Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Plzeň, 2015

Jakub Rinkes

Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Migrace nástroje pro statickou analýzu Java byte-code do cloudu

Plzeň, 2015 Jakub Rinkes

Originál zadání diplomové práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 11. 05. 2015 Jakub Rinkes

Poděkování

Touto cestou bych rád poděkoval všem, kteří mi pomáhali a podporovali mě během vytváření této práce.

Abstract

Migration of tool for static Java byte-code analysis to cloud

This thesis extends JaCC, which is a static analysis tool for Java bytecode. JaCC performs computationally intensive operations. The operations may take a long time and use all resources of one computer and it limits the size and number of tasks that the tool can handle. Solution is to scale many computers.

Cloud services are now very widespread and used in various fields of informatics. They allow clients to rent a service that can replace the software, development environment, hardware resources or complete IT infrastructure. One of these options is renting hardware for computing tasks that require a lot of computing resources.

The main goal of thesis is to verify the technical feasibility of migration JaCC into cloud. The purpose is to increase available hardware resources and performance tasks processing. We found that migration is feasible but it is benefits may be gained only for some of the bigger tasks.

Abstrakt

Migrace nástroje pro statickou analýzu Java byte-code do cloudu

Diplomová práce rozšiřuje nástroj JaCC, který provádí statickou analýzu Java bytecodu. Nástroj provádí velké výpočetně náročné operace. Ty mohou trvat dlouhou dobu a využít všechny zdroje jednoho počítače a to limituje velikost a počet úloh, kterou může nástroj zpracovávat. Jednou z možností je využít škálování hardwaru více počítačů pro zpracování úloh.

Cloudové služby jsou v dnešní době hodně rozšířené a používané v různých oblastech informatiky. Poskytují možnost pronajmout si službu, která může nahradit software, vývojové prostředí, hardware nebo celou IT infrastrukturu. Jednou z těchto možností je pronájem hardwaru pro výpočetní úlohy, které vyžadují více výpočetních zdrojů.

Cílem této práce je ověřit technickou proveditelnost migrace nástroje JaCC do cloudu. Účelem je zvýšení dostupných hardwarových zdrojů a výkonu pro zpracování úloh nástroje. Migrace je proveditelná, ale záleží na tom, jak bude nástroj využíván, a na velikosti úkolů pro nástroj.

Obsah

1 Úvod

Cloudové služby jsou v dnešní době hodně rozšířené a používané v různých oblastech informatiky. Poskytují možnost pronajmout si službu, která může nahradit software, vývojové prostředí i celý hardware či IT infrastrukturu.

Jednou z těchto možností je pronájem hardwaru pro velké výpočetní operace, které využívají najednou hardware více počítačů a během výpočtu dochází ke komunikaci mezi počítači pomocí sítě. Před vznikem cloudovových služeb byly k takovým výpočtům používány výpočetní gridy[1] nebo clustery[2], ale problém byl v jejich dostupnosti. V době cloudových služeb není problém pronajmout si takovou infrastrukturu, provést výpočet a zase infrastrukturu vrátit a zaplatit jen za dobu používání.

Tato diplomová práce je zaměřena na JaCC, nástroj pro statickou analýzu a ověření Java bytecodu, který pro zpracování zadaných úloh spotřebuje velké množství hardwarových zdrojů v podobě velkého množství operační paměti a výkonu procesorů.

Cloudové služby se jeví jako vhodné řešení hardwarových problémů nástroje a také zdrojem urychlení zpracování úloh. Cílem této práce je prozkoumat možnosti cloudových služeb a zjistit, zda je možné migrovat nástroj do těchto služeb. Analýza migrace se bude zabývat výhodami a nevýhodami jednotlivých způsobů migrace. Následná realizovaná migrace bude podrobena testování, sledovány budou výkonnostní charakteristiky a její přínos v dalším rozvoji možností nástroje.

[1] Grid – skupina různě výkonných počítačů propojených většinou internetovou sítí.
[2] Cluster – skupina většinou stejně výkonných počítačů propojených lokální sítí.

2 Statická analýza Java byte-code

Většina dnešních Enterprise[3] aplikací obsahuje kromě vlastního vytvořeného kódu také velké množství použitých knihoven třetích stran, aby se vývoj zjednodušil a hlavně urychlil.

Java je staticky a silně typovaný jazyk. Ve zdrojovém kódu je nutné uvádět datové typy a kontrolu kompatibility typů provádí překladač během kompilace. Java také umožňuje integrovat do programu předkompilované knihovny, ale kontrolu kompatibility není vždy možné ověřit překladem nebo testy integrace. Nekompatibilita v podobě chybějící třídy, metody nebo špatných datových typů se projeví až za běhu programu. Obvykle dojde k některé z chyb ze skupiny označované jako linkage error, v Javě to jsou potomci výjimky `java.lang.LinkageError`. Nejčastějším způsobem vzniku tohoto problému je přidání nekompatibilní verze předkompilované knihovny během procesu sestavení aplikace nebo při instalaci aplikace, kdy se instalují použité externí knihovny. [1]

Nástroje pro statickou analýzu by měly tuto problematiku hlídat a pomocí ověření sestavené aplikace zjistit, zda nedošlo ke špatnému sestavení. Dalším místem pro statickou analýzu jsou systémy, které umožnují přidávání modulů v podobě předkompilovaných programů. Systém nemá možnost rozpoznat, že modul nepoužívá správné knihovny, dokud se neobjeví chyba během chodu aplikace.

2.1 JaCC - Java class compatibility checker

JaCC je nástroj pro statickou analýzu Java bytecode, který vznikl na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni.

V současnosti je JaCC tvořen menšími moduly, které provádí různé činnosti spojené se statickou analýzou a ověřením. Jádro nástroje umožnuje načíst bytecode ze souboru .jar nebo přímo z třídy .class a obsahuje komparátor pro porovnání kompatibility. Ostatní menší nástroje, které JaCC obsahuje, pak využívají jádro pro statickou analýzu a ověřují různé problémy s kompatibilitou v sestavené aplikaci.

[3] Enterprise – rozsáhlé podnikové aplikace.

Základní moduly nástroje

- **Loader** – část nástroje, která načítá bytecode do vnitřní objektové podoby nástroje. Nástroj obsahuje několik různých implementací, které umožňují načítat bytecode z různých zdrojů (soubory .jar, .class) a obsahují různé dodatečně vlastnosti pro zpracování.
- **Komparátor** – část nástroje, která provádí statickou analýzu a ověření na načtených datech.

Proces statická analýzy

Statická analýza se provádí na zkompilovaném bytecodu. Bytecode se pomocí nástroje načte a získají se z něho informace pro sestavení původního syntaktického stromu zdrojového kódu před kompilaci. Tímto procesem získáme informace o podobě tříd a jejich metod, počtu vstupních parametrů a návratových hodnotách.

Z načteného bytecodu aplikace získáme informace o použitých třídách a metodách závislých knihoven, a jakým způsobem jsou použity v aplikaci. Načtením bytecodu závislých knihoven získáme informace o jejich třídách a metodách. Získaná data se porovnávají a ověřuje se, zda použité třídy a metody jsou shodné s třídami a metodami, které byly přiloženy k aplikaci při sestavení.

2.2 Hardwarové požadavky nástroje

Statická analýza nástrojem JaCC se skládá z procesu načtení bytecode a jeho porovnání. Proces načtení bytecode je v tomto případě nejnáročnější operace na procesorové zdroje. Pokud má nástroj k dispozici více procesorů, začne Java Virtual Machine (dále jen JVM) provádět některé části kódů paralelně a dojde k úplnému vytížení všech procesorů. Částečně se tím urychlí načtení dat nástrojem, ale systém se tím stává neresponzivní na uživatelskou interakci, dokud není úloha zpracována.

Pro proces porovnání načtených dat udržuje nástroj všechny data načtená v paměti. Spotřeba operační paměti je v řádech gigabytů, protože je nutné udržovat v paměti kompletní načtený bytecode v podobě, která umožňuje provést statické porovnání.

Velikost úlohy je přímo úměrná spotřebě hardwarových zdrojů a dostupný hardware takto omezuje použití nástroje jen na úlohy do určité velikosti.

3 Cloud computing

Cloud computing (zkráceně cloud) je oblast aplikací nebo služeb, které jsou dostupné pomocí internetových protokolů nebo sítových standardů. Tyto aplikace a služby jsou dostupné na internetu a využívají virtualizovaných zdrojů. Zdroje jsou obvykle tvořeny servery, které se propojí do distribuované sítě a poskytují se jako služba neboli cloud. Uživatelé k těmto službám obvykle přistupují pomocí webového prohlížeče nebo klientské aplikace. Uživatelé neplatí za celý software, ale jen za tu část, kterou používají. [2] [3] [4]

Základními koncepty cloudu je abstrakce a virtualizace. Cloud je abstrakcí zdrojů, uživatelé nevědí o tom, které zdroje přesně využívají nebo kde přesně jsou data uložena. Virtualizace zdrojů zajišťuje rozdělení fyzických zdrojů na zdroje, které se poskytuje uživatelům. Díky tomu dochází ke sdílení zdrojů a jejich efektivní využívání.

3.1 Distribuční modely

Cloudové služby se dělí do více distribučních modelů podle služeb, které poskytují uživatelům. Služby jsou poskytovány na základě tří základních modelů, které obsahují v názvu příponu „as a service“[4]. Tyto modely jsou infrastruktura (IaaS), platforma (PaaS) a software (SaaS) a jejich vzájemný vztah zobrazuje *obrázek 1*. [4] [3] [5]

- **IaaS – Infrastructure as a service[5]**

IaaS poskytuje uživatelům přístup k virtualizovanému hardwaru. Někdy je tento model také označován jako HaaS – Hardware as a Service (překlad „Hardware jako služba“). V tomto modelu poskytuje poskytovatel svůj vlastní hardware a uživatel si pronajímá místo pro uložení dat, procesorové jednotky pro výpočty, paměťové bloky a síťová zařízení. Uživatelé si pak můžou na této infrastruktuře vyvinout vlastní aplikaci nebo službu.

Mezi největší poskytovatele této služby je Amazon se svojí službou Amazon Web Service. Tato služba bude podrobně představena v kapitole 3.4. Další významný poskytovatel je i Microsoft se službou Microsoft Azure.

[4] as a service – překlad „jako služba“
[5] Infrastructure as a service – překlad „Infrastruktura jako služba“

Výhodou tohoto modelu je, že uživatel ušetří na provozu hardwaru. Pokud by uživatel používal vlastní hardware, bude platit i za provoz a inovaci. Takto platí pouze za používání virtualizovaného hardwaru, ale poplatky se můžou dostat až na úroveň, kdy bude levnější pořídit si vlastní hardware.

- **PaaS – Platform as a service[6]**

PaaS model je rozšířený IaaS model o programové vybavení, které umožňuje vývoj dalších aplikací a služeb. Vývojářům odpadá starost s instalací programového vybavení a nástrojů, které používají pro svůj vývoj. Nejčastěji jsou dostupné nástroje pro návrh, vývoj, testování a nasazení služeb a aplikací. Další služby můžou být předinstalované databáze, webové servery a služby a prostředky pro zabezpečení.

Mezi nejznámější poskytovatele těchto služeb je Google s jeho Google App Engine službou. Tato služba umožňuje vývoj webových aplikací a jejich nasazení. Výhodou je absence finančních nákladů spojených s provozem vlastní infrastruktury pro provoz webové aplikace a pro její vývoj.

- **SaaS – Software as a service[7]**

SaaS je model obsahující hardware a software jako službu pro uživatele. Software je hostovaná aplikace, která slouží uživatelům za nějakým účelem. Software není potřeba instalovat do klientského zařízení uživatele a je uživateli dostupný dle jeho potřeb. Uživatel se nemusí starat o správu aplikace a platí pouze za používání aplikace. Odpadají tak investice do pořízení softwaru, jeho provozu a údržbě. Správu aplikace zajišťuje poskytovatel služby, ale také mu to dává právo měnit aplikaci dle svého uvážení. Uživatel ztrácí kontrolu nad aplikací.

Mezi tyto služby dnes patří hojně používané aplikace od velkých společností jako je Google, Apple, IBM a Oracle. Nejrozšířenější služby jsou Google Apps od společnosti Google a iCloud od společnosti Apple. Služba iCloud umožňuje sdílet data mezi zařízeními od společnosti Apple a tyto zařízení jsou tudíž klientem pro tuto službu. Společnost Google poskytuje většinu svých služeb pro normální uživatele zdarma, pro firmy jsou některé služby zpoplatněny. Mezi nejznámější patří Gmail, Google Calendar, Google Doc a Google Groups.

[6] Platform as a service – překlad „platforma jako služba“
[7] Software as a service – překlad „program jako služba“

Obrázek 1 : **Distribuční modely cloudových služeb**

3.2 Cloud computing charakteristiky

Klíčové charakteristiky pro cloud computing. [5] [3] [4]

- **Na požádání** (On-demand service) – uživatelé mohou přistoupit ke službám nebo zdrojům bez nutné interakce s poskytovatelem těchto služeb.
- **Síťový přístup** (Broad network access) – Služby a zdroje jsou dostupné po celé síti a jsou přístupné přes klientské aplikace nebo zařízení.
- **Sdružování zdrojů** (Resource pooling) – Výpočetní nebo datové zdroje poskytovatele se sdružují a slouží potřebám uživatelů dle jejich požadavků. Zdroje jsou uživatelům přidělovány a odebírány podle jejich potřeb.
- **Pružnost** (Rapid elasticity) – Množství zdrojů může uživatel velice rychle a automatizovaně navyšovat snižovat nebo úplně změnit. Pro uživatele se zdroje často jeví jako neomezené.
- **Monitorovaná služba** (Measured service) – Většina služeb je poskytována s nástroji pro monitorování využití zdrojů a služeb. Systém díky nástrojům může poskytovat informace o využití zdrojů poskytovateli a uživateli. Některé systémy umí díky těmto informacím i optimalizovat používané cloudové služby.

3.3 Model nasazení

Modely nasazení cloud technologie. [5] [3] [4]

- **Soukromý cloud** – cloudové služby jsou provozovány pouze pro jednu organizaci. Provoz, údržbu a rozvoj služeb zajišťuje organizace sama nebo službu zajišťuje pro organizaci třetí strana.
- **Komunitní cloud** – cloudové služby jsou provozovány pro skupinu organizací nebo uživatelů, které spojují společné zájmy nebo problémy. Cloud může vlastnit a provozovat jedna nebo více organizací najednou nebo službu zajišťuje pro komunitu třetí strana.
- **Veřejný cloud** – cloudové služby jsou volně dostupné pro celou veřejnost a provoz těchto služeb může být zajištěn obchodní, akademickou nebo vládní organizací.
- **Hybridní cloud** – tento model je složen ze dvou nebo více odlišných modelů nasazení (veřejných, soukromých a komunitních) a využívá různých výhod těchto modelů. Obvykle organizace poskytuje některé cloudové služby a jiné jsou organizaci poskytnuty externě, například z veřejného cloudu.

3.4 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (dále jen EC2) je cloudová služba od společnosti Amazon. EC2 je součástí dalších webových a cloudových služeb, které společnost Amazon provozuje pod názvem Amazon Web Services.

Uživatelé si přes EC2 pronajímají virtualizovaný hardware pro své potřeby v podobě virtuálních serveru, na kterém si spustí Amazon Machine Image (AMI). AMI je uložený obraz spustitelného systému, který uživatel spustí na virtuálním serveru a ten je pak označován jako instance. Uživatel může libovolně spouštět a ukončovat instance serverů dle svých požadavků a platby jsou pouze za ten čas, který opravdu danou službu používal. [6] [2]

3.4.1 Základní EC2 služby

- **Elastic compute units** – tato služba je abstrakcí k fyzickému hardwaru v podobě serverů. Servery obsahují operační pamětí a výpočetní procesory (AMD Opteron, Intel Xeon).

- **Elastic block storage** – služba poskytuje uložiště blokových dat, které je možné provázat s pronajatými EC2 instancemi. Obvykle se používá pro souborové systémy a služba poskytuje také služby pro zálohování a replikaci dat.
- **Elastic load balancing** – služba umožňuje směrovat příchozí komunikaci mezi více instancí a tím rovnoměrně rozdělovat zátěž instancí.

3.4.2 Možnosti placení za služby

- **On-demand** – používání instance se platí za každou celou hodinu bez dalších závazků.
- **Reserved** – uživatel zaplatí jednorázově za pronájem instance a pak jsou platby za hodinu velice nízké oproti cenám za instance On-demand. Obvykle se vyplatí při dlouhodobém používání služeb.
- **Spot** – unikátní typ instance. Cena za hodinu běhu těchto instancí se s časem mění. Uživatel zadá cenu, kterou je ochoten zaplatit a jakmile cena instance vyhovuje je automaticky spuštěna a pokračuje ve své práci. Běžně se tento typ instancí využívá pro dávkové výpočty a zpracování, kde není potřeba výsledek hned.

3.4.3 Typy instancí

- **T2** – typ instance, která poskytuje nárůst výkonu procesoru nad hranice základní úrovně. Za využívání výkonu této instance se platí CPU kredity[8]. Kredity jsou získávány za hodiny, kdy instance nepracuje, a jsou spotřebovány v době, kdy instance pracuje. Tento typ instance se hodí pro situace, kdy je krátkodobě potřeba výkon (testovací server, server pro sestavení, repositáře, vývojové nástroje).
- **M3** – obecná instance poskytující rovnováhu mezi počty výpočetních jednotek, velikostí operační paměti a sítových prostředků pro univerzální použití.
- **C3/C4** – instance, zaměřené na vysoký výkon procesorů pro výpočetní úlohy, které potřebují rychlé a výkonné procesory. Uživatel si může pronajmout server, který obsahuje 2 až 60 procesorů s podporou pro vytvoření vlastního počítačového clusteru.

[8] CPU kredity – oficiální název je CPU credits.

- **R3** – instance, zaměřené na operační paměť pro operace náročné na dostatek operační paměti. K dispozici jsou servery, které mají od 15 do 240 GB operační paměti.
- **G2** – instance s vysoce výkonnými grafickými kartami pro grafiky nebo pro výpočty prováděnými na grafických jádrech.
- **I2/D2** – instance pro ukládání dat. Instance I2 využívá rychlých SSD disků pro ukládání dat, kde se předpokládá vysoké množství vstupně-výstupních operací. D2 instance je navržena pro ukládání obrovských množství dat a každá instance může mít kapacitu až 48TB dat.

3.4.4 Služba zdarma

Společnost Amazon umožňuje novým zákazníkům využívat, některé služby zdarma. Služby je možné využívat 12 měsíců od registrace v podobě:

- T2 základní instance (1x CPU, 1 GB operační paměť, pevný disk přes službu Elastic block storage) se 750 hodinami CPU kreditů / měsíc,
- Elastic load balancing 15 GB dat se 750 hodinami / měsíc,
- uložiště dat ve velikosti do 30 GB přes službu Elastic block storage,
- a další menší služby.

Služby, které Amazon poskytuje na rok, dovolují uživateli využít jednu instanci T2 na celý jeden rok bezplatně spolu s dalšími službami. Nebo je tu možnost používat 10 instancí najednou po dobu 75 hodin každý měsíc. [6]

4 Migrace nástroje JaCC do cloudu

Nástroj JaCC je při zpracování úlohy omezen dostupným hardwarem. Zpracování velkých korpusů dat tak trvá dlouho nebo je nemožné tak velký korpus zpracovat, protože na to nejsou dostupné hardwarové zdroje na jednom počítači. Další možností je nechat nástroj využít hardware více počítačů během výpočtu jedné úlohy. Tato možnost vede na využití vlastností distribuovaných systémů a cloudových služeb.

Předpokladem migrace nástroje JaCC do cloudu je:

- umožnit nástroji využívat více hardwarových zdrojů pro zpracování úloh,
- neomezovat uživatelskou interakci s nástrojem díky delegování časově náročných operací na jiné stroje,
- umožnit zadávat nástroji úlohy opakovaně a zpracovávat více úloh najednou
- a poskytnout nástroj jako službu.

Služba Amazon EC2 byla zvolena referenční prostředí pro migraci, ale jedním z dalších předpokladů je nezávislost na této službě. Nástroj by mělo být možné nasadit i v jiné cloud službě, která poskytuje služby s distribučním modelem IaaS, nebo nástroj nasadit na skupinu počítačů propojených počítačovou sítí.

4.1 Možnosti migrace

Možnosti umístění existující aplikace do cloudu jsou dvě.

1. Základní migrací je umístit celou aplikaci do cloudové služby bez větších změn architektury.
2. Distribuovanou migrací je rozdělení aplikace na části, které budou umístěné do cloudu odděleně a budou spolupracovat. Rozdělení aplikace obvykle určuje architektura aplikace.

4.2 Základní migrace

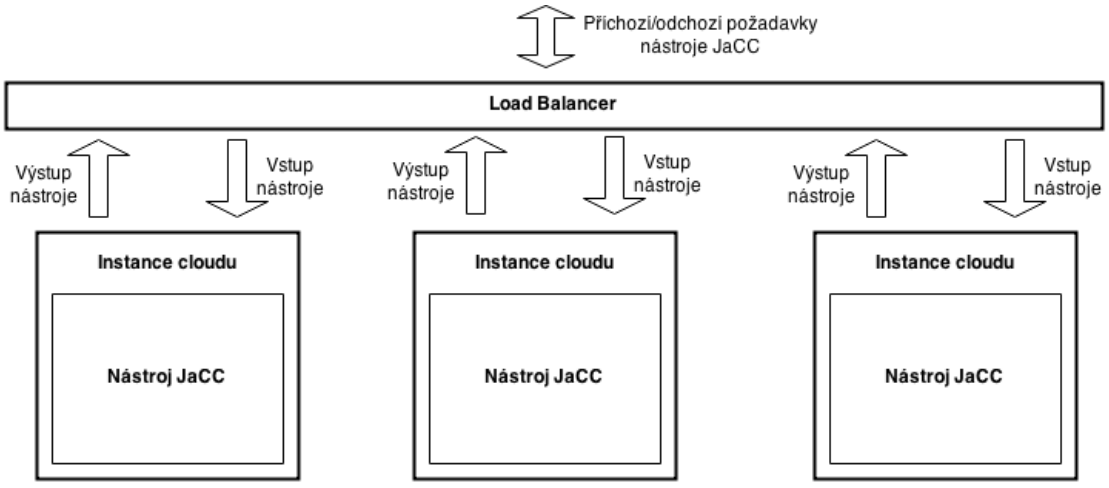
Základní podoba migrace nástroje vychází z běžného použití cloudové služby pro určitou aplikaci nebo webovou stránku. Aplikace se umístí celá na jednu instanci služby. Pokud daná instance dosáhne určitých limitů, provede se spuštění další instance s aplikací a příchozí požadavky se rozloží mezi obě běžící instance, aby se rozložila

zátěž a nedocházelo k zbytečnému přetěžování a zhoršování kvality služeb. Limity můžou být využití procesorů, operační paměti nebo počet aktivních spojení na aplikaci.

4.2.1 Architektura

Při základní podobě migrace nástroje JaCC do cloudu bude celý nástroj umístěn do cloudu jako jednotná aplikace. Vstupní body aplikace pro zadávání jednotlivých úloh pro zpracování se vystaví jako služba. Rozhraní pro vystavení aplikace v podobě služby může být vytvořeno jako webová služba nebo rozhraní pro vzdálené volání metod. Nástroj bude dostupný přes vystavěné rozhraní a bude možné mu zadávat jednotlivé úlohy pro zpracování. Nástroj přijme vstupní data a provede statickou analýzu. Po zpracování úlohy vrátí výsledek statické analýzy. Výsledek úlohy bude uložen ve strukturované podobě, aby bylo možné ho dále zpracovat dle požadavku uživatele.

Příchozí požadavky budou procházet službou pro vyvážení zatížení označované jako Load Balancer [9]. Tato služba bude zajišťovat rovnoměrné rozložení příchozích požadavků mezi všechny běžící instance s nástrojem JaCC. Každá úloha bude mít k dispozici svoji instanci, aby měla dostatečný počet hardwarových zdrojů pro výpočet, nebo bude možné provést více menších úloh na jedné instanci. To bude záležet na velikosti úlohy a hardwaru instance. Model umístění nástroje do cloudu zobrazuje *obrázek 2*.



Obrázek 2 : **Model základní migrace nástroje JaCC do cloudu**

[9] Amazon poskytuje službu Load balancer pod názvem Elastic load balancing.

4.2.2 Zhodnocení základní migrace

Výhody

- Mezi největší výhodu patří implementační nenáročnost. Nástroj bude umístěn na instanci v současném stavu, implementace rozhraní pro používání nástroje nebude implementačně náročná a v případě rozšíření možností nástroje bude nutné provádět úpravy pouze na přístupovém rozhraní.
- Služba pro rozložení příchozích požadavků vyřeší problém s přetěžování instancí.
- Výpočetně náročné operace budou delegovány od uživatele na instanci. Uživatel bude pouze čekat, než mu nástroj vrátí výsledek úlohy.

Nevýhody

- Migrace nevyřeší omezení velikosti úlohy. Instance má většinou dané hardwarové zdroje a ty stále budou omezovat nástroj ve smyslu velikosti úlohy, protože nástroj nebude moci využívat další hardware mimo instanci. Tento problém lze částečně vyřešit správnou volbou typu instance cloudové služby.
- Pro potřeby nástroje tak bude nutné pronajmout instance poskytující velké množství operační paměti a zároveň dostatečný výkon procesorů, které by mohly být finančně dražší na provoz.
- Další nevýhodou může být plýtvání zdroji, pokud nebudou instance efektivně využity.

4.3 Distribuovaná migrace

Distribuovaná podoba migrace využije principů distribuovaného výpočtu. Aplikace rozloží svůj výpočet na více uzlů a výsledek výpočtu pak centrální uzel složí. Tento model výpočtu má více označení, ale nejběžnější je Farmer-Worker nebo Master-Worker.

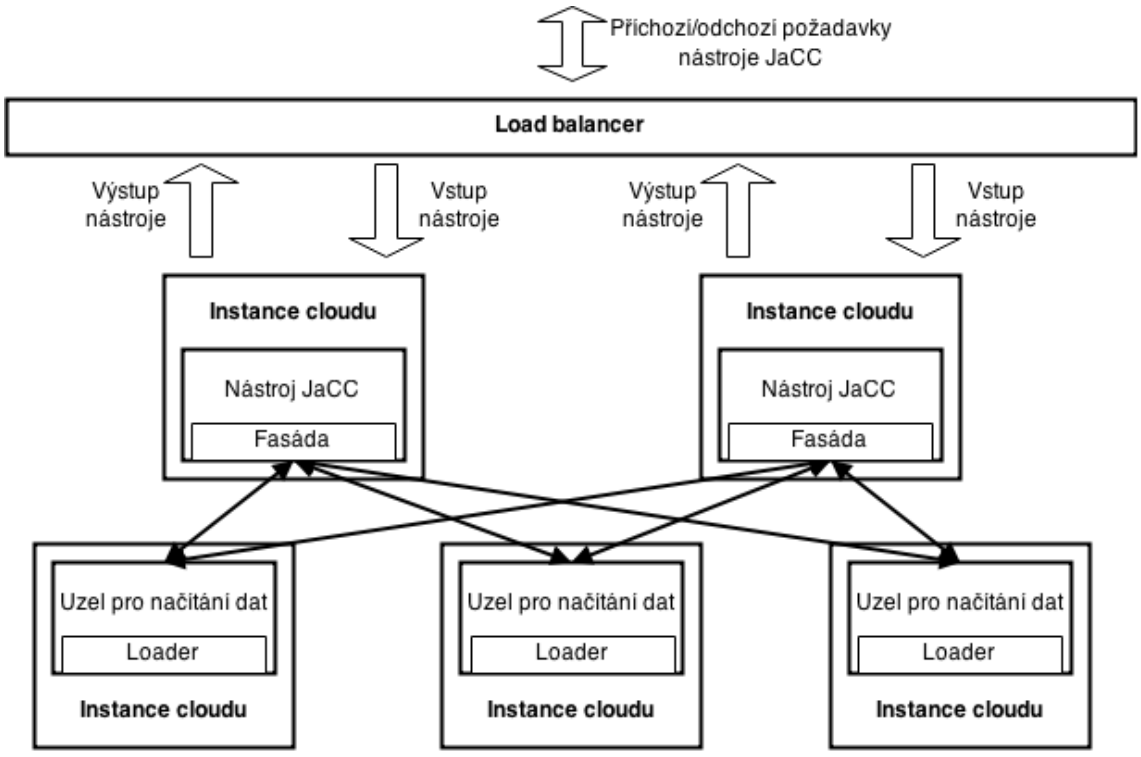
Při použití tohoto modelu v cloudu bude jedna nebo více instancí fungovat jako centrální uzly a ostatní instance budou fungovat jako výpočetní uzly. Centrální uzel bude využívat výpočetní uzly pro provedení daného výpočtu aplikace. Pokud je nutné zvýšit dostupné hardwarové zdroje pro výpočet, spustí se pouze další výpočetní uzel. V případě, že dojde k přetížení centrálního uzlu, dojde ke spuštění dalšího centrálního uzlu. [7]

4.3.1 Architektura

Nástroj JaCC je možné rozdělit na dvě části podle postupu, jak provádí statickou analýzu. Toto logické rozdělení bude místem rozdělení nástroje do distribuované podoby s využitím modelu Farmer-Worker. Jedna část aplikace bude obsahovat loader pro načítání dat. Tu budeme označovat dále jako uzel. Druhá část aplikace bude tvořit původní nástroj JaCC a fasáda pro komunikaci s uzly.

Uzel bude provádět načítání jednotlivých vstupních dat. Pro načtení dat nejsou nutné velké paměťové nároky a bude možné využít dostupné zdroje instance pro paralelizaci načítání vstupních dat. Data pro zpracování mu bude předávat fasáda, která pak obdrží načtená data zpět pro další část procesu statické analýzy. Nástroj JaCC bude provádět statickou analýzu nad daty, které mu připraví uzly a fasáda je bude poskytovat už načtené.

Distribuovaná verze nástroje bude umístěna na instance cloudu (viz *obrázek 3*). Pro přístup k nástroji bude sloužit webová služba nebo rozhraní pro vzdálené volání metod. Rovnoměrné rozložený příchozích požadavků bude zajišťovat příslušná služba, která bude úlohy rozdělovat mezi běžící centrální uzly, které budou mít možnost běžet ve více instancích.



Obrázek 3 : **Model distribuované migrace nástroje JaCC do cloudu**

4.3.2 Zhodnocení distribuované migrace

Výhody

- Distribuovaná migrace má lepší škálovatelnost hardwaru pro provádění statické analýzy nástrojem JaCC. Uzly starající se o načítání vstupních dat můžou zpracovávat vstupní data pro různé centrální uzly, které zatím budou čekat, než dostanou svá data zpět pro porovnání.
- Uzel pro načítání dat nemusí mít velké množství operační paměti, ale bude potřebovat více výkonné procesory a další výhodou uzlu může být paralelní zpracování vstupních dat, které by mohlo urychlit proces načítání vstupních dat.
- Centrální uzel bude pouze provádět vyhodnocení dat a nebude nutné, aby měl dostupné vysoce výkonné procesory, bude potřebovat mít velké množství operační paměti, aby data mohl zpracovat a vrátit výsledek.
- Počty obou typů uzlů půjde navyšovat dle aktuálního vytížení a o rovnoměrné rozložení příchozích požadavků se postará služba cloudu. O rovnoměrné rozložení jednotlivých požadavků na zpracování vstupních dat pro worker uzly se už postarají samy centrální uzly.
- Velká úloha bude mít dostatek prostoru pro zpracování a malá úloha se provede spolu s většími a nebude muset obsadit celou instanci pro sebe.
- Výpočetně náročné operace delegovány od uživatele na instance cloudu.
- Distribuovaná podoba nástroje bude efektivnější ve využití hardwarových zdrojů v případě zpracování více nezávislých úloh najednou.

Nevýhody

- Mezi zásadní nevýhody této migrace je náročná implementace a obtížné ladění. Je nutné implementovat distribuovanou podobu jádra nástroje, kterou bude možné nasadit na instance cloudu. Vyřešit komunikaci mezi jednotlivými uzly, ošetřit vzájemné vyloučení při více vláknovém zpracování a implementovat systém zotavení po chybě v systému.
- Další nevýhodou bude přidaná hodnota komunikace mezi uzly. Komunikace mezi uzly může způsobovat zpomalení celého procesu načtení dat v případě, že nebude úloha dostatečně velká. V tomto případě se neprojeví ani paralelní zpracování vstupních dat a úloha se může paradoxně zpomalit.

5 Realizace migrace

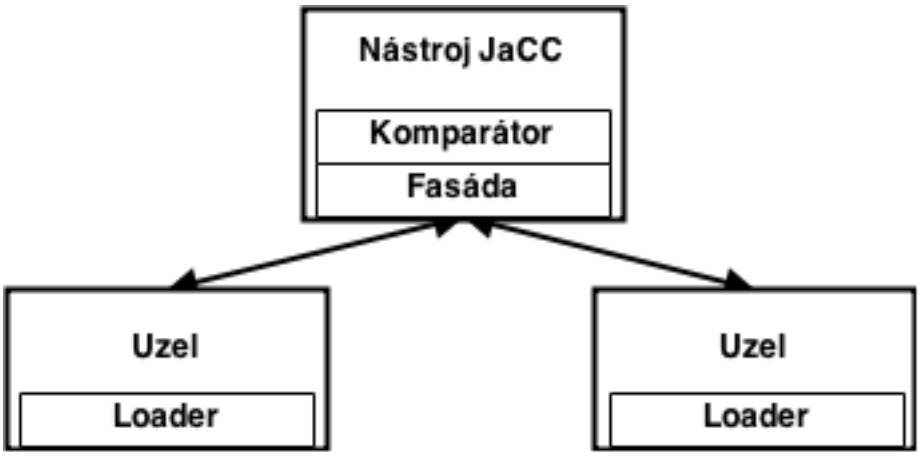
Základní migrace je implementačně nenáročná a její výsledky jsou srovnatelné s nástrojem, který se spustí na lokálním počítači. Jediné co by bylo nutné realizovat je rozhraní pro ovládání nástroje z cloudové služby a provést správnou konfiguraci cloudové služby.

Cílem praktické části práce je ověřit technickou realizovatelnost distribuované podoby nástroje a ověřit výhody a nevýhody tohoto řešení. Ověření výhod a nevýhod u distribuované migrace totiž není možné teoreticky stanovit a můžeme je pouze teoreticky odhadnout.

5.1 Architektura

Architektura distribuované podoby nástroje JaCC vychází z modelu v kapitole 4.3. Původní implementace nástroje je rozdělena na dvě části (viz *obrázek 4*):

- Uzel – část, která se v modelu Farmer-Worker označuje jako worker a bude provádět načítání vstupních dat.
- Fasáda – část, která se v modelu Farmer-Worker označuje jako Farmer, bude využívat dostupné uzly k načítání vstupních dat, které bude poskytovat nástroji pro zpracování.



Obrázek 4 : **Realizované rozdělení nástroje pro distribuovanou architekturu**

Jednotlivé uzly a fasády běží na různých instancích cloudových služeb. Uzel je samostatná aplikace, která přijímá požadavky na zpracování vstupních dat od fasády. Uzel použije pro zpracování vstupních dat Loader nástroje JaCC. Po načtení vstupních

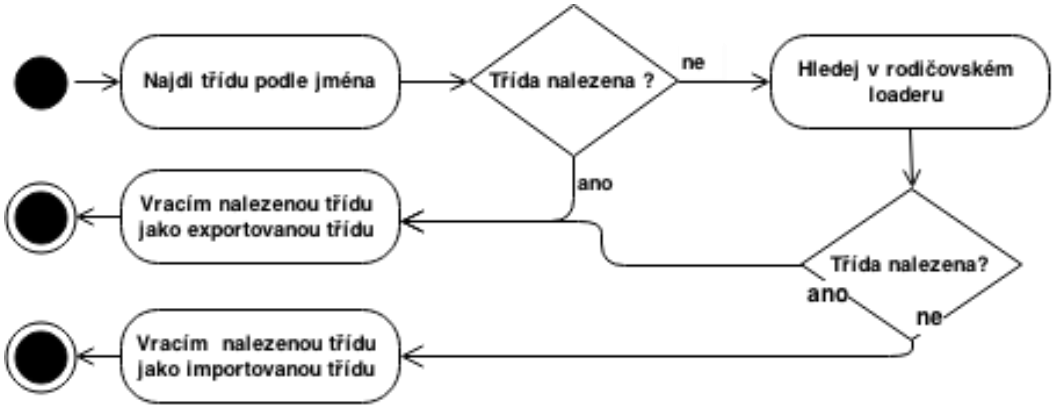
dat odešlou načtená data v objektové podobě zpět na fasádu k dalšímu zpracování. Implementace uzly umožňuje paralelní zpracování vstupních dat (viz kapitola 5.3).

Fasáda je součástí nástroje JaCC. Přijímá úlohu ke zpracování, zajišťuje si načtení dat pomocí dostupných uzlů a po obdržení načtených je poskytuje nástroji ke zpracování. Fasáda také udržuje komunikaci s uzly a při inicializaci zajišťuje napojení na uzly.

5.2 Problém rodičovského loaderu a jeho důsledky

Objekt Loader načítá vstupní data do vnitřní objektové podoby. Během procesu načítání je procházen Java bytecode a z jeho obsahu se provádí rekonstrukce tříd a metod do objektové podoby. Objekt `JClass` je objektová reprezentace načtené třídy a obsahuje metody, atributy a seznam importovaných tříd.

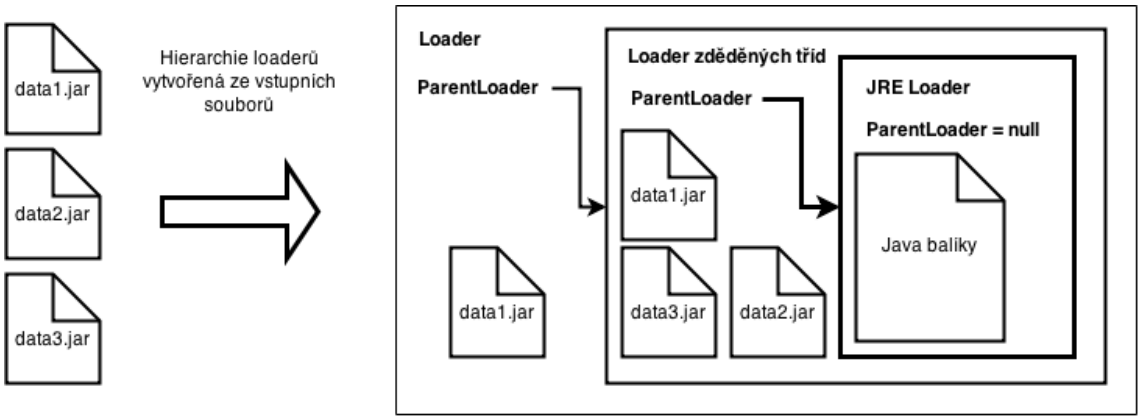
Pokud při načítání `JClass` hledá loader třídu, která není v aktuálních zpracovávaných datech, má možnost zkusit získat třídu přes rodičovský Loader (viz *obrázek 5*).



Obrázek 5 : **Načtení třídy s využitím rodičovských loaderů**

Vzniká tím hierarchie postupně do sebe vnořených loaderů, kteří propojují vstupní data. Jedním z používaných rodičovských loaderů je loader pro čtení tříd z balíků jazyka Java. V každé knihovně nebo aplikaci jsou importované třídy z těchto balíků. Další použití rodičovského loaderu je pro načítání zděděných tříd, které pochází z jiných souborů vstupních dat.

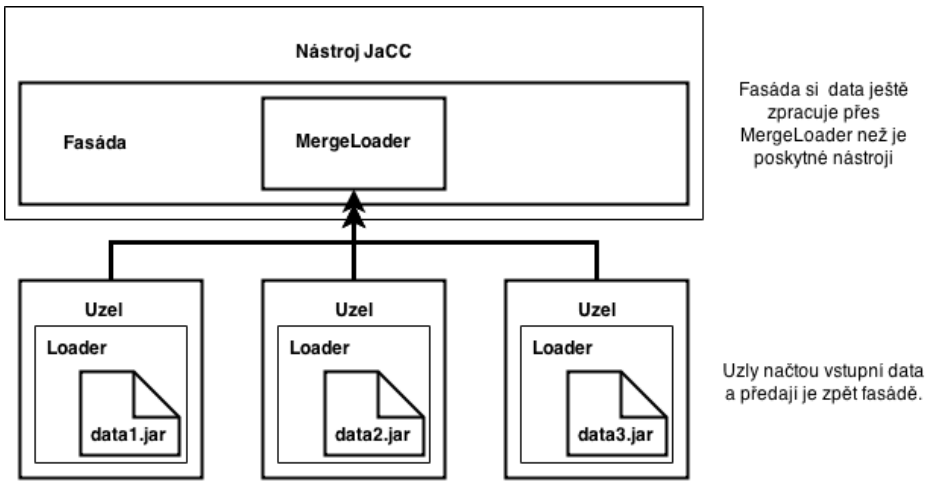
Běžný Loader pro načtení souboru .jar obsahuje ještě rodičovské loadery pro načtení tříd z balíků Javy a zděděných tříd z ostatních souborů ve vstupních datech (viz *obrázek 6*).



Obrázek 6 : **Schéma vytvořené hierarchie Loaderů pro načtení vstupních souborů**

Loader během statické analýzy postupně načítá třídy ze vstupních dat dle potřeb komparátora a během celého procesu zachovává svůj stav. Zachování vnitřního stavu loaderu a propojení vstupních dat rodičovskými loadery znemožňuje provést jejich načtení na různých uzlech, protože by bylo nutné zajistit propojení a komunikaci i mezi jednotlivými uzly, které budou načítání vstupních dat provádět.

Aby nebylo nutné zajišťovat komunikaci mezi uzly, bude pro distribuovanou podobu nástroje nutné tento proces načítání upravit. Uzly budou načítat jednotlivá vstupní data odděleně bez přítomnosti rodičovských loaderů. Po zpracování odešlou data zpět na fasádu, která si provede zpracování vstupních dat pomocí MergeLoaderu, který nahradí přítomnost rodičovských loaderů při procesu načtení (viz *obrázek 7*).

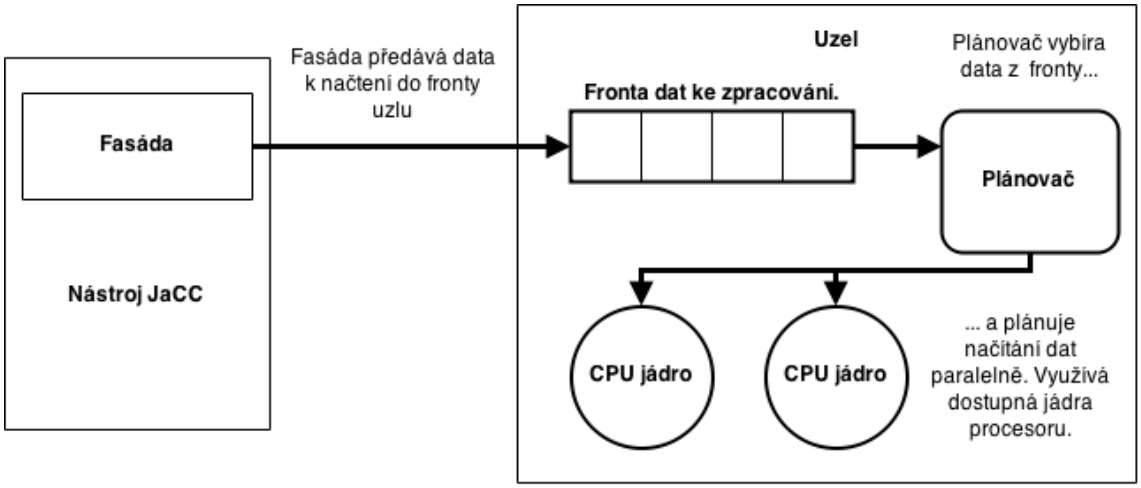


Obrázek 7 : **Schéma načítání dat s využitím MergeLoaderu místo použití rodičovských loaderů**

5.3 Paralelní zpracování vstupních dat

Zpracování vstupních dat probíhá na uzlu přes část nástroje označené jako Loader. Implementace Loaderu umožňuje načítat vstupní data, ale při procesu čtení se nevyužívá žádného paralelního zpracování. Využití více jader procesorů a více procesorů v počítači by bylo nevyužito. O paralelizaci provádění se tak doposud alespoň částečně pokoušela JVM.

Implementace (viz *obrázek 8*) uzlu umožňuje provádět zpracování jednotlivých vstupních dat paralelně najednou. Uzel obsahuje frontu požadavků na zpracování vstupních dat. Požadavek fasády na zpracování vstupních dat se vkládá do této fronty. O jeho provedení se stará plánovač, který vyjme požadavek z fronty a naplánuje jeho provedení na dostupné jádro procesoru. Plánovač zajišťuje paralelní provádění, pokud má ve frontě dostatek požadavků ke zpracování a počítač má více procesorových jader, které může plánovač využívat.



Obrázek 8 : **Realizovaná paralelizace načítání dat na uzlu v distribuovaném modelu**

Paralelní zpracování vstupních dat na uzlech by mohlo přinést určité výhody oproti původní implementaci. Paralelní zpracování by mohlo zkrátit dobu načítání vstupních dat a lépe využívat dostupný hardware na jednotlivých uzlech.

5.4 Komunikace

Komunikace bude probíhat obousměrně mezi fasádou a uzlem. Obě části aplikace budou napsané v jazyce Java a bylo nutné zvolit jednu z dostupných technologií.

Technologie

- **Socket** – Socket je nízko-úrovňové rozhraní pro zasílání dat mezí dvěma uzly v síti. Posílaná data jsou v binární podobě a pro určení místa doručení dat se používá IP adresa, která určuje stroj v síti a port, který určuje aplikaci, která obdrží data. Sockety jsou platformě a systémově nezávislé komunikační rozhraní, ale implementačně náročnější. Výhoda použití socketů je lepší kontrola nad komunikací, vyšší výkon a flexibilita použití. [8]
- **Webová služba** – dnes velice často používaná technologie pro komunikaci mezi dvěma stroji. Realizace komunikace využívá běžně známý a používaný protokol Hyper Text Transfer Protokol (zkratka http) a komunikace probíhá zasíláním dat ve formátu XML[10] nebo JSON[11]. Webová služba je rozhraní vyšší úrovně a je nezávislá na použité technologie. Umožňuje propojovat různé platformy a programovací jazyky. [9]
- **Remote method Invocation** – RMI – Technologie jazyka Java, který vychází z principů vzdáleného volání metod. Pro komunikaci vystaví jedna strana rozhraní s metodami, které bude možné vzdáleně volat. Druhá strana se k danému rozhraní připojí a volá metody, které má k dispozici. Data se předávají při volání metody vstupními parametry nebo návratovou hodnotou metody. Data se předávají v podobě objektů, o jejich převod do binární podoby pro přenos po síti provádí RMI automaticky. RMI je komunikační rozhraní vyšší úrovně ale její použití je pouze v případě že obě strany komunikačního kanálu jsou napsané v jazyce Java. [10]

Formát přenášených dat

Na volbu technologie pro komunikaci mezi jednotlivými částmi distribuované podoby nástroje budu mít vliv i formát přenášených dat. Komunikace bude probíhat v obou směrech, ale v každém směru komunikace budou proudit jiná data.

Při posílání dat ke zpracování jsou vstupní data v podobě souborů .jar nebo .class. Soubory se budou přenášet v podobě pole bajtů.

[10] XML - Extensible Markup Language – značkovací jazyk pro strukturované uložení dat.

[11] JSON - JavaScript Object Notation – alternativa k XML, ukládá strukturovaná data.

Při vrácení načtených vstupních dat bude podoba dat už velice komplikovaná. Bude se jednat o kolekce načtených objektů obsahující další objekty a kolekce objektů. Objektová struktura načtených dat může být i velice rozsáhlá podle obsahu načítaných vstupních dat. Ideální stav je přenášet data už v podobě, ve které je načte loader.

Volba technologie

Obě části distribuovaného nástroje budou v jazyce Java a při výběru technologie se přihlíželo i k náročnosti implementace komunikace v této technologii. V případě socketů by byla implementace rozsáhlejší a náročnější, ale měli bychom větší kontrolu nad komunikací. Ale předpokládaná komunikace se skládá z jednoho odeslání dat na uzel a z jednoho odeslání dat z uzlu. V tomto případě bude použití socketů zbytečné.

Při implementaci použijeme technologii jednodušší na implementaci, a která využije výhod přítomnosti jazyka Java na obou stranách komunikace. RMI bude implementačně jednodušší a při komunikaci budou data odesílána v binární podobě. Alternativou by byla webová služba, ale při přenosu rozsáhlých objektových struktur webovou službou se může negativně projevit převod do XML nebo JSON formátu. Objektová struktura převedená do XML nebo JSON formátu může být několikanásobně větší, než binární podoba, kterou vnitřně používá RMI při komunikaci. Pak by bylo nutné zajistit, aby se prováděla komprese dat před odesláním a při přijetí dekomprese. To jsou další operace s daty navíc, které by se musely provádět.

RMI - Remote Method Invocation

RMI je vysoko-úrovňové aplikační rozhraní pro vzdálené volání metod programovacího jazyka Java. Je jedním z možností jak implementovat komunikaci mezi dvěma procesy běžící na různých strojích nebo na stejném stroji. RMI se v jazyce Java objevilo už ve verzi 1.1 a prošla velkou změnou ve verzi 1.5. [10] [11]

Tato technologie se využívá ve spoustě aplikací a nástrojích, aniž by se o tom vědělo. Příkladem může být nástroj VisualVM, který se používá pro měření aplikací a profilování, nebo jakákoliv aplikace, která je implementovaná pomocí technologie Enterprise JavaBeans. Enterprise JavaBeans jsou serverové komponenty pro modulární vývoj aplikací, které jsou kompletně postavené na technologii RMI. [12]

Výhodou RMI je implementační nenáročnost a jednoduchý komunikační model, ale jeho zásadní nevýhodou je komunikace mimo lokální sít. Realizace komunikace

mimo lokální síť nebo komunikace z jedné lokální sítě do jiné sítě přes internet je implementačně náročnější. Problémy způsobují firewally jednotlivých sítí a překlad síťových adres[12].

Implementace vzdáleného volání metody

Implementace vlastní komunikace není složitá. Vytvoří se rozhraní s metodou, kterou budeme chtít vzdáleně volat a toto rozhraní bude dědit od rozhraní `java.rmi.Remote`. Vytvoří se implementace tohoto rozhraní a implementuje se tělo metody, kterou budeme volat. [10]

Implementace rozhraní se pomocí mechanismů RMI vystaví jako rozhraní pro vzdálené volání metody.

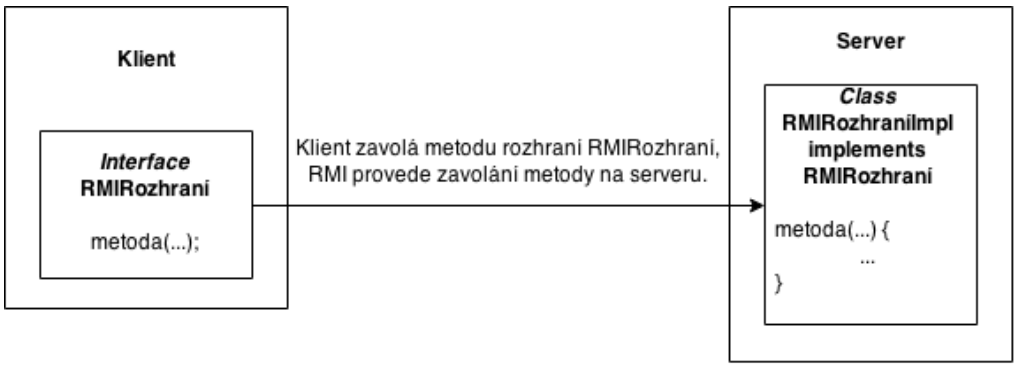
Takto vystavěná implementace rozhraní je dostupná na URL adrese, která má podobu:

```
„adresa stroje : komunikační port / unikátní identifikátor rozhraní“
```

Příklad může vypadat takto:

```
„192.168.1.1:1099/IdentifikátorRozhraní“
```

Implementaci rozhraní dostupnou na URL adrese si lze představit jako server, který budeme volat. Pro vzdálené zavolání metody je nutné vědět adresu, kde je rozhraní vystavěné a vědět jeho datový typ. Aplikace (klient), která chce volat vzdálené metody rozhraní, využije mechanismů RMI a adresy rozhraní pro vytvoření objektu rozhraní, které chceme zavolat. My pak zavoláme metodu rozhraní a RMI samo zařídí provedení vzdáleného volání metody. *Obrázek 9* popisuje klienta, který zavolá metodu na serveru.

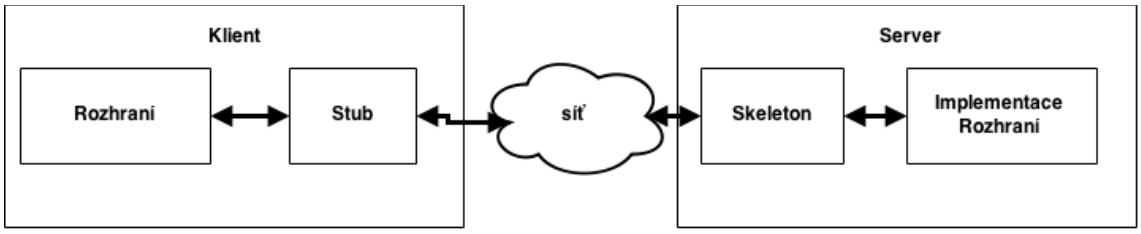


Obrázek 9 : **Model volání vzdálené metody na serveru z klientské aplikace**

[12] Překlad síťových adres – oficiálně označováno jako NAT (z Network address translation)

Vnitřní realizace vzdáleného volání metody

Celý proces komunikace je o něco složitější z pohledu RMI. Při vystavění implementace rozhraní a při získání objektu, přes které se bude vzdálené volání provádět, vznikají další objekty, které danou komunikaci umožňují. Jeden se označuje jako stub a druhý jako skeleton (viz *obrázek 10*).

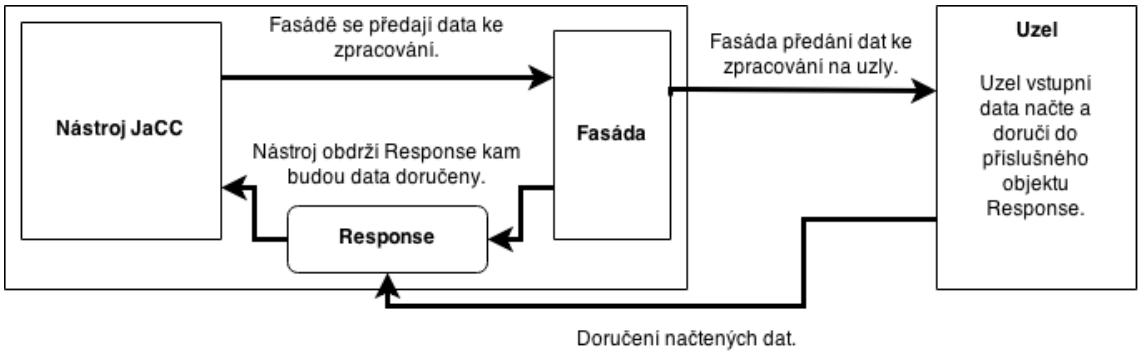


Obrázek 10 : **Model vnitřní realizace vzdáleného volání metody**

Stub a skeleton jsou objekty, které RMI vytvoří přímo na míru vystavěného rozhraní a oba tyto objekty realizují vzdálené volání metody. Realizace je provedena posláním informace o volané metodě a binární podobě jejích vstupních parametrů skrz Socket. Na druhé straně volání dojde k převodu vstupních parametrů zpět do objektové podoby a zavolání správné metody. Návratová hodnota metody je opět poslána v binární podobě zpět. [10]

Komunikační model

Model implementace komunikace zobrazuje *obrázek 11* a je složen ze dvou volání vzdálených metod. První volání volá fasáda uzel a předávat mu data ke zpracování. Při tomto volání by šlo využít návratovou hodnotu metody pro získání zpracovaných dat zpět, ale tato možnost není využita. Pokud dojde k zavolání metody, volající čeká, dokud se daná metoda celá neprovede a nevrátí návratovou hodnotu. Takže by volající čekal, dokud by načtení nebylo dokončené. Uzel zpracovává data z fronty a provádí paralelní zpracování. Tím přicházíme o možnost využít pro doručení zpracovaných dat přes návratovou hodnotu.



Obrázek 11 : **Model komunikace mezi uzlem a fasádou**

Zpracovaná data se doručují druhým voláním vzdálené metody. Tentokrát komunikaci realizuje uzel a předává fasádě načtená vstupní data. Aby fasáda nebyla nucená doručovat načtená data zpět vláknu nástroje, který požádal o jejich načtení, vrací fasáda na požadavek o zpracování vstupních dat objekt response. Do tohoto objektu se načtená data vkládají a nástroj si je odtud může vybrat. Doručení probíhá přímým vložením načtených dat z uzlu do objektu response. Nástroj si data z komponenty response postupně čte, jak jsou data doručovány.

6 Implementace

JaCC je vytvořen v programovacím jazyce Java a využívá nástroj Apache Maven pro řízení a sestavení projektu. Projekt nástroje se skládá z rodičovského projektu a obsahuje jednotlivé části nástroje jako podprojekty. Implementace nástroje využívá pro jednodušší implementaci externí knihovny Apache Commons.

Distribuovaná verze nástroje byla realizovaná jako nový podprojekt `javatypes-remote` do existující projektové struktury nástroje JaCC.

6.1 Rozšíření nástroje JaCC

Původní implementaci nástroje bylo nutné upravit pro realizaci distribuované architektury nástroje.

Serializace

Aby bylo možné přenášet načtená data v objektové podobě mezi fasádou a uzly, bylo nutné všem objektům, které tvoří objektovou podobu načtených dat, implementovat rozhraní `java.io.Serializable`. Rozhraní umožní využít proces serializace objektů do binární podoby a zpět z binární podoby do objektové. Tento proces využívá automaticky RMI při předávání parametrů vzdálené metodě a při předávání návratové hodnoty zpět volajícímu.

Seznam rozšířených tříd: `CanBeImported`, `JAnnotatedElement`, `JAnnotation`, `JClass`, `JGenericDeclaration`, `JMember`, `JModifier`, `JPackage`, `JType`, `ImporterTuple`, `Pair` a `JBlackBoxComponent`.

Loader

Implementace loaderu jsou stavové a udržují si načtenou objektovou strukturu. Pro potřeby komparátoru je nutné načíst kompletní objektovou podobu dat a umožnit ji přenést.

Pro přenos těchto dat byl vytvořen objekt `JBlackBoxComponent` a jeho implementace, který během přenosu obsahuje kolekce objektů, které představují načtený bytecode. Zároveň obsahuje název souboru, ze kterého byly načteny.

Zároveň došlo k rozšíření rozhraní `JClassLoaderFacade` a jeho implementace o metody (viz *ukázka 6-1*), která jsou schopné ze vstupních souborů provést načtení bytecodu a vrátit ho v podobě objektu `JBlackBoxComponent`.

```
• JBlackBoxComponent getAPI(File... files);
• Map<File, JBlackBoxComponent> getAPIComponents(File... files);
```

***Ukázka 6-1* : Přidané metody do rozhraní JClassLoaderFacade**

6.2 Projekt javatypes-remote

Project `javatype-remote` obsahuje kompletní aplikační rozhraní realizované distribuované podoby nástroje. Project je realizován jako Maven podproject nástroje JaCC se snahou držet se použitých konvencí v implementaci nástroje.

V implementaci je oddělená řídící logika uzlu od komunikační v různých rozhraních. Při realizaci komunikace přes RMI musí všechny metody obsahovat definici výjimky `RemoteException`, které je nutnou podmínkou vystavení metody pro vzdálené volání. Oddělením řídící logiky od komunikační do různých rozhraní není nutné, aby všechny metody komponenty obsahovaly definici na výjimku `RemoteException`.

Rozhraní je možné použít i pro implementaci dalších částí nástroje, které by měly využívat výhod distribuované architektury, nebo je možné změnit původní implementaci bez nutných zásahů do ostatních částí implementace. Realizované aplikační rozhraní je možné po menších úpravách použít i pro realizaci distribuované architektury jiné aplikace.

Struktura balíčků projektu

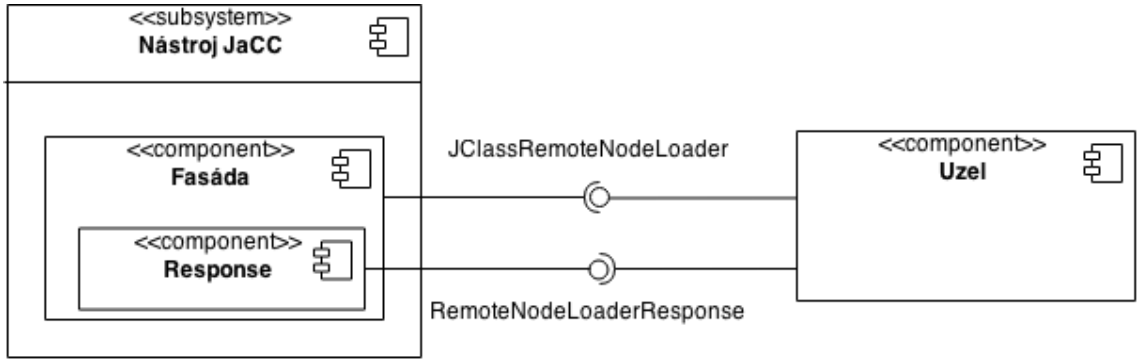
Struktura balíčků projektu (viz *ukázka 6-2*) byla vytvořena podle jednotlivých komponent tvořící realizované aplikační rozhraní distribuované architektury nástroje.

- **cz.zcu.kiv.jacc.remote**
- **cz.zcu.kiv.jacc.remote.facade**
- **cz.zcu.kiv.jacc.remote.monitor**
- **cz.zcu.kiv.jacc.remote.node**
- **cz.zcu.kiv.jacc.remote.response**
- **cz.zcu.kiv.jacc.remote.task**

***Ukázka 6-2 :* Kompletní struktura balíčků projektu**

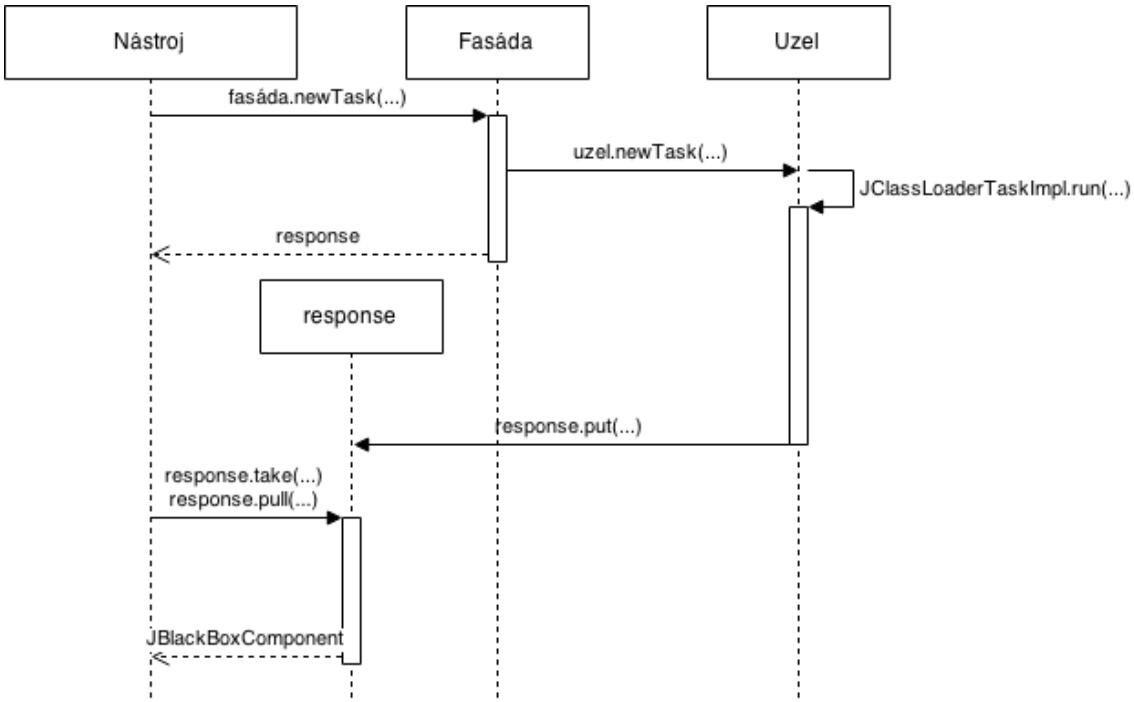
6.3 Komponentový model implementace

Komponentový model implementace (viz *obrázek 12*) se skládá ze tří komponent a vychází přímo z Komunikačního modelu v kapitole 5.4.



***Obrázek 12* : Komponentový diagram implementace**

Komponenty fasáda a response jsou součástí nástroje JaCC a komponenta uzel je samostatná aplikace, která pro nástroj JaCC provádí načítání vstupních dat, které obdrží od fasády. Celý proces zpracování dat začíná tím, že nástroj předá data fasádě. Fasáda data rozdělí mezi aktivní uzly. Uzly načtou vstupní data a načtená data vloží do objektu response. Z objektu response si nástroj přečte zpět svá data. *Obrázek 13* popisuje kompletní průchod vstupních dat fasádou a uzlem.



***Obrázek 13 :* Sekvenční diagram zpracování vstupních dat fasádou s využitím uzlu**

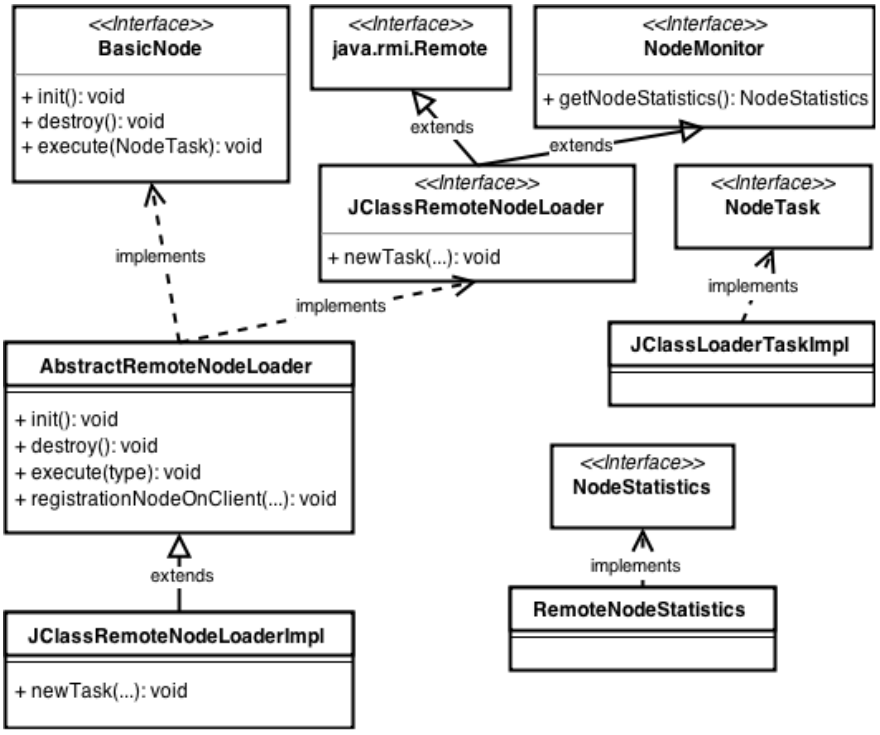
6.4 Uzel

Uzel je jednou z komponent, které tvoří distribuovanou architekturu nástroje. Je implementovaná, aby po inicializaci vyčkávala, dokud ji komponenta fasáda nezadá zpracovat vstupní data. Po zpracování vstupních dat do objektové podoby jsou data odeslána zpět fasádě do příslušné komponenty Response.

Implementace uzlu je tvořena skupinou rozhraní a tříd. Celá implementace se nachází v balíčkách (viz *ukázka 6-3*) a hierarchie tříd zobrazuje diagram tříd (viz *obrázek 14*).

- `cz.zcu.kiv.jacc.remote`
- `cz.zcu.kiv.jacc.remote.monitor`
- `cz.zcu.kiv.jacc.remote.node`
- `cz.zcu.kiv.jacc.remote.task`

Ukázka 6-3 : **Balíčky implementace komponenty Uzel**



Obrázek 14 : **Diagram tříd implementace komponenty Uzel**

Rozhraní

- **`BasicNode`** – Základní rozhraní definující komponentu. Definuje metody pro inicializaci `init(…)`, zničení objektu `destroy(…)` a provádění zadaných úkolů

`execute(…)`. Úkol, který má uzel provést je v podobě objektu, který implementuje rozhraní `NodeTask`.

- **`NodeTask`** – Rozhraní, které definuje podobu úkolu, který bude uzel provádět. Rozhraní je potomkem rozhraní `java.lang.Runnable`.
- **`NodeMonitor`** – Monitorovací rozhraní pro komponentu. Umožňuje zjistit počet provedených úloh, zjistit stav uzlu, volnou operační paměť, počet paralelně prováděných úloh a maximální počet paralelně prováděných úloh.
- **`NodeStatistics`** – Rozhraní, které definuje objekt pro reprezentaci kompletní statistiku uzlu, kterou poskytuje rozhraní `NodeMonitor`. Zavoláním metody `getNodeStatistics(…)` v rozhraní `NodeMinitor` se získá kompletní statistika v podobě objektu `NodeStatistics` místo získání dat jednotlivými metodami rozhraní `NodeMonitor`.
- **`JClassRemoteNodeLoader`** – Rozhraní pro definici komunikace s využitím RMI. Rozhraní je potomkem rozhraních `java.rmi.Remote` a `NodeMonitor`. Rozhraní definuje základní metody pro komunikaci s uzlem a metodu `newTask(…)` pro vytvoření nové úlohy zpracování vstupních dat na uzlu.

Třídy

- **`AbstractRemoteNodeLoader`** – Abstraktní třída s hlavní implementací komponenty. Obsahuje řídící logiku životního cyklu komponenty a implementuje část komunikace spojenou s rozhraním `NodeMonitor`.
- **`JClassRemoteNodeLoaderImpl`** – Třída s implementací metody `newTask(…)` pro zadávání úloh načtení vstupních dat nástroje. Metoda z parametrů volání metody vytvoří objekt `JClassLoaderTaskImpl`, který reprezentuje zadaný úkol na zpracování vstupních dat a pomocí metody `execute(…)` nechá danou úlohu provést.
- **`JClassLoaderTaskImpl`** – Objekt představující zadaný úkol na zpracování vstupních dat nástroje. Po vytvoření je úkol vykonán uzlem. O vykonání úkolu se stará plánovač.
- **`RemoteNodeStatistics`** – Implementace objektu pro kompletní statistiku uzlu. Slouží jako transportní objekt těchto informací.

Algoritmy

Paralelní provádění zadaných úkolů

Implementace úkolu načtení vstupních dat je v podobě vlákna. Úkol implementuje rozhraní `java.lang.Runnable` a paralelní provádění je možné provádět prostým vytvořením více vláken najednou a jejich spuštění. Taková implementace je vysoce neefektivní. Pokud bude paralelně prováděno mnohem více vláken, než je dostupných procesorových jader, bude docházet k častému přeplánování vláken a vlákna se budou navzájem zpomalovat. Zároveň je opakované vytváření velkého množství vláken nákladná operace z pohledu JVM, protože je nutné pro každé vlákno vytvořit jeho kontext[13].

Implementace paralelního provádění je z těchto důvodů realizovaná pomocí plánovače `java.util.concurrent.ThreadPoolExecutor`. Plánovač realizuje provádění jednotlivých úkolů a nevytváří přitom nová vlákna, ale dochází k používání předem vytvořených vláken. Zároveň obsahuje spoustu nastavení pro úpravu chování plánovače a umožňuje získávat z něj statistiky o prováděných úkolech. Pro plánování vláken na jednotlivá procesorová jádra je nutné, aby plánovač měl vytvořeno stejně vláken jako je dostupných procesorových jader. Java zajistí mapování jednotlivých vláken na procesorová jádra. [13]

Inicializace probíhá v metodě uzlu `init(…)`, kde dojde k vytvoření instance plánovače (viz *ukázka 6-4*).

```
executor = new ThreadPoolExecutor(corePoolSize, corePoolSize, 1, TimeUnit.MINUTES,
new LinkedBlockingQueue<Runnable>());
```

Ukázka 6-4 : **Inicializace plánovače ThreadPoolExecutor v metodě init(...)**

Důležité jsou vstupní parametry:

- **`corePoolSize`** – počet prováděných vláken najednou
- **`maximumPoolSize`** – maximální počet prováděných vláken najednou
- **`BlockingQueue<Runnable> workQueue`** – fronta úkolů – v implementaci použita instance `java.util.concurrent.LinkedBlockingQueue`

Ostatní vstupní parametry jsou nastaveny defaultně.

[13] Kontext vlákna obsahuje stav zásobníku, priority a data vlákna.

Metoda implementace metody `execute(…)` z rozhraní `BasicNode` nerealizuje provádění zadaných úloh v podobě objektů `NodeTask`, ale vkládá tyto objekty do fronty plánovače. O provádění se stará plánovač, když je úkol na vrcholu frontě.

Načtení vstupních dat

Implementace načtení vstupních dat je realizovaná jako objekt třídy `JClassLoaderTaskImpl`. Při zavolání metody uzlu `newTask(…)` dojde k vytvoření instance `JClassLoaderTaskImpl`. Parametry konstruktoru instance jsou shodné s parametry volané metody `newTask(…)`. Hlavička metody `newTask(…)` (viz *ukázka 6-5*). V konstruktoru dojde k uložení vstupních dat v podobě bytového pole na disk v podobě dočasného souboru, aby vstupní data nezabíraly operační paměť do té doby, než bude úkol prováděn. Instance je pak vložena do fronty plánovače metodou `execute(…)`.

```
void newTask(String responseUrl, String responseID, byte[] file, String fileName)
```

***Ukázka 6-5* : Metoda pro předání úkolu z fasády na uzel**

- Pří provádění úkolu se první provede načtení vstupních dat nástrojem JaCC ze vstupního souboru. Pro načtení se použije instance `JClassLoaderFacade` (viz *ukázka 6-6*).

```
JClassLoaderFacade facade = JClassLoaderFacade.JAR_MEMORY_FACADE;
Map<File, JBlackBoxComponent> boxComponentMap = new HashMap<File,
JBlackBoxComponent>();
boxComponentMap = facade.getAPIComponents(file);
```

***Ukázka 6-6 :* Načtení vstupních dat komponentou loader**

- Po načtení vstupních dat dochází k odeslání dat na fasádu do příslušné instance komponenty response. Pro odeslání se provádí inicializace spojení na příslušnou instanci response, kam mají být data předána. K předání dat slouží metoda `put(…)` komponenty response. Po odeslání dat dojde k uklizení vstupních dat z disku a ukončení úkolu.

Inicializace spojení s objektem Response

Komponenta response (viz kapitola 6.6) slouží pro doručení načtených vstupních dat z uzlu zpět vláknu, které požádalo fasádu o načtení dat.

Response objekt je identifikovaný pomocí jednoznačného identifikátoru. Každá fasáda má své instance komponenty response. Pro doručení dat do správné schránky je nutné znát adresu stroje, kde fasáda běží a jednoznačný identifikátor instance response. Všechny tyto údaje se uzel dozví ve chvíli, kdy fasáda předává data metodou `newTask(…)`.

Inicializace spojení (viz *ukázka 6-7*) realizuje metoda `initResponseConnection(…)` z objektu `JClassLoaderTaskImpl`. Inicializace spojení probíhá získáním objektu `Registry`, který slouží pro inicializaci spojení s rozhraním pro vzdálené volání metod. Zavoláním metody `lookup(…)` z objektu `Registry` dojde k vytvoření spojení na vzdálené rozhraní a metoda vrací instanci tohoto rozhraní, které realizuje vzdálené volání. Pro inicializaci spojení je nutné znát adresu stroje, kde je vzdálená služba dostupná a identifikátor této služby.

```
Registry registry = LocateRegistry.getRegistry(
    responseUrl,
    RemoteNodeLoaderResponse.REMOTE_LOADER_NODE_RESPONSE_PORT,
    socketFactory);

response = (RemoteNodeLoaderResponse)registry.lookup(
RemoteNodeLoaderResponse.REMOTE_LOADER_NODE_RESPONSE_PREFIX + responseID);
```

Ukázka 6-7 : **Inicializace spojení na RMI rozhraní komponenty response**

Inicializace uzlu

Vytvořená instance uzlu se inicializuje do základního stavu metodou init(..). Pro inicializaci je potřeba znát pouze adresu stroje, kde uzel poběží, a název vzdálené služby, pod kterou bude dostupný.

Metoda `init(…)` (viz *ukázka 6-8*) zavolá metody `exportObject(…)` třídy `UnicastRemoteObject`, tím dojde k rozšíření aktuální instance o mechanismy pro vzdálené volání metod. Aktuální instance bude schopná realizovat vzdálené volání. Dalším krokem je získání objektu `Registry`. Pomocí tohoto objektu vystavíme metodou `bind(…)` aktuální instanci jako službu pro vzdálené volání metod.

```
JClassRemoteNodeLoader stub = (JClassRemoteNodeLoader)
UnicastRemoteObject.exportObject(this, 0);
Registry registry = LocateRegistry.createRegistry(rmiRegistryPort);
registry.bind(serviceName, stub);
```

Ukázka 6-8 : **Inicializace RMI rozhraní uzlu pro vzdálené volání metod**

Parametry pro vystavění služby jsou jen identifikátor služby a komunikační port, na kterém bude služba dostupná.

Po inicializaci RMI služby se inicializuje plánovač `ThreadPoolExecutor`. Pokud není před metodou `init(…)` nastaven počet vláken, která má plánovač používat k plánování úkolů, dojde k nastavení dle zjištěných údajů metodou `Runtime.getRuntime().availableProcessors()`.

Dalším krokem inicializace může být zavoláním metody `registrationNodeOnClient(…)`, pokud se má uzel sám registrovat u fasády jako dostupný uzel pro načtení vstupních dat.

Parametry komponenty

Uzel umožňuje inicializaci bez jakéhokoliv zadaného vstupního parametru a bude funkční a dostupný na adrese stroje, kde byl spuštěn. K tomu využívá defaultní parametry. Jinak je možné uzel kompletně přenastavit dle potřeb aplikace. Parametry jsou atributy instance uzlu.

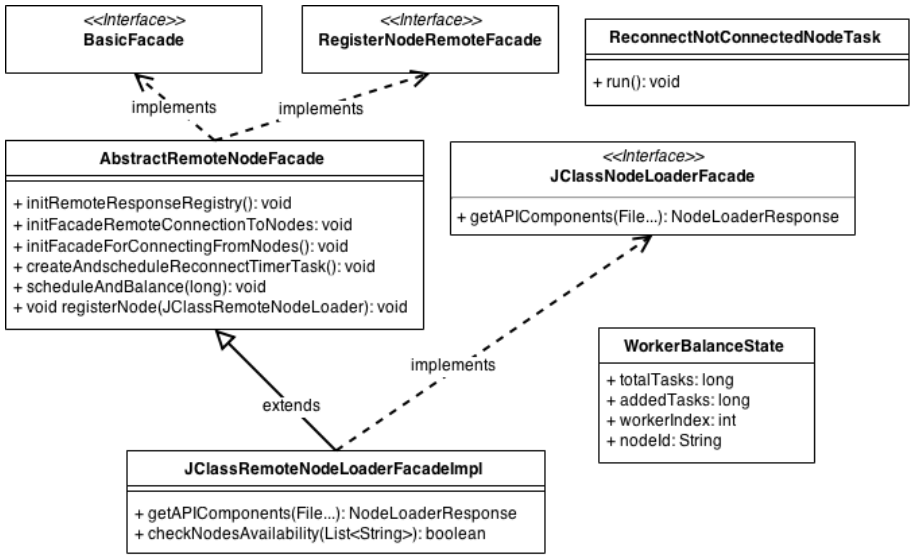
- **`nodeID`** – jednoznačný identifikátor uzlu, slouží pro identifikaci uzlu mezi ostatními uzly dle potřeb.
- **`serviceName`** – identifikátor RMI služby, pod kterou bude uzel dostupný.
- **`rmiRegistryPort`** – komunikační port RMI služby, základní nastavení je 1099.
- **`corePoolSize`** – počet vláken, která bude plánovač využívat pro provádění úkolů.
- **`clientFacadeServicePort`** – komunikační port, pod kterým jsou dostupné fasády pro registraci uzlu.
- **`clientFacadeServiceName`** – identifikátor RMI služby, pod kterým jsou dostupné fasády pro registraci uzlu.
- **`clientFacadeHosts`** – seznam adres, na kterých jsou dostupné fasády. U těchto fasád se pokusí uzel registrovat jako dostupný uzel.

6.5 Fasáda

Implementace objektu se nachází v balíčkách (viz *ukázka 6-9*) a hierarchie tříd zobrazuje diagram tříd (viz *obrázek 15*).

- `cz.zcu.kiv.jacc.remote.facade`

Ukázka 6-9 : **Balíček s implementací komponenty fasáda**



Obrázek 15 : **Diagram tříd implementace komponenty fasáda**

Rozhraní

- **`BasicFacade`** – rozhraní definující metody pro řídící logiku fasády. Obsahuje metody pro inicializaci, nastavení fasády a metody pro zotavení systému z chyb (viz kapitola 6.9).
- **`RegisterNodeRemoteFacade`** – rozhraní definující RMI rozhraní pro registraci uzlů na fasádě v případě, že se uzly mají samy registrovat na fasádě.
- **`JClassNodeLoaderFacade`** – rozhraní definuje metodu pro odeslání dat na jednotlivé uzly. Nástroj používá metodu pro předání dat ke zpracování a získá response, ze kterého si zpracovaná data vyzvedne.

Třídy

- **`AbstractRemoteNodeFacade`** – Abstraktní třída s implementací komponenty fasáda. Obsahuje řídící logiku životního cyklu komponenty, implementaci

plánovacího algoritmu pro rozložení úloh na uzly, metody pro zotavení systému z chyby a inicializaci RMI rozhraní fasády.

- `JClassRemoteNodeLoaderFacadeImpl` – Třída s kompletní implementací fasády pro nástroj JaCC. Implementuje metodu `getAPIComponents(…)` pro zadání požadavku na zpracování vstupních dat pro nástroj.
- `ReconnectNotConnectedNodeTask` – vlákno, pro testování nedostupných uzlů a obnovování jejich spojení. Vlákno se provádí opakovaně s předem nastavenou periodou spouštění, otestuje nedostupné uzly a dostupné uzly zařadí zpět mezi aktivní uzly pro plánování.
- `WorkerBalanceState` – datový objekt, který uchovává informace pro plánování rozložení vstupních dat na jednotlivé uzly.

Algoritmy

Inicializace fasády

Komponentu vytvoříme jedním z konstruktorů, dle použitého konstruktoru může dojít rovnou k automatické inicializaci fasády, pokud má fasáda data, která potřebuje nebo je nutné použít metody k inicializaci určené. Každá z inicializačních metoda inicializuje na fasádě jinou část a požaduje jiná nastavení nebo vstupní parametry.

Metoda `initRemoteResponseRegistry(…)` je první z metod, které je nutné zavolat pro inicializaci fasády. Inicializace je nutná pro správné fungování fasády a response komponent. Připravuje fasádě mechanismy RMI rozhraní, která budou na fasádě použitá pro komponenty response. Metoda vytvoří pro komponentu response globální objekt Registry pro komunikační port komponenty response. Dalším krokem je vytvoření instance `RMISocketFactory` s upraveným nastavením socketu, které RMI bude používat. Takto vytvořená instance se globálně nastaví, aby byla společná pro všechny instance komponenty response, které budou vytvořeny (viz *ukázka 6-10*).

```
       . . .

try {
    RMISocketFactory.setSocketFactory(socketFactory);
} catch (IOException e) {
    logger.warn(e.getMessage()); // factory is already set
}
```

Ukázka 6-10 : **Inicializace RMISocketFactory**

Další metodou inicializace je metoda `initFacadeRemoteConnectionToNodes(…)`, která provede spojení s dostupnými uzly. Pro provedení inicializace potřebuje metoda URL adresy uzlů, na které se bude pokoušet připojit. Pokud se s daným uzlem spojí, zařadí uzel mezi aktivní uzly, které jsou používány pro zpracování vstupních dat nástroje. Seznam URL strojů s uzly se předává konstruktorem, který tento parametr obsahuje a zároveň dochází k inicializaci fasády.

Poslední metodou pro inicializaci je metoda `initFacadeForConnectingFromNodes(…)` a její použití závisí na zvoleném způsobu inicializace spojení (viz kapitola 6.8). Metoda slouží pro vystavění RMI rozhraní, které umožňuje uzlům registraci na fasádě. V tomto případě se nebude fasáda připojovat na uzly, ale uzly se po inicializaci zkusí spojit s fasádou a zaregistrovat se jako aktivní uzel. Pro registraci mají uzly implementovanou metodu `registrationNodeOnClient(…)`, které stačí předat informace o fasádách, kde se mají registrovat. Metoda `initFacadeForConnectingFromNodes(…)` nemůže být volána v konstruktoru, protože objekt v té době ještě není vytvořen a nelze ho vystavit jako RMI službu. Nešlo by tak vystavit fasáda pro registraci uzlů.

Plánování úkolů na jednotlivé uzly

Nástroj si pro načtení vstupních dat zavolá metodu fasády `getAPIComponents(…)` a předá fasádě vstupní data, která chce načíst. Fasáda se podívá, kolik vstupních dat celkem dostala a provede jejich předání jednotlivým uzlům. Kolik se předá dat ke zpracování na uzly, je záležitostí vyvažovacího algoritmu.

Vyvažovací algoritmus se snaží data rozložit na uzly tak, aby uzly zbytečně nepřetěžoval a všechny uzly měly nějakou práci. Metoda `scheduleAndBalance(…)` provádí vyvážení vstupních dat na uzly a vrací kolekci objektů `WorkerBalanceState`, které obsahují informaci, kolik vstupních dat se má odeslat na daný uzel.

Algoritmus vychází z informace, kolik dosud nezpracovaných vstupních dat má uzel ve frontě. Připraví si pole objektů `WorkerBalanceState` pro každý dostupný uzel. Pole si seřadí podle velikosti jejich front. Dalším krokem je už samotné rozdělování úkolů mezi jednotlivé uzly.

Rozdělování probíhá tak, že uzel s nejkratší frontou dostane přesně tolik vstupních dat, dokud nemá stejně dlouhou frontu jako jeho soused v poli. Algoritmus funguje

na principu doplnění seřazeného pole na stejné hodnoty. Průběh vyvažování délek front zobrazuje *tabulka 1*.

	Uzel 1	Uzel 2	Uzel 3	Uzel 4
	Délka fronty	Délka fronty	Délka fronty	Délka fronty
Výchozí stav	**7**	**1**	**3**	**10**
Seřazení	**1**	**3**	**7**	**10**
1. vyvážení	**1**+2 = 3	**3**	**7**	**10**
2. vyvážení	**1**+2+4 = 7	**3**+4 = 7	**7**	**10**
3. vyvážení	**1**+2+4+3 = 10	**3**+4+3 = 10	**7**+3 = 10	**10**
Závěrečné vyvážení	**1**+2+4+3+5 = 15	**3**+4+3+5 = 15	**7**+3+5 = 15	**10**+5 = 15
Přidané hodnoty	**14**	**12**	**8**	**5**

Tabulka 1 : **Ukázka algoritmu pro rozdělení dat mezi uzly s vyvážením**

Tabulka zobrazuje postupné vyvažování počátečního stavu na výsledný stav. V každém kroku vyvážení se přidá taková hodnota, jako je rozdíl aktuální hodnoty a nejbližší vyšší hodnoty. Další krok vyvážení vychází z konečného stavu předchozího vyvážení. Ve chvíli vyvážení všech na stejnou hodnotu dojde k rovnoměrnému rozdělení zbytku. Poslední řádek tabulky pak zobrazuje informaci, kolik každý uzel dostane vstupních dat k načtení. Informace, které zobrazuje tabulka, jsou uložené po výpočtu v objektech `WorkerBalanceState`.

Rozdělování úkolů mezi uzly v závislosti na délkách front není úplně ideální způsob. Podstatná je spíš velikost vstupních dat, od které se dá určit orientační doba, nutná pro načtení a tím i lepšímu rovnoměrnému rozdělení vstupních dat mezi uzly. Vylepšení plánovacího algoritmu je jedním z návrhů na vylepšení v kapitole 10.

Po provedení výpočtu rozložení se data začnou postupně rozesílat na jednotlivé uzly metodou `newTask(…)` (viz *ukázka 6-11*).

```
WorkerBalanceState[] balanceWorkerState = scheduleAndBalance(files.length);
for (int workerIndex = 0; workerIndex < balanceWorkerState.length; workerIndex++)
{

    int nodeIndex = balanceWorkerState[workerIndex].getWorkerIndex();
    long countOfSentTask = balanceWorkerState[workerIndex].getAddedTasks();
    nodeIds.add(balanceWorkerState[workerIndex].getNodeId());

    for (int taskCounter = 0; taskCounter < countOfSendedTask; taskCounter++) {
        try {
                                    . . .
            remoteNodeLoader.newTask(getResponseUrl(), responseId,
                bFile, originFile.getName());
                                    . . .
        }
    }
}RMISocketFactory socketFactory = new RMISocketFactory() {
```

***Ukázka 6-11* : rozdělení dat mezi uzly dle vypočteného vyvážení z metody scheduleAndBalance(...)**

Parametry komponenty

Parametry komponenty jsou realizovány jako vnitřní atributy komponenty. Všechny parametry mají základní hodnotu po vytvoření instance, nebo si mohou základní hodnotu zjistit.

- **`responseUrl`** – parametr důležitý pro odesílání dat zpět z uzlu do objektu response. URL by měla mít podobu IP adresy nebo doménového jména, pod kterým bude daný stroj dostupný v síti. Pokud není hodnota nastavena konstruktorem nebo příslušnou metodou, je nastavena základní hodnota, kterou zjistí komponenta zavoláním metody síťového rozhraní jazyka Java `InetAddress.getLocalHost().getHostName()`. Ne hodnotu zjištěnou metodou `getHostName(…)` není dobré se spoléhat, protože získané doménové jméno může být jedno z mnoha a mohl by nastat problém v komunikaci.
- **`rmiRegistryPort`** – komunikační port RMI rozhraní, které využívají uzly, pokud se mají samy registrovat na fasádu jako aktivní uzel. Základní nastavení je 1099.
- **`serviceName`** – název RMI rozhraní, které využívají uzly, pokud se mají samy připojovat na fasádu jako aktivní uzel.
- **`reconnectTaskPeriod`** – hodnota v milisekundách, doba mezi jednotlivými pokusy o připojení neaktivních uzlů vláknem `ReconnectNotConnectedNodeTask`. Základní nastavení je 30 minut.

- `reconnectTaskDelay` – hodnota v milisekundách, doba před prvním spuštěním vlákna `ReconnectNotConnectedNodeTask`, které připojuje neaktivní uzly.
- `nodeURLs` – seznam URL adres strojů, na kterých jsou běžící uzly. URL by měla mít podobu IP adresy nebo doménového jména, pod kterým bude daný stroj dostupný v síti.

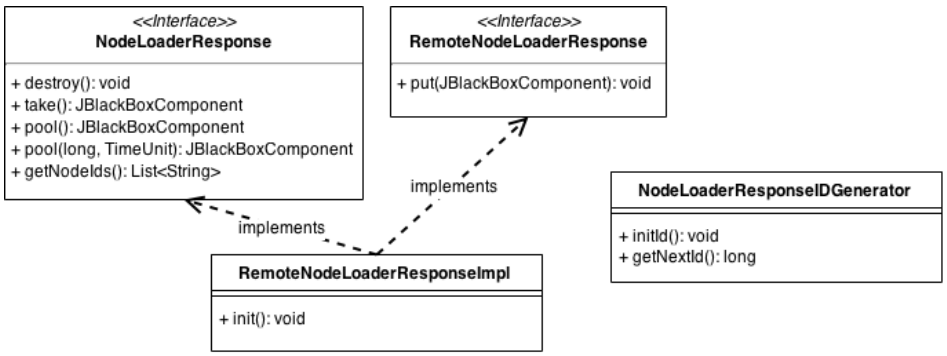
6.6 Response

Komponenta response představuje schránku, do které se posílají načtená data. Každá úloha nástroje má svojí instanci komponenty, která je vytvořena ve chvíli, kdy nástroj požádá fasádu o zpracování dat. Fasáda vrátí instanci komponenty response, který je inicializován a připraven pro příjem dat z uzlů.

Implementace komponenty se nachází v balíčkách (viz *ukázka 6-12*) a hierarchie tříd zobrazuje diagram tříd (viz *obrázek 16*).

- **cz.zcu.kiv.jacc.remote.response**

Ukázka 6-12 : **Balíčky implementace komponenty Response**



Obrázek 16 : **Diagram tříd implementace komponenty Response**

Rozhraní

- `NodeLoaderResponse` – rozhraní pro definici řídící logiky komponenty response. Rozhraní definuje metody `take(…)` a `pool(…)` pro výběr dat vložených do komponenty a metodu `destroy(…)` ukončení životního cyklu objektu.
- `RemoteNodeLoaderResponse` – rozhraní pro definici komunikační logiky vzdáleného volání metod. Rozhraní je potomkem rozhraní `java.rmi.Remote`. Definuje metodu `put(…)` pro vložení načtených dat z uzlu zpět na fasádu a metodu `init(…)` pro inicializaci rozhraní pro vzdálené volání metody `put(…)`.

Třídy

- `NodeLoaderResponseIDGenerator` – generátor identifikátorů komponenty response. Generátor po inicializaci generuje identifikátory pro jednotlivé instance response objektu, aby byla zaručena unikátnost.
- `RemoteNodeLoaderResponseImpl` – Implementace komponenty response. Třída obsahuje kompletní implementaci komponenty. Vnitřní realizace komponenty je tvořena frontou, uzly vkládají do fronty načtená data a z druhé strany si data vybírá nástroj dle dostupnosti. Z komunikačního pohledu funguje instance jako schránka pro uložení dat. Komponenta poskytuje blokující i neblokující metody pro výběr dat z komponenty. RMI nezaručuje bezpečný přístup z více vláken, ale metoda `put(…)` nemusí být ošetřena jako kritická sekce. Vnitřní implementace komponenty je tvořena implementací `BlockingQueue`, která zaručuje bezpečný přístup z více vláken pro implementované metody. Ošetřit kritickou sekci bude nutné v případě změny vnitřní implementace komponenty.

Algoritmy

Inicializace komponenty Response

Komponenta se vytváří konstruktorem. Během toho se instance pouze připraví, ale ještě není schopná přijímat data od uzlů. Pro inicializaci komunikace je potřeba, aby se objekt vystavil jako RMI rozhraní.

Inicializace komponenty pro komunikaci slouží metoda `init(…)` (viz *ukázka 6-13*), která zavolá příslušné mechanismy aplikačního rozhraní RMI a dojde k vystavení komponenty jako rozhraní pro vzdálené volání metod.

```
RemoteNodeLoaderResponse stub = (RemoteNodeLoaderResponse)
UnicastRemoteObject.exportObject(
this,0);
Registry registry = LocateRegistry.getRegistry(REMOTE_LOADER_NODE_RESPONSE_PORT);
registry.rebind(REMOTE_LOADER_NODE_RESPONSE_PREFIX + taskID, stub);
```

Ukázka 6-13 : **Inicializace RMI rozhraní komponenty response**

Vystavěná komponenta je po vystavění dostupná na adrese stroje a příslušném identifikátoru služby. Protože pro každou fasáda může být aktivních víc komponent Response v závislosti kolik je běžících úloh, je nutné přesně určit instanci komponenty,

aby byly data správně doručeny. K tomu slouží unikátní identifikátor, který je součástí identifikátoru RMI rozhraní.

Přístup k načteným datům

Načtená data vkládají uzly do komponenty metodou `put(…)`. Pro výběr dat jsou implementovány blokující a neblokující metody. Metoda `take(…)` vrací načtená data v případě, že jsou dostupná nebo je volající do té doby v blokovaném stavu. Metoda `pull(…)` vrací data nebo `null` hodnotu, pokud nejsou žádná dostupná. Druhá verze metody `pull(long timeout,…)` vrací data nebo dobu danou parametrem `timeout` čeká, pokud se data do objektu nevloží. Pak vrací data nebo také hodnotu `null`. Metody pro výběr využívají implementace fronty `BlockingQueue` v komponentě.

Parametry komponenty

Komponenta neobsahuje mnoho parametrů nastavení. Potřebuje pouze znát identifikátor případně velikost úlohy pro inicializaci fronty pro ukládání dat. Parametry jsou realizovány jako vnitřní atributy instance.

- **`taskID`** – identifikátor, který jednoznačně identifikuje instanci komponenty Response při vzdáleném volání metod.
- **`queueSize`** – parametr je možné nastavit pouze přes konstruktor, inicializuje velikost použité fronty `BlockingQueue` na hodnotu parametru. Urychluje vkládání prvků do fronty, protože frontě odpadá zbytečné zvětšování své velikosti.
- **`nodeIds`** – kolekce adres uzlů, na které byly odeslány data pro zpracování. Slouží pro zotavení systému z chyby (viz kapitola 6.9).

6.7 Merge loader

Komponenta MergeLoader, která měla nahradit rodičovské loadery (viz kapitola 5.2), nebyla z časových důvodů implementována. Komponenta byla přidána mezi návrhy na vylepšení a rozšíření (viz kapitola 10.1).

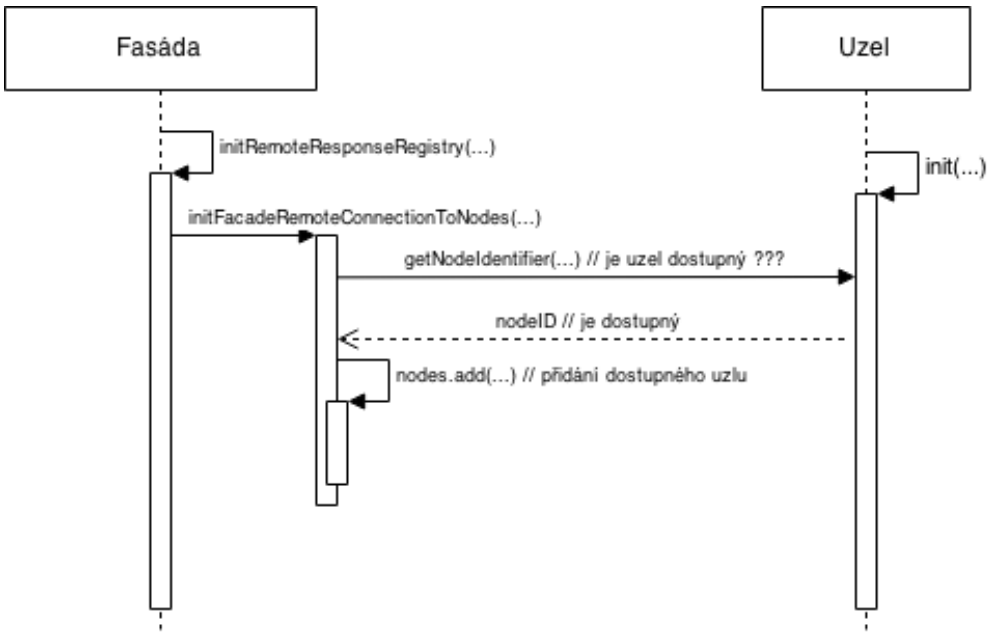
Na uzlech byla implementována filtrace (viz kapitola 6.4 - Načítání vstupních dat) instancí `JClass`, která částečně nahradila chybějící komponentu.

6.8 Inicializace spojení fasády a uzlu

Pro správné fungovaní je nutné, aby fasáda měla dostupný aktivní uzel a mohla mu zadávat vstupní data ke zpracování. Aby systém byl na takovou činnost připraven, je nutné provést inicializaci fasády a uzlů.

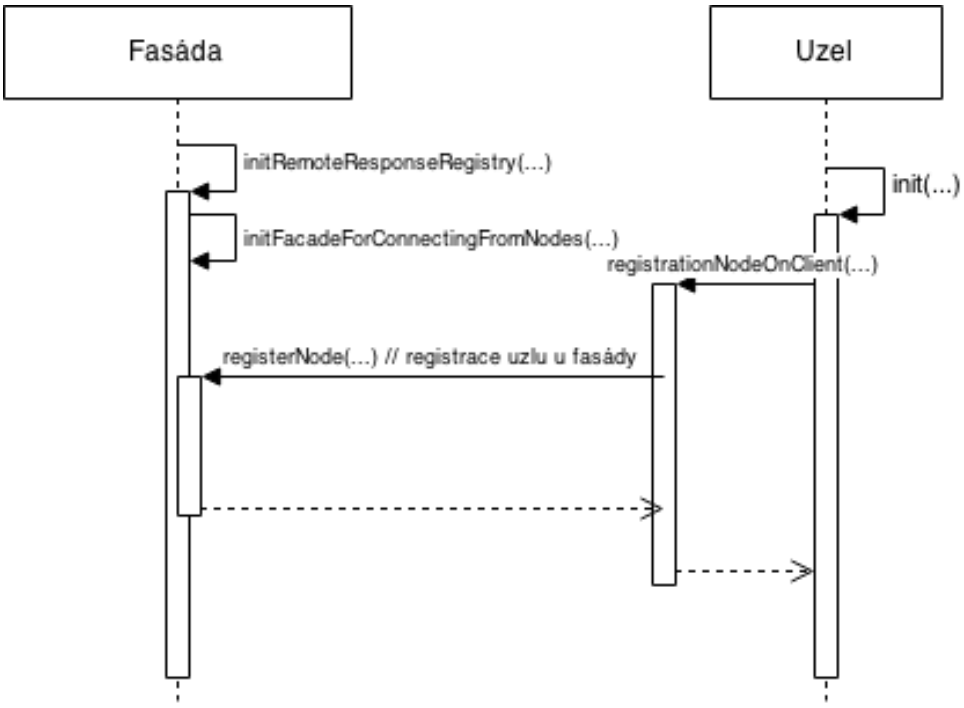
Spojení fasády a uzlu lze vytvořit z obou stran. A oba způsoby je možné kombinovat. Lze mít připravenou skupinu uzlů a zároveň zapínat nové uzly za chodu. Tím je možné dynamicky měnit počet uzlů a využít škálování cloudových služeb

- První způsob spojení s uzlem - fasáda se spojí s předem danými uzly a zaregistruje si je jako dostupné uzly pro načítání dat. K tomu potřebuje fasáda znát URL adresy uzlů, se kterými se má spojit při inicializaci, a uzly musí být už inicializované. Proces spojení od fasády popisuje *obrázek 17*.



Obrázek 17 : **Spojení je vytvořené od fasády**

- Druhý způsob spojení s uzlem - fasáda provede inicializaci a vyčkává, dokud se uzel sám nespojí s fasádou a nezaregistruje se jako dostupný uzel pro načítání dat. Fasáda se inicializuje bez znalosti URL adres uzlů. Uzel po své inicializaci provede registraci na příslušné fasádě přes metodu `registrationNodeOnClient(…)`. Fasáda musí být v tu chvíli už inicializovaná, aby byla schopná provádět registrace. Proces spojení od uzlu popisuje *obrázek 18*.



Obrázek 18 : **Spojení je vytvořené od uzlu**

6.9 Zotavení systému z chyb

Chyba v běžné aplikaci se ošetří a aplikace může běžet dál. Chyba v distribuované aplikaci je komplikovanější. V případě že dojde k chybě na některé části distribuované aplikace, je nutné chybu ošetřit nejen v místě, kde vznikla, ale musí se s tím vypořádat celý systém, pokud je to nutné.

Distribuovaná část nástroje JaCC se stará o načítání dat. V případě že dojde k jakékoliv chybě na uzlu, který čte data. Musí se celý systém umět vypořádat s tím, že daná data nejsou a možná už nebudou k dispozici. V aplikaci je mnoho způsobů, jak může dojít k chybě. Obecně se dají rozdělit do dvou skupin:

- chyby spojené s komunikací a
- chyby spojené se zpracováním dat.

Chyby spojené s načtením dat

Chyby spojené s načtením dat vznikají přímo na uzlu při zpracování vstupních dat. Důvod k chybě při zpravování dat je mnoho, může být problém se vstupním souborem, ze kterého se data čtou, nebo může být neošetřená chyba přímo v kódu nástroje.

Pokud k takové chybě dojde na uzlu, je nutné, aby se s chybou uměl uzel vypořádat, a nástroj musí vědět, že data nebyla načtena. Pokud by nástroj četl data z response přes

blokující metodu `take(…)` a čekal, že mu data dorazí všechny a jedny data nedorazily, dojde k zablokování do aplikace, protože nedostane očekávaná data.

Implementace uzlu se umí sama zotavit z případné chyby při zpracování dat. Aby ale nedošlo k zablokování nástroje, odešle do Response prázdný objekt `JBlackBoxComponent` s názvem souboru, ze kterého měly být data načteny. Nástroj se tak nezablokuje z důvodu, že by data nedostal, ale dostane data prázdná. Na takový případ už musí reagovat nástroj sám.

Chyby spojené s komunikací

Chyby spojené s komunikací vznikají ve chvíli, kdy je jedna strana komunikačního kanálu nedostupná. To může mít ale více příčin. Může dojít v dané části aplikace k takové chybě, že celá aplikace se stává nefunkční nebo se může jen jednat o problémy při komunikaci po síti.

V případě nástroje se může jednat o kompletní selhání nástroje nebo dojde pouze k tomu, že nástroj neobdrží nějaká data, která očekává. Zjištění, že je v systému něco špatně, už je obtížnější než v případě chyby při čtení vstupních dat. Obvykle je chyba detekována pokud je uzel nebo response objekt nedostupný. Implementace obsahuje prostředky pro zotavení systému z chyb, ale jejich použití a reakce na chybu je na integraci nástroje s distribuovanou částí aplikace.

Nedostupný uzel

Pokud je nedostupný uzel, je celý problém ošetřen jednoduše. Fasáda na takový uzly nepošle data ke zpracování a uzel zařadí do seznamu nedostupných uzlu. Ve fasádě je samostatné vlákno `ReconnectNotConnectedNodeTask`, které se opakovaně provádí a jednou za čas zkusí, jestli jsou nedostupné uzly stále nedostupné. Dostupné uzly zařadí mezí aktivní uzly a nedostupné uzly nechá na příští pokus o připojení.

Pokud na právě nedostupný uzel byla dříve poslána data, je nutné ošetřit, aby nástroj věděl, že data nedorazí. Ale to je problém ve chvíli, kdy už nástroj čeká na data v blokujícím režimu. Pro tyto případy je dobré nepoužívat metodu `take(…)`, ale metodu `pull(…)` s časovačem, jak dlouho se bude čekat. Pokud uběhne čas a metoda stále nevrací data, je možné ověřit, že jsou dostupné uzly, kam byla data odeslána. Response obsahuje seznam identifikátorů uzlu, kam byla data odeslána a fasáda má metodu `checkNodesAvailability(…)` pro ověření, že jsou uzly dostupné.

Nedostupný response objekt

Nedostupný response objekt ze strany uzlu může mít dva důvody. Jedním může být kompletní selhání fasády a všech jejich response objektů nebo selhání pouze daného response objektu. V tom případě není způsob, jak problém vyřešit ze strany uzlu. Data se nemají kam odeslat a uzel data může jednoduše zahodit. Dalším důvodem můžou být komunikační chyby.

Aby se zbytečně data nezahazovaly kvůli chvilkovému výpadku na síti, zkouší uzel inicializaci připojení na response vícekrát. Metoda `initResponseConnection(…)` se pokouší více pokusy spojit s objektem response a mezi jednotlivými pokusy je čas, kdy vyčkává. Kolikrát se pokusí uzel připojit je dáno hodnotou proměnné `maxConnectionAttempt` a doba mezi jednotlivými pokusy určuje proměnná `connectionWaitTime`. Základní nastavení je 10 pokusů s odstupem jedné minuty. Obě proměnné jsou v instanci `JClassLoaderTaskImpl`.

Pokud se uzel nespojí ani po všech pokusech, data zahodí, protože detekuje dlouhodobější problém. Nástroj v tomto případě musí mít implementované ošetření přes kontrolu dostupných uzlů a metodu `pull(…)` s časovačem.

6.10 Systémové požadavky implementace

Požadavky implementace na hardwarové zdroje jsou podobné jako u původního nástroje. Uzel akorát si vystačí s pamětí řádově nižší původní nástroje, protože načtená data neudržuje v paměti, ale odesílá fasádě.

Nastavení zásobníku JVM

Nejdůležitější systémový požadavek je na nastavení velikosti zásobníku JVM. Zásobník se nastavuje parametrem `-Xss` při spuštění aplikace. Základní nastavení zásobníku je na 1MB, který je příliš malý pro implementaci. Pro implementaci je dobré nastavit zásobník aspoň na 50MB.

Důvod zvětšení zásobníku

Protože serializace a deserializace dat při odeslání načtených dat probíhá rekurzivně, je nutné upravit velikost zásobníku, aby byla aplikace schopná objektovou strukturu serializovat. Dochází k rekurzivnímu průchodu objektové struktury a postupně se serializují všechny objekty do binární podoby. Velikost zásobníku se dá těžko odhadnout, protože je závislý na velikost a struktuře objektových dat, která se budou

serializovat. Rozhodně je dobré nastavit aplikaci s fasádou a uzlům zásobník aspoň na 50MB a více z původních 1MB.

Nastavení firewallu pro komunikaci

Pro komunikaci mezi komponentami je nutné povolit komunikační porty aplikace.

- **Port 1098, 1099** – základní komunikační porty pro RMI.
- **Port 56565** – základní komunikační port pro komunikaci s komponentou response.

Pokud dojde ke změně parametru `rmiRegistryPort` v komponentách fasáda a uzel, je nutné povolit na firewallu systému nové použité porty.

7 Integrace s compatibility-checker-utils

V rámci implementace praktické části došlo k integraci realizovaného systému do JaCCu. Distribuovaný systém byl integrován do nástroje pro ověření vnitřní kompatibility v projektu `compatibility-checker-utils`.

Nástroje v tomto projektu používají pro načtení vstupních dat implementaci rozhraní `JClassApplicationData`. Implementace tohoto rozhraní `JClassApplicationDataImpl` používá lokálně vytvořené instance Loaderu pro načtení vstupních dat. Komparátor při procesu ověření kompatibility vstupních dat využívá průběžně tuto implementaci a říká si o data, které ho zajímají.

V rámci integrace došlo k vytvoření nové implementace `JClassApplicationRemoteDataImpl`, která pro načtení dat využívá fasádu a uzly. Při vytvoření instance dojde k inicializaci fasády a spojení s dostupnými uzly. Uzly obdrží od fasády vstupní data a začnou vstupní data načítat a doručovat je do objektu response, který vrátí fasáda a je referencován v instanci `JClassApplicationRemoteDataImpl`.

7.1 Načtení vstupních dat

K získání načtených dat z objektu response dojde až ve chvíli, kdy si o data poprvé řekne komparátor. Data se z objektu response přečtou, vytvoří se mapa načtených dat dle vstupních souborů a mapa načtených dat dle názvů tříd. Z těchto dvou map jsou komparátoru data poskytována dle jeho požadavků. Vytvoření map urychluje nalezení požadovaných dat. Čas k vytvoření těchto map je mnohonásobně kratší než když by bylo nutné procházet pokaždé vstupní data a hledat v nich data, která zrovna komparátor požaduje. Kompletní zpracování dat z objektu response a příprava obou map pro vyhledávání provádí metoda `readDataFromResponse(…)`.

7.2 Nastavení JClassApplicationRemoteDataImpl

Nastavení implementace `JClassApplicationRemoteDataImpl` vyžaduje nastavení fasády. Nastavení fasády lze udělat přímo přes jednotlivé parametry konstruktoru nebo je možné použít konfigurační soubor.

Parametry

Parametry konstruktoru třídy `JClassApplicationRemoteDataImpl` jsou shodné s parametry fasády (viz Parametry komponenty v kapitole 6.5).

- **`responseURL`** – parametr shodný s parametrem fasády. URL adresa stroje .
- **`nodeURLs`** – parametr shodný s parametrem fasády. URL adresy s běžícími uzly.

Konfigurační soubor

Konfigurační soubor obsahuje oba výše uvedené parametry. Má podobu běžného properties souboru jazyka Java. Obsahuje dvě položky typu klíč-hodnota, které si implementace ze souboru přečte a použije jako parametry (viz *ukázka 7-1*).

```
response=responseURL
nodes=URL1,URL2,URL3
```

Ukázka 7-1 : **položky konfiguračního souboru pro implementaci JClassApplicationRemoteDataImpl**

Projekt obsahuje `remote-config.properties` soubor jako ukázkový konfigurační soubor. Stejný soubor bude obsahovat výsledná aplikace, která bude používat nástroj z projektu, pokud by měl používat implementaci `JClassApplicationRemoteDataImpl`.

7.3 Konfigurace nástroje compatibility-checker-utils

Nástroje z projektu `compatibility-checker-utils` používají nástroj Google Guice pro konfiguraci a doplnění závislostí. Konfigurace přes nástroj Guice se musí upravit, aby nástroje používali novou implementaci `JClassApplicationRemoteDataImpl` pro rozhraní `JClassApplicationData`.

Pro konfiguraci používá Guice moduly. V modulu je deklarováno, jaká implementace se má použít pro dané rozhraní. Úprava byla provedena v modulu `cz.zcu.kiv.ccu.inter.ApiInterCmpModule`, kde se nahradila původní konfigurace (viz *ukázka 7-2*) za novou konfiguraci používající třídu `JClassApplicationRemoteDataImpl` (viz *ukázka 7-3*). [14]

```
install(new FactoryModuleBuilder()
.implement(JClassApplicationData.class, JClassApplicationDataImpl.class)
.build(JClassApplicationLoader.class));
```

Ukázka 7-2 : **původní konfigurace modolu ApiInterCmpModule**

```
install(new FactoryModuleBuilder()
    .implement(JClassApplicationData.class, JClassApplicationRemoteDataImpl.class)
    .build(JClassApplicationLoader.class));
bind(String.class).annotatedWith(Names.named(REMOTE_NODE_SETTINGS_FILE_NAMED))
    .toInstance(REMOTE_NODE_SETTINGS_FILE_PATH);
```

Ukázka 7-3 : **nová konfigurace modulu pro implementaci JClassApplicationRemoteDataImpl**

Nová konfigurace použije třídu `JClassApplicationRemoteDataImpl` jako implementaci rozhraní `JClassApplicationData` a konfigurační soubor předá jako parametr konstruktoru.

8 Sestavení a spuštění

Pro překlad programu je nutné mít na počítači nainstalovaný sestavovací nástroj Apache Maven. Jelikož je tento nástroj kompletně vytvořen v programovacím jazyce Java, je nutné mít na počítači nainstalované její vývojové prostředí v minimální verzi 1.6.

Sestavení

Pro sestavení a používání aplikace je nutné sestavit samostatnou aplikaci, která provede spuštění komponenty uzel a provést integraci fasády do nástroje nebo použít nástroj, který už integraci fasády obsahuje.

- Uzel – projekt `javatypes-remote` obsahuje hlavní spouštěcí třídu `App`, která umožňuje spustit instanci uzlu. Po sestavení projektu příkazem `mvn package` vznikne spustitelný soubor `javatypes-remote-VERZE.jar`, který lze spustit.
- Fasáda – fasádu je nutné integrovat do nástroje nebo použít už integrovanou část nástroje. V projektu `compatibility-checker-utils` byla provedena integrace (viz kapitola 7) a pokud se z daného projektu použije nástroj `ApiInterCompatibilityChecker` s integrovanou fasádou. Pak je nutné vytvořit spouštěcí třídu pro spuštění nástroje. Sestavit projekt `compatibility-checker-utils` s integrací fasády a příslušnou aplikaci, která bude používat nástroj z projektu. Měl by být vytvořen spustitelný soubor.

Spuštění

Pro spuštění je nutné mít sestavenou aplikace s uzlem a s nástrojem, který bude obsahovat integrovanou fasádu. Aplikace s integrovanou fasádou by měla mít konfiguraci s informacemi o dostupných uzlech. V případě použití existující integrace (viz kapitola 7), je nutné využít konfigurační soubor `remote-config.properties`, který by měla obsahovat aplikace, která bude nástroj spouštět.

Spuštění uzlu se provádí příkazem „`java -jar soubor.jar parametr1 …`„, a je nutné při spuštění přidat parametry pro nastavení použité operační paměti `-Xms` a zásobníku `-Xss`. Pokud jsme použili sestavení projektu `javatypes-remote` pro sestavení uzlu, parametry spuštění uzlu se vypíšou s nápovědou při spuštění bez parametru.

```
java -Xmx2048m -Xss256m -jar javatypes-remote-1.0.5-SNAPSHOT.jar node1 1 localhost
```

Ukázka 8-1 : **Příklad spuštění sestavené aplikace uzlu**

Spuštění fasády určuje způsob integrace fasády s aplikací a nástrojem. Pokud máme integrovanou fasádu do nástroje a pro nástroj existuje spouštěcí třída, spustí se nástroj spouštěcí třídou. Před spuštěním aplikace bude nutné zajistit nastavení parametrů fasády (viz kapitola 6.10) a správnou konfiguraci integrace (viz kapitola 7.2).

Příkladem integrace, sestavení a spuštění je Demo aplikace použitá pro testování (viz Příloha A - Demo Aplikace).

9 Testování

Implementace byla podrobena testování pro ověření funkčnosti. Při testování byly sledovány také výkonnostní charakteristiky, aby bylo možné ověřit výhody a nevýhody realizované distribuované podoby nástroje pro použití v cloudu.

9.1 Testovací prostředí

Testování probíhalo na více testovacích prostředích, které bylo většinou tvořeno skupinou počítačů. Dostupná prostředí také omezovala velikost testovacích úloh. Pro testování bylo také využíváno cloudových služeb Amazon EC2 v rozsahu služeb zdarma.

- Amazon EC2 – pro testování bylo použito pět instancí typu T2, jedna instance obsahovala běžící fasádu a čtyři instance fungovaly jako její uzly pro načítání vstupních dat. Operační systém – Amazon Linux. Testování chování implementace v prostředí cloudu.
- Osobní notebook - Intel Core™ i3-2310M 2.10GHz, 2jádra, celkem 4 vlákna, 8GB RAM, Win7, vývoj implementace a základní testování
- Osobní notebook - Intel Core™ i7-4700MQ 2.40GHz, 4jádra, celkem 8 vláken, 16 GB RAM, Win8, výkonnostní testování
- Skupina PC ověřila výslednou implementaci a integraci, finální testování.
    - Notebook - Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz 8GB RAM
    - PC1 - Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz 64GB RAM
    - PC2 - Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz 8GB RAM

9.2 Výkonnostní testování

Jedním z bodů zadání práce je sledování výkonnostních charakteristik. Je ale nutné provést testování na dostatečně velkém testovacím scénáři, který umožní sledování výkonu implementace. Také je nutné si uvědomit, že výkonnostní testování nelze provádět bez znalostí o fungování JVM.

Základní problematika výkonnostního testování

Java je jazyk interpretovaný, při spuštění je bytecode prováděn interpretem v JVM. Interpret se také stará optimalizace kódu, který provádí. Optimalizace se provádí až při spuštění a Java provádí optimalizace i za běhu aplikace, aby zvýšila svůj výkon.

Celý proces optimalizací má vliv na výkon aplikace a je zbytečné měřit výkon kódu, který nebyl ještě optimalizován. Výkon takové aplikace bude několikanásobně horší než optimalizovaný kód. [15]

Další problém při testování výkonu je intepretace bytecodu. Pokud budeme provádět testování nějakého bloku kódu a celý blok kódu proběhne a jeho výsledek je pouze v lokálních proměnných a žádným způsobem nedochází k ovlivnění instance nebo třídy, může JVM vyhodnotit tento blok jako nepotřebný a úplně ho odstraní z prováděného kódu. Z výsledků bude pak vidět, že kód je vysoce výkonný, ale přitom se daný kód vůbec neprovedl nebo se provedl jen párkrát a po té byl odstraněn interpretem. [15]

Při výkonnostním testování implementace jsme provedli měření hodnot až na optimalizovaném kódu. Obvykle se celý testovací scénář pustil aspoň dvakrát, aby JVM provedlo optimalizace pro druhý běh testu. Problému s odstraňováním nepotřebných bloků kódu jsme vyvarovali tak, že testy obvykle pracovaly s daty, které vznikly v měřených úsecích kódu. JVM tak nemohlo data vzniklá v daných blocích vyhodnotit jako nedůležitá a celý blok nemohl být odstraněn, ale musel být správně proveden.

9.3 Testování před implementaci

Před samotnou implementací došlo k testování technologie RMI a nativní serializace objektů jazyka Java. Testování mělo ověřit už v počátku, že zvolené technologie nebudou mít velký dopad na výkon a celá implementace se díky tomu stane nepoužitelnou.

Testování Remote method Invocation

Výkonnostní test měl ověřit efektivnost technologie pro přenos dat v implementaci.

První test ověřil rychlost přenosu vstupních dat (.class, .jar) v podobě bytového pole. Když uzel rozesílá vstupní data ke zpracování uzlům, dochází k přenosu souboru v podobě bytového pole. Pokud by byl problém poslat běžně velká vstupní data v této podobě, bylo by nutné implementovat jiný způsob přenosu souboru na uzel. Při testu docházelo k posílání různých bytových polí s danou velikostí a byla měřena přibližná doba doručení dat. Výsledky měření zobrazuje *tabulka 2*. Naměřené hodnoty jsou v milisekundách.

Data	Dva Amazon servery Linux server/klient (milisekundy)										Průměr
10MB	928	133	128	124	130	123	135	142	131	139	132
20MB	318	264	274	259	267	258	270	285	259	260	265
50MB	714	689	704	690	697	691	820	713	696	688	696
100MB	1475	1470	1414	1416	1410	1458	1509	1572	1416	1414	1437

Tabulka 2 : **Naměřená data přenosu binárních dat s využitím RMI**

Data byla měřena v prostředí cloudových služeb Amazon, protože je to předpokládaná podoba prostředí, kde bude implementace používána. Přenos dat se dle měřených časů měnil pouze v závislosti na velikost posílaných dat. Po převedení naměřených časů z milisekund do sekund vychází přenosová rychlost na průměrných 70MB za sekundu. Taková přenosová rychlost je dostačující pro přenos datových souborů i v případě, že by bylo nutné přenášet soubory v řádu stovek megabytů. Z opakovaného měření hodnot je zřejmé, že rychlost odesílání dat je stabilní. Naměřená data potvrdili použití RMI pro přenos binárních dat mezi uzlem a fasádou.

Druhým testem se ověřilo odeslání načtených dat už v objektové podobě, která jsou tvořena instancemi tříd `JClass`. Při odeslání dat v objektové podobě dojde k serializaci dat do binární podoby. Data se přenesou na druhou stranu komunikačního kanálu a provede se deserializace binárních dat zpět do objektové podoby. Načtená data mají podobu instancí objektů v kolekcích. Pro měření byly vybrány dva vstupní soubory s různou velikostí, aby se ověřil výkon přenosu menších a větších dat. Velikost dat je pouze orientační. Instance třídy `JClass` obsahuje další instance jiných objektů. Malá data obsahovaly přibližně 1300 instací `JClass`, velká data měla 3000 instancí `JClass`. Výsledky měření zobrazuje *tabulka 3*. Naměřené hodnoty jsou v milisekundách.

Data	Dva Amazon servery Linux server/klient (milisekundy)										Průměr
malá	1899	1073	737	723	741	781	740	710	793	651	740
velká	2587	1951	2170	1899	2242	1930	2284	2051	1928	2073	2062

Tabulka 3 : **Naměřená data přenosu objektových dat s využitím RMI**

Průměrný čas pro odeslání malých dat je 740milisekund a průměrný čas odeslání velkých dat se pohybuje okolo 2000milisekund. Přibližný počet objektů odeslaných za sekundu je přibližně 1600. Z naměřených dat lze usuzovat, že čas odeslání objektových dat poroste lineárně v závislosti na velikost objektových dat, ale růst času by mohl být i horší než lineární. Předchozí test zjistil, že průměrná rychlost odeslání binárních dat je 70MB za sekundu. Z výsledků tohoto a předchozího testu lze usuzovat,

že buď dochází k přenosu velkých objemů dat, nebo má na odeslání dat velký vliv serializace dat do binární podoby.

Výkon serializace při převodu objektových dat do binárních je předmětem dalšího testování.

Testování Serializace

Serializace dat je nutná v případě, že dochází k přenosu objektových dat přes RMI. Testování serializace mělo odhalit dopad serializace a následné deserializace dat při posílání objektových dat a zároveň došlo ke zjištění, jak velký objem dat je posílán. Tím by měl být odhalen důvod velkého zpoždění při posílání objektových dat.

Sada testovacích dat

Pro testování byly vybrány 4 vstupní soubory. Každý vstupní soubor je jinak velký, aby se zjistil dopad serializace na různě velká vstupní data. Velikost vstupního souboru nemusí přesně vypovídat o velikost načtených dat.

- Data 1 - `groovy-1.8.3.jar` – velikost 5,2MB, velikost po serializaci 32MB
- Data 2 - `ojdbc6-11.2.0.3.jar` – velikost 2,5MB, velikost po serializaci 22MB
- Data 3 - `mockito-all-1.9.0.jar` – velikost 1,4MB, velikost po serializaci 4MB
- Data 4 - `commons-beanutils-1.8.3.jar` – velikost 0,25MB, velikost po serializaci 1,5MB

Naměřená časy operací s daty

Naměřená data zobrazuje *tabulka 4*. Naměřené hodnoty jsou čas potřebný na provedení dané operace v milisekundách.

Operace	Data 1	Data 2	Data 3	Data 4
Načtení loaderem	2017	1004	188	59
Serializace	2100	1626	269	67
Deserializace	1681	1302	142	7

Tabulka 4 : **Výsledky měření serializace objektových dat nástroje**

Naměřená data (viz *Tabulka 4*) zobrazují čas potřebný pro vykonání operace nad danými daty. Při odeslání objektových dat dojde k serializaci dat do binární podoby, přenosu dat po síti a deserializaci dat zpět do objektové podoby. Pro Data 1 trvá serializace a deserializace dat dohromady 3,8 sekundy. Velikost serializovaných dat je 32MB a z průměrné rychlosti 70MB/s dostaneme rychlost odeslání těchto dat 0,5

sekundy. Ve výsledku trvá odeslání dat 4,3 sekundy ale samotné poslání dat po síti jen 0,5 sekundy a zbytek času je samotná serializace a deserializace. Pro Data 2 vychází doba odeslání na 3,2 sekundy ale samotné odeslání těchto dat po síti jen 0,3 sekundy.

Z výsledků tohoto měření je zřejmé, že delší doba odeslání objektových dat není způsobena velikostí přenášených dat, ale procesem převodu dat objektových do binárních a zpět. Doba serializace a deserializace je dokonce několikanásobně vyšší než samotné poslání dat po síti nebo načtení dat loaderem. Provedení serializace a deserializace objektových dat je v tomto případě proces, který bude způsobovat velké výkonnostní ztráty celé implementace, které částečně sníží paralelní načítání vstupních dat na uzlech.

9.4 Testovaní implementace

Pro testování implementace byly použity data z kolekce projektů označované jako korpus open-source software, která obsahuje velké množství open-source[14] projektů rozdělených do přibližně 400 verzí. Každá verze obsahuje soubory aplikace a použitých knihoven a každá verze tak může sloužit jako testovací úloha pro nástroj JaCC. Nástroj JaCC zpracovává jednotlivé úlohy sekvenčně. Celá sada úloh obsahuje od malých úloh až po ty větší. Velikost úlohy je dána počtem použitých knihoven pro danou úlohu, které musí nástroj zpracovat a porovnat.

Pro zpracování se používají nástroje z `compatibility-checker-utils`, do kterého byla provedena integrace implementace distribuovaného nástroje. Díky tomu bylo možné použít tento projekt pro testování.

Za původní implementaci je označovaná verze nástroje JaCC před provedením změň a rozšíření týkajících se distribuované migrace. Nástroj provádí zpracování úlohy jedním vláknem a k načítání vstupních dat dochází až ve chvíli, kdy jsou potřeba pro porovnání.

Za distribuovanou implementaci je označovaná upravená verze nástroje JaCC pro realizaci distribuované migrace využívající integrovanou fasádu a uzly pro načítání vstupních dat.

[14] Open-source - projekty nebo aplikace dostupné nebo šiřitelné pod licencí, která umožňuje jejich svobodné užívání a rozšiřování. Obvykle obsahují i zdrojové kódy.

Testovací scénář

Při testování došlo ke zpracování vzorku dat projektu původní verzí nástrojů `compatibility-checker-utils` a verzí s integrovanou fasádou. Při testování byl měřen čas dokončení celého vzorku úlohy. Oba testy měly stejný vzorek testovacích dat. Zpracování úloh probíhá sekvenčně úloha za úlohou.

Po provedení dvou testů došlo ještě k testování, kdy se úlohy neprováděli sekvenčně, ale došlo k paralelizaci provádění úloh. Paralelně se prováděly vždy 3 úlohy najednou.

Testovací prostředí

Testovacím prostředím byla skupina PC:

- NB - Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz 8GB RAM
- PC1 - Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz 64GB RAM
- PC2 - Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz 8GB RAM

Na počítačích PC1 a PC2 byly nasazeny uzly, které měly provádět načítání vstupních dat pro fasádu. Nástroj JaCC s integrovanou fasádou byl nasazen na počítač NB. Vytvořený systém tak kopíroval architekturu distribuované implementace (viz *obrázek 4*).

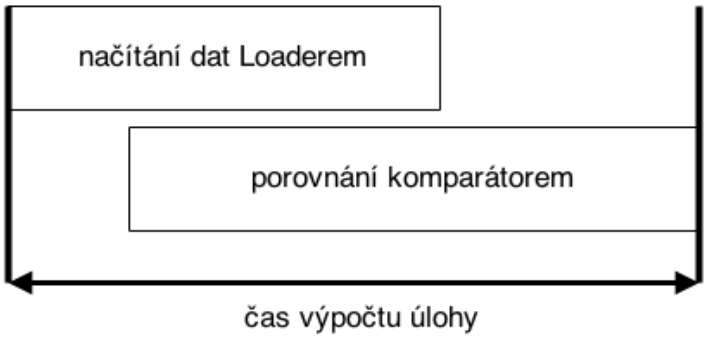
Naměřená data

- Původní implementace – přibližně 40 minut.
- Distribuovaná implementace sekvenční provádění úloh – přibližně 45 minut.
- Distribuovaná implementace paralelní provádění úloh – přibližně 36 minut

Výsledky testování

Původní implementace zvládla úlohu za 40 minut a distribuovaná implementace se sekvenčním prováděním za 45 minut. Z těchto výsledných časů je zřejmé, že původní implementace je rychlejší v sekvenčním zpracování úloh, ale distribuovaná implementace je na tom přibližně stejně. Pokud ale dojde k zadávání úloh paralelně pro distribuovanou implementaci, začne se projevovat částečné urychlení a distribuovaná implementace pak stejnou úlohu zvládne za 36 minut. Výsledky odhalily problém sekvenčního zpracování dat v distribuované implementaci.

Původní implementace nástroje při svém zpracování provádí načítání a porovnání najednou. Komparátor provádí porovnání a data, která potřebuje pro porovnání, se postupně načítají (viz *obrázek 19*).



Obrázek 19 : **Diagram zpracování úlohy původní implementací nástroje**

Implementace s integrovanou fasádou změnila proces zpracování úlohy. Data se nejdříve načtou a jsou připravena pro porovnání, až pak začne provádět komparátor porovnávání (viz *obrázek 20*).

Obrázek 20 : **Diagram zpracování úlohy distribuovanou implementací**

Při sekvenčním zpracování jednotlivých úloh tak dochází k místům, kdy se data nenačítají a část systému stojí a nic nedělá, protože zrovna pracuje komparátor. V případě paralelního zpracování úloh dojde k vyplnění míst, kdy uzel nenačítá data a čeká na dokončení porovnání, načítáním dat pro jiný komparátor.

9.5 Demo

Demo aplikace (viz Příloha A - Demo Aplikace) využívá projekt `compatibility-checker-utils` a provede jednoduché zpracování vstupních dat nástrojem JaCC. Aplikace se skládá z hlavní třídy, která vytvoří instanci nástroje s distribuovanou implementací a provede zpracování dat. Při testování je měřen čas

provedení úlohy od začátku do konce. Cílem tohoto testování je sledovat vliv paralelního načítání vstupních dat s různým počtem paralelně prováděných načítání.

Testování probíhá pouze na jednom stroji. Distribuovaná implementace se dělí o stroj s uzlem, který provádí načítání dat. Obě části běží na stejném stroji, ale protože nejdřív dochází k načtení a až pak k porovnání, není problém s tím, že by se okrádaly obě části o výkon stroje. Měření proběhlo na Osobní notebook - Intel Core™ i7-4700MQ 2.40GHz, 4jádra, celkem 8 vláken, 16 GB RAM

Testovací data tvoří aplikace `Portal of EEG/ERP Experiments` [16]. Aplikace používá skoro 300 různých knihoven. Aplikace je dobrý testovací případ pro nástroj a díky velkému množství knihoven bude aplikace dobrý testovací případ pro testování paralelního načítání vstupních dat.

Testovací vzorky dat

Pro testování byly připraveny dva vzorky dat.

- Data 1 – 278 knihoven do velikosti 2MB.
- Data 2 – 19 knihoven ve velikosti od 2MB do 11MB.

Naměřená data

Test	Data 1	Data 2
Distribuovaná implementace 1 uzel 1 jádro	51 s	49 s
Distribuovaná implementace 1 uzel 2 jádro	43 s	38 s
Distribuovaná implementace 1 uzel 4 jádro	40 s	49 s

Tabulka 5 : **Měření vlivu paralelního načítání dat na dobu zpracování úlohy**

Výsledky měření

Naměřené hodnoty (viz *tabulka 5*) jsou v sekundách. Paralelní načítání dat na uzlech urychluje celé zpracování vstupních dat. V tabulce je vidět, jak se časy sníží v případě, že dojde k paralelnímu načítání vstupních dat na dvou jádrech místo na jednom jádru. Pro Data 1 dochází k urychlení z 51sekund na 43 sekund. Pokud ale použijeme 4 jádra, nemusí už dojít ke zrychlení. V tomto případě dochází k urychlení pouze pro dvě jádra provádějící výpočet. Pokud se použijí 4 jádra pro uzel, nezbyde jádro pro obsluhu operačního systému a klientské aplikace a dojde k sdílení jader pro výpočet a celý výpočet se zpomalí. Je tedy rozumné paralelizovat načítání dat jen na určitý počet jader, aby nedocházelo ke sdílení procesorových jader mezi více procesů.

9.6 Zhodnocení testování

Při testování se ověřovala implementace distribuované části nástroje.

Použité RMI pro odeslání binárních dat je dostatečně výkonné. Průměrná rychlost přenosu dat je 70MB/s.

Serializace při převodu objektových dat do binární pro odeslání dat z uzlu má velký vliv na výkon implementace, ale stále to úplně nebrání v jejím použití.

Výkonnostní testování paralelního zpracování dat na uzlech (viz kapitola 9.5) jasně ukazuje, že dochází k urychlení načítání dat. Paralelní načítání umožní urychlení zpracování úloh nástroje a částečně pokryje ztráty výkonu kvůli serializaci.

Testování původní a distribuované implementace na Studii kompatibility přineslo srovnání obou implementací při zpracování velkého množství úloh. Původní implementace dokončila testovací úlohu za 40 minut a distribuovaná implementace úlohu splnila za 45 minut. Oba časy jsou pouze orientační, ale i tak se dá říci, že implementace jsou při zpracování sekvenčních dat přibližně stejně výkonné. Pokud ale distribuovaná implementace dostává úlohy zadané paralelně, dojde k lepšímu využití dostupných uzlů a projeví se více paralelní načítání vstupních dat. To vede k urychlení distribuované aplikace a dosažení vyššího výkonu.

10 Návrhy vylepšení a rozšíření

10.1 MergeLoader

V rámci praktické části se nestihl implementovat MergeLoader, který měl nahradit chybějící rodičovské loadery (viz kapitola 5.2). Při integraci s nástrojem `compatibility-checker-utils` došlo k částečnému vyřešení chybějící komponenty MergeLoader. Uzly po načtení vstupních dat provedou filtraci načtených dat. Při filtraci jsou z dat odstraněna instance `JClass`, které odkazují na třídy ze základních balíčků jazyka Java. Tím je částečně nahrazen chybějící rodičovský loader pro balíčky Javy.

10.2 Systém zotavení z chyb

Aktuální implementace obsahuje metody a postupy jak detekovat chybu v systému a jak chybu aspoň částečně odstranit (viz kapitola 6.9).

Systém má nástroje pro detekci chyb při komunikaci nebo čtení dat. Nedostatek tohoto systému je v ošetření těchto chyb. Systém je schopen rozpoznat, že se stala chyba, ale většinou z toho vyplývá, že systém neobdrží některá vstupní data načtená. To ovlivní výsledek úlohy.

Rozšíření systému by mělo umožnit systému detekovat přesněji, kde nastala chyba a která data to ovlivní, a provést ošetření této události. Ošetření této události může být další pokus o načtení vstupních dat na jiném dostupném uzlu v případě chyby při komunikaci. Pokud došlo k chybě při čtení dat, je možné nechat data také znovu načíst, ale chyba se může opakovat.

10.3 Vylepšení plánovacího algoritmu

Plánovací algoritmus (viz Algoritmy – Plánování úkolů na jednotlivé uzly v kapitole 6.5) pro rozdělování vstupních dat mezi uzly lze vylepšit o lepší způsob plánování než je rozdělení vstupních dat podle délky front na jednotlivých uzlech. Více podstatná je velikost souboru, ze které může být odhadnuta orientační doba nutná pro načtení vstupních dat. Tím dojde k lepšímu odhadu, jak dlouho bude trvat uzlu frontu zpracovat a jak moc je tedy uzel zaměstnán. K tomu je nutné doimplemenetovat mechanismy pro určení metriky dle obsahu fronty uzlu pro lepší rozdělení vstupních dat na uzly.

10.4 Serializace

Při testování serializace (viz kapitola Testování Serializace v kapitole 9.3) bylo zjištěno, že serializace při komunikaci má značný vliv na době doručení načtených dat zpět nástroji. Tím je ovlivněn výkon nástroje, kolik je schopen zpracovat úloh nebo jak rychle zpracuje úlohu. Výkonnostní ztráty částečně pokrylo paralelní načítání vstupních dat, ale výměna serializace by mohla výkonnostní ztráty úplně odstranit.

V Implementace by mohla být vyměněna nativní serializace objektů do binární podoby při komunikaci. Existují nástroje, které serializaci nahradí za jinou metodu převodu objektu do binární podoby a zpět. Nebo nemusí být výsledná podoba dat binární, ale důležitý požadavek je vyšší výkon při převodu dat než použitá nativní serializace.

Většina nástrojů, které by mohly serializaci nahradit, fungují jako mapper objektů. Použití těchto nástrojů vyžaduje implementaci tříd, které se postarají o převod objektů do binární podoby a zpět. Serializace využívá základních mechanismů jazyka Java pro převod dat. Jedním z takových nástrojů je Kryo [17], který na svých webových stránkách uvádí výkonnostní srovnání proti nativní serializaci. Dostupné jsou kompletní výsledky výkonnostního srovnání i s testy a testovacími daty. Ve výkonnostních testech dosahuje Kryo vyššího výkonu než nativní serializace. Hypoteticky by mohl Kryo být výkonnější než nativní serializaci i nad relativně složitou objektovou strukturou dat JaCCu, ale je nutná další analýza pro potvrzení této hypotézy a další otázkou je jak náročná bude implementace mapperů nástroje Kryo pro objektovou strukturu.

11 Závěr

Cílem této práce bylo analyzovat technickou proveditelnost migrace nástroje do cloudových služeb a využití jejich vlastností. Při analýze byly navrženy dva způsoby umístění nástroje do cloudu: základní migrace celého nástroje a distribuovaná migrace. Základní migrace předpokládá použití celého nástroje bez větších zásahů do implementace. Distribuovaná migrace rozloží nástroj na části a takto rozdělený nástroje je umístěn do cloudu jako distribuovaný výpočetní systém. Rozdělení nástroje na části je implementačně náročné oproti základní migraci.

Základní migrace je implementačně nenáročná a její výkon a chování budou srovnatelné s chováním a výkonem původní implementace nástroje běžícím na lokálním počítači. Testování v kapitole 9.4 ukázalo, že v případě sekvenčního zpracování velkého množství úloh bude výkonnější než distribuovaná implementace. Původní implementace zvládla úlohu dokončit o 5 minut dříve při sekvenčním zpracování. Původní implementace není vhodná pro paralelní zpracování úloh, běžící úloha provádí načítání dat během procesu porovnání a načítání dat je výkonově náročná operace. Paralelně běžící úlohy na stejné instanci by vyžadovaly výkon stroje pro načtení a došlo by k soupeření o zdroje a zpoždění úlohy. Paralelní úlohy by tak musely běžet odděleně na různých instancích. Škálování cloudových služeb bude mít podobu spuštění další instance pro každou úlohu, která má být provedena paralelně s dalšími úlohami.

Distribuovaná migrace byla obsahem praktické části práce a pro její realizaci vznikla distribuovaná implementace nástroje. Při testování v kapitole 9.4 bylo zjištěno, že distribuovaná implementace je výkonnější v případě paralelního zpracování úloh, než při sekvenčním, protože dochází k lepšímu využití hardwaru. Sekvenční zpracování úloh trvalo přibližně 45 minut, zatímco paralelní bylo dokončeno za 36 minut a tím je o čtvrtinu rychlejší. V době porovnání načtených dat již může nástroj provádět načtení vstupních dat jiné úlohy a nebude docházet k ovlivnění výkonu, protože načítání vstupních dat se provádí na jiné instanci než proces porovnání. To umožní provádět paralelní úlohy na stejné instanci a bude nutné pouze zajistit dostatek operační paměti pro udržení načtených dat.

Velký vliv na výkon distribuované implementace může mít serializace dat při jejich odeslání fasádě. Při testování (kapitola 9.3) na testovacích datech bylo zjištěno, že doba provedení serializace může být až několikanásobně delší než jen odeslání binárních dat

po síti. Serializace tímto může způsobovat výkonnostní ztráty, ale paralelní zpracování vstupních dat tuto ztrátu částečně nebo zcela vyrovná. Paralelizace tuto ztrátu nevyrovná v případě, kdy paralelně načtená data s přidanou dobou serializace a komunikace překročí dobu sekvenčního načtení vstupních dat původní implementace. Obecně nelze určit vhodnou velikost úlohy pro distribuovanou migraci s paralelním zpracováním vstupních dat, protože velikost vstupních dat neurčuje dobu načtení vstupních dat nebo jejich odeslání fasádě. To závisí až na struktuře načtených objektových dat.

Vhodný způsob migrace závisí na způsobu používání nástroje. Pokud by měl být nástroj umístěn v cloudu a používán pro sekvenční zpracování dávek úloh, bude základní migrace dosahovat vyššího výkonu a realizace migrace je jednodušší než v případě distribuované migrace. Pokud by měl být nástroj dostupný pro běžné používání více uživatelům, kteří budou do nástroje zadávat úlohy pro zpracování, může docházet k paralelnímu zpracování úloh. Distribuovaná migrace pak bude dosahovat vyššího výkonu a bude lépe využívat dostupný hardware nástroje.

Při realizaci distribuované migrace s použitím implementace z praktické části práce je nutné počítat s dalším vývojem. V implementaci nebyla vytvořena komponenta MergeLoader a bez této komponenty je ovlivněn výsledek zpracované úlohy nástrojem. Komponentu je nutné do systému doplnit a otestovat, že nástroj zpracovává data stejně jako původní implementace. Dalším rozšířením může být vylepšení systému zotavení z chyb v systému nebo náhrada serializace za výkonnější ekvivalent (viz kapitola 10.4).

Seznam zkratek

- **RMI** – Remote Method Invocation
- **JVM** – Java Virtual Machine
- **XML** – Extensible Markup Language
- **JSON** – JavaScript Object Notation
- **URL** – Uniform resource locator
- **http** – Hypertext Transfer Protocol
- **IP** – Internet protocol
- **EC2** – Amazon Elastic Cloud Computing
- **SaaS** – Software as a service
- **PaaS** – Platform as a service
- **IaaS** – Infrastructure as a service
- **HaaS** – Hardware as a service
- **AMI -** Amazon Machine Image

Literatura

[1] **Kamil Ježek, Lukáš Holý, Antonín Slezáček, Přemek Brada.** Software Components Compatibility Verification Based on Static Byte-Code Analysis. Proceeding of SEAA, IEEE, 2013. [Citace: 24. 4 2015.]

[2] **Toby J. Velte, Rober Elsenpeter, Anthony T. Velte.** *Cloud Computing.* Computer Press, 2011. [Citace: 24. 4 2015.] 978-80-251-3333-0.

[3] **Reese, George.** *Cloud Application Architectures.* O'Reilly Media, Inc., 2009. [Citace: 24. 4 2015.] 978-0-596-15636-7.

[4] **Sosinsky, Barrie.** *Cloud Computing Bible.* Wiley Publishing, Inc, 2011. [Citace: 24. 4 2015.] 978-0-470-90356-8.

[5] **Peter Mell, Timothy Grance.** The NIST Definition of Cloud Computing. *Computer Security Resource Center National Institute of Standards and Technology.* [Online] 2011. [Citace: 12. 4 2015.] http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[6] *Amazon Web Services.* [Online] Amazon Inc., 2015. [Citace: 12. 4 2015.] aws.amazon.com.

[7] **Vibhor Aggarwal, Shubhashis Sengupta, Vibhu Saujanya Sharma, and Aravindan Santharam.** A Scalable Master-Worker Architecture for PaaS Clouds. *Data-Intensive Distributed Systems Laboratory.* [Online] [Citace: 19. 4 2015.] http://datasys.cs.iit.edu/events/MTAGS12/p01.pdf.

[8] All About Sockets. *Oracle Documentation.* [Online] Oracle, 2015. [Citace: 24. 4 2015.] https://docs.oracle.com/javase/tutorial/networking/sockets/.

[9] Web Services. *W3Schools.* [Online] W3C, 2015.[Citace: 24. 4 2015.] http://www.w3schools.com/webservices/.

[10] Java™ Remote Method Invocation API. *Java™ Documentation.* [Online] Oracle, 2014. [Citace 20. 4 2015.] http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html.

[11] Java Remote Method Invocation - Distributed Computing for Java. *Oracle Technology Network.* [Online] Oracle, 2014. [Citace: 20. 4 2015.] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html.

[12] Enterprise JavaBeans Technology. *Oracle Technology Network.* [Online] Oracle, 2015. [Citace: 21. 4 2014.] http://www.oracle.com/technetwork/java/javaee/ejb/index.html.

[13] Executor. *Java SE 7 Documentation.* [Online] Oracle, 2015. [Citace: 24. 4 2015.] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html.

[14] User's Guide. *Google Guice.* [Online] Google, 2015. [Citace: 24. 4 2015.] https://github.com/google/guice/wiki.

[15] **Oaks, Scott.** *Java Performance The Definitive Guide.* O'Reilly Media, Inc, 2014. [Citace: 24. 4 2015.] 978-1-449-35845-7.

[16] **Ježek Petr, Mouček Roman.** Portal of EEG/ERP Experiments. 2010. [Citace: 24. 4 2015.]

[17] Kryo Documentation. *Kryo Google Project.* [Online], 2015. [Citace: 24. 4 20 15.] https://code.google.com/p/kryo/wiki/V1Documentation.

Přílohy

Příloha A - Demo Aplikace

Demo aplikace

Demo aplikace je projekt obsahující pouze spouštěcí třídu. Spouštěcí třída načte vstupní data pro nástroj JaCC, provede proces porovnání a vypíše výsledek. Aplikace byla vytvořena pro testovací účely a velice dobře demonstruje integraci praktické části práce a procesy sestavení a spuštění.

Aplikace

Aplikace je vytvořena jako maven projekt. Mezi závislostmi projektu jsou uvedeny jednotlivé části nástroje. Protože aplikace má využívat nástroj s integrovanou distribuovanou implementací, je nutné při sestavení zajistit, že bude přiložena správná verze nástroje, která bude integraci obsahovat.

Hlavní třída demo aplikace obsahuje spouštěcí metodu (viz *ukázka 1*). Metoda načte vstupní data aplikace, vytvoří instanci nástroje `ApiInterCompatibilityChecker` a spustí proces porovnání vstupních dat metodou `checkInterCompatibility(…)`. Po zpracování vstupních dat je vytisknut výsledek porovnání.

```
private static void main(String[] args) {
       ApiInterCompatibilityChecker<File> apiInterCompatibilityChecker =
       ApiCheckersFactory.getApiInterCompatibilityChecker();

    File libDir = new File("libs");
    File[] libFiles = libDir.listFiles();

    File appDir = new File("app");
    File[] appFiles = appDir.listFiles();

    long start = System.currentTimeMillis();
    ApiInterCompatibilityResult result = apiInterCompatibilityChecker
       .checkInterCompatibility(appFiles, libFiles);
    ApiInterCmpResultTxtPrinter printer = CmpResultPrintersFactory
        .getApiInterTxtPrinter();
    StringPrinterCallback stringPrinterCallback = new StringPrinterCallback();
    printer.print(result, stringPrinterCallback);
    long end = System.currentTimeMillis();
    System.out.println(stringPrinterCallback.getCollectedString());
    System.out.println("Run time " + (end - start));
}
```

Ukázka 1 : **Spouštěcí metoda demo aplikace**

Konfigurace aplikace

Projekt obsahuje ve složce `src/main/resources` konfigurační soubor `remote-config.properties`. Tento soubor slouží pro konfiguraci integrace provedené v kapitole 7. Protože aplikace je předpřipravená pro běh na jednom stroji lokálně. Obě nastavení v souboru jsou stejná a nastavená na hodnotu `localhost`.

Pokud bychom chtěli obě části pustit na různých strojích je nutné nastavení upravit.

- **`response`** – IP adresa nebo doménové jméno stroje, kde poběží aplikace demo.
- **`nodes`** – IP adresa nebo doménové jméno stroje, kde poběží aplikace uzel.

Sestavení aplikace

Pro překlad programu je nutné mít na počítači nainstalovaný sestavovací nástroj Apache Maven. Jelikož je tento nástroj kompletně vytvořen v programovacím jazyce Java, je nutné mít na počítači nainstalované její vývojové prostředí v minimální verzi 1.6.

Pro sestavení je důležité, aby měl maven přístup k repositáři, kde se nachází sestavené knihovny nástroje JaCC. Může se jednat o lokální repositář nebo nějaký dostupný online. Aby byla použita distribuovaná verze nástroje, je nutné, aby v repositáři byla přítomna verze projektu `compatibility-checker-utils` s integrací fasády (viz kapitola 7)

Pomocí příkazové řádky provedeme zavolání příkazu `mvn package` ze složky, kde se nachází soubor `pom.xml` nebo spuštění souboru `build.bat`. Po sestavení projektu je nutné zkopírovat vytvořený soubor `demo.jar` ze složky `demo/target` do stejné složky kde se nachází soubor `runDemo.bat`.

Spuštění

Před samotným spuštěním demo aplikace je nutné ještě upravit nastavení firewallu. Konfigurace firewallu je popsána v kapitole 6.10. Ostatní nastavení systémových požadavků z této kapitoly už obsahují spouštěcí soubory. Pro běh demo aplikace je nastaveno dohromady 5.5GB operační paměti ve spouštěcích souborech a je nutné, aby takové množství bylo volné v operačním systému.

Pro spuštění je nutné nejdřív spustit uzel v podobě souboru `Node.jar`. Uzel je možné spustit souborem `runNode.bat`. Pro změnu počtu paralelního načítání na uzlu stačí

změnit příkaz v souboru `runNode.bat`. V příkazu se změní parametr „2“ na počet, kolik požadujeme paralelně běžících načítání vstupních dat. Základní nastavení spuštění je použití dvou procesorových jader pro načítání vstupních dat.

Po spuštění uzlu souborem runNode.bat je nutné spustit samotnou aplikaci. Aplikace se spouští souborem `runDemo.bat`. Po spuštění se spustí proces nástroje a začnou se porovnávat vstupní data.

Demo aplikace je k práci přiložena už sestavená a připravená ke spuštění. Stačí spustit nejdřív uzel souborem `runNode.bat` a pak spustit demo aplikaci souborem `runDemo.bat`. Demo aplikace načte přiložená vstupní data a zobrazí výsledek.

Změna vstupních dat aplikace

Demo má předem přiložené vstupní data, ale je možné přiložená vstupní data změnit za jinou úlohu. Musí se dodržet umístění a pojmenování vstupních dat. Aplikace, kterou má demo zpracovat se umístí do složky `app`. Knihovny, které testovaná aplikace využívá a které bude nástroj porovnávat, se umístí do složky `lib`, ve které se nachází původní data demo aplikace.