TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Modul pro spouštění úloh na vzdálených výpočetních clusterech

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Jan Gabriel**
Vedoucí práce: doc. Ing. Jiřina Královcová, Ph.D.

Liberec 2016

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Software tool for running jobs on remote computer clusters

Diploma thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Jan Gabriel**
Supervisor: doc. Ing. Jiřina Královcová, Ph.D.

Liberec 2016

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2015/2016**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Gabriel**
Osobní číslo:	**M14000156**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Modul pro spouštění úloh na vzdálených výpočetních clusterech**
Zadávající katedra:	**Ústav mechatroniky a technické informatiky**

Zásady pro vypracování:

1. Seznamte se s prostředky aktuálně vyvíjeného systému GeoMop zaměřenými na serverovou komunikaci a obsluhu automaticky spouštěných úloh.
2. Navrhněte a implementujte prvky grafického uživatelského rozhraní pro konfiguraci a spouštění úloh na výpočetních clusterech a navrhněte součásti pro komunikaci uživatelského rozhraní se serverovou stranou s využitím existujících prostředků knihovny systému GeoMop.
3. Vytvořte prvky pro kontextové generování konfiguračních souborů v závislosti na vybraném komunikačním scénáři.
4. Na základě konzultace s vedoucím práce realizujte testy pro vybrané scénáře komunikace.

Rozsah grafických prací: **dle potřeby dokumentace**

Rozsah pracovní zprávy: **cca 40–50 stran**

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

Seznam odborné literatury:

[1] **CESNET/torque – GitHub [online]. 2015 [cit. 2015-10-08]. Dostupné z: https://github.com/CESNET/torque**

[2] **Overview – Python 3.4.3 documentation [online]. 2015 [cit. 2015-10-08]. Dostupné z: https://docs.python.org/3.4/**

[3] **SUMMERFIELD, Mark. Python 3: výukový kurz. Vyd. 1. Brno: Computer Press, 2010, 584 s. ISBN 978-80-251-2737-7.**

Vedoucí diplomové práce: **doc. Ing. Jiřina Královcová, Ph.D.**
Ústav mechatroniky a technické informatiky

Konzultant diplomové práce: **Ing. Pavel Richter**
Ústav nových technologií a aplikované informatiky

Datum zadání diplomové práce: **14. ledna 2016**

Termín odevzdání diplomové práce: **16. května 2016**

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

L.S.

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 14. ledna 2016

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 10. 5. 2016

Podpis: Gabriel

Poděkování

Rád bych poděkoval celému vývojářskému týmu aplikace GeoMop za trpělivost a vstřícnost projevenou během vývoje klientské aplikace. Dále pak doc. Ing. Jiřině Královcové, Ph.D. za cenné rady a odborný dohled při zpracování písemné části této práce.

Abstrakt

Tato práce se zabývá návrhem a implementací klientské části aplikace usnadňující provádění komplexních výpočetních úloh. Uplatnění si najde hlavně mezi tvůrci matematicko-fyzikálních modelů určených převážně pro paralelní zpracovaní na výpočetních clusterech. Uživatelé budou vhodným způsobem odstíněni od složitostí a konfigurace paralelních výpočetních systému a budou se tak moci plně soustředit pouze na svou práci. V případě požadavku na zpracování výsledku si uživatel jednoduše vybere z několika předkonfigurovaných scénářů jako například lokální počítač, dedikovaný server nebo gridová infrastruktura. Následně vloží potřebná data pro úlohu a spustí výpočet. Monitorování i vyzvednutí výsledku úlohy probíhá v klientské aplikaci a není třeba se starat o to, kde se výpočet fyzicky provedl. Aplikace vzniká jako jeden z dílčích celků projektu Presage, na kterém spolupracuje Technická univerzita v Liberci se specializovanou externí firmou.

Klíčová slova

Torque, PBS, Frontend, Cluster, Grid, MetaCentrum, Python, PyQt5

Abstract

This paper discusses the design and implementation of the client side application for facilitation and management simplification of the complex computing tasks execution. Usage is found primarily amongst makers of mathematical and physical models designed mainly for parallel processing on computer clusters. Users will be adequately protected from the complexity and configuration of computational systems, so they will be able to fully focus on their work. If the result of computation is required, user simply selects from several predefined scenarios such as a local computer or a dedicated server or grid infrastructure. Then inserts the data needed for the task execution and starts his calculation. Monitoring and results retrieving is done within client application, so there is no need to worry about where the calculation is done physically. Applications is created as one of the sub-units in Presage project, which based on cooperation between Technical University of Liberec and specialized external company.

Keywords

Torque, PBS, Frontend, Cluster, Grid, MetaCentrum, Python, PyQt5

Obsah

Seznam obrázků

Seznam ukázek kódu

Seznam zkratek a pojmů

NGI	Národní gridová infrastrukura
MetaCentrum VO	virtuální organizace, pro správu a přístup ke gridové infrastruktuře
Frontend	čelní uzel výpočetního systému, přes který se úlohy posílají ke zpracování
PBS	software, který řídí přiřazování zdrojů na výpočetním clusteru, známý též jako plánovač úloh (*Portable Batch System*)
MPI	knihovna implementující stejnojmennou specifikaci (protokol) pro podporu paralelního řešení výpočetních úloh na počítačových clusterech (*Message Passing Interface*)
Torque, SGE	jména nejčastěji používaných softwarů pro plánování úloh
qsub, qstat, qdel	základní příkazy plánovače úloh
GUI	grafické uživatelské rozhraní (*Graphical User Interface*)
CLI	příkazový řádek (*Command Line Interface*)
SSH	zabezpečený přístup pomocí příkazové řádky, především u Unixových systémů (*Secure Shell*)
SSD	pevný disk, který je navržen jako náhrada disků s pohyblivými částmi, vyznačuje se především podstatně vyššími přístupovými a přenosovými rychlostmi (*Solid State Drive*)
SCP	protokol pro kopírování souborů přes SSH (*Secure Copy*)
JSON	způsob zápisu dat (datový formát) nezávislý na počítačové platformě (*Javascript Object Notation*)
Python	pokročilý interpretovaný multiplatformní programovací jazyk
Qt	oblíbená knihovna pro tvorbu multiplatformních grafických rozhraní
PyQt	zapouzdření standardní multiplatformní knihovny Qt programovací jazyk Python

Úvod

Počítačové modelování a všemožné simulace jsou v současné době velice perspektivním oborem, který si snadno najde své uplatnění v nejrůznějších oblastech lidské činnosti. Se zvyšujícím se výpočetním výkonem běžných počítačů je nyní modelování mnohem dostupnější i běžným uživatelům a ne pouze specializovaným pracovníkům. To vede k intenzívnímu rozvoji v tomto oboru. Podpůrné nástroje jsou tak čím dál tím propracovanější a modely se více přibližují skutečnému světu. Takto přesné modely pak nacházejí nové uplatnění i v místech, kde by to jen před několika málo lety bylo nemyslitelné.

Například společnost Tesla Motors byla schopna převedením významné části takzvaných crash testů svého vozu do počítačové simulace odhalit mnoho bezpečnostních problémů již během návrhu. Tyto problémy pak byly zapracovány a opraveny ještě předtím, než byl vůz fyzicky vyroben. To umožnilo společnosti ušetřit nemalé množství finančních prostředků a dohnat své konkurenty, kteří i několik měsíců usilovně zkoušeli nejrůznější fyzické prototypy na testovací trati. Toto vynaložené úsilí se vyplatilo a výsledný vůz dostal nejvyšší možné bezpečnostní hodnocení hned v několika kategoriích. Lze tedy s trochou nadsázky tvrdit, že kvalitní modely a přesné simulace pomáhají chránit lidské životy a šetřit finanční prostředky při vývoji. I přes rozmanité možnosti využití a široké uplatnění mají tyto modely a simulace jedno slabé místo, pro jejich efektivní zpracování je často potřeba velké množství výpočetního výkonu. Obzvláště pro složitější výpočty si rozhodně nevystačíme se stolním počítačem a je třeba využít například výpočetní cluster, který požadovaného výkonu dosahuje vhodnou paralelizací úlohy.

I zde na Technické univerzitě se někteří zaměstnanci zabývají modelováním. Část z nich se pak více soustředí na modelování procesů, které se odehrávají při prostupu nejrůznějších chemických látek skrze horninové prostředí. To je dobré například ke studiu lokalit, vhodných pro dlouhodobé skladování radioaktivních odpadů. Může se jednat o poměrně náročné výpočty a často tedy narážíme na problémy s jejich efektivním zpracováním. V současné době se výpočty většinou zpracovávají na výpočetních clusterech, pro potřeby univerzity pak obvykle na univerzitním clusteru.

Mluvíme-li o clusteru, jedná se zpravidla o skupinu sítí propojených počítačů s unixovým operačním systémem. Na něm pak běží software, určený k plánování paralelních výpočtů. Tento software, zvaný též plánovač úloh, i přes svoji vnitřní složitost poskytuje uživatelům pouze základní funkce pro řízení a plánování úloh. Ty jsou navíc přístupné jen z příkazového řádku, což může být zejména pro nezkušené uživatele poměrně velkou překážkou.

Pro jakékoliv zpracování výsledků je totiž třeba věnovat velké úsilí konfiguraci systému, přenášení dat a celkové režii okolo zpracování výpočtu. Tyto přípravné procesy je navíc nutné neustále dokola opakovat a některých dalších žádaných vlastností lze dosáhnout jen velmi obtížně. Například pokročilejší sledování průběhu úlohy je velice problematické a řetězení výpočtů na základě výsledků předchozích úloh není možné téměř vůbec.

Všechny tyto problémy rozhodně nejsou specifické pouze pro náš konkrétní případ a jejich úspěšné vyřešení by ulehčilo práci mnoha uživatelům výpočetních systémů. Jako odpověď na tyto a mnohé další problémy začala vznikat aplikace GeoMop, která má pod hlavičkou projektu Presage sloužit jako podpůrný nástroj simulátoru Flow123d. Celá aplikace je rozdělena do několika dílčích kontextů. Tyto kontexty se zabývají různými problémy od samotného modelování přes specifikaci konfiguračních souborů, parametrizaci úloh až po automatizaci výpočtů na vzdálených výpočetních systémech.

Já jsem se v minulém roce v rámci diplomového projektu zabýval kontextem Jobs a zkoumal jsem možnosti zpracování výpočtů na různých serverových i lokálních konfiguracích. Na tomto základě začala v rámci kontextu vznikat serverová část aplikace. V této práci jsem tedy plynule navázal, ale zaměřil jsem se spíše na klientskou aplikaci. Ta má za úkol zmíněnou serverovou část ovládat. Výsledná klientská aplikace by měla uživateli v přehledných nabídkách poskytovat všechna potřebná nastavení k pohodlnému zadávání úloh a konfiguraci výpočetních systémů. Dále by měla zajišťovat efektivní komunikaci se vzdálenými servery a poskytovat uživateli rozhraní ke snadnému ovládání úloh a monitorování jejich průběhu. S problematikou celého kontextu Jobs a následným vývojem takovéto aplikace se seznámíme podrobněji na následujících stránkách.

1 Problematika

V této kapitole prozkoumáme podrobněji problematiku nastíněnou v úvodu. Rozebereme současný stav situace a podíváme se na konkrétní požadavky pro kontext Jobs. Dále pak více nahlédneme do problematiky paralelních výpočtů a vzdálených výpočetních systémů Upřesníme si, jaká omezení z jejich použití vyplývají a jak se promítnou do výsledné klientské aplikace. I když se budeme v dalších kapitolách zabývat převážně klientskou aplikací, je pochopení těchto serverových technologií a jejich omezení velice důležité, všechny další aspekty výsledné práce se od nich totiž odvíjejí.

1.1 Současný stav

Jak již bylo zmíněno v úvodu, na Technické univerzitě v Liberci máme určitou skupinu pracovníků, zabývajících se modelováním. V tuto chvíli jsou jejich výpočetní úlohy odesílány na univerzitní cluster manuálně a konfigurace sytému je prováděna přes ručně psaný konfigurační skript.

Každý, kdo tedy potřebuje cokoliv odeslat ke zpracování, musí být schopen napsat si vlastní konfigurační skript v závislosti na použitém plánovací softwaru. Dále pak musí zvládnout nahrát na server (obvykle přes protokol **SCP**) všechny důležité soubory pro běh úlohy a případně ještě nastavit programy, potřebné k jejímu spuštění. Se znalostí příkazů plánovače úloh, stačí už jen odeslat výpočet ke zpracovaní a po jeho dokončení vyzvednout výsledky.

Celý systém je přístupný pouze přes **SSH** a uživatel tedy musí být schopen celou úlohu obsloužit skrze příkazovou řádku (takzvané **CLI**). Situace se navíc komplikuje, pokud uživatel pracuje na operačním systému Windows a je nucen si nainstalovat potřebné aplikace třetích stran (pro Windows například aplikace *WinSCP* a *PuTTY*). Uživatelé si tak musí cestou k požadovanému výsledku projít celou řadou poměrně složitých úkonů. Tyto úkony je navíc nutné s každou úlohou neustále opakovat. Přitom se jedná pouze o obsluhu, která ve skutečnosti uživatele zdržuje od jejich skutečné práce. Otázkou tedy je, jak tento proces uživatelům usnadnit a zpříjemnit?

1.2 Paralelní výpočty

Ve světě informačních technologií je paralelizace velice zásadní. V posledních letech se ukazuje, že výkon nelze dostatečně rychle a efektivně zvyšovat nárůstem výkonu jednotlivých výpočetních jednotek. Narážíme totiž na fyzikální limity pro taktovací frekvence. Proto je trendem spíše zvyšování počtu těchto jednotek v rámci jednoho systému. Například běžný telefon má v současnosti běžně dvou a více jádrový procesor. Nicméně nelze jednoznačně říci, že by zdvojnásobením počtu výpočetních jednotek vzrostl dvojnásobně i výsledný výkon. Této problematice se více věnuje například Amhdálův zákon [1].

Na stejném principu fungují paralelní počítače, v určitém čase máme procesor o nějakém konkrétním výkonu. K dosažení obrovského výkonu je tedy jedinou možností použít takovýchto procesorů řádově třeba stovky až tisíce. Efektivita tohoto přístupu se však do velké míry odvíjí od druhu úlohy, kterou chceme řešit. Pokud bychom uvažovali výpočet, který není z principu paralelní, nikdy nedosáhneme vyššího výkonu než na jedné výpočetní jednotce. Jednoduše proto, že ostatní jednotky nebude možné při výpočtu použít vůbec.

Úlohy tedy musí být takzvaně dobře paralelizovatelné. Znamená to, že celkovou úlohu lze rozdělit na několik menších podúloh a později lze celkový výsledek složit z výsledků těchto podúloh. V reálném světe je takovouto úlohou například natírání plotu, kde můžeme při zanedbání jistých omezení zvyšovat počet natěračů a celková práce bude hotová o to rychleji. Opakem jsou úlohy, jejichž výsledky jsou závislé na výsledku těch předchozích. Tedy analogicky například oprava počítače, zde má každý krok jisté pořadí a nelze všechny kroky vykonávat naráz. Podrobněji lze danou problematiku nastudovat například v příručce **MetaCentra VO** [2].

1.2.1 Superpočítač

Jako první si definujeme **superpočítač**. Jedná se v podstatě o jeden velice výkonný počítač, který má velké množství procesorů, diskového prostoru a operační paměti. Použité procesory však nemusejí být nutně výkonnější než ten u obyčejného stolního počítače, za to jich je v systému opravdu velké množství. Použité procesory navíc mohou mít nějakou speciální architekturu, vhodnou pro určitý typ úlohy, například vektorové či maticové výpočty. Všechny tyto procesory jsou propojeny velmi rychlou vnitřní sítí s minimálním zpožděním a sdílejí přístup do společné paměti. Kvůli této architektuře je nutné mít hardware co nejpodobnější, tím je možné zajistit superpočítači obrovský výpočetní výkon. Nároky na komunikaci a sdílení paměti však často značně komplikují snahy o další rozšiřování.

1.2.2 Cluster

Cluster má naopak mnohem volnější vnitřní strukturu než superpočítač. Celek se skládá z takzvaných uzlů, což jsou unixové počítače propojené rychlou místní sítí. Každý uzel je tak nezávislým počítačem, který může mít dokonce rozdílnou hardwarovou konfiguraci. Společnou synchronizaci uzlů zajišťuje operační systém se softwarovou nadstavbou, ten spolupracuje s plánovacím softwarem, který se stará o rozložení úloh do jednotlivých uzlů. Cluster je díky volnějšímu provázání počítačů možné jednoduše rozšiřovat přidáváním dalších uzlů. To je obvykle možné přímo za běhu a pokud je systém navržen dostatečně robustně, není nijak zásadní ani selhání určité části uzlů. Uživatel to obvykle nepocítí a jeho úlohy se rovnoměrně rozloží mezi ostatní uzly. Rozšiřování má však určité limity, jejichž dosažením se stává vzájemná komunikace tak náročnou, že přidávání dalších uzlů již neposkytuje téměř žádný nárůst výpočetního výkonu.

1.2.3 Grid

Posledním pojmem je **grid**, ten do jisté míry vychází z celkového konceptu clusteru. Na rozdíl od clusteru jsou ale jednotlivé počítače volně vázané, heterogenní a mohou být rozmístěny v různých zeměpisných lokacích. Grid tedy můžeme chápat jako skupinu nezávislých počítačů, spolupracujících současně na různých částech stejné úlohy. Jejich vzájemná komunikace je zajištěna pomocí zabezpečeného kanálu skrze privátní síť nebo v rámci internetu. Kvůli tomuto rozložení a komunikační struktuře je grid vhodný zejména na paralelní úlohy, které jsou velmi nezávislé a snadno se dělí na menší části. V prostředí MetaCentra se organizace do gridu využívá pro efektivní řízení zdrojů a sjednocení přístupu k těmto zdrojům. Clustery jednotlivých organizací jsou propojeny do jednoho velkého gridu a uživatel je tak může využívat jako jeden velký virtuální cluster.

1.3 Dostupné zdroje výpočetního výkonu

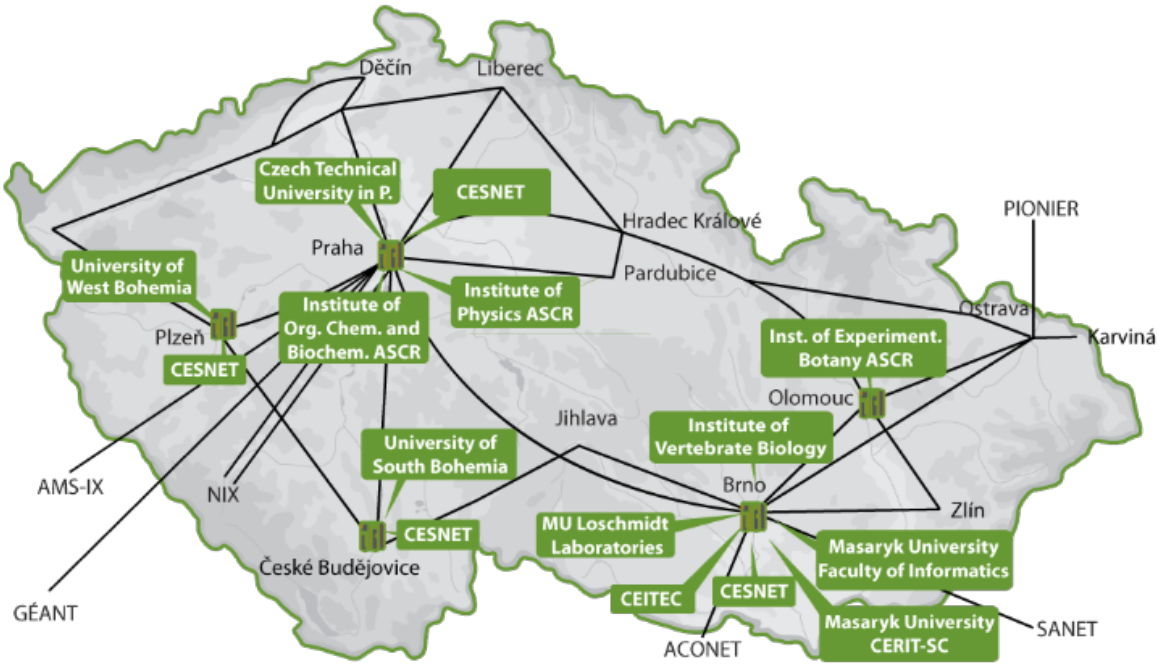
Pro naše potřeby budeme předpokládat pouze zdroje výpočetního výkonu, které jsou snadno dostupné pro Českou republiku. Samozřejmě by bylo možné si pronajmout nějaký strojový čas na soukromém superpočítači v zahraničí, to bychom se však pravděpodobně pohybovali v cenových relacích, které jsou mimo rozsah celého projektu. Výpočetní infrastruktura tedy bude v dalších kapitolách popisována především z pohledu *České Národní Gridové Infrastruktury* (**NGI**). Ta je známá spíše pod názvem **MetaCentrum VO** a spadá pod aktivity sdružení *CESNET, z.s.p.o*. Toto užší zaměření je zvoleno především proto, že **MetaCentrum VO** je dobře dostupné pro celou akademickou obec a většina budoucích výpočetních úloh bude s největší pravděpodobností prováděna právě tam. Ostatní výpočetní systémy jsou povětšinou provozovány obdobně, hlavním rozdílem bývají různorodé varianty a specifické implementace plánovače úloh.

1.3.1 Univerzitní cluster Hydra

Jako nejjednoduší varianta pro spouštění časově náročných paralelizovatelných úloh se jeví využití univerzitního clusteru, který je pro studenty a zaměstnance volně dostupný. Celý cluster se skládá z celkem 24 uzlů, které mají dohromady 48 procesorů se 70 výpočetními jádry. V porovnání s průměrně provozovanými clustery se jedná spíše o menší cluster, ale pro námi požadované výpočty naprosto dostačuje. Velkou výhodou této varianty je hlavně možnost instalovat si další potřebné aplikace a konfigurovat některé parametry dle specifických potřeb. Na clusterech třetích stran nejsou jakékoliv změny obvykle možné, proto pro potřeby vývoje a testování pravděpodobně využijeme tuto variantu. Více o univerzitním clusteru Hydra se lze dozvědět na informačním webu s dokumentací [3].

1.3.2 MetaCentrum VO

Další možností je využít služeb *Národní Gridové Infrastruktury*, která poskytuje jednotný přístup k výpočetním systémům, rozprostřeným na vědeckých pracovištích po celé České republice, viz Obrázek 1. Pro akademické účely je přístup do tohoto systému zcela zdarma a je možné získat poměrně zajímavý výpočetní výkon na různých clusterech. Centrální přístup k mnoha zdrojům najednou je bezesporu velkou výhodou, běhové prostředí ale nelze plně přizpůsobit, protože jsou podmínky použití vázány na konkrétního poskytovatele clusteru. Podrobnosti a přihlášky lze nalézt na webu **MetaCentra VO** [4].



*Obrázek 1: Infrastruktura **MetaCentra VO** v ČR, převzato z [4]*

1.3.3 Superpočítač Salomon v Ostravě

Poměrně čerstvým kandidátem je nedávno dostavený superpočítač v Ostravě. Ten nabízí aktuálně největší výpočetní výkon v ČR a ve světě se pohybuje v první čtyřicítce [5]. Nicméně přístup k tomuto počítači také podléhá nejpřísnějším podmínkám a přidělení výpočetního času může být podmíněno podáním projektu. Nehodí se tedy například na vývoj a testování, pro komplikované výpočty je ale pravděpodobně nejlepší možnou volbou. Více o superpočítači lze dohledat na webu národního superpočítačového centra *IT4Innovations* [6].

1.4 Plánovací software

Součástí každého výpočetního systému je výkonný plánovač úloh, zvaný též **PBS**. Na tomto softwaru závisí zpracování požadavků od uživatelů a následné přidělování zdrojů na dostupné infrastruktuře. Plánovacích softwarů existuje hned několik, v našem prostředí se můžeme setkat hlavně se dvěma variantami. První variantou je **Sunfire Grid Engine** [8], který je využíván na univerzitním clusteru Hydra [3] a druhou variantou je **TorquePBS** [9], [10], jehož upravenou [11] variantu je možné najít na strojích **MetaCentra VO**. Existuje však mnoho dalších možností, které nejsou tolik rozšířené. Jedná se často o specializované komerční verze těch volně dostupných. Všechny tyto systémy však používají obdobné příkazy a liší se pouze v syntaxi dostupných parametrů. Nejpoužívanějšími příkazy jsou `qsub`, `qstat` a `qdel`, které budou podrobněji popsány v kapitole 1.5.4. Vybrané parametry jsou pak vysvětleny v následujících třech podkapitolách.

1.4.1 Fronty

Všechny úlohy odeslané ke zpracování jsou na serveru zařazeny do některé z předdefinovaných **front**. Odtud se pak spouštějí v závislosti na času přidání, požadovaných zdrojích, odhadované délce a dalších nastavených parametrech. Tento systém zajišťuje optimální a férové přidělování zdrojů mezi všechny uživatele. Úlohy jsou do front řazeny primárně podle dvou kriterií. Prvním kriteriem je optimalizace z hlediska času, která je zmíněna v kapitole 1.4.2. Druhým je pak požadavek na specifický hardware nebo nároky, vyplývající z vlastnictví nějakého hardwaru, tomu se věnuje kapitola 1.4.3. Fronty lze přidělovat i přímo na základě požadavků. Některé úlohy je například vhodné provádět na grafických kartách, hodí se tedy tuto úlohu zařadit do fronty zvané *gpu*, ve které se čeká na stroje s vysokým grafickým výkonem. O jiných úlohách je naopak známo, že díky optimalizacím pracují lépe na procesorech značky Intel. Poslouží nám tedy lépe fronta *intel*. Takto bychom mohli pokračovat dále, pro lepší představu lze nahlédnout do seznamu front, dostupných na MetaCentru [7].

1.4.2 Walltime

Dalším důležitým parametrem plánovače je takzvaný **walltime**, což je maximální možná doba běhu úlohy. Obvykle se tento čas pohybuje v řádu hodin až dnů a po uplynutí této stanovené doby je úloha násilně ukončena. To zajišťuje, že žádný z uživatelů nemůže mít přiděleny výpočetní zdroje do nekonečna, ať už úmyslně nebo z důvodu chyby a zacyklení v programu. Tento parametr se často využívá v návaznosti na fronty. Některé fronty jsou například vhodné pro krátké úlohy, kterých může být opravdu mnoho. Jiné se naopak hodí pro úlohu, která sice není příliš náročná, ale běží velmi dlouho. Z toho důvodu jsou v systému fronty, které mají přidělen určitý pevný maximální walltime, manuálním výběrem vhodné fronty lze tedy pomoci plánovači lépe přerozdělit dostupné zdroje.

1.4.3 Požadavky na hardware

Po mnoha plánovačích lze kromě **fronty** a **walltime** požadovat také konkrétní výpočetní výkon nebo specifický hardware. Nejběžněji se jedná o počet potřebných uzlů, počet procesorů na jeden uzel a nebo minimální přípustné množství operační paměti. Ve speciálních případech pak můžeme požádat například o stroje, které disponují lokálním **SSD** diskem. To je velmi výhodné, pokud se délka zpracování úlohy odvíjí od rychlosti přístupu na pevný disk. Tyto požadavky a jejich syntaxe jsou ale velmi závislé na používaném výpočetním systému a jejich výsledná implementace se může různit.

1.5 Použití vzdáleného systému

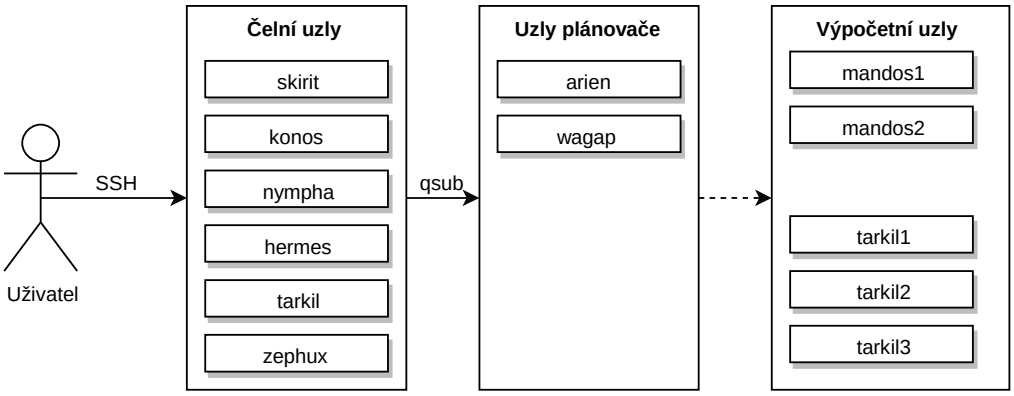
Pro přiblížení celé problematiky plánovačů úloh si zde uvedeme stručný postup pro odeslání požadovaného výpočtu ke zpracování. Na tomto příkladu si lze totiž snadno představit problémy, se kterými se uživatelé těchto systémů mohou potýkat. Zmíněný příklad navíc poslouží i jako recept pro jakýkoliv software zaměřený na zjednodušení práce s výpočetními systémy. I přesto, že bude tento postup specifický pro MetaCentrum, lze ho s náležitými úpravami použít i jinde, spousta výpočetních systémů totiž funguje na podobném principu. Pro podrobnější informace lze opět nahlédnout do dokumentace MetaCentra [2].

1.5.1 Přístupové údaje a registrace

MetaCentrum VO je přístupné všem institucím, které jsou zapojeny v systému **EDUid**. Než však začneme systém využívat, je nutné se zaregistrovat a vytvořit si vlastní přístupové údaje. To lze provést přihlášením přes přístupový portál vybrané univerzity a vyplněním potřebných údajů do webového rozhraní. Uživatelé, kteří nemají možnost přístupu přes **EDUid**, se mohou také zaregistrovat, ale musí navíc vyplnit své osobní údaje a projít schvalovacím procesem.

1.5.2 Připojení do systému

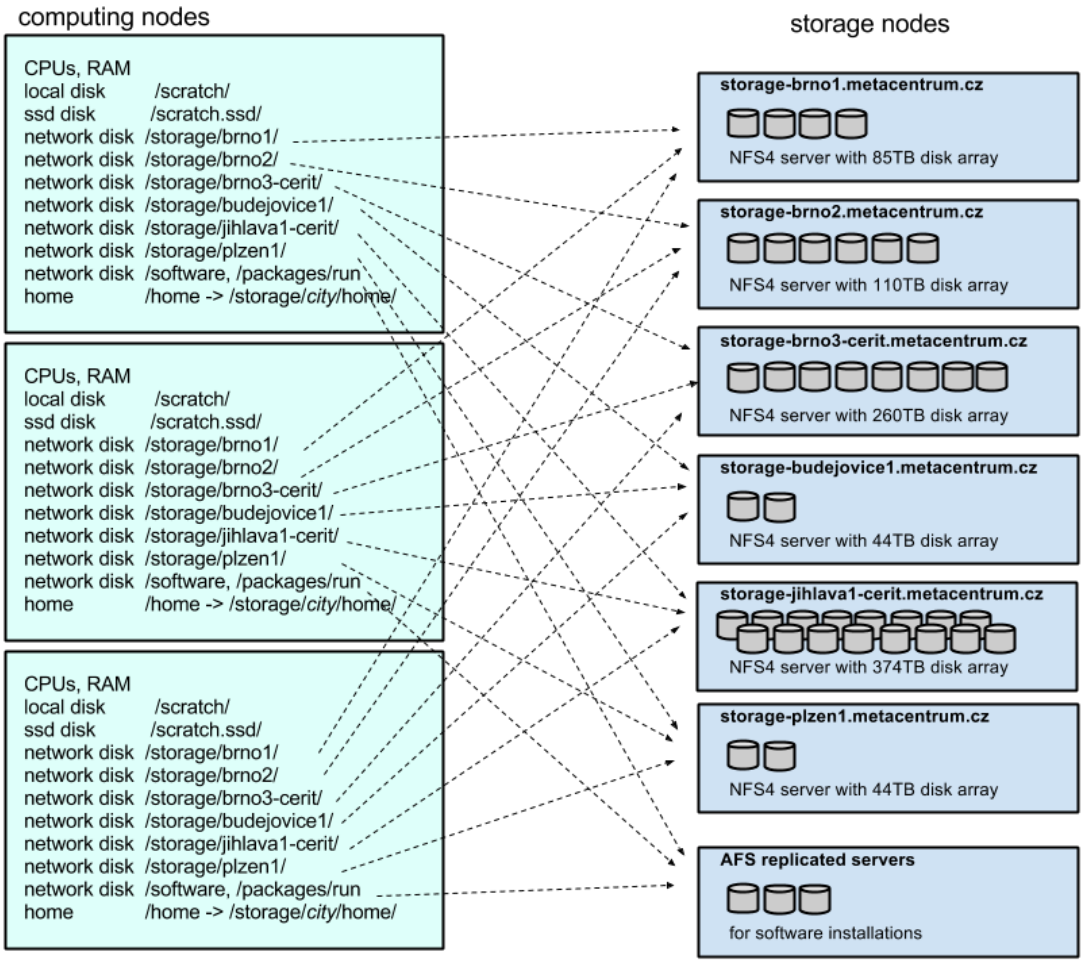
Získanými přístupovými údaji se uživatelé mohou přes **SSH** přihlásit na jeden z čelních uzlů (**frontend**). Pohodlnější přístup si lze zajistit pomocí veřejného klíče, který je možné nahrát do příslušné složky v domovském adresáři. Pak již probíhá identifikace pomocí klíče a přístupové údaje nejsou nadále potřeba. Z čelních uzlů můžeme provádět všechny další důležité operace jako například kopírovat soubory, připravovat si úlohy nebo komunikovat s plánovačem úloh. Pro lepší představu je na Obrázku 2 znázorněno zjednodušené schéma předních uzlů, ke kterým je možné se přihlásit. Při pokusu kontaktovat jiný než čelní uzel je běžný uživatel automaticky odmítnut.



*Obrázek 2: Rozložení čelních uzlů ve výpočetním systému **MetaCentra VO** dle [2]*

1.5.3 Příprava úlohy a kopírování dat

Aby mohla být úloha úspěšně spuštěna, je nezbytné mít na vzdálený systém nahrána všechna data, potřebná pro výpočet. Pro tyto účely je systému obvykle vyhrazen domovský adresář uživatele na sdíleném diskovém poli. K přenášení dat mezi systémem a lokálním počítačem nám poslouží protokol **SCP**. Pro větší objemy dat jsou v případě MetaCentra z každého počítače dostupná i různá rozšiřující disková úložiště, viz Obrázek 3. Obecně lze říct, že síťové disky mají menší přenosové rychlosti, za to však poskytují řádově větší kapacitu a naopak.



Obrázek 3: Diskové prostory a souborové systémy MetaCentra VO, převzato z [2]

S výhodou lze tedy na výpočty použít rychlejší lokální disk, připojený přímo k uzlu, a po dokončení výpočtu výsledky překopírovat na sdílené síťové disky. Cesta do vyhrazeného lokálního prostoru je na MetaCentru vždy dostupná skrze systémovou proměnnou `$SCRATCHDIR`, v proměnné `$HOME` je pak uložena cesta k domovskému adresáři aktuálního uživatele. Všechny potřebné disky jsou na strojích automaticky připojeny.

1.5.4 Odeslání výpočtu ke zpracování

Odeslání výsledku se provádí příkazem `qsub`, plánovač úloh dle parametrů příkazu zajistí zařazení do správné fronty. Pokud je uvedena fronta, tak podle ní, v opačném případě je rozhodnuto dle parametrů požadovaného výkonu a uvedených rozšiřujících specifikací. Plánovač následně fronty prochází a úlohám dle priorit přiřazuje volné výpočetní kapacity. Uvažujeme-li následující příklad, tak v případě spuštění úlohy je na požadovaných strojích spuštěn konfigurační skript `mojeuloha.sh`, zajišťující přípravu a spuštění veškerých potřebných aplikací.

```
$ qsub -l walltime=1:00:00 -l nodes=1:ppn=4,mem=4gb,scratch=50gb mojeuloha.sh
```

Kód 1: Přímé zadaní parametrů pro příkaz `qsub`

Parametry není nutné psát přímo do příkazu. Pro jejich specifikaci lze využít i konfigurační skript, který umožňuje parametry příkazu `qsub` uvést v komentářích přímo ve svém těle. Pokud by došlo ke konfliktu příkazů, plánovač dá přednost hodnotám zapsaným ve skriptu, případně použije výchozí hodnoty. Pokud by tedy skript v následující ukázce neobsahoval žádné parametry, bude přidělen uzel v základní konfiguraci dle použitého systému.

```
$ qsub mojeuloha.sh
```

Kód 2: Parametry příkazu `qsub` v externím souboru

Ukázkový konfigurační skript by potom mohl vypadat jako následující příklad, převzatý z dokumentace MetaCentra [2]. Jako první jsou uvedeny v komentářích parametry pro plánovač, následuje přípravny proces, spuštění úlohy a nakonec kopírovaní spolu s úklidem, podrobněji je vše popsáno v příložných komentářích.

```
#!/bin/bash
#PBS -N mujprvnijob
#PBS -l nodes=1:ppn=1
#PBS -l mem=500mb
#PBS -l scratch=1gb
#PBS -j oe
#PBS -m e
# Chybový výstup připojí ke standardnímu výstupu a pošle mail při skončení úlohy
# nastavení úklidu SCRATCHE při chybě nebo ukončení (pokud neřekneme jinak, uklidíme po sobě)
trap 'clean_scratch' TERM EXIT
# Nastavení pracovního adresáře pro vstupní/výstupní data
DATADIR="$PBS_O_WORKDIR"
# Příprava vstupních dat
cp $DATADIR/vstup.txt $SCRATCHDIR || exit 1
# Přechod do pracovního adresáře a zahájení výpočtu
cd $SCRATCHDIR
# Nahrání požadovaného modulu a spuštění výpočtu
module add maple
maple input.mpl
# Vykopírování výsledků ze scratche
cp $SCRATCHDIR/output.gif $DATADIR || export CLEAN_SCRATCH=false
```

Kód 3: Ukázkový spouštěcí skript převzatý z [2]

1.5.5 Monitorování a vyzvednutí výsledků

Pokud jsou úlohy časově náročné, přijde vhod funkce odesílání informativního emailu po dokončení úlohy, jak si lze všimnout v předchozím příkladě. Během výpočtu pak můžeme průběh monitorovat použitím příkazu `qstat` jako je uvedeno v následujícím příkladu. Tento příkaz vypíše dostupné informace o požadované úloze, výstup je následně možné přesměrovat například do souboru.

```
$ qstat 12345.arien.ics.muni.cz # vypíše základní informace o dané úloze
$ qstat -f <jobID> # vypíše kompletní informace o dané úloze
$ qstat -u <username> # vypíše informace o aktuálních úlohách patřících uživateli
```

Kód 4: Různé možnosti pro příkaz `qstat`

Pokud je úlohu nutné kdykoliv v průběhu výpočtu (případně i před spuštěním) ukončit například kvůli zjištěné chybě, je možné použít příkaz `qdel`. Ten úlohu násilně ukončí, případně odstraní z čekací fronty.

```
$ qdel <jobID> # ukončí běžící úlohu dle zadaného id
```

Kód 5: Ukončení úlohy použitím příkazu `qdel`

Po skončení úlohy (chybovém, násilném nebo korektním) se v pracovním adresáři objeví dva soubory, `<job_name>.o<jobID>` který reprezentuje standardní unixový výstup `STDOUT` a `<job_name>.e<jobID>` pro `STDERR`. Tyto soubory je možné spolu s dalšími vytvořenými soubory přenést zpět a dále zpracovat, čímž je životní cyklus celého výpočtu dokončen.

1.6 Požadavky kontextu Jobs

Na začátek pro jistotu ještě krátké připomenutí. Kontext Jobs je dílčí součástí aplikace GeoMop a zabývá se vzdáleným zpracováním úloh na výpočetních systémech. V tuto chvíli bychom měli mít z předešlých kapitol dostatečný teoretický základ, abychom se mohli blíže zaměřit na některé konkrétní požadavky kontextu Jobs. Na základě těchto požadavků celý kontext vzniká a jsou tedy rozhodující i pro výslednou klientskou aplikaci.

1.6.1 Zjednodušení postupu

Jedním z hlavních požadavků kontextu je celkové zjednodušení práce s úlohou. Po přečtení předchozích kapitol není pochyb, že spuštění úlohy je pro nezkušeného uživatele poměrně nelehký úkol. Celý postup by se tedy měl výrazně zjednodušit tak, aby byl uživatel odstíněn od složitých příprav úlohy a fungování serverové logiky. V ideálním případě by jen zadal data pro zpracování úlohy do uživatelského rozhraní a provedl by základní konfiguraci výpočetního systému. Aplikace by pak automaticky provedla překopírování dat na vzdálený systém a spuštění úlohy. Uživateli by stačilo z klientské aplikace kontrolovat průběh výpočtu a po jeho dokončení by si nechal zpracované výsledky zobrazit.

1.6.2 Sjednocení přístupu

Dalším velice důležitým požadavkem je sjednocení přístupu k výpočetním zdrojům. V praxi to znamená, že uživatel může jednotně přistupovat k řízení úloh bez ohledu na to, zda-li se bude výpočet provádět na lokálním počítači, vyhrazeném serveru nebo na libovolném výpočetním systému. Na tyto scénáře se podíváme podrobněji v kapitole 1.7.

1.6.3 Podrobnější sledování průběhu

Přehledné a podrobné monitorování úloh je z pohledu uživatelů velmi žádané. Současný stav je ale značně nekonzistentní, dostupné informace jsou obvykle pouze povrchní a přístup k nim se liší systém od systému. Například složitější úlohy se z pohledu výpočetního systému tváří pouze jako jedna velká úloha a dílčí výsledky nelze zjišťovat v průběhu. To je poměrně neefektivní pokud se v průběhu vyskytne chyba. Místo okamžitého ukončení výpočtu a okamžité korekce, se na chybu přijde až po dokončení všech částí úlohy. Nově navrhovaný systém monitorování by tedy měl rozšířit možnosti sledování, zvětšit spektrum sledovatelných údajů a usnadnit vyzvedávání informací na různých systémech.

1.6.4 Podpora podmíněných výpočtů

Některé výpočty mohou vyžadovat jistou vnitřní logiku pro svůj průběh, například zřetězené výpočty, které probíhají v několika iteracích podmíněně, na předchozím výsledku. Výsledné řešení by tedy mělo poskytovat dostatečně flexibilní podporu pro podobné případy, které by mohly nastat.

1.7 Požadované scénáře spouštění úloh

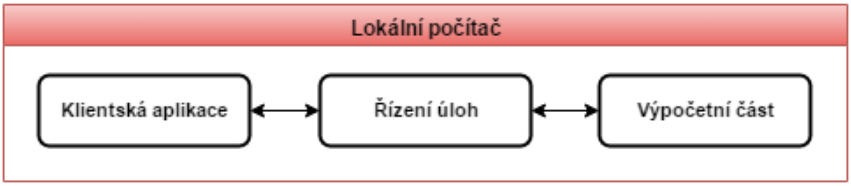
Pro efektivní využití dostupných zdrojů a dostatečnou flexibilitu při provádění výpočtu bylo navrženo několik scénářů spouštění. Tyto scénáře by měly pokrývat všechny požadavky případných uživatelů, celá problematika je ale mnohem složitější. Situaci komplikují různé restrikce, vyplývající z fungování výpočetních systémů a rozdíly v implementaci plánovačů úloh. K základnímu porozumění si však vystačíme se zjednodušujícími diagramy, uvedenými dále.

Základním kamenem všech níže uvedených scénářů jsou různé druhy systémů pro zpracování výpočtu. Ty jsou na diagramech označeny červeným rámečkem, viz například Obrázek 9. Lokální počítač můžeme chápat jako libovolné zařízení, které je schopno spustit klientskou aplikaci, obvykle tedy stolní počítač nebo notebook. Jako server si pak můžeme představit jakýkoliv unixový počítač, dostupný přes **SSH** a cluster nám reprezentuje jeden z paralelních výpočetních systémů, zmíněných výše.

Samotný výkonný kód se rozpadá na tři hlavní části. *Výpočetní část*, což je samostatný výpočet rozšířený o jednoduché komunikační a monitorovací rozhraní. Dále pak je to *řízení úloh*, které je v podstatě komunikačním mostem mezi klientem a výpočty. Stará se o řízení výpočtů a obsluhuje frontu čekajících podúloh. Poslední částí je *klientská aplikace*, která pomocí uživatelského rozhraní vytváří jednotlivé úlohy a jejich konfigurační soubory. Tyto úlohy pak dále ovládá a monitoruje za pomoci *řízení úloh*. Tyto tři základní části jsou společné pro všechny uvedené scénáře. Rozdíl je pouze v tom, na kterém systému je zrovna daná část spuštěna.

1.7.1 Scénář 1 (klient, klient, klient)

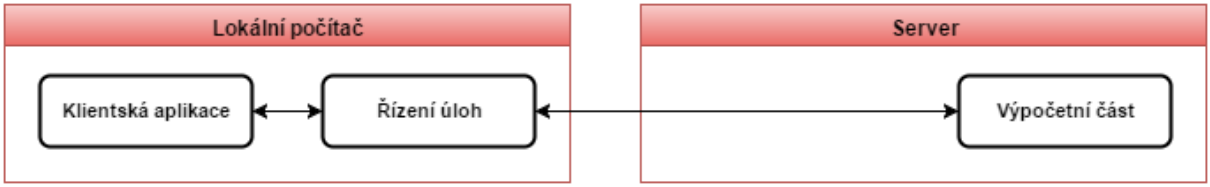
Tento scénář je velice jednoduchý, všechny potřebné výpočty i jejich řízení se provádí na lokálním počítači. To je vhodné zejména pro jednoduché a krátké výpočty, svoje využití si najde převážně při práci v terénu či bez připojení k internetu.



Obrázek 4: Schéma pro scénář spouštění číslo 1

1.7.2 Scénář 2 (klient, klient, server)

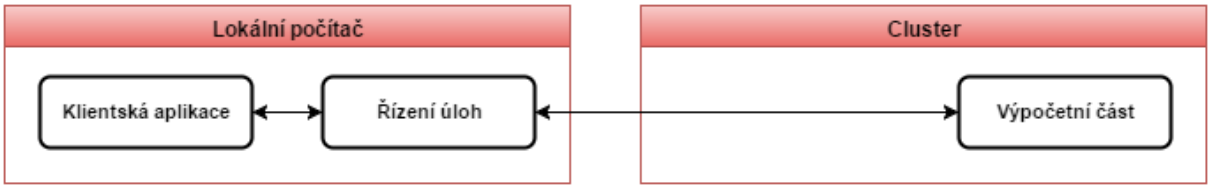
Za předpokladu, že máme k dispozici výkonný server, je výhodné přenést náročné výpočty na něj. Výsledky tak můžeme získat podstatně rychleji, záleží však samozřejmě na výkonu daného serveru. Nevýhodou ale zůstává, že lokální počítač musí neustále běžet, aby nám vyhodnocoval výsledky a zařazoval nové úlohy. Pro delší výpočty to může být poměrně limitující faktor, v takovém případě lépe poslouží některý z následujících scénářů.



Obrázek 5: Schéma pro scénář spouštění číslo 2

1.7.3 Scénář 3 (klient, klient, cluster)

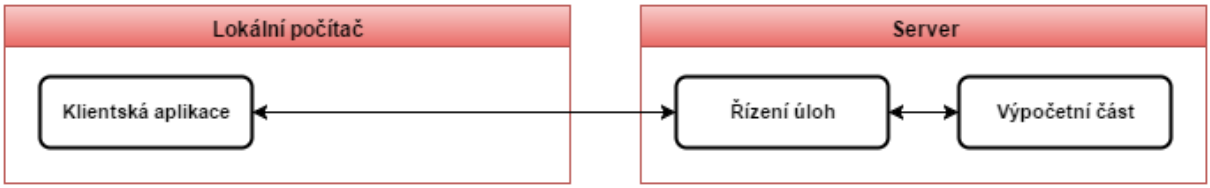
Scénář číslo tři je v podstatě obdobou předchozího případu, na místo serveru je ale využit výpočetní cluster. Ten má obvykle ještě větší výkon a pro rozsáhlé paralelní výpočty může být i mnohem vhodnější. Opět však platí, že lokální počítač musí zůstat zapnutý v průběhu celého výpočtu.



Obrázek 6: Schéma pro scénář spouštění číslo 3

1.7.4 Scénář 4 (klient, server, server)

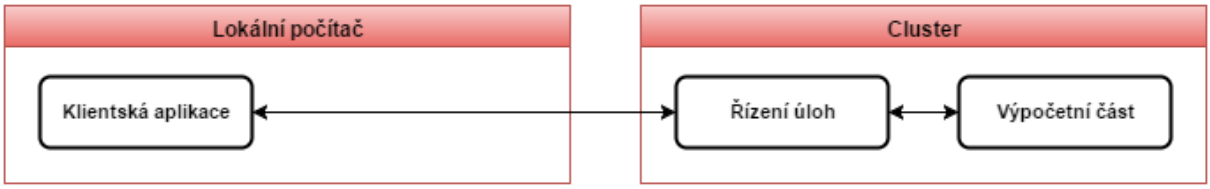
Aby nemusel lokální počítač neustále běžet, je možné přesunout řídicí logiku také na server. To je výhodné zejména při dlouhých výpočtech, klientská aplikace se může bez problému odpojit a zkontrolovat si výsledky například až ráno. Pro tento případ je však nutné mít dostatečně nadimenzovaný serverový počítač, který se k danému účelu použije.



Obrázek 7: Schéma pro scénář spouštění číslo 4

1.7.5 Scénář 5 (klient, cluster, cluster)

Toto řešení funguje na podobném principu jako předchozí scénář, opět je zde server nahrazen výkonnějším clusterem a klientská aplikace se tedy může také libovolně odpojovat. Zde je ale třeba dát si pozor na provozní politiku použitého výpočetního systému, v některých případech je provoz takovýchto řídících částí omezen nebo podléhá nějakým zvláštním pravidlům. Může se tedy stát, že bude řízení úloh administrátorem nebo automatickým skriptem neustále ukončováno.



Obrázek 8: Schéma pro scénář spouštění číslo 5

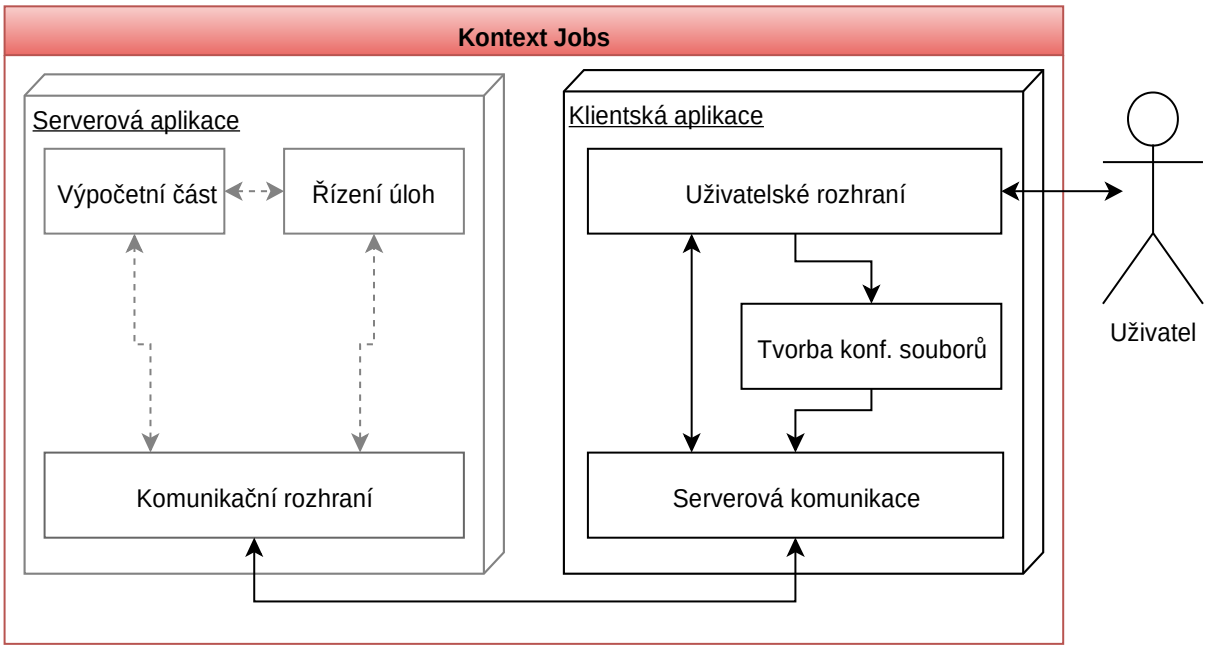
1.7.6 Scénář 6 (klient, server, cluster)

Odpovědí na oba výše zmiňované nedostatky je tento scénář. Řídící logika je přenesa na server a klient se tedy může opět bez problému odpojit. Samotné výpočty jsou pak prováděny na clusteru a na server tím pádem nejsou kladeny žádné velké nároky, je tedy možné pro jeho realizaci využít například nějaký úsporný minipočítač. Zároveň se tímto řešením dá elegantně obejít provozní politika výpočetních systémů, bránící případnému běhu řídící logiky.

Obrázek 9: Schéma pro scénář spouštění číslo 6

1.8 Vytyčení cílů práce

Z předchozích kapitol je zřejmé, že problematika kontextu Jobs je poměrně rozsáhlá a komplikovaná. Stanovené požadavky a specifika výpočetních systémů přinášejí mnoho komplikovaných problémů, se kterými je nutné se vypořádat. Proto byl celý kontext rozdělen na dvě dílčí části. První z nich je serverová a druhou pak klientská aplikace, detailnější oddělení znázorňuje Obrázek 10.



Obrázek 10: Blokový diagram součástí v rámci kontextu Jobs

V této práci se zabývám návrhem a implementací klientské aplikace. Ta se dále dělí na tři dílčí celky, jejichž návrhem a následnou implementací se budeme zabývat v dalších kapitolách. Bez základních znalosti serverové architektury se ale neobejdeme.

Pro mou práci jsem si tedy stanovil následující cíle:

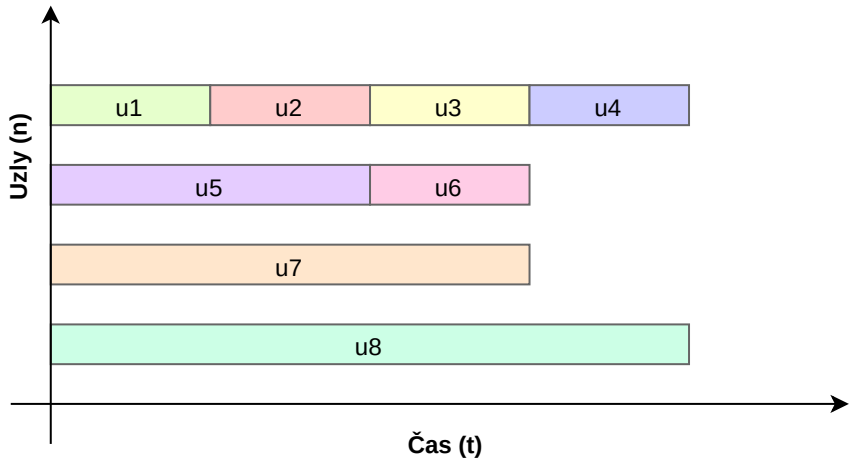
1) Seznámit se s problematikou výpočetních systémů a požadovaných scénářů spuštění v kontextu Jobs.
2) Prostudovat důkladně architekturu serverové aplikace.
3) Navrhnout a implementovat uživatelské rozhraní, které umožňuje pohodlné zadání všech dat pro požadované scénáře.
4) Vytvořit komponentu pro generování konfiguračních souborů, v závislosti na vybraném scénáři.
5) Pomocí vhodných prostředků propojit navržené uživatelské rozhraní se serverovou architekturou.
6) Otestovat scénáře, které se podaří zprovoznit v rámci programování serverové aplikace.

2 Analýza

Klientská a serverová aplikace jsou úzce spjaty a i když se tato práce bude zabývat především klientskou částí, bude nezbytné alespoň zjednodušeně některé detaily serverové aplikace rozebrat. Jak tedy zpracovávaná úloha vypadá, jaké implementační problémy výpočetní systémy přinášejí? Jak jsou tyto problémy řešeny v serverové aplikaci a jakým způsobem se výsledné řešení promítne do návrhu klientské aplikace? Odpovědí na tyto otázky se budeme zabývat na následujících několika stranách.

2.1 Definice úlohy

Než se ponoříme do systému komunikace a zpracování úlohy, podívejme se nejprve na to, jak taková úloha může vypadat a jak je zadána. Mluvíme-li o úloze, máme obvykle na mysli sadu podúloh (označeno **u1** až **u8**, viz Obrázek 11) určitého typu, souhrnně jeden komplexní výpočet.



Obrázek 11: Grafické znázornění komplexní výpočetní úlohy

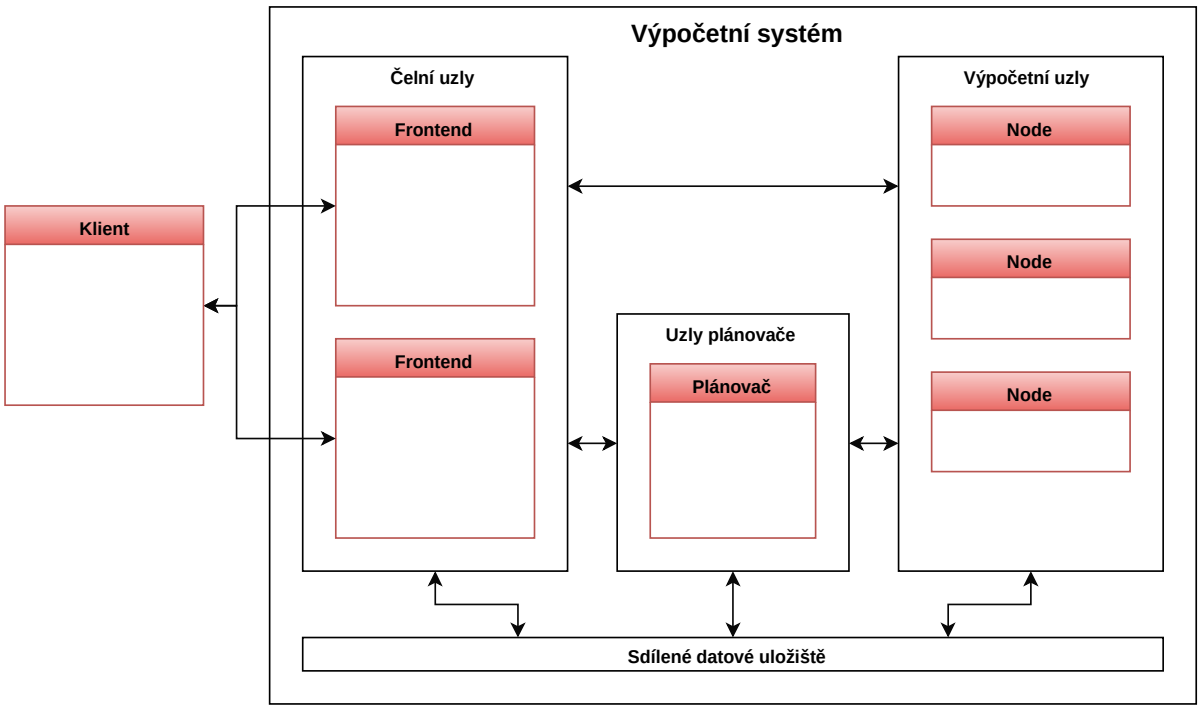
Směrem vzhůru máme mezi výpočetní uzly rozloženy podúlohy, které lze paralelizovat v čase. Naopak podúlohy **u1**, **u2**, **u3** a **u4** jsou zařazeny sériově. Jak již bylo zmíněno, může se jednat například o výpočty, které závisí na předchozím výsledku, případně jde o optimalizaci na počet využitých uzlů. Jak je patrné z Obrázku 11, podúloha **u8** vyžaduje na zpracování nejvíce času, můžeme si tedy dovolit některé výpočty zařadit po sobě a redukovat počet potřebných uzlů.

Podrobněji se skládáním a optimalizací úlohy zabývat nebudeme, z pohledu klientské aplikace je vždy převzat pouze definiční soubor požadované úlohy. Tento soubor vytváří uživatelé externím nástrojem na základě znalosti úlohy. Uživatel tedy musí pro svou úlohu znát optimální počet využitelných uzlů, nejhorší možný čas jejího zpracování a případně i další specifika. Tyto údaje bude při zadávání úlohy požadovat uživatelské rozhraní, protože jsou nutné při komunikaci s plánovačem úloh.

2.2 Omezení serverové komunikace

Většina dostupných výpočetních systémů je konstruována velice podobně. Uvnitř systému nalezneme tři typy uzlů, jak je znázorněno na Obrázku 12. Některé uzly jsou vyhrazeny čistě jako výpočetní a poskytují potřebný výkon pro zpracování náročných úloh. Pak zde máme uzly, které se starají o přerozdělování zdrojů a běží na nich plánovač úloh. Posledním zastoupeným typem jsou takzvané čelní uzly (**frontendy**). Ty jsou jako jediné přístupné z vnější sítě. Pravděpodobně se jedná o bezpečnostní opatření, díky kterému je možné ponechat komunikaci uvnitř systému bez omezení. Knihovny používané pro vnitřní komunikaci tak nemusí řešit zabezpečení. Nicméně systém jako takový zůstává přístupný pouze přes **SSH** na čelní uzly. To je celkem podstatné omezení, které je třeba brát v potaz.

Další omezení se týká čelních uzlů. Ty slouží primárně na přípravu úloh a komunikaci s plánovačem, měly by tudíž poskytovat spolehlivý přístup všem uživatelům systému. Jejich nadměrné vytěžování je z toho důvodu často omezeno nějakou vnitřní politikou provozovatele sytému. Spouštění náročných aplikací, dlouho trvajících úloh či provozování aplikace serverového charakteru tedy obvykle nepřipadá v úvahu nebo je administrátory značně limitováno.



Obrázek 12: Architektura výpočetního systému a vnitřní rozložení uzlů

Použití výpočetních systémů nám tak přináší dvě podstatná omezení. Komunikovat lze pouze s čelními uzly, a to jen přes **SSH**, navíc ve většině výpočetních systémů nesmí na čelních uzlech dlouhodobě běžet řízení výpočtů. Tato omezení je nutné v serverové aplikaci překonat, aby bylo možné realizovat všechny navrhované scénáře. Proto byly vytvořeny dvě nové serverové komponenty, které nejsou zohledněny v původních scénářích z kapitoly 1.7. Tyto komponenty operují na čelním uzlu a slouží jako síťový most a opakovač odeslaných příkazů. Funkce jednotlivých komponent bude podrobněji vysvětlena v následující kapitole.

2.3 Serverové komponenty

Původní scénáře spouštění získaly během vývoje konkrétnější podobu a výsledná serverová logika byla k překlenutí nastíněných komunikačních omezení a splnění kladených požadavků rozdělena do několika samostatných komponent. V této kapitole si tyto komponenty pojmenujeme, přiblížíme rámcově jejich funkci a také jejich vztah k původním spouštěcím scénářům navrženým v kapitole 1.7.

2.3.1 komponenta APP

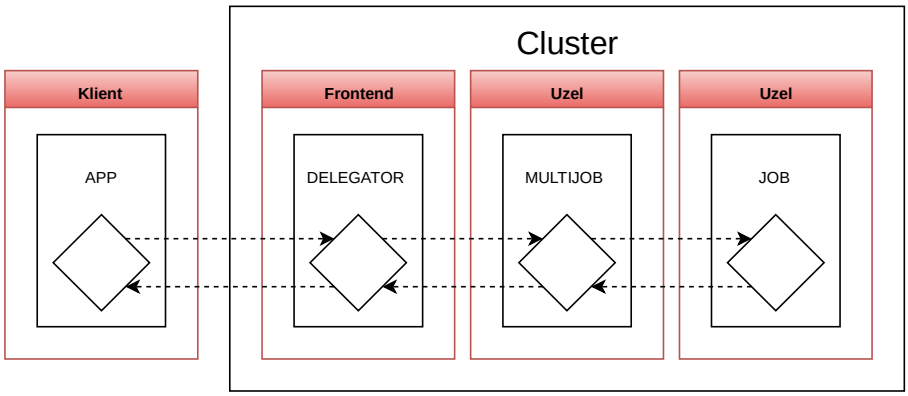
Komponentu **`APP`** chápejme jako vstupní bod celé komunikace. Jedná se sice o komponentu serverovou, v původních scénářích by však byla zakreslena uvnitř klientské aplikace. Její instance jsou totiž využívány ke komunikaci se vzdálenými servery. Na podrobnější popis metod a komunikačního rozhraní se zaměříme v kapitole 2.6.

2.3.2 komponenta MULTIJOB

V kapitole 2.1 jsme si definovali, jak vypadá komplexní úloha, tu můžeme chápat také jako jeden samostatný **`MULTIJOB`**. Každá takováto komponenta je samostatným serverem, který komunikuje s klientskou aplikací, obsluhuje plánovač úloh a spravuje své přiřazené podúlohy. Klientská aplikace tedy může ve výsledku komunikovat současně s několika komponentami **`MULTIJOB`** na několika různých systémech. V původních scénářích najdeme analogii s částí pro řízení úloh.

2.3.3 komponenta DELEGATOR

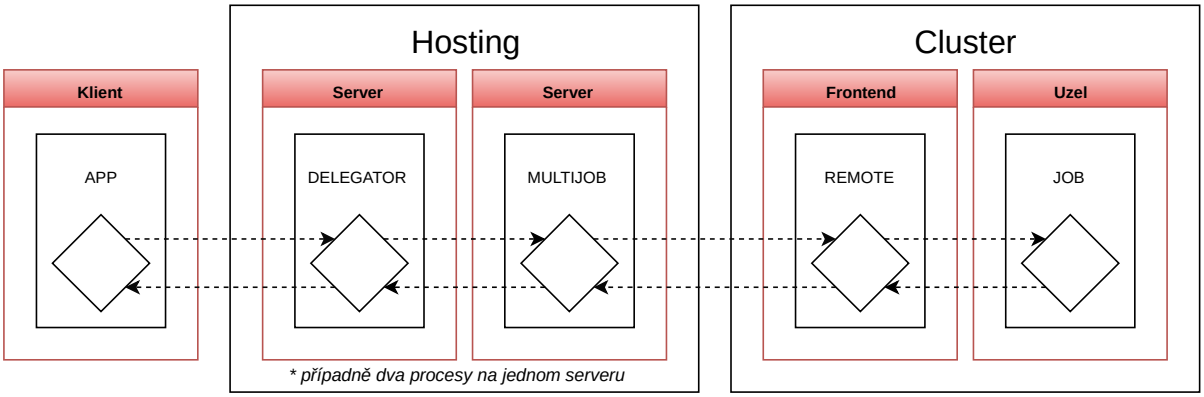
Abychom uspokojili požadavky plánovaného scénáře 5 v kapitole 1.7.5, bylo nutné obejít omezení pro spouštění aplikací serverového typu na čelních uzlech. Proto **MULTIJOB** v některých případech musí být spuštěn na samostatném výpočetním uzlu, z těchto uzlů ale nelze komunikovat přímo ven ze systému. Vznikla tedy komponenta **DELEGATOR**, která se stará o přemostění komunikace mezi klientem a výpočetním uzlem přeš uzel čelní. Obrázek 13 zobrazuje výsledné rozložení komponent na reálném výpočetním systému dle scénáře číslo 5.



*Obrázek 13: Diagram komunikace s použitím komponenty **DELEGATOR***

2.3.4 komponenta REMOTE

Obdobnou situaci je nutné řešit i pro navrhovaný scénář číslo 6 v kapitole 1.7.6. Jako zjednodušená verze komponenty **DELEGATOR** tak vznikla komponenta **REMOTE**. Ta má podobné využíti, disponuje však pouze zjednodušenou řídící logikou a slouží především jako opakovač pro komponenty **JOB**. Podrobnější rozložení znázorňuje Obrázek 14.



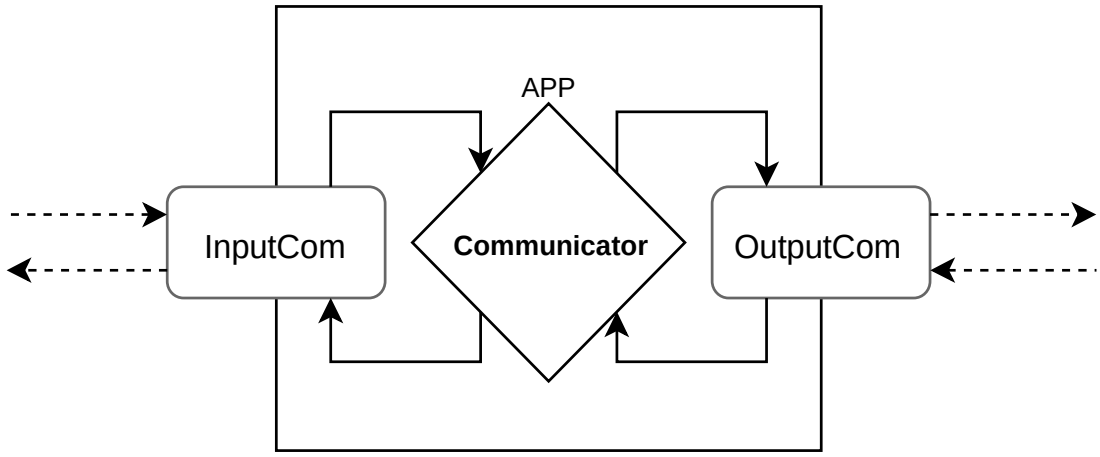
*Obrázek 14: Diagram komunikace s použitím komponenty **REMOTE***

2.3.5 komponenta JOB

Jako obal pro libovolnou podúlohu slouží komponenta **JOB**, ta zajišťuje správnou konfiguraci prostředí a následné spuštění případných dalších podúloh. Zjednodušuje také monitorování úlohy a může sloužit jako kostra k implementaci další logiky pro složitější řízení výpočtů. V původním návrhu je uvedena jako výpočetní část.

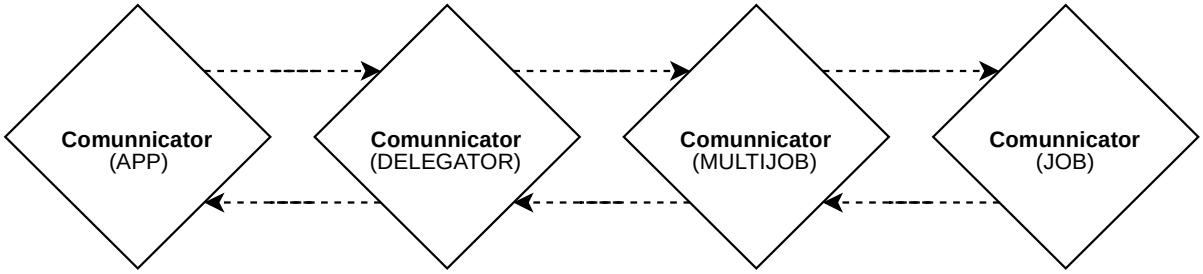
2.4 Architektura serverových komponent

Prozatím jsme jednotlivé komponenty zjednodušovali, nyní se však podíváme podrobněji na jejich vnitřní fungování. Pro naše potřeby ale pomineme obslužnou logiku a zaměříme se pouze na části realizující komunikaci. Ty v komponentě nalezneme tři, viz Obrázek 15. Hlavní částí je rozhraní `Communicator`, které se stará o řízení komunikace a definuje dostupné metody. Celou komunikaci lze tedy realizovat voláním metod příslušné instance implementující právě toto rozhraní. Dále pak zde najdeme vstupní a výstupní rozhraní, která slouží k abstrakci komunikace mezi různými komponentami. Pokud tedy komunikujeme s další komponentou například přes **SSH,** je `OutputCom` realizován příslušnou implementací `SshOutputCom`.



Obrázek 15: Zjednodušeny diagram komunikační komponenty

Komunikace jednotlivých tříd `Communicator` pak probíhá zřetězeně, jak je znázorněno na Obrázku 16. Různé instance na různých systémech si mezi sebou posílají předem definované zprávy s daty. Na základě takovéto zprávy je komponentou vykonána přiřazená akce nebo je (modifikovaná) zpráva přeposlána k dalšímu zpracování. Tento řetěz je vytvořen dynamicky dle předložených konfiguračních souborů, které nastavují chování každé komponenty a jejího komunikačního rozhraní, jak je rozebráno v kapitole 2.7.2.



Obrázek 16: Zřetězení komunikačních komponent

2.5 Konfigurační soubory

Každou komponentu a její instanci rozhraní `Communicator` je před použitím nutné nejprve správně nakonfigurovat. To se provádí pomocí konfiguračního souboru, lépe řečeno sady souborů. Každý takovýto konfigurační soubor je klasickým textovým souborem ve formátu **JSON**, pojmenovaný podle příslušné komponenty, které náleží. Parsováním těchto souborů vzniká zřetězení komunikačních komponent, jak je naznačeno v předchozí kapitole na obrázku 16. Obsah každého z těchto souborů pak nastavuje konkrétní komponentu, její komunikační rozhraní a zmíněná vstupně výstupní rozhraní.

Struktura souborů pro Scénář 1 (klient, klient, klient) může vypadat například takto:

- *app.json*
- *multijob.json*
- *job.json*

Obsah souboru *multijob.json* pak vypadá například takto:

```
{
    "app_version": "0.1.1",
    "communicator_name": "multijob",
    "direct_communication": false,
    "input_type": 2,
    "libs_env": {
        "install_job_libs": true,
        "libs_mpicc": null,
        "mpi_module_add": null,
        "mpi_scl_enable_exec": null,
        "start_job_libs": false
    },
    "log_level": 10,
    "mj_name": "EXEC_EXEC",
    "next_communicator": "job",
    "number_of_processes": 1,
    "output_type": 2,
    "pbs": null,
    "port": 5723,
    "python_env": {
        "module_add": null,
        "python_exec": "python3",
        "scl_enable_exec": null
    },
    "ssh": null
}
```

Kód 6: Vzorový konfigurační soubor `multijob.json`

Za povšimnutí stojí především zvýrazněné řádky:

- `communicator_name` – typ (jméno) použité komponenty
- `next_communicator` – typ následující komponenty dalšího uzlu v řadě
- `input_type` – typ vstupu, který má být nastaven (odkaz do číselníku, viz kapitola 3.1)
- `output_type` – typ výstupní komunikace (odkaz do číselníku, viz kapitola 3.1)

Jméno určuje, jaká komponenta se má použít a jaká implementace rozhraní `Communicator` bude instancována. Obdobně je to i u vstupně výstupní komunikace. Podle uvedených hodnot se opět vybírá implementace, která bude správně rozumět vstupům, případně korektně odesílat výstupní data. Typ další komponenty v řadě pak poskytuje doplňující informace pro komunikaci a případnou validaci správnosti sestaveného komunikačního řetězu.

Dále jsou zajímavé klíče:

- `ssh` – přihlašovací údaje při komunikaci přes **SSH**
- `pbs` – parametry **PBS** pro plánovač úloh

V tomto případě jsou však obě tyto hodnoty `null`, protože Scénář 1 (klient, klient, klient) komunikaci s plánovačem ani připojení přes **SSH** nevyužívá. V jiných scénářích bychom na tomto místě našli přihlašovací údaje a parametry pro plánovač. Ty jsou využívány implementací výstupní komunikace pro připojení k dalším článkům komunikačního řetězu.

2.6 Komunikační rozhraní

Jednou z hlavních funkcí klientské aplikace je komunikace se serverovou aplikací. Ta probíhá voláním metod instance komunikačního rozhraní `Comunicator`. Jelikož budeme tyto metody ke komunikaci v klientské aplikaci hojně využívat, pojďme si ty základní stručně představit.

- `install()` – zařídí překopírování souborů na vzdálené počítače a zajistí spuštění všech potřebných serverových komponent v řetězové struktuře
- `send_long_action()` – odešle požadovanou zprávu k akci dalším uzlům v řadě (obecně lze říci, že libovolná použitá metoda vždy vykoná akci pro danou instanci a tímto příkazem se akce odešle dalšímu uzlu)
- `download()` – stažení souborů s výsledky na lokální počítač, odkud jsou prezentovány v klientské aplikaci
- `interupt()` – korektní ukončení spojení mezi klientskou aplikací a serverovými komponentami (nepřeruší probíhající podúlohy)
- `restore()` – opětovné navázání spojení po korektním ukončení (například opětovné spuštění klientské aplikace)
- `close()` – úplné ukončení (zastaví všechny výpočty a ukončí vzdálené komponenty)

Nyní se pojďme podívat na celý proces komunikace, který by měla klientská aplikace voláním příslušných metod napodobovat. Než však začneme jakékoliv metody volat, je nutné mít instanci třídy `Communicator`. Do konstruktoru je jí jako první parametr předán objekt `config` s nastavením, které je parsováno z konfiguračního souboru. Druhým parametrem je pak identifikátor `id`, používaný v rámci aplikace. Na nově vytvořené instanci je již možno volat metody, viz následující příklad.

```
# vytvoření nové instance
comunicator = Communicator(config, id)

# spuštění instalačního procesu
comunicator.install()
comunicator.send_message(ActionType.installation).get_message())

# stažení souborů s průběhem či výsledky
comunicator.send_long_action(Action(ActionType.download_res))
comunicator.download()

# vyčtení základních stavových informací
mess = comunicator.send_long_action(Action(ActionType.get_state))
data = mess.get_action().data

# korektní přerušení spojení
comunicator.send_long_action(Action(ActionType.interupt_connection))
comunicator.interupt()

# obnovení spojení
comunicator.restore()
comunicator.send_long_action(Action(ActionType.restore_connection))

# opětovné stažení souborů s průběhem či výsledky
comunicator.send_long_action(Action(ActionType.download_res))
comunicator.download()

# ukončení celé serverové aplikace
mess = comunicator.send_long_action(Action(ActionType.stop))
comunicator.close()
```

Kód 7: Příklad použití komunikačního rozhraní

Nejprve je zavolána metoda `install()` spolu s příkazem k instalaci. To zajistí, že se serverová aplikace překopíruje na zvolené (vzdálené) systémy, kde se následně také spustí. Poté jsou metodou `download()` staženy průběžné výsledky, které popisují stav vzdálených částí. Následně proběhne vyčtení stavu komponenty **`MULTIJOB`**, což slouží k rychlému ověření probíhajících operací. Potom je komunikace pozastavena metodou `interupt()` a hned poté opět obnovena pomocí `restore()`, v tomto případě pouze za účelem testování. Potom následuje opětovné stažení průběžných výsledků, po kterém je celá serverová aplikace ukončena pomocí metody `close()`.

2.7 Funkce klientské aplikace

Doposud jsme se zabývali problémy výpočetních systémů a výslednou architekturou serverové aplikace. Správné porozumění a orientace v této problematice je sice zásadní pro další postup, konečným cílem této práce je ale vytvoření klientské aplikace. Proto se v následujících několika podkapitolách zaměříme na ty nejdůležitější funkce klientské aplikace, které z dosavadních poznatků vyplývají a vysvětlíme si jejich napojení na serverovou logiku.

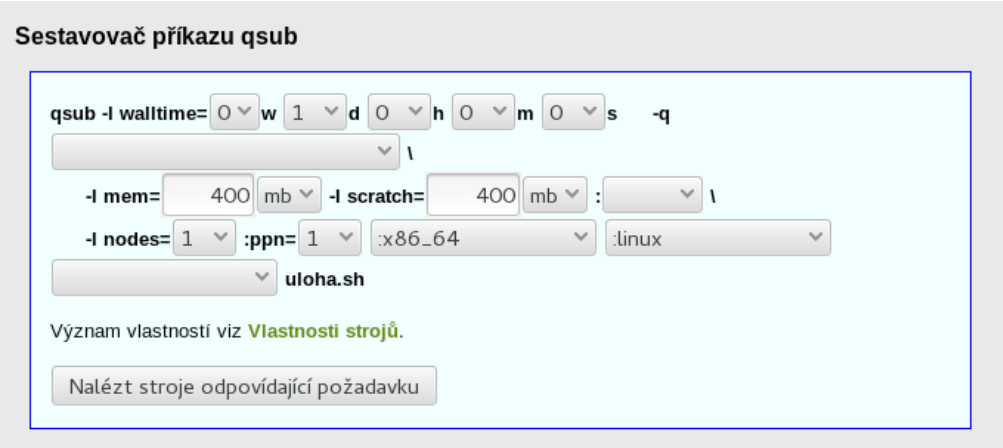
2.7.1 Nastavení a nabídky

Je zřejmé, že klientská aplikace bude obsahovat nějaký základní pohled do uživatelského rozhraní, kterým se bude ovládat serverová komunikace. Ten však nechme prozatím stranou a soustřeďme se nyní raději na prvky a nastavení, které vyplývají přímo z architektury serverové aplikace.

Různé spouštěcí scénáře jsou od počátku nedílnou součástí serverové aplikace, to by mělo reflektovat i uživatelské rozhraní. Proto bude nutné navrhnou grafickou komponentu, která nám umožní pohodlně nastavit požadovaný scénář spuštění. Tato data poslouží pro vygenerování konfiguračních souborů popisujících logiku zřetězení serverových komponent. Pro začátek bychom si mohli vystačit s jednoduchým výběrem scénáře 1 až 6, lepší však samozřejmě bude navrhnout systém, který nevyžaduje podrobnou znalost jednotlivých scénářů.

Určitá část komunikace bude probíhat přes **SSH** a soubory budeme na vzdálený systém přenášet přes protokol **SCP**. Pro větší přehlednost by tedy bylo dobré mít nějakého správce přihlašovacích údajů na různé servery. Ten by měl umožnit pohodlně prohlížet, přidávat, editovat a nebo mazat přihlašovací údaje. Správce bude nakládat s hesly uživatelů a do budoucna by se tedy mohla hodit i rozšířená podpora pro ověřování veřejným klíčem.

V některých scénářích budou severové komponenty spouštěny přímo na výpočetních uzlech, ty je však nutné vyžádat přes plánovač úloh příkazem `qsub`. Aby mohla být komponenta spuštěna správně, je třeba do nastavení předchozího článku v komunikačním řetězu přibalit parametry tohoto příkazu. Uživatelské rozhraní tedy musí umožňovat správu profilů s různými sadami parametrů a jejich pohodlné zadávání. To vše by mělo být ideálně poměrně blízko samotnému příkazu `qsub`, proto se jeví jako vhodné řešení inspirovat se sestavovačem příkazu přímo na webu (přístup pouze pro přihlášené uživatele) MetaCentra [13]. Musíme ale počítat s tím, že se od sebe různé systémy liší a později bude nutné provést abstrakci i pro jiné plánovací systémy s odlišnou syntaxí parametrů.



Obrázek 17: Webové rozhraní sestavovače příkazu `qsub`, převzato z [13]

2.7.2 Konfigurační soubory

Nesmírně důležitou funkcí klientské aplikace je generování konfiguračních souborů. Pro tyto účely by měla vzniknout komponenta, která z uložených nastavení vytvoří konfigurační soubory. Generování bude probíhat na základě dat vybraného scénáře a každý ze souborů si ponese mnoho vlastních rozšiřujících nastavení. Konfigurační soubory mají přímý vliv na sestavování komunikačního řetězu a na jejich správnost spoléhají všechny serverové komponenty. V nepřeberném množství parametrů je velmi snadné udělat chybu, proto by měl být výsledný generátor souborů jednoduchý, přehledný a po stránce kódu dobře čitelný.

2.7.3 Obsluha komunikačního rozhraní

Během zkoumání serverových komponent se ukázalo, že komunikace může být velice časově náročná a některé operace mohou trvat až desítky sekund. To je nepříjemné, protože celý program by byl během těchto operací blokován. Jedná se o klasický problém uživatelských rozhraní. Máme časově náročnou operaci a chtěli bychom, aby během ní zůstalo uživatelské rozhraní použitelné a odpovídalo na další akce uživatelů. To se obvykle řeší více vlákny v aplikaci, jedno se stará o běh uživatelského rozhraní a druhé provádí ony časově náročné operace. Bude tedy nutné navrhnout komunikační model typu *Boss/Worker*, který zajistí provádění uživatelem zadaných operací na pozadí a následné doručování získaných dat zpět do uživatelského rozhraní.

3 Návrh

V předchozích kapitolách jsme se seznámili s výpočetními systémy, jejich omezeními a také těžkostmi, se kterými se potýkají uživatelé při jejich používání. Prošli jsme si také architekturu serverové aplikace a její specifika, která bude nutné při návrhu klienta zohlednit. Nyní bychom tedy měli mít dostatečný nadhled v dané problematice a můžeme začít s návrhem hlavních částí klientské aplikace.

3.1 Tvorba konfiguračních souborů

Začneme nejprve konfiguračními soubory, na kterých si vysvětlíme základní strukturu vytvářených dat. To by nám mělo pomoci později při návrhu uživatelského rozhraní, které bude různorodé volby pro generování souborů reprezentovat. Úkolem rozhraní bude dynamicky vybrat požadovaný scénář spuštění, na základě kterého bude sestaven komunikační řetěz a vygenerovány jednotlivé soubory. Každý výsledný soubor nám reprezentuje právě jednu serverovou komponentu ve výsledném zřetězení.

Pravidla, podle kterých se soubory generují, mohou být poměrně komplikovaná a implementace některých scénářů se může lišit od původního návrhu. Proto se na začátek seznámíme s několika konkrétními scénáři a na jejich základě sestavíme úplný diagram pro libovolnou variantu. Kvůli zjednodušení však zanedbáme některé parametry uvedené v konfiguračních souborech a budeme brát v úvahu jen ty, které jsou nezbytné pro konfiguraci výsledné komunikace.

Níže uvedené konfigurační soubory (kapitola 3.1.1 a 3.1.2) obsahují v klíči `input_type` *a* `output_type` *odkaz na číselník, pro lepší srozumitelnost zde číselník uvádím.*

```
class InputCommType(IntEnum):
    none = 0
    std = 1
    socket = 2
    pbs = 3
```

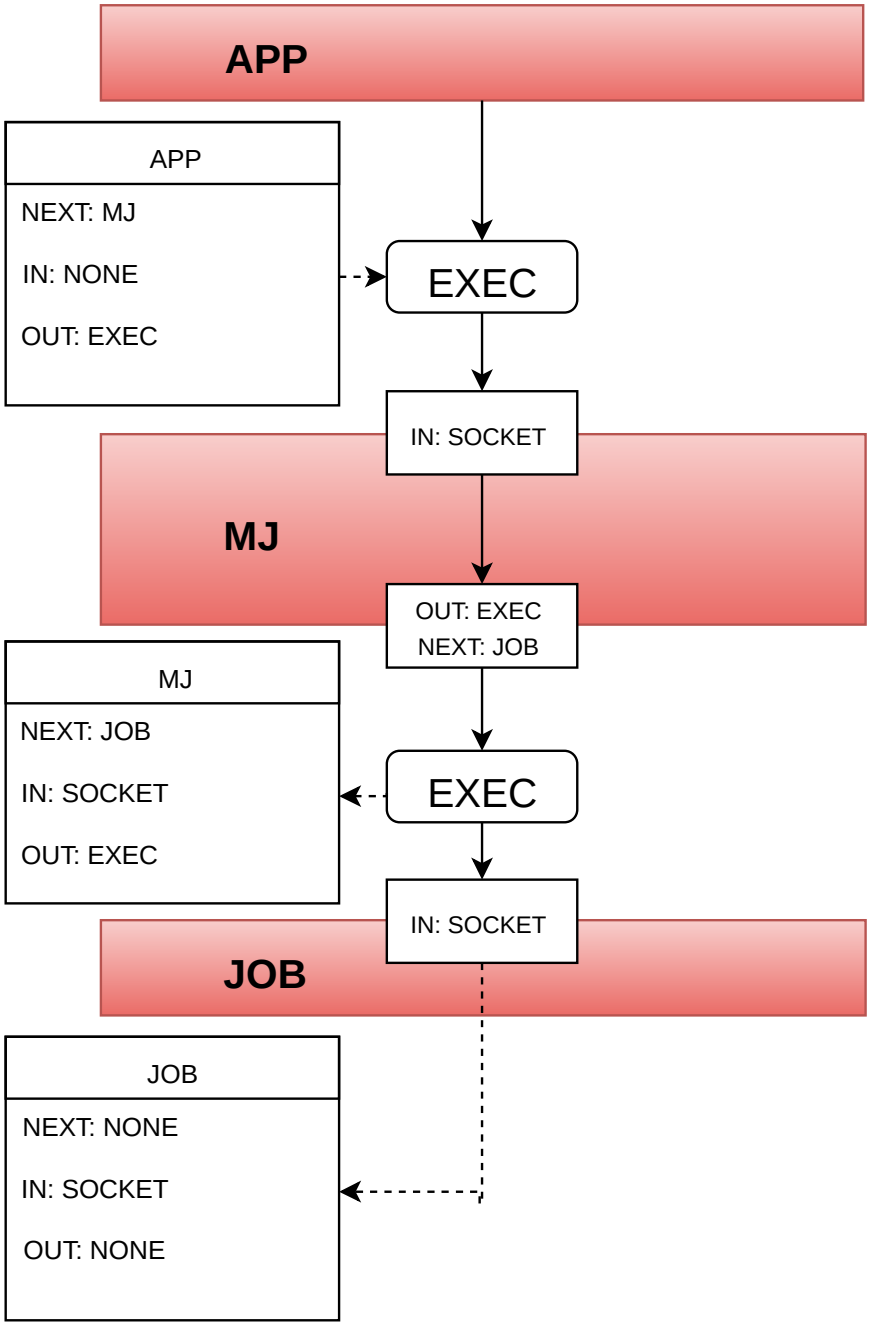
Kód 8: Číselník `InputCommType`

```
class OutputCommType(IntEnum):
    none = 0
    ssh = 1
    exec_ = 2
    pbs = 3
```

Kód 9: Číselník `OutputCommType`

3.1.1 Scénář 1 (klient, klient, klient)

Nejjednoduší variantou je Scénář 1 (klient, klient, klient) z kapitoly 1.7.1. Všechny operace provádí na lokálním počítači a není tedy nutné hned v začátcích přikládat příliš mnoho rozšiřujících nastavení pro vzdálenou komunikaci nebo práci se plánovačem úloh. Z obrázku 18 je možné vyčíst, že jsou vygenerovány tři konfigurační soubory (na obrázku představované bílými obdélníky). Diagram čteme shora a začínáme u vstupní komponenty **`APP`**, odtud je provedeno lokální (*EXEC*) spuštění komponenty **`MULTIJOB`**. Ta následně provádí opět lokální (*EXEC*) spuštění jednotlivých komponent **`JOB`**. Všechny použité přechody jsou typu *EXEC*, to znamená, že každá část aplikace je spouštěna na lokálním počítači.



Obrázek 18: Diagram generování souborů pro scénář číslo 1

Vygenerované soubory pro Scénář 1 (klient, klient, klient) by měly vypadat takto:

`app.json`

```
{
    "communicator_name": "app",
    "next_communicator": "multijob",
    "input_type": 0,
    "output_type": 2,
}
```

Kód 10: Konfigurační soubor `app.json` pro Scénář 1 (klient, klient, klient)

`multijob.json`

```
{
    "communicator_name": "multijob",
    "next_communicator": "job",
    "input_type": 2,
    "output_type": 2
}
```

Kód 11: Konfigurační soubor `multijob.json` pro Scénář 1 (klient, klient, klient)

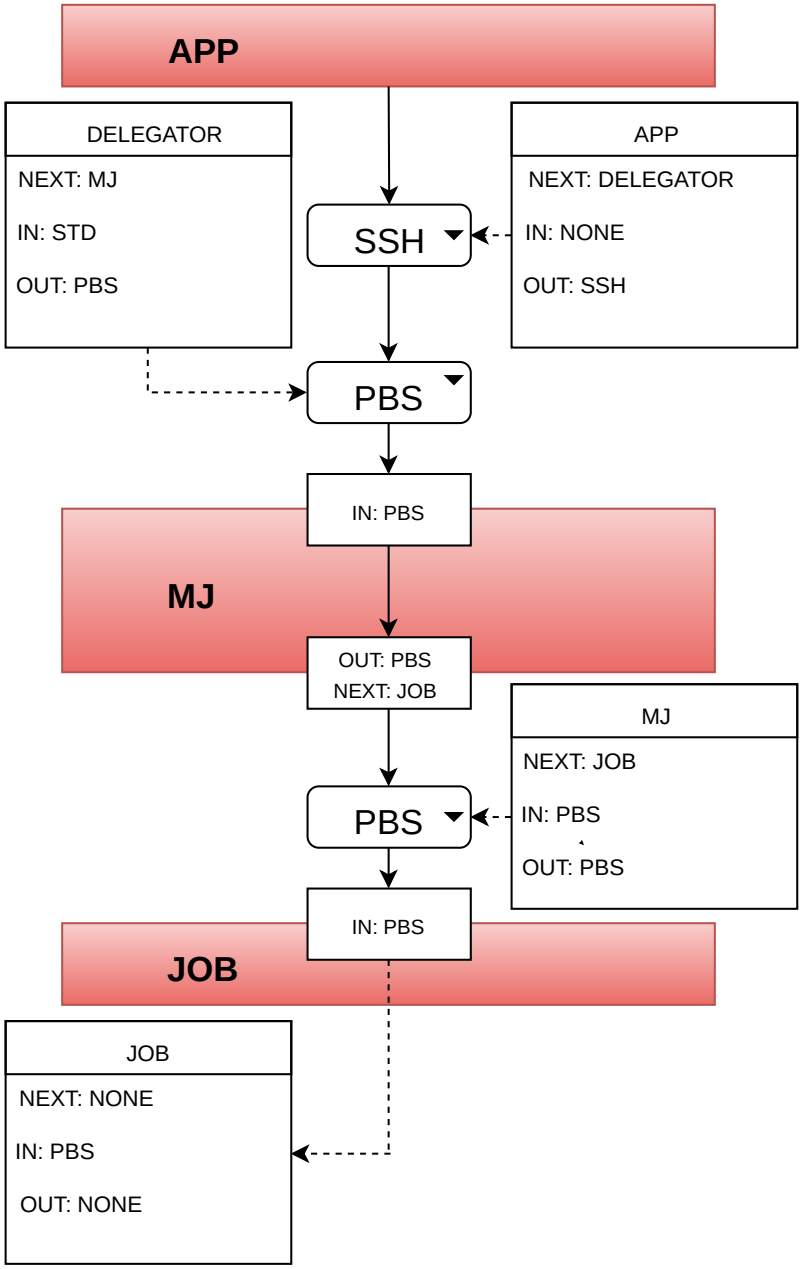
`job.json`

```
{
    "communicator_name": "job",
    "next_communicator": "none",
    "input_type": 2,
    "output_type": 0
}
```

Kód 12: Konfigurační soubor `job.json` pro Scénář 1 (klient, klient, klient)

3.1.2 Scénář 5 (klient, cluster, cluster)

Druhou velmi využívanou variantou je Scénář 5 (klient, cluster, cluster) z kapitoly 1.7.5, tedy klient pouze řídí požadované operace a vše ostatní probíhá na výpočetním clusteru. V tomto případě se nejprve ze vstupní komponenty **APP** připojujeme (*SSH*) na čelní uzel. Odtud je přes **DELEGATOR** komunikováno skrze plánovač (*PBS*) s jednotlivými komponentami **MULTIJOB**, které již běží na samostatných výpočetních uzlech. Jednotlivé komponenty **MULTIJOB** následně, opět přes plánovač (*PBS*), komunikují s jednotlivými komponentami **JOB**, které jsou také spuštěny na samostatných výpočetních uzlech. V tomto návrhu si již můžeme všimnout odlišností od původního scénáře, je zde totiž vygenerován i konfigurační soubor pro rozšiřující komponentu **DELEGATOR**. Ve skutečnosti tedy získáváme čtyři konfigurační soubory (bílé obdélníky), viz Obrázek 19.



Obrázek 19: Diagram generování souborů pro scénář číslo 5

Vygenerované soubory pro Scénář 5 (klient, cluster, cluster) by měly vypadat takto:

`app.json`

```
{
    "communicator_name": "app",
    "next_communicator": "delegator",
    "input_type": 0,
    "output_type": 1,
    "ssh": {}
}
```

Kód 13: Konfigurační soubor `app.json` pro Scénář 5 (klient, cluster, cluster)

`delegator.json`

```
{
    "communicator_name": "delegator",
    "next_communicator": "multijob",
    "input_type": 1,
    "output_type": 3,
    "pbs": {}
}
```

Kód 14: Konfigurační soubor `delegator.json` pro Scénář 5 (klient, cluster, cluster)

`multijob.json`

```
{
    "communicator_name": "multijob",
    "next_communicator": "job",
    "input_type": 3,
    "output_type": 3,
    "pbs": {}
}
```

Kód 15: Konfigurační soubor `multijob.json` pro Scénář 5 (klient, cluster, cluster)

`job.json`

```
{
    "communicator_name": "job",
    "next_communicator": "none",
    "input_type": 3,
    "output_type": 0
}
```

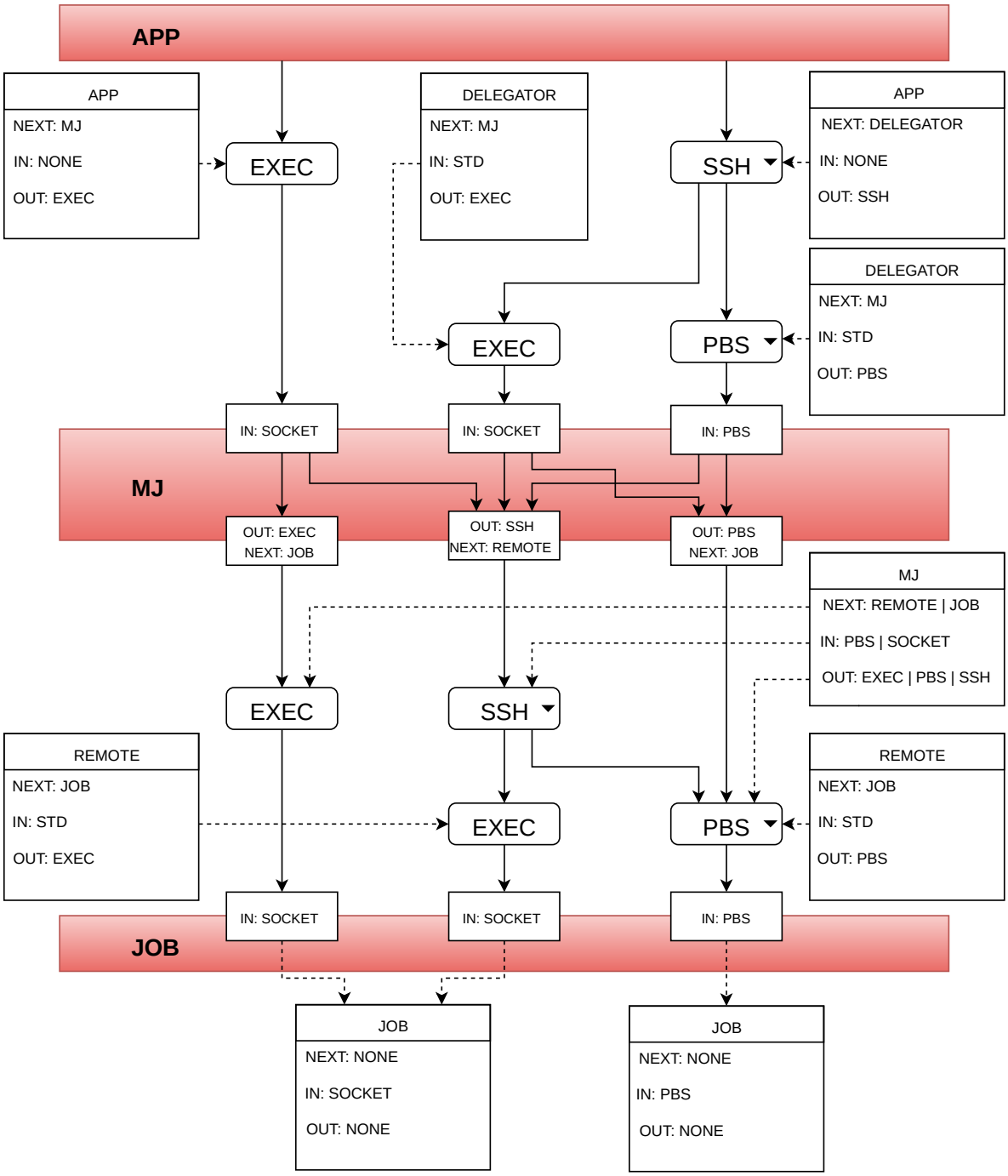
Kód 16: Konfigurační soubor `job.json` pro Scénář 5 (klient, cluster, cluster)

V prvních třech souborech jsou zvýrazněny klíče `ssh` a `pbs`, v těch se budou nacházet rozšiřující nastavení s přihlašovacími údaji pro **SSH** a parametry **PBS** pro komunikaci s plánovačem úloh.

3.1.3 Univerzální postup

Základní konfigurační soubory pro dva nejčastěji používané scénáře již vytvořit umíme, to ovšem nestačí. Generátor souborů by měl obsáhnout všechny požadované scénáře z kapitoly 1.7. Proto jsem navrhl následující diagram, který je mnohem univerzálnější a zohledňuje všechny požadované možnosti. Navíc je možné podle něj generovat soubory flexibilněji a ne jen prostým výběrem scénáře. Obrázek 20 na další straně znázorňuje univerzální diagram pro generování souborů. Z tohoto diagramu by již mělo být zřejmé, jak vygenerovat všechny potřebné konfigurační soubory pro libovolný scénář.

Postupovat budeme obdobně jako v předchozích případech. Celý diagram čteme opět shora dolů a začínat se bude také v komponentě **APP**. Dále postupujeme po plných šipkách a v případě větvení si můžeme vybrat vždy pouze jednu z dostupných cest. Bílé obdélníky nás opět navádějí, jak by měly vypadat výsledné konfigurační soubory. Nejdůležitější je, dát si pozor na správnou posloupnost zřetězení komponent a především pak na korektní spárování komponent vstupního a výstupního rozhraní. V předchozích případech byla cesta vždy pouze jedna, nyní však máme vetší volnost a ve výsledku lze vytvořit celkem šest různých variant. Stále se však jedná o zjednodušené konfigurační soubory a mnoho specifických parametrů zde chybí.



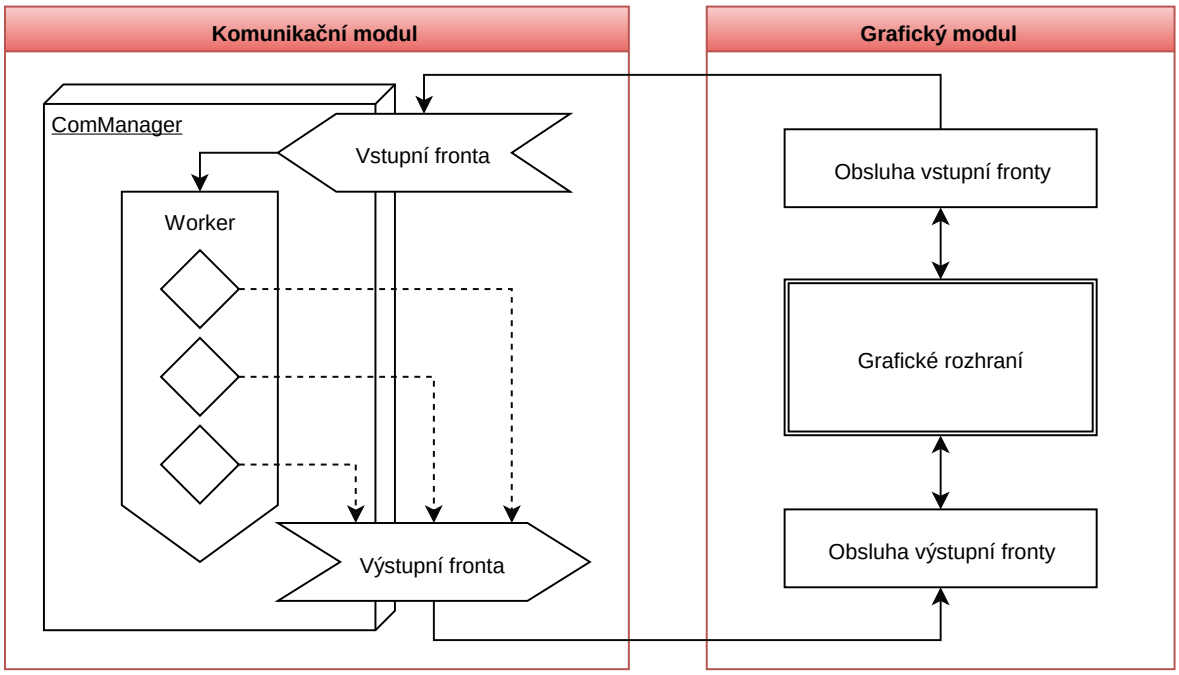
Obrázek 20: Univerzální diagram pro generování konfiguračních souborů

3.2 Serverová komunikace

Aby během komunikace se servery nedocházelo ke zmiňovanému blokování celé aplikace, navrhl jsem model umožňující komunikovat se servery v samostatných vláknech. Celý model je rozdělen na komunikační modul a grafický modul. Mezi nimi je synchronizace vláken prováděna pomocí dvou synchronizačních front, viz Obrázek 21.

Úkolem grafického modulu je odchytávat uživatelem generované události a překládat je na požadavky, kterými je plněna vstupní fronta. Dále pak má kontrolovat výstupní frontu a nalezená data vykreslovat ve správných oknech grafického rozhraní.

Komunikační modul si drží všechny instance třídy `Communicator` a pro každou má své vlastní vlákno. Neustále kontroluje vstupní frontu a jakmile se zde objeví data, tak je předá příslušnému vláknu ke komunikaci. Data jsou ve vlákně zpracována a výsledek je vložen do výstupní fronty, odkud si ho vyzvedne komponenta grafického modulu.



Obrázek 21: Blokové schéma komponenty pro obsluhu serverové komunikace

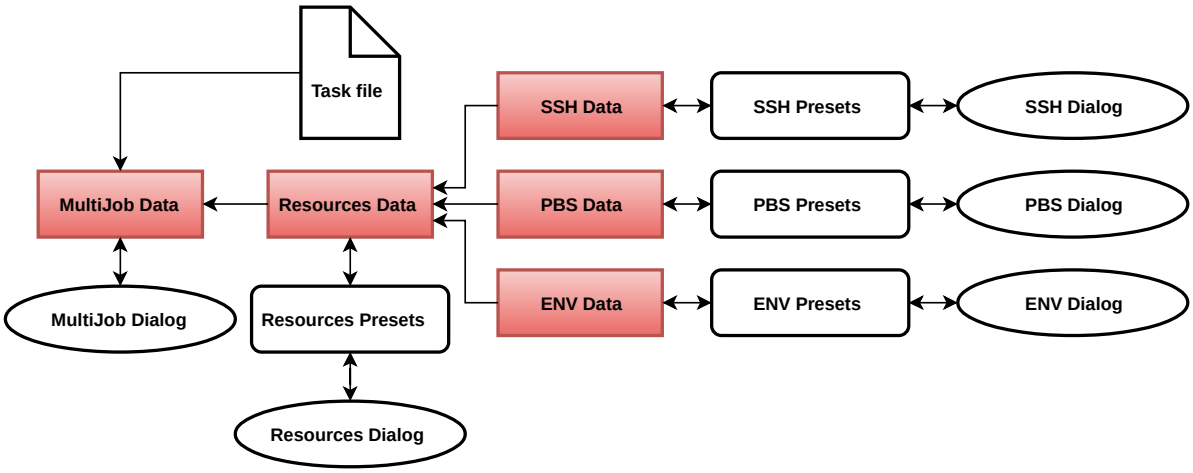
Celá komunikace tedy probíhá následovně. Nejprve putují příslušné požadavky na základě uživatelských akcí do vstupní fronty, do té jsou navíc doplněny časově závislé akce. Například pokud požadujeme cyklicky obnovovat stavová data. Ze vstupní fronty se data dělí mezi jednotlivá vlákna, ve kterých se provádí příslušný typ serverové komunikace. Požadovaný příkaz je ve svém vlákně vykonán a čeká na odpověď. Jakmile odpověď dorazí, tak jsou data vložena do výstupní fronty. Odtud je vyzvedne obsluha výstupní fronty a předá je v náležitém formátu grafickým komponentám k vykreslení.

3.3 Uživatelské rozhraní

Návrh uživatelského rozhraní jsem si ponechal až na konec, slouží totiž jako sjednocující prvek a reaguje na požadavky vzniklé během návrhu ostatních komponent. Kompletní uživatelské rozhraní se skládá ze tří hlavních komponent, které si podrobněji rozebereme v následujících kapitolách.

3.3.1 Nabídky a nastavení

Obrázek 22 znázorňuje strukturu dialogů pro různá nastavení. Při hlubší analýze celého problému se ukázalo, že je v aplikaci nutné spravovat vždy několik profilů pro různá nastavení stejného druhu. Vytvořil jsem tedy koncept takzvaného správce presetů, který umožňuje s jednotlivými profily efektivně pracovat. Tento správce je speciálním dialogem, který po načtení dat zobrazí v řádcích jednotlivé profily požadovaných nastavení, identifikované přiděleným jménem. Chceme-li tedy například editovat dříve nastavený profil pro **SSH**, otevřeme si data profilů pomocí dialogu `SSH Presets`. V tomto dialogu můžeme vybrat z několika akcí (přidat, editovat, odebrat, vytvořit kopii), v našem případě vybereme kurzorem požadovaný profil a stiskneme tlačítko editovat. To zapříčiní, že se zobrazí `SSH Dialog`, ve kterém je možné editovat požadované parametry a následně profil opět uložit.



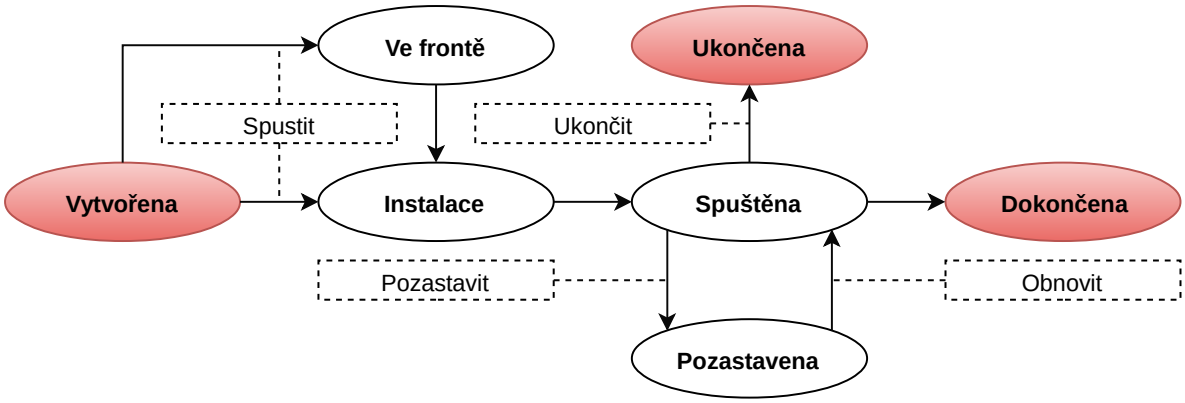
Obrázek 22: Struktura dialogů a nastavení

Takto vytvořená data základních profilů (*SSH, PBS, ENV*) nám umožní nakonfigurovat nové běhové prostředí. Toho je možné docílit vyvoláním dialogu `Resources Presets`, v němž zvolíme akci přidat. Otevře se nám prázdný `Resource Dialog`, do kterého nastavíme data pro požadovaný scénář spouštění. Konfigurace scénáře probíhá dynamicky (viz Obrázek 20 v kapitole 3.1.3) a k jednotlivým krokům komunikace je možné přiřadit rozšiřující nastavení. Pokud tedy například komunikujeme se vzdáleným serverem, tak přiřadíme příslušný profil pro **SSH** z nabídky. Výsledné nastavení si uložíme, čímž jsme vytvořili univerzální profil běhového prostředí podle námi vybraného scénáře. Takto vytvořené profily jsou nezávislé na konkrétní úloze a je tak možné stejnou úlohu snadno spouštět podle libovolného scénáře.

Úlohu je možné vytvořit pomocí dialogu `MultiJob Dialog`, ten mimo jiných obsahuje tři důležité parametry. Jednak je to orientační jméno, které budeme v systému pro orientaci uživatelů zobrazovat, potom je to cesta k souboru se zadáním úlohy, zmíněno v kapitole 2.1, a pak běhové prostředí, které je možné navolit z dostupných profilů, nastavených v předchozím odstavci. Takto vytvořená úloha se objeví v hlavním okně aplikace, odkud jí můžeme pomocí příkazů ovládat.

3.3.2 Ovládání úlohy

Každá úloha po uložení začíná ve stavu `vytvořena`, viz Obrázek 23. Po spuštění se úloha dostává do jednoho ze dvou stavů. Pokud nepoužíváme systém s plánovačem úloh, přechází rovnou do stavu `instalace`. V opačném případě je nutné nejprve vyčkat `ve frontě` na přidělení požadovaných zdrojů. Jakmile jsou požadované zdroje přiděleny, pokračuje se automaticky do stavu `instalace`. Během `instalace` se překopírují všechny potřebné soubory a spustí se požadované serverové komponenty. Pokud vše proběhne bez problému, pokračuje úloha do stavu `spuštěna` a zahájí se zpracovávání jednotlivých podúloh. V této fázi lze zpracovávání pozastavit nebo v případě špatného směřování výpočtu úplně zastavit. Po dokončení všech zařazených podúloh jsou jednotlivé serverové komponenty postupně odstaveny a úloha přechází do stavu `dokončena`.

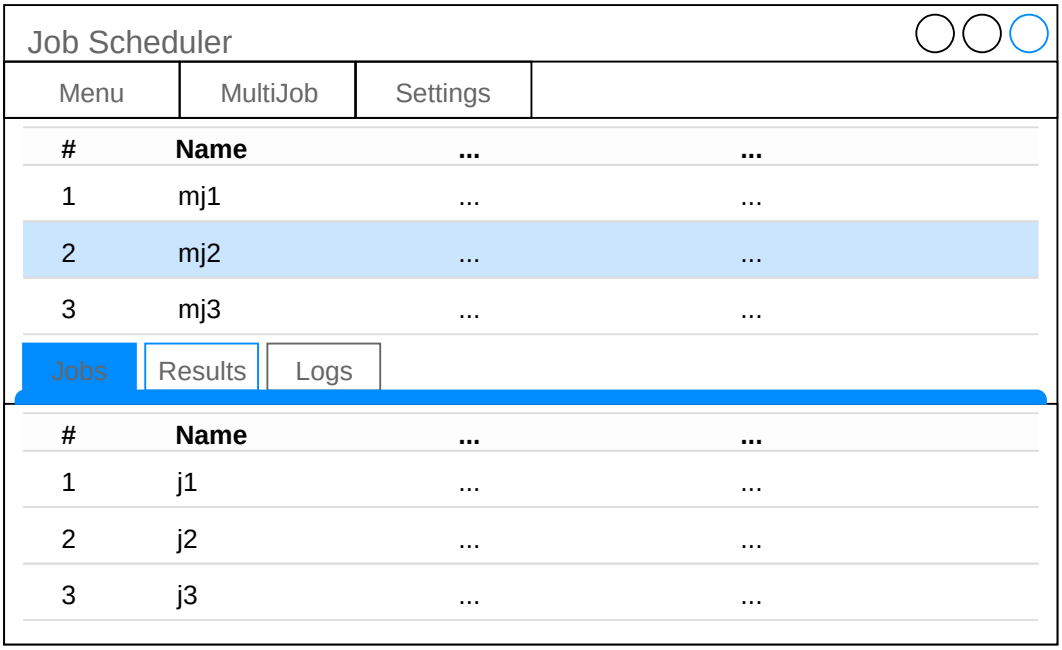


Obrázek 23: Životní cyklus úlohy

Abychom mohli úlohy ovládat a zajistit tak přechod mezi některými stavy, budeme v menu aplikace potřebovat některé základní ovládací prvky. Ty jsou na obrázku znázorněné přerušovanými obdélníky. Není však přípustné, aby bylo možné tyto ovládací prvky používat mimo povolený stav. To by způsobovalo chyby a nekonzistenci dat. Výše uvedený diagram nám tak poslouží i k blokování grafického rozhraní pro akce, které nejsou v aktuálním stavu povolené. Kromě výše znázorněných akcí by tedy nemělo být možné ani úlohu editovat, pokud právě probíhá.

3.3.3 Hlavní okno

Ve vrchní části hlavního okna je umístěno proužkové menu, v němž nalezneme tři rozevírací podnabídky. Všechna nastavení zmíněná v kapitole 3.3.1 jsou zařazena do nabídky `Settings`, tedy kromě dialogu pro konfiguraci úloh. Ten je spolu s ovládáním úlohy, uvedeným v kapitole 3.3.2, zařazen do nabídky `MultiJob`. Co se `Menu` týká, to v této fázi obsahuje pouze akci pro ukončení celé aplikace, není však žádný problém ho případně rozšířit dle potřeb.



Obrázek 24: Návrh hlavního okna aplikace

Hlavní část celého okna je rozdělena na dvě poloviny, viz Obrázek 24. V horní polovině se nachází seznam všech zadaných úloh spolu s informacemi o jejich stavu. Po zvýraznění určité úlohy (řádku) pomocí kurzoru je možné danou úlohu ovládat, a to buď příkazy z nabídky `MultiJob` nebo přímo pomocí klávesových zkratek. V dolní polovině okna jsou pak, v závislosti na právě vybrané úloze, zobrazeny rozšiřující informace pro tuto úlohu. Informace jsou dle jejich typu členěny do několika záložek, kterými je možné v průběhu výpočtu libovolně listovat. Na obrázku výše je vybrána záložka Jobs, která nám poskytuje informace o jednotlivých probíhajících podúlohách. V záložce Results budou po skončení zobrazeny výsledky úlohy a v Logs nejdeme logovací soubory jednotlivých serverových komponent. V budoucnu lze očekávat, že se nabídka rozšíří o záložky se speciálními funkcemi pro nějaký specifický typ úlohy. Data všech těchto záložek jsou spolu se stavem úlohy průběžně obnovována na základě komunikace se serverovou částí aplikace.

4 Implementace

Výsledná implementace je mnohem bohatší, než jak je možné popsat v této práci. Proto si na následujících několika stranách detailněji projdeme implementaci pouze těch nejzásadnějších celků, které byly představeny během návrhu. Spojovací logika a některé méně důležité části, jsou pro zjednodušení záměrně vynechány. K hlubšímu studiu výsledné implementace poslouží pro případné zájemce repozitář projektu [14] s kompletními zdrojovými kódy aplikace.

4.1 Použité technologie

Použité technologie se netýkají pouze kontextu Jobs, ale reflektují i potřeby ostatních kontextů se kterými bude výsledná aplikace distribuována jako celek. Bylo by tedy velice neuvážené, kdyby se pro každý kontext muselo instalovat jiné podpůrné prostředí. Požadavky proto budeme zvažovat z tohoto podhledu souhrnně.

Základním požadavkem je nezávislost na operačním systému. Někteří budoucí uživatelé pracují na operačním systému *Windows* a jiní zase na *Linuxu*. Zvolená platforma by tedy měla rozumně podporovat oba tyto operační systémy. V potaz je nutné brát i uživatelské rozhraní. To se bude vyvíjet pouze jednou a mělo by se tedy na obou cílových systémech chovat stejně a vypadat jednotně.

Nelze opomenout ani fakt, že část aplikace pracuje na unixových serverech. Zvolená platforma by tedy měla být dobře podporována ze strany výpočetních systémů. Důležitá je zejména její role v celém tomto ekosystému a případné podpůrné nástroje, které bude možné integrovat či jinak využít. Systém postavený například na platformě *.Net* by tedy rozhodně nedával žádný smysl.

Konečná volba nakonec padla na programovací jazyk **Python [15]**, který je dobře podporován na obou požadovaných operačních systémech. Jedná se o jazyk interpretovaný, což je pro náš případ poměrně příhodné. Aplikaci lze skládat pouze kopírováním souboru, rychle prototypovat a snadno opravovat vzniklé chyby. Konkrétně je použita verze 3, pro kterou je dostupná kvalitní integrace (**PyQt5 [17]**) s velmi rozšířenou a oblíbenou multiplatformní grafickou knihovnou **Qt5 [16]**. Tato knihovna je napsána v jazyce **C**, což zaručuje dostatečný výkon i pro náročnější aplikace a také optimální integraci s jazykem **Python**. Takto zvolené řešení by tedy mělo vyhovovat požadavkům všech kontextů.

4.2 Generátor konfiguračních souborů

I když se může na první pohled zdát implementace generátoru konfiguračních souborů poměrně snadným a přímočarým úkolem, není to tak úplně pravda. Celou implementaci jsem prováděl na dvakrát, v první verzi jsem si zvolil celkem jednoduché datové struktury, což vedlo na poměrně složitou kaskádu vnořených podmínek. To by samo o sobě nebylo tolik problematické, veškerou potřebnou funkcionalitu jsem odladil a vše fungovalo tak, jak mělo. Problém ale nastal, když jsem byl později nucen se k napsanému kódu vrátit a zapracovat změny, které nastaly během vývoje serverové části. Zapracování kterékoliv změny bylo velmi pracné a ovlivnilo to vždy několik dalších podmíněných větví programu. Když se mi nakonec podařilo vše opravit a otestovat, tak bylo kvůli dalším nenadálým změnám nutné se vrátit o několik kroků zpět a celý proces zopakovat. Kód byl tou dobou velice komplikovaný a složitě napojovaný a jakákoliv snaha ho upravit byla neúměrně náročná. Rozhodl jsem se ho tedy kompletně předělat.

Dlouhou dobu jsem hledal vhodné řešení, které by bylo dobře čitelné a odstranilo kaskády podmínek. Poměrně uspokojivě čitelný se mi zdál přístup používaný například při tvorbě databázových dotazů v objektovém prostředí. Tento přístup se nazývá *method chaining* [18] a v prostředí databází se pro objekty typu `QueryBuilder` využívá spíše jeho expresivnější varianta *fluent interface*. Funguje tak, že metody typu set vrací jako návratovou hodnotu referenci sama sebe. Toto chování umožňuje metody libovolně řetězit za sebe, což zjednodušuje a zpřehledňuje proces nastavování při velkém množství požadovaných parametrů objektu. Výsledná třída `ConfBuilder` je uvedena v následující ukázce.

```
class ConfBuilder:
    def __init__(self, basic_conf):
        self.conf = copy.deepcopy(basic_conf)

    def set_comm_name(self, comm_name):
        self.conf.communicator_name = comm_name.value
        return self

    def set_next_comm(self, next_comm):
        self.conf.next_communicator = next_comm.value
        return self

    def set_in_comm(self, in_comm):
        self.conf.input_type = in_comm
        return self
    def set_out_comm(self, out_comm):
        self.conf.output_type = out_comm
        return self

    def set_ssh(self, ssh_conf):
        self.conf.ssh = ssh_conf
        return self

    def set_pbs(self, pbs_conf):
        self.conf.pbs = pbs_conf
        return self

    def get_conf(self):
        return self.conf
```

Kód 17: Třída `ConfigBuilder`

Podmíněné větvení nebylo sice možné zcela odstranit, ale na základě znalostí z předchozí implementace a optimalizace podmínek se podařilo kaskády výrazně zjednodušit a celý kód znatelně zpřehlednit. V následující ukázce kódu je popsán celý postup vytváření konfiguračního souboru.

```
# vytvoření nové instance z kopie základní konfigurace
delegator = ConfBuilder(basic_conf)

# základní nastavení komponenty delegator
delegator.set_comm_name(CommType.delegator)\
    .set_next_comm(CommType.multijob)\
    .set_in_comm(InputCommType.std)

# podmíněné parametry v závislosti na datech požadovaného scénáře
if mj_remote_execution_type == UiResourceDialog.EXEC_LABEL:
    delegator.set_out_comm(OutputCommType.exec_)
    mj.set_in_comm(InputCommType.socket)

# podmíněné parametry v závislosti na datech požadovaného scénáře
elif mj_remote_execution_type == UiResourceDialog.PBS_LABEL:
    delegator.set_out_comm(OutputCommType.pbs)\
        .set_pbs(mj_pbs)
    mj.set_in_comm(InputCommType.pbs)

# příprava datového kontejneru pro zapsání do souboru
delegator_config = delegator.get_conf()
```

Kód 18: Kód pro vytváření konfiguračních souborů

Jako první je vytvořen objekt `delegator`, který přebírá obecné vlastnosti z objektu `basic_conf`. Následně se nastaví některé parametry, které jsou pevně svázané s typem komponenty. Původně to bylo řešeno automaticky podle typu komponenty, snižovala se však jen čitelnost kódu a komplikovalo se výsledné generování souboru. V dalším kroku jsou přiřazeny parametry v závislosti na vybraném scénáři. Kvůli úsporám jsou ve stejné podmínce nastaveny parametry i pro objekt `mj` (**`MULTIJOB`**), který je napojen na výstup komponenty **`DELEGATOR`** a je tak ovlivněn stejnou logikou. Nakonec jsou všechny parametry vráceny jako jeden datový kontejner, který je uložen do souboru ve formátu **JSON**.

4.3 Správce serverové komunikace

Můj návrh počítá s prací v několika vláknech, to se však ukázalo jako velice problematické při krokování aplikace a průběžném sledování vnitřního stavu. Abych tyto problémy překonal, rozhodl jsem se postupovat od nejmenších možných celků směrem výše, ke složeným komponentám. To se mi velmi osvědčilo a byl jsem tímto způsobem schopen celý návrh úspěšně implementovat. Výsledná implementace se nijak zásadně neliší od původního návrhu. K drobným změnám však došlo kvůli integraci komponent grafické knihovny **Qt**, ty mají jistou vnitřní logiku a způsob použití. To jsem se ve své implementací snažil respektovat. Dalším drobným rozdílem je implementace vstupní fronty, ta je realizována jednotlivě pro každou úlohu. Pokud tedy mluvíme o vkládání požadavku do vstupní fronty, ve skutečnosti jde o vytvoření požadavku a vložení do konkrétní fronty pro danou úlohu.

4.3.1 Obsluha vstupní fronty (ReqHandler)

Obsluha vstupní fronty je mou implementací časovače z knihovny **Qt**. (`QTimer`). Každý takový časovač má vlastní událost `timeout`, která se vyvolá vždy po uplynutí nastaveného času. Na tuto událost je možné připojit zpracování libovolné metody. V našem případě v této metodě procházíme všechny aktivní úlohy a do vstupní fronty zařazujeme **požadavky** (`ReqData`) na obnovení stavových dat. Pokud je navíc některá úloha aktivně vybrána, provede se kompletní obnova všech jejich stavových dat a to včetně rozšiřujících informací.

Požadavek je objektem s touto strukturou:

```
class ReqData(object):
    def __init__(self, key, com_type, data=None):
        # unikátní identifikátor v rámci aplikace
        self.key = key
        # typ požadavku
        self.com_type = com_type
        # případná data pro zpracování
        self.data = data
```

Kód 19: Definice třídy s daty požadavku

4.3.2 Obsluha komunikace (ComManager)

Hlavní komponentou celé komunikace je `ComManager`. Ten si drží instance všech výkonných částí pro komunikaci a také výstupní frontu. Jeho hlavní funkcí je vytváření komunikačních požadavků pro jednotlivé výkonné části. Zavoláním příslušné metody se vytvoří objekt (`ReqData`) s požadavkem na komunikaci a vloží se do vstupní fronty požadované výkonné části. Například zavoláním metody `resume()` (obnovení) se provede následující kód.

```
def resume(self, key):
    worker = self._workers[key]
    req = ReqData(key=key, com_type=ComType.resume)
    worker.req_queue.put(req)
```

Kód 20: Ukázka vytvoření požadavku

4.3.3 Výkonná část (ComWorker)

Každá úloha má samostatnou výkonnou část, která disponuje vlastní vstupní frontou. Výkonná část neustále v cyklu prochází svou frontu a objekty s požadavky překládá na volání metod komunikační komponenty. Výsledky komunikace pak ukládá do nově vytvořeného objektu s odpovědí, který je umístěn do výstupní fronty, čímž je zpracování jednoho požadavku dokončeno.

4.3.4 Obsluha výstupní fronty (ResHandler)

Obsluha výstupní fronty je opět implementací časovače. (`QTimer`) Ve své metodě na událost `timeout` kontroluje výstupní frontu. V závislosti na nalezeném objektu s odpovědí (`ResData`) pak vyvolává své události s vyzvednutými daty v požadovaném formátu. Na tyto události je napojeno uživatelské rozhraní, které data přebírá a zařizuje vykreslování ve správných oknech aplikace.

Objekt s odpovědí má následující strukturu:

```
class ResData(object):
    def __init__(self, key, com_type, mess=None, err=None, data=None):
        # unikátní identifikátor v rámci aplikace
        self.key = key
        # typ požadavku
        self.com_type = com_type
        # odpověď vzdáleného systému
        self.mess = mess
        # případné detaily chybového hlášení
        self.err = err
        # přiložená data
        self.data = data
```

Kód 21: Definice třídy s daty pro odpovědi

4.4 Uživatelské rozhraní

Klíčovou částí implementace uživatelského rozhraní je správná integrace klientské aplikace s knihovnou **Qt**. V návrhu uživatelského rozhraní se nachází hned několik nabídek, které mají velice podobné chování. Kladl jsem tedy důraz na oddělení společné logiky a uživatelského rozhraní do vlastních tříd, které by měly poskytovat dostatečně robustní základ pro konkrétní implementace různých dialogů. Integrace se standardně provádí pomocí dědění od základních komponent knihovny **Qt**. V jazyce **Python** je však možná vícenásobná dědičnost a tak lze požadovaného výsledku dosáhnout více způsoby. Možnosti napojení grafických komponent na řídicí logiku a pokročilé využití dědičnosti je podrobněji vysvětleno v dokumentaci pro podpůrnou knihovnu **PyQt** [19].

Ve své implementaci jsem si zvolil strukturu, ve které si třída s řídící logikou drží instanci uživatelského rozhraní. V následující ukázce je struktura naznačena, tento návrh je velice flexibilní a grafickým rozhraním lze pomocí metody `setup_ui()` obalit libovolný `Qdialog`. To je v naší situaci velmi výhodné, umožňuje nám to používat stejný kód pro společné části a jednotlivé rozdíly vyřešit až v konkrétní implementaci. Navíc jsou všechny grafické komponenty zařazeny pod `ui`, což je praktické, logické a ve výsledném kódu dobře čitelné.

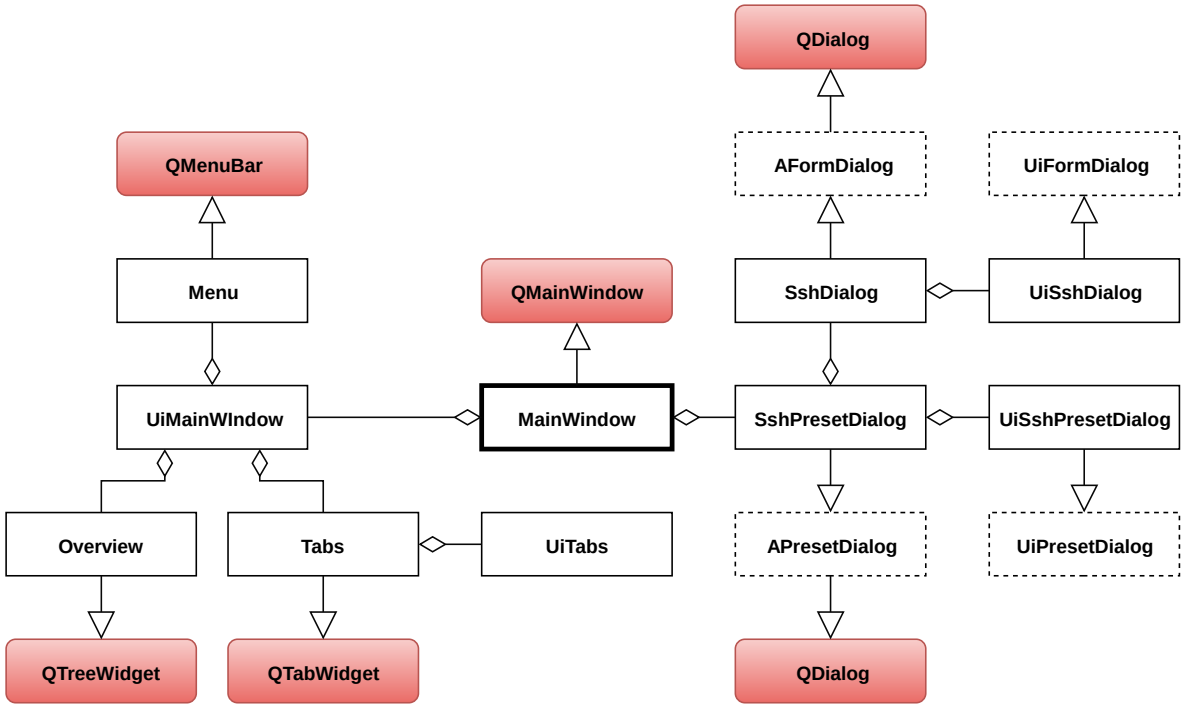
```
class SshDialog(AFormDialog):
    def __init__(self, parent=None):
            super().__init__(parent)
            self.ui = UiSshDialog()
            self.ui.setup_ui(self)
```

Kód 22: Kompozice uživatelského rozhraní

Pro podobné nabídky jsem se rozhodl vytvořit abstrakční vrstvu. Děděním od třídy `QDialog`, získáme abstraktní formulářový dialog `AformDialog`, ten slouží jako základ ostatním formulářům pro zadávání dat. Jsou v něm implementovány společné funkce a navíc obsahuje i abstraktní metody, které mají být v konkrétní implementaci překryty. Díky tomu nemusíme některé části kódu opakovat a máme zajištěn jednotný přístup například k validaci dat, jejich vyzvedávání z formuláře či přednastavování načtených hodnot. Stejné sjednocení je žádoucí i v uživatelském rozhraní. Na to nám slouží třída `UiFormDialog`, která je kostrou definující výchozí vzhled příslušného dialogu. Nejsou zde žádná datová políčka, jde nám spíše o základní velikost okna, rozložení jednotlivých položek, rozmístění tlačítek a další společné předvolby.

Požadovaný formulář pak získáme pouhým poděděním navržených tříd do konkrétní implementace, ve které doladíme specifický vzhled a chování. Složením nově vzniklých tříd (dle ukázky kódu výše) získáváme výslednou formulářovou komponentu. Obdobným postupem můžeme vytvořit i správce souborů ze tříd `APresetDialog` a `UiPresetDialog`. Graficky je celá implementace znázorněna níže, viz Obrázek 25.

Obrázek 25 znázorňuje diagram použitých tříd. Červené obdélníky se zaoblenými rohy reprezentují třídy knihovny **Qt**, od kterých jsem podědil mou implementaci, která je znázorněna bílými obdélníky. V kapitole 3.3.3 jsme si popsali strukturu hlavního okna, ta je znázorněna na obrázku 25 vlevo. Všechny tři hlavní části jsou realizovány jako samostatné komponenty, které vykresluje uživatelské rozhraní hlavního okna `UiMainWindow`. Instanci tohoto uživatelského rozhraní si pak drží třída `MainWindow`, ve které je implementována veškerá řídící logika.



Obrázek 25: Class diagram implementace uživatelského rozhraní

Z pravé strany se do hlavního okna napojují jednotlivé konfigurační dialogy, pro zjednodušení je zde zakreslena pouze struktura **SSH** dialogů. V dolní části děděním `APresetDialog` s využitím grafického rozhraní `UiPresetDialog` získáváme komponentu `SshPresetDialog`. Nahoře vpravo pak ze třídy `AFormDialog` spolu s `UiSshDialog` vzniká komponenta `SshDialog`. Po správném sestavení komponent se na kliknutí v příslušné nabídce hlavního okna otevře dialog s **SSH** presety (`SshPresetDialog`). Po volbě přidat se zobrazí **SSH** dialog (`SshDialog`) pro vyplnění potřebných dat. Následným uložením dialogu nám vznikne profil, který lze v aplikaci dále využívat. Ostatní dialogy jsou realizovány obdobně a napojení na abstraktní komponenty zůstává stejné, mění se pouze konkrétní implementace.

5 Testování

Součástí každé realizace je otestování jednotlivých dílčích částí i výsledného celku. Jelikož je velká část aplikace tvořena uživatelským rozhraním, zvolil jsem si systémové testy. Tento typ testů ověřuje aplikaci z pohledu koncového uživatele. Jako uživatel si nastavíme určitá kriteria a ta budeme následně vyhodnocovat. Celá aplikace se tak stane takzvanou černou skříňkou, která reaguje na naše vstupy patřičnými výstupy.

Než začneme s testováním, je třeba uvést dvě drobná zjednodušení, kterých jsem se dopustil. V našem případě nebudeme testovat spouštění reálných výpočtů, to by bylo zbytečně složité a není to v tuto chvíli ani potřeba. Cílem totiž není ověřit výpočetní výkon cílových strojů, ale otestovat uživatelské rozhraní a spouštěcí logiku. Vystačíme si tedy s jednoduchou testovací úlohou, která v krátkých časových intervalech vypisuje počet vteřin od startu a v mezičase se hlavní vlákno uspává. To probíhá tak dlouho dokud nevyprší předem stanovený časový limit. V některých scénářích je navíc využívána komunikace na vzdálený server. Pro zjednodušení však nepoužijeme fyzický server, ale aplikace bude komunikovat přes **SSH** zpět na lokální počítač pomocí jiného uživatelského účtu. Jedná se o jednoduchý trik, který nám velice usnadní testování, pro aplikaci je tato zkratka od vzdálené komunikace téměř k nerozeznání.

V současném stavu je serverová aplikace poměrně hodně svázána s univerzitním clusterem Hydra, testování s použitím výpočetního systému tedy budeme provádět právě zde. Do budoucna se ale počítá s postupnou abstrakcí některých dílčích částí serverové aplikace tak, aby bylo možné využít libovolný z dostupných clusterů. V tomto stavu prozatím nejsme, není to však výrazný problém. Většina omezení vychází ze současného stavu serverové části, z toho důvodu nemá smysl se tímto problémem dále zaobírat.

5.1 Příprava

Pro přípravu dat jsem se rozhodl využít uživatelské rozhraní. Sice ztratím oproti manuálnímu sepsání konfiguračních souborů část kontroly nad konečným výsledkem, získám však lepší představu o použitelnosti implementovaného rozhraní. To mi pomůže například zjistit, zda-li jsou nabídky logicky uspořádány a výsledné nastavení odpovídá požadovanému chování. Je zřejmé, že autor není nikdy schopen testovat stejně jako nezaujatý uživatel. Jedná se pouze o pomyslný první krok, další úpravy bude nutné provést po důkladnějším testování více uživateli, které ale před dokončením diplomové práce provedené nebylo.

Ze všeho nejdříve jsem si nastavil **SSH** profily pro vzdálenou komunikaci. Pro otestování všech scénářů budeme potřebovat dva. Jeden bude sloužit pro lokální komunikaci (náhrada serveru) přes jiný uživatelský účet v systému a druhý pro univerzitní cluster Hydra [3], na kterém je nutné mít předem vytvořený uživatelský účet. S tím nebyl sebemenší problém, drobné problémy se vyskytly až při vytváření profilů s parametry pro plánovač úloh. V prvotních testech se parametry nijak neprojevovaly a úloha se vůbec nezařadila ke zpracování. Po hlubším průzkumu se ukázalo, že potřebné volby asi nejsou podporovány a všechny úlohy se musí řadit do jedné hlavní fronty, jinak odeslání úlohy selže. Absence této podpory je z hlediska poměrně nízké vytíženosti clusteru celkem pochopitelná. Profil s parametry jsem tedy nechal nevyplněný, příkaz `qsub` se tak bude volat bez parametrů a musíme se spokojit s výchozím nastavením fronty na clusteru. S přihlédnutím k testovací úloze nám beztak postačí v podstatě libovolný přidělený výkon.

V dalším kroku následovalo nastavování běhových prostředí pro jednotlivé úlohy. Podle scénářů uvedených v kapitole 1.7 jsem jich sestavil všech šest. Nakonec jsem vytvořil jednotlivé testovací úlohy a nastavil jsem jim parametry, jako jeden z parametrů slouží příslušné běhové prostředí. Kvůli přehlednosti jsou jednotlivé úlohy pojmenovány podle odpovídajícího scénáře.

5.2 Kritéria

Při testování se zaměříme na schopnost aplikace úspěšně dokončit spuštěnou úlohu. Libovolnou úlohu můžeme považovat za úspěšně dokončenou jen tehdy, pokud si projde celým životním cyklem až do koncového stavu *ukončeno* nebo *dokončeno*. Tyto stavy mohou nastat pouze tehdy, jsou-li splněna všechna níže stanovená kriteria. Podrobněji o jednotlivých stavech úlohy pojednává kapitola 3.3.2.

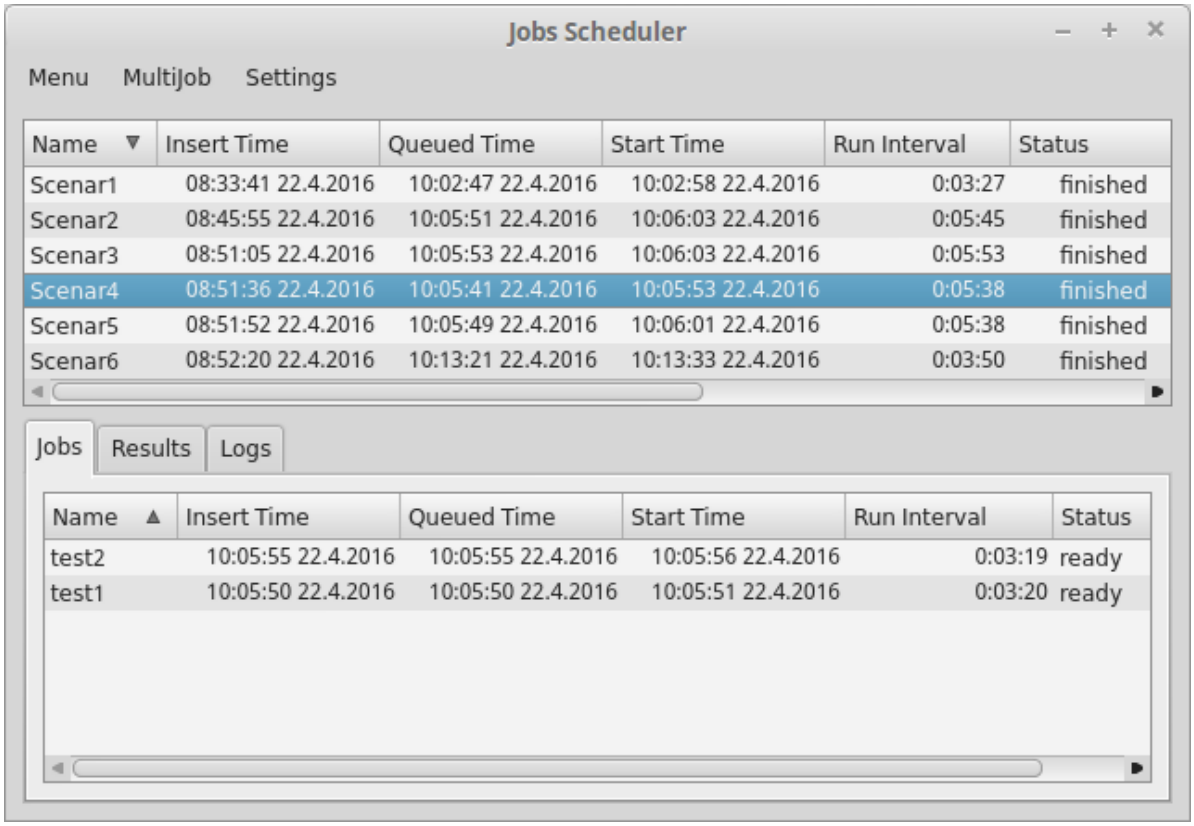
Stanovená kriteria:

1) Aplikace překopíruje všechny potřebné soubory na vzdálený systém a spustí se.
2) Komponenta pro řízení úloh je dostupná ke komunikaci a odpovídá na požadavky.
3) Jednotlivé dílčí podúlohy jsou zařazeny ke zpracování.
4) Komunikaci lze bez problému pozastavit.
5) Po obnovení komunikace aplikace naváže na předchozí činnost.
6) Po dokončení přejde úloha do koncového stavu a jsou dostupné výsledky.

Zhodnocením těchto kriterií ovšem nelze považovat problematiku testování za vyčerpanou, vždy bychom mohli jít více do detailu a zabývat je jednotlivými komponentami. Cílem je však ověřit základní funkčnost požadovaných scénářů. A i přesto, že stanovená kriteria mohou působit jednoduše, ve skutečnosti musí mnoho dílčích součástí správně fungovat, aby mohl být každý z bodů naplněn.

5.3 Zhodnocení

Jakmile jsem měl všechna nastavení vyladěna, začal jsem s testováním. Spouštěl jsem postupně různé scénáře a pečlivě jsem sledoval jejich průběh, a to jak na lokálním počítači, tak na výpočetním clusteru. Ne všechny scénáře však fungovaly od začátku bezproblémově a mnohokrát se stalo, že úloha neznámo proč uvázla v nějakém stavu. Jediným způsobem, jak daný problém odstranit, bylo projít si důkladně všechny logy a na základě získaných informací zkusit pozměnit parametry v nastavení. Získané chybové zprávy však často neměly moc velkou informační hodnotu a původ chyby jsem spíše odhadoval, což mi značně ztěžovalo práci. Když jsem po několika neúspěšných pokusech všechna špatná nastavení náležitě opravil a úlohy znovu spustil, začalo vše fungovat dle mého očekávání. Všechny scénáře se tak i přes počáteční obtíže podařilo úspěšně zrealizovat. Implementované uživatelské rozhraní s dokončenými úlohami znázorňuje Obrázek 26.



Obrázek 26: Hlavního okno klientské aplikace s dokončenými testovacími úlohami

Závěr

Výsledkem této diplomové práce je klientská aplikace, která uživatelům pomáhá se zpracováním náročných úloh na různorodých výpočetních systémech. Grafické rozhraní umožňuje nastavit všechny potřebné konfigurační profily a zadat data úlohy určené ke zpracování. Uživatel pak z nabídek jednoduše vybere, ve kterém z nastavených běhových prostředí se má úloha vykonat a potom jí jedním příkazem spustí. Po dokončení úlohy není nutné výsledky složitě dohledávat, samy se automaticky zobrazí v příslušném okně aplikace. Odpadá tak nutnost pracovat s úlohami přímo v příkazovém řádku a ručně si psát komplikované spouštěcí skripty. Od toho všeho je běžný uživatel kompletně odstíněn a zpracování celé úlohy je možné pohodlně obsloužit z uživatelského rozhraní. Čas strávený obsluhou jednotlivých úloh je tak redukován na minimum, což budoucím uživatelům významně šetří práci. Navíc je díky aplikaci sjednocen přístup k různým zdrojům výpočetního výkonu, které tak mohou být efektivněji využity.

V průběhu vypracovávání této práce jsem narazil na mnoho dílčích těžkostí. Na počátku jsem se hodně potýkal s nepochopením jednotlivých komunikačních scénářů, které jsou ale pro celou práci stěžejní a přináší sebou mnoho neznámých. Pokud má tedy být celá aplikace dostatečně univerzální, je třeba věnovat se detailně jejich analýze. Důkladné pochopení celé problematiky mi zabralo několik týdnů, během kterých jsem se k problému mnohokrát vracel a překresloval své původní návrhy. Nakonec se mi podařilo do problematiky dostatečně proniknout a mohl jsem se posunout ke studiu serverové části aplikace.

Na základě studia serverových částí vznikl generátor konfiguračních souborů pro jednotlivé serverové komponenty. Jak se serverová část postupně vyvíjela, měnily se i konfigurační soubory. Tyto změny jsem postupně zapracovával, celek se však brzy stal velice nepřehledným a rozhodl jsem se proto celý generátor od základů předělat. Nově vytvořený generátor bylo třeba zásobovat daty s různými konfiguracemi, implementoval jsem si tedy základní uživatelské rozhraní, které mi umožňovalo tato data pohodlně zadávat. Jelikož jsou si použité dialogy mnohdy velmi podobné, využil jsem dědičnost a oddělil jsem grafické části od těch s výkonným kódem. Společná funkcionalita byla přesunuta do abstraktních tříd a konkretní implementace s novým chováním tak lze vytvářet velmi snadno.

Posledním krokem bylo napojit klientskou aplikaci na serverovou část. Pro tyto účely vznikla komunikační komponenta, která zajišťuje provádění jednotlivých komunikačních příkazů ve vlastních vláknech a není tak během komunikace blokováno uživatelské rozhraní. S dalšími požadavky, které přicházely v průběhu, jsem uživatelské rozhraní postupně rozšiřoval a vylepšoval až do současné podoby.

V tuto chvíli je možné pomocí klientské aplikace spouštět úlohy v různých konfiguracích na lokálním počítači, vzdáleném serveru a univerzitním clusteru Hydra. Takto specifické sestavení vhodné pouze pro jeden konkrétní cluster prozatím zůstává slabým místem celé aplikace. V dalších verzích bude nutné některé serverové části více abstrahovat a napsat konkrétní implementace, sloužící jako šablony pro dostupné výpočetní systémy. V uživatelském rozhraní by se pak měly objevit příslušné nabídky pro výběr či konfiguraci jednotlivých šablon. Dokud však tento problém není uspokojivě vyřešen v serverové části, nelze na to reagovat v rámci klientské aplikace. Vývoj aplikace ale v rámci projektu neustále pokračuje. Lze tedy předpokládat, že v některých dalších verzích aplikace bude tento klíčový nedostatek odstraněn.

Seznam použité literatury

[1] ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ. *Učební text předmětu: Obvody a architektura počítačů* [online]. [cit. 2015-11-10]. Dostupné z: http://kfe.fjfi.cvut.cz/~voltr/arch/paralel.pdf

[2] METACENTRUM VO. *Informační Wiki* [online]. [cit. 2015-12-5]. Dostupné z: https://wiki.metacentrum.cz/wiki/O_superpočítačích_-_tutoriál

[3] TECHNICKÁ UNIVERZITA V LIBERCI. *Dokumentace pro výpočetní cluster Hydra* [online]. [cit. 2016-01-08]. Dostupné z: http://www.nti.tul.cz/cz/Hydra

[4] METACENTRUM NGI. *Domovská stránka iniciativy* [online]. [cit. 2016-01-10]. Dostupné z: https://www.metacentrum.cz/

[5] TOP 500 SUPER COMPUTER LIST. *Aktuální pořadí počítače Solomon* [online]. [cit. 2016-01-23]. Dostupné z: http://top500.org/site/50561

[6] NÁRODNÍ SUPERPOČÍTAČOVÉ CENTRUM. *Web IT4Innovations* [online]. [cit. 2016-01-25]. Dostupné z: https://www.it4i.cz

[7] METACENTRUM VO. *Seznam dostupných front* [online]. [cit. 2016-02-05]. Dostupné z: http://metavo.metacentrum.cz/pbsmon2/queues/list

[8] SUNFIRE GRID ENGINE. *Manuálová stránka* [online]. [cit. 2016-02-12]. Dostupné z: http://gridscheduler.sourceforge.net/htmlman/manuals.html

[9] ADAPTIVE COMPUTING. *Stránka projektu TorquePBS* [online]. [cit. 2016-02-18]. Dostupné z: http://www.adaptivecomputing.com/products/open-source/torque/

[10] GITHUB. *Repozitář TorquePBS* [online]. [cit. 2016-03-05]. Dostupné z: https://github.com/adaptivecomputing/torque

[11] GITHUB. *Repozitář CESNET verze TorquePBS* [online]. [cit. 2016-03-08]. Dostupné z: https://github.com/CESNET/torque

[12] ADAPTIVE COMPUTING. TORQUE. *Příručka administrátora* [online]. 2015. [cit. 2015-03-12]. Dostupné z: http://docs.adaptivecomputing.com/torque/2-5-12/help.htm

[13] METACENTRUM VO. *Sestavovač příkazu qsub* [online]. [cit. 2016-03-05]. Dostupné z: http://metavo.metacentrum.cz/pbsmon2/person

[14] GITHUB. *Repozitář aplikace GeoMop* [online]. [cit. 2016-03-15]. Dostupné z: https://github.com/GeoMop/GeoMop

[15] PYTHON SOFTWARE FOUNDATION. *Python 3.x dokumentace* [online]. [cit. 2016-03-18]. Dostupné z: https://docs.python.org/3/

[16] QT COMPANY. *Domovská stránka Qt projektu* [online]. [cit. 2016-03-18]. Dostupné z: http://www.qt.io/

[17] RIVERBANK COMPUTING LIMITED. *Dokumentace knihovny PyQt5* [online]. [cit. 2016-03-18]. Dostupné z: http://pyqt.sourceforge.net/Docs/PyQt5/

[18] JAN TUROŇ. Od *kodéra k analytikovi – Řetězení metod* [online]. [cit. 2016-03-20]. Dostupné z: http://janturon.cz/kniha/retezeni-metod

[19] RIVERBANK COMPUTING LIMITED. *Ukázka implementace grafického rozhraní* [online]. [cit. 2016-03-21]. Dostupné z: http://pyqt.sourceforge.net/Docs/PyQt5/designer.html

Přílohy

A Obsah přiloženého CD

Na přiloženém CD jsou kompletní zdrojové kódy aplikace GeoMop aktuální k datu odevzdání práce (polovina května 2016), podklady použité v písemné části a kompletní text této práce v elektronické podobě.

Seznam souborů na CD:

- **dp_jan_gabriel_2016.odt** – text diplomové práce ve formátu odt (LibreOffice)
- **dp_jan_gabriel_2016.pdf** – text diplomové práce ve formátu pdf
- **dp_titulni_strana_2016.pdf** – titulní strana ve formátu pdf
- **podklady_dp_2016.zip** – obrázky a podklady použité v textu diplomové práce
- **klientska_aplikace_2016.zip** – kompletní zdrojové kódy klientské aplikace