Nástroj pro distribuci komplexních výpočtů z oblasti umělé inteligence

Bc. Pavel Babák

Diplomová práce
2021

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Pavel Babák**
Osobní číslo:	**A16724**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**Kombinovaná**
Téma práce:	**Nástroj pro distribuci komplexních výpočtů z oblasti umělé inteligence**
Téma práce anglicky:	**A Distributed Computing Tool for Complex Artificial Intelligence Tasks**

Zásady pro vypracování

1. Vypracujte literární rešerši na dané téma.
2. Zvolte vhodnou metodu pro implementaci distribuovaných výpočtů.
3. Implementujte zvolenou metodu na výpočetní technice dostupné v rámci laboratoře umělé inteligence na FAI UTB.
4. Experimentálně ověřte funkčnost řešení na vybraných testovacích scénářích.
5. Věnujte pozornost zabezpečení aplikace.
6. Vytvořte dokumentaci k vytvořenému nástroji pro distribuci výpočtů.

Forma zpracování diplomové práce: **Tištěná/elektronická**

Seznam doporučené literatury:

1. ŽÁRA, Ondřej. *JavaScript: programátorské techniky a webové technologie*. Brno: Computer Press, 2015, 180 s. ISBN 9788025145739. Dostupné také z: http://knihy.cpress.cz/K2209
2. DARWIN, Ian F. Java: kuchařka programátora: [vzory a řešení pro vaše aplikace]. Brno: Computer Press, 2006, 798 s. ISBN 8025109445.
3. SPELL, Brett. Java: programujeme profesionálně. Praha: Computer Press, 2002, 1022 s. Programujeme profesionálně. ISBN 8072266675.
4. PREVE, Nikolaos P., ed. *Grid computing: towards a global interconnected infrastructure*. London: Springer, c2011, xv, 312 s. Computer communications and networks. Dostupné z: doi:9780857296764
5. DOSTÁLEK, Libor. *Velký průvodce protokoly TCP/IP: bezpečnost*. Praha: Computer Press, 2001, xvi, 565 s. Komunikace & sítě. Rychle a jistě. ISBN 807226513X.

Vedoucí diplomové práce: **Ing. Adam Viktorin**
Ústav informatiky a umělé inteligence

Konzultant diplomové práce: **Ing. Tomáš Kadavý**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **15. ledna 2021**
Termín odevzdání diplomové práce: **17. května 2021**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D. v.r.
ředitel ústavu

Ve Zlíně dne 15. ledna 2021

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 17.5.2021

Pavel Babák v.r.
podpis autora

ABSTRAKT

Cílem diplomové práce bylo vytvořit nástroj, který by umožňoval distribuci komplexních výpočtů v rámci výpočetního gridu. Teoretická část pojednává o současném využití umělé inteligence a také o možných technologiích pro implementaci výpočetního gridu. Výsledný návrh řešení je založen na technologiích Java, REST a Apache ActiveMQ Artemis a jeho současné využití je pro testování benchmarkových algoritmů v rámci laboratoře umělé inteligence.

Klíčová slova: grid, grid computing, Java, REST, Apache ActiveMQ Artemis

ABSTRACT

The aim of the diploma thesis was to create a tool that would enable distribution of complex calculations within the grid computing. The theoretical part deals with the current use of artificial intelligence and also with possible technologies for the implementation of the grid computing. The final design of the solution is based on Java, REST and Apache ActiveMQ Artemis technologies and its current use is for testing benchmark algorithms within the artificial intelligence laboratory.

Keywords: grid, grid computing, Java, REST, Apache ActiveMQ Artemis

Rád bych poděkoval vedoucímu své diplomové práce Ing. Adamovi Viktorinovi za množství konzultací, cenné rady a připomínky. Dále bych chtěl poděkovat svému kolegovi z práce Vladimírovi Blažíkovi za jeho rady ohledně technologií. V neposlední řadě bych chtěl poděkovat všem přátelům, kteří mě v této době podporovali.

OBSAH

ÚVOD

Umělá inteligence je jedno z nejrychleji se rozvíjejících odvětví informačních technologií. V dnešní době je již téměř všudypřítomná. Ovlivňuje naše rozhodování. Určuje, jaké příspěvky uvidíme na sociálních sítích a vytváří řadu různých doporučení, kterými ovlivňuje naše podvědomí. Pomáhá ale také ve výzkumu a ve zdravotnictví. Umožňuje věci, které by dříve nebyly vůbec možné.

Obsahem této práce je nejdříve seznámení s oblastí umělé inteligence a jejím praktickým využitím. Poté je v práci popsána složitost úloh, která je hlavním důvodem pro vytvoření výpočetního gridu. V další části je popsána architektura gridu a možné způsoby rozdělování úloh na výpočetní uzly. Velký prostor je věnován srovnání jednotlivých technologií pro implementaci výpočetního gridu. V praktické části je pak především popis vytvořeného systému a je zde uveden způsob jeho využití.

I. TEORETICKÁ ČÁST

1 UMĚLÁ INTELIGENCE

Umělá inteligence označuje systémy nebo stroje, které napodobují lidskou inteligenci k plnění úkolů a mohou se iterativně zlepšovat na základě shromážděných dat.

1.1 Typy umělé inteligence

Rozeznáváme tři základní typy umělé inteligence.

1.1.1 Omezená umělá inteligence

Omezená umělá inteligence (Artificial Narrow Intelligence) označuje schopnost počítačového sytému provádět úzce definovaný úkol lépe než člověk. Tento typ představuje v současnosti nejvyšší úroveň vývoje umělé inteligence. Důvodem je to, že i když se zdá, že umělá inteligence sama myslí v reálném čase, ve skutečnosti koordinuje několik omezených procesů a rozhoduje se v mezích předem určeného rámce. Její myšlení nezahrnuje vědomí ani emoce [1].

1.1.2 Obecná umělá inteligence

Obecná umělá inteligence (Artificial General Intelligence) označuje schopnost počítačového systému překonávat lidi v jakýchkoliv intelektuálních úkolech. Počítačový systém, který by dosáhl této úrovně, by byl schopný řešit velmi složité problémy, uplatňovat úsudek v nejasných situacích a při uvažování uplatňovat předchozí znalosti. Byl by schopný tvořivosti a představivosti srovnatelné s člověkem a dokázal by řešit mnohem širší rozsah úkolů než omezená umělá inteligence [1].

1.1.3 Umělá super inteligence

Počítačový systém, který by dosáhl umělé super inteligence (Artificial Super Intelligence), by byl schopný překonávat lidi téměř ve všech oblastech, včetně vědecké tvořivosti, všeobecných znalostí a sociálních dovedností [1].

1.2 Praktické využití umělé inteligence

Umělá inteligence je dnes již všudypřítomná. Setkáváme se s ní v každodenním životě, například ve fulltextových vyhledávačích na internetu, v marketingu, textových překladačích a stále více i v bankovnictví. Bezstarostně si od ní necháváme doporučit produkty podle našich zájmů a preferencí. Umělá inteligence nachází své uplatnění v komunikaci se zákazníky, ale také při vývoji autonomních vozidel, řízení elektrických sítí, údržbě strojů a

zařízení či v medicíně. Zvyšuje tak produktivitu práce, ale také umožňuje dělat věci, které by dříve nebyly možné.

1.2.1 Autonomní vozidla

Mezi nejkomplexnější příklady umělé inteligence na světě patří systémy na řízení a ovládání autonomních vozidel a dalších autonomních prostředků. Tyto systémy koordinují několik procesů a tím simulují uvažování lidských řidičů. S využitím rozpoznávání obrazu identifikují značky, semafory, dopravní provoz a překážky. Optimalizují trasy k cílům a v reálném čase odesílají a přijímají data, což jim umožňuje aktivně diagnostikovat problémy a aktualizovat svůj software [1].

1.2.2 Roboti a digitální asistenti

Konverzace jsou přirozeným způsobem komunikace lidí a s tím, jak se rozvíjí technologie umělé inteligence, jsou konverzační rozhraní čím dál rozšířenější. Některá rozhraní slouží úzce vymezenému účelu. Další se chovají více jako osobní asistenti, kteří dokáží pomoci s celou řadou úkolů. Ale všechna konverzační rozhraní k interpretaci požadavků (označovaných také jako výroky) a poskytování odpovědí s relevantními informacemi využívají rozpoznávání přirozeného jazyka (Natural Language Understanding) [1].

1.2.3 Moduly doporučování

Jedním z nejběžnějších způsobů použití umělé inteligence je doporučování položek na základě historických dat. Například když vám služba streamování médií doporučuje, co sledovat nebo poslouchat dál, s využitím umělé inteligence analyzuje, co jste sledovali nebo poslouchali v minulosti, na základě atributů vyfiltruje všechny dostupné možnosti a doporučí možnost s největší pravděpodobností vás zabavit. Podobným způsobem se umělá inteligence využívá na webech, kde se při nákupu zobrazuje doporučené příslušenství nebo související položky k přidání do košíku [1].

1.2.4 Filtry nevyžádané pošty

Celá řada emailových platforem využívá umělou inteligenci k zajištění, aby emailová schránka nebyla zahlcená nevyžádanou poštou. Když do systému přijde nový email, umělá inteligence ho analyzuje a pokusí se odhalit signály, které značí nevyžádanou poštu. Pokud email splní dostatek kritérií, označí se za nevyžádanou poštu a umístí se do karantény. Systém se učí a upravuje své parametry na základě zpětné vazby (prostřednictvím oprav

nesprávných příznaků nebo označováním e-mailů, které filtr nezachytil, příznakem nevyžádané pošty) [1].

1.2.5 Technologie chytrých domácností

Téměř vše, co v moderní domácnosti zajišťuje automatizaci, využívá umělou inteligenci. Příkladem můžou být inteligentní žárovky, které naslouchají příkazům, inteligentní termostaty, které se učí preference a během dne se jim přizpůsobují, nebo inteligentní vysavače, které se učí procházet domácností bez pokynů. [1]

1.2.6 Analýza zdravotnických dat

Umělá inteligence pomáhá zdravotnickým organizacím po celém světě s výzkumem, testováním, diagnostikou, léčbou a monitorováním. Některé zdravotnické organizace využívají umělou inteligenci k analýze vzorků tkání a určování přesnějších diagnóz. Umělá inteligence je také využívána k analýze klinických dat a odhalování nedostatků v léčbě pacientů. Při vývoji nových léků se umělá inteligence využívá k analýze miliard sloučenin, která pomáhá chemikům rychleji odhalovat nové objevy a identifikovat vhodné kandidáty pro klinické studie [1].

1.2.7 Hraní deskových her

Zástupcem této kategorie je například počítačový program AlphaGo Zero, který hraje deskovou hru Go. Byl vytvořen společností Google DeepMind a stal se prvním počítačovým programem, který v roce 2017 porazil světovou jedničku ve hře Go čínského hráče 柯洁 (Ke Jie).

Go se hraje na čtvercové hrací desce 19x19 průsečíků, kde proti sobě hrají 2 hráči, jeden s černými kameny a druhý s bílými kameny. Během hry se hráči snaží pokládáním svých kamenů na průsečíky obsadit co největší území a zajmout soupeřovi kameny. Cílem je mít na konci hry větší skóre, které je tvořeno zajatými kameny a obklíčenými průsečíky [2].

Kvůli své složitosti je Go jedna z nejnáročnějších klasických her pro umělou inteligenci. I přes desítky let práce mohly nejsilnější počítačové programy Go hrát pouze na úrovni lidských amatérů, protože standardní metody AI, které testují všechny možné tahy a pozice pomocí vyhledávacího stromu, nezvládají vyhodnotit všechny tyto kombinace.

AlphaGo Zero, ale kombinuje vyhledávací strom Monte Carlo s hlubokými neuronovými sítěmi. Tyto neuronové sítě berou popis hrací desky jako vstup a zpracovávají ji prostřednictvím řady různých síťových vrstev obsahujících miliony neuronových spojení. Jedna neuronová síť „policy network“ vybere další tah a druhá neuronová síť „value network“, předpovídá vítěze hry. AlphaGo Zero se hru naučil jen hraním proti sobě počínaje zcela náhodnou hrou, přičemž ale rychle překonal výkon všech svých předchozích verzí a také objevil nové znalosti, rozvíjel nekonvenční strategie a kreativní nové tahy, včetně těch, kterými prorazil světové šampiony Lee Sedol a Ke Jie [2].

Obrázek 1. Superpočítač AlphaGo

1.3 Celulární automaty

Architektura výpočetního gridu se dá modelovat jako celulární automat. Celulární automat (CA) je dynamický systém, diskrétní v hodnotách, prostoru i čase. Je tvořen pravidelnou

strukturou buněk v N-rozměrném prostoru (nejčastěji je N=2, tzv. 2D CA, kde buňky tvoří čtvercovou mřížku). Každá buňka může nabývat jeden z K možných stavů. Často jde pouze o dva stavy: 0 - mrtvá buňka, 1 - živá buňka. V tomto případě se stav 1 označuje jako buňka a 0 jako prázdné políčko (mřížky). Hodnoty stavů buněk v dalším časovém kroku (v následující generaci) se vypočtou synchronně na základě lokální přechodové funkce (stejně jako všechny buňky). Argumenty této funkce jsou aktuální hodnoty stavů vyšetřované buňky a všech sousedů (buněk v jejím okolí). V případě 1D CA je okolí charakterizováno tzv. poloměrem, tj. počtem sousedů po obou stranách vyšetřované buňky. V případě 2D CA tvoří okolí čtyři přilehlé buňky (tzv. neumannovské okolí), anebo se do okolí zařadí i čtyři další sousedi dotýkající se vyšetřované buňky jen v rozích (tzv. úplné okolí). Zpravidla se předpokládá, že struktura buněk je nekonečná. V praktických realizacích se buď předpokládají okrajové buňky identicky za nulové (prázdné), nebo jsou okraje propojeny a tvoří v případě 1D smyčku a v případě 2D anuloid. Některé z K možných stavů jsou označovány za klidové, když buňka v klidovém stavu má ve svém okolí také jenom buňky v klidovém stavu, potom se hodnota jejího stavu v další generaci nemění [4].

Pro celulární automaty jsou charakteristické 3 klíčové vlastnosti:

- **Paralelismus** – Výpočet nových hodnot stavů všech prvků probíhá současně.
- **Lokalita** – Nový stav prvků závisí jen jeho původním stavu a na původních stavech prvků z jeho okolí
- **Homogenita** – Pro všechny prvky platí stejná lokální přechodová funkce.

Nevyžaduje se přitom nutně pravidelná prostorová struktura, proto je v uvedené charakteristice použit pojem prvek místo buňka.

Základní rozdělení celulárních automatů:

- **Statické** – Mění se pouze stav každé z buněk, ale nemění se jejich počet a struktura.
- **Dynamické –** Každá z buněk je v další generaci nahrazena konečným počtem nových buněk s definovanými stavy.

2 ÚLOHY P / NP

Úlohy (algoritmy) rozdělujeme do několika kategorií podle jejich časové nebo paměťové náročnosti. Paměťová náročnost představuje požadavek na velikost paměti počítače, jež je zapotřebí k provedení úlohy. Podobně časová náročnost představuje čas potřebný pro vykonání úlohy. Dále se budeme zabývat pouze časovou náročností, protože paměťová náročnost funguje analogicky.

U NP-úloh může být jejich časová složitost i přes použití heuristické analýzy příliš vysoká a je proto vhodné rozdělit části výpočtu na více uzlů v rámci výpočetního gridu. Dosáhneme tím snížení zátěže pro jeden výpočetní uzel a rychlejšího zpracování úlohy díky distribuci výpočtu na více strojů.

2.1 P-úlohy

Jedná se o úlohy, jejich řešení lze nalézt deterministickým Turingovým strojem v polynomiálním čase. Úlohu (algoritmus) nazveme řešitelnou v polynomiálním čase, jestliže její časovou složitost můžeme shora ohraničit polynomem. Třídu úloh s polynomiální složitostí označíme P. Úlohy třídy P považujeme za řešitelné (v přiměřeném čase). Například třídění na haldě je úloha třídy P, neboť má složitost $n \log n \leq n^2$.

Definice: *„Třída P je třída všech rozhodovacích úloh U, pro něž existuje polynomiální algoritmus, který řeší U, tj. algoritmus, který má složitost O(p(n)) pro nějaký polynom p(n).“* [4]

2.2 NP-úlohy

Jedná se o úlohy, které lze řešit v polynomiálně omezeném čase na nedeterministickém Turingově stroji, který umožnuje v každém kroku rozvětvit výpočet na *n* větví, v nich se následně řešení hledá paralelně. Úlohu nazveme nedeterministicky polynomiální, jestliže existuje nedeterministický algoritmus, který ji řeší v polynomiálním čase. Tuto třídu úloh označíme NP.

Typickým představitelem třídy NP je problém, který je řešen algoritmem (s exponenciální časovou složitostí) procházejícím všechny varianty, kde ověření správnosti každé z těchto variant vyžaduje polynomický čas.

2.3 NP-úplné úlohy

Definice: *„Rozhodovací úloha U je NP-úplná (NP-complete), jestliže je NP-úlohou a každá NP-úloha se na U polynomiálně redukuje. Třídu všech NP-úplných úloh označujeme NPC.“* [4]

Jinými slovy, NP-úplné úlohy jsou ty úlohy, které jsou nejtěžší mezi všemi NP-úlohami. V praxi se NP-úplné úlohy (s většími vstupy) obvykle řeší pouze přibližně (heuristickými algoritmy, genetickými algoritmy, …). Tím se za cenu vzdání se nároků na nalezení přesného řešení dosahuje prakticky použitelných časů.
NP-úplných problémů jsou tisíce. Příkladem NP-úplné úlohy je nalezení hamiltonovské kružnice v neorientovaném grafu. Kružnice se nazývá hamiltonovskou, jestli obsahuje všechny vrcholy právě jednou.

2.4 NP-těžké úlohy

Definice: *„Řekněme, že rozhodovací úloha U je NP-těžká (NP-hard), jestliže se na ni polynomiálně redukuje každá NP-úloha.“* [4]
Z tranzitivity redukce úloh dostáváme, že NP-těžké jsou ty úlohy, na které se polynomiálně redukuje některá NP-úplná úloha. Úloha je NP-těžká, je-li alespoň tak těžká jako NP-úplné úlohy (možná i těžší).

2.5 Problém P versus NP

Problém P versus NP je důležitý otevřený problém v teoretické informatice. Pokud by byl nalezen polynomiální deterministický algoritmus pro některou z NP-úplných úloh, znamenalo by to, že všechny nedeterministicky polynomiální problémy jsou řešitelné v polynomiálním čase a tedy že NP = P. Otázka, zda nějaký takový algoritmus existuje, zatím nebyla vyřešena, předpokládá se však, že NP ≠ P. Vztah mezi třídami P a NP je jedním ze sedmi problémů tisíciletí, které vypsal Clayův matematický ústav v roce 2000 a za jehož vyřešení je nabízena finanční odměna ve výši 1 000 000 USD [5].

3 GRID

Grid computing je metoda pro zapojení různých výpočetních zdrojů propojených sítí tak, aby mohly být využity pro řešení rozsáhlých úkolů. Rozložením zátěže do jednotlivých uzlů gridu se docílí výrazně rychlejšího zpracování úlohy, než kdyby byla zpracovávána na jednom výpočetním zdroji.

Myšlenka gridu byla poprvé objevena počátkem 90. let. Carl Kesselman, Ian Foster a Steve Tuecke vyvinuli standard Globus Toolkit, který zahrnoval grid pro správu ukládání dat, zpracování dat a správu výpočtů [6].

Využití gridu je zvláště vhodné pro aplikace, ve kterých lze nezávisle na sobě provádět více paralelních výpočtů bez toho, aniž by bylo nutné provádět jejich synchronizaci nebo sdílení dat mezi jednotlivými stanicemi zapojenými do gridu.

Jednou z nevýhod gridu, v případě jeho použití ve veřejné síti, je, že počítače, které provádějí výpočty, nemusí být zcela důvěryhodné. Je pak vhodné zavést taková opatření, aby se minimalizovali riziko, že některé výpočetní uzly budou produkovat falešné, zavádějící nebo chybné výsledky. Jedním z takových opatření může být například vykonání stejné výpočetní úlohy více uzly a porovnáním jejich výsledků. Nalezené nesrovnalosti by identifikovaly nefunkční a škodlivé uzly. Dalším problémem ve veřejné síti mohou být pak výpadky síťového připojení, kdy některé z uzlů mohou být dočasně nedostupné. Tento problém lze ošetřit tak, že uzly dostanou ke zpracování větší blok dat, čímž se sníží potřeba nepřetržitého síťové komunikace se serverem a případně opětovným delegováním úlohy v gridu, když daný uzel nevrátí své výsledky v očekávaném čase [7]. Při použití vnitřní sítě (intranetu) se však tato dvě výše uvedená rizika minimalizují.

Výpočetní grid slouží pro poskytování služeb v oblasti distribuovaných výpočtů. Tato koncepce má za cíl výrazně zkrátit čas potřebný pro vykonání výpočetní úlohy [8].

Výpočetní grid se skládá z těchto tří typů strojů:

- **Grid server** udržuje aktuální seznam klientských počítačů připojených k síti a zajišťuje distribuci úkolů pomocí plánovače úloh.
- **Klientský počítač** poskytuje svou výpočetní kapacitu a případně uložiště pro konkrétní úlohy zadané plánovačem.
- **Uživatelský počítač** vstupuje do systému prostřednictvím požadavku na server o vykonání zvolené výpočetní úlohy.

Datový grid slouží jako rozsáhlé datové uložiště, které poskytuje data napříč virtuálními organizacemi. Hlavní výhodou je sdílení datových struktur a jejich údržba. To má pozitivní vliv na jednotnou správu dat a správu bezpečnostních politik [8].

Cloud computing je obecné označení pro gridové služby, které v sobě spojují oblast datového a výpočetního gridu, jejichž hlavním cílem je komerční poskytování hardwarových a softwarových služeb. To má pozitivní vliv na správu IT infrastruktury v řadě organizací. Data v cloudech však mohou na druhou stranu znamenat vyšší zranitelnost a existenční závislost na poskytovateli cloudových služeb. Rozdíl mezi gridovou a cloudovou službou je v charakteru zpracovávaných úloh. Cloud je především využíván pro on-line uživatelské služby a transakce. Jedním z jeho představitelů je například cloud S/4 HANA od společnosti SAP [9].

3.1 Plánovač úloh

Plánovač úloh se stará o optimální rozložení zátěže v prostředí gridu, kde dostupné zdroje mohou mít různou výpočetní kapacitu.

3.1.1 Topologie plánovačů

Plánovače gridových služeb můžeme rozdělit do tří nejčastěji se vyskytujících kategorií:

- **Centrální plánovač** řeší plánování úloh v rámci celého systému. Jeho výhodou je, že má přehled o všech dostupných zdrojích a všech výpočetních úlohách a může tak nabídnout lepší optimalizaci plánovacího rozvrhu. Nevýhodou je, že tento systém není příliš škálovatelný se zvyšujícím se počtem výpočetních zdrojů. Jeho nasazení je tedy vhodné spíše pro malé sítě v rámci intranetu. Další nevýhodou je, že při jeho výpadku dojde k výpadku celé gridové služby [9].
- **Systém decentralizovaných plánovačů** bývá navržen tak, aby každý z plánovačů měl na starosti určitou lokální oblast. Úkoly směřované do konkrétního geografického uzlu jsou tak obvykle distribuovány přes tento lokální plánovač. Výhodou je, že při výpadku některé z komponent systému je možné výpočet odklonit jinam. Tento systém také poskytuje vyšší škálovatelnost a možnost rozšiřování systému. Nevýhodou je, že tento systém nemusí do plánovacího rozvrhu zahrnout nevytížené zdroje z jiné lokality [9].
- **Hierarchická struktura plánovačů** v sobě spojuje výhody centralizovaného i decentralizovaného systému plánování. Hlavní plánovač se snaží o optimální rozložení

výpočetního výkonu v jednotlivých větvích gridové služby. V této struktuře mohou plánovače zvolit odlišnou strategii při rozdělování úloh, než jakou uplatňuje hlavní plánovač či plánovač na jiném stupni hierarchie [9].

3.1.2 Strategie plánovačů

Algoritmy pro plánování jsou srdcem plánovacích systémů. Dobrý plánovací algoritmus by měl vytvořit téměř optimální prováděcí plán, na jehož sestavení by však neměl spotřebovat příliš mnoho zdrojů nebo času.

First-Come-First-Served (FCFS) je základní plánovací algoritmus, kdy jsou úlohy odbavovány ve stejném pořadí, v jakém přišly do fronty bez jakýchkoliv dalších preferencí. Jedná se o spravedlivý způsob obsluhy požadavků, protože čas dokončení úlohy je nezávislý na jakékoliv úloze odeslané po něm. FCFS se snadno implementuje a vyžaduje jen velmi malé výpočetní úsilí. Nevýhodou je, že některé jednoduché úlohy mohou čekat příliš dlouho ve frontě, než budou odbaveny úlohy složité [10].

Backfilling Scheduling je vylepšená verze algoritmu FCFS. Pokud úlohu v čele seznamu nelze spustit z důvodu nedostatku dostupných zdrojů, pak se plánovač pokusí najít jinou úlohu, kterou lze spustit s dostupnými prostředky za předpokladu, že to nezdrží plánovaný začátek úlohy na začátku seznamu. Tento mechanismus se snaží zaplnit mezery v časových slotech procesoru. Důležité faktory, které ovlivňují plánování úlohy jsou doba běhu úlohy (délka) a počet výpočetních uzlů vyžadovaných úlohou (šířka) [10].

K dispozici jsou dvě varianty tohoto algoritmu:

- **Conservative Backfilling**: V této variantě jsou zdroje vyhrazeny pro každou úlohu, když vstoupí do fronty. Úlohy se mohou ve frontě posunout dopředu, pokud se nezpozdí žádná úloha ve frontě nad svůj vyhrazený čas zahájení. Tento přístup není nijak agresivní při předbíhání ve frontě, a tak pro dlouhé úlohy je obtížné posunout se v pořadí směrem nahoru. Z tohoto přístupu však těží krátké úlohy s vyššími nároky na počet výpočetních uzlů [10].
- **Aggressive Backfilling**: Rezervace je vyhrazena pouze úloze na začátku fronty. Ostatní úlohy se mohou posouvat ve frontě za předpokladu, že nezdržují plánovaný začátek této úlohy. Delší úlohy se tak mohou dostat dříve na řadu kvůli přítomnosti pouze jedné blokující rezervace v prováděcím plánu. Znevýhodněny jsou však širší úlohy, které mohou být neustále přebíhány až do doby, kdy najdou dostatek volných výpočetních uzlů [10].

Selektivní rezervace využívá obou těchto variant algoritmu, ale pro každou z těchto variant má samostatnou frontu. První fronta, ve které není zaručen čas zahájení úlohy a druhá fronta, ve které jsou všem úlohám přiděleny počáteční časy zahájení prostřednictvím rezervace. Když úloha vstoupí do systému, je nejdříve zařazena do první fronty. Když čekací doba úlohy překročí prahovou hodnotu, je selektivně vybrána a přesunuta do druhé fronty, která již zaručuje čas startu [10].

Náhodné plánování: Mechanismus náhodného plánování je nedeterministickým přístupem k plánování úloh, ve kterém je náhodně vybrána další úloha, která má být provedena ze všech úloh čekajících ve frontě. Žádný z úkolů nemá vyšší prioritu, jen dříve přijaté úlohy mají vyšší šanci na provedení [11].

4 POPIS TECHNOLOGIÍ

Pro implementaci výpočetního gridu lze využít rozličné technologie a případně i jejich kombinaci. V následující částí budou popsány hlavní technologie, jejich výhody a nevýhody a také důvody pro výběr finálního řešení použitého v praktické části.

4.1 Java RMI

Java Remote Method Invocation (**Java RMI**) je technologie programovacího jazyka Java, která umožňuje z jednoho virtuálního stroje (Java Virtual Machine) volat metody objektů na jiném virtuálním stroji, který běží na jiném počítači. Výhodou RMI je, že programátor zachází se vzdáleným objektem, jako by to byl objekt místní. Rozhraní a třídy, které jsou zodpovědné za funkčnost RMI, jsou dostupné v balíčku *java.rmi* [12].

4.1.1 Architektura

Architektura Java RMI se skládá z několika vrstev, kterými prochází komunikace mezi klientskou a serverovou aplikací. Objekty stub a skeleton patří do takzvané proxy vrstvy.

Stub je objekt na straně klienta a slouží pro volání metod vzdálených objektů. Je tedy jakousi reprezentací vzdáleného objektu na klientovi a implementuje stejné rozhraní jako vzdálený objekt. Při volání metody na stubu provede stub následující kroky:

- Naváže spojení s vzdáleným JVM, který obsahuje vzdálený objekt, prostřednictvím vrstvy vzdálené reference (Remote Reference Layer).
- Serializuje parametry metody do streamu (marshalling).
- Informuje vrstvu vzdálené reference o požadavku na vyvolání metody.
- Počká na dokončení vzdálené metody.
- Deserializuje vrácenou hodnotu ze streamu (unmarshalling).
- Vrátí výsledek klientovi.
- Informuje vrstvu vzdálené reference, že volání bylo dokončeno [14].

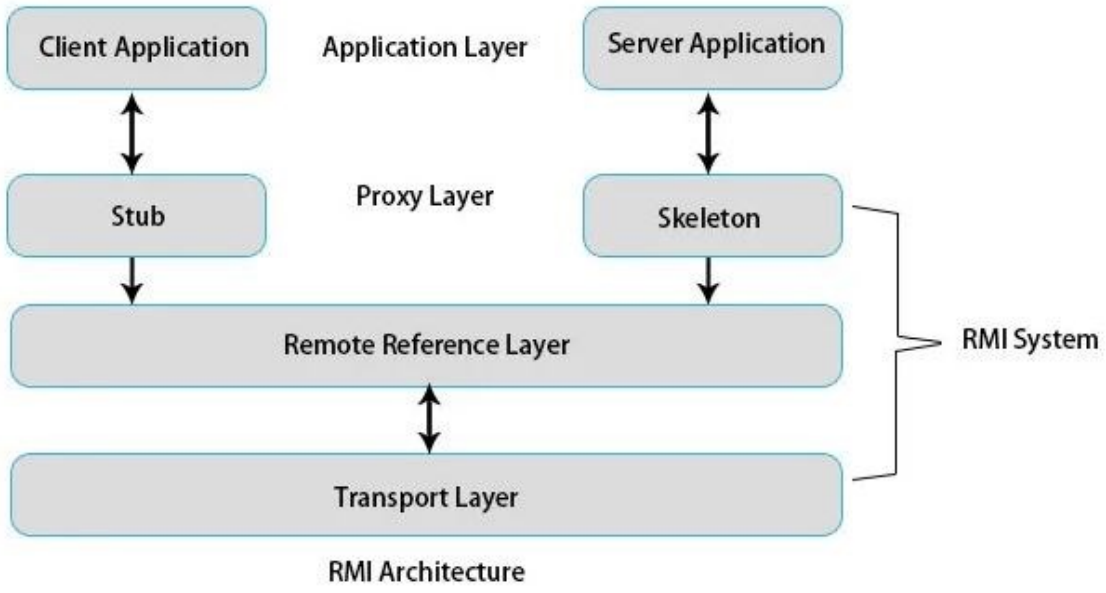
Skeleton je protějšek stubu na straně serveru a má na starosti tyto činnosti:

- Rozbalení parametrů ze streamu
- Vyvolání metody skutečného objektu
- Zachycení návratové hodnoty nebo výjimky této metody, která byla zavolána
- Zabalení návratové hodnoty nebo výjimky do streamu (marshalling)
- Odeslání přes vrstvu vzdálené reference zpátky na klienta [14]

Dřívější verze Javy vyžadovaly přítomnost skeletonu. Od Java 2 není skeleton vyžadován. Pro generování stubů a případně i skeletonů se používá RMI překladač *rmic* [15].

Vrstva vzdálené reference (Remote Reference Layer) je vrstva, která je odpovědná za udržování relace během volání metody. Spravuje odkazy klienta na objekt vzdáleného serveru. Tato vrstva je také zodpovědná za manipulaci s duplikovanými objekty [13].

Transportní vrstva (Transport Layer) vytváří a udržuje spojení mezi dvěma JVM. Tato vrstva využívá pro komunikaci standardní protokol TCP/IP. Nad protokolem TCP/IP RMI využívá protokol Java Remote Method Protocol (JRMP). Od verze 1.3 Java 2 SDK je k dispozici další verze RMI zvaná RMI/IIOP, která namísto JRMP používá Internet Inter-ORB Protocol (IIOP) [12].



Obrázek 2. Architektura Java RMI [13]

4.1.2 Předávání parametrů

V RMI lze jako parametr či návratovou hodnotu posílat veškeré primitivní typy, vzdálené objekty a objekty, které implementují rozhraní *java.io.Serializable*. Posílání objektů, které nejsou vzdálené, se řeší jejich serializováním a posláním ve streamu. Při přijetí takového objektu ze streamu je provedena jeho deserializace, čímž vznikne nový objekt. Změna tohoto

objektu nemá žádný vliv na původní objekt, který byl serializován. Naproti tomu vzdálené objekty se předávají odkazem, což znamená že klient obdrží příslušný stub.

Vzdálený objekt musí implementovat rozhraní, které splňuje tyto vlastnosti:

- Rozhraní vzdáleného objektu rozšiřuje rozhraní *java.rmi.Remote* v klauzuli *extends.*
- Každá metoda z jeho rozhraní deklaruje *java.rmi.RemoteException* v klauzuli *throws* [16].

4.1.3 Získávání vzdálených objektů

Aby mohl klient využívat metody vzdáleného objektu, je nutné, aby na tento objekt měl referenci, kterou musí nejdříve získat. Pro získávání a poskytování objektů se využívá RMI registry, kam server pomocí třídy *java.rmi.Naming* zanáší stub pro daný objekt a umožní klientu si tento stub z RMI registry s pomocí třídy *java.rmi.Naming* stáhnout. Referenci lze získat i v návratové hodnotě metody, nicméně minimálně první stub se musí stáhnout přes RMI registry [12].

4.2 CORBA

Common Object Request Broker Architecture (CORBA) je průmyslový standard vyvinutý společností Object Management Group (OMG) na pomoc při programování distribuovaných objektů. Je důležité si uvědomit, že CORBA je pouze specifikace. Implementace CORBA je známá jako ORB (Object Request Broker). Na trhu je k dispozici několik implementací CORBA, například VisiBroker, ORBIX a další [19].

CORBA byla navržen tak, aby byla nezávislá na hardwarové platformě, operačním systému a programovacím jazyce. Objekty CORBA proto mohou běžet na jakékoli platformě umístěné kdekoli v síti a lze je psát v jakémkoli jazyce, který má mapování na Interface Definition Language (IDL). Systémy, které používají CORBA, nemusí být ani objektově orientované. CORBA je příkladem paradigmatu distribuovaných objektů.

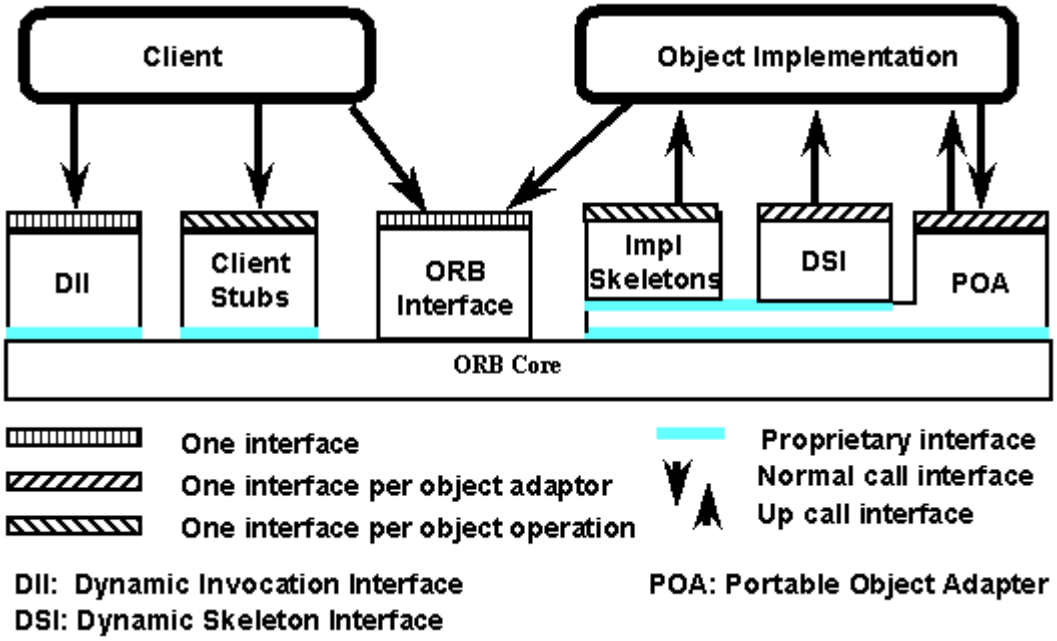
Podobně jako u Java RMI jsou objekty v CORBA specifikovány pomocí rozhraní. Rozhraní v CORBA jsou však specifikována v IDL. Je důležité si uvědomit, že IDL není skutečný programovací jazyk, ale pouze popisuje rozhraní [19].

Společnost OMG poskytuje standardizované mapování IDL do jazyků C, C++, Java, Ruby, COBOL, Smalltalk, Ada, Lisp, Python a IDLscript [20].

4.2.1 Architektura

CORBA se skládá z klientského prostředí a prostředí na straně serveru. Jednotlivé komponenty mají tyto funkce:

- **Klient:** Při využívání vzdálených objektů musí mít klient k dispozici jeho rozhraní v jazyce IDL a jednoznačnou adresu objektu (Interoperable Object Reference) [22].
- **Stuby:** Jsou vygenerovány kompilátorem jazyka IDL. Pro každou operaci každého rozhraní existuje samostatný stub. Vzhledem k tomu, že typický klient vyvolá operace pouze u několika typů objektů, existuje pouze omezený počet stubů. Klient může vyvolat operace na stovkách nebo více instancích těchto typů, ale to obnáší pouze změnu odkazů na objekty a nezvyšuje počet stubů [21].
- **Dynamic Invocation Interface (DII):** Klient může používat také objekty, ke kterým získá definici rozhraní až za běhu programu. Vyvolání operace na základě DII se liší od vyvolání na základě stubu stejným způsobem, jakým se skripty liší od programů. Jsou tedy interpretovány až za běhu a nejsou zkompilovány v přechozím kroku [21].
- **Object Request Broker (ORB):** Jádrem architektury CORBA je zprostředkovatel objektových služeb. Každý stroj, který je zapojený do architektury CORBA, musí mít spuštěný ORB, aby mohly procesy na tomto stroji interagovat s objekty CORBA ve vzdálených procesech [23]. ORB zahrnuje veškeré vnitřní mechanismy pro vyhledání požadovaného objektu, generování a přenos požadavků, parametrů a výsledků na úrovni komunikace mezi systémy [22].
- **Portable Object Adapter (POA):** Objektový adaptér propojuje implementaci objektu s ORB. Demultiplexuje přicházející požadavky, aktivuje nebo deaktivuje objekty a předává jim požadavky prostřednictvím volání metod kostry objektu [22].
- **Skeletony:** Slouží jako bázová třída odpovídající definici objektu v jazyce IDL. Stejně jako stuby jsou vygenerovány kompilátorem jazyka IDL.
- **Dynamic Skeleton Interface (DSI):** Dynamicky vytvořená kostra objektu na straně serveru. Jedná se obdobu DII na straně klienta. Typickým použitím je most pro transformaci požadavků z jednoho komunikačního protokolu do jiného [22].
- **Implementace objektu:** Kód objektu, který implementuje jeho zveřejněné rozhraní. Jeho implementace může být napsána v libovolném jazyce, který je podporován ze strany IDL.



Obrázek 3. Architektura CORBA [21]

4.3 REST

REST je zkratka pro Representational State Transfer. Jedná se o architektonický styl pro distribuované systémy a poprvé ho představil Roy Fielding v roce 2000 ve své slavné disertační práci. REST obsahuje šest hlavních zásad, které musí být splněny, abychom mohli rozhraní označit jako RESTful. Hlavní pravidla pro REST jsou následující [24]:

- **Klient - server** – Oddělení vrstvy uživatelského rozhraní od vrstvy pro ukládání dat zlepšuje přenositelnost uživatelského rozhraní napříč různými platformami a zlepšuje škálovatelnost zjednodušením serverových komponent.
- **Bezstavovost** – Každý požadavek od klienta k serveru musí obsahovat všechny informace potřebné k pochopení požadavku a nesmí využívat jakéhokoli uloženého kontextu na serveru. Stav relace je proto zcela ponechán na klientovi.
- **Cache** – Každý požadavek může být explicitně označený jako cacheable nebo non-cacheable, což značí, zda se má odpověď ukládat do cache. Pokud je odpověď na stejný požadavek již uložená v cache, je klientovi vrácena odpověď z cache, což přispívá k výraznému zrychlení komunikace mezi klientem a serverem.

- **Jednotné rozhraní** – Jednotné omezení rozhraní je zásadní pro návrh jakékoli REST služby. Zjednodušuje a odděluje architekturu, což umožňuje vývoj každé části nezávisle. Čtyři hlavní zásady tohoto rozhraní jsou:
    - **Identifikace zdrojů** – Jednotlivé zdroje jsou identifikovány v požadavcích, například pomocí identifikátorů URI ve webových službách REST [25].
    - **Manipulace se zdroji prostřednictvím zastoupení –** Klient, který má k dispozici reprezentaci zdroje včetně všech připojených metadat, má dostatek informací k modifikaci nebo odstranění stavu zdroje [25].
    - **Samopopisné zprávy** – Zpráva obsahuje dostatek informací, které popisují, jak zprávu zpracovat [25].
    - **Hypermedia –** Princip spočívá v komunikaci klientské aplikace s aplikačním serverem skrze dynamicky poskytované hypermédium. REST klient nepotřebuje žádné předchozí znalosti o tom, jak komunikovat s jakoukoliv částí aplikačního serveru kromě porozumění poskytovanému hypermédiu [26].
- **Systém více vrstev** – Umožňuje architektuře skládat se z hierarchických vrstev tím, že omezuje chování komponent tak, že každá komponenta nemůže „vidět“ za bezprostřední vrstvu, se kterou interaguje.
- **Kód na vyžádání** – REST umožňuje rozšířit funkčnost klienta stažením a spuštěním kódu ve formě appletů nebo skriptů. To zjednodušuje klientské aplikace snížením počtu funkcí, které musí implementovat.

4.3.1 Zdroje

Klíčová abstrakce informací v REST je zdroj. Zdrojem mohou být jakékoli informace, které lze pojmenovat: soubor, obrázek, dočasná služba, seznam dalších zdrojů atd. REST používá identifikátor zdroje k identifikaci konkrétního zdroje zapojeného do interakce mezi komponentami [24].

Stav zdroje v jakémkoli konkrétním časovém razítku se nazývá reprezentace zdroje. Reprezentace se skládá z dat, metadat popisujících data a odkazů, které mohou klientům pomoci při přechodu do dalšího požadovaného stavu.

Datový formát reprezentace je známý jako typ média (media type). Typ média identifikuje specifikaci, která definuje, jak má být reprezentace zpracována.

4.3.2 Metody pro přístup ke zdrojům

REST definuje čtyři základní metody, které jsou známé pod označením CRUD:

- **Create** slouží k vytvoření zdroje. Používá metodu POST. Volání probíhá na obecný endpoint, protože daný zdroje ještě nemá unikátní URI. Server by měl vracet kód 201 Created, nebo chybový kód v případě neúspěchu [27].
- **Retrieve** slouží k získání zdroje. Používá metodu GET.
- **Update** pro úpravu existujících dat. Používá metodu PUT.
- **Delete** slouží k mazání dat. Používá metodu DELETE.

4.4 Apache ActiveMQ Artemis

Apache ActiveMQ Artemis je zprostředkovatel zpráv s otevřeným zdrojovým kódem napsaným v jazyce Java spolu s úplným klientem Java Message Service (JMS). Podporuje velké množství komunikačních protokolů a nabízí široké možnosti konfigurace [30]. Umožnuje komunikaci mezi klienty a servery a také vytváření clusterů. Používá asynchronní systém zasílání zpráv.

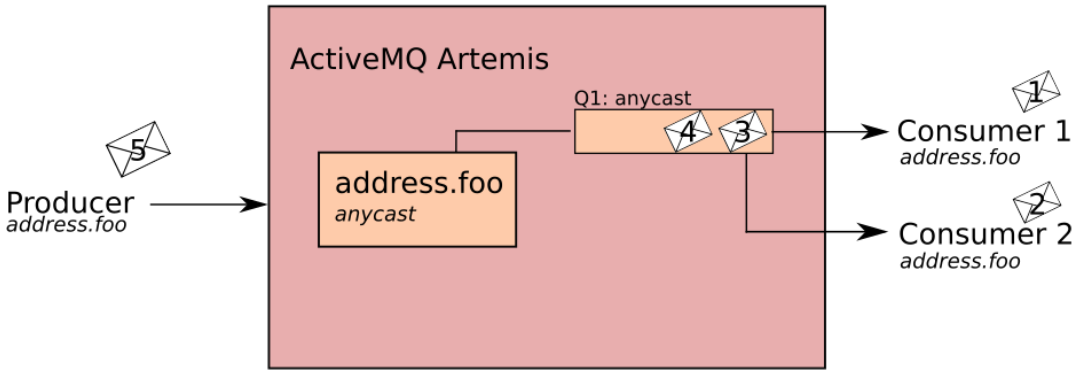
Zasílané zprávy mohou být buď perzistentní nebo neperzistentní. Perzistentní zprávy jsou uloženy v žurnálu zpráv, a tedy nedojde k jejich ztrátě při restartu nebo selhání serveru. V implicitním nastavení jsou zprávy vytvářeny jako perzistentní.

Model adresování zpráv zahrnuje tři hlavní koncepty:

- **Adresa** představuje koncový bod zasílání zpráv. V rámci konfigurace je adrese přidělen jedinečný název, žádná nebo více front a typ směrování.
- **Fronta** je spojena s adresou. Ke každé adrese může být více front. Jakmile je příchozí zpráva přiřazena k adrese, bude zpráva odeslána na jednu nebo více jejích front, v závislosti na nakonfigurovaném typu směrování. Fronty lze nakonfigurovat tak, aby se automaticky vytvářely a mazaly.
- **Typ směrování** určuje, jak se zprávy odesílají do front přidružených k adrese. Adresu Apache ActiveMQ Artemis lze konfigurovat pomocí dvou různých typů směrování.
    - **Anycast** – Jedna fronta v rámci odpovídající adresy způsobem point-to-point. Pokud však adresa používá oba typy směrování a klient nevykazuje preference ani jednoho z nich, použije se jako výchozí typ směrování anycast.

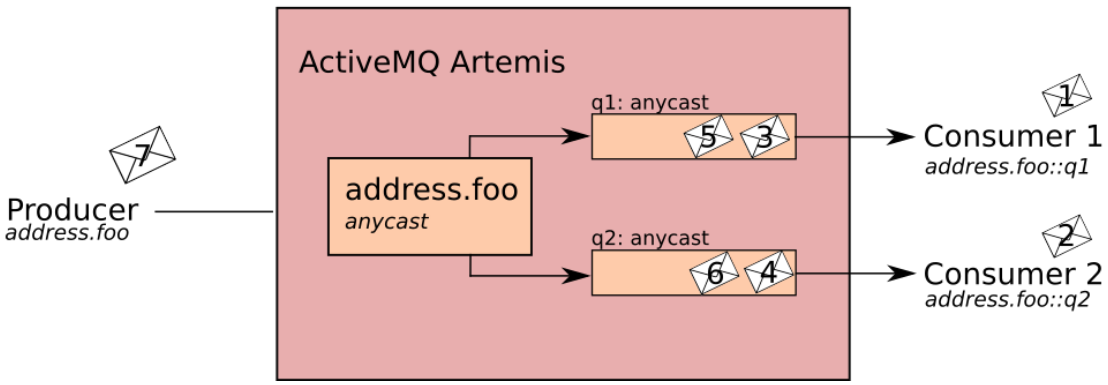
- **Multicast –** Každá fronta v rámci shodné adresy způsobem publikování a odběru (publish-subscribe).

Point-to-Point zasílání zpráv je nejčastější typ směrování zpráv, ve kterém má každá zpráva pouze jednoho příjemce. Když je zpráva přijata na adresu pomocí anycast, systém vyhledá frontu spojenou s adresou a směruje ji na ni. Ve chvíli, kdy příjemci zpráv požádají o zaslání zprávy z adresy, vyhledá zprostředkovatel příslušnou frontu a přidruží tuto frontu k příjemcům. Pokud je ke stejné frontě připojeno více příjemců, jsou zprávy rozdělovány mezi příjemce rovnoměrně [31].



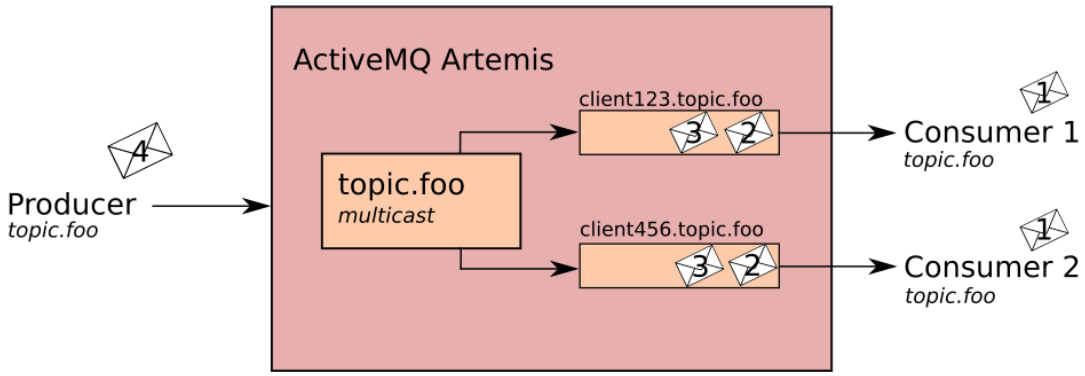
Obrázek 4. Point-to-point směrování zpráv [31]

Je také možné definovat více než jednu frontu na adresu s typem směrování anycast. Když jsou zprávy přijímány na takové adrese, jsou nejprve distribuovány rovnoměrně ve všech definovaných frontách. Pomocí názvu si mohou klienti vybrat frontu, ze které chtějí odebírat zprávy. Pokud by se k jedné frontě připojil více než jeden odběratel, budou mezi ně rovnoměrně distribuovaný zprávy stejně jako v předchozím příkladě [31].



Obrázek 5. Point-to-point s použitím dvou front [31]

Publish-Subscribe je typ směrování zpráv, který každou zprávu odesílá na všechny přihlášené odběratele.



Obrázek 6. Publish-subscribe směrování zpráv [31]

4.5 Srovnání technologií

Tato část je zaměřena na srovnání technologií vhodných pro implementaci výpočetního gridu. Kromě samostatných technologií existují i různé open source frameworky, z nichž za zmínku stojí například Java Parallel Processing Framework (JPPF) [32].

4.5.1 Java RMI

Hlavní charakteristiky:

- Java RMI je technologie specifická pro Javu.
- Pro implementaci využívá rozhraní Javy.
- Odstranění nepoužívaných objektů zajišťuje automaticky Garbage Collector.
- Programy v RMI mohou stahovat nové třídy ze vzdálených JVM.
- Umožnuje předávání primitivních typů, serializovaných objektů nebo předávání vzdálených objektů odkazem.
- Pro komunikaci využívá JRMP protokol (starší verze RMI) nebo IIOP protokol (novější verze RMI) [17].

Výhody:

- Poskytuje použitelnost na mnoha platformách.
- Může zavést nový kód do jiného JVM.

- Vývojáři Javy již mohou mít zkušenosti s RMI (je k dispozici již od JDK 1.02) [18]

Nevýhody:

- Svázáno pouze s platformami s podporou Javy.
- Bezpečností hrozby spojené se vzdáleným spuštěním kódu nebo případné omezení funkcí vynucené bezpečnostními restrikcemi.
- Může fungovat pouze se systémy podporujícím jazyk Java [18].

4.5.2 CORBA

Hlavní charakteristiky:

- Umožňuje implementaci v mnoha programovacích jazycích.
- K oddělení rozhraní od implementace používá IDL (Interface Definiton Language).
- Nezajišťuje automatické odstranění nepoužívaných objektů z paměti, kvůli podpoře různých jazyků jako je například C++.
- Nemá mechanismus pro sdílení kódu.
- Objekty se předávají odkazem.
- Pro komunikaci vyžívá jako svůj hlavní protokol IIOP [17].

Výhody:

- Služby lze psát v mnoha různých jazycích, spouštět na mnoha různých platformách a přistupovat k nim jakýmkoliv jazykem s mapováním pro rozhraní IDL.
- S IDL je rozhraní jasně oddělené od implementace a vývojáři mohou vytvářet různé implementace založené na stejném rozhraní.
- Podporuje primitivní datové typy a širokou škálu datových struktur jako parametry.
- Ideální pro použití se staršími systémy a pro zajištění toho, že aplikace napsané nyní budou i v budoucnu přístupné.
- Poskytuje snadný způsob, jak propojit objekty a systémy dohromady [18].

Nevýhody:

- Služby vyžadují použití IDL, což znamená další jazyk, který se musí vývojář před začátkem implementace naučit, a tedy i jisté zdržení vývoje.
- Implementace nebo používání služeb vyžadují mapování IDL na požadovaný jazyk, ale ne všechny jazyky musí být ze strany IDL podporovány.
- Nepodporuje přenos objektů ani kódu [18].

4.5.3 REST

Hlavní charakteristiky:

- Chování a stav aplikace je vyjadřován za pomoci takzvaného resource. Každý z těchto zdrojů musí mít přiřazený unikátní identifikátor, tedy URL nebo URN.
- Stav aplikace je určován za pomoci URL. Další stavy pak můžeme získat přímo z odkazů, které klient dostane v odpovědi poslané serverem.
- Definuje jednotný přístup pro získávání zdrojů a manipulaci s nimi v podobě čtyř operací, tzv. CRUD (Create, Read, Update, Delete).
- Zdroj může být reprezentován různě, například za pomoci XML, HTML, JSON, SVG nebo PDF. Klient pak nepracuje s tímto zdrojem přímo, ale jen s jeho reprezentací [28].

Výhody:

- Jednoduché a změnám odolné rozhraní umožňující snadnou rozšiřitelnost.
- Malé nároky na klienta z hlediska porozumění sémantice operací.
- Zdroj lze velice snadno cacheovat a transformovat [29].

Nevýhody:

- Chybějící podpora na úrovní middleware [29].

4.5.4 Výběr vhodné technologie pro implementaci

Dle dohledaných informací se Java RMI nebo CORBA v současnosti využívají již jen ve starších projektech. Je tedy otázkou, zda při vytváření nového projektu raději nezvolit modernější technologii, která se těší široké oblibě a větší podpoře ze strany komunity vývojářů. Z tohoto pohledu se mi zdá jako vhodnější volba použití REST. U Apache ActiveMQ Artemis je zase velkou výhodou, že dle nadefinovaných front řeší odesílání zpráv na klienty, přijímání zpráv z klienta na server a také přihlašování nových klientů k odběru zpráv. Další výhodou je, že zprávy jsou v implicitním nastavení perzistentní a nedojde tak ke ztrátě dat při výpadku zařízení. Z uvedených skutečností se mi zdá jako nejvýhodnější volba použití kombinace REST a Apache ActiveMQ Artemis.

II. PRAKTICKÁ ČÁST

5 VYTVOŘENÍ VÝPOČETNÍHO GRIDU

Výpočetní grid by měl sloužit k experimentálním výpočtům pro potřeby AI Lab na Fakultě aplikované informatiky Univerzity Tomáše Bati ve Zlíně. Měl by být platformě nezávislý a fungovat na operačních systémech Windows a Linux, pro implementaci proto byla vybrána jako hlavní technologie Java. Implementace je rozdělena do tří projektů vytvořených ve vývojovém prostředí Intellij IDEA.

- *grid-server* – Projekt obsahuje funkcionalitu serverové aplikace.
- *grid-node* – Projekt obsahuje funkcionalitu výpočetního nodu.
- *grid-common* – Projekt obsahuje společné části. Jsou v něm umístěny především třídy, které slouží pro zasílání zpráv mezi serverem a nodem. Oba přechozí projekty mají na tento projekt nastavenu závislost.

Pro implementaci byla dále využita technologie SpringBoot, která umožňuje snadnou konfiguraci Spring frameworku a také obsahuje zabudovaný aplikační server Apache Tomcat.

Pro zasílání zpráv byl mezi serverem využit JMS Broker Apache ActiveMQ Artemis. Tento však není přímou součástí projektů. Je potřeba ho samostatně nainstalovat a nakonfigurovat.

Systém v současnosti podporuje pouze zpracování úloh, které se skládají ze spustitelného souboru **jar** a doplňujícího konfiguračního souboru **txt**, který na jednotlivých řádcích obsahuje parametry pro jedno spuštění programu. Počet řádků souboru tedy udává celkový počet běhů aplikace. Systém je ale vytvořený tak, že v budoucnu by měl umožňovat snadné rozšíření, tak aby podporoval i jiné druhy spustitelných souborů.

Dále již bude následovat popis jednotlivých komponent systému.

5.1 Grid server

Grid server slouží jako vstupní brána do aplikace. Přijímá požadavky na zpracování od klientů, kteří k němu přistupují pomocí rozhraní REST API. Po přijetí požadavku ke zpracování na základě vstupních dat vygeneruje samostatné úlohy. Vytvořená úloha obsahuje všechny parametry potřebné pro její vykonání na výpočetním nodu. Úloha je v rámci implementace reprezentována třídou TaskRequest. Grid server vytvořený objekt třídy TaskRequest zařadí do fronty zpráv.

Na grid serveru musí být nainstalován broker ApacheMQ Artemis pro komunikaci s výpočetními nody. Tento broker je nakonfigurován tak, aby obsahoval dvě samostatné a na sobě

nezávislé fronty zpráv. První fronta *requestQueue* slouží pro zasílání zpráv obsahující požadavek na zpracování úlohy. Druhá fronta *responseQueue* slouží k přijímání zpráv z výpočetních nodů. Touto frontou se zasílají informace o zpracování úlohy zpět na server. ApacheMQ Artemis je nakonfigurován tak, aby vytvářel perzistentní zprávy, které jsou odolné vůči dočasnému výpadku serveru.

Grid server zajišťuje příjem zpráv od výpočetních nodů, které obsahují informace o tom, zda byla úloha úspěšně zpracována. Informace o výsledku zpracování úlohy zapisuje do logovacího souboru na serveru, kde je pak možné dohledat i chyby při zpracování. Neúspěšně zpracované úlohy se ale znovu nezařazují do fronty zpráv *requestQueue.* Lze předpokládat, že pokud došlo k chybě, mohlo to být způsobeno i chybným zadáním parametrů pro spuštění úlohy ze strany klienta a snahou o opakované zpracování by mohlo dojít k zacyklení komunikace mezi grid serverem a grid nodem, protože i při opětovném zpracování by znovu došlo k chybě. Chyba je tak tedy pouze zapsána do výstupního logu na serveru.

5.1.1 Adresářová struktura

Hlavní adresář aplikace je *grid-server*. Je umístěn v domovském adresáři uživatele, pod kterým je grid server spuštěn. Obsahuje tyto podadresáře a soubory:

- *executable* – Adresář obsahuje spustitelné soubory *jar*.
- *output* – Adresář obsahuje veškeré výstupní soubory, které byly vytvořeny na nodech a odeslány zpět na server.
- *application.log* – Soubor obsahuje chyby při zpracování úloh, ale také informace o úspěšně zpracovaných úlohách. Zaleží na nastavení úrovně logování. V současné konfiguraci je logování nastaveno na úroveň INFO.

5.2 Grid node

Přijímá výpočetní úlohy ke zpracování. K tomu mu slouží broker Apache ActiveMQ Artemis, který musí běžet na každém výpočetním nodu. Instance brokera musí být navíc nakonfigurovaná tak, aby ve svém konfiguračním souboru *broker.xml* obsahoval údaje pro připojení na brokera běžícího na grid serveru. Musí tedy obsahovat IP adresu serveru, název vzdáleného brokera, přihlašovací údaje (pokud broker, na kterého se připojujeme nepovoluje anonymní přihlášení) a název lokální fronty *requestQueue* i vzdálené fronty *requestQueue* mezi kterými má vzniknout propojení. Tím se grid node přihlásí k odběru zpráv z uvedené fronty na grid serveru. Stejným způsobem je také potřeba vytvořit propojení pro odchozí

frontu zpráv *responseQueue*. Po napojení vstupních a výstupních front již může grid node přijímat požadavky ke zpracování úlohy.

Příklad konfigurace v souboru broker.xml v Apache ActiveMQ Artemis:

```
<broker-connections>
   <amqp-connection uri="tcp://10.0.0.40:5672" name="my-broker" retry-in-
terval="100" reconnect-attempts="-1" user="admin" password="test">
      <receiver queue-name="requestQueue"/>
      <sender queue-name="requestQueue"/>
      <receiver queue-name="responseQueue"/>
      <sender queue-name="responseQueue"/>
   </amqp-connection>
</broker-connections>
```

Při přijetí požadavku na zpracování, který je reprezentován objektem třídy TaskRequest, vyhledá grid-node spustitelný soubor *jar* ve svém lokálním adresáři *executable*. Pokud daný soubor ještě nemá uložený, vyžádá si přes REST API jeho zaslání ze serveru. Po obdržení spustitelného soboru ho uloží do svého lokální adresáře *executable* a z parametrů v objektu TaskRequest sestaví příkaz pro jeho spuštění. Jako poslední parametr příkazu doplní cestu k výstupnímu adresáři, do kterého má spustitelný soubor vytvořit výstupní soubory. Poté spustí soubor jar a pokud jeho zpracování proběhlo úspěšně, vzniklé soubory odešle na server zavoláním REST API endpointu. Bez ohledu na to, jestli zpracování úlohy proběhlo úspěšně nebo ne, vytvoří grid node objekt třídy TaskResponse, který zařadí do výstupní fronty zpráv responseQueue. Pokud při zpracování došlo k chybě, tak je součástí této odesílané zprávy na server i informace o chybě. Tím je zpracování úlohy ukončeno a grid node čeká na přijetí dalšího požadavku ke zpracování úlohy z fronty *requestQueue*.

5.2.1 Adresářová struktura

Hlavní adresář výpočetního nodu je *grid-node*. Je umístěn v domovském adresáři uživatele, pod kterým je node spuštěn. Obsahuje tyto podadresáře a soubory:

- *executable* – Adresář obsahuje spustitelné soubory *jar*.
- *output* – Adresář obsahuje dočasné podadresáře a v nich soubory ze zpracování úloh. Při spuštění aplikace je obsah tohoto adresáře vymazán. Předpokladem je, že výstupní soubory byly již odeslány úspěšně na server.
- *application.log* – Soubor obsahuje chyby při zpracování úloh a doplňující informace určené především k odladění funkcionality aplikace. V aktuálním nastavení je úroveň logování nastavena na INFO.

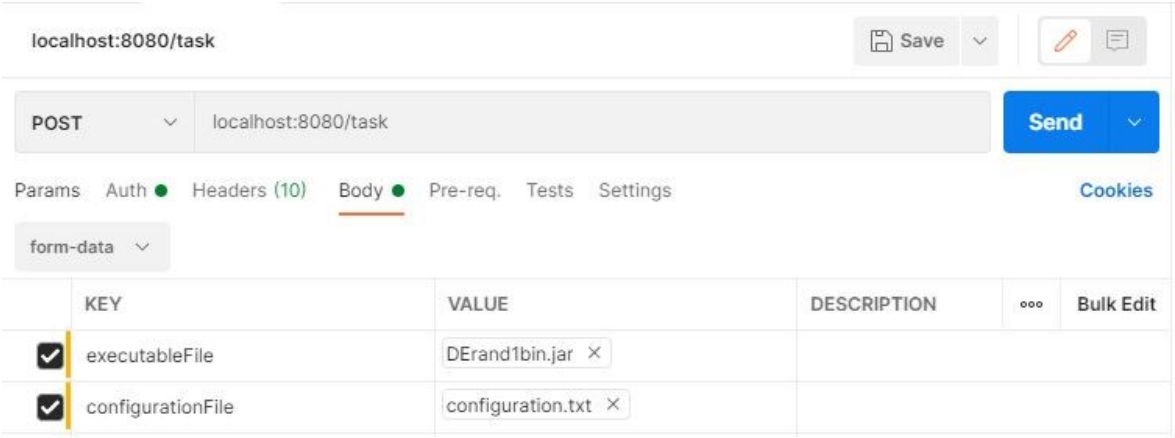
6 PŘÍPADY UŽITÍ

Grid server byl testován v prostředí AI Labu. Jako testovací algoritmus byl použit benchmarkový algoritmus DERand1Bin. Tento algoritmus přijímá tyto parametry:

- [prefix] - textový prefix vzniklého souboru, např. název algoritmu
- [f] – id volané funkce, povolený rozsah 1-10
- [dim] - je dimenze problému, povolené hodnoty 5, 10, 15 a 20
- [fes] – počet ohodnocení účelové funkce (pro dim 5 - 50 000, 10 - 1 000 000, 15 - 3 000 000 a 20 - 10 000 000)
- [`runId`] - je ID běhu, které slouží k unikátní identifikaci výsledku

Pro účely statistiky bylo provedeno 30 běhů pro každou funkci v každém dimenzionálním nastavení, tedy 30 x 10 x 4 = 1200 běhů.

Pro nahrání souborů na server a vytvoření úlohy slouží REST endpoint /task. V příkladu však není uvedena skutečná IP adresa serveru. Tu je potřeba doplnit místo *localhost*.



Obrázek 7. Ukázka zasílání úkolu na server z programu Postman

7 POPIS TŘÍD

V této kapitole budou zmíněny některé důležité třídy, ze kterých se projekty skládají a jejich hlavní funkcionalita.

7.1 GridServerController

Tato třída je součástí projektu *grid-server* a stará se o přijímání nových úkolů zaslaných na server a jejich zařazení do fronty zpráv ke zpracování. Umožňuje také výpočetním nodům stahovat spustitelné soubory ze serveru a ukládat svoje výsledky na server.

```
@RestController
public class GridServerController {
  @Autowired
  DispatcherService dispatcherService;

  @Autowired
  FileService fileService;

  @PostMapping(value = "/task")
  public ResponseEntity<String> uploadTask(
      @RequestParam("executableFile") MultipartFile executableFile,
      @RequestParam("configurationFile") MultipartFile configurationFile
  ) {
    final String executableFileName = StringUtils.cleanPath(
        Objects.requireNonNull(executableFile.getOriginalFilename()));
    fileService.saveExecutableFile(executableFile);
    final String controlHash = fileService.getFileMD5Hash(executableFileName);

    String serverIP;
    try {
      InetAddress inetAddress = InetAddress.getLocalHost();
      serverIP = inetAddress.getHostAddress();

    } catch (UnknownHostException ex) {
      return new ResponseEntity<>(
          "Local host name could not be resolved into an address.",
          HttpStatus.INTERNAL_SERVER_ERROR
      );
    }

    final List<String> lines = fileService.readLines(configurationFile);

    for (final String parameters : lines) {
      final TaskRequest taskRequest = new TaskRequest();
      taskRequest.setExecutableFileName(executableFileName);
      taskRequest.setControlHash(controlHash);
      taskRequest.setParameters(parameters);
      taskRequest.setServerIP(serverIP);
      dispatcherService.sendTaskRequest(taskRequest);
    }

    return new ResponseEntity<>("Ok", HttpStatus.OK);
  }

  @GetMapping(path = "/files/executable/{filename:.+}")
  public ResponseEntity<Resource> downloadExecutableFile(
      @PathVariable String filename
  ) throws IOException {
    final File executableFile = fileService.getExecutableFile(filename);

    if (executableFile == null) {
      return ResponseEntity.notFound().build();
    }

    final InputStreamResource resource = new InputStreamResource(
        new FileInputStream(executableFile)
```

```
    );

    final HttpHeaders headers = new HttpHeaders();
    headers.add(
        HttpHeaders.CONTENT_DISPOSITION,
        "attachment; filename="
        + executableFile.getName());
    headers.add("Cache-Control",
        "no-cache, no-store, must-revalidate");
    headers.add("Pragma", "no-cache");
    headers.add("Expires", "0");

    return ResponseEntity
        .ok()
        .headers(headers)
        .contentLength(executableFile.length())
        .contentType(MediaType.APPLICATION_OCTET_STREAM)
        .body(resource);
  }

  @PostMapping("/files/output")
  public ResponseEntity<String> uploadOutputFiles(
      @RequestParam("files") MultipartFile[] files
  ) {
    try {
      final List<String> fileNames = new ArrayList<>();

      for (final MultipartFile file : files) {
        fileService.saveOutputFile(file);
        fileNames.add(file.getOriginalFilename());
      }

      return ResponseEntity
          .ok("Uploaded the files successfully: " + fileNames);
    }
    catch (Exception e) {
      return ResponseEntity
          .status(HttpStatus.EXPECTATION_FAILED)
          .body("Fail to upload files!");
    }
  }
}
```

7.2 FileService

Tato třída je součástí projektu *grid-server*. Poskytuje veškeré služby při ukládání a načítání souborů. Projekt *grid-node* obsahuje také třídu FileService, ale s upraveným chováním u některých metod.

```
@Service
public class FileService {
  @Value("${user.home}")
  public String userHome;

  @Value("${directory.project}")
  public String projectDirectoryName;

  @Value("${directory.executable}")
  public String executableDirectoryName;

  @Value("${directory.output}")
  public String outputDirectoryName;

  /* key = file name, value = MD5 sum */
  private final Map<String, String> fileMap = new HashMap<>();

  @EventListener(ApplicationReadyEvent.class)
  public void init() {
    final String executableDirectory = getExecutableDirectory();
```

```
  final Path executablePath = Paths.get(executableDirectory);

  try {
    final List<Path> executableFiles = Files
        .list(executablePath)
        .filter(file -> !Files.isDirectory(file))
        .collect(Collectors.toList());

    for (final Path path : executableFiles) {
      final String fileName = path.toFile().getName();
      final InputStream newInputStream = Files.newInputStream(path);
      final String md5InputStream = DigestUtils.md5DigestAsHex(newInputStream);
      newInputStream.close();

      fileMap.put(fileName, md5InputStream);
    }
  }
  catch (IOException ex) {
    throw new RuntimeException("Could not read files in directory "
        + executableDirectory + ". Error: " + ex.getMessage());
  }
}

@NonNull
public String getExecutableDirectory() {
    final Path executablePath = Paths
        .get(userHome + File.separator + projectDirectoryName
            + File.separator + executableDirectoryName);
  final File file = executablePath.toFile();

  if (!file.exists()) {
    file.mkdirs();
  }

  return file.getAbsolutePath();
}

@NonNull
public String getOutputDirectory() {
    final Path outputPath = Paths
        .get(userHome + File.separator + projectDirectoryName
            + File.separator + outputDirectoryName);
  final File file = outputPath.toFile();

  if (!file.exists()) {
    file.mkdirs();
  }

  return file.getAbsolutePath();
}

@Nullable
public File getExecutableFile(final String fileName) {
  final String executableDirectory = getExecutableDirectory();
  final Path filePath = Paths.get(executableDirectory + File.separator + fileName);
  final File file = filePath.toFile();

  if (!file.exists()) {
    return null;
  }

  return file;
}

public void saveExecutableFile(final MultipartFile file) {
  try {
      final String fileName = StringUtils.cleanPath(
          Objects.requireNonNull(file.getOriginalFilename()));
    final String md5SavedFile = fileMap.get(fileName);
    final String md5InputStream = DigestUtils.md5DigestAsHex(file.getInputStream());

    if (md5SavedFile == null || !md5SavedFile.equals(md5InputStream)) {
      // This file has not been saved yet or the control hash is different.
      final String executableDirectory = getExecutableDirectory();
      final Path filePath = Paths.get(executableDirectory + File.separator + fileName);

      Files.copy(file.getInputStream(), filePath, StandardCopyOption.REPLACE_EXISTING);
```

```
        fileMap.put(fileName, md5InputStream);
      }
    }
    catch (IOException e) {
      throw new RuntimeException("Could not store the file. Error: " + e.getMessage());
    }
  }

  public void saveOutputFile(final MultipartFile file) {
    try {
      final String fileName = StringUtils.cleanPath(
          Objects.requireNonNull(file.getOriginalFilename()));
      final String outputDirectory = getOutputDirectory();
      final Path filePath = Paths.get(outputDirectory + File.separator + fileName);

      Files.createDirectories(filePath.getParent());
      Files.copy(file.getInputStream(), filePath, StandardCopyOption.REPLACE_EXISTING);

      System.out.println("The file " + fileName + " has been saved.");
    }
    catch (IOException e) {
      throw new RuntimeException("Could not store the file. Error: " + e.getMessage());
    }
  }

  @Nullable
  public String getFileMD5Hash(final String fileName) {
    return fileMap.get(fileName);
  }

  @NonNull
  public List<String> readLines(final MultipartFile file) {
    try {
      final InputStreamReader isr = new InputStreamReader(
          file.getInputStream(),
          StandardCharsets.UTF_8
      );
      final BufferedReader reader = new BufferedReader(isr);
      return reader.lines().collect(Collectors.toList());
    }
    catch (Exception e) {
      throw new RuntimeException("Could not read the file. Error: " + e.getMessage());
    }
  }
}
```

7.3 ExecutorService

Tato třída je umístěna v projektu grid-node a je v ní umístěna hlavní logika pro zpracovávání úloh.

```
@Service
public class ExecutorService {
  @Autowired
  FileService fileService;

  private final Logger log = LoggerFactory.getLogger(ExecutorService.class);

  @JmsListener(destination = "${jms.inbound-queue}")
  @SendTo("${jms.outbound-queue}")
  public TaskResponse receiveTaskRequest(final TaskRequest taskRequest) {
    final String executableFileName = taskRequest.getExecutableFileName();
    final String controlHash = taskRequest.getControlHash();
    File executableFile = fileService.getExecutableFile(
            executableFileName,
            controlHash
    );

    if (executableFile == null) {
      final String url = "http://" + taskRequest.getServerIP()
          + ":8080/files/executable/" + executableFileName;
```

```
  log.info("URL: " + url);

  final RestTemplate restTemplate = new RestTemplate();
  executableFile = restTemplate.execute(url, HttpMethod.GET,
      null, clientHttpResponse -> {
    fileService.saveExecutableFile(
            executableFileName,
            clientHttpResponse.getBody()
    );
    return fileService.getExecutableFile(executableFileName);
  });

  if (executableFile == null) {
    final String errorMessage = "Executable file "
            + executableFileName + " not found.";
    log.error(errorMessage);

    return createTaskResponse(taskRequest, false, errorMessage);
  }
}

final String outputDirectory = fileService.getNewOutputDirectory();
final String parameters = taskRequest.getParameters();
String command = null;

try {
  command = "java -jar " + executableFile.getAbsolutePath()
          + " " + parameters + " " + outputDirectory;
  final Process process = Runtime.getRuntime().exec(command);

  // Wait for the application to Finish
  process.waitFor();

  if (process.exitValue() != 0) {
    final String errorMessage = "Error while execution command: " + command
        + "/nProccess finished with exit value: " + process.exitValue();
    log.error(errorMessage);
    return createTaskResponse(taskRequest, false, errorMessage);
  }
}
catch (Exception ex) {
  final String errorMessage = "Exception " + ex.getMessage()
      + "while execution command: " + command + "/nException: ";
  log.error(errorMessage, ex);
  return createTaskResponse(taskRequest, false, errorMessage);
}

final List<File> outputFiles = fileService.getFilesInDirectory(outputDirectory);

if (outputFiles.isEmpty()) {
  final String errorMessage = "No output files created.";
  return createTaskResponse(taskRequest, false, errorMessage);
}

final MultiValueMap<String, Object> body = new LinkedMultiValueMap<>();

for (final File outputFile : outputFiles) {
  final FileSystemResource fileSystemResource = new FileSystemResource(outputFile);
  body.add("files", fileSystemResource);
}

final String url = "http://" + taskRequest.getServerIP() + ":8080/files/output";
log.info("URL: " + url);

final HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.MULTIPART_FORM_DATA);

final HttpEntity<MultiValueMap<String, Object>> requestEntity
        = new HttpEntity<>(body, headers);
final RestTemplate restTemplate = new RestTemplate();
final ResponseEntity<String> response;

try {
  response = restTemplate.postForEntity(url, requestEntity, String.class);
}
catch (Exception ex) {
  final String errorMessage = "Error connecting to server. Exception: "
```

```
                + ex.getMessage();
        return createTaskResponse(taskRequest, false, errorMessage);
      }

      if (response.getStatusCode() != HttpStatus.OK) {
        final String errorMessage = "Error uploading output files to server. Reason: "
            + response.getStatusCodeValue() + " - "
                + response.getStatusCode().getReasonPhrase();
        return createTaskResponse(taskRequest, false, errorMessage);
      }

      return createTaskResponse(taskRequest, true, null);
    }

    @NonNull
    private TaskResponse createTaskResponse(
        final TaskRequest taskRequest,
        final boolean successfullyProcessed,
        final String errorMessage
    ) {
      String nodeIP = null;

      try {
        InetAddress inetAddress = InetAddress.getLocalHost();
        nodeIP = inetAddress.getHostAddress();
      }
      catch (UnknownHostException ex) {
        log.error("Local host name could not be resolved into an address.");
      }

      final TaskResponse taskResponse = new TaskResponse();
      taskResponse.setExecutableFileName(taskRequest.getExecutableFileName());
      taskResponse.setControlHash(taskRequest.getControlHash());
      taskResponse.setParameters(taskRequest.getParameters());
      taskResponse.setNodeIP(nodeIP);
      taskResponse.setSuccessfullyProcessed(successfullyProcessed);
      taskResponse.setErrorMessage(errorMessage);
      return taskResponse;
    }
}
```

ZÁVĚR

Cílem této diplomové práce bylo vytvoření výpočetního gridu pro laboratoř umělé inteligence z důvodu urychlení zpracování komplexních výpočtů. V teoretické části je proto popsáno, jaké jsou časové složitosti úloh a dále se teoretická část důkladně zabývá porovnáním technologií, které by byly vhodné pro implementaci výpočetního gridu. Z této rešerše vychází jako nelepší volba technologie REST a zprostředkovatel zasílání zpráv Apache Active ActiveMQ Artemis, které byly následně použity pro implementaci výsledného řešení. Hlavním důvodem pro volbu Apache ActiveMQ Atemis je, že řeší fronty zpráv do kterých jsou zařazovány úlohy na výpočetní uzly a toto rozdělení úloh je přímo závislé na tom, jak uzly rychle zpracovávají své úlohy. Po dokončení implementace bylo provedeno ověření funkčnosti řešení na gridu vytvořeném na výpočetních strojích v laboratoři umělé inteligence s vyžitím poskytnutého benchmarkového algoritmu DERand1Bin a dle očekávání došlo k zrychlení zpracování zadané úlohy, které odpovídá počtu zapojených výpočetních uzlů.

Tato práce nabízí do budoucna možnosti dalšího rozšíření, a to hlavně směrem větší podpory zpracovaných úloh, které by již nemusely být nutně implementovány pouze v jazyce Java.

SEZNAM POUŽITÉ LITERATURY

[1] Microsoft Azure: Typy umělé inteligence [online]. [cit. 2021-02-21]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/what-is-artificial-intelligence/

[2] Pravidla hry Go. *Portál České asociace Go* [online]. Praha: Česka asociace GO, 2018 [cit. 2021-03-20]. Dostupné z: https://goweb.cz/pravidlahry/

[3] AlphaGo [online]. [cit. 2021-03-20]. Dostupné z: https://deepmind.com/research/case-studies/alphago-the-story-so-far

[4] MAŘÍK, Vladimír, Jiří LAŽANSKÝ a Olga ŠTĚPÁNKOVÁ. *Umělá inteligence*. Praha: Academia, 2001. ISBN 9788020004727.

[5] NP-úplnost. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2021 [cit. 2021-03-27]. Dostupné z: https://cs.wikipedia.org/wiki/NP-%C3%BAplnost

[6] Father of the Grid. In: *The University of Chicago Magazine* [online]. Chicago: The University of Chicago, 2007 [cit. 2021-03-28]. Dostupné z: http://magazine.uchicago.edu/0404/features/index.shtml

[7] Grid Computing [online]. Augsburg, Germany: Networx Security, 2021 [cit. 2021-04-02]. Dostupné z: https://www.networxsecurity.de/glossary-d1/g-d1/grid-computing/

[8] BROWN, Michael, Bart JACOB, Kentaro FUKUI a Nihar TRIVEDI. *Introduction to Grid Computing*. New York, United States: IBM, 2005. ISBN 0738494003.

[9] LUKAŠÍK, Petr. *Využití paralelních výpočtů a technologie Gridu pro rozsáhlé vědeckotechnické výpočty*. Zlín: Univerzita Tomáše Bati ve Zlíně, 2013, 197 s. Dostupné také z: http://hdl.handle.net/10563/37262. Univerzita Tomáše Bati ve Zlíně. Fakulta aplikované informatiky, Ústav informatiky a umělé inteligence.

[10] MAGOULES, Frederic, Jie PAN, Kiat-An TAN a Abhinit KUMAR. *Introduction to Grid Computing*. London (UK): CRC Press, 2009. ISBN 9780367385828.

[11] YOUSIF, Adil, Sulaiman MOHD NOR, Abdul Hanan ABDUALLA a Mohammed BAKRI BASHIR. Job Scheduling Algorithms on Grid Computing: State-of- the Art. *International Journal of Grid and Distributed Computing* [online]. 2015, **2015**(6), 125-140 [cit. 2021-04-02]. Dostupné z: doi:10.14257/ijgdc.2015.8.6.13

[12] Java remote method invocation. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-04-05]. Dostupné z: https://cs.wikipedia.org/wiki/Java_remote_method_invocation

[13] PEDAMKAR, Priya. RMI Architecture. *EDUCBA* [online]. EDUCBA, 2020 [cit. 2021-04-05]. Dostupné z: https://www.educba.com/rmi-architecture/

[14] WOLLRATH, Ann, Roger RIGGS a Jim WALDO. *A Distributed Object Model for the Java System* [online]. Toronto: USENIX, 1996 [cit. 2021-04-05]. Dostupné z: https://www.cc.gatech.edu/classes/AY2009/cs4210_fall/papers/wollrath.pdf

[15] GROSSO, William. *Java RMI*. O'Reilly, 2001. ISBN 9781565924529.

[16] An Overview of RMI Applications. *Java Documentation* [online]: Oracle, 2020 [cit. 2021-04-05]. Dostupné z: https://docs.oracle.com/javase/tutorial/rmi/overview.html

[17] Difference between RMI and CORBA. *GeeksforGeeks* [online]. 2020 [cit. 2021-04-10]. Dostupné z: https://www.geeksforgeeks.org/difference-between-rmi-and-corba/

[18] REILLY, David. Java RMI & CORBA – A comparison of two competing technologies. *Java Coffee Break* [online]. 2006 [cit. 2021-04-10]. Dostupné z: http://www.javacoffeebreak.com/articles/rmi_corba/index.html

[19] Distributed Java Programming with RMI and CORBA. *Oracle* [online]. Oracle, 2002 [cit. 2021-04-11]. Dostupné z: https://www.oracle.com/technical-resources/articles/javase/rmi-corba.html

[20] CORBA FAQ. *CORBA* [online]. Object Management Group, 2021 [cit. 2021-04-11]. Dostupné z: https://www.corba.org/faq.htm

[21] ORB Basics. *CORBA* [online]. Object Management Group [cit. 2021-04-11]. Dostupné z: https://www.corba.org/orb_basics.htm

[22] LAMPA, Petr. CORBA a IIOP. In: *Fakulta informačních technologií VUT v Brně* [online]. Brno [cit. 2021-04-12]. Dostupné z: http://www.fit.vutbr.cz/~lampa/papers/corba.html

[23] CRAWFORD, William, Jim FARLEY a David FLANGAN. *Java Enterprise in a Nutshell*. Second Edition. O'Reilly Media, 2002. ISBN 9780596001520.

[24] What is REST. *REST API Tutorial* [online]. [cit. 2021-5-15]. Dostupné z: https://restfulapi.net/

[25] Representational State Transfer. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-15]. Dostupné z: https://en.wikipedia.org/wiki/Representational_state_transfer

[26] HATEOAS. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-16]. Dostupné z: https://cs.wikipedia.org/wiki/HATEOAS

[27] REST API. *Parse Error* [online]. [cit. 2021-5-16]. Dostupné z: https://www.parseerror.cz/nodejs-tutorial/4/co-je-to-rest-api

[28] REST API. *VIP Trust* [online]. VIP Trust, 2021 [cit. 2021-5-15]. Dostupné z: https://viptrust.com/technologie/ostatni/rest-api

[29] Representational State Transfer. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-5-15]. Dostupné z: https://cs.wikipedia.org/wiki/Representational_State_Transfer

[30] TIŠNOVSKÝ, Pavel. Apache ActiveMQ. *Root* [online]. 2019 [cit. 2021-5-15]. Dostupné z: https://www.root.cz/clanky/apache-activemq-dalsi-system-implementujici-message-brokera/

[31] Apache ActiveMQ Artemis Documentation. *Apache ActiveMQ* [online]. The Apache Software Foundation [cit. 2021-5-15]. Dostupné z: https://activemq.apache.org/components/artemis/documentation/

[32] Java Parallel Processing Framework. *JPPF* [online]. [cit. 2021-4-26]. Dostupné z: https://www.jppf.org/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AI	Artificial Intelligence
CA	Celulární automat
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
FCFS	First-Come-First-Served
IDL	Interface Definiton Language
IIOP	Internet Inter-ORB Protocol
JMS	Java Message Service
JPPF	Java Parallel Processing Framework
JRMP	Java Remote Method Protocol
JVM	Java Virtual Machine
NLU	Natural Language Understanding
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
REST	REpresentational State Transfer
RMI	Remote Method Invocation
RRL	Remote Reference Layer

SEZNAM OBRÁZKŮ