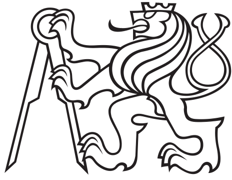
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Centralizované řízení distribuovaných procesů

Bc. Miroslav Bauer

Studijní program: Elektrotechnika a informatika
Obor: Výpočetní technika

Květen 2014
Vedoucí práce: doc. Ing. Petr Fišer, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Miroslav Bauer**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Výpočetní technika

Název tématu: **Centralizované řízení distribuovaných procesů**

Pokyny pro vypracování:

Navrhněte způsob centralizovaného řízení distribuovaných procesů a shromažďování dat. Jedná se o procesy, které jsou vzdáleně spouštěny na různých počítačích (výpočetních serverech), ze zadaných dat generují jiná data. Zdrojová data je nutné procesům rozesílat (uvažujte i alternativu, kdy si procesy o data mohou žádat) a vygenerovaná data shromažďovat v databázi.

Cílem této práce je:
1) Provést analýzu možností.
2) Navrhnout a naimplementovat "back-end" tohoto systému, tj. serverovou a klientskou část aplikace, která bude dle požadavků zadaných zvenku (z databáze, příp. nadřazené aplikace) spravovat rozesílání a sběr dat.
3) Vytvořit databázi, ve které budou uložena veškerá zdrojová data a data aplikací vytvořená. Zde se jedná o kolektivní práci společně s tvůrcem předního konce (uživatelské rozhraní systému).

Platforma: Linux, případně i Windows.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Petr Fišer, Ph.D.

Platnost zadání: do konce letního semestru 2013/2014

doc. Ing. Miroslav Šnorek, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 2. 2013

Poděkování

Rád bych tímto poděkoval vedoucímu práce doc. Ing. Petru Fišerovi, PhD. za vedení práce. Dále pak Pavlu Novákovi za spolupráci při návrhu databáze a za vytvoření uživatelského rozhraní pro tuto práci. V neposlední řadě děkuji také všem blízkým a rodině za veškerou podporu a především trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 5. 5. 2014

. . . .

Abstrakt / Abstract

Tato práce si klade za cíl analyzovat současné možnosti centralizovaného řízení a distribuce výpočetních úloh na výpočetní zdroje. Cílem je návrh a implementace vybraného řešení. Toto řešení by mělo sloužit jako vrstva spojující jednotlivé výpočetní zdroje s databází a grafickým rozhraním (jejichž implementace je součástí související bakalářské práce), ve kterém jsou výpočetní úlohy zadávány. Výsledný systém bude použit ke zpracování syntéz diskrétních logických obvodů na různých výpočetních zdrojích.

Klíčová slova: distribuované výpočty, clustery, gridy, správa úloh

The goal of this thesis is to analyze current available solutions for centralized control and distribution of computational jobs onto compute nodes. In the next step, the goal is to design and implement one of possible solutions. Resulting software is intended to function as a middleware between a GUI (which is subject to corresponding bachelor's thesis), a database (where data about jobs are stored) and a compute resources. Application will be used specifically for running discrete circuit synthesis jobs on various compute resources.

Keywords: distributed computing, clusters, grids, job management

Title translation: Centralized Control of Distributed Processes

Obsah

1 Úvod . . . 1
1.1 Distribuované systémy . . . 1
1.2 Distribuované výpočty . . . 2
1.3 Členění textu . . . 3
1.4 Vymezení pojmů . . . 4
2 Specifikace cílů a požadavků . . . 5
2.1 Motivace, současný stav . . . 5
2.2 Cíle . . . 6
2.3 Funkční požadavky . . . 7
2.4 Nefunkční požadavky . . . 8
3 Analýza dostupných řešení . . . 10
3.1 Druhy řešení . . . 10
3.1.1 Resource Manager . . . 10
3.1.2 Scheduler . . . 10
3.1.3 Meta-scheduler . . . 10
3.2 BOINC . . . 11
3.2.1 Popis architektury . . . 11
3.2.2 Zhodnocení . . . 12
3.3 PBS . . . 13
3.3.1 Popis architektury . . . 14
3.3.2 Zadání úlohy . . . 16
3.3.3 Zhodnocení . . . 16
3.4 MOAB . . . 17
3.4.1 Popis architektury . . . 17
3.4.2 Zadání úlohy . . . 18
3.4.3 Zhodnocení . . . 18
3.5 GRAM . . . 19
3.5.1 Popis architektury . . . 19
3.5.2 Zadání úlohy . . . 20
3.5.3 Zhodnocení . . . 21
3.6 DIANE/GANGA . . . 21
3.6.1 Popis architektury . . . 22
3.6.2 Zadání úlohy . . . 23
3.6.3 Zhodnocení . . . 24
3.7 SAGA . . . 24
3.7.1 Popis architektury . . . 25
3.7.2 Zadání úlohy . . . 26
3.7.3 Zhodnocení . . . 26
3.8 Výběr výpočetních zdrojů . . . 27
3.8.1 MetaCentrum . . . 27
3.9 Výběr řešení . . . 29
4 Návrh řešení . . . 30
4.1 Návrh komunikačního protokolu . . . 30
4.2 Komunikační middleware . . . 33
4.2.1 ZeroMQ . . . 34
4.3 Diagram nasazení . . . 36
4.4 Diagram komponent . . . 37
4.5 E-R model databáze . . . 38
5 Implementace . . . 40
5.1 Konfigurace . . . 40
5.2 Rozhraní backend-frontend . . . 40
5.3 Scheduler . . . 41
5.4 JobExecutor . . . 43
5.5 ResourceManager . . . 44
5.6 Pomocná infrastruktura . . . 44
5.6.1 Cache souborů . . . 44
5.6.2 Notifikace . . . 44
5.6.3 RPC proxy . . . 44
5.7 Struktura projektu . . . 45
5.8 Testování . . . 45
6 Závěr . . . 47
Literatura . . . 48
A Uživatelská příručka . . . 51
A.1 Instalace . . . 51
A.2 Použití . . . 51
B Obsah přiloženého CD . . . 52

Tabulky

3.1. Hodnocení BOINC 13
3.2. Hodnocení PBS 17
3.3. Hodnocení MOAB 19
3.4. Hodnocení GRAM 21
3.5. Hodnocení DIANE 24
3.6. Hodnocení SAGA 27

Obrázky

1.1. Distribuce úloh na výpočetní uzly v gridu 3
3.1. Architektura BOINC systému 12
3.2. Uživatelské rozhraní BOINC klienta 12
3.3. Architektura PBS 14
3.4. Proces zpracování úlohy v systému MOAB 18
3.5. Architektura systému GRAM 20
3.6. Architektura DIANE/GANGA 22
3.7. Architektura knihovny SAGA 25
3.8. Mapa MetaCentra 28
3.9. Vývoj množství jader procesorů zapojených do MetaCentra 28
4.1. Komunikační vzor Router – Dealer 36
4.2. Diagram nasazení pr servers. 36
4.3. Diagram nasazení pro MetaCentrum 37
4.4. Interakce mezi komponentami aplikace 38
4.5. E-R diagram databáze úloh 39

Kapitola 1
Úvod

Již po mnoho let se ve výpočetním světě mění dřívější trend nárůstu výkonu individuálních strojů a začíná se výrazně upřednostňovat množství před výkonem jednotlivých strojů. Motivací jsou zejména ekonomické důvody. Začíná být výhodnější pořídit větší množství strojů pouze s průměrnou hardwarovou specifikací, než pořizovat jediný stroj s výkonem odpovídajícím výkonu všech slabších strojů dohromady. Takováto seskupení vzájemně propojených a spolupracujících strojů (často označovaných jako výpočetní uzly) jsou známá pod označením *„cluster“* nebo *„grid“*.

Smyslem clusterů a gridů je sdílení a poskytování zdrojů komunitě uživatelů. Snahou je optimalizovat využití těchto zdrojů a poskytnout komunitě zdroje potřebné k řešení náročných a distribuovaných úloh. Sdílenými zdroji bývá nejen výpočetní výkon procesorů a grafických karet, ale i prostor pro ukládání dat nebo také zpravidla drahé softwarové licence. Dále v této práci budou popisovány především stroje poskytující výpočetní výkon. Tyto stroje budou označnovány jako výpočetní uzly.

1.1 Distribuované systémy

Oproti samostatným, byť výkonným serverům, jsou clustery a gridy mnohem lépe škálovatelné, v případě potřeby většího výkonu se do skupiny jednoduše zařadí další výpočetní uzel. Stabilita (dostupnost) celého systému je také mnohem vyšší. V případě libovolného selhání výpočetního uzlu ve skupině jeho funkci rychle zastoupí uzel jiný a uživatel obvykle výpadek ani nepocítí. U samostatných strojů je toto nutno řešit použitím redundantního hardwaru (síťových karet, napájecích zdrojů, procesorů, atd.). Dále narozdíl od samostatných strojů umožňují clustery a gridy dynamické vyrovnávání zátěže (load balancing). V případě, že jeden z uzlů začíná mít příliš mnoho práce, další přicházející požadavky mohou být jednoduše směrovány na méně zatížené uzly. Pokud v jeden okamžik přijde špičková zátěž v podobě mnoha požadavků, je například možné tuto zátěž distribuovat rovnoměrně na všechny nezatížené uzly. Tím dojde k rovnoměrnému rozložení celkové zátěže a k zamezení přetížení konkrétního uzlu.[1]

Stále nejefektivnějším systémem pro běh paralelních úloh jsou superpočítače. Jejich procesory jsou propojeny specializovanou a velmi rychlou vnitřní sítí s nej-

nižším možným zpožděním. Jednotlivé procesory sdílí společnou paměť. Je to však oproti clusterům a gridům ekonomicky nejnákladnější řešení a také nejméně škálovatelné. Clustery představují podobně jako superpočítače homogenní prostředí, provozované typicky v rámci jedné lokality. Jednotlivé výpočetní uzly clusteru jsou propojeny lokální sítí (např. Infiniband nebo Myrinet)[2] a mají obvykle shodnou hardwarovou a softwarovou výbavu. Výpočetní uzly zapojené do gridů jsou naopak mnohem více heterogenní a distribuované do více geografických lokalit. Zdroje do gridu poskytuje často vícero spolupracujících institucí propojených přes internet. Propustnost a zpoždění na takovýchto spojích je pak řádově horší. Stejně tak je mnohem vyšší dynamika celého systému (četnost přidávání nebo ubírání uzlů). Na gridy se dá pohlížet jako na hierarchii složenou z clusterů, superpočítačů a individuálních uzlů.[1]

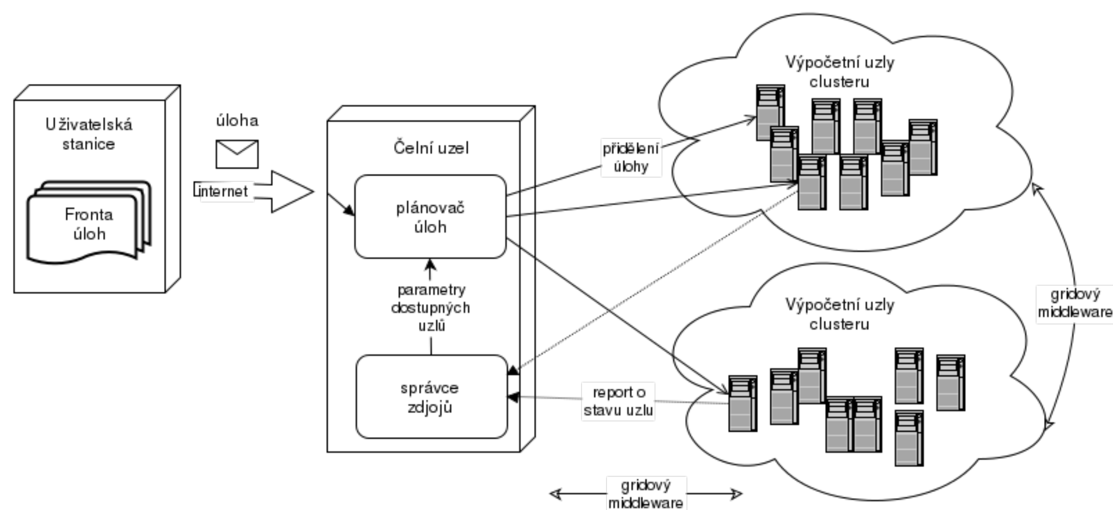
1.2 Distribuované výpočty

Clustery i gridy poskytují vhodné prostředí pro zpracování paralelních, distribuovaných či podle současného populárního označení *„cloudových“* výpočtů (cloud computing). Na superpočítačích paralelní výpočty představují úlohy, jejichž dílčí části jsou v jeden okamžik řešeny na více procesorech sdílících společnou paměť. Případná komunikace mezi jednotlivými částmi úlohy bývá velmi rychlá a realizuje se přes sdílenou paměť. Typicky jde o značně výpočetně náročné úlohy, které jsou úzce vázané na společná vstupní nebo výstupní data.[3] Paralelní úlohy jsou spouštěny v homogenním prostředí, kde se všechny výpočetní uzly nachází v jedné lokalitě a jejichž topologie umožňuje velmi rychlou komunikaci mezi procesory. Distribuované úlohy vycházejí z úloh paralelních. Za distribuovanou úlohu lze označit takovou úlohu, jejíž dílčí části mezi sebou komunikují zasíláním zpráv.[4, 3, 5] Takové úlohy předpokládají běh v distribuovaných systémech, kde má každý výpočetní uzel svou vlastní paměť, fyzická vzdálenost mezi uzly bývá větší a vzájemná komunikace mezi procesory účastnícími se na výpočtu řádově pomalejší. Termín „cloud computing“ [1]) je pak jen populárnější označení pro distribuované výpočty. Obvykle označuje distribuované výpočty za služby nebo aplikace běžící na strojích rozmístěných po celém světě. Služby umožňující počítání v cloudu jsou označnovány jako IaaS (Infrastructure as a Service). Za takovouto službu lze například považovat EC2 od Amazonu.[6] Tyto služby pak uživatelům bývají obvykle dostupné odkudkoliv ze sítě Internet. V rámci této práce se budeme dále zaměřovat pouze na distribuované výpočty.

Aby byly zajištěny vlastnosti popisované v předchozím textu a dále také distribuce úloh na jednotlivé výpočetní uzly, na uzlech gridu musí být nainstalo-

[1]) `http://en.wikipedia.org/wiki/Cloud_computing`

ván specializovaný software. Grid by měl uživateli zajistit naplánování a spuštění jeho úlohy na uzlech nejlépe vyhovujících požadavkům, a to v nejkratším možném čase. Přitom nesmí být příliš omezováni ostatní uživatelé využívající stejné zdroje. Tento úkol má na starosti plánovač úloh. Plánovač je většinou, ne však nutně (v rozsáhlých gridech jsou provozovány celé hierarchie plánovačů), centrální komponenta sídlící na čelních uzlech, ze kterých jsou zadávány úlohy k výpočtu. Ke své činnosti však nezbytně potřebuje mít znalost o dostupných zdrojích a jejich aktuálním stavu nebo využití. Tuto znalost plánovači poskytuje komponenta pro správu, monitoring a získávání či sledování vlastností zdrojů. V jejím případě jde typicky o klient-server systém, kdy na každém uzlu v gridu musí být nainstalována klientská část. Posledním prvkem je pak vrstva, zprostředkovávající zasílání zpráv mezi uzly gridu, označovaná jako komunikační middleware. Middleware zajišťuje integraci různorodých částí gridu do jednoho funkčního celku.[6] Naprostá většina dostupných řešení pro distribuované výpočty v distribuovaném prostředí, jakým jsou gridy, v sobě tyto klíčové komponenty v nějaké podobě zahrnuje.



Obrázek 1.1. Distribuce úloh na výpočetní uzly v gridu.

1.3 Členění textu

Následující kapitola 2 má za úkol seznámit čtenáře s hlavní motivací a cílem této práce. Dále bude zaměřena na podrobnější specifikaci funkčních či nefunkčních požadavků na hledané řešení.

V další, obsáhlejší kapitole 3, bude následovat rešerše v podobě popisu a rozboru vlastností dostupných řešení pro řízení distribuovaných výpočtů. Součástí této kapitoly je vyhodnocení a výběr konkrétního řešení k realizaci v rámci praktické části práce.

Na rešerši navazuje kapitola 4 zabývající se návrhem a modelováním výsledného systému na základě poznatků získaných z rešerše. Po této kapitole následuje část 5 pojednávající o samotné implementaci. Práci pak završuje závěrečné otestování vytvořeného systému a zhodnocení výsledků práce v kapitolách 5.8 a 6, včetně poznatků ohledně jejího budoucího možného rozšíření.

1.4 Vymezení pojmů

Za **výpočetní uzel** budeme označovat komponentu poskytující hardwarové a softwarové prostředky potřebné ke zpracování libovolné výpočetní úlohy. Výpočetní uzel budeme občas označovat také za **výpočetní zdroj**. Výpočetní zdrojem je také grid nebo cluster umožňující výpočty na výpočetních uzlech.

Distribuovaný systém je soustava komponent (výpočetních uzlů), jež jsou vzájemně propojeny sítí a komunikují mezi sebou zasíláním zpráv.[5]

Výpočetní úlohy budeme považovat za **distribuované výpočty**. Distribuovaný výpočet je takový spustitelný program, který je schopen běhu v distribuovaném systému.[5]

Frontendem bude označováno uživatelské rozhraní a databáze s informacemi o úlohách (viz sekce 2.2).

Backendem rozumíme aplikaci vyvíjenou v rámci této práce, jenž běží pod frontend vrstvou (více viz 2.3).

Kapitola 2
Specifikace cílů a požadavků

2.1 Motivace, současný stav

Primární motivací této práce je absence jakékoliv automatizace při zadávání úloh k výpočtu u současného řešení a nedostatečné množství a výkon vlastních výpočetních zdrojů. Popisováno je současné řešení používané výzkumnými pracovníky Katedry číslicového návrhu na FIT ČVUT za účelem provádění syntéz logických obvodů. Tyto syntézy jsou prováděny za účelem optimalizace a následného vyhodnocení charakteristik syntézou vzniklých obvodů. Jednou výpočetní úlohou tedy v rámci této práce bude provedení jedné syntézy konkrétního obvodu, kde výstupem z takovéto úlohy bývá nejčastěji popis dalšího logického obvodu vzniklého syntézou. Syntézou se v tomto případě zjednodušeně myslí parametrizovaný skript, který na vstupní popis obvodu aplikuje definovanou sadu transformací, jehož cílem je nejčastěji vytvoření obvodu se stejným chováním, ale lepšími vlastnostmi.

Úlohy jsou v současnosti spouštěny ručně, nebo pomocí jednoduchých skriptů, za využití SSH[1]). Při zadávání výpočetní úlohy je potřeba předem určit, na kterém stroji se úloha má provést.

Dalším zjevným nedostatkem je nutnost přesunu dat, které úloha ke svému výpočtu potřebuje či produkuje. Veškerá data, včetně úloh, jsou trvale umístěna na uživatelském stroji. Před zahájením samotného výpočtu je potřebné provést *„stage-in“*, tj. proces nakopírování spustitelného souboru úlohy a všech potřebných vstupních souborů (včetně podpůrných skriptů, které může úloha při svém běhu potřebovat) na stroj, na kterém má být úloha spuštěna. Po skončení výpočtu je naopak potřeba stáhnout výsledky (výstupní soubory, stdout a případně stderr[2])) zpátky na uživatelský stroj a uklidit z výpočetního uzlu soubory týkající se úlohy. Tento proces budeme označovat *„stage-out“*. Provádění těchto dvou procesů u každé úlohy ručně je značně pracné. Ošetření přenosu dat v rámci skriptu zase není dostatečně flexibilní a přehledné pokud jsou úlohy spouštěny na různých uzlech s různým prostředím. Takový skript by navíc neměl možnost

[1]) Secure Shell, viz `http://www.openssh.com/`.

[2]) Standardní a chybový výstup úlohy, viz `http://en.wikipedia.org/wiki/Standard_streams`.

spolehlivě zjistit, jaké soubory úloha potřebuje, jaké produkuje a odkud kam je přenést.

Tím se dostáváme k dalšímu problému. Tuto znalost o požadavcích a očekávaných výstupech úloh má v současnosti pouze uživatel a leží na něm břímě v podobě udržování této znalosti stále aktuální.

Posledním ze zásadních omezujících faktorů, které je potřeba řešit, je nedostatek strojů použitelných ke zpracování úloh. Toto vede, spolu s problémy popsanými výše, zpravidla k sekvenčnímu spouštění úloh jedné po druhé, což není příliš časově (a z hlediska využití zdrojů) efektivní.

2.2 Cíle

Tato diplomová práce má za úkol najít řešení, které by redukovalo, a v ideálním případě zcela eliminovalo, problémy popsané v části 2.1.

Prvním z vytyčených cílů je podrobná analýza současných možností řešení distribuce výpočetních úloh na výpočetní uzly. Tato řešení by měla umožňovat centralizovanou správu a monitoring stavu úloh i dostupných výpočetních uzlů. Dále by měla počítat nejen s použitím samostatných uzlů, ale také s využitím zdrojů gridů. Cílem je najít takové řešení, které bude umět využít co nejvíce typů výpočetních zdrojů, a bude splňovat další požadavky definované v podkapitolách 2.3 a 2.4.

Hlavním cílem této práce je návrh a implementace vlastní backend[1]) aplikace, popřípadě využití a přizpůsobení některého z již dostupných řešení pro centralizované řízení distribuovaných úloh. Při výběru, návrhu a realizaci se bude hledět především na multiplatformnost a modularitu (rozšiřitelnost) řešení. Velmi důležité je zbavit uživatele nutnosti přovádět zdlouhavé a často se opakující kroky a celý proces zpracování úloh co nejvíce automatizovat a zjednodušit.

Cílem celého takto vzniklého systému tedy je, aby uživatel nemusel již řešit, kam se má úloha poslat, na kterém konkrétním stroji se má spustit nebo řešit přesuny dat tam a zpět a kontrolování běhu úlohy. V uživatelském rozhraní pouze vybere úlohu ke zpracování a počká na výsledky, které jsou zároveň ukládány do databáze za účelem dalšího použití k výpočtům.

K dosažení tohoto cíle bude potřeba strukturovat a přesunout veškeré znalosti (viz 2.1) a data týkající se obvodů do databáze. To bohužel vyžaduje jistou počáteční časovou investici. Naplnění takové databáze daty by bylo kvůli různorodosti obvodů a syntézních skriptů velmi obtížné nějakým způsobem automatizovat (zjistit potřebné parametry a závislosti ze skriptů a někdy i binárních programů). Ve výsledku však tato investice mnohem více času ušetří při spouštění úloh a značně pomůže plánovači úloh při plánování na výpočetní zdroje.

[1]) Část systému poskytující služby pro uživatelské rozhraní – frontend.

Databáze je tvořena ve spolupráci s navazující bakalářskou prací Pavla Nováka[1]) (jejíž součástí je i nadstavba v podobě uživatelského rozhraní). Tato diplomová práce se bude dotýkat především návrhu schématu databáze, jenž vznikl v kooperaci s autorem bakalářské práce.

I přesto, že například databáze a uživatelské rozhraní navázané na backend jsou řešení šitá na míru pro specifické použití, výsledná aplikace vyvíjená v rámci této práce by měla být pokud možno spíše generická. Mělo by být umožněno použití aplikace i pro jiný než výše popisovaný účel provádění syntéz logických obvodů. Aplikace by také neměla být úzce svázaná s nadřazeným uživatelským rozhraním ani s databází.

2.3 Funkční požadavky

Analýzou funkčních požadavků definujeme funkcionalitu, kterou by aplikace (backend) měla poskytovat svým klientům. Těmi jsou uživatelé interagující s uživatelským rozhraním (frontend) a nepřímo i s databází. Funkční požadavky tak reflektují kontrakt mezi backendem a frontendem.

- Backend přijímá úlohy zaslané klienty ke zpracování.
- Na vyžádání poskytuje seznam výpočetních zdrojů, které má k dispozici.
- Sleduje průběh úloh a informuje klienta o změnách stavu úlohy.
- Informuje klienta, pokud během zpracování výpočtu nastane v libovolný okamžik chyba.
- Po dokončení výpočtu zašle klientovi výsledky v jím specifikovaném formátu.

Realizovaná aplikace má klientovi (frontendu) k ní připojenému umožnit distribuovat z jednoho centrálního bodu úlohy k výpočtu na nakonfigurované či zjištěné výpočetní zdroje. Jejím závazkem vůči klientovi je zajištění průběžného reportování průběhu úlohy a zasílání výsledků. Formát, v jakém mají být výsledky zaslány, je určen klientem při zadání úlohy aplikaci. Tento formát definuje, k jakým parametrům přiřadit jaké hodnoty z výsledků. Výše uvedené požadavky tak naznačují potřebu zvolit nebo nadefinovat vhodný komunikační protokol mezi frontendem a backendem. Dále následují čistě interní funkční požadavky na backend, jež jsou zcela nezávislé na frontendu:

- Backend načítá všechny dostupné zdroje z konfiguračního souboru nebo je schopen tyto zdroje dynamicky detekovat.
- Zajišťuje výběr vhodného výpočetního zdroje pro zadanou úlohu.
- Sleduje stav a dostupnost nakonfigurovaných zdrojů.
- Provádí sběr metrik vyjadřujících výkon a zatížení zdrojů.

---

[1]) novakp76@fit.cvut.cz, zadání práce: `https://portal.fit.cvut.cz/zp/?tzp=417806635305`.

- Naplánuje úlohu na vybraný zdroj a zajistí její spuštění.
- Plánovač úloh zohledňuje metriky při výběru zdroje pro úlohu.
- Zajistí korektní obousměrný přenos všech souborů a dat spojených s úlohou.
- Měl by se pokoušet vypořádat s náhodnými selháními výpočtů a přeposílat v takových případech výpočet na jiný zdroj.

Tyto požadavky z velké části vycházejí z modelu naznačeného v závěru kapitoly 1 a pak také ze snahy vyřešit problémy ze sekce 2.1. Navrhované řešení by mělo zastávat jak funkci plánovače úloh, tak i správce zdrojů. Na stroj, na kterém tato aplikace poběží, se tak dá nahlížet jako na čelní uzel (viz obr. 1.1) pomyslného gridu.

Pro správce zdrojů se nabízí dva základní přístupy, jak se dozvědět o spravovaných zdrojích. Prvním, jednodušším, je nakonfigurování zdrojů, které se mají použít, uživatelem ve speciálním konfiguračím souboru. To obnáší zadání adresy zdroje, metody přístupu na zdroj a přístupových údajů a nakonec zadání několika jeho vlastností (např. seznamu nainstalovaných nástrojů, operačního systému, systému pro zadávání úloh a dalších). Druhý model předpokládá, že se výpočetní zdroje zaregistrují u správce zdrojů z vlastní iniciativy. Tato druhá varianta již vyžaduje, aby byl na zdrojích nainstalován speciální software, který toto umožní.

2.4 Nefunkční požadavky

Analýzou nefunkčních požadavků budou vymezeny základní požadavky na použité softwarové komponenty a vlastnosti aplikace.

- Formátem pro přenos zpráv mezi fronendem a backendem je JSON[1]).
- Řešení pokud možno multiplatformní (podpora Linuxu, případně Windows a dalších platforem).
- Podpora co nejvíce typů výpočetních zdrojů (individuální stroje, clustery, gridy, různé typy systémů pro správu úloh na gridech).
- Co nejkratší doba zpracování úlohy (doba uplynulá od příchodu zadání po zaslání výsledků klientovi).
- Co nejvyšší propustnost (množství úloh, které je možno odbavit za časovou jednotku).
- Modifikovatelnost, rozšiřitelnost, otevřený zdrojový kód

Vzhledem k tomu, že frontend je implementován v jazyce PHP a u backendu se očekává implementace v jiných jazycích, k výměně dat byl zvolen univerzální

[1]) http://www.json.org/json-cz.html

a velmi rozšířený textový formát JSON. Tento formát je dnes podporován v drtivé většině současných jazyků. Je požadováno, aby řešení fungovalo na platformě Linux, v ideálním případě by však nemělo být omezeno pouze na tuto platformu. Požadovanou modifikovatelností a rozšiřitelností je myšlena možnost snadného přizpůsobení pro jiný účel použití než je popisován v sekci 2.1 a snadného přidání či změny stávající funkcionality.

Kapitola 3
Analýza dostupných řešení

V této kapitole je obsažen podrobnější přehled a analýza některých používaných řešení pro distribuované výpočty včetně jejich krátkého zhodnocení. Důraz při hodnocení řešení je vždy kladen na to, zda a do jaké míry dané řešení vyhovuje požadavkům specifikovaným v sekcích 2.3 a 2.4.

Pro účely této práce byla vybrána a zohledňována pouze nekomerční řešení (vzhledem k poměrně vysoké finanční náročnosti řešení komerčních). Komerční řešení navíc obvykle obtížně splňují požadavky na modifikovatelnost a otevřený zdrojový kód, nebo tyto požadavky nesplňují vůbec. V závěrečném zhodnocení kapitoly je následně určeno případné řešení a další postup práce.

3.1 Druhy řešení

Řešení dané problematiky jsou několika základních druhů, mohou se vzájemně kombinovat a doplňovat za účelem splnění požadavků na aplikaci.

3.1.1 Resource Manager

Řešení tohoto typu typicky poskytuje možnosti, jak spravovat, monitorovat či detekovat výpočetní uzly v systému. Dále také umí alokovat zdroje na uzlech pro příchozí úlohy dle jejich požadavků. Někdy bývá označováno jako **LRM** (Local Resource Manager).[7] Bývá obvykle nasazováno pro lokální správu uzlů a spouštění úloh v rámci clusteru. Spouštění úloh však potřebuje spolupracovat s často externí komponentou plánovače.

3.1.2 Scheduler

Řešení, které je čistě plánovačem, zase vyžaduje spolupráci se správcem zdrojů. Plánovač vybírá výpočetní uzly vhodné pro běh úlohy a k tomu potřebuje mít pokud možno úplnou znalost systému a jeho stavu. Tuto znalost získává od správce zdrojů. Taktéž je nasazován pouze v lokální doméně clusteru.

3.1.3 Meta-scheduler

Narozdíl od klasických plánovačů, tento druh plánovače neprovádí přímý výběr výpočetních uzlů pro úlohu. Namísto toho směruje úlohu na některý z nižších

plánovačů v hierarchii (ty již provádí výběr vhodných uzlů). Toto řešení se používá k integraci více domén (clusterů) do jediné.

Meta-schedulery v kombinaci se správci zdrojů abstrahují specifická rozhraní clusterů do jednotného rozhraní a umožňují tak vytvořit grid. Uživatelé pak přistupují ke všem zdrojům gridu pouze z jednoho centrálního bodu a přes jedno jediné rozhraní.

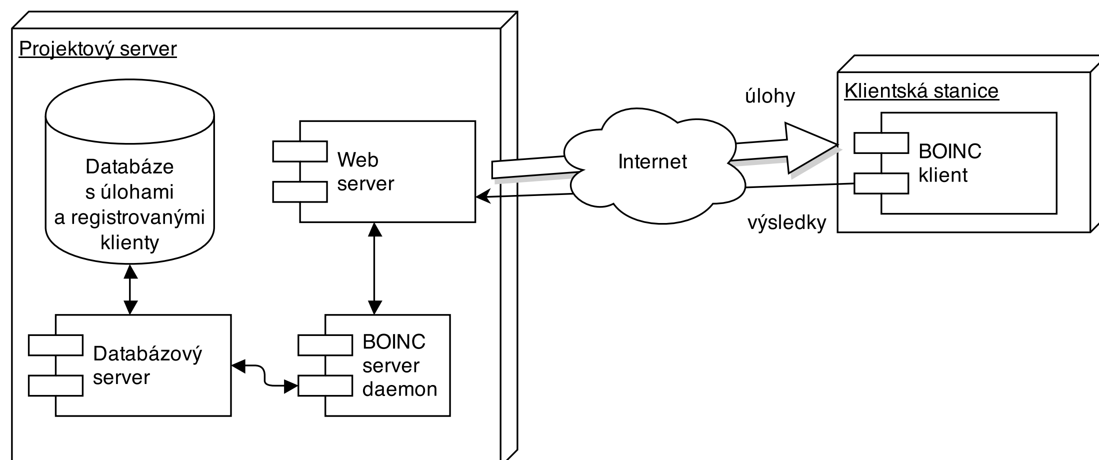
3.2 BOINC

BOINC (zkratka pro Berkeley Open Infrastructure for Network Computing) je platformou pro distribuované výpočty v rámci vědeckých či výzkumných projektů, napsanou v jazyce C a C++. Podporována je široká škála platforem a operačních systémů klientů.[1])

Asi nejvíce jej proslavil projekt SETI@home[2]) (aneb „hledání mimozemšťanů“), pro jehož potřeby BOINC původně v roce 2002 vznikl.[8] Projekt SETI@home je zaměřen na vyhledávání mimozemského vysílání v zachycených rádiových signálech. Podstatou tohoto projektu, stejně tak jako mnoha dalších projektů, je rozkouskování řešení jinak velmi složitého problému na mnoho malých částí, jež jsou distribuovány k výpočtu na uzly rozeseté po celém světě.

BOINC je schopen poskytnout projektům značnou výpočetní sílu, která převýší i výkon superpočítačů právě díky obrovskému množství výpočetních uzlů zapojených do systému. Všechny stroje připojené do Internetu by byly schopny propůjčit výkon v řádu jednotek PFLOPS. [9] A to i přesto, že klient v operačním systému běží s nejnižší prioritou a na výpočtech se začne podílet až ve chvílích, kdy je hostitelský stroj jinak nevyužit.

3.2.1 Popis architektury



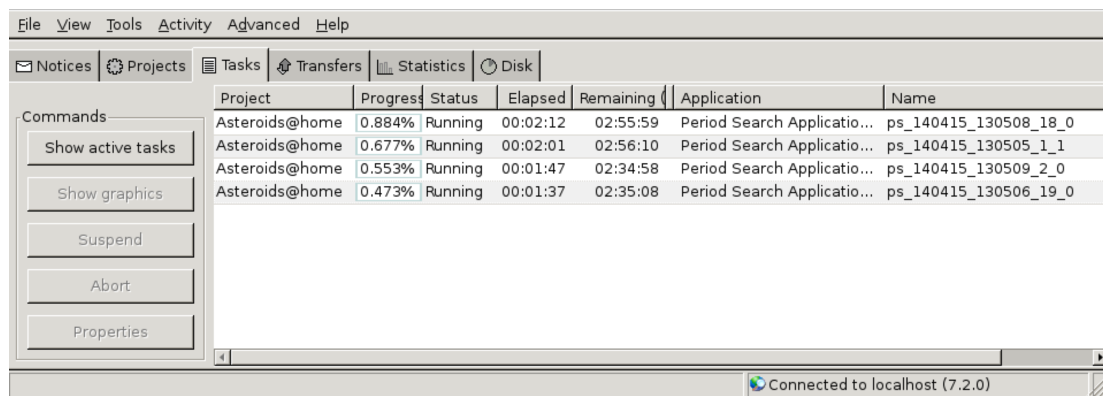
[1]) http://boinc.berkeley.edu/trac/wiki/BoincPlatforms

[2]) http://en.wikipedia.org/wiki/SETI@home

Obrázek 3.1. Architektura BOINC systému.

BOINC je vystavěn na klient-server architektuře. Každý projekt provozuje svůj vlastní server s webovým frontendem a databází. Na serveru jsou umístěny úlohy, aplikace a data k výpočtům. Serverový daemon se stará především o výběr vhodných úloh pro klienty (plánování úloh na výpočetní uzly) a dále o validaci a nakonec parsování a ukládání výsledků do databáze. BOINC používá strategii, kdy jednu úlohu vyšle na více uzlů a jejich výsledky porovná, aby zabránil možnému podvržení výsledků některým z klientů nebo detekoval chybu ve výpočtu.[9]

Výpočetní uzly projektu poskytují dobrovolníci, kteří si na svůj stroj nainstalují BOINC klienta a zaregistrují se u projektu.



Obrázek 3.2. Uživatelské rozhraní BOINC klienta.

To znamená, že poskytnout výpočetní zdroje pro vybrané projekty může prakticky kdokoliv. Jedná se tak o globální, veřejně dostupnou alternativu ke gridům [9], složenou ze zdrojů dobrovolníků a nadšenců. S gridy sdílí snahu o lepší využití málo využívaných zdrojů. Klient pak může zpracovávat úlohy pro více projektů zároveň (z více serverů). Klient žádá v době nečinnosti hostitelského stroje servery projektů o zaslání úloh k výpočtu. Server zasílá spolu s úlohou všechny potřebné aplikace a data přizpůsobené na platformu a operační systém klientského stroje.

3.2.2 Zhodnocení

Je poměrně obtížné zařadit BOINC podle druhů řešení popsaných výše. Řešení je z pohledu serveru spíše druhem plánovače, možnost nějaké centrální správy zdrojů je zde vzhledem k míře distribuovanosti zdrojů velmi omezená.

Toto řešení je velmi vhodné pro značně výpočetně náročné a populární vědecké projekty, kde se dá očekávat velký zájem dobrovolníků a nadšenců. V takových případech je toto řešení schopné zajistit pro úlohy velmi vysoký distribuovaný

výpočetní výkon. Vzhledem k tomu, že míra distribuce výpočetních zdrojů je u tohoto řešení velmi vysoká, ne-li nejvyšší, hodí se pro takové distribuované úlohy, jejichž komunikace mezi částmi úlohy je minimální.

I přesto, že povaha projektu není vhodná pro zpracování na zdrojích široké veřejnosti, nebo je předpoklad, že nezíská podporu dobrovolníků, je možné vytvořit si vlastní projekt[1]) a připojit k němu pouze své vlastní klienty. V tom případě jsme pak ale odkázáni pouze na své vlastní výpočetní zdroje, což příliš nevyhovuje požadavkům na řešení.

Jako mírně problematický se také jeví použitý komunikační model, kdy výpočetní uzly (BOINC klienti) žádají v době nečinnosti hostitele server o práci. Tento model je v nesouladu s modelem popsaném ve funkčních požadavcích 2.3, kde je zadání práce iniciováno uživatelem na straně centrálního prvku (serveru). K tomu, aby byla zachována interaktivita při zadávání úloh, by bylo potřeba větší množství klientů. Čím méně klientů, tím větší bude pravděpodobnost, že uživatel bude velmi dlouho čekat, než některý z vhodných klientů požádá o práci a dojde k naplánování a spuštění úlohy.

klady	zápory
- možnost získat značný výkon - podpora široké škály zdrojů - odolné proti chybám	- nepříliš vhodný komunikační model - závislost na zdrojích dobrovolníků

Tabulka 3.1. Hodnocení vlastností BOINC.

3.3 PBS

PBS (zkratka pro Portable Batch System) je systém pro zpracování dávkových úloh a správu výpočetních zdrojů původně vyvinutý jako společný projekt NASA a superpočítačového centra NERSC na Lawrence Livermore National Laboratory již v roce 1995.[10–11] PBS byl vyvíjen tak, aby odpovídal standardu *„POSIX 1003.2d Batch Environment“* [2]) z roku 1994, jež definuje povinné a volitelné vlastnosti a některé funkce systémů pro správu a spouštění dávkových úloh [3])[4]). Tento systém akceptuje jako výpočetní úlohy (dávky) shellové skripty se vstupními parametry a řídícími příkazy pro PBS.[10–11] Dávky ke zpracování jsou řazeny do front. PBS se následně postará o výběr dávky z fronty, její spuštění, monitoring stavu s možností restartu dávky na jiném stroji v případě problému a nakonec o zaslání výsledků zadavateli.

[1]) http://boinc.berkeley.edu/trac/wiki/ProjectMain

[2]) http://standards.ieee.org/findstds/standard/1003.2d-1994.html

[3]) http://www.open-std.org/JTC1/SC22/WG15/docs/options.htm

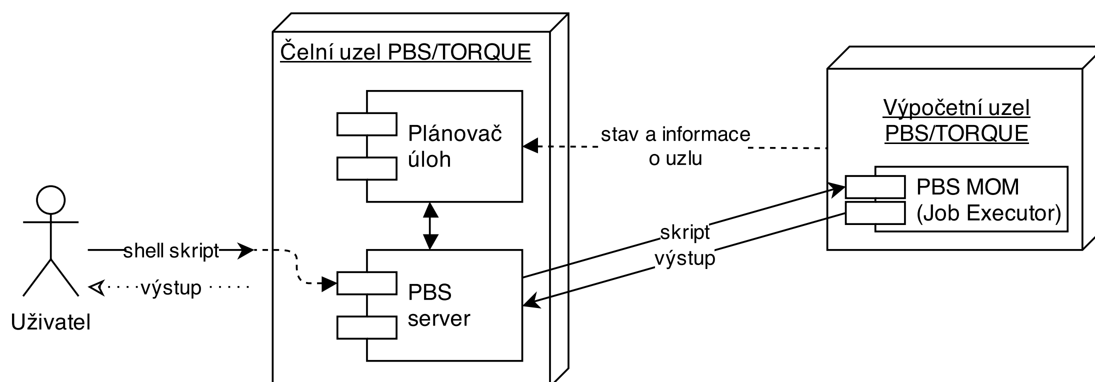
[4]) http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap03.html

Zdrojový kód původní implementace PBS byl proprietární, o několik let později byl však uvolněn pod open source licencí jako OpenPBS. Postupem času pak z OpenPBS vznikly odvozené projekty TORQUE vyvinutý společností Adaptive Computing, Inc. a PBS Professional (PBS Pro), kde PBS Pro je narozdíl od TORQUE komerčním řešením. OpenPBS jako takový již není nadále vyvíjen.[1])

Existují ještě další alternativy k TORQUE, které fungují na stejných principech. Za zmínku stojí např. LoadLeveler od firmy IBM, HTCondor a nebo GridEngine od firmy Oracle. Všechna tato řešení se dají klasifikovat jako lokální správci zdrojů (LRM). Dále bude popisován především systém TORQUE jakožto jedna z implementací LRM, většina základních principů a vlastností je však stále shodná s původní implementací PBS. Oproti původní verzi však přináší mnoho vylepšení v oblasti stability, výkonu a rozšiřitelnosti.

3.3.1 Popis architektury

Systém PBS je složen z několika komponent. Část komponent je přítomna na všech strojích v systému a druhá část pouze na vyhrazeném čelním uzlu, jež je přístupný uživatelům. Architektura systému je podobná jako u obrázku z úvodní kapitoly 1.1.



Obrázek 3.3. Architektura PBS.

První, nejdůležitější komponentou, je „Job Server“ či „PBS server“ běžící typicky na čelním uzlu. Ten poskytuje základní služby týkající se výpočetních úloh, jako je příjem úloh od klientů, zadání úlohy do systému a spuštění výpočtu, zastavení, restartování a zjištění stavu úlohy. Server představuje backend, se kterým komunikují veškeré uživatelské příkazy (např. příkaz „*qsub*“, který zařadí úlohu do fronty na serveru) spouštěné na čelním uzlu. Server se stará

[1]) `http://en.wikipedia.org/wiki/Portable_Batch_System`

také o fronty úloh. Front na jednom serveru může být více a každá z front může mít jinou prioritu při plánování a další atributy či omezení na úlohy v ní zařazené (např. maximální doba běhu úloh). Uživatel si pak při zadávání úlohy může vybrat, do jaké konkrétní fronty má server úlohu zařadit.

Plánovač úloh „Job Scheduler“ komunikuje se serverem a zjišťuje, zda nemá ve frontách nějaké úlohy připravené k naplánování na zdroje. Poté zajišťuje výběr zdrojů pro úlohy a spuštění úlohy (nepřímo přes server). Kromě této role má za úkol zjišťovat stav a informace o výpočetních zdrojích. Zajímavostí je, že komunikace mezi plánovačem a serverem probíhá přes stejné API, jaké využívají i uživatelské příkazy. Toto API se dá jednoduše dále využít k rozšíření celého řešení doimplementováním vlastních částí a příkazů.[10] S TORQUE je dodáván pouze jednoduchý FIFO[1]) plánovač. Je však možné napsat si plánovač vlastní, vyhovující konkrétnímu nasazení, nebo nasadit některý z již existujících plánovačů třetí strany.[11]

Komponenta „Job Executor“ (PBS MOM) má na starosti samotné provedení úlohy. V okamžiku, kdy obdrží úlohu od serveru, spustí pro úlohu na stroji shellovou session co nejvíce podobnou shellu uživatele. K tomu musí uživatel zadávající úlohy mít na výpočetních uzlech založen uživatelský účet. Podporovány jsou neinteraktivní i interaktivní úlohy, kdy je uživateli spuštěn na některém výpočetním stroji interaktivní shell. V tomto shellu je následně úloha spuštěna. Po dokončení výpočtu je serveru zaslán výstup a návratový kód skriptu úlohy. Návratový kód je poté přeposlán uživateli. Samotná zkratka *„MOM“* naznačuje, že komunikace se serverem probíhá prostřednictvím midleware zasíláním zpráv (Message Oriented Middleware).

Soubory potřebné k nahrání na uzel a soubory s výsledky, které se mají stáhnout, je možné uvést do skriptu a PBS zajistí nakopírování těchto souborů.

Jinou možností je přenos souborů vyvolaný skriptem úlohy, kdy se před začátkem výpočtu nakopírují soubory přes SCP, FTPS, RSYNC, či některým jiným přenosovým protokolem. V tu chvíli se však doba přenosu, narozdíl od možnosti výše, započítává do doby běhu úlohy (což může být u větších souborů nežádoucí).

Poslední přístup představuje využití distribuovaných síťových souborových systémů jako jsou NFS[2]) a AFS[3]). Tyto souborové systémy jsou připojeny na všechny výpočetní uzly i na čelní uzel. Potom mají všechny uzly přístup ke stejnému jmennému prostoru se stejnými soubory. V případě NFS si tento prostor může připojit i uživatel na svou stanici, čímž úplně odpadá nutnost ja-

[1]) První úloha, která na server přijde, bude zároveň první naplánovanou a spuštěnou

[2]) `http://cs.wikipedia.org/wiki/Network_File_System`

[3]) `http://en.wikipedia.org/wiki/Andrew_File_System`

kéhokoliv explicitního přesunu souborů mezi uživatelem a uzly a vše probíhá na pozadí.

3.3.2 Zadání úlohy

Shellový skript s úlohou uživatel zařadí do požadované fronty ke zpracování příkazem „*qsub*“ s následující syntaxí [10]:

```
qsub [-I] [-q fronta] [-N nazev_ulohy] -l pozadavky_na_zdroj skript
```

Direktiva $-I$ specifikuje interaktivní úlohu a požadavky na zdroje jsou řetězcem tvořeným z dvojic $vlastnost = hodnota$. Direktivy je také možné zapsat přímo do skriptu jako komentáře:

```
#!/bin/bash
#PBS -N ulohaSleep
#PBS -q long
#PBS -l nodes=1:ppn=1,mem=1kB
#PBS -o /home/user/ulohy/sleep.stdout

sleep 31536000
echo 'Slept for one year!'
```

Příkaz „*qsub*“ je schopný tyto komentáře načíst a předat PBS serveru. Ve výše uvedeném případu tak PBS zařadí úlohu do fronty „long“ (fronta pro dlouho běžící úlohy) a vyhradí pro ni jedno jádro jednoho uzlu. Specifikováním požadavků pomocí direktivy $-l$ jsou před spuštěním úlohy alokovány zdroje v zadaném rozsahu. K dispozici jsou například následující vlastnosti [10]:

- **mem**: Požadovaná maximální velikost fyzické paměti na každém uzlu.
- **walltime**: Maximální předpokládaná doba běhu úlohy.
- **file**: Velikost největšího možného souboru vygenerovaného úlohou.
- **host**: Název preferovaného výpočetního uzlu.
- **nodes**: Množství potřebných výpočetních uzlů.
- **ppn**: Kolik procesorů má být vyhrazeno na každém uzlu.

Úplný seznam PBS direktiv je k dispozici v manuálových stránkách příkazu „*qsub*“. Seznam vlastností zdrojů se však u různých provozovatelů systémů liší, je částečně konfigurovatelný.

3.3.3 Zhodnocení

PBS se jeví jako vhodnější řešení než BOINC. Požadavky na aplikaci z větší části splňuje. Díky dostupnému API je možné ho integrovat do vlastní aplikace nebo rozšířit jeho funkcionalitu podle potřeby. Poskytuje příkazy pro zadávání a správu úloh včetně monitoringu průběhu výpočtu. Obsahuje také plánovač

úloh. Plánovač v systému figuruje jako plugin, dá se nahradit libovolným plánovačem s jinou politikou plánování než je výchozí triviální FIFO. Zajištěn je i monitoring výpočetních uzlů a v případě potřeby i přenosy souborů spojených s úlohami.

Ke své funkci vyžaduje nainstalovanou komponentu „PBS mom“ na výpočetních uzlech a vyhrazený uzel, který bude sloužit jako PBS server. Tento čelní uzel slouží jako centralizovaný přístupový bod k distribuovaným výpočetním zdrojům.

Hlavním limitujícím faktorem tohoto řešení může být fakt, že vzhledem k uzpůsobení na spouštění shellových skriptů je použitelné pouze pro Unix/Linux. Dalším omezením je, že opět vyžaduje nainstalovaný konkrétní specializovaný software na všech výpočetních uzlech. Řešení je vhodné pro správu úloh v rámci clusteru a gridu, kde je toto u všech uzlů typicky zajištěno. Samostatně není příliš použitelné k posílání úloh do více různých gridů/clusterů.

klady	zápory
- rozšiřitelnost	- pouze pro Unix/Linux
- flexibilita	- spíše jen v rámci jednoho gridu, clusteru
- vhodné pro grid, cluster	

Tabulka 3.2. Hodnocení vlastností PBS.

3.4 MOAB

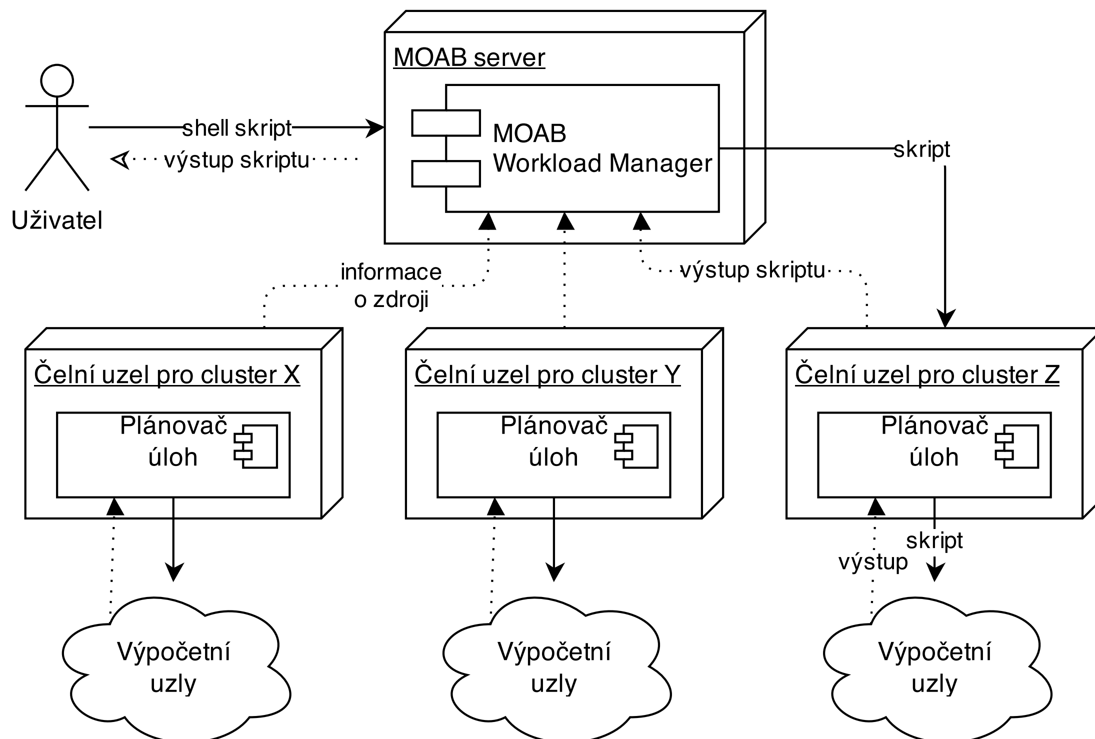
Moab Workload Manager je sotware vyvinutý stejnou společností jako systém PBS TORQUE. Stejně jako u TORQUE se jedná o systém pro správu a distribuci dávkových úloh na výpočetní zdroje a sdílí s ním mnoho principů.

Narozdíl od něj však funguje na vyšší vrstvě. Workload manager je definován jako plánovač úloh, který pracuje nad vícero správci zdrojů a plánovači. Sdružuje všechny tyto zdroje do jediné domény.[12] Uživatelům pak poskytuje jednotné uživatelské rozhraní a skrývá před nimi odlišnosti při zadávání úloh na různorodé zdroje v doméně.

3.4.1 Popis architektury

Takovému plánovači se také často říká **meta-plánovač**, jelikož neplánuje úlohy na výpočetní uzly, ale pouze předává úlohu nižšímu plánovači některé subdomény. Teprve ten zajistí spuštění úlohy a hlášení stavu nadřazenému plánovači. Tento postup je ilustrován na obrázku 3.4. Přerušovanou čarou je naznačen tok

výsledků a informací o zdrojích, plnou čarou pak cesta skriptu systémem k výpočtu na konečném uzlu.



Obrázek 3.4. Proces zpracování úlohy v systému MOAB.[12]

3.4.2 Zadání úlohy

Způsob popisu výpočetní úlohy a její zadání do systému MOAB je analogický k postupu u systému TORQUE. Je zde pouze zaměněn příkaz *„qsub“* za příkaz *„msub“*. Tento příkaz má stejné přepínače. Za zmínku však stojí přidaná vlastnost zdrojů „partition“ použitelná pro určení, na který z clusterů MOABu se má úloha naplánovat.[12] Pokud tedy chce uživatel zajistit, aby byla jeho úloha spuštěna na clusteru „X“, zadá následující příkaz:

```
msub -q nazev_fronty -l partition=X skript
```

Obdobně jako u PBS lze zadat řídící direktivy na začátek skriptu [12]:

```
#!/bin/bash
#MSUB -q nazev_fronty
#MSUB -l partition=X
...
```

3.4.3 Zhodnocení

MOAB vychází ze systému TORQUE a rozšiřuje jeho možnosti na použití při plánování úloh v prostředí více clusterů. MOAB je spíše meta-plánovač,

klady	zápory
- výhody TORQUE - multi-clusterové řešení	- stále pouze Unix/Linux

Tabulka 3.3. Hodnocení vlastností řešení MOAB.

zatímco TORQUE je více než plánovač správce zdrojů a server pro správu úloh. To znamená, že je možné výhodně použít kombinaci obou zmíněných řešení tak, aby se jejich možnosti navzájem doplňovaly.

3.5 GRAM

GRAM5 (Grid Resource Allocation and Management) je, jak již z názvu vyplývá, opět spíše správce výpočetních zdrojů jako TORQUE a spoléhá se na externí plánovače úloh. Avšak podobně jako MOAB je využitelný pro správu zdrojů z více odlišných domén a distribuci úloh na tyto zdroje. K distribuci úloh na konečné výpočetní uzly v různých doménách využívá lokálních správců zdrojů (LRM). GRAM je součástí většího balíku nástrojů pro gridová prostředí pod názvem **Globus Toolkit**.[13]

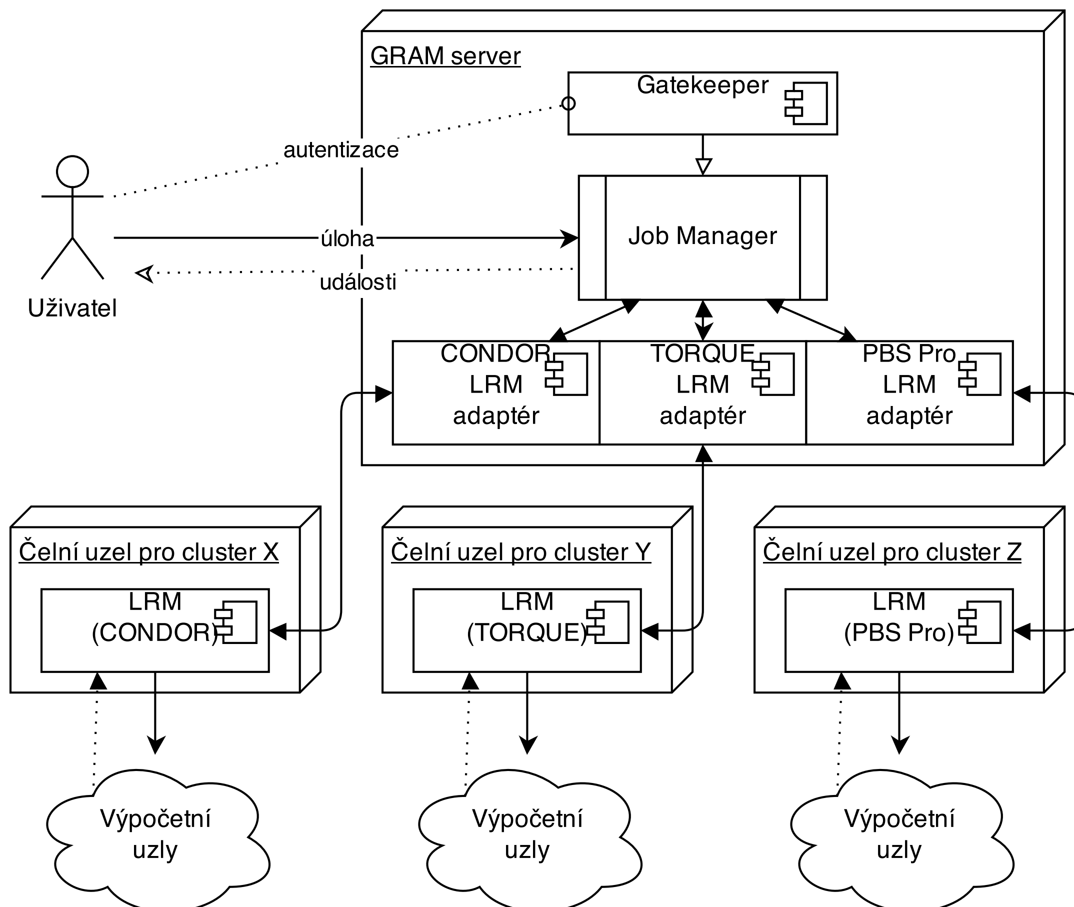
3.5.1 Popis architektury

GRAM je opět modulárním řešením složeným z komponent a démonů. Jednou z nich je komponenta „Gatekeeper“. Ta přijímá spojení od uživatelů a zajišťuje jejich autentizaci. Po úspěšné autentizaci spustí pro uživatele „Job Manager“ proces.

Job Manager přijímá úlohy od uživatele a stará se o spuštění úlohy na některém LRM. Zároveň zajišťuje přesuny dat a souborů spojených s úlohou. Ke splňení tohoto úkolu lze využít nástroj GridFTP, který je součástí Globus Toolkit.

Další důležitou komponentou je zde „Scheduler Event Generator“. Ten poslouchá zprávy týkající se úloh uživatele (změna stavu, spuštění či ukončení úlohy, atd.), které dostává od lokálních správců úloh, a převádí je na události nezávislé na konkrétní implementaci LRM. Tyto události jsou pak přeposílány uživateli nebo zachyceny a zpracovány některou z komponent Globus Toolkit.

Poslední, neméně důležitou částí jsou „LRM adaptéry“. Ty slouží jako komunikační rozhraní mezi Job Managerem (a případně dalšími komponentami GRAM) a konkrétními implementacemi LRM na podřízených zdrojích.[13] Adaptér má na jedné straně jednotné rozhraní v rámci systému GRAM, na straně druhé pak volá specifické funkce (zadání, monitoring, rušení úlohy) rozhraní konkrétní implementace LRM.



Obrázek 3.5. Architektura systému GRAM.

3.5.2 Zadání úlohy

Pro zadávání úloh uživateli do systému GRAM jsou dostupné klientské nástroje. V rámci Globus Toolkit jsou to nástroje „*globusrun*“, „*globus-job-submit*“ a „*globus-job-run*“. K zadávání úloh do systému GRAM se dají použít také nástroje či meta-schedulery třetích stran.[13] K popisu samotných úloh a jejich závislostí se zde používá formátu JSDL (Job Submission Description Language), jež vychází z formátu XML. Definice úlohy v něm vypadá následovně:

```
<jsdl:JobDefinition>
  <JobDescription>
    <Application>
      <jsdl-posix:POSIXApplication>
        <Executable>/bin/sleep</Executable>
      </jsdl-posix:POSIXApplication>
    </Application>
    <Resources>
      <OperatingSystem>...</OperatingSystem>
      ...
    </Resources>
  </JobDescription>
<jsdl:JobDefinition>
```

3.5.3 Zhodnocení

GRAM poskytuje uživatelům rozhraní pro zadávání a monitoring úloh, přičemž zajišťuje i přenosy souborů. Úlohy je schopen distribuovat na clustery s různými správci zdrojů (LRM). Poskytuje adaptéry pro většinu používaných LRM řešení. Co se týká plánování úloh (s podporou front) na zdroje, vyžaduje některý z dostupných pluginů [1]). Distribuce úloh je možná i na výpočetní uzly jiné platformy než Unix/Linux, pokud je použit adaptér a LRM, který to umožňuje (např. Oracle Grid Engine zmíněný kapitole 3.3).

GRAM oproti předchozím řešením (kde bylo potřeba toto řešit jinými mechanismy) navíc řeší i zabezpečení a autentizaci uživatelů.

Celkově se řešení jeví jako vhodné k použití a splňující definované požadavky. Jedinou zjištěnou nevýhodou je o něco vyšší komunikační režie při zpracovávání úloh a tedy pomalejší odezva systému. Toto je znatelné obzvlášť u jednoduchých nenáročných úloh.[13]

klady	zápory
- bezpečnost - distribuce úloh na více clusterů - modulárnost, pluginy - podpora mnoha správců zdrojů	- pomalejší odezva

Tabulka 3.4. Hodnocení vlastností systému GRAM.

3.6 DIANE/GANGA

DIANE (Distributed Analysis Environment) a GANGA (Gaudi Athena and Grid Alliance) představují dva separátní, avšak spolupracující softwarové produkty, vyvinuté v CERN IT.[14] DIANE vznikl za účelem zprovoznění aplikací CERNu v největším světovém gridu EGEE.[15] DIANE je framework pro takzvané **pilotní úlohy**. Opět pracuje obdobně jako meta-plánovače v prostředí více clusterů.

Odlišností oproti předchozím řešením je způsob, jakým jsou úlohy uživatele distribuovány na zdroje. Pilotní úlohy jsou kontejnery pro uživatelské úlohy. Do správců zdrojů jsou pak zadány ke spuštění právě tyto pilotní úlohy. Ty po spuštění na výpočetním uzlu žádají o práci (tedy o uživatelské úlohy). Tento přístup se také někdy označuje jako *„late binding"*.[2] Tímto způsobem uživatelské úlohy obcházejí frontu plánovače a jsou rovnou spoušteny v kontejnerech na výpočetních uzlech. Alternativně lze realizovat plánování vlastní, nezávisle na tom, jak jsou nastaveny plánovače na zdrojích.

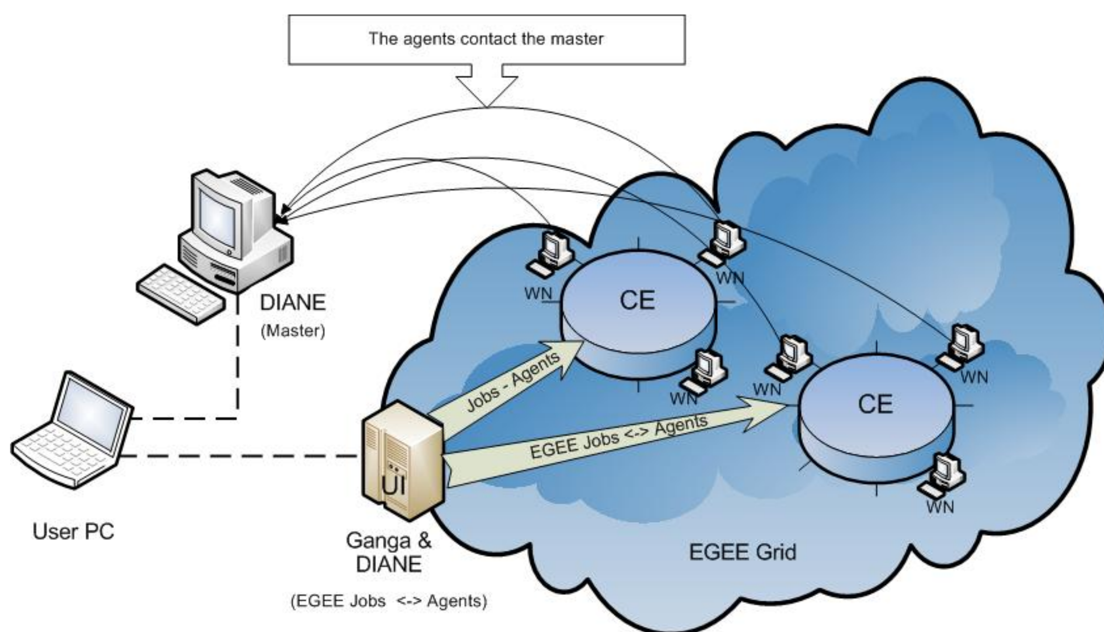
[1]) `http://toolkit.globus.org/grid_software/computation/gram-plugins.php`

Hlavním smyslem nástrojů s pilotními úlohami je co nejvíce omezit dobu čekání ve frontě plánovačů a tím zkrátit celkovou dobu výpočtu. Efekt zrychlení se ještě více násobí při spouštění velkých dávek úloh najednou. V případě tohoto řešení se čeká jen na spuštění pilotních úloh. Ty se navíc dají zadat do plánovačů s předstihem.

GANGA je gridové rozhraní vyvinuté v rámci projektů LHCb[1]) a Atlas[2]).[15] Poskytuje standardní příkazy pro zadávání úloh či interakci s gridem a dokonce i GUI. Úlohy jsou zadávány ke zpracování LRM. Z těch jsou podporovány LSF, PBS, LCG (gLite) a HTCondor.[15]

3.6.1 Popis architektury

Architekturu řešení tvoří tři základní komponenty – Master, Worker a gridové rozhraní GANGA. První z nich, tzv. „Master“, je server spouštěný uživatelem. Pro každou dávku úloh se vytváří jedna instance Mastera. Master je také zodpovědný za plánování úloh z dávky na Workery. Worker je pilotní úloha (v obrázku 3.6 jako *„agent“*), která již byla spuštěna na některém výpočetním uzlu. Workery po svém spuštění kontaktují Master server a žádají od něj instrukce a soubory k úlohám. Dále pak s Masterem komunikují ohledně výsledků. Pro pokročilejší monitoring úloh je k dispozici plugin IMonitoringService.[15]



Obrázek 3.6. Architektura systému DIANE/GANGA. Převzato z: [16].

Mezi Workery a Masterem používá DIANE jako komunikační middleware standard CORBA (konkrétně implementaci omniORB[3])).[15] Tento middleware

[1]) https://lhcb.web.cern.ch/lhcb/

[2]) http://atlas.ch/

[3]) http://omniorb.sourceforge.net/

umožňuje provádět mezi Masterem a Workery vzdálená volání metod (RPC). Pro tuto komunikaci je však velmi důležité, aby Master byl pro každý výpočetní uzel dostupný po síti (měl veřejnou IP adresu). Výpočetní uzly jako takové nebývají z vnějšku přímo dostupné.

Poslední komponentou je rozhraní GANGA. Přes něj jsou zadávány pilotní úlohy do systému. Dá se však obejít i bez ní. K dispozici jsou „Submitter“ (jinde také jako „AgentFactory“ [16]) python skripty, které plní stejný účel.

3.6.2 Zadání úlohy

Úlohy jsou v DIANE zadávány skripty napsanými v jazyce Python. Následující skript (budiž pojmenován *„diane-test“*) zajistí spuštění Master serveru u uživatele [14]:

```
from diane_test_applications import ExecutableApplication as application

def run(input,config):
        d = input.data.task_defaults
        d.input_files = ['program']
        d.output_files = ['program.output']
        d.executable = 'program'

        for i in range(10):
                t = input.data.newTask()
                t.args = [str(i)]
```

Tímto jsme definovali dávku deseti úloh. Každá z těchto úloh má spustit soubor *„program“* s číselným argumentem. Zároveň se před spuštěním postará o nakopírování tohoto souboru. Po jeho provedení nakopíruje zpátky na Mastera soubor *„program.output“*. Soubor *„program“* může být libovolný spustitelný soubor nebo skript. Triviální případ může vypadat například následovně:

```
#!/usr/bin/env bash
echo "Job $1 has been run on `hostname`." > program.output
```

Dávka úloh je pak rozeslána na volné Worker uzly příkazem:

```
diane-run diane-test
```

Tímto dojde ke spuštění Mastera a naplnění jeho fronty deseti úlohami. Tyto úlohy si pak postupně vyzvednou Worker uzly ke spočítání. K tomu, aby měl Master k dispozici nějaké Workery, je potřeba zadat je jako pilotní úlohy přes standardní rozhraní gridu. Nápomocny jsou připravené skripty (Submittery), které se o toto zadávání Workerů starají [14]:

```
ganga PBSSubmitter.py \
--diane-worker-number=10 \
--qsubq short \
--qsubl nodes=1:nfs4
```

Takto spuštěný submitter požádá grid o spuštění deseti Workerů ve frontě short (krátkodobě běžící úlohy). Kromě fronty umí předat i další PBS direktivy příkazu „*qsub*“. Submitter je potřeba spouštět na čelním uzlu gridu.

3.6.3 Zhodnocení

DIANE volí trochu jiný přístup než předešlá řešení a díky tomu má své specifické výhody a nevýhody. Implementace v Pythonu zajišťuje do značné míry multiplatformnost a snadnou modifikovatelnost. Řešení zásadně urychluje spouštění v případě velkých dávek úloh, kdy nemusí každá úloha zvlášť čekat ve frontě plánovače gridu.

Tato výhoda však může být zároveň nevýhodou. Důvodem je, že úlohy uživatele takto obchází interní plánovače a správce úloh gridu. To může mít za následek neoptimální výběr zdrojů a výpočetních uzlů pro úlohy. Tyto zdroje jsou vybírány pouze jednou pro pilotní úlohy (kontejnery). Lokální správce zdrojů nemůže z popisu pilotní úlohy vybrat optimální zdroj pro všechny úlohy, které v tomto kontejneru kdy poběží.

Tímto přístupem přicházíme o výhody, které lokální správci a plánovače poskytují (monitoring stavu a zatížení zdrojů, výběr optimálních zdrojů pro úlohu, atp.).

klady	zápory
- rychlost při velkém množství úloh v dávce	- obcházení interních komponent gridu/clusteru

Tabulka 3.5. Hodnocení vlastností DIANE v kombinaci s GANGOU.

3.7 SAGA

SAGA (Simple API for Grid and Distributed Applications) je specifikací aplikačního rozhraní GFD.90[1]) od sdružení OGF[2]). Toto rozhraní má za úkol poskytovat přístupovou vrstvu k nejčastěji používaným komponentám v gridech.

Existuje několik implementací této specifikace. Nejpopulárnější a nejaktivněji vyvíjenou je v současné době **SAGA-Python** napsaná v jazyce Python. Hlavní důraz u tohoto řešení je, jak již vyplývá z názvu, kladen na co největší jednoduchost. Jednoduchost nasazení, používání či modifikování a rozšiřování.

Jednoduchost by však sama o sobě nestačila. SAGA-Python podporuje množství gridových plánovačů, správců zdrojů, ale i přenosových protokolů. Podporovány jsou jak jednoduché izolované úlohy, tak složité se závislostmi mezi sebou.

[1]) `http://www.ogf.org/documents/GFD.90.pdf`

[2]) Open Grid Forum – `https://www.ogf.org`

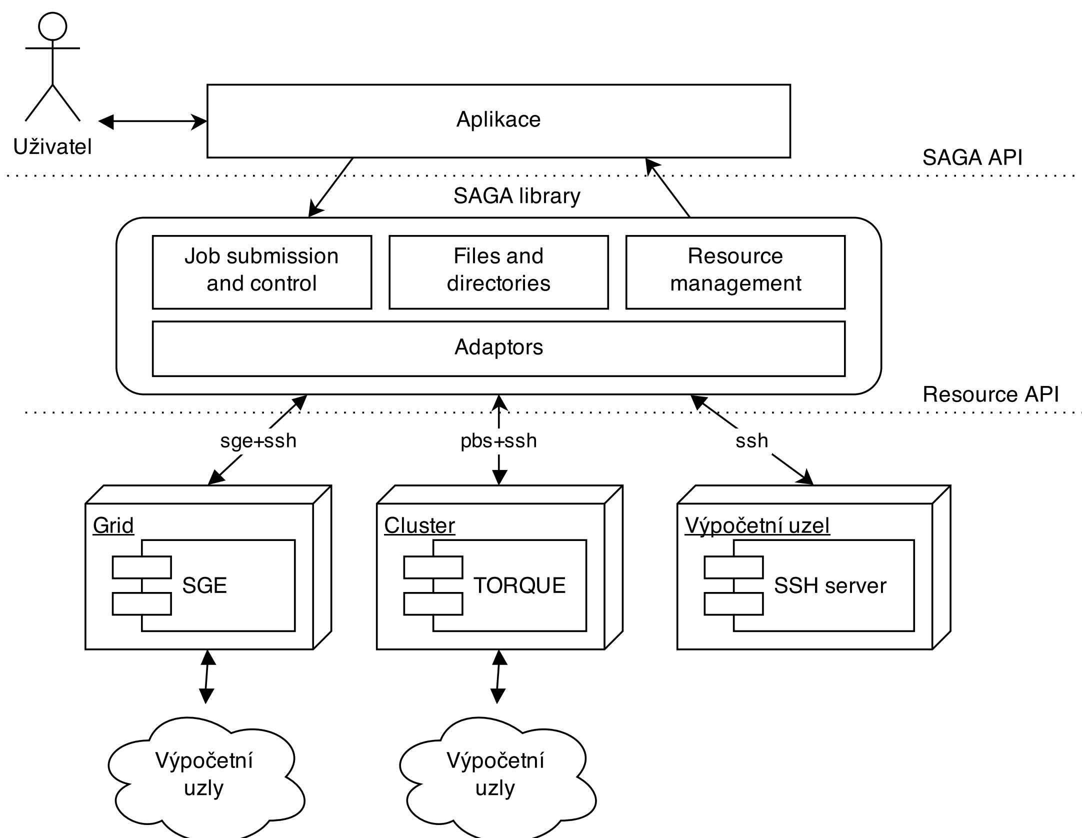
Toto řešení používá mnoho rozsáhlých distribuovaných infrastruktur, mezi nimi například XSEDE[1]), OSG[2]) a FutureGrid[3]).

SAGA-Python v současnosti podporuje následující rozhraní a protokoly [17]:

- **Zadávání úloh** – SSH, GSISSH, Condor a Condor-G, PBS a TORQUE, Sun Grid Engine, SLURM, LSF
- **Přenosy a správa souborů** – SFTP, GSIFTP, HTTP/HTTPS, iRODS
- **Správa zdrojů / Cloudy** – Amazon EC2 (libcloud)

3.7.1 Popis architektury

Knihovna SAGA se skládá z modulu pro správu úloh, modulu pro operace nad vzdálenými či lokálními soubory a modulu pro správu zdrojů a adaptérů. Na obrázku ještě není znázorněn modul, který zajišťuje autentizaci uživatele na zdrojích.



Obrázek 3.7. Architektura knihovny SAGA.

Uživatel vytváří a zadává úlohy pomocí libovolné vlastní aplikace. Tato aplikace využívá služeb knihovny SAGA. Tato knihovna je pak schopna realizovat

[1]) Extreme Science and Engineering Discovery Environment – `https://www.xsede.org/`

[2]) Open Science Grid – `http://www.opensciencegrid.org/`

[3]) `https://portal.futuregrid.org/`

skrze adaptéry služby na výpočetních zdrojích. To, s jakým zdrojem a jakým protokolem má SAGA komunikovat, určuje uživatelská aplikace.

Knihovna jako taková neobsahuje žádný plánovač, pouze poskytuje jednotné rozhraní nad gridovými řešeními a protokoly. Je možná i kombinace protokolů. Například na obrázku 3.7 znázorněné *pbs+ssh* znamená, že SAGA bude spouštět na cílovém stroji (typicky čelním uzlu clusteru/gridu) příkazy pro PBS skrze SSH relaci.

3.7.2 Zadání úlohy

Úlohy jsou v knihovně SAGA reprezentovány instancemi třídy:

```
job = saga.job.Description()
```

Popis takové úlohy může vypadat následovně [17]:

```
job.environment = {'MYOUTPUT':'Hello from SAGA'}
job.executable  = 'skript'
job.arguments   = ['MYOUTPUT']
job.output      = '/tmp/mysagajob.stdout'
job.error       = '/tmp/mysagajob.stderr'
job.queue       = 'short'
```

Tento popis říká, jaké se mají úloze nastavit proměnné prostředí a jaký soubor se má spustit. Další atributy říkají, kam se má uložit standardní a chybový výstup úlohy. Poslední atribut určuje frontu, do které má plánovač na cílovém zdroji úlohu zařadit (pokud tento atribut podporuje).

Potřebné přenosy souborů úlohy se deklarují atributem „input_file_transfer“, popř. „output_file_transfer“. Zde si lze povšimnout způsobu zápisu cesty k souboru, která je ve formě URI[1]). Tento zápis se objevuje i u jiných zdrojů. První část URI specifikuje, jaký adaptér má být použit.

```
job.input_file_transfer = [file://localhost/root/skript > 'skript']
```

3.7.3 Zhodnocení

SAGA je velmi elegantní knihovna, jednoduchá na nasazení a použití. Umožňuje využít k výpočtům širokou škálu strojů. Poskytuje metody pro spouštění úloh jak v gridech a clusterech, tak i na samostatných strojích (kde je jediným požadavkem běžící SSH server).

Nevýhodou je, že toto řešení není samostatné. Předpokládá se použití v rámci vlastní aplikace. Tato vlastní aplikace pak musí řešit výběr zdrojů pro úlohy a vytváření úloh.

[1]) Uniform Resource Identifier

klady	zápory
- jednoduchost - množství adaptérů	- nemá plánovač úloh - pouze knihovna

Tabulka 3.6. Hodnocení vlastností SAGA.

3.8 Výběr výpočetních zdrojů

V této podkapitole jsou krátce rozebrány možnosti, které se nabízí jako výpočetní zdroje pro využití s konečným řešením.

V současnosti je k dispozici pouze několik málo samostatných (byť relativně výkonných) strojů. Na nich jsou nainstalovány všechny potřebné nástroje a bylo by vhodné je i nadále k výpočtům využívat.

Vzhledem k tomu, že úloh je velké množství, a jejich zpracování trvá netriviální dobu, je třeba poohlédnout se, jak tyto výpočetní zdroje výraznějším způsobem rozšířit. Vzhledem k výše popisovaným řešením bude toto poohlédnutí se směřovat k prostředí gridů. Pokud chceme zůstat v českém akademickém a vědeckém prostředí, je jedinou takovou možností MetaCentrum.

3.8.1 MetaCentrum

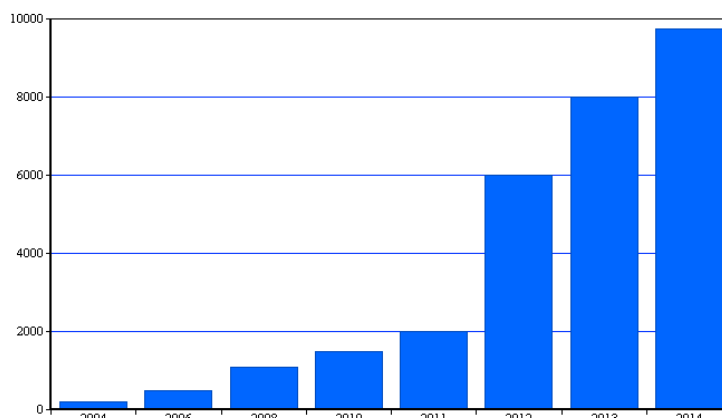
MetaCentrum[1]) je aktivitou sdružení CESNET[2]) a představitelem české národní gridové infrastruktury (NGI). Je oficiálně uznávanou součástí Evropské Gridové Iniciativy (EGI). Úkolem Metacentra je sdružovat již existující a nově pořizovaná výpočetní centra akademické a vědecké komunity v ČR.[18]

[1]) http://www.metacentrum.cz/cs/

[2]) http://www.cesnet.cz/

Obrázek 3.8. Mapa MetaCentra. Převzato a upraveno z: [1])

V současnosti se MetaCentrum skládá z institucí a jejich zdrojů v lokalitách znázorněných na mapce 3.8. K dispozici je výkon celkem **9732** jader procesorů na **547** výpočetních uzlech v **31** clusterech [2]). Vzhledem k tomu, že koncem roku 2010 se počet procesorů pohyboval pouze kolem 1500, jedná se o dramatický nárůst výkonu během posledních několika let.



Obrázek 3.9. Vývoj množství jader procesorů zapojených MetaCentra. Převzato z: [19]

Přístup na tyto zdroje je řízen členstvím ve virtuální organizaci „Metacentrum VO“. Do této VO se mohou zaregistrovat všichni akademičtí pracovníci, zaměstnanci a studenti vědeckovýzkumných či akademických institucí v České republice. V MetaCentru je možné založit ještě další virtuální organizace pro potřeby koordinace a sdílení mezi členy takovéto organizace.

[1]) `https://metavo.metacentrum.cz/`

[2]) Zdroj dat: `http://metavo.metacentrum.cz/pbsmon2/hardware`.

Z technického hlediska je v MetaCentru ke správě zdrojů a úloh provozován komerční systém PBS Pro v kombinaci s vlastním forkem TORQUE[1]). Pro správu a distribuci virtuálních organizací a uživatelských účtů na zdroje je použit vlastní systém Perun[2]). Sdílení dat mezi uzly gridu je zajištěno síťovými souborovými systémy AFS a NFS (viz 3.3). Svazky, jež jsou připojeny na jednotlivých uzlech, se z velké části fyzicky nacházejí na Datových úložištích CESNETu[3]). Autentizace a autorizace uživatelů je řešena pomocí systému Kerberos[4])

3.9 Výběr řešení

Z výše uvedených možností se jeví jako nejméně výhodná platforma BOINC. Pro účel využití v této práci (který není veřejně prospěšný jako boj s rakovinou nebo podobné projekty na platformě BOINC) hrozí, že by se nenašlo dostatek dobrovolníků. V takovém případě není BOINC schopen konkurovat ostatním řešením.

PBS je taktéž léty ověřená a běžně používaná technologie. Nicméně toto řešení je nasaditelné pouze v lokální doméně jednoho clusteru či gridu. Což omezuje nebo přímo zabraňuje využívání dalších zdrojů mimo cluster nebo grid.

Další v řadě, MOAB a GRAM, využívají předností meta-schedulerů, a je schopen distribuce úloh do více clusterů. Jejich slabinou je pouze fakt, že nejsou schopni plánovat úlohy na zdroje přímo. Všechny zdroje použitelné s tímto řešením musí mít nainstalován gridový software (LRM nebo plánovač).

Požadavek na tento sofware je o něco slabší u DIANE, avšak stále přetrvává. Teoreticky by šlo spustit pilotní úlohu i na samostatném uzlu mimo grid, ale není to běžnou praxí a nedává to příliš smysl.

Poslední z možností je framework SAGA. Jako jediný je schopen kombinovaného využití gridových i samostatných zdrojů (kde jediné, co vyžaduje, je SSH). Nevýhodou je nutnost vlastní implementace některých komponent. Avšak díky jednoduchosti a přehlednosti knihovny bylo i přes tuto nevýhodu rozhodnuto využít k řešení právě **SAGA-Python**.

[1]) `https://github.com/CESNET/torque`

[2]) `http://perun.cesnet.cz/web/index.shtml`

[3]) `https://du.cesnet.cz/`

[4]) `https://wiki.metacentrum.cz/wiki/Autentizační_systém_Kerberos`

Kapitola 4
Návrh řešení

Tato kapitola se bude zaobírat podrobnějším návrhem aplikace. Řešení bude založené na knihovně pro zadávání úloh do distribuovaného prostředí SAGA-Python která je celá napsaná v jazyce Python. Cílovou aplikaci tedy dává smysl naimplementovat též v jazyce Python.

Knihovna SAGA-Python nevyžaduje na výpočetních zdrojích žádný dodatečný klientský software. Tuto výhodu (z hlediska jednoduchosti nasazení, údržby a přidávání dalších zdrojů) se budeme snažit zachovat a postačí tak naimplementovat pouze server (v kontextu kapitoly 2.3 jako „backend“).

V první řadě je však vhodné nejprve stanovit komunikační kontrakt mezi uživatelským rozhraním a backendem. Tímto úkolem se bude zabývat následující část kapitoly.

4.1 Návrh komunikačního protokolu

V kapitole 2.4 již bylo deklarováno, že backend si bude s frontendem vyměnovat data ve formátu JSON. Ke specifikaci syntaxe jednotlivých příkazů byl použit standard ABNF[1]) (Augmented Backus-Naur Form). S vyjímkou toho, že pravidla, která už jsou u některého příkazu deklarována, nebudou znovu deklarována u dalších příkazů, ale budou rovnou použita.

Příkaz ECHO

- *Syntaxe příkazu v ABNF:*

```
$ECHO_COMMAND ::= "ECHO" $data
$data         ::= 1*$item | STRING
$item         ::= $key $value
$key          ::= STRING
$value        ::= STRING
```

- *Slovní popis:*

  Strana, která obdrží tento příkaz, musí zaslat obsah $\$data$ v nezměněné podobě zpět odesílateli příkazu. Tento příkaz slouží jako kontrola, že druhá strana je naživu a odpovídá na dotazy.

[1]) `http://tools.ietf.org/html/rfc2234`

- *Příklad v JSON:*

```
["ECHO", "Hello world, is anybody there?"]
```

Příkaz COMPUTE

- *Syntaxe příkazu v ABNF:*

```
$COMPUTE_COMMAND  ::= "COMPUTE" $job_id $job_description
$job_id           ::= INTEGER
$job_description  ::= $script_block $parameters_block [$environment]
$environment      ::= $data /* stejné jako u ECHO */

/* Popis skriptu */
$script_block   ::= "runscript" $script_data
$script_data    ::= $executable $files $resource_needs $preferred
$executable     ::= "file" STRING
$files          ::= "required" *STRING
$resource_needs ::= "requires" *$feature
$feature        ::= STRING STRING /* name, value */
$preferred      ::= STRING

/* Popis parametrů */
$parameters_block ::= "parameters" *$parameter
$parameter        ::= $position $type $direction $value [$name]
$position         ::= INTEGER | "stderr" | "stdout"
$type             ::= "file" | "string"
$direction        ::= "inout" "in" | "out"
$value            ::= STRING
$name             ::= STRING
```

- *Slovní popis:*
Tento příkaz slouží k zadání úlohy backendu k provedení výpočtu. Atribut $\$job_id$ slouží jako jednoznačný identifikátor úlohy během zpracování backendem. Atribut $\$environment$ nastavuje proměnné prostředí pro úlohu. Následuje specifikace spustitelného souboru a jeho požadavků a závislostí. Zde je možné atributem $\$preferred$ vyžádat naplánování úlohy na konkrétní zdroj. Blok s parametry definuje vstupně výstupní parametry úlohy, jejich pozici na příkazovém řádku, hodnotu a případně název. Pokud má parametr název, objeví se hodnota parametru ve výsledcích právě pod tímto názvem.
- *Příklad v JSON:*

```
["COMPUTE", "1",
  {"runscript":
    {"file":"skript",
     "required":["skript-data"],"requires":{"arch":"x86_64"}},
  "parameters":[
     {"pos":"stdout","type":"string","inout":"out","name":"x"}]
}]
```

Příkaz RESULTS

- *Syntaxe příkazu v ABNF:*

```
$RESULTS_COMMAND ::= "RESULTS" $result_data
$result_data     ::= "jobID" $job_id "results" *$result
$result          ::= $output_parameter_name $value
$value           ::= STRING
$output_parameter_name ::= STRING
```

- *Slovní popis:*
  Tento příkaz bude používán k zaslání výsledků výpočtu frontendu. Výsledky obsahují hodnoty všech pojmenovaných výstupních parametrů. Pokud je parametr typu „*file*“, je jako hodnota výsledku použita cesta k souboru.
- *Příklad v JSON:*

```
["RESULTS", {"jobID":"1","results":{"x":"3,14"}}]
```

Příkaz NOTIFY

- *Syntaxe příkazu v ABNF:*

```
$NOTIFY_COMMAND ::= "NOTIFY" $notification
$notification   ::= "jobID" $job_id "state" $state
$state ::= "queued" | "scheduled" | "preparing" |
           "executing" | "finishing" | "finished"
```

- *Slovní popis:*
  Tímto příkazem je frontend informován o změně stavu úlohy.
- *Příklad v JSON:*

```
["NOTIFY", {"jobID":"1","state":"finished"}]
```

Příkaz ERROR

- *Syntaxe příkazu v ABNF:*

```
$ERROR_COMMAND ::= "ERROR" $error_data
$error_data    ::= "jobID" $job_id "errType" $etype "errMsg" $emsg
$etype ::= STRING
$emsg  ::= STRING
```

- *Slovní popis:*
  Tento příkaz slouží k informování druhé strany o tom, že během zpracování úlohy nastala chyba.
- *Příklad v JSON:*

```
["ERROR", {"jobID":"_?#4","errType":"InvalidArgument",
"errMsg":"Job_id is invalid!"}]
```

Příkaz RELOADAGENTS

- *Syntaxe příkazu v ABNF:*

```
$RELOADAGENTS_COMMAND ::= "RELOADAGENTS"
```

- *Slovní popis:*
  Na tento bezparametrický příkaz by měl zareagovat správce zdrojů backendu. Reakcí by mělo být znovunačtení konfigurace pro výpočetní zdroje. Frontend tímto příkazem informuje backend, že došlo k její změně uživatelem.
- *Příklad v JSON:*

```
["RELOADAGENTS"]
```

Příkaz GETAGENTS

- *Syntaxe příkazu v ABNF:*

```
$GETAGENTS_COMMAND ::= "GETAGENTS"
```

- *Slovní popis:*
  Tímto příkazem frontend žádá backend o sdělení seznamu nakonfigurovaných a spuštěných agentů pro výpočetní zdroje.
- *Příklad v JSON:*

```
["GETAGENTS"] => ["agent1","agent2"]
```

Touto základní sadou příkazů by měly být pokryty všechny vzájemné komunikační potřeby frontendu a backendu u finálního řešení.

4.2 Komunikační middleware

Po té, co bylo zadefinováno aplikační rozhraní, je potřeba poohlédnout se po řešení, které by zajišťovalo vlastní předávání zpráv z jednoho bodu do druhého. Navíc je předpokládána potřeba vnitřní komunikace (na způsob RPC) mezi jednotlivými komponentami backendu v rámci jednoho procesu. Ideální řešení by tedy mělo být uplatnitelné jak pro vnější, tak pro vnitřní komunikaci. Jedním z takových řešení je **ZeroMQ**.

4.2.1 ZeroMQ

ZeroMQ (neboli Zero Message Queue) je v současné době velmi populární a aktivně vyvíjenou knihovnou fungující jako MOM (Message Oriented Middleware), tedy middleware na zasílání zpráv. Výraz „Zero“ v názvu znamená více věcí. Prvním významem je, že narozdíl od většiny jiných middleware řešení neobsahuje žádný broker (centrální prvek zajišťující výměnu zpráv mezi dvěma body). Dalšími významy jsou latence blížící se nule, nulová údržba a náklady.[20] Knihovna je multiplatformní, napsaná v jazyce C++. K dispozici jsou však mapování do téměř všech myslitelných jazyků. Mapování pro Python je dostupné pod názvem **„pyzmq“**.

Knihovna poskytuje vývojářům sockety s rozhraním vycházejícím ze standardních BSD socketů[1]). Oproti nim však sockety ZeroMQ lépe využívají dostupný hardware a přináší možnost vytvářet různé komunikační vzory. Dalším rozdílem je, že socket může být připojen k více než jednomu socketu. ZeroMQ podporuje nejen propojení přes standardní TCP a UDP ale také přes prostředky OS, jakými jsou např. pojmenované roury (protokol „IPC“[2])) nebo přes sdílenou paměť v rámci jednoho procesu (protokol „inproc“). Následují krátce shrnuté komunikační modely, které lze s ZeroMQ snadno implementovat a které by mohly být pro aplikaci užitečné.

Request – Reply

Sockety propojené v tomto režimu představují klasickou architekturu **klient-server**. Komunikace probíhá striktně synchronně. Server má na své straně socket příjmající požadavky a zasílající odpovědi. Ten je vytvořen několika málo následujícími řádky.

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://127.0.0.1:1234")
request = socket.recv()
# Do something with request...
socket.send(reply)
```

V tuto chvíli je vytvořen socket typu „Reply“, který poslouchá na portu „1234“. Klient je pak vytvořen obdobně, pouze se socketem typu „Request“:

```
socket = context.socket(zmq.REQ)
socket.connect("tcp://127.0.0.1:1234")
socket.send("some request...")
reply = socket.recv()
```

[1]) http://cs.wikipedia.org/wiki/Berkeley_sockets

[2]) Inter-process communication

Push – Pull

Tento model se uplatňuje při takzvaném *„pipeline"* zpracování, sběru zpráv z několika socketů. Socket na jedné straně pouze „tlačí" zprávy ven a socket na druhé straně tyto zprávy pouze odebírá.

```
puller = context.socket(zmq.PULL)
#..bind puller to some address..
pusher = context.socket(zmq.PUSH)
#..connect pusher to puller..
pusher.send("message")
message = puller.recv()
```

Publish – Subscribe

Tento režim reprezentuje klasický komunikační model **producent-konzument**. Producent rozesílá broadcastem zprávy s různými tématy. Konzument se může přihlásit k odběru vybraných témat (provést subskripci), viz krátká ukázka:

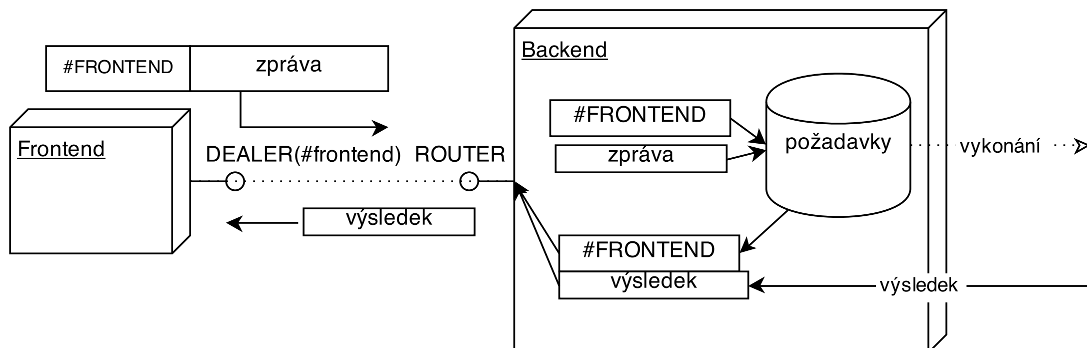
```
producent = context.socket(zmq.PUB)
#..bind producent..
konzument = context.socket(zmq.SUB)
konzument.setsockopt(zmq.SUBSCRIBE, "tema")
#..connect konzument..
producent.send("jinetema zprava")
producent.send("tema zprava2")
konzument.recv() # precte "zprava2"
```

Dealer – Router

Tento model je speciálním případem modelu Request – Reply. Narozdíl o něj zde komunikace probíhá asynchronně. ZeroMQ socket ve skutečnosti posílá před každou zprávou ještě hlavičku s identifikátorem socketu.

U předchozích typů socketů toto nešlo postřehnout, zpracovávají hlavičku interně a ven předávají pouze obsah zprávy. Dealer a Router jsou tzv. „raw" sockety, které tuto hlavičku předávají mimo knihovnu spolu se zprávou.

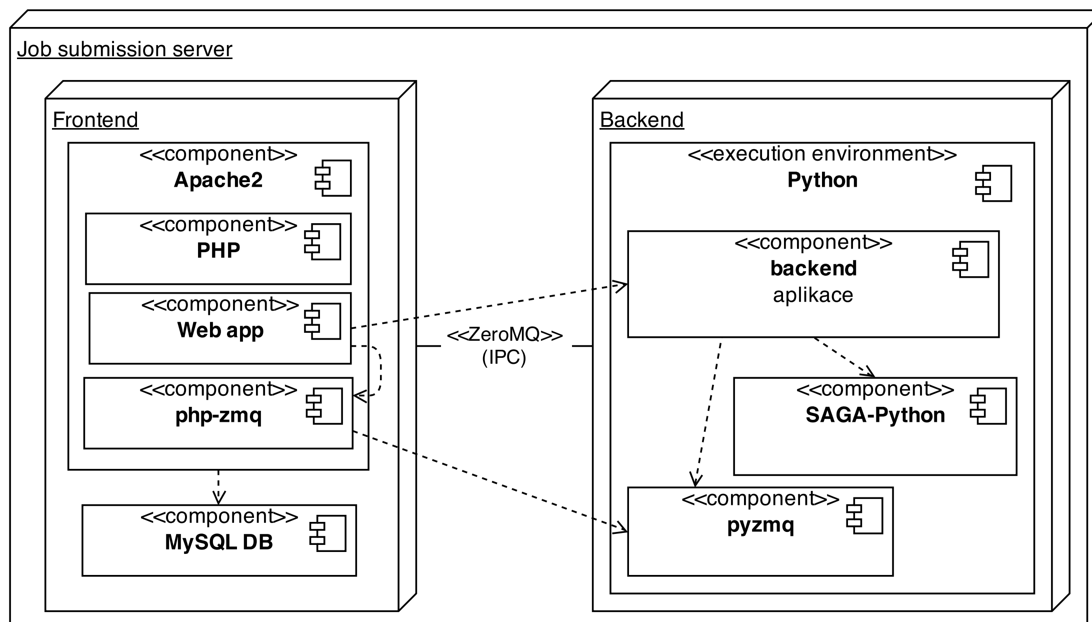
Tento model se bude výborně hodit pro účely rozhraní mezi backendem a frontendem. Je zde totiž potřeba zasílat příkazy obousměrně a asynchronně. Router socket je schopen v libovolný okamžik zasílat zprávy na Dealer socket s daným identifikátorem a zároveň i zprávy přijímat. Na straně Routeru je možné uložit si identifikátor socketu, ze kterého požadavek přišel pro účely pozdějšího zaslání odpovědi. V okamžiku, kdy jsou k dispozici výsledky, backend vyhledá k výsledkům identitu socketu, na který je má zaslat. Dealer může odeslat několik požadavků za sebou na Router socket, aniž by musel čekat po každém na odpověď Routeru. Celý tento proces je znázorněn na obrázku 4.1.



Obrázek 4.1. Komunikační vzor Router – Dealer.

4.3 Diagram nasazení

V tuto chvíli jsou již známy všechny potřebné technologie, které budou použity a je možné sestavit diagram nasazení pro výsledné řešení.



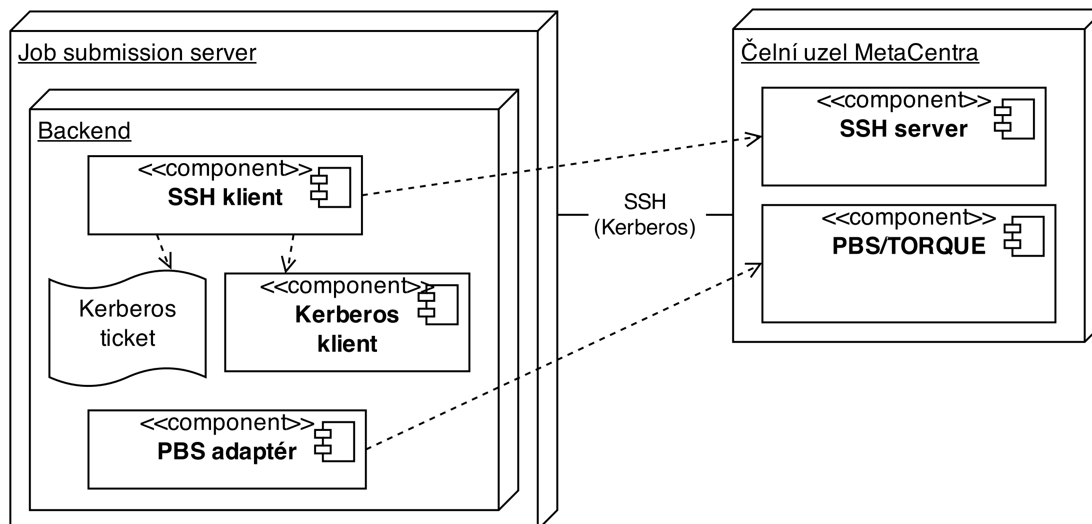
Obrázek 4.2. Diagram nasazení pro server.

Dle diagramu nasazení je potřeba vyhradit jeden stroj, který bude pro uživatele sloužit jako server pro zadávání a ukládání úloh. Na tomto stroji je potřeba nasadit klasický LAMP stack (webový server Apache2, MySQL databázový server a modul PHP).

Na web serveru poběží webové rozhraní vyvíjené kolegou Pavlem Novákem. To k realizaci komunikace s backendem vyžaduje rozšíření ZeroMQ pro jazyk PHP pod názvem „php-zmq“.

Jako komunikační protokol mezi frontendem a backendem byl pro ZeroMQ zvolen IPC (komunikace prostřednictvím pojmenované roury). Toto se jeví na linuxovém systému jako nejrychleší varianta komunikace mezi dvěma procesy.

Backend část běží kompletně v interpretu jazyka Python. Backendová aplikace v sobě obsahuje všechny komponenty potřebné pro správu úloh a zdrojů. Ke své funkci využívá pouze knihovny „SAGA-Python“ a „pyzmq“.



Obrázek 4.3. Diagram nasazení pro MetaCentrum.

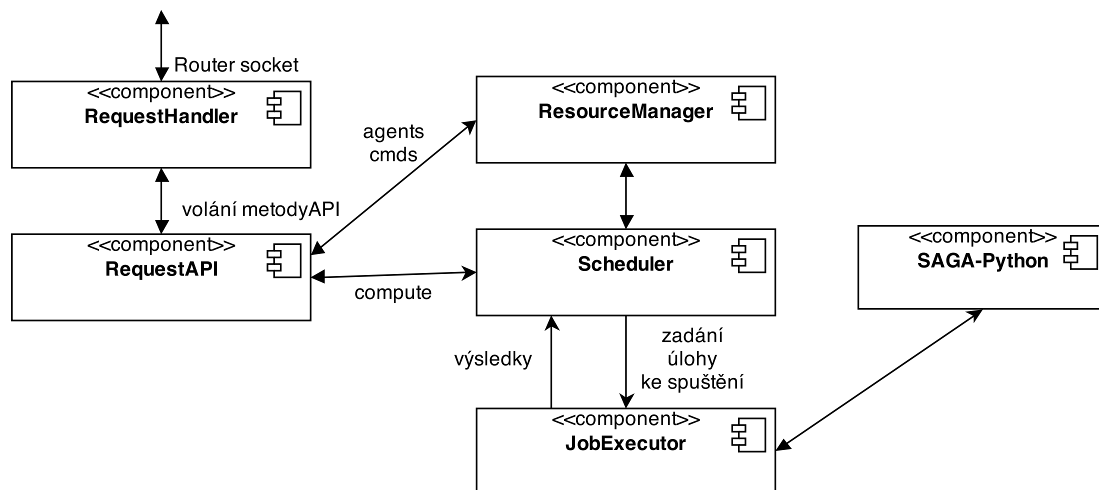
Na obrázku 4.3 je znázorněn externí pohled na server v případě použití s MetaCentrem. Komponenty „SSH klient“ a „PBS adaptér“ jsou oboje součástí knihovny SAGA-Python, která zde již není znázorněna.

Vzhledem k tomu, že MetaCentrum řeší autentizaci uživatelů systémem Kerberos, na serveru je potřeba nainstalovat klientské nástroje pro Kerberos. K prokázání se na čelním uzlu musí mít server platný kerberovský lístek (ticket). Ten lze periodicky obnovovat příkazem *„kinit“*, jež je součástí klientských nástrojů.

Diagram pro použití s obyčejnými výpočetními uzly zde není třeba uvádět. Jednalo by se o jednodušší variantu obrázku 4.3, kde by chyběly komponenty Kerbera a PBS.

4.4 Diagram komponent

V této podkapitole je rozvrhnuta budoucí vnitřní architektura aplikace backendu. Jak již bylo probráno v kapitole 3.7, tato aplikace musí obsahovat komponentu vlastního plánovače úloh a správce zdrojů. Dále bude nezbytná komponenta pro realizaci rozhraní mezi backendem a frontendem.



Obrázek 4.4. Interakce mezi komponentami aplikace.

Komponenta **RequestHandler** provede bind Router socketu a čeká na příchozí příkazy od frontendu. Vzhledem k tomu, že tyto příkazy obdrží ve formátu JSON, musí je převést do jejich reprezentace v Pythonu.

Následně je určeno, o jakou metodu **RequestAPI** se jedná, a tato metoda je zavolána. Metody aplikačního rozhraní mezi frontendem a backendem mohou na backendu interagovat pouze s dvěma komponentami – plánovačem úloh správcem zdrojů.

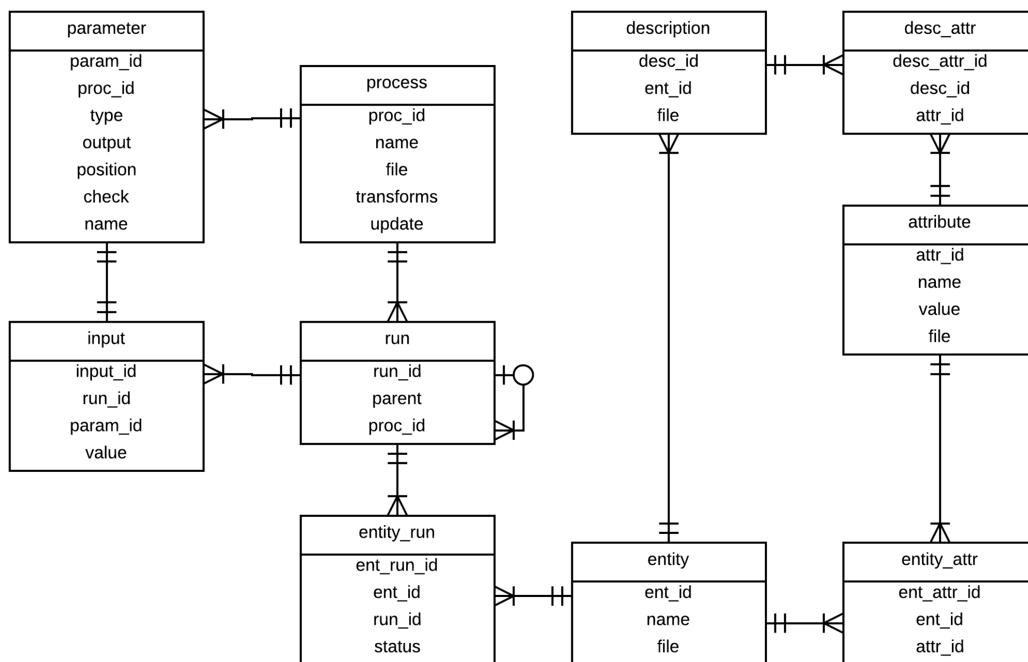
Plánovač úloh zjišťuje ze správce zdrojů aktuální nakonfigurované zdroje a jejich stav. Dále vybírá, na který zdroj úlohu poslat. Úlohu, spolu s informací o vybraném zdroji předává komponentě **JobExecutor**.

JobExecutor využívá knihovny SAGA-Python za účelem zadání úlohy k výpočtu a monitoringu jejího stavu. Zároveň je jeho zodpovědností nakopírování souborů pro úlohu (stage-in a stage-out).

4.5 E-R model databáze

Poslední částí této kapitoly je návrh schématu databáze, v níž budou data týkající se úloh na serveru uloženy. Tato podkapitola je zde poněkud bokem a je nutné hned na začátku upozornit na fakt, že tato databáze vznikala ve spolupráci s Pavlem Novákem (viz 2.2 a zadání této práce).

Tato databáze bude primárně sloužit k ukládání výpočetních úloh, dále pak zdrojových a výstupních dat těchto úloh.



Obrázek 4.5. E-R diagram databáze úloh.

Z hlediska backend aplikace jsou zajímavé především entity *„process“*, *„run“*, *„parameter“* *„input“* a *„entity_run“*. Entita *„process“* obsahuje informace o dostupných skriptech, jež jsou spouštěny na výpočetních uzlech.

Entita *„entity_run“* má význam konkrétní výpočetní úlohy (jednoho konkrétního běhu výpočtu nad konkrétními daty). Pro daný atribut *„ent_run_id“* je vybrán běh s konkrétním procesem (skriptem) a parametrům tohoto skriptu jsou pro daný běh přiřazeny konkrétní hodnoty z tabulky *„input“*. Data, nad kterými má proces počítat, jsou nalezena v tabulce *„entity“* přes atribut *„ent_id“*.

Výsledky výpočtů jsou shromažďovány ve zbývajících tabulkách. Ukládají se zde výpočtem vzniklé popisy obvodů a jejich vlastnosti (atributy).

Kapitola 5
Implementace

Poté, co byla v předchozí kapitole zadefinována celková architektura požadovaného systému, bude v této kapitole následovat implementace jednotlivých částí za použití výše popsaných technologií a postupů.

Při implementaci bude použita standardní implementace Pythonu CPython. Vzhledem k tomu, že aplikaci bude potřeba strukturovat do více vláken, je dobré v tomto ohledu upozornit na jedno omezení. Implementace CPython má takzvaný GIL[1]) (Global Interpreter Lock), který zabraňuje spouštění pythonovských vláken na více než jednom jádru procesoru výměnou za větší thread-safety. Nicméně uživatel si může na server nainstalovat implementaci Pythonu, která toto omezení nemá (např. Jython nebo IronPython) a aplikaci pustit v jejím interpretu. To zlepší výkon aplikace, ale může to také vést k nestabilitě a problémům.

Díky použitému frameworku ZeroMQ, který značně usnadňuje komunikaci mezi vlákny, si lze dovolit všechny důležité komponenty aplikace umístit do vlastních vláken.

5.1 Konfigurace

Před zahájením implementace vlastních aplikačních komponent byla vytvořena komponenta schopná načítat konfigurační volby z konfiguračního souboru, který je instalován do cesty „/etc/datamorph/datamorph.conf“. V dalším textu, a zejména ve výpisech kódu, jsou proměnné s velkými písmeny v názvu načítané právě z tohoto konfiguračního souboru.

5.2 Rozhraní backend-frontend

Jak bylo popsáno v kapitole 4, backend bude ve výchozím stavu přijímat požadavky na jednom z konců pojmenované roury. Toto má podle diagramu 4.4 na starost komponenta **RequestHandler**. Komunikační jádro této komponenty je popsáno následujícím úryvkem kódu.

[1]) `https://wiki.python.org/moin/GlobalInterpreterLock`

```
self._responder = self._context.socket(zmq.ROUTER)
self._responder.bind("%s://%s" % (CTRL_PROTO, CTRL_ENDPOINT))
...
poll = util.createPoller([self._responder, self._collector])
while True:
  socks = dict(poll.poll())
  if self._responder in socks and socks[self._responder] == zmq.POLLIN:
    self._handleRequest(self._responder.recv_multipart())
```

Z kódu uvedeného výše je vidět, že typ použitého socketu je ROUTER a následně tento socket provede bind na protokol, adresu a port uvedený v konfiguračním souboru. Novinkou je zde funkce „createPoller“. Ta vytváří zařízení zmq.Poller a zaregistruje mu předané sockety. Toto zařízení pak nekonečném cyklu kontroluje, zdali na libovolný zaregistrovaný socket nepřišla zpráva.

Pokud přijde zpráva od frontendu, je předána metodě „handleRequest“ jako pole, kde prvním prvkem je identifikátor socketu frontendu a dalšími prvky samotný obsah zprávy.

Metoda handleRequest, místo aby volala požadovanou metodu z RequestAPI, která se může potencionálně dlouho zpracovávat, zadá spuštění této metody některému z RequestWorkerů. RequestWorkery implementují rozhraní RequestAPI a běží ve svém vlastním vlákně. Jejich počet se dá ovlivnit z konfiguračního souboru. Distribuce požadavků na tyto workery probíhá pomocí modelu „PUSH–PULL“. RequestHandler k tomu využívá tyto sockety:

```
self._distributor = self._context.socket(zmq.PUSH)
self._collector = self._context.socket(zmq.PULL)
...
while True:
  socks = dict(poll.poll())
  ...
  elif self._collector in socks and socks[self._collector]==zmq.POLLIN:
    data = self._collector.recv_multipart()
    self._responder.send_multipart(data)
...
def _handleRequest(self, request):
   ...
   self._distributor.send_multipart(
     [originator, cmd, pickle.dumps(request[2:])])
```

Collector poté sbírá výsledky z Workerů. Ty jsou pak jednoduše přeposlány frontendu.

5.3 Scheduler

Scheduler je opět komponenta běžící ve svém vlastním vlákně. Po svém spuštění začne v podobné POLL smyčce, jako v ukázkách výše, čekat na úlohy

od RequestWorkerů a výsledky provádění úloh od JobExecutorů. Poté, co od některého RequestWorkera obdrží popis úlohy, začne proces plánování na zdroj. Scheduler v rámci tohoto procesu udělá následující kroky:

- Zařadí úlohu do seznamu zpracovávaných úloh.
- Změní stav úlohy na *„queued“* a informuje o tomto RequestWorkera.
- Pokud má úloha preferovaný zdroj, naplánuje jí na něj. Jinak pokračuje dál.
- Podle požadavků úlohy vybere z dostupných zdrojů vyhovující kandidáty.
- Pro tyto kandidáty spočítá skóre. Výše získaného skóre je závislá na naměřených metrik (zatížení, výkonu, spolehlivosti) pro daný zdroj.
- Je vybrán zdroj s nejvyšším skóre.
- Změní stav úlohy na *„scheduled“* a informuje RequestWorkera.
- Aktualizuje metriku „last_job_assignment_time“ na vítězném zdroji.
- Je spuštěno vlákno JobExecutor, kterému se předá úloha a zdroj, na kterém má být spuštěna.

Výsledné skóre zdrojů při výběru úlohy se skládá z více složek, kde každá má uživatelem nastavitelnou váhu. Kalkulaci skóre ilustruje následující úryvek kódu:

```
def _calculateScore(self, job, resourcename, jobsize):
  stats = self._resources[resourcename].getStats()
  score = 0.0
  # Score for reliability
  rel = (float(stats['jobs_processed'] - stats['jobs_failed'])/
    (stats['jobs_processed']))*WEIGHT['reliability']

  # Least recently used based score
  lru = (float(time.time() - stats['last_job_assignment_time'])/
    (60*15))*WEIGHT['last_job_assignment_time']

  # Resource load and performance based score
  perf = float(WEIGHT['exec_time_avg'])/(stats['exec_time_avg'])
  perf += float(WEIGHT['preparing_jobs'])/(stats['preparing_jobs'])
  perf += float(WEIGHT['executing_jobs'])/(stats['executing_jobs'])
  perf += float(WEIGHT['unstaging_jobs'])/(stats['unstaging_jobs'])

  # Approx. filetransfer speed based score
  xfer = (WEIGHT['transfer_speed_avg']*stats['transfer_speed_avg'])/
    (jobsize)

  return int(rel + lru + perf + xfer)
```

Aby naměřené statistiky zdrojů přežily ukončení aplikace, jsou při ukončení uloženy na disk a při startu aplikace opětovně načteny. Pokud plánovač obdrží od JobExecutoru zprávu, že úloha selhala, pokusí se o její přeplánování (restart) na jiný zdroj. Počet takových pokusů je omezený a opět konfigurovatelný. Pokud je počet pokusů o přeplánování překročen, zahlásí plánovač chybu

na RequestWorker. Ta je poté vypropagována až na frontend. Mechanismus přeplánování je implementován následujícím způsobem:

```
def _handleExecutorResponse(self):
  jid, response = pickle.loads(self._sockets['executors'].recv())
  task = self._tasks[jid]
  if isinstance(response, Exception) and task['try'] < RESCHEDULE_TRIES:
    self._rescheduleJob(jid)
...
def _rescheduleJob(self, jid):
  task = self._tasks[jid]
  self._tasks[jid]['try'] += 1
  badResource = task['resource']
  self._scheduleJob(task['jobData'], [badResource])
...
def _scheduleJob(self, job, excludingResources=[]):
```

5.4 JobExecutor

Tato komponenta je spouštena plánovačem po naplánování úlohy na konkrétní zdroj. Má za úkol provést úlohu fázemi přípravy ke spuštění, naplánování na zdroj a spuštění, běhu, ukončení a sběru výsledků. JobExecutor hojně využívá funkcí knihovny SAGA. Fáze přípravy úlohy vypadá následovně:

```
self.session  = saga.Session()
jDescription, sagaJob = self._prepareSagaJob()
endpoint     = self.resource.getServiceEndpoint()
xfer         = self.resource.getXferBackend()
serviceHost = '%s://%s' % (xfer, endpoint)
jobBaseDir  = '%s%s' % (serviceHost, jDescription.working_directory)
if CACHING_ENABLED:
  cacheDir = self.resource.getCacheDir()
  self._stagein(jobBaseDir, cacheDir)
else:
  self._stagein(jobBaseDir)
```

Nejprve se vytvoří session, poté se připraví popis úlohy použitelný knihovnou SAGA. Nakonec se provede stagein souborů na zdroj (pokud již nejsou v cache). O souborové cache bude pojednávat část 5.6.1.

Další fází je spuštění úlohy a čekání na výsledky:

```
sagaJob.run()
self._changeJobState(state['executing'])
log.info('Running #%s under SID #%s' % (self.job.jid, sagaJob.id))
sagaJob.wait()
```

Funkce „wait()“ blokuje až do skončení úlohy. Poté je proveden stageout výsledků a úklid pracovního adresáře úlohy. Výsledky jsou nakonec zaslány plánovači.

5.5 ResourceManager

Poslední důležitou komponentou aplikace je správce zdrojů. Jeho starostí je kontrola dostupnosti a stavu zdrojů a poskytování seznamu aktuálně dostupných zdrojů plánovači úloh.

Při startu se o strojích, které má spravovat, dozvídá z konfiguračního souboru „agents.cfg“. Tento soubor načte a nakonfiguruje podle něj poskytované zdroje (ResourceAgenty). Konfigurační soubor obsahuje například informace o tom, jakou přístupovou metodu a systém pro zadávání úloh daný zdroj používá. Další informace se vztahují k vlastnostem zdroje jako jsou operační systém, architektura či nainstalované nástroje připravené k použití úlohami. Konfigurační soubor může být měněn i prostřednictvím webové aplikace.

5.6 Pomocná infrastruktura

5.6.1 Cache souborů

Za účelem co nejvíce zkrátit dobu přenosu vstupních souborů a zabránit opakovanému nahrávání stejných souborů na zdroj byla implementována jednoduchá cache souborů na zdrojích. Každý ResourceAgent má k dispozici cache (instanci FileCache) obsahující soubory nacházející se na zdroji. Pokud přijde úloha, která chce nahrát soubor, který je již v cache, použije se pro ní soubor z cache. Tím dojde pouze k lokálnímu kopírování v rámci zdroje, což je násobně rychlejší než přenos po síti.

5.6.2 Notifikace

Všechny komponenty, které běží ve svém vlastním vláknu (dědí třídu ZmqThread), jsou schopny přijímat notifikace odeslané kteroukoli jinou takovou komponentou. Komunikační model byl zvolen PUBLISH-SUBSCRIBE, takže je možné odebírat pouze určené notifikace. Tato funkce je v aplikaci hojně využívána pro notifikace o změně stavu úloh.

5.6.3 RPC proxy

Komunikace mezi některými komponentami aplikace pomocí zasílání zpráv či notifikací by byla značně nepraktická. Kvůli tomu byla vyvinuta nad sockety ZeroMQ funkcionalita vzdáleného volání metod.

Komponenta pak volá metody na proxy objektu jiné komponenty. Proxy objekt je vytvořen vytvořením instance třídy „RpcProxy“ v souboru „objproxy.py“.

5.7 Struktura projektu

Projekt je strukturován podle pravidel pro tvorbu Python balíčků:

```
.
|-- bin
|   `-- datamorphd      /* Daemon s aplikací */
|-- config
|   |-- agents.cfg      /* Konfigurace agentů výpočetních zdrojů*/
|   `-- datamorph.conf  /* Konfigurace aplikace*/
|-- datamorph
|   |-- __init__.py
|   |-- constants.py    /* Načítání konfigurace, konstanty */
|   |-- filecache.py    /* Cache souborů na zdrojích */
|   |-- jexecutor.py    /* JobExecutor */
|   |-- job.py          /* Reprezentace úlohy */
|   |-- logger.py       /* Vlastní logování */
|   |-- objproxy.py     /* RpcProxy */
|   |-- requestapi.py   /* RequestAPI */
|   |-- resman.py       /* ResourceManager */
|   |-- resource.py     /* Resource */
|   |-- rhandler.py     /* RequestHandler */
|   |-- rworker.py      /* RequestWorker */
|   |-- scheduler.py    /* Scheduler */
|   |-- util.py         /* Pomocné funkce pro parsování JSON */
|   `-- zmqthread.py    /* Thread komunikující přes ZeroMQ */
|-- setup.py            /* Instalátor balíčku */
|-- tests
|   |-- jobs
|   |   |-- bigdata
|   |   |   |-- 5xp1.blif
|   |   |   |-- required_files
|   |   |   |   `-- put_your_bigfile_here
|   |   |   |-- result
|   |   |   `-- run
|   |   `-- cpuheavy
|   |       |-- 5xp1.blif
|   |       |-- _description.txt
|   |       |-- required_files
|   |       |   |-- resynscript
|   |       |   |-- script
|   |       |   `-- total.awk
|   |       |-- result
|   |       `-- run
|   |-- templates.py
|   `-- testsuite.py
```

5.8 Testování

Pro účely testování aplikace je připraven skript „testsuite.py“, který umí simulovat všechny příkazy zadávané frontendem. K dispozici jsou připraveny dvě

šablony úloh - Big Data a Cpu Heavy. Další se dají snadno přidat do souboru „templates.py“.

Kapitola 6
Závěr

V rámci této diplomové práce byly nastudovány a analyzovány současné možnosti centralizované správy distribuovaných výpočtů. Jedna z těchto možností byla vybrána k vytvoření backendu umožňujícího provádět distribuovné výpočty. Dále byly prozkoumány možnosti národní gridové infrastruktury MetaCentrum, zejména z hlediska využití zdrojů výslednou aplikací.

Návrh backend aplikace byl založen na využití knihovny SAGA-Python pro distribuované výpočty a ZeroMQ pro implementaci komunikační vrstvy. Při využití těchto knihoven se ukázalo, že implementace klientské části aplikace (běžící na výpočetních zdrojích) není ke splnění účelu a funkčních požadavků potřeba.

Serverová část backendu podporuje plánování a spouštění úloh na většině platforem, v gridech i na individuálních strojích (jediným požadavkem je potřeba přístupu na stroj alespoň přes protokol SSH). Tyto úlohy jsou přijímány z frontendu (uživatelského rozhraní), který vytvořil Pavel Novák. Backend se dále stará o přesun dat spojených s úlohami na výpočetní uzly. Po skončení výpočtů pak zajišťuje přesun výsledků zpátky na server, kde jsou předány frontendu a uloženy na databáze. Součástí společného řešení s Pavlem Novákem byl pak návrh databáze pro uložení dat spojených s úlohami.

Pro testování a vývoj aplikace byly využity výpočetní zdroje DDD FIT ČVUT[1]) a MetaCentra.

Možnost rozšíření současného řešení vidím ve vylepšení komponenty plánovače. V současné době aplikuje jednoduchou politiku FIFO. Dále by plánovač mohl umět využít souborové cache, která byla naimplementována. Mohl by například ještě více optimalizovat přenosy souborů tím, že naplánuje úlohy na zdroje, které mají v cache soubory požadované úlohou.

[1]) `http://ddd.fit.cvut.cz/`

Literatura

[1] JACOB, Bart. *Introduction to grid computing* [online]. 1st ed. United States: IBM, International Technical Support Organization, 2005, xiv, 248 p. [cit. 2014-05-03]. ISBN 07-384-9400-3. Dostupné z:
http://www.redbooks.ibm.com/redbooks/pdfs/sg246778.pdf

[2] MOSCICKI, J.T. *Understanding and mastering dynamics in computing grids: processing moldable tasks with user-level overlay* [online]. 2011, viii, 178 p. [cit. 2014-05-04]. Dostupné z:
http://dare.uva.nl/document/212268. University of Amsterdam. Vedoucí práce M.T. Bubak.

[3] JANEČEK, Jan. *Distribuované systémy.* Praha: ČVUT, 2000, s. 6-9.

[4] KLIMEŠ, Cyril. *Distribuované systémy.* Ostrava: Ostravská univerzita v Ostravě. Dostupné z:
http://www1.osu.cz/~prochazka/ds/SkriptaKlimes.pdf

[5] Distributed computing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z:
http://en.wikipedia.org/wiki/Distributed_computing

[6] KAHANWAL a Tejinder PAL SINGH. The Distributed Computing Paradigm: P2P, Grid, Cluster, Cloud, and Jungle. In: RAJIV, Kumar. *Nternational Journal of Latest Research in Science and Technology* [online]. Vol. 1, Issue 2: MNK Publication, 2012 [cit. 2014-05-04]. ISSN 2278-5299. Dostupné z:
http://arxiv.org/pdf/1311.3070.pdf

[7] GT 5.2.5 GRAM5 Key Concepts. In: *Globus Toolkit* [online]. University of Chicago [cit. 2014-05-04]. Dostupné z:
http://toolkit.globus.org/toolkit/docs/5.2/5.2.5/gram5/key/gram5KeyConcepts.pdf

[8] Berkeley Open Infrastructure for Network Computing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-03]. Dostupné z:
http://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing

[9] ANDERSON, David. *BOINC: A System for Public-Resource Computing and Storage* [online]. University of California at Berkeley [cit. 2014-05-03]. Dostupné z:
https://boinc.berkeley.edu/grid_paper_04.pdf

[10] CORBATTO, M. *An introduction to PORTABLE BATCH SYSTEM* [online]. 2000 [cit. 2014-05-04]. Dostupné z:
http://hpc.sissa.it/pbs/pbs.html

[11] BAYUCAN, Albeaus, Robert L. HENDERSON, Lonhyn T. JASINSKYJ, Casimir LESIAK, Bhroam MANN, Tom PROETT a Dave TWETEN. *Portable Batch System: Administrator Guide* [online]. Release: 2.2. Mountain View, CA 94043: MRJ Technology Solutions, 1999 [cit. 2014-05-04]. Dostupné z:
https://www.jlab.org/hpc/PBS/v2_2_admin.pdf

[12] BLAISE, Barney. Using Moab. In: [online]. Lawrence Livermore National Laboratory [cit. 2014-05-03]. Dostupné z:
https://computing.llnl.gov/tutorials/moab/

[13] GT 5.2.5 GRAM5. GLOBUS. *Globus Toolkit* [online]. University of Chicago [cit. 2014-05-04]. Dostupné z:
http://toolkit.globus.org/toolkit/docs/5.2/5.2.5/gram5/#gram5

[14] Diane. CESNET. *MetaCentrum Wiki* [online]. 2012 [cit. 2014-05-01]. Dostupné z: https://wiki.metacentrum.cz/wiki/DIANE

[15] HNÍZDIL, Tomáš. *Pilot jobs* [online]. Brno, 2010 [cit. 2014-05-01]. Dostupné z:
http://is.muni.cz/th/255497/fi_b/. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Miroslav Ruda.

[16] DIANE Tutorial. MOSCICKI, Jakub. *DIANETutorial ArdaGrid TWiki* [online]. CERN, 2007 [cit. 2014-05-01]. Dostupné z:
https://twiki.cern.ch/twiki/bin/view/ArdaGrid/DIANETutorial

[17] RADICAL RESEARCH. *SAGA-Python: A Light-Weight Access Layer for Distributed Computing Infrastructure* [online]. The Cloud and Autonomic Computing Center, Rutgers University, 2013 [cit. 2014-05-01]. Dostupné z:
http://saga-project.github.io/saga-python/

[18] METACENTRUM. *Česká národní gridová infrastruktura.* Dostupné z:
http://www.metacentrum.cz/export/sites/metacentrum/downloads/PR/letak_NGI_tisk.pdf

[19] KŘENKOVÁ, Ivana, Tomáš REBOK, Aleš KŘENEK, Miroslav RUDA a Luděk MATYSKA. CESNET. *Yearbook 2011-2012: National Grid*

Infrastructure Annual Report [online]. 2012 [cit. 2014-04-21]. ISBN 978-80-904689-7-9. Dostupné z:
http://www.metacentrum.cz/export/sites/metacentrum/cs/about/results/yearbooks/2013_rocenka_web_mensi.pdf

[20] CRIPPA, Francesco. Europycon2011: Implementing distributed application using ZeroMQ. In: [online]. 2011 [cit. 2014-05-01]. Dostupné z:
http://www.slideshare.net/fcrippa/europycon2011-implementing-distributed-application-using-zeromq

Příloha A
Uživatelská příručka

A.1 Instalace

Instalace aplikace se provede příkazem:

```
python setup.py install
```

Instalátor by spolu s aplikací měl nainstalovat i všechny potřebné závislosti. Po instalaci se umístí konfigurační soubory do:

```
/etc/datamorph/datamorph.conf
/etc/datamorph/agents.cfg
```

A.2 Použití

Aplikace běží jako démon, ovládá se příkazy:

```
datamorphd [start|stop]
```

Tento démon loguje do souborů:

```
/var/log/datamorph/{log,errors}
```

Příloha B
Obsah přiloženého CD

```
.
|-- aplikace
|    |-- bin           /*Spustitelný soubor aplikace*/
|    |-- config        /*Konfigurační soubory*/
|    |-- datamorph     /*Zdrojový kód aplikace*/
|    |-- docs          /*Ukázková konfigurace zdrojů*/
|    |-- setup.py      /*Instalátor aplikace*/
|    `-- tests         /*Sada testů a testovací skripty*/
`-- text               /*Zdrojové soubory tohoto textu*/
```