - webovej aplikácie. Podobne ako API brána, aj tento komponent zlepšuje **bezpečnosť** a odolnosť systému, využitím podobných mechanizmov ako spomínaná API brána. Vďaka cache mechanizmu dokáže dáta krátkodobo uložiť, čím zníži zaťaženie backend služieb systému. Pre webovú aplikáciu taktiež **agreguje** dáta získané z volaní viacerých backendových komponentov. Dokáže taktiež pripraviť a vyrenderovať väčšinu statických častí webovej aplikácie.

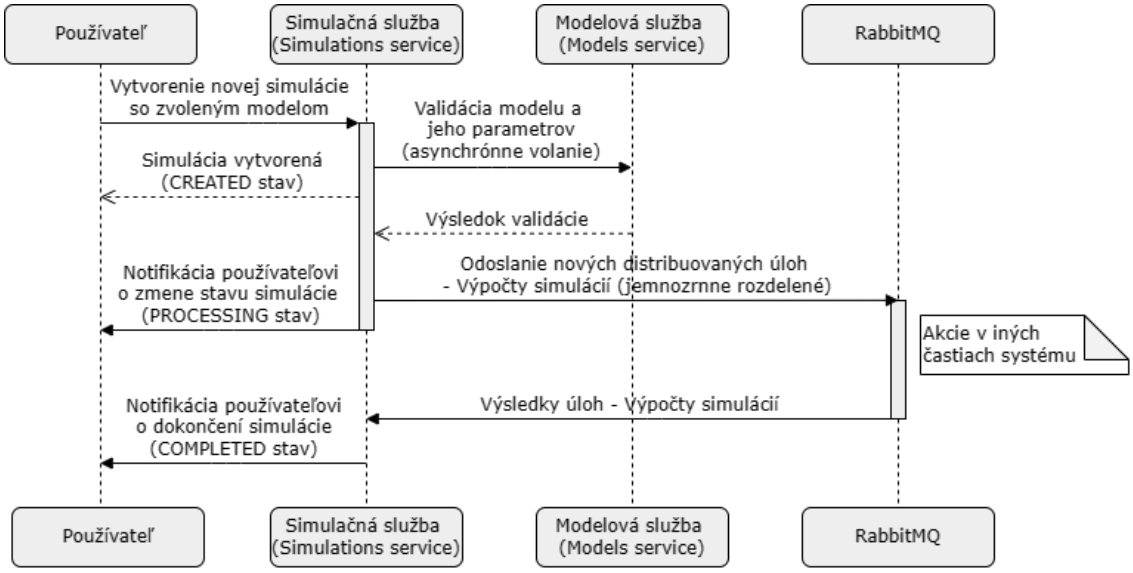
3.3 Zmeny v existujúcich komponentoch systému

Ako je spomínané v predošlých podkapitolách, optimalizácia architektúry si vyžaduje určité zmeny v existujúcich komponentoch systému. Pridaním nových komponentov do návrhu vznikla tiež potreba integrovať existujúce komponenty s tými novými. Okrem toho je iným dôvodom zmien splnenie niektorých požiadaviek, definovaných v podkapitole 1.2. Komponenty, ktoré si vyžadujú zmeny v ich návrhu a implementácii sú popísané v samostatných nasledujúcich podkapitolách.

3.3.1 Simulačná služba - Simulations service

Najväčšie zmeny postihli v novom návrhu práve *Simulačnú službu* (*Simulations service*). Kvôli podpore vlastných simulačných modelov, pre ktorú je navrhnutý samostatný komponent - *Modelová služba* (*Models service*), je potrebné čiastočne upraviť tok vykonávania v tejto službe. Ako bolo už čiastočne spomenuté v sekcii 3.2.2, je pri vytváraní simulácie potrebné validovať vstupné parametre. Validácia však presahuje doménu simulačnej služby, a teda je navrhnutá spoločne s logikou modelov v modelovej službe. Z hľadiska simulačnej služby je **validácia vykonávaná asynchrónne**, **vzdialeným volaním** modelovej služby.

Zo simulačnej služby je taktiež nutné vyňať prebytočnú logiku plánovania a distribúcie. V prototype z bakalárskej práce [4] existoval len 1 typ distribuovanej úlohy - *výpočet simulácie*, preto bola logika plánovania a distribúcie navrhnutá a implementovaná v rámci tejto služby. V tomto návrhu už však existuje samostatná služba - *Jobs service*, popísaná v sekcii 3.2.1, ktorej doménou je práve plánovanie a distribúcia úloh. Táto služba je v tomto návrhu adaptovaná na použitie vyššie



Obr. 3.6: Sekvencia akcií pri vytvorení novej simulácie v simulačnej službe

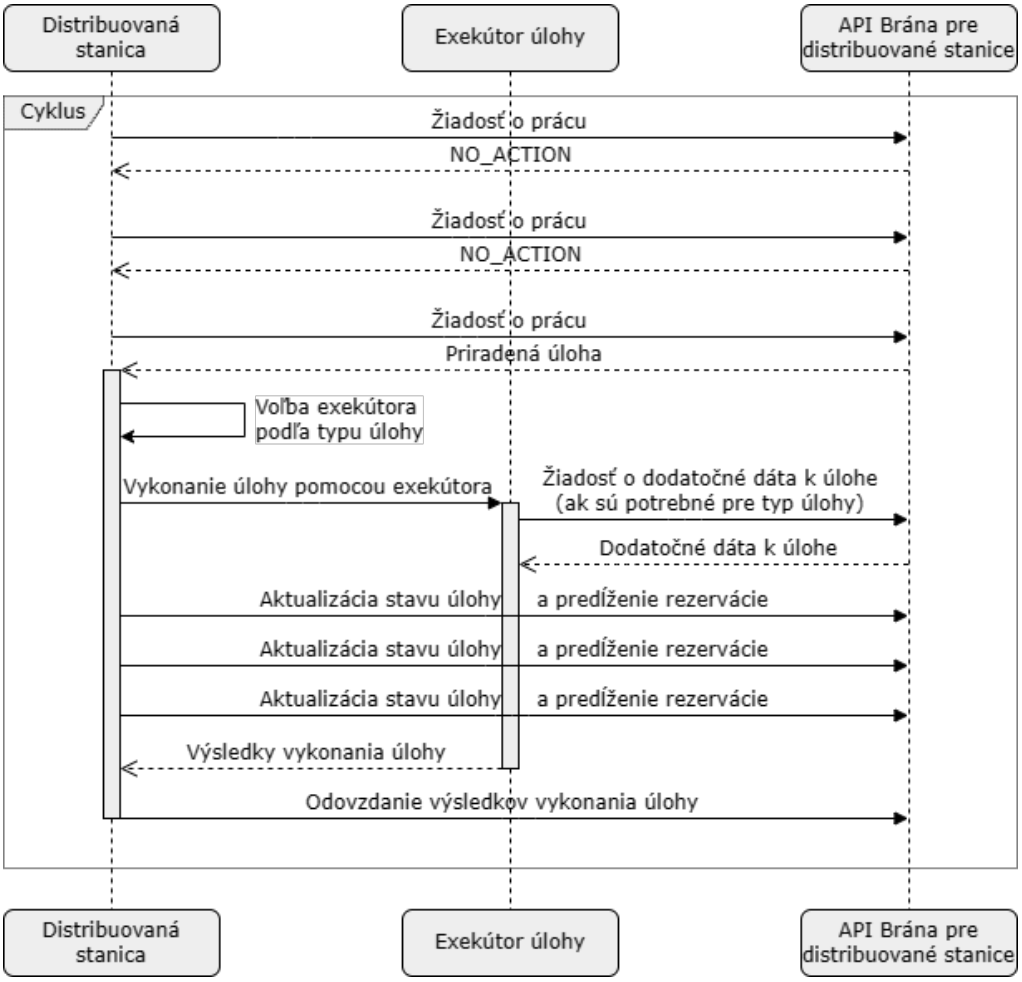
spomenutej služby. Simulačná služba definuje nový typ distribuovanej úlohy - **Výpočet simulácie** a vytvára (zadáva) úlohy daného typu. Pre získanie výsledkov zadaných úloh, služba aktívne počúva na udalosti pre svoj definovaný typ úlohy, rovnako ako *Modelová služba* zo sekcie 3.2.2. Sekvenčný diagram na obrázku 3.6 reprezentuje postupnosť akcií v systéme pri vytvorení novej simulácie.

Okrem vyššie spomínaných zmien, je pôvodne implementovaná doménová logika stále platná a zvyšné časti komponentu simulačnej služby zostávajú nezmenené.

3.3.2 Distribuovaný uzol systému - Distributed Worker

Návrh počíta aj so zmenami na úrovni jednotlivých distribuovaných uzloch systému. V pôvodnej implementácií distribuovaných staníc bola logika vykonávania jednoduchá, nakoľko existoval len jeden typ distribuovanej úlohy - *Výpočet simulácie*. Stanica tak mohla ihneď interagovať s programom Geliosphere, vypočítať simuláciu a odovzdať výsledky výpočtu, resp. chybu, ak pri vykonávaní nejaká nastala. V tomto návrhu však existuje viacero typov úloh. Pre každý typ úlohy musí distribuovaná stanica počas behu programu dynamicky zvoliť správneho **exekútora úlohy**, podobne, ako je to implementované na platforme *GitLab*, ktorá je opísaná v podkapitole 2.1. Proces vykonávania po obdržaní úlohy v upravenom návrhu distribuovanej stanice systému je popísaný v časti sekvenčného diagramu na obrázku 3.7.

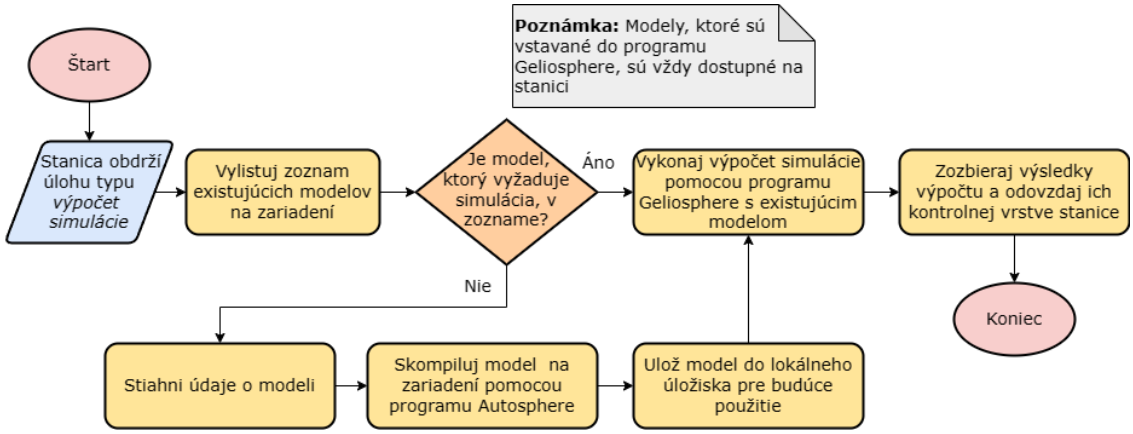
Návrh taktiež pozmeňuje formu komunikácie s centralizovanou vrstvou. Nakoľko nie je možný externý prístup do internej komunikačnej vrstvy, tak stanica



Obr. 3.7: Sekvencia hlavného vykonávacieho cyklu distribuovanej stanice systému podľa optimalizovaného návrhu

stráca prístup do služby *RabbitMQ*. Namiesto toho je stanica nútená komunikovať pomocou API brány, ktorá je navrhnutá v sekcii 3.2.3. Navrhnutý mechanizmus je inšpirovaný platformou *GitLab*, ktorá používa podobný mechanizmus, ako je popísané v sekcii 2.1.2. Periodickými volaniami, si stanica aktívne pýta prácu. Následne, počas vykonávania úlohy, stanica **periodicky reportuje svoj stav** a obnovuje si rezerváciu úlohy (uzamknutie z pohľadu plánovača). Hlavný vykonávací cyklus distribuovanej stanice podľa nového návrhu je popísaný sekvenčným diagramom na obrázku 3.7.

Osobitnú pozornosť je nutné venovať návrhu exekútora pre typ úlohy **výpočtu simulácie**. Kvôli požiadavke o *vlastných simulačných modelov*, zo sekcie 1.2.1 je možné, že v čase priradenia úlohy a jej vykonávania sa na danej distribuovanej stanici nemusí skompilovaný model nachádzať. V takom prípade je nutné najprv **stiahnuť metadáta** modelu a **skompilovať** ho pred samotným výpočtom simulácie. Okrem toho zmena nastala aj v integrácii s programom Geliosphere, tak



Obr. 3.8: Vývojový diagram vykonávania úlohy typu výpočet simulácie na distribuovanej stanici, podľa nového návrhu

ako je to popísané v sekcii 1.2.5. Geliosphere už nie je spustená v interaktívnom móde pri spustení stanice, ako to bolo v prototype z bakalárskej práce [4], ale namiesto toho je pri každom výpočte spustený vždy nový dcérsky proces. Presný postup exekútora pre spomínaný typ úlohy je popísaný vývojovým diagramom na obrázku 3.8.

3.4 Návrh aktualizovaného používateľského rozhrania

Vlastnú kapitolu v návrhu zohráva aj používateľské rozhranie, keďže je to dané požiadavkou zo sekcie 1.2.4. Realizácia používateľského rozhrania je rovnaká ako v pôvodnom prototype systému [3], **prostredníctvom webovej aplikácie**. Navrhnutá webová aplikácia obsahuje 3 hlavné časti, a to konkrétne:

- **Správa distribuovaných staníc**

  Táto časť aplikácie spravuje distribuované stanice. Používateľ je schopný zaregistrovať (vytvoriť) nové stanice do systému a taktiež si prezrieť zoznam existujúcich staníc v systéme. Zoznam existujúcich staníc zobrazuje ich aktuálny stav, posledný čas aktualizácie a ďalšie metadáta, ako napríklad názov, popis, a podobne.

  Po zaregistrovaní novej stanice je používateľovi poskytnutý **autentifikačný token**, slúžiaci na autentifikáciu danej stanice, rovnako, ako je to na oboch analyzovaných platformách *GitLab* a *GitHub*, v podkapitolách 2.1 a 2.2. Táto funkcionalita je úzko prepojená s požiadavkou o autentifikácii a autorizácii distribuovaných staníc zo sekcie 1.2.2.

- **Správa simulačných modelov**

  V tejto časti aplikácie používateľ spravuje dostupné simulačné modely. Okrem zoznamu dostupných modelov v systéme a ich aktuálnom stave, si používateľ môže **pridať nové**, vlastné **modely**. Podľa návrhu táto časť aplikácie umožňuje používateľovi vytváranie modelov a následné obdržanie notifikácií o stave modelu.

  Po úspešnom vytvorení modelu je model pripravený na použitie - používateľ si vie vybrať daný model pri neskoršom vytváraní simulácií. Táto časť aplikácie spĺňa používateľskú interakciu so správou vlastných modelov, podľa definície zo sekcie 1.2.1.

- **Správa simulácií**

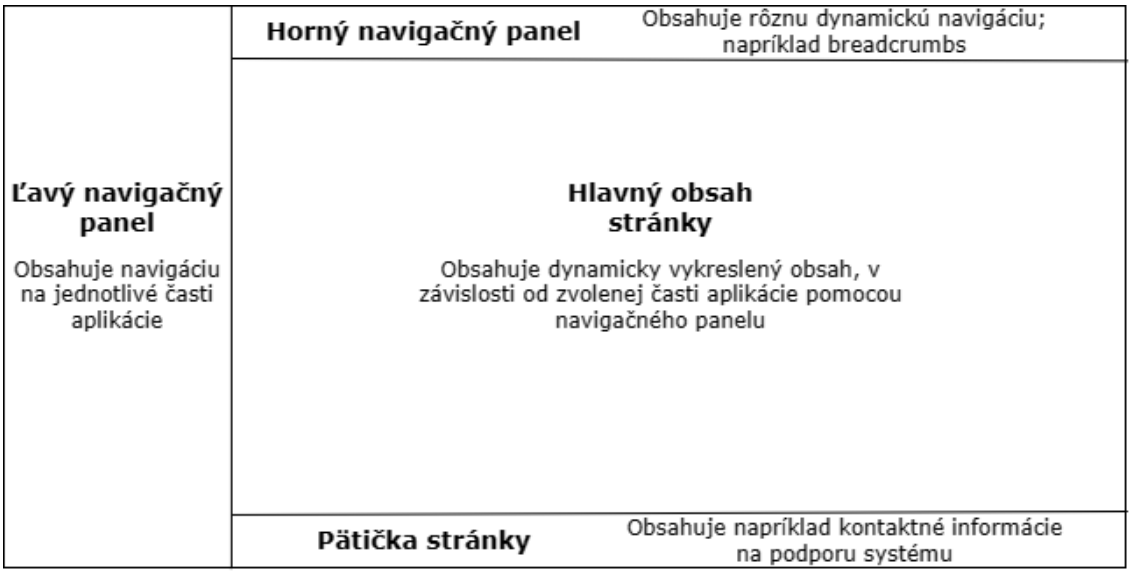
  Najhlavnejšou časťou aplikácie, na základe domény systému, je *správa simulácii*. Jednotlivé stránky tejto časti aplikácie sú inšpirované pôvodným používateľským rozhraním z [3]. Poskytujú prehľad simulácií, detailný prehľad zvolenej simulácie, zobrazenie výsledkov dokončených simulácií a vytváranie nových simulácii. Rozdielom je však spôsob vytvárania novej simulácie. V novom návrhu si používateľ najprv musí zvoliť jeden z existujúcich modelov, ktoré budú použité pri výpočte simulácie. Na základe toho je potom zobrazený formulár pre vytvorenie simulácie, ktorý obsahuje správne parametre, definované v danom modeli.

  Po vytvorení simulácie vie používateľ sledovať jej aktuálny stav, keďže väčšina akcií je vykonávaných asynchrónne, mimo kontextu jednej HTTP žiadosti. Kvôli *jemnozrnnej distribúcii*, ktorej sa venovala bakalárska práca [4] je simulácia rozdelená na mnoho menších častí, ktoré sú počítané nezávisle od seba. Preto má používateľ možnosť taktiež sledovať priebežný stav výpočtov celej simulácie na stránke detailného prehľadu simulácie.

Navrhnutá webová aplikácia komunikuje s internými službami systému výhradne pomocou jej brány, podľa vzoru Backend for Frontend (BFF)[13]. Používa na to nový komponent, ktorý je navrhnutý v sekcii 3.2.4 - *Frontend BFF*. Takýmto návrhom je čiastočne **zlepšená bezpečnosť** systému a taktiež používateľská skúsenosť, kvôli cache mechanizmom popísaným v spomínanej kapitole. Vďaka **agregácií** dát z viacerých volaní, ktoré poskytuje *Frontend BFF* komponent, sa používateľ a navrhnutá webová aplikácia rýchlejšie a jednoduchšie dostane ku koncovým dátam, ktoré sú upravené pre špecifické potreby tejto aplikácie.

Z hľadiska rozloženia elementov a komponentov na úrovni stránok aplikácie, je navrhnuté používateľské rozhranie podľa predlohy pôvodného prototypu z [3],

upravené podľa teoretického dizajnu, ktorý Solanik navrhol v [5]. Konceptuálne rozloženie finálneho návrhu je zobrazené na obrázku 3.9.



Obr. 3.9: Konceptuálne rozloženie komponentov v aktualizovanom návrhu používateľského rozhrania - webovej aplikácie

4 Implementácia systému podľa návrhu optimalizovanej architektúry

Táto kapitola sa venuje implementačným detailom jednotlivých komponentov systému, ktoré sú implementované na základe návrhu optimalizovanej architektúry z kapitoly 3. Pri implementovaní nového návrhu je kladený dôraz na zachovanie čo najviac existujúcej implementácie komponentov z pôvodného prototypu z bakalárskej práce [4], pokiaľ nie sú v rozpore s novým návrhom. Nasledujúci zoznam ponúka zhrnutie implementačných detailov a použitých technológií v prototype systému z bakalárskej práce:

- Všetky komponenty systému sú implementované v jazyku `C#` pre platformu `.NET`.
- Komponenty sú implementované ako nadstavba nad distribuovaným aplikačným rámcom *MassTransit*[25].
- Perzistenciu dát pre každú centralizovanú službu zabezpečuje SQL databáza *Microsoft SQL Server* (*MS-SQL*).
- Prístup k databáze je na každej centralizovanej službe realizovaný pomocou rámca *Entity Framework Core*[1].
- Systém nie je inštrumentovaný z hľadiska telemetrie a neposkytuje telemetrické dáta.

Systém obsahuje niekoľko implementačných zmien na globálnej úrovni. Prvou globálnou zmenou je zmena databázového riešenia z MS-SQL na **PostgreSQL**. Hlavným dôvodom pre túto zmenu je rozdielna licenčná politika riešení. Systém *PostgreSQL* je licencovaný pod *open-source* licenciou, zatiaľ čo plná verzia systému *MS-SQL* je licencovaná pod komerčnou, platenou licenciou. Hľadiac na

[1] `https://learn.microsoft.com/en-us/ef/core/`

zameranie tohto systému je vhodnejšie sa vyhnúť komerčným, plateným riešenia, pokiaľ je to možné. Podľa článkov [26] a [27], ktoré porovnávajú a analyzujú spomínané databázové systémy, je možné konštatovať, že *PostgreSQL* je rovnako vyspelý systém ako *MS-SQL*. Oba systémy sú dlhoročne používané ako databázové riešenia v enormnom množstve aplikácií. Zmenou databázového riešenia na *PostgreSQL* systému nevzniknú žiadne komplikácie a nevýhody.

Kvôli zmene databázového riešenia je nutné taktiež zmeniť **ovládač**, pomocou ktorého rámec *Entity Framework Core* komunikuje s databázou. Našťastie, tento rámec poskytuje dostatočnú abstrakciu nad databázou, takže je potrebné len zameniť samotný ovládač a vygenerovať databázové migrácie pre nový databázový systém. Rámec použije nový ovládač a na základe entitného modelu vygeneruje kompatibilné migračné skripty pre dané databázové riešenie.

Druhou zmenou na globálnej úrovni je pridanie podpory **telemetrie** a publikovanie **telemetrických dát** v systéme. Správcom systému poskytnú telemetrické dáta lepší prehľad nad celkovým stavom systému a vystopovateľnosť jednotlivých akcií v systéme. Pre inštrumentáciu jednotlivých komponentov je použitý štandard **OpenTelemetry** [28]. *OpenTelemetry* poskytuje viacero inštrumentačných knižníc, ktoré dokážu automaticky inštrumentovať aplikáciu a generovať telemetrické dáta [28]. Tento fakt prináša obrovskú výhodu v existujúcom systéme, nakoľko nie je potrebné upravovať existujúci kód. Pre automatickú inštrumentáciu sú využité nasledujúce knižnice (vo forme `NuGet` balíčkov):

- **OpenTelemetry.Instrumentation.AspNetCore** - Zbiera metriky a distribuované stopy o prichádzajúcich HTTP žiadostiach.
- **OpenTelemetry.Instrumentation.Http** - Zbiera metriky a stopy o odchádzajúcich HTTP žiadostiach.
- **OpenTelemetry.Instrumentation.Runtime** - Zbiera metriky o využití zdrojov procesu a platformy.
- **Npgsql.OpenTelemetry** - Zbiera metriky a stopy pri práci s *PostgreSQL* databázou.
- **MassTransit** - Zbiera metriky a stopy pri vymieňaní a spracovávaní správ a komunikácii so sprostredkovateľom správ - *RabbitMQ*.

Lokálne zmeny v implementáciách existujúcich komponentov, resp. implementačné detaily nových komponentov v systéme sú popísané v nasledujúcich samostatných podkapitolách.

4.1 Implementácia služby pre distribuované úlohy - Jobs service

Pri implementácii tejto služby je dôležité popísať jej najhlavnejší bod - distribuované úlohy. Jednotlivé distribuované úlohy sú implementované na základe konceptu **konečno-stavových automatov**. Podľa [29] môže koncept *konečno-stavových automatov* výrazne zjednodušiť a zefektívniť implementáciu tejto problematiky. Dokonca rámec MassTransit má priamo zabudovanú podporu distribuovaných ság udalostí, ktoré sú definované pomocou konečno-stavových automatov[25]. Na základe deklaratívnej definície stavov a prechodov automatu, v našom prípade distribuovanej úlohy, dokáže MassTransit na základe korelovaných udalostí orchestrovať konečno-stavový automat automaticky. Zdrojový kód 4.1 znázorňuje zjednodušenú ukážku deklaratívnej definície konečno-stavového automatu pre distribuovanú úlohu.

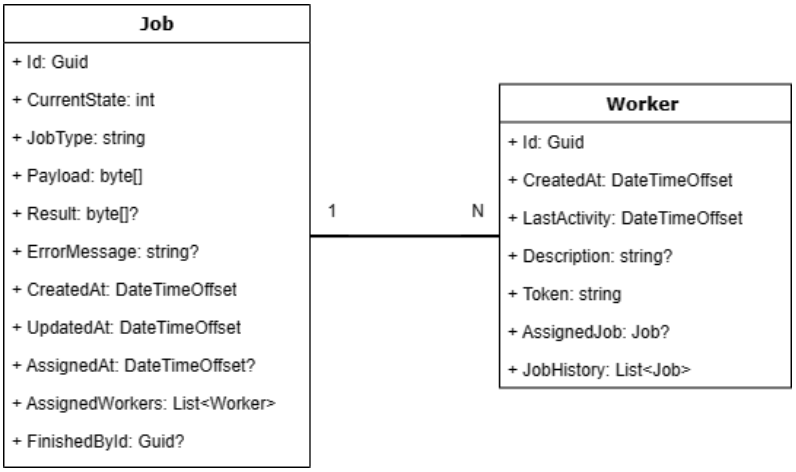
Zdrojový kód 4.1: Zjednodušená deklaratívna definícia konečno-stavového automatu pre entitu distribuovanej úlohy

```
public class JobStateMachine : MassTransitStateMachine<Job>
{
    // States
    public State Queued { get; set; }
    public State InProgress { get; set; }
    public State Completed { get; set; }
    public State Failed { get; set; }

    // Events
    public Event<EnqueueJob> EnqueueJob { get; set; }
    public Event<CompleteJob> JobCompletion { get; set; }

    public JobStateMachine()
    {
        // Behaviours
        Initially(When(EnqueueJob).TransitionTo(Queued));
        During(InProgress, JobLeaseProlongationHandler,
            JobCompletionHandler);

        // Other behaviours & configuration ...
    }
}
```



Obr. 4.1: Doménový model služby pre distribuované úlohy

Implementovaný plánovač úloh využíva použitého mechanizmy rámca MassTransit, vďaka ktorým je jeho implementácia zjednodušená. Prijaté správy dokáže pomocou vyššie spomínanej deklaratívnej konfigurácie MassTransit správne spárovať s inštanciou konečno-stavového automatu úlohy. Dokonca poskytuje aj integráciu s rámcom *Entity Framework Core*, pomocou ktorého dokáže automaticky získať jednotlivé entity úloh z databázy a po vykonaní deklarovaných akcií a prechodoch konečno-stavového automatu ich aj naspäť do databázy uložiť.

Entita distribuovanej úlohy je generická, bez viazanosti na doménovú logiku alebo dátové typy nejakého konkrétneho typu úlohy. Distribuovaná úloha obsahuje atribút `JobType`, ktorý kategorizuje úlohu podľa určitého typu. Tento atribút je veľmi dôležitý, nakoľko sa používa pre správne trasovanie a filtrovanie úloh. Okrem toho úloha obsahuje aj jedinečný identifikátor `ID`, ktorý ju jednoznačne identifikuje. Keďže vytváranie úloh je k dispozícii len interným komponentom systému, tvorbu identifikátoru je bezpečné zveriť zadávateľom úlohy. Vďaka tomu si dokážu priamo držať referenciu na úlohu, ktorú vytvorili. Telo úlohy (ak nejaké existuje), musí byť **serializované** ako platný `JSON` reťazec. Táto služba nepotrebuje poznať telo úlohy a preto ho nikdy **nedeserializuje**. Na obrázku 4.1 je zobrazený doménový model tejto služby.

Implementovaný plánovač v tejto službe obsahuje niekoľko konfigurovateľných parametrov, ktoré modifikujú jeho predvolené správanie. Všetky konfiguračné parametre, ktorými je možné ovplyvniť správanie plánovača, sú popísané v tabuľke 4.1.

Názov parametra	**Popis parametra**	**Predvolená hodnota**
UpdateFrame	Definuje jedno okno aktualizácie stavu úloh v plánovači (v sekundách)	10
UpdateMaxTimeout	Definuje maximálny časový interval, po ktorom je úloha považovaná za stratenú (v sekundách)	Trojnásobok hodnoty `UpdateFrame`
JobGracePeriod	Definuje časový interval, počas ktorého nie je možné pre úlohu aplikovať *redundantnú distribúciu* (v sekundách)	180

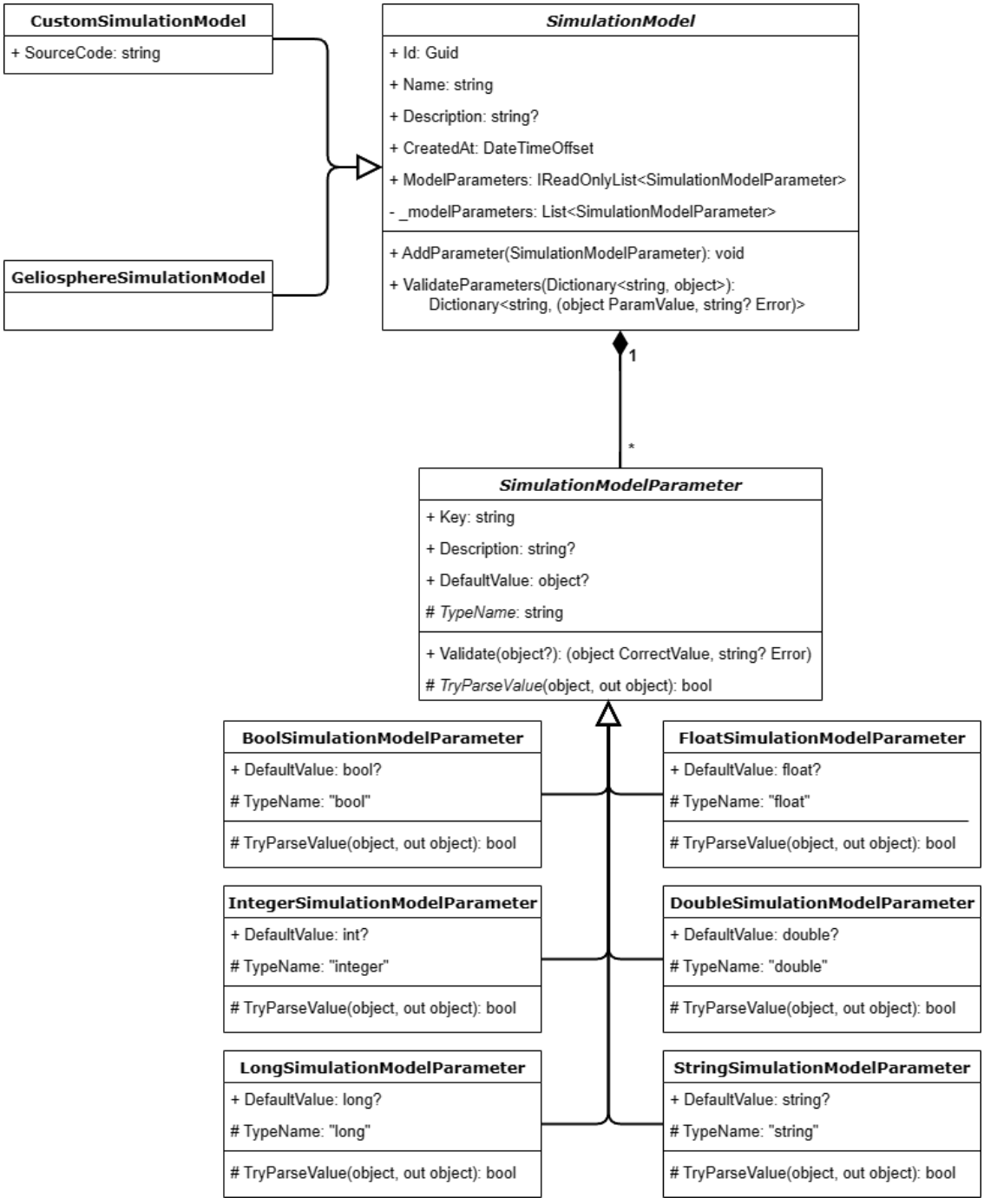
Tabuľka 4.1: Zoznam konfiguračných parametrov pre implementovaný plánovač distribuovaných úloh

4.2 Implementácia modelovej služby

Nová modelová služba je implementovaná na základe konvencií v systéme. Dodržiava implementačný štýl, používa rovnaké metódy a technológie ako ostatné služby. Špecifickým implementačným detailom je integrácia s programom *Autosphere*[5], ktorý využíva pri spracovávaní žiadosti na vytvorenie nového, vlastného modelu. Podľa návrhu zo sekcie 3.2.2 je program *Autosphere* riadený touto službou, z hľadiska procesov vzťahom rodič-dieťa. Táto služba využíva dcérsky proces aplikácie *Autosphere* na analýzu a validáciu zdrojového kódu potenciálne nového modelu, písaného v *doménovo-špecifickom jazyku*, ktorý samotná služba nedokáže sama spracovať.

Z pohľadu distribuovaných úloh, táto služba je správcom jedného typu úlohy, ako to je opísané v jej návrhu v sekcii 3.2.2. Typovým kľúčom (atribút `JobType`) tejto úlohy je reťazec `ModelCompilation`. Služba počúva na udalosti v systéme o distribuovaných úlohách, pričom spracováva len také, ktoré sú odfiltrované na základe vyššie spomenutého reťazca.

Pre následnú validáciu parametrov simulácie, po tom čo je model označený ako **platný** a uvedený do prevádzky, používa služba internú logiku priamo vo svojom procese. Entita simulačného modelu obsahuje všetky potrebné detaily a implementácia validačného mechanizmu dokáže na základe doménového modelu vykonávať validáciu parametrov pre simulačný model priamo v procese služby. Doménový model tejto služby je zobrazený na obrázku 4.2.



Obr. 4.2: Doménový model modelovej služby

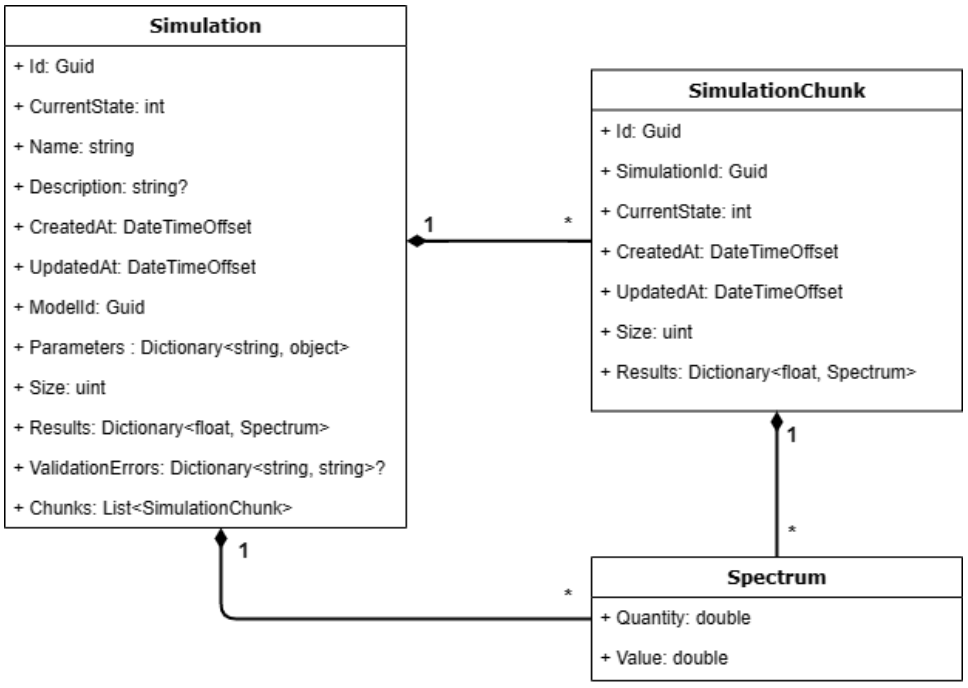
4.3 Implementačné zmeny v simulačnej službe

V tejto službe sú vykonané len minimálne implementačné zmeny. Logika delenia simulácie na jemnozrnné výpočty zostáva pôvodná z bakalárskej práce [4]. Zmena nastala v samotnej distribúcii a plánovaní úloh. Ako bolo spomínané v návrhu tejto služby v sekcii 3.3.1, mechanizmus plánovania a distribúcie je extrahovaný z tejto služby do *služby pre distribuované úlohy*. Na základe toho je implementácia čiastočne pozmenená a táto služba je integrovaná s vyššie spomenutou službou.

Podobne ako modelová služba z predchádzajúcej sekcie, aj táto služba spravuje svoj typ úlohy, ako je to opísané v jej návrhu, v sekcii 3.3.1. Typovým kľúčom (atribút `JobType`) úlohy je reťazec `SimulationChunkCalculation`. Táto služba rovnako počúva na udalosti systéme o distribuovaných úlohách a spracováva len odfiltrované udalosti podľa jej typového kľúča - reťazca.

Služba taktiež využíva princíp konečno-stavových automatov na zefektívnenie implementácie simulácií a jej čiastkových výpočtov. Využíva rovnaké mechanizmy rámca MassTransit ako *služba pre distribuované úlohy*, ktoré sú opísané v podkapitole 4.1.

Okrem vyššie popísaných zmien ostáva implementácia tejto služby z prototypu z bakalárskej práce [4] nezmenená. Obrázok 4.3 reprezentuje aktualizovaný doménový model simulačnej služby.



Obr. 4.3: Doménový model simulačnej služby

4.4 Implementácia API brány pre distribuované stanice

Komponent *API brány pre distribuované uzly*, ako je popísané v návrhu v sekcii 3.2.3, slúži ako *middleware* medzi internou komunikačnou vrstvou systému, ktorý zabezpečuje služba *RabbitMQ*, a externou komunikáciou s distribuovanými stanicami systému. Keďže sa jedná o *webovú službu*, na jej implementáciu je použitý štandardný webový rámec, ktorý platforma `.Net` poskytuje - **ASP.NET Core**[2].

Pre zabezpečenie komunikácie je zvolený rámec **gRPC**[3], práve kvôli jeho rýchlosti. Oproti tradičným *JSON serializovaných HTTP* službám sú služby *gRPC* niekoľkonásobne rýchlejšie. Dokazujú to aj nedávne výsledky z článku [30], ktorý porovnával práve tieto typy služieb, najmä s ohľadom na rýchlosť. Webový rámec *ASP.NET Core* zahŕňa podporu *gRPC* služieb. Na serializáciu správ používa rámec *gRPC* mechanizmus **Protobuf**[4]. Výhodou tohto mechanizmu je deklaratívny zápis typov správ, resp. služieb a následná podpora automatického generovania kódu na oboch stranách - klienta aj serveru. Na definovanie typov správ a služieb sa používajú v špecifické *Protobuf* - `.proto` súbory. Definícia *gRPC* služby *API brány* je v tomto systéme je obsiahnutá v súbore `workers.gateway.proto`. Tento súbor obsahuje zdrojový kód 4.2.

Zdrojový kód 4.2: Definícia gRPC služby API brány pre distriuované stanice

```
syntax = "proto3";
package CudaHelio.Workers.ApiGateway.Proto;
import "workers.messages.proto";

service WorkersGateway {
  rpc RequestAJob (RequestAJobRequest)
    returns (RequestAJobResponse);
  rpc ProlongJobLease (ProlongJobLeaseRequest)
    returns (ProlongJobLeaseResponse);
  rpc UploadJobResults (UploadJobResultsRequest)
    returns (UploadJobResultsResponse);
  rpc GetModelById (GetModelByIdRequest)
    returns (GetModelByIdResponse);
}
```

[2] https://dotnet.microsoft.com/en-us/apps/aspnet
[3] https://grpc.io/
[4] https://protobuf.dev/

Komunikáciu pomocou *gRPC* potom brána správne mapuje a preposiela na internú komunikačnú vrstvu.

Táto služba zabezpečuje aj autentifikáciu distribuovaných staníc, ako je to popísané v návrhu zo sekcie 3.2.3. Implementácia autentifikačného procesu je taktiež zabezpečená webovým rámcom *ASP.NET Core*. Kvôli upravenému formátu tokenu, ktorý začína prefixom `chworker-`, je implementovaný špecifický autentifikačný mechanizmu, ktorý pri procese autentifikácie rámec používa. Pre zníženie počtu dopytov na *službu pre distribuované úlohy*, ktorá spravuje okrem iného aj autentifikačné dáta jednotlivých staníc, obsahuje *API brána* implementáciu vyrovnávacej cache pamäte, pomocou rozhrania `IDistributedCache`. V predvolenej konfigurácii je použitá na dočasné ukladanie stavu **lokálna pamäť** (angl. **memory cache**). Vďaka flexibilite rozhrania je možné však implementáciu kedykoľvek zameniť za iný typ **distribuovanej vyrovnávacej pamäte** - napríklad *Redis*[5]. Exspirácia dát vo vyrovnávacej pamäti je nastavená na **5 minút** od posledného použitia (tzv. *sliding expiration*).

4.5 Implementačné zmeny na distribuovanom uzle

Implementácia *distribuovaného uzla* systému bola, kvôli novému návrhu zo sekcie 3.3.2 rozdelená na 2 projekty - `DistributedWorker.Controller`, ktorý zabezpečuje kontrolné mechanizmy a komunikáciu s centralizovanou vrstvou systému a `DistributedWorker.Executor`, ktorý obsahuje doménové logiky na vykonanie jednotlivých typov distribuovaných úloh.

Implementácia prvého zo spomínaných projektov je len čiastočne upravená od pôvodnej implementácie z bakalárskej práce [4]. Prvou nutnou zmenou v tomto projekte je zmena komunikačného rozhrania, z pôvodnej priamej integrácie so službou *RabbitMQ*, na rozhranie *API brány*, popísanej v predošlej sekcii, a teda na rámec *gRPC*. Sú použité aj rovnaké definície *gRPC* služieb, pomocou tých istých *Protobuf* súborov. Vďaka automatickému generovaniu kódu na strane klienta je zmena rozhrania jednoducho realizovateľná, bez väčších komplikácií. Podľa nového návrhu, samozrejme, musí stanica pracovať aj so svojím autentifikačným tokenom, ktorý jej je dodaný pri štarte. Po obdržaní distribuovanej úlohy je táto časť distribuovaného uzla zodpovedná za výber správneho exekútora a delegovanie vykonávania úlohy na zvoleného exekútora.

Implementácia časti exekútora obsahuje logiku vykonávania pre každý typ distribuovanej úlohy, ktorú rôzne časti systému definujú. Na tomto mieste sa tak-

[5] `https://redis.io/`

tiež nachádzajú **integrácie s dcérskymi procesmi** aplikácií *Geliosphere* a *Autosphere*, v závislosti od vykonávanej úlohy. Pôvodný interaktívny mód, v ktorom bol pôvodne spúšťaný program *Geliosphere* je nahradený jednoduchším modelom vytvorenia procesu pri vykonávaní úlohy, tak ako je to popísane v sekcii 1.2.5. Po dokončení, resp. predčasnom zrušení vykonávania úlohy je dcérsky proces ukončený.

Distribuovaná stanica obsahuje niekoľko konfiguračných parametrov, ktoré sú popísané v tabuľke 4.2. Konfiguračné parametre, ktoré neobsahujú predvolenú hodnotu, sú povinné a bez ich zadania stanica vyhodí chybu hneď pri štarte a ukončí sa.

Názov parametra	**Popis parametra**	**Predvolená hodnota**
ServerUri	Definuje adresu *API brány*, pomocou ktorej uzol komunikuje so zvyškom systému	-
Token	Definuje autentifikačný token uzla, slúžiaci na overenie identity a autorizáciu uzla	-
UpdateFrame	Definuje časový interval medzi jednotlivými žiadosťami o predlženie rezervácie úlohy (v sekundách)	10
HasGpu	Indikuje, či uzol môže používať na vykonávanie úloh aj GPU, alebo len CPU	true
TestMode	Aktivuje testovací mód, v ktorom sú použité **mock** implementácie exekútorov namiesto reálnych (*Poznámka: Tento konfiguračný parameter je účinný len v prípade, keď je aplikácia skompilovaná a spustená v DEBUG konfigurácii*)	false

Tabuľka 4.2: Zoznam konfiguračných parametrov pre distribuovaný uzol systému

4.6 Implementácia komponentu Frontend BFF a používateľského rozhrania - Webovej aplikácie

Implementácia týchto komponentov, podľa ich návrhov v sekcii 3.2.4 a v podkapitole 3.4, sú realizované spoločne, v rámci jednej aplikácie. Na jej implementáciu je

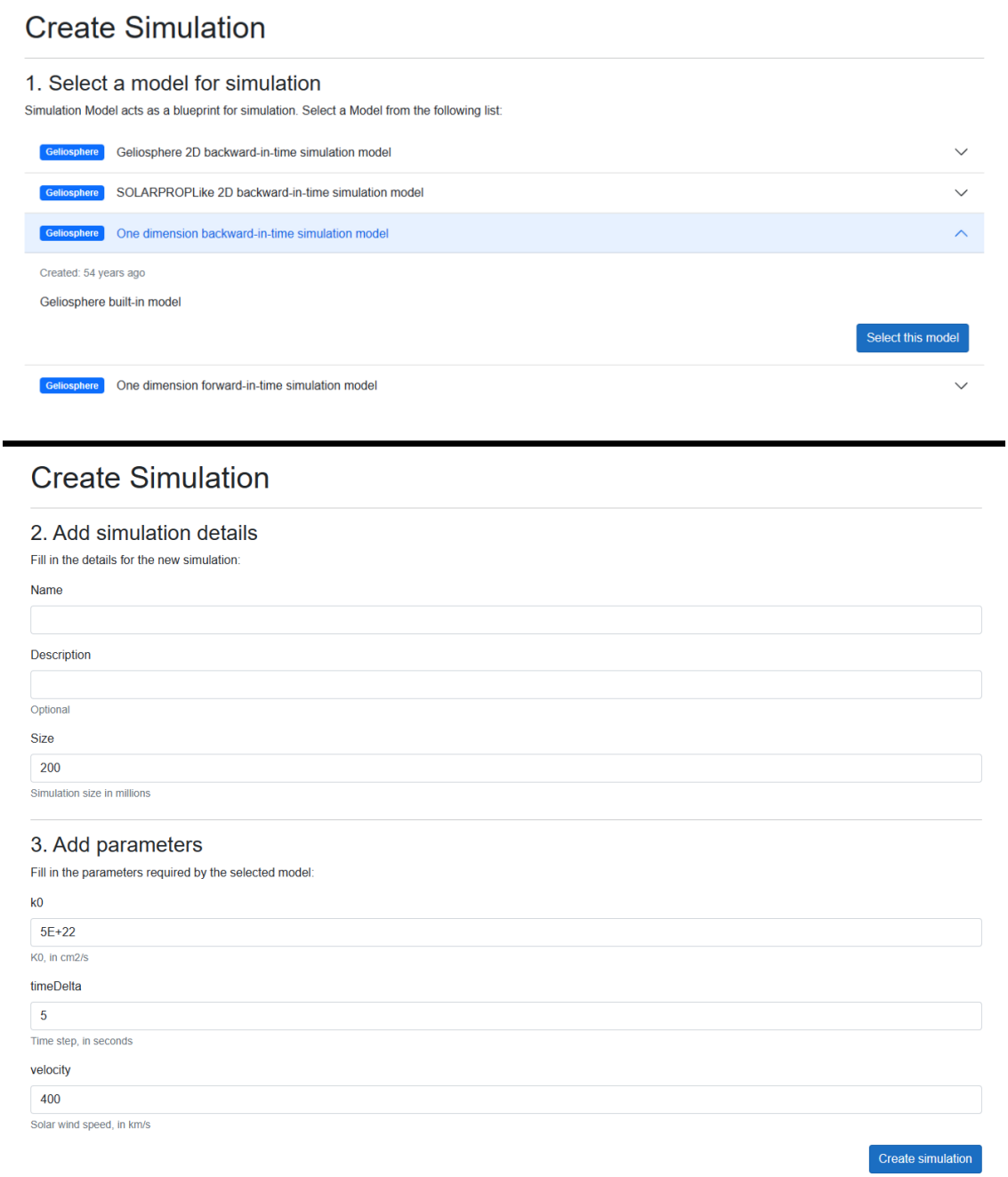
použitý rámec **Blazor**[6], ktorý je súčasťou webového rámca *ASP.NET Core*. Tento rámec poskytuje možnosť tvorby dynamických používateľských rozhraní, ktoré sú hybridne renderované, najmä na serveri, ale čiastočne aj na strane klienta. Výhodou takéhoto hybridného prístupu je odľahčenie klienta a nízke požiadavky na hardvér na zariadeniach klienta - používateľa.

Serverový komponent má navyše **prístup k internej komunikačnej sieti systému**, vďaka čomu dokáže priamo pomocou služby *RabbitMQ* komunikovať so zvyškom systému. Taktiež dokáže počúvať na vybrané typy udalostí v systéme a **v reálnom čase notifikovať klienta** - používateľa o zmenách v systéme. Tento komponent generuje dynamicky renderované stránky aplikácie, ktoré sú zobrazované v zariadení klienta. Pre zabezpečenie čo najrýchlejšieho času odozvy, sú renderované stránky krátkodobo uložené vo vyrovnávacej cache pamäti. Použitím vyrovnávacej cache pamäte sa redukuje čas potrebný na opätovnú prípravu stránky a taktiež odľahčuje interné služby systému, nakoľko nie je potrebné žiadať opätovne dáta z interných služieb systému, pokiaľ sa stránka stále nachádza vo vyrovnávacej cache pamäti.

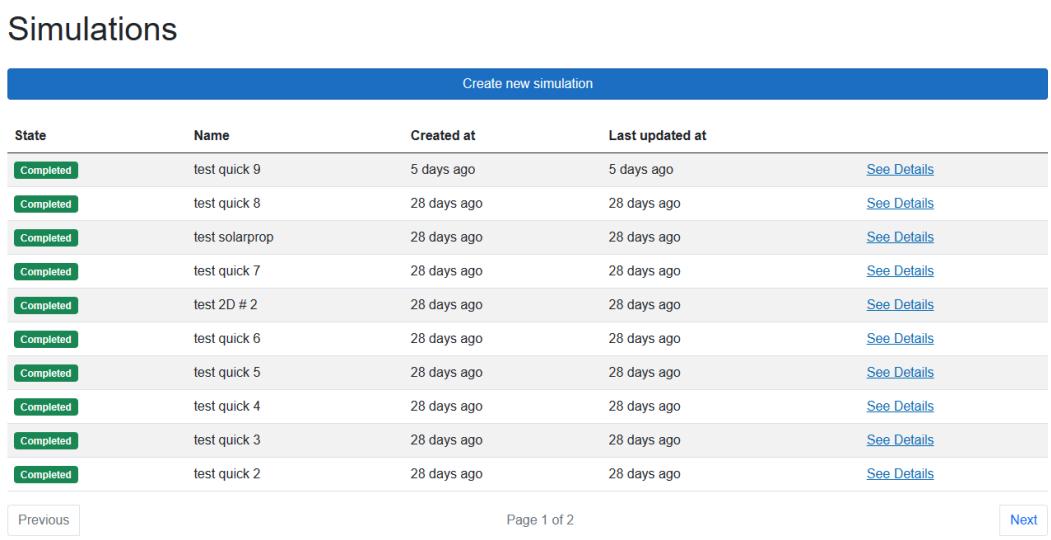
Pre webovú aplikáciu na strane klienta je použitých niekoľko podporných knižníc. Pre štýlovanie stránok a komponentov je použitá CSS knižnica **Bootstrap 5**[7]. Aplikácia zobrazuje výsledky simulácií vo forme grafov. Pre ich zobrazovanie a používateľskú interakciu s nimi je použitá knižnica **Chart.js**[8].

Na obrázkoch 4.4, 4.5 a 4.6 sa nachádzajú implementácie stránok vytvárania simulácie, zoznamu simulácií a detailu simulácie. Na daných obrázkoch je zachytená však len hlavná sekcia stránky, bez bočného a horného navigačného panelu. Všetky stránky, ktoré zobrazujú zoznam entít, napríklad simulácií, majú implementovaný **stránkovací mechanizmus** (angl. pagination), s možnosťou voľby počtu zobrazených kusov, až do maximálneho povoleného počtu - **100 kusov**. Ďalšie časti webovej aplikácie, ako je to popísané v návrhu v podkapitole 3.4, sú implementované rovnakým spôsobom a používajú rovnaké UI komponenty, ktoré sú generický pripravené, aby podporovali zobrazenie rôznych typov dát.

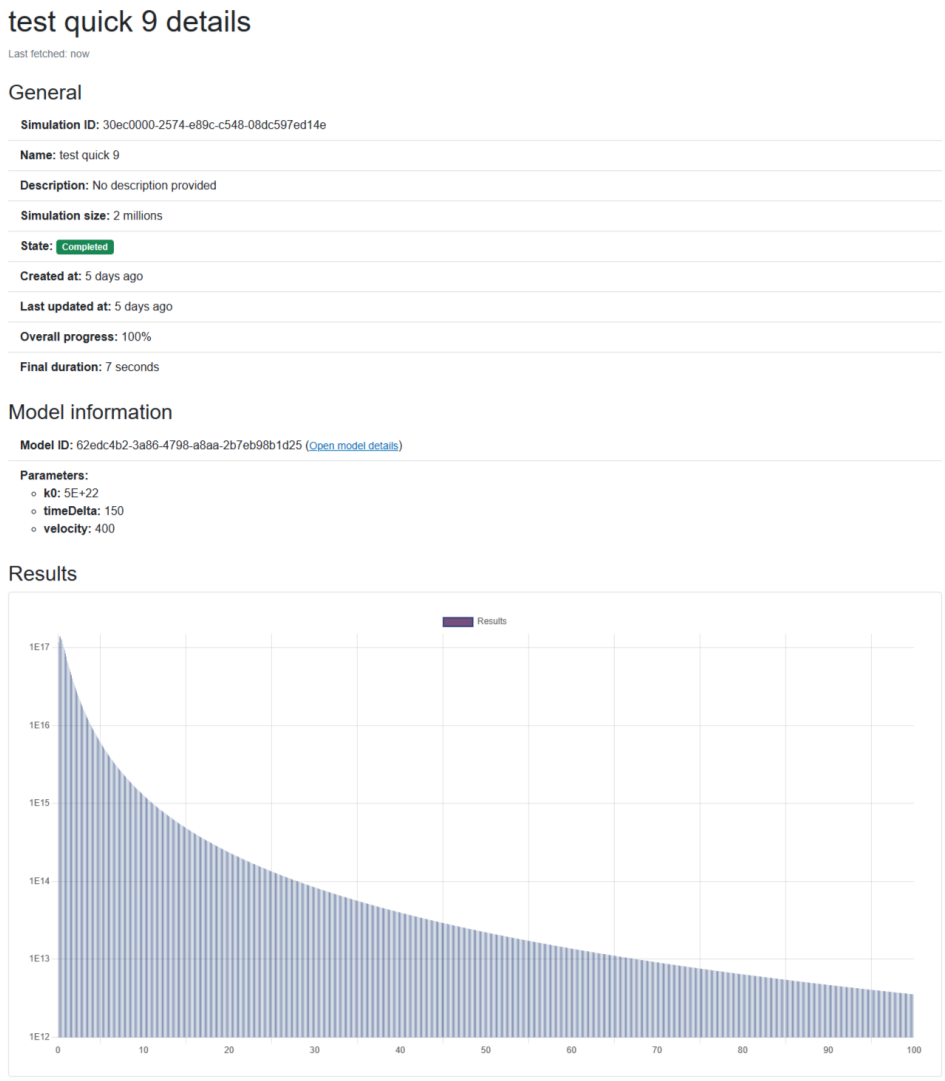
[6] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`
[7] `https://getbootstrap.com/docs/5.3/`
[8] `https://www.chartjs.org/`



Obr. 4.4: Implementácia webovej aplikácie - Vytvorenie simulácie



Obr. 4.5: Implementácia webovej aplikácie - Zoznam simulácií



Obr. 4.6: Implementácia webovej aplikácie - Detail simulácie

5 Mechanizmy nasadenia systému

Z dôvodu špecifikácie požiadavky o nasadení systému z podkapitoly 1.3, sa táto práca venuje aj mechanizmom automatizovaného nasadenia celého distribuovaného systému. V nasledujúcich podkapitolách sú popísane techniky používané pri jednak lokálnom vývoji systému a taktiež aj automatizovaného nasadenia do vzdialených prostredí, napríklad do produkčného.

5.1 Lokálny vývoj systému

Počas lokálneho vývoja systému je samozrejmé možné spúšťať komponenty samostatne a pracovať na každom z nich v izolácii. Počas vývoja distribuovaného systému sa však často stáva, že je nutné pracovať s viacerými komponentami naraz. Spúšťanie celého systému sa tak stáva neorganizované, vyžaduje viac času a pozornosti. Pre zjednodušenie práce pri vývoji distribuovaného systému je v tomto systéme využitý **.NET Aspire**, ktorý ponúka platforma `.NET`. *.NET Aspire* je podľa oficiálnej dokumentácie [31] definovaný ako súbor nástrojov pre vývoj cloudových, transparentných (z hľadiska telemetrie), produkčne pripravených, distribuovaných aplikácii. Ponúka orchestráciu systému počas lokálneho vývoja, bez nutnosti existencie **Dockerfile**, resp. iných definícii. Komponentov, ktoré majú byť orchestrované spomínaným nástrojom je deklarované a nakonfigurované v špeciálnom `C#` projekte, ktorý má nastavený atribút *IsAspireHost* na hodnotu `true`. V zdrojovom kóde projektu sú deklaratívne popísané jednotlivé komponenty systému a závislosti medzi nimi. Zdrojový kód 5.1 predstavuje ukážku deklaratívnej registrácie komponentov, ktoré majú byť orchestrované. Pri vývoji je nutné spustiť len tento jeden projekt a príslušné nástroje sa postarajú o prípravu potrebnej infraštruktúry a následne spustenie komponentov systému. Týmto spôsobom je možné efektívne a jednoducho vyvíjať celý distribuovaný systém. Distribuované stanice je možné spúšťať aj samostatne, čím je možné simulovať pripojenie viacero nezávislých staníc v systéme.

Zdrojový kód 5.1: Ukážka registrácie systémových komponentov v orchestračnom projekte .NET Aspire

```
var builder = DistributedApplication.CreateBuilder(args);

// Infrastructure - Postgres DB
var postgres = builder
    .AddPostgres("postgres", 5432, "SECRET_PASSWORD")
    .WithImageTag("latest")
    .WithVolumeMount("pg-data", "/var/lib/postgresql/data")
    .PublishAsContainer();

// System components
_ = builder
    .AddProject<Models_Service>("modelsservice")
    .WithReference(postgres.AddDatabase("modelsdb"));

builder.Build().Run();
```

5.2 Vzdialené nasadenie systému

V podkapitole 1.3 sú popísané možnosti nasadenia systému do produkčného prostredia. V závere kapitoly je odporučaný spôsob nasadenia na platformu *Kubernetes*[16], spôsobom deklaratívnych manifestov. Tieto manifesty je možné písať ručne, avšak pri akejkoľvek zmene v distribuovanom systéme je možné ľahko zabudnúť na prípadné nutné zmeny v manifestoch. Oveľa lepšou cestou je celý tento proces automatizovať.

V predchádzajúcej podkapitole o lokálnom vývoji je spomínaný nástroj *.NET Aspire*, ktorý zjednodušuje prácu s distribuovaným systémom pri lokálnom vývoji. Tento nástroj však pri spustení s prepínačom `--publisher manifest` nespustí všetky orchestrované komponenty distribuovaného systému, ale namiesto toho vygeneruje deklaratívny manifest vo formáte JSON, ktorý detailne popisuje celý systém. Pre ukážkový zdrojový kód 5.1 z predošlej podkapitoly, je vygenerovaný manifest, ktorý je vyobrazený na ukážke zdrojového kódu 5.2. Vygenerovaný súbor slúži ako základ pre nasadenie, ktorý je neskôr transformovaný na špecifické manifesty pre platformu *Kubernetes*. Pre túto transformáciu je použitý komunitný nástroj **Aspirate** (**Aspir8**)[1].

[1] `https://github.com/prom3theu5/aspirational-manifests`

Zdrojový kód 5.2: Ukážka vygenerovaného manifestu pomocou nástroja .NET Aspire z ukážkovej definície orchestračného projektu

```
{
  "resources": {
    "postgres": {
      "type": "postgres.server.v0"
    },
    "modelsservice": {
      "type": "project.v0",
      "path": "../Models.Service/Models.Service.csproj",
      "env": {
        "ConnectionStrings__modelsdb":
            "{modelsdb.connectionString}"
      }
    },
    "modelsdb": {
      "type": "postgres.database.v0",
      "parent": "postgres",
      "connectionString":
        "{postgres.connectionString};Database=modelsdb"
    }
  }
}
```

Príkazom `aspirate generate` nástroj *Aspirate* analyzuje vygenerovaný manifest, vytvorený nástrojom *.NET Aspire*, na základe ktorého vygeneruje špecifické finálne manifesty v jazyku *YAML* kompatibilné s platformou *Kubernetes*. Vytvorené manifesty sú plne pripravené na nasadenie, bez potreby ďalších manuálnych úprav. Jedinou podmienkou je mať skonštruované a publikované Docker obrazy v registri obrazov. Aj tento scenár však nástroj *Aspirate* pokrýva a pomocou príkazu `aspirate build` dokáže automaticky vytvoriť Docker obrazy pre všetky lokálne projekty, ktoré sú spomenuté v manifeste, vytvorenom nástrojom *.NET Aspire*. Spomínaný príkaz používa vlastnú metódu stavby Docker obrazov a jednotlivé projekty **nemusia obsahovať vlastnú Dockerfile definíciu**. Infraštruktúrne komponenty, ako napríklad databáza alebo sprostredkovateľ správ, sú pri príkaze `build` ignorované. Výsledné obrazy sú publikované do registra, zvoleného pomocou konfiguračného prepínača. V prípade predmetného distribuovaného systému sú obrazy publikované do Docker registra na platforme *GitHub* a spojené s Git repozitárom projektu. Vďaka vyššie spomínaným funkcionalitám ponúka

spojenie nástrojov *.NET Aspire* a *Aspirate* plne automatizovaný spôsob prípravy nasadenia systému. V ideálnom prípade je ich vhodné integrovať s platformami Continuous integration a continuous delivery (CI/CD) pre plne automatizovaný DevOps vývojový cyklus.

Distribuované stanice sú určené pre beh v kontajnerizovanom prostredí. Vyššie popísané riešenia slúžia na nasadenie centralizovanej vrstvy systému, nie pre samotné distribuované stanice. Projekt distribuovanej stanice obsahuje **Dockerfile** definíciu, vytvorenú v rámci bakalárskej práce [4]. Okrem malých zmien, ako úprava ciest projektov, zdvihnutie verzií použitých balíčkov a prechod na najnovšiu verziu programu *Geliosphere*, je predošlá *Dockerfile* definícia z bakalárskej práce stále dostačujúca a použitá aj v tejto implementácii distribuovanej stanice systému.

6 Verifikácia optimalizácie distribuovaného systému

Táto kapitola overuje správnosť návrhu a implementácie optimalizovaného distribuovaného systému. Pri testovaní a verifikácii systému sa kladie dôraz hlavne na nasledujúce oblasti:

- Zachovanie pôvodných funkcionalít a požiadaviek z bakalárskej práce [4],
- Splnenie požiadaviek z tejto práce.
- Zlepšenie bezpečnosti systému.
- Zrýchlenie systému oproti pôvodnému.
- Odolnosť systému voči zlyhaniam a následné zotavenie z chýb

Nasledujúce podkapitoly definujú metodiku testovania systému a vyhodnocujú výsledky z testovania.

6.1 Testovacie prostredie

Kritériom vhodného testovacieho prostredia je, aby sa čo najviac podobal reálnemu produkčnému prostrediu, v ktorom má byť systém nasadený za bežnej prevádzky. Preto nie je najvhodnejšie komponenty systému nasadiť na simulovaný **Kubernetes klaster**, bežiaci na jedinom (lokálnom) stroji. Pri testovaní prototypu bakalárskej práce boli komponenty systému nasadené na jednom z poskytnutých strojov, pričom tento stroj slúžil zároveň aj ako distribuovaný uzol[4]. Takéto nasadenie vyťažuje zdroje testovacieho uzla, čo môže viesť k zníženiu výpočtového výkonu. Práve preto sú riešenia ako napríklad **minikube**[1] alebo **kind**[2] neprijateľné pre verifikáciu distribuovaného systému.

[1] `https://minikube.sigs.k8s.io/`
[2] `https://kind.sigs.k8s.io/`

Pre testovanie systému bolo zvolené cloudové prostredie **Azure Kubernetes Service (AKS)**[3]. Jedná sa o inštanciu **Kuberenetes klastra**, ktorá je spravovaná priamo cloudovou platformou *Microsoft Azure*. Priradené virtuálne stroje disponujú dostatočným množstvom hardvérových zdrojov, aby sa predišlo strate výkonu systému pri testovaní, kvôli nedostatku zdrojov v samotnom klastri. Taktiež bol zapnutý **mechanizmus automatického škálovania zdrojov**, v prípade, že by existujúce zdroje nestačili.

V spolupráci s *ústavom experimentálnej fyziky Slovenskej akadémie vied*, boli pre účely testovania systému poskytnuté dve pracovné stanice, ktoré boli použité ako distribuované uzly systému. Rovnaké stanice boli poskytnuté aj pri testovaní bakalárskej práce [4], takže je možné priamo porovnávať výsledky testovaní oboch systémov, bez rozdielu v hardvérovom výkone. Tabuľka 6.1 udáva hardvérové konfigurácie týchto staníc.

	Pracovná stanica SAV #242	**Pracovná stanica SAV #244**
CPU	Intel Core i5-8400 @ 2.80GHz	Intel Core i5-7500 @ 3.40GHz
RAM	16GB	16GB
GPU	NVIDIA RTX 2060	NVIDIA GTX 1080 TI
VRAM	6GB	11GB

Tabuľka 6.1: Hardvérové konfigurácie distribuovaných uzlov systému použitých pri testovaní

6.1.1 Vyhodnotenie mechanizmov automatizovaného nasadenia systému

Pri nasadzovaní centralizovaných komponentov do pripraveného *Kubernetes klastra* na cloudovú platformu *Microsoft Azure* **nenastal žiaden problém**. Vďaka pripraveným vygenerovaným manifestom bolo nasadenie veľmi jednoduché. Nástroj *Aspirate*, ktorý je opísaný v podkapitole 5.2, poskytuje aj príkaz, s názvom `aspirate apply`, ktorý dokáže vygenerovať citlivé údaje (angl. secrets)[4] pred nasadením a následne systém nasadiť - vrátane vygenerovaných citlivých kľúčov v podobe **Kubernetes secret** objektov [16]. Po skončení testovania bol systém odstránený z klastra a všetky zdroje boli vyčistené od známok systému. Pre uľahčenie odstránenia ponúka nástroj *Aspirate* príkaz `aspirate destroy`, ktorý na zá-

[3] `https://azure.microsoft.com/products/kubernetes-service/`

[4] napríklad prístupové heslá do databázy a podobne

klade existujúcich manifestov dokáže odstrániť z klastra všetky predtým nasadené komponenty distribuovaného systému.

Nasadenie distribuovaného uzla systému je taktiež veľmi jednoduché, keďže aplikácia je kontajnerizovaná. Pre úspešné spustenie aplikácie distribuovaného uzla je potrebný taktiež len **jeden príkaz**, ktorý je zobrazený v ukážke 6.1.

Zdrojový kód 6.1: Príkaz na nasadenie distribuovaného uzla systému na platforme Docker, použitý pri testovaní systému

```
docker run -it --name cudahelio-worker-1 --gpus all      \
    -e Worker__ServerUri=<URL> -e Worker__Token=<TOKEN>  \
     --restart always ghcr.io/dadadcko/distributed-worker:latest
```

Vďaka použitému prepínaču `--restart always` je v prípade neočakávaného zlyhania uzla aplikácia ihneď reštartovaná.

Počas prvého pokusu nasadenia sa však vyskytol problém, z dôvodu nekompatibility nainštalovaných ovládačov *NVIDIA CUDA* na pracovných staniciach a požadovaných verzií týchto ovládačov aplikáciou. Riešením je pripraviť a vypublikovať rôzne verzie Docker obrazu, v závislosti od nainštalovaných verzií ovládačov *NVIDIA CUDA* v danom obraze. Po úprave inštalovaných verzií ovládačov v obraze, aby boli kompatibilné s ovládačmi nainštalovanými na testovacích zariadeniach, už nasadenie prebehlo v poriadku, bez problémov.

Je teda možné konštatovať, že navrhnuté a implementované **mechanizmy automatizovaného nasadenia systému priniesli efektívny a jednoduchý spôsob pre nasadenie celého distribuovaného systému**.

6.2 Testovanie systému

Testovanie systému prebiehalo v dvoch iteráciách. Pri každej iterácii bola použitá rovnaká konfigurácia, ktorá je popísaná v predošlej sekcii. Testovací scenár pozostával z výpočtu jednej simulácie, pričom počas behu systému sa sledovali aj ostatné oblasti verifikácie. Parametre simulácie, použitej v rámci testovacieho scenára, boli rovnaké ako pri testovaní prototypu z bakalárskej práce[4]. V úvode tejto kapitoly sú popísané hardvérové konfigurácie distribuovaných uzlov, ktoré sú použité pri testovaní tohto systému. Tieto konfigurácie sa taktiež zhodujú s konfiguráciami uzlov, ktoré boli použité pri testovaní prototypu systému z bakalárskej práce [4]. Na základe toho je možné priamo porovnávať výsledky testovania oboch verzií systému. Tabuľka 6.2 udáva parametre simulácie použitej v rámci testovacieho scenára.

Názov simulácie	Test1
Veľkosť simulácie	1600 miliónov
Použitý simulačný model	1D backward-in-time
k0	5×10^{22}
timeDelta	5
velocity	400

Tabuľka 6.2: Parametre simulácie použitej pri testovaní distribuovaného systému

6.2.1 Prvá iterácia testovania

Pri testovaní jemnozrnného distribučného modelu v bakalárskej práci [4] sa ukázalo, že ideálna veľkosť jednej simulácie pre výpočet na distribuovanom uzle je **50 miliónov**. Preto aj v tomto systéme pri testovaní bol nastavený konfiguračný parameter delenia simulácií na menšie, 50 miliónové časti.

Čas potrebný na výpočet jednej časti simulácie v tejto iterácií testovania sa pohyboval v rozmedzí **4.5 až 5 minút**. Oproti pôvodnému systému z bakalárskej práce, kde výpočet jednej takejto časti trval v rozmedzí 3 až 4 minút, ide o **výrazné spomalenie**. Pri sledovaní celkového stavu systému bolo zistené, že spomalenie nastalo na úrovni distribuovaného uzla - konkrétne v časti exekútora. Po diskusií s vývojármi programu Geliosphere bolo zistené, že v novších verziách programu bola znížená predvolená miera paralelizácie, za účelom lepšej používateľskej skúsenosti a odozvy. Výsledky tejto iterácie testovania z hľadiska rýchlosti systému preto **nie sú obsiahnuté vo výslednom vyhodnotení**.

Počas tejto iterácie bola takisto otestovaná funkcionalita predčasného zrušenia vykonávania distribuovanej úlohy a taktiež aj redundantná distribúcia. Distribuovaná úloha výpočtu simulácie bola pridelená dvom distribuovaným staniciam, po uplynutí chráneného intervalu, definovaného parametrom `JobGracePeriod`, ktorý je popísaný v tabuľke 4.1. Tieto stanice istú chvíľu počítali úlohu súbežne, až pokiaľ ju jedna z nich nedokončila. Po odovzdaní výsledkov bola druhá distribuovaná stanica notifikovaná mechanizmom predčasného zrušenia, s uvedeným dôvodom **ALREADY_FINISHED**. Po obdržaní signálu stanica okamžite zrušila vykonávanie úlohy a presunula sa na ďalšiu iteráciu jej životného cyklu, ako je definované v návrhu zo sekcie 3.3.2.

Okrem toho boli v tejto iterácií otestované aj zlyhania jednotlivých distribuovaných uzlov, podobne ako pri testovaní prototypu systému v bakalárskej práci [4]. Zlyhania boli pri testovaní simulované náhodným ukončením procesu dis-

tribuovaného uzla, resp. stopnutím Docker kontajneru s uzlom.

6.2.2 Druhá iterácia testovania

Po dohode s vedúcim práce a menšej úprave konfigurácie programu Geliosphere bola realizovaná aj druhá iterácia testovania systému. Keďže distribuovaný uzol spravuje aplikáciu Geliosphere v dcérskom procese, zmeny, ktoré viedli k zlepšeniu používateľskej skúsenosti a odozvy skôr uškodili distribuovanému systému, ako prilepšili. Používateľ nepotrebuje vidieť priebeh výpočtov na úrovni uzla. Na základe toho bola pre účely distribuovaného systému v programe Geliosphere spätne zvýšená miera paralelizácie, ktorá bola použitá v starších verziách programu. Ostatné funkcionality boli otestované a verifikované v prvej iterácii testovania, preto sa táto iterácia sústredí výhradne na verifikáciu rýchlosti systému.

Po aplikovaní vyššie spomínaných zmien, sa čas potrebný pre výpočet jednej časti simulácie v tejto iterácii testovania zaberal približne **3 minúty**. Zvýšením stupňa paralelizácie v programe Geliosphere sú dosahované časy priamo porovnateľné s časmi, ktoré boli namerané pri testovaní pôvodného systému z bakalárskej práce [4].

Vo finálnom vyhodnotení výsledkov z hľadiska rýchlosti systému sú použité výsledky práve z tejto iterácie testovania.

6.3 Vyhodnotenie rýchlosti systému

Jednou z požiadaviek práce je pri optimalizácii systému dbať ohľad na rýchlosť systému a samozrejme aj rýchlosť výpočtov samotných simulácií. Dôležitou metrikou, ktorá je porovnávaná medzi všetkými testovanými verziami distribuovaného systému z [3], [4] a tejto práce, je **celkový čas výpočtu testovacej simulácie**. Porovnanie výsledkov testovania, z hľadiska rýchlosti výpočtu testovacej simulácie, z vyššie spomínaných testovaných verzií systému, je reprezentované tabuľkou 6.3.

Verzia distribuovaného systému	**Celkový čas výpočtu [hod.]**
Pôvodný prototyp systému z [3]	1.17
Prototyp systému z bakalárskej práce [4]	0.942
Optimalizovaný systém z tejto práce	0.874

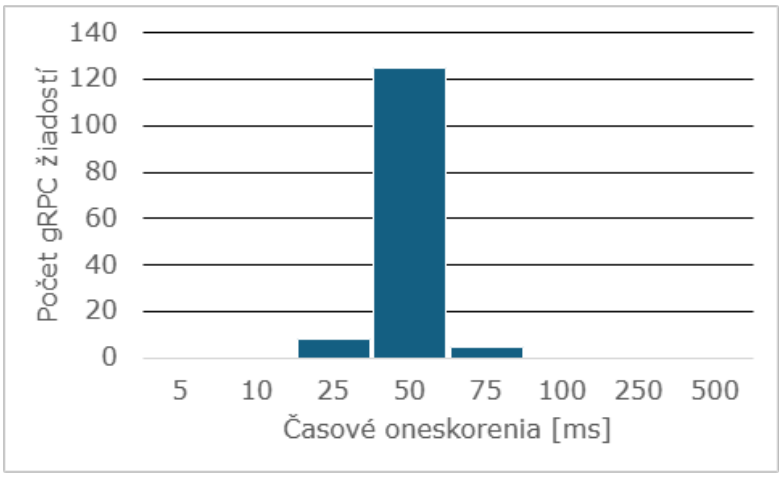
Tabuľka 6.3: Celkové časy výpočtu jednotlivých testovaných verzií distribuovaného systému

Počas druhej iterácie testovania dosiahol systém celkový čas výpočtu testovacej simulácie presne **52 minút a 28 sekúnd**. Pri porovnaní s predošlými verziami distribuovaného systému, ktorým výpočet rovnakej simulácie trval *56 minút a 32 sekúnd*[5], respektíve *1 hodinu, 10 minút a 12 sekúnd*[6], sa jedná opäť jedná o ďalšie celkové zrýchlenie systému. V percentuálnom zobrazení ide o zrýchlenie systému o ďalších **7,2%**, respektíve, pri porovnaní s pôvodným prototypom systému až o **25,26%**. Aj napriek tomu, že táto práca sa nevenovala priamej optimalizácii mechanizmov, ktoré majú za cieľ zrýchlenie systému, vďaka prechodu na najvyššiu verziu programu Geliosphere a optimalizáciou architektúry bola nepriamo aj pozitívne ovplyvnená rýchlosť systému, vďaka čomu je výsledný implementovaný distribuovaný systém rýchlejší, ako kedykoľvek predtým. Je teda možné konštatovať, **že optimalizácia architektúry distribuovaného systému priniesla pozitívne výsledky z hľadiska zrýchlenia systému**.

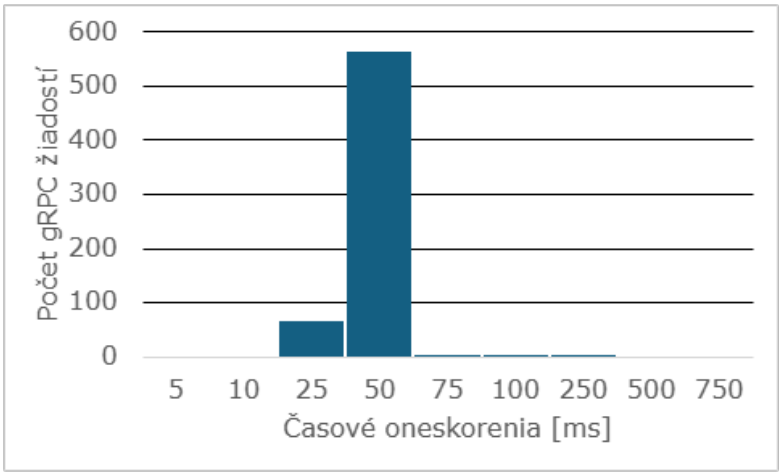
Počas druhej iterácie testovania boli zozbierané taktiež serverové metriky zo systémového komponentu *API brány pre distribuované stanice*, navrhnutého v sekcii 3.2.3 a implementovaného v podkapitole 4.4. V metrikách je nameraná celková dĺžka sieťovej komunikácie počnúc od vyslania žiadosti distribuovanou stanicou, cez komunikáciu brány s internými komponentmi, až po vrátenie odpovede distribuovanej stanici z *API brány*. Zozbierané metriky slúžia na verifikáciu časového oneskorenia pri komunikácii distribuovaných uzlov systému s centralizovanou vrstvou distribuovaného systému. Histogramy na obrázkoch 6.1, 6.2 a 6.3 zobrazujú vyššie spomínané časové oneskorenia, v závislosti od typu žiadosti, vykreslené na základe získaných metrík. Pri pohľade na tieto histogramy je zjavné, že takmer časy oneskorenia takmer všetkých žiadosti, ktoré vysielali distribuované stanice na centralizovanú vrstvu systému, sú v rozmedzí od **25 až 75 milisekúnd**. Komunikácia pomocou rámca *gRPC* udržiava veľmi nízke časy oneskorenia jednotlivých volaní. Na základe vyhodnotenia zozbieraných metrík je dokázané, že rámec *gRPC* je vhodne vybraným komunikačným rámcom pre tento typ komunikácie v distribuovanom systéme. Všetky režijné náklady na komunikáciu pre každú žiadosť sú v rozmedzí niekoľko desiatok *milisekúnd*, čo sa dá považovať ako zanedbateľné. Režijné náklady na komunikáciu v systéme tak nespomaľujú celkový výkon systému. Je teda možné konštatovať, **že navrhnutá a implementovaná optimalizovaná architektúra distribuovaného systému je veľmi výkonná z hľadiska komunikácie medzi jednotlivými vrstvami a komponentami systému**.

[5] celkový čas výpočtu v prototype systému z bakalárskej práce[4]

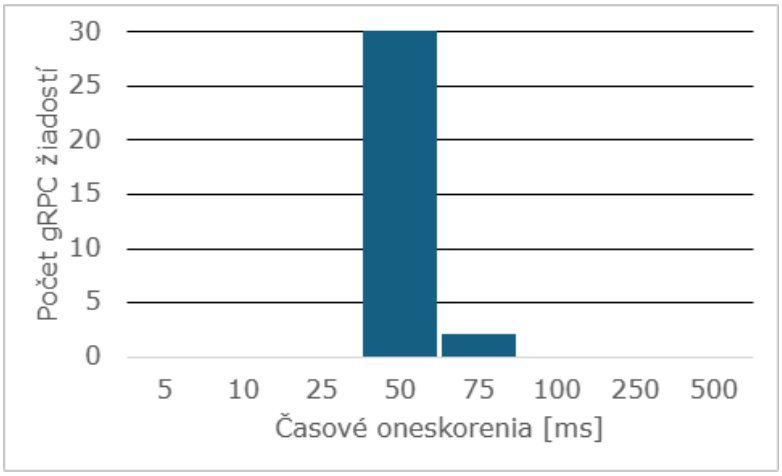
[6] celkový čas výpočtu v pôvodnom distribuovanom systéme[3]



Obr. 6.1: Histogram časových oneskorení žiadostí typu RequestAJob zozbieraných pri testovaní systému



Obr. 6.2: Histogram časových oneskorení žiadostí typu ProlongJobLease zozbieraných pri testovaní systému



Obr. 6.3: Histogram časových oneskorení žiadostí typu UploadJobResults zozbieraných pri testovaní systému

6.4 Vyhodnotenie bezpečnosti systému

Pri testovaní systému boli taktiež verifikované mechanizmy bezpečnosti systému, konkrétne autentifikácia a autorizácia distribuovaných staníc, ktorá je jednou z požiadaviek tejto práce, opísaná v sekcii 1.2.2. Odskúšané boli pokusy o neautorizované akcie zo strany simulovanej škodlivej distribuovanej stanice v systéme. Keďže interná komunikačná vrstva systému, zabezpečená službou *RabbitMQ* nie je prístupná z vonkajšieho prostredia, distribuované stanice sú nútené komunikovať so zvyškom systému len pomocou komponentu *API brány*, ktorej návrh a implementácia sú opísané v sekcii 3.2.3 a v podkapitole 4.4. Znemožnenie prístupu do internej komunikačnej siete distribuovaného systému **zmenšuje potenciálny vektor útoku**, a tým sám o sebe zlepšuje bezpečnosť systému.

Implementované autentifikačné mechanizmy na úrovni *API brány* sa ukázali ako dostačujúce, nakoľko pri odskúšaní volania brány s neplatným autentifikačným tokenon, respektíve úplne bez priloženia tokenu, brána ihneď odpovedala chybovým kódom `401 Unauthenticated`. Platný autentifikačný token pre distribuovanú stanicu je možné získať len prostredníctvom používateľského rozhrania po prvotnom zaregistrovaní distribuovanej stanice, ku ktorému má prístup len správca, resp. používatelia systému. Neznámym škodlivým distribuovaným staniciam je preto **prístup do systému nepovolený**, čím je zvýšená integrita a bezpečnosť systému.

Druhým bezpečnostným prvkom, ktorý bol pri testovaní systému verifikovaný boli implementácie autorizačných mechanizmov jednotlivých distribuovaných staníc na úrovni *služby pre distribuované úlohy*. Verifikované boli nasledujúce implementované autorizačné mechanizmy:

- **Distribuovaná stanica môže mať vždy priradenú len jednu úlohu.**

  Pri viacnásobnom žiadaní o úlohu služba detegovala, že žiadajúca distribuovaná stanica už existujúcu priradenú úlohu má a odpovedala stále tou istou úlohou. Plánovač nepriradil žiadne ďalšie úlohy distribuovanej stanici. Týmto mechanizmom je **zabránené uzamknutie všetkých úloh** škodlivou stanicou.

- **Distribuovaná stanica nemôže odovzdať výsledky úlohy, ktorá jej nie je priradená.**

  Pri pokuse o odoslaní výsledkov úlohy, ktorá nebola priradená žiadajúcej distribuovanej stanici, služba výsledky **neprijala** a namiesto toho odpovedala signálom predčasného zrušenia, s uvedeným dôvodom **JOB_LOST**.

Táto správa signalizuje, že úloha bola „stratená“ - nepatrí žiadajúcej stanici. Týmto mechanizmom je **zabránené nedovolené odovzdávanie výsledkov** iných úloh, s cieľom poskytnúť nepravdivé, zmanipulované výsledky.

- **Distribuovaná stanica nesmie mať uzamknutú úlohu donekonečna.**

  Ako bolo popísané v návrhu tejto služby v sekcii 3.2.1, implementovaný plánovač obsahuje funkcionalitu *redundantných distribúcií*, pomocou ktorých je úloha priradená viacerým distribuovaným staniciam súčasne, pokiaľ je jedna z nich priradená už príliš dlho. Počas testovania bolo simulované vyžiadanie a držanie jednej distribuovanej úlohy škodlivou stanicou, čo za normálnych okolností by spôsobilo riziko čiastočného uviaznutia systému (angl. deadlock). Napríklad, pri výpočte simulácií by sa výpočet celej simulácie nikdy nedokončil. Po uplynutí ochrannej doby, počas ktorej nie je možné aplikovať mechanizmus *redundantnej distribúcie* na úlohu, bola úloha pomocou tohto mechanizmu priradená súčasne aj na inú distribuovanú stanicu, ktorá ju úspešne vykonala. Po odovzdaní výsledkov dostala škodlivá stanica notifikáciu o predčasnom ukončení vykonávania úlohy, s uvedeným dôvodom **ALREADY_FINISHED**. Týmto mechanizmom je **zabránené nedovolené držanie úlohy**, čím by sa spôsobil čiastočné uviaznutie systému (angl. deadlock).

Na základe vyššie popísaných a verifikovaných bezpečnostných mechanizmov, je možné konštatovať, že implementácie bezpečnostných mechanizmov a nový návrh a implementácia plánovača výrazne zlepšili celkovú bezpečnosť a integritu distribuovaného systému a minimalizovali možnosti útoku zo strany škodlivých distribuovaných staníc.

6.5 Vyhodnotenie odolnosti systému a zotavenia z chýb

Distribuovaný systém bol pri testovaní verifikovaný aj z hľadiska jeho celkovej odolnosti voči zlyhaniam a následného zotavenia z prípadných neočakávaných zlyhaní. Komponenty centralizovanej vrstvy distribuovaného systému boli po celú dobu behu systému v testovacom prostredí monitorované. Po skončení testovania boli telemetrické dáta zo systému zozbierané a vyhodnotené. Hoci platforma *Kubernetes* poskytuje automatické uzdravovanie formou reštartu neočakávane ukončených kontajnerov, vyhodnotené telemetrické dáta nenaznačovali nutnosť využitia tejto funkcionality platformy. Pri výskyte chyby na úrovni centrali-

zovanej mikroslužby sa **mikroslužba dokázala správne zotaviť**, bez neočakávaných pádov a ukončenia.

Telemetria však ukázala, že pri niektorých akciách v *službe pre distribuované úlohy*, počas súbežného prístupu viacerých vlákien k rovnakým úlohám, služba vyhadzovala výnimku typu `DbConcurrencyException`, čo má za následok použitie mechanizmu *optimistického uzamykania* na úrovni použitej databázy. Pre tento typ výnimky je však pri vykonávaní predmetných akcií nastavený **opakovací mechanizmus**, ktorý vykonanie akcie zopakuje znovu. Pri opätovnom vykonávaní však plánovač už pracuje s inou množinou dát, nakoľko pôvodné dáta boli zmenené, čo je predpokladom vyššie spomínanej výnimky. Táto výnimka teda nie je neočakávaná a služba s ňou dokáže pracovať. Po zopakovaní, niekedy aj viacnásobnom, je akcia eventuálne vykonaná úspešne. Okrem tohto typu výnimky, resp. chyby telemetrické dáta nepreukázali žiadne iné formy zlyhania systému.

V prípade verifikácie odolnosti distribuovaného systému voči náhodným zlyhaniam distribuovaných staníc bol pri testovaní využitý podobný scenár ako pri prototype z bakalárskej práce [4]. Jedna distribuovaná stanica bola náhodne vypnutá počas vykonávania úlohy výpočtu simulácie, s cieľom zistiť, či systém dokáže pokračovať a výpočet celej simulácie je eventuálne dokončený. V testovaní bolo preukázané, že úloha, ktorá bola priradená vypnutej stanici, bola zo začiatku uzamknutá pre túto stanicu. Po uplynutí časového intervalu, definovaného konfiguračným parametrom `UpdateMaxTimeout`, ktorý je popísaný v tabuľke 4.1, bola úloha plánovačom odomknutá a neskôr priradená inej, funkčnej stanici a výpočet celej simulácie bol eventuálne dokončený. V testovaní bol tento konfiguračný parameter nastavený na hodnotu **30 sekúnd**.

Posledný scenár testovania odolnosti systému mal za úlohu verifikovať odolnosť samotných distribuovaných staníc. V tomto scenári bol komponent *API brány pre distribuované stanice* považovaný za nedostupný. Volania brány z jednotlivých staníc postupne vypršali a *gRPC* klient vyhodil výnimku so stavovým kódom `Unavailable`. Stanica však tento typ výnimky dokázala odchytiť a správne na ňu reagovať výpisom upozornenia a opakovaním volania v priebehu **3 sekúnd**. Vyskúšaným scenárom bolo preukázané, že distribuovaná stanica sa dokázala zotaviť zo zlyhania jednotlivých volaní, bez jej neočakávaného ukončenia. Navyše dokázala stále fungovať aj v prípade, že jej hlavný komunikačný bod nebol dostupný.

Testovaním sa potvrdilo, že systém je dostatočne odolný voči zlyhaniam a dokáže sa bez problémov zotaviť zo zlyhaní rôznych typov.

6.6 Vyhodnotenie aktualizovaného používateľského rozhrania

Pri testovaní distribuovaného systému bolo používateľovi systému sprístupnené používateľské rozhranie, pomocou ktorého dokázal systém ovládať. Používateľ nemal predtým žiadne znalosti o implementovanom používateľskom rozhraní tohto systému, len o doméne systému. Počas testovania bol kladený dôraz na overenie intuitívnosti a jednoduchosti používateľského rozhrania pre používateľa.

Používateľ po ukončení testovania systému označil používateľské rozhranie ako **intuitívne** a **jednoduché na obsluhu**. Rýchlosti načítavania jednotlivých stránok a potrebných dát ohodnotil taktiež pozitívne. Pri používaní webovej aplikácie sa cítil sebaisto, a dokázal ju ovládať bez potreby pomoci. Kladne ohodnotil aj konzistenciu jednotlivých stránok webovej aplikácie.

Na základe otestovania používateľského rozhrania používateľom systému je možné konštatovať, že **navrhnuté a implementované aktualizované používateľské rozhranie je dostačujúce, intuitívne a jednoduché na použitie**. Na druhú stranu bola implementácia používateľského rozhrania len okrajovou témou tejto práce a vyžaduje si ďalšie úpravy, resp. vylepšenia v budúcnosti.

7 Záver

Táto diplomová práca sa venovala optimalizácii architektúry existujúceho distribuovaného systému pre výpočty simulácií v heliosfére. V úvodnej časti práce bola vykonaná analýza existujúceho prototypu tohto distribuovaného systému z bakalárskej práce [4], na základe ktorej boli identifikované oblasti na optimalizáciu a definované nové požiadavky v systéme. Pre vytvorenie vhodného návrhu optimalizovanej architektúry tohto systému boli preskúmané iné existujúce riešenia, ktoré využívajú techniku distribuovaného plánovania a distribuovaných úloh.

Navrhnutá a implementovaná optimalizovaná architektúra systému čerpala základ z architektúry, ktorá bola navrhnutá v bakalárskej práci [4]. Hlavnými oblasťami jej optimalizácie v tejto práci boli **izolácia internej komunikačnej vrstvy** systému, **zvýšenie bezpečnosti a robustnosti** systému a **rozšírenie pôvodnej implementácie distribuovaného plánovača úloh** o podporu ďalších typov distribuovaných úloh.

Implementácie plánovania a distribuovania úloh boli extrahované z pôvodnej *simulačnej služby* do nového navrhnutého komponentu - **služby pre distribuované úlohy**. Služba dokáže plánovať a distribuovať úlohy distribuovaným staniciam systému bez viazanosti na doménovú logiku jednotlivých typov úloh. V rámci tohto komponentu boli do systému pridané viaceré funkcionality, ktoré vylepšujú základnú implementáciu plánovača, ako napríklad mechanizmus **redundantnej distribúcie** alebo mechanizmus **predčasného ukončenia vykonávania** distribuovanej úlohy. Navrhnutý plánovač taktiež obsahuje **autorizačné mechanizmy**, ktoré zvyšujú bezpečnosť a integritu celého distribuovaného systému. Škodlivým distribuovaným staniciam je tak **zabránená nepovolená manipulácia dát** a pokus o čiastočné uzamknutie systému (angl. deadlock).

V rámci navrhnutej architektúry bol do systému pridaný nový komponent - *modelová služba*, ktorá do systému pridáva podporu **vlastných simulačných modelov**, definovaných používateľom. Pre validáciu a kompiláciu vlastných simulačných modelov je v tejto službe obsiahnutá príprava na integráciu s aplikáciou *Autosphere* [5]. Nakoľko kompilácia modelov musí prebiehať na distribuovaných

staniciach, *modelová služba* definuje nový typ distribuovanej úlohy v systéme a používa *službu pre distribuované úlohy* pre plánovanie a distribuovanie týchto úloh. Implementácia existujúcej *simulačnej služby* bola upravená a rozšírená o podporu vlastných simulačných modelov a integráciu s novými službami systému.

Pre účely zvýšenia bezpečnosti systému bola izolovaná interná komunikačná vrstva systému, zabezpečovaná službou *RabbitMQ*. Pre účely komunikácie distribuovaných staníc s centralizovanou vrstvou systému bol preto v tejto práci navrhnutý a implementovaný nový komponent systému - **API brána pre distribuované stanice**. Pre komunikáciu je použitý rámec *gRPC*. Pri testovaní systému bolo na základe zozbieraných systémových metrík dokázané, že tento rámec poskytuje veľmi rýchlu komunikáciu a časy oneskorenia jednotlivých volaní, pri komunikácii distribuovaných staníc s centralizovanou vrstvou systému, sú veľmi nízke. V tomto komponente je taktiež implementovaný **autentifikačný mechanizmus**, vďaka ktorému je do systému povolený **prístup len známym a overeným distribuovaným staniciam**.

Na úrovni distribuovaných staníc systému boli realizované úpravy pre zabezpečenie kompatibility s novou architektúrou systému, ako aj pridanie podpory mechanizmu predčasného ukončenia vykonávania úlohy. Distribuované stanice boli rozšírené o možnosť vykonávania všetkých typov úloh, ktoré sú v systéme definované, pomocou konceptu **exekútorov**. Po prijatí úlohy je na základe typu úlohy zvolený správny exekútor, ktorý obsahuje potrebnú implementáciu na vykonanie tejto úlohy. Aplikácia distribuovanej stanice bola taktiež upravená tak, aby bola kompatibilná s najnovšou verziou programu Geliosphere.

Kvôli novým funkcionalitám, ktoré boli v rámci tejto práce do systému pridané, bolo aktualizované aj pôvodné používateľské rozhranie vo forme webovej aplikácie. Navrhnutá a implementovaná **webová aplikácia** umožňuje používateľom ovládať všetky časti systému, od ovládania simulácií, cez prípravu vlastných simulačných modelov až po správu distribuovaných staníc systému. Počas testovania s používateľom systému bolo potvrdené samotným používateľom, že výsledná webová aplikácia je **intuitívna**, **jednoduchá na použitie** a jej funkcionality sú zatiaľ pre tento distribuovaný systém **dostačujúce**.

Práca sa venovala aj príprave **automatizovaných mechanizmov nasadenia tohto systému**. Výsledné riešenie používa nástroje *.NET Aspire* a *Aspirate* na **automatizované generovanie manifestov** nasadenia na platformu **Kubernetes**, podľa deklaratívneho popisu komponentov systému a závislostí medzi nimi. Nástroje spravujú vlastné definície pri vytváraní kontajnerových obrazov, preto už v systéme nie je nutné definovať osobitné *Dockerfile* definície pre každý komponent.

Mechanizmus nasadenia jednotlivých distribuovaných staníc ostal pôvodný z bakalárskej práce [4], realizovaný pomocou kontajnerového obrazu. V *Dockerfile* definícii obrazu boli však aktualizované verzie balíčkov použitého rámca *Nvidia CUDA* a verzia programu Geliosphere.

Verifikácia navrhnutej optimalizovanej architektúry distribuovaného systému a implementovaných komponentov systému bola realizovaná prostredníctvom dvoch iterácií testovania v simulovanom produkčnom prostredí. Centralizovaná vrstva systému bola pre účely testovania nasadená do **Kubernetes** klastra na platforme *Azure Kubernetes Service* (*AKS*), v cloudovom prostredí *Microsoft Azure*. Pre potreby distribuovaných staníc boli zapožičané 2 počítačové systémy prostredníctvom *ústavu experimentálnej fyziky Slovenskej akadémie vied*. Testovaním bolo preukázané, že navrhnutá a implementovaná optimalizovaná architektúra systému **spĺňa definované bezpečnostné požiadavky**, **zlepšila odolnosť systému voči zlyhaniam** a jeho následné zotavenie z chýb. Na základe zozbieraných serverových metrík z testovania bolo dokázané, že komunikácia medzi jednotlivými vrstvami a komponentami systému je **veľmi rýchla** a efektívna, pričom časy oneskorenia sa držali v okolí **50 milisekúnd**. Pri testovacom scenári výpočtu simulácie bolo dokonca zistené, že optimalizáciou architektúry a prechodom na najnovšiu verziu programu Geliosphere, **bol výpočet ešte viac zrýchlený**, v porovnaní výsledkov testovania prototypu z bakalárskej práce [4], **o 4 minúty a 4 sekundy**, čo v percentuálnom prepočte predstavuje **7.2%**. Pri porovnaní *s úplne pôvodným prototypom* tohto distribuovaného systému z [3] to predstavuje **zrýchlenie o približne 25.3%**.

Optimalizácia architektúry zabezpečila rozvoj tohto distribuovaného systému do budúcnosti. Generická implementácia plánovača umožňuje v budúcnosti pridanie nových typov úloh, napríklad analýza simulačných modelov na báze umelej inteligencie, s cieľom ešte lepšej optimalizácie výslednej paralelnej implementácie. Systém má pripravené mechanizmy na integráciu s aplikáciou *Autosphere* [5], avšak nakoľko táto aplikácia je stále na úrovni prototypu, pre plnú integráciu tejto aplikácie do distribuovaného systému je nutné počkať do vydania jej plnej verzie. V budúcnosti je plánovaný aj ďalší vývoj distribuovaného systému, hlavne v časti používateľského rozhrania, kde je možné implementovať rôzne podporné funkcionality, ktoré uľahčia používateľovi prácu so systémom. Medzi takéto funkcionality môžeme zaradiť napríklad plnohodnotný editor pre písanie kódu vlastných simulačných modelov alebo pomocníka pri vytváraní simulácie.

Literatúra

1. COULOURIS, George F; DOLLIMORE, Jean; KINDBERG, Tim. *Distributed systems: concepts and design*. pearson education, 2005.
2. TANENBAUM, Andrew S; VAN STEEN, Maarten. *Distributed systems: principles and paradigms*. Prentice-hall, 2007.
3. SOLANIK, Michal. *Paralelizácia dvojrozmerných modelov modulácie kozmického žiarenia v heliosfére*. 2020. Dipl. pr. Technická univerzita v Košiciach.
4. SLANINA, Daniel. *Jemnozrnný paralelizmus v distribuovanom systéme*. 2022. Technická univerzita v Košiciach.
5. SOLANIK, Michal. *Methods and algorithms for acceleration / parallelization of physical model calculations*. 2023. Diz. pr. Technical university of Košice.
6. VMWARE. *RabbitMQ documentation* [online]. [cit. 2023-11-14]. Dostupné z : `https://www.rabbitmq.com/admin-guide.html`.
7. MAGNONI, L. Modern Messaging for Distributed Sytems. 2015, roč. 608, s. 012038. Dostupné z DOI: `10.1088/1742-6596/608/1/012038`.
8. MARCU, Ovidiu-Cristian; BOUVRY, Pascal. *Colocating Real-time Storage and Processing: An Analysis of Pull-based versus Push-based Streaming*. 2022. Dostupné z arXiv: `2211.05857 [cs.DC]`.
9. LEE, Edward A.; AKELLA, Ravi; BATENI, Soroush; LIN, Shaokai; LOHSTROH, Marten; MENARD, Christian. *Consistency vs. Availability in Distributed Real-Time Systems*. 2023. Dostupné z arXiv: `2301.08906 [cs.DC]`.
10. BROWN, Scott; HARMAN, David; ANDERSON, Cleon; DWYER, Matthew. Measuring Data Transmissions from the Edge for Distributed Inferencing with gRPC. *2023 IEEE International Conference on Big Data (BigData)*. 2023, s. 3853–3856. Dostupné tiež z: `https://api.semanticscholar.org/CorpusID:267167697`.
11. LÄHTEVÄNOJA, Vili et al. *Communication Methods and Protocols Between Microservices on a Public Cloud Platform*. 2021. Dipl. pr. Aalto University.

12. BHATTACHARYA, Sourav; JHA, Susmit; SAMARATI, Pierangela. Secure Messaging in Distributed Systems: A Survey. *IEEE Communications Surveys & Tutorials*. 2018, roč. 20, č. 4, s. 2974–3004.

13. CALÇADO, Phil. The Back-end for Front-end Pattern (BFF). *Phil Calçado Engineering Blog*. 2015. Dostupné tiež z: `https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff`.

14. DOE, John; SMITH, Jane. Applying the Backend for Frontend Pattern in Microservices Architecture. In: *International Conference on Software Architecture*. 2019.

15. MERCL, Lubos; PAVLIK, Jakub. The comparison of container orchestrators. In: *Third International Congress on Information and Communication Technology: ICICT 2018, London*. Springer, 2019, s. 677–685.

16. THE KUBERNETES AUTHORS. *Kubernetes Documentation* [online]. [cit. 2023-11-18]. Dostupné z : `https://kubernetes.io/docs/`.

17. PINEDO, Michael L. *Scheduling: Theory, Algorithms, and Systems*. Springer Science & Business Media, 2016. Dostupné tiež z: `https://doi.org/10.1007/978-3-319-26580-3`.

18. GITLAB B.V. *GitLab Documentation* [online]. [cit. 2023-11-20]. Dostupné z : `https://docs.gitlab.com/ee/`.

19. GITHUB INC. *GitHub Documentation* [online]. [cit. 2023-11-20]. Dostupné z : `https://docs.github.com/en`.

20. MAJUMDER, Rajashree. *Maximizing Efficiency: Automated Software Testing With CI/CD Tools and Docker Containerization for Parallel Execution*. 2023. Dipl. pr. Ohio University.

21. SHARIF, Muddsair; JANTO, Skowronek; LUECKEMEYER, Gero. COaaS: Continuous Integration and Delivery framework for HPC using Gitlab-Runner. In: *Proceedings of the 2020 4th International Conference on Big Data and Internet of Things*. Singapore, Singapore: Association for Computing Machinery, 2020, s. 54–58. BDIOT '20. ISBN 9781450375504. Dostupné z DOI: `10.1145/3421537.3421539`.

22. TRONGE, Jake; CHEN, Jieyang; GRUBEL, Patricia; RANDLES, Tim; DAVIS, Rusty; WOFFORD, Quincy; ANAYA, Steven; GUAN, Qiang. BeeSwarm: Enabling Parallel Scaling Performance Measurement in Continuous Integration for HPC Applications. In: *2021 36th IEEE/ACM International Conference*

on Automated Software Engineering (ASE). 2021, s. 1136–1140. Dostupné z DOI: 10.1109/ASE51524.2021.9678805.

23. PACHEV, Benjamin; STUART, Georgia; DAWSON, Clint. Continuous Integration for HPC with Github Actions and Tapis. In: *Practice and Experience in Advanced Research Computing*. Boston, MA, USA: Association for Computing Machinery, 2022. PEARC '22. ISBN 9781450391610. Dostupné z DOI: 10.1145/3491418.3535124.

24. VETRIVEL, R.S. Possibilities of DoS/DDOS Attacks and the use of Filtering and Rate Limiting Mechanisms to Alleviate Message Security. *International Journal of Trend in Research and Development*. 2016, roč. 3, č. 6. Dostupné tiež z: https://www.ijtrd.com/papers/IJTRD5407.pdf.

25. PATTERSON, Chris. *MassTransit documentation* [online]. [cit. 2024-01-15]. Dostupné z : https://masstransit.io/.

26. SOLARZ, Arkadiusz; SZYMCZYK, Tomasz. Oracle 19c, SQL Server 2019, Postgresql 12 and MySQL 8 database systems comparison. *Journal of Computer Sciences Institute*. 2020, roč. 17, s. 373–378. Dostupné z DOI: 10.35784/jcsi.2281.

27. HASSAN KILAVO, Salehe I. Mrutu; DUDU, Robert G. Securing Relational Databases against Security Vulnerabilities: A Case of Microsoft SQL Server and PostgreSQL. *Journal of Applied Security Research*. 2023, roč. 18, č. 3, s. 421–435. Dostupné z DOI: 10.1080/19361610.2021.2006032.

28. OPENTELEMETRY AUTHORS. *OpenTelemetry Documentation* [online]. [cit. 2024-01-15]. Dostupné z : https://opentelemetry.io/docs/.

29. WAGNER, F.; SCHMUKI, R.; WAGNER, T.; WOLSTENHOLME, P. *Modeling Software with Finite State Machines: A Practical Approach*. 1. vyd. Auerbach Publications, 2006. Dostupné z DOI: 10.1201/9781420013641.

30. HAMO, Najem; SABERIAN, Simon. *Evaluating the performance andusability of HTTP vs gRPC in communication between microservices*. 2023. Dostupné tiež z: https://www.diva-portal.org/smash/record.jsf?dswid=-4884.

31. MICROSOFT. *.NET Aspire Documentation* [online]. [cit. 2024-02-08]. Dostupné z : https://learn.microsoft.com/dotnet/aspire/.

Zoznam skratiek

API Application Programming Interface.

BFF Backend for Frontend.

CI/CD Continuous integration a continuous delivery.

CLI Command-line interface.

CPU Central Processing Unit.

CSS Cascading style sheets.

DoS Denial of Service.

GPU Graphics Processing Unit.

HPC High-performance computing.

HTTP Hypertext Transfer Protocol.

MS-SQL Microsoft SQL Server.

RAM Random Access Memory.

RBAC Role-based Access Control.

REST Representational State Transfer.

TLS Transport Layer Security.

VRAM Video Random Access Memory.

Slovník

Geliosphere Open-source program pre výpočty simulácií kozmického žiarenia v heliosfére. Dostupný na `https://github.com/msolanik/Geliosphere/`.

MassTransit Open-source aplikačný rámec pre tvorbu distribuovaných aplikácií [25]. Dostupný na `https://github.com/MassTransit/MassTransit/`.

Zoznam príloh

A Serverové metriky API brány pre distribuované stanice

Táto príloha obsahuje serverové metriky systémového komponentu *API brány pre distribované stanice*, exportovaných vo formáte JSON, zozbieraných pri testovaní systému. Exportované sú iba hodnoty, ktoré boli relevantné pre verifikáciu systému.


```
{
 "resourceMetrics": [
  {
   "resource": {
    "attributes": [
     {
      "key": "service.name",
      "value": {"stringValue": "WorkersApiGateway"}
     },
     {
      "key": "telemetry.sdk.name",
      "value": {"stringValue": "opentelemetry"}
     },
     {
      "key": "telemetry.sdk.language",
      "value": {"stringValue": "dotnet"}
     },
     {
      "key": "telemetry.sdk.version",
      "value": {"stringValue": "1.7.0"}
     }
    ]
   },
   "scopeMetrics": [
```

```
{
 "scope": {
  "name": "Microsoft.AspNetCore.Hosting"
 },
 "metrics": [
  {
   "name": "http.server.request.duration",
   "description": "Duration of HTTP server requests.",
   "unit": "s",
   "histogram": {
    "dataPoints": [
     {
      "attributes": [
       {
        "key": "http.request.method",
        "value": {"stringValue": "POST"}
       },
       {
        "key": "http.response.status_code",
        "value": {"intValue": "200"}
       },
       {
        "key": "http.route",
        "value": {"stringValue": "/CudaHelio.Workers.
           ApiGateway.Proto.WorkersGateway/RequestAJob"}
       },
       {
        "key": "network.protocol.version",
        "value": {"stringValue": "2"}
       },
       {
        "key": "url.scheme",
        "value": {"stringValue": "http"}
       }
      ],
      "startTimeUnixNano": "1712932035014010700",
      "timeUnixNano": "1712935514982950700",
      "count": "139",
      "sum": 5.243373699999997,
      "bucketCounts": ["0", "0", "8", "125", "5", "0", "0",
```

```
      "0", "0", "1", "0", "0", "0", "0", "0"],
    "explicitBounds": [0.005, 0.01, 0.025, 0.05, 0.075,
      0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10],
    "min": 0.0228708,
    "max": 0.8087114
   },
   {
    "attributes": [
     {
      "key": "http.request.method",
      "value": {"stringValue": "POST"}
     },
     {
      "key": "http.response.status_code",
      "value": {"intValue": "200"}
     },
     {
      "key": "http.route",
      "value": {"stringValue": "/CudaHelio.Workers.
        ApiGateway.Proto.WorkersGateway/ProlongJobLease"}
     },
     {
      "key": "network.protocol.version",
      "value": {"stringValue": "2"}
     },
     {
      "key": "url.scheme",
      "value": {"stringValue": "http"}
     }
    ],
    "startTimeUnixNano": "1712932035014010700",
    "timeUnixNano": "1712935514982950700",
    "count": "634",
    "sum": 19.310237299999994,
    "bucketCounts": ["0", "0", "65", "563", "3", "1", "2",
      "0", "0", "0", "0", "0", "0", "0"],
    "explicitBounds": [0.005, 0.01, 0.025, 0.05, 0.075,
      0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10],
    "min": 0.020727,
    "max": 0.1898529
```

```
    },
    {
     "attributes": [
      {
       "key": "http.request.method",
       "value": {"stringValue": "POST"}
      },
      {
       "key": "http.response.status_code",
       "value": {"intValue": "200"}
      },
      {
       "key": "http.route",
       "value": {"stringValue": "/CudaHelio.Workers.
          ApiGateway.Proto.WorkersGateway/UploadJobResults
          "}
      },
      {
       "key": "network.protocol.version",
       "value": {"stringValue": "2"}
      },
      {
       "key": "url.scheme",
       "value": {"stringValue": "http"}
      }
     ],
     "startTimeUnixNano": "1712932035014010700",
     "timeUnixNano": "1712935514982950700",
     "count": "32",
     "sum": 1.1004444,
     "bucketCounts": ["0", "0", "0", "30", "2", "0", "0", "
        0", "0", "0", "0", "0", "0", "0", "0"],
     "explicitBounds": [0.005, 0.01, 0.025, 0.05, 0.075,
        0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10],
     "min": 0.0267483,
     "max": 0.070523
    }
   ],
   "aggregationTemporality": 2
  }
```

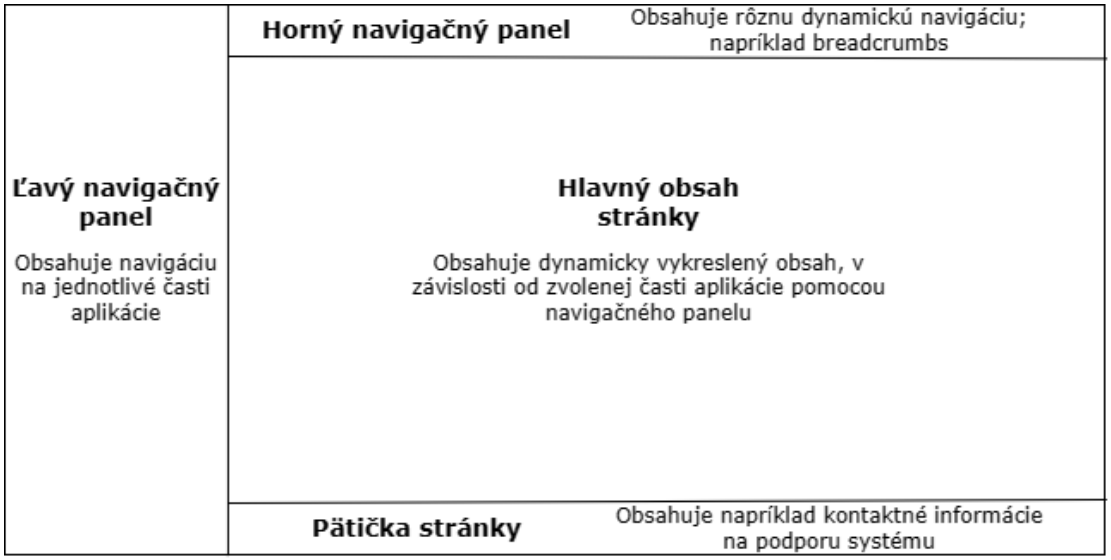
```
        }
       ]
      }
     ]
   }
 ]
}
```

B Používateľská príručka webovej aplikácie

Používateľské rozhranie je rozdelené na 3 hlavné časti, a to konkrétne:

- **Komponent hlavného navigačného panelu.** Nachádza sa v ľavej časti obrazovky a obsahuje hlavnú navigáciu medzi rôznymi časťami aplikácie.
- **Komponent doplnkového navigačného panelu.** Nachádza sa v hornej časti obrazovky a obsahuje doplnkovú navigáciu a statické odkazy.
- **Komponent hlavného obsahu stránky.** Zaberá zvyšok stránky a obsahuje všetok obsah stránky, v závislosti od aktuálnej stránky.

Okrem vymenovaných hlavných častí sa na konci komponentu hlavného obsahu každej stránky nachádza ešte **komponent pätičky stránky**. Obsahuje všeobecné informácie o aplikácii, resp. kontakt na podporu. Rozdelenie komponentov používateľského rozhrania je tiež znázornené na obrázku B.1.



Obr. B.1: Rozloženie komponentov používateľského rozhrania - webovej aplikácie

Prácu s distribuovaným systémom pomocou tejto webovej aplikácie je možné rozdeliť do 3 oblastí - **správa distribuovaných staníc**, **správa simulačných modelov** a **správa simulácií**. Návod na obsluhu každej z vymenovaných oblastí je popísaný zvlášť.

Správa distribuovaných staníc

Táto oblasť webovej aplikácie sa nachádza v ceste `/workers`. Pre otvorenie tejto oblasti aplikácie je možné použiť navigačné tlačidlo **Distributed Workers**, ktoré je k dispozícii v *komponente hlavného navigačného panelu*.

Na stránke v ceste `/workers` sa nachádza zoznam všetkých zaregistrovaných distribuovaných staníc v systéme. Každý záznam v zozname obsahuje informácie o **aktuálnom stave stanice**, **popis stanice**, **časovú značku vytvorenia stanice** a **časovú značku poslednej aktivity stanice**. Aktuálny stav stanice môže zobrazovať jednu z nasledujúcich hodnôt:

- **Working** - Stanica práve vykonáva nejakú zadanú distribuovanú úlohu.
- **Idle** - Stanica práve nevykonáva žiadnu zadanú distribuovanú úlohu, ale je dostupná a čaká na úlohu.
- **Offline** - Aktuálny stav stanice je neznámy, stanica svoj stav nehlási. Pri tomto prípade je veľmi pravdepodobné, že stanica je vypnutá.

Na tejto stránke sa nachádza taktiež navigačné tlačidlo **Create new worker**, pomocou ktorého je používateľ presmerovaný na cestu `/workers/create`. Na tejto ceste sa nachádza formulár pre zaregistrovanie novej distribuovanej stanice do systému. Po vyplnení potrebných údajov a stlačení tlačidla **Create** je distribuovaná stanica zaregistrovaná. Následne sa používateľovi zobrazí **autentifikačný token** pre distrubuovanú stanicu. *Poznámka: Tento token si je potrebné skopírovať a uchovať v bezpečí, nakoľko sa po prvom zatvorení už nedá žiadnym spôsobom znovu získať*. Následne pomocou tlačidla **Go back** je možné sa dostať späť na zoznam distribuovaných staníc, na ceste `/workers`. V aktualizovanom zozname sa bude nachádzať už aj nová zaregistrovaná stanica.

Správa simulačných modelov

Táto oblasť webovej aplikácie sa nachádza v ceste `/models`. Pre otvorenie tejto oblasti aplikácie je možné použiť navigačné tlačidlo **Simulation Models**, ktoré je k dispozícii v *komponente hlavného navigačného panelu*.

Na stránke v ceste `/models` sa nachádza zoznam všetkých simulačných modelov, dostupných v systéme. Každý záznam v zozname obsahuje informácie modeli, konkrétne jeho **názov**, **popis**, **typ** a **časovú značku vytvorenia**. Typ modelu môže zobrazovať jednu z nasledujúcich hodnôt:

- **Geliosphere** - Simulačný model je priamo definovaný v programe *Geliosphere*.
- **Custom** - Simulačný model je definovaný používateľom, pričom systém model skompiloval na paralelnú implementáciu používanú programom *Geliosphere*.

Navigačné tlačidlo **Create new simulation model**, pomocou ktorého je možné vytvárať nové simulačné modely, je dočasne **zakázaný**, nakoľko program *Autosphere* ešte nie je v plnej verzii, a teda nie je plne integrovaný s distribuovaným systémom.

Každý model v zozname obsahuje taktiež navigačné tlačidlo **See Details**, pomocou ktorého je používateľ presmerovaný na cestu `/models/<MODEL_ID>`. Na tejto ceste sú zobrazené detailné informácie o simulačnom modeli. Okrem vyššie spomenutých informácií, ktoré sú k dispozícii aj pri zobrazení zoznamu, táto stránka obsahuje taktiež **unikátny identifikátor modelu** a **detailné informácie o parametroch modelu**. Každý parameter modelu je popísaný jeho **kľúčom**, **popisom**, **dátovým typom** a **predvolenou hodnotou** (ak existuje).

Správa simulácií

Táto oblasť webovej aplikácie sa nachádza v ceste `/simulations`. Pre otvorenie tejto oblasti aplikácie je možné použiť navigačné tlačidlo **Simulations**, ktoré je k dispozícii v *komponente hlavného navigačného panelu*.

Na stránke v ceste `/simulations` sa nachádza zoznam všetkých simulácií, ktoré boli vytvorené v systéme. Každý záznam v zozname obsahuje informácie o simulácii modeli, konkrétne **aktuálny stav simulácie**, **názov simulácie**, **časovú značku vytvorenia** a **časovú značku poslednej aktualizácie**. Aktuálny stav simulácie môže zobrazovať jednu z nasledujúcich hodnôt:

- **Validating** - Simulácia je vytvorená ale systém ešte kontroluje niektoré detaily, ako napríklad parametre modelu.
- **ValidationFailed** - Vstupná kontrola simulácie zlyhala. *Poznámka: Toto je koncový stav simulácie*.

- **Processing** - Vstupná kontrola simulácie prebehla úspešne a simulácia bola zaradená na výpočet.
- **Completed** - Všetky výpočty sú dokončené a výsledky simulácie sú k dispozícii. *Poznámka: Toto je koncový stav simulácie*.
- **Failed** - Niektoré výpočty simulácie zlyhali, výsledky nie sú k dispozícii. *Poznámka: Toto je koncový stav simulácie*.

Každá simulácia v zozname taktiež obsahuje navigačné tlačidlo **See Details**, pomocou ktorého je používateľ presmerovaný na stránku s detailnými informáciami simulácie, ktorá sa nachádza na ceste `simulations/<SIMULATION_ID>`.

Na tejto stránke sa nachádza navigačné tlačidlo **Create new simulation**, pomocou ktorého je používateľ presmerovaný na cestu `/simulations/create`. Na tejto ceste sa nachádza formulár pre zaregistrovanie novej distribuovanej stanice do systému. Formulár je dvoj-krokový, pričom v prvom kroku je potrebné zvoliť si simulačný model zo zoznamu dostupných modelov. Zvolením simulačného modelu a následným kliknutím na tlačidlo **Select this model**, sa používateľ dostane do druhého kroku formulára. V druhom kroku je potrebné vyplniť formulár, ktorý obsahuje polia pre základné údaje o simulácii, ako aj polia pre všetky hodnoty parametrov zvoleného simulačného modelu. Po vyplnení potrebných údajov a stlačení tlačidla **Create simulation**, je simulácia vytvorená a používateľ je presmerovaný na stránku s jej detailnými informáciami, rovnako ako je to popísané vyššie.

Na ceste sú `simulations/<SIMULATION_ID>` sú zobrazené detailné informácie o simulácii modeli. Okrem vyššie spomenutých informácií, ktoré sú k dispozícii aj pri zobrazení zoznamu, táto stránka obsahuje taktiež **unikátny identifikátor simulácie**, **detailné informácie o parametroch simulácie a použitom modeli** a **priebežný postup výpočtu simulácie**, vyjadrený v percentách. V závislosti od aktuálneho stavu simulácie je dynamicky zobrazený rozdielny komponent, a to nasledovne:

- **V stave *Validating* nie je zobrazený žiaden prídavný komponent**.
- **V stave *ValidationFailed* sú zobrazené validačné chyby**, ktoré nastali pri overovaní simulácie.
- **V stave *Processing* je zobrazený zoznam s detailami o čiastkových distribuovaných výpočtoch**. Každý prvok zoznamu obsahuje **aktuálny stav výpočtu**, **veľkosť výpočtu** a **časovú značku poslednej aktualizácie výpočtu**.

Aktuálny stav čiastkového výpočtu môže zobrazovať jednu z nasledujúcich hodnôt:

  - **Processing** - Čiastkový výpočet je zaradený do fronty, resp. práve vykonávaný na distribuovanej stanici.
  - **Completed** - Čiastkový výpočet je dokončený.
  - **Failed** - Čiastkový výpočet zlyhal. V tomto prípade je celá simulácia označená ako *Failed*.

- **V stave *Failed* sú zobrazené chyby**, ktoré nastali pri výpočtoch simulácie.
- **V stave *Completed* sú zobrazené výsledky simulácie** vo forme grafu.

C Systémová príručka

Táto príručka obsahuje dokumentáciu jednotlivých komponentov distribuovaného systému. Taktiež poskytuje návod pre prípravu potrebného prostredia pre vývoj a spustenie projektu.

Distribuovaný systém sa skladá z nasledujúcich komponentov:

- Služba pre distribuované úlohy,
- Modelová služba,
- Simulačná služba,
- API brána pre distribuované stanice,
- Frontend BFF + Webová aplikácia - používateľské rozhranie,
- Distribuovaná stanica systému.

Lokálny vývoj a spustenie systému

Pre lokálny vývoj distribuovaného systému sú požadované nasledujúce predpoklady:

- **.NET 8 SDK**,
- **Prostredie Docker, resp. Podman**.

Po otvorení projektu je nutné ešte nainštalovať sadu nástrojov **.NET Aspire**, pomocou príkazu `dotnet workload restore`. Následne je možné spustiť celý distribuovaný systém, resp. jeho centralizované komponenty, pomocou projektu *Aspire.AppHost*, ktorý sa nachádza v priečinku `src/Aspire.AppHost`, príkazom `dotnet run --project ./src/Aspire.AppHost`. Nástroj *.NET Aspire* sa postará o prípravu potrebnej infraštruktúry, prípravy lokálnych projektov komponentov systému a následne všetky spustí a orchestruje.

Pre lokálne spustenie projektu distribuovanej stanice je odporúčané používať konfiguračný parameter `TestMode = true`. V inakšom prípade je nutné mať hardvérovú podporu grafickej karty *NVIDIA* a nainštalované ovládače *NVIDIA CUDA*. Lokálne spustenie projektu distribuovanej stanice je možné pomocou príkazu `dotnet run --project ./src/DistributedWorker.Controller`.

Služba pre distribuované úlohy

Táto služba sa skladá z 3 projektov:

- **Jobs.Core** - projekt typu *Class Library*. Obsahuje doménovú logiku služby, definíciu konečno-stavových automatov, jadro služby a implementáciu distribuovaného plánovača úloh.
- **Jobs.Infrastructure** - projekt typu *Class Library*. Obsahuje implementáciu rozhraní slúžiacich pre integráciu s externými službami, resp. inou infraštruktúrou. Taktiež obsahuje konfiguráciu entít pre rámec *Entity Framework Core* a vygenerované databázové migrácie.
- **Jobs.Service** - projekt typu *Worker Service*. Tvorí vrchnú vrstvu tejto mikroslužby a vytvára spustiteľnú aplikáciu. Inicializuje všetky nižšie vrstvy aplikácie.

Služba neposkytuje verejné REST API, všetka komunikácia je realizovaná vymieňaním správ, pomocou internej komunikačnej vrstvy, zabezpečovanej službou RabbitMQ. Služba počúva na nasledujúce udalosti:

- **CudaHelio.Common.Contracts.Jobs.ListWorkersRequest**

  Spracuje aktuálny zoznam distribuovaných staníc a vráti ho. Vráti správu typu
  CudaHelio.Common.Contracts.Jobs.ListWorkersResponse.

- **CudaHelio.Common.Contracts.Jobs.WorkerAuthenticationDataRequest**

  Vyhľadá údaje o distribuovanej stanici, podľa zadaného autentifikačného tokenu. Vráti správu typu
  CudaHelio.Common.Contracts.Jobs.WorkerAuthenticationDataResponse.

- **CudaHelio.Common.Contracts.Jobs.EnqueueJob**

  Založí novú úlohu do fronty úloh. Nevracia žiadnu správu.

- **CudaHelio.Common.Contracts.Jobs .WorkerJobLeaseProlongationRequest**

  Predĺži uzamknutie úlohy pre danú distribuovanú stanicu. Vráti správu typu *CudaHelio.Common.Contracts.Jobs .WorkerJobLeaseProlongationResponse*.

- **CudaHelio.Common.Contracts.Jobs.CompleteJob**

  Uloží výsledky úlohy a označí ju ako dokončená, resp. zlyhaná, podľa prijatej správy. Nevracia žiadnu správu.

- **CudaHelio.Common.Contracts.Jobs.CreateWorker**

  Zaregistruje novú distribuovanú stanicu do systému. Vráti správu typu *CudaHelio.Common.Contracts.Jobs.WorkerCreated*.

- **CudaHelio.Common.Contracts.Jobs.WorkerJobAssignmentRequest**

  Priradí distribuovanej stanici voľnú úlohu - pokiaľ taká existuje. Vráti správu typu *CudaHelio.Common.Contracts.Jobs.WorkerJobAssignmentResponse*.

Modelová služba

Táto služba sa skladá z jedného projektu, **Models.Service**. Je to projekt typu *Worker Service*. Tento projekt obsahuje doménovú logiku služby, implementáciu rozhraní integrácie s inými službami a infraštruktúrou, konfiguráciu entít pre rámec *Entity Framework Core* a vygenerované databázové migrácie. Taktiež tvorí vrchnú vrstvu tejto mikroslužby a vytvára spustiteľnú aplikáciu.

Služba neposkytuje verejné REST API, všetka komunikácia je realizovaná vymieňaním správ, pomocou internej komunikačnej vrstvy, zabezpečovanej službou RabbitMQ. Služba počúva na nasledujúce udalosti:

- **CudaHelio.Common.Contracts.Models.ListModelsRequest**

  Spracuje aktuálny zoznam simulačných modelov a vráti ho. Vráti správu typu *CudaHelio.Common.Contracts.Models.ListModelsResponse*.

- **CudaHelio.Common.Contracts.Models.GetModelByIdRequest**

  Vyhľadá simulačný model na základe poskytnutého identifikátora. Ak model existuje, vráti správu typu

CudaHelio.Common.Contracts.Models.GetModelByIdResponse.
Pokiaľ však model neexistuje, vráti správu typu
CudaHelio.Common.Contracts.Models.ModelNotFoundResponse.

- **CudaHelio.Common.Contracts.Models.ValidateModelParameters**

  Overí poskytnutú kolekciu parametrov voči validačným pravidlám špecifikovného simulačného modelu.

  Ak sú parametre platné, vráti správu typu
  CudaHelio.Common.Contracts.Models.ModelParametersValidationSuccess.
  Pokiaľ však parametre sú neplatné, resp. niektoré chýbajú, vráti správu typu
  CudaHelio.Common.Contracts.Models.ModelParametersValidationFailed.

Simulačná služba

Táto služba sa skladá z 3 projektov:

- **Simulations.Core** - projekt typu *Class Library*. Obsahuje doménovú logiku služby, definíciu konečno-stavových automatov a jadro služby.
- **Simulations.Infrastructure** - projekt typu *Class Library*. Obsahuje implementáciu rozhraní slúžiacich pre integráciu s externými službami, resp. infraštruktúrou. Taktiež obsahuje konfiguráciu entít pre rámec *Entity Framework Core* a vygenerované databázové migrácie.
- **Simulations.Service** - projekt typu *Worker Service*. Tvorí vrchnú vrstvu tejto mikroslužby a vytvára spustiteľnú aplikáciu. Taktiež inicializuje všetky nižšie vrstvy aplikácie.

Služba neposkytuje verejné REST API, všetka komunikácia je realizovaná vymieňaním správ, pomocou internej komunikačnej vrstvy, zabezpečovanej službou RabbitMQ. Služba počúva na nasledujúce udalosti:

- **CudaHelio.Common.Contracts.Simulations.ListSimulationsRequest**

  Spracuje aktuálny zoznam simulácii a vráti ho. Vráti správu typu
  CudaHelio.Common.Contracts.Simulations.ListSimulationsResponse.

- **CudaHelio.Common.Contracts.Simulations**
  .ListSimulationChunksRequest

  Spracuje aktuálny zoznam čiastkových výpočtov - chunks pre simuláciu a vráti ho. Vráti správu typu
  CudaHelio.Common.Contracts.Simulations.ListSimulationChunksResponse.

- **CudaHelio.Common.Contracts.Simulations.CreateSimulation**

  Vytvorí novú simuláciu v systéme. Nevracia žiadnu správu.

- **CudaHelio.Common.Contracts.Simulations.GetSimulationByIdRequest**

  Vyhľadá simuláciu na základe poskytnutého identifikátora. Ak simulácia existuje, vráti správu typu *CudaHelio.Common.Contracts.Simulations.GetSimulationByIdResponse*. Pokiaľ však model neexistuje, vráti správu typu *CudaHelio.Common.Contracts.Simulations.SimulationNotFoundResponse*.

- **CudaHelio.Common.Contracts.Simulations.ChunkCalculationDone**

  Spracuje dokončenie čiastkového výpočtu simulácie na úrovni celej simulácie. Nevracia žiadnu správu.

- **CudaHelio.Common.Contracts.Jobs.JobCompleted**

  Spracuje výsledok dokončenej distribuovanej úlohy. Táto služba spracováva len úlohy typu `SimulationChunkCalculation`. Nevracia žiadnu správu.

API brána pre distribuované stanice

Tento komponent systému sa skladá z 2 projektov:

- **Workers.ApiGateway.Proto** - projekt typu *Class Library*. Obsahuje *Protobuf* definície *gRPC* služieb a správ.
- **Workers.ApiGateway** - projekt typu *ASP.NET Core Web App*. Tvorí vrchnú vrstvu tohto komponentu a vytvára spustiteľnú aplikáciu. Obsahuje autentifikačné mechanizmy pre distribuované stanice a taktiež server implementáciu *gRPC* služieb, definovaných v predošlom spomínanom projekte.

Komponent *API brány* poskytuje verejné *gRPC* API, podľa **Protobuf definície** z ukážky C.1.

Zdrojový kód C.1: Definícia gRPC služby API brány pre distriuované stanice

```
syntax = "proto3";
package CudaHelio.Workers.ApiGateway.Proto;
import "workers.messages.proto";

service WorkersGateway {
  rpc RequestAJob (RequestAJobRequest)
```

```
    returns (RequestAJobResponse);
  rpc ProlongJobLease (ProlongJobLeaseRequest)
    returns (ProlongJobLeaseResponse);
  rpc UploadJobResults (UploadJobResultsRequest)
    returns (UploadJobResultsResponse);
  rpc GetModelById (GetModelByIdRequest)
    returns (GetModelByIdResponse);
}
```

Detailnú dokumentáciu **gRPC** API je možné vygenerovať príkazom z ukážky C.2 (vyžaduje si nainštalované prostredie Docker). Výsledná dokumentácia je vygenerovaná v priečinku `docs`.

Zdrojový kód C.2: Príkaz pre vygenerovanie podrobnej dokumentácie pre gRPC API

```
docker run --rm \
  -v $(pwd)/docs:/out \
  -v $(pwd)/src/Workers.ApiGateway.Proto/Protos:/protos \
  pseudomuto/protoc-gen-doc
```

Frontend BFF a webová aplikácia

Tento komponent sa skladá z jedného projektu, **WebApp**. Jedná sa o projekt typu *ASP.NET Core Web App*. Tento projekt implementáciu používateľského rozhrania, agregátor žiadostí a implementáciu rozhraní integrácie s inými službami a infraštruktúrou. Taktiež tvorí vrchnú vrstvu tejto mikroslužby a vytvára spustiteľnú aplikáciu.

Tento komponent poskytuje verejné API, ktoré vracia webové stránky - odpovede teda majú telo typu `text/html`. Implementované sú nasledujúce cesty:

- `/` - úvodná stránka webovej aplikácie.
- `/simulations` - stránka so zoznamom existujúcich simulácii v systéme.
- `/simulations/<SIMULATION_ID>` - stránka s detailnými informáciami o vybranej simulácii v systéme.
- `/simulations/create` - stránka s formulárom na vytvorenie novej simulácie.
- `/models` - stránka so zoznamom existujúcich simulačných modelov v systéme.

- `/models/<MODEL_ID>` - stránka s detailnými informáciami o vybranom simulačnom modeli v systéme.
- `/workers` - stránka so zoznamom existujúcich distribuovaných staníc v systéme.
- `/workers/create` - stránka s formulárom na registráciu novej distribuovanej stanice do systému.

Distribuovaná stanica systému

Tento komponent systému sa skladá z 2 projektov:

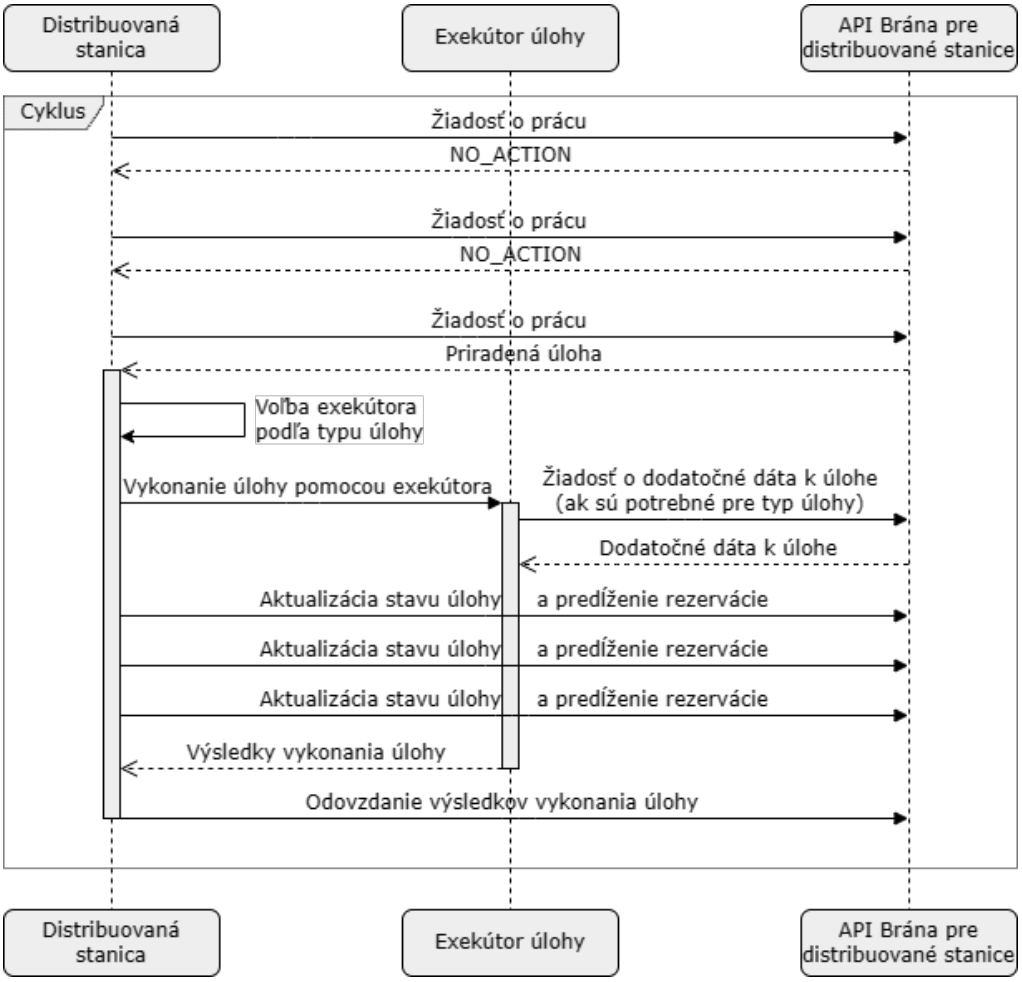
- **DistributedWorker.Executor** - projekt typu *Class Library*. Obsahuje doménové logiky a implementácie vykonávania jednotlivých typov distribuovaných úloh v systéme.
- **DistributedWorker.Controller** - projekt typu *Worker Service*. Tvorí vrchnú vrstvu tohto komponentu a vytvára spustiteľnú aplikáciu. Obsahuje implementáciu kontrolných mechanizmov exekútorov a komunikačných mechanizmov pre komunikáciu so zvyškom distribuovaného systému.

Tento komponent neposkytuje žiadne verejné API. Aplikácia je určená na beh ako služba na pozadí (angl. background daemon). Sekvencia vykonávania hlavného životného cyklu tejto aplikácie je znázornená sekvenčným diagramom na obrázku C.1.

Zdieľané moduly

Distribuovaný systém obsahuje 2 projekty zdieľaných modulov a to konkrétne:

- **Common.Contracts** - projekt typu *Class Library*. Obsahuje definície správ a ďalších kontraktov, ktoré sú používané v celom distribuovanom systéme.
- **ServiceDefaults** - projekt typu *Class Library*. Obsahuje základné konfigurácie komponentov systému, ktoré sú používané v takmer všetkých komponentoch systému. Zaraďuje sa sem **konfiguráciu telemetrie**, pomocou *OpenTelemetry*, **konfigurácia distribuovaného rámca MassTransit**, **konfiguráciu HTTP API na kontrolu stavu komponentu**, a podobne.



Obr. C.1: Sekvencia hlavného životného cyklu distribuovanej stanice systému

Okrem zdieľaných modulov, obsahuje distribuovaný systém ešte špeciálny projekt **Aspire.AppHost**. Tento projekt slúži na orchestráciu systému pri lokálnom vývoji. Tvorí spustiteľnú aplikáciu, ktorá je pomocou nástroja *.NET Aspire* spustená a vďaka nej dokáže tento nástroj orchestrovať všetky definované komponenty systému.