Masarykova univerzita
Fakulta informatiky

Návrh a implementace rozhraní pro monitorování komponenty Engine systému Perun

Bakalářská práce

Jana Čecháčková

Brno, jaro 2014

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Jana Čecháčková

Vedoucí práce: Mgr. Slávek Licehammer

Poděkování

Shrnutí

Systém Perun spravuje uživatelské identity, virtuální organizace a jejich přístup ke službám. Skládá se z několika komponent, mezi které se řadí také Perun Engine, který má na starost propagaci změn v systému Perun na jednotlivé stroje. Cílem této práce bylo navrhnout monitorovací rozhraní, které umožní přístup k informacím o vnitřních procesech Perun Engine, které jsou v původním návrhu neviditelné a nepřístupné. V práci byly řešeny otázky přístupu k informacím z externích implementací, které Perun Engine využívá. Dále byly v řešení uváženy návrhy na vhodnou organizaci těchto procesů v Perun Engine, aby bylo možné tyto informace přehledně poskytnout. Hlavním přínosem této práce je přehlednost a větší dohled nad vnitřními procesy Perun Engine, což umožňuje např. snazší zachycení potencionálních chyb.

Klíčová slova

Perun, monitoring, Task Executor, správa událostí

Contents

1 Úvod

2 Představení systému Perun

Systém Perun vznikl jako projekt sdružení CESNET[1] a jedná se o systém spravující identity uživatelů a přístup ke službám.

Původní motivací pro vývoj systému Perun bylo vytvořit systém, který bude schopný řídit uživatele a služby aktivity MetaCentrum . Metacentrum[2] je česká národní gridová infrastruktura, která má výpočetní a úložné zdroje rozprostřeny na místech, kde jsou uživatelé, administrátoři a uživatelská podpora z různých organizací. Metacentrum poskytuje ve svých službách téměř 10 000 CPU jader a 3.5 PB místa k uložení dat. Tyto služby využívá okolo 700 uživatelů z akademické půdy. MetaCentrum potřebovalo ke své činnosti zjednodušení řízení uživatelů a služeb. Bylo potřeba zajistit automatické vytvoření uživatelských účtů na všech strojích a také jejích pozdější automatickou expiraci - tyto úkony nebylo možné z důvodu velkého počtu uživatelů i služeb provádět manuálně. Za tímto účelem vznikl systém Perun.

Perun podporuje správu uživatelů, delegování práv přístupu, řízení skupin a zápisu členů za účelem zjednodušit správu uživatelů. Perun je nyní vyvíjen už ve své třetí verzi a jeho funkcionalita již vzrostla nad rámec aktivity MetaCentrum. V nynější podobě je spravován sdružením CESNET a oproti předchozím verzím nabízí správu virtuálních organizací.

2.1 Virtuální organizace

Virtuální organizace je jednoduchá skupina skládající se z uživatelů, definovaného správce a souborem pravidel, které definují, kdo se může stát členem této virtuální organizace. Výhodou virtuální organizace je, že pokud chtějí její členové používat určitě služby, správce vyjedná přístup ke službám s poskytovatelem pouze jednou tzn. správce udělá práci za všechny členy jeho virtuální organizace (bez virtuální organizace by si každý uživatel musel vyjednat přístup ke službě sám).

Perun může spravovat neomezené množství virtuálních organizací, které jsou složeny z tisíců členů a služeb. Uživatelé jsou zapisováni do virtuálních organizací, kde mohou být dodatečně organizováni do skupin a podskupin. Každá skupina má definovaného svého správce, který spravuje členství ve skupině.

---

1. CESNET
2. MetaCentrum

2.2 Správa identit uživatelů

Zvyšující se počet služeb, které jsou využívány výzkumnými pracovníky, vyžaduje nějaký typ autentizace a autorizace. Důvodů může být několik např. služby mohou být zpoplatněné, mohou reprezentovat jedinečné a důležité zařízení nebo nemohou být využívány bez přístupu a poskytovatelé těchto služeb musí vědět, kdo k těmto službám přistupuje i v případě, že služba není zpoplatněna.

Obecně je k identifikaci uživatelů používán Identity Management System[3] a ten může být realizován v domovské instituci uživatele nebo poskytován třetí stranou (např. Sociální sítě). Tento typ identifikace ale nemusí být vždy dostatečný, protože poskytovatelé služeb musí v tomto případě vyslovit určitou důvěru k třetím stranám, že tyto identity dostatečně prověřily. To v některých případech z důvodu „peer to peer"[4] důvěry a otázky ochrany soukromí možné. Systém správy identit a přístupu - Perun, řeší tento problém řízení přístupových práv a identit.

Systém Perun v sobě zahrnuje celý cyklus uživatele, od jeho zápisu, přes správu přístupových práv až k expiraci uživatelského účtu. Perun nepracuje pouze s uživatelskými identitami, je schopný uchovat další informace o uživatelích, organizovat uživatele do skupin a virtuálních organizací. V neposlední řadě mohou být tyto skupiny přiřazeny ke službám tzn. že členové této skupiny mají nastaveno právo pro používání služby. To je hlavní výhoda ve srovnání s klasickými IdM systémy.

Ve většině institucí nebo výzkumných skupin již nějaká správa uživatelů existuje, stejně tak i správa služeb. Tato správa ale většinou nesplňuje všechny požadavky, které jsou kladeny. Není robustní, neposkytuje programovatelné prostředí a postrádá přívětivé prostředí pro uživatele. Perun je vytvořen i k nasazení do existujících infrastruktur, kde přináší robustní a škálovatelné řešení.

V porovnání s běžnými systémy spravující identity, Perun nabízí také správu služeb a přístupu. Perun je komplexní nástroj, který zjednodušuje správu výzkumných spolků nebo uživatelů a služeb napříč organizacemi. Systém Perun je používán na národní i mezinárodní úrovni, je dnes reálně využíván několika organizacemi, ke kterým se řadí MetaCentrum, C4E nebo Fedcloud. Budoucí potencionál systému Perun by mohl spočívat např. i v nasazení do sítě Eduroam. Perun je dobře uzpůsoben také pro organi-

3. IdM System
4. peer to peer

zace nebo výzkumné skupiny, které chtějí řídit přístup ke svým službám a nemají žádný systém správy identit nebo jich mají několik odlišných systémů a chtějí z nich získat a propojit uživatelské identity. Perun podporuje i složitější nasazení, jako je sdílení služeb mezi několika institucemi.

2.3 Správa služeb

Vytvořit řízení služeb, které bude efektivní, vyžaduje účast jak poskytovatelů služeb, tak i správců virtuálních organizací. Poskytovatelé služeb vyžadují jednoduchý způsob, jakým udělat své služby dostupné pro virtuální organizace. Navíc chtějí, aby byla provedena minimální nebo žádná změna na jejich službách a také požadují plnou kontrolu nad celým konfiguračním procesem. Na druhou stranu správci virtuálních organizací potřebují poskytnout uživatelům využití zdrojů. Proto byla vytvořena základní jednotka pro management zdrojů, která se nazývá facility.

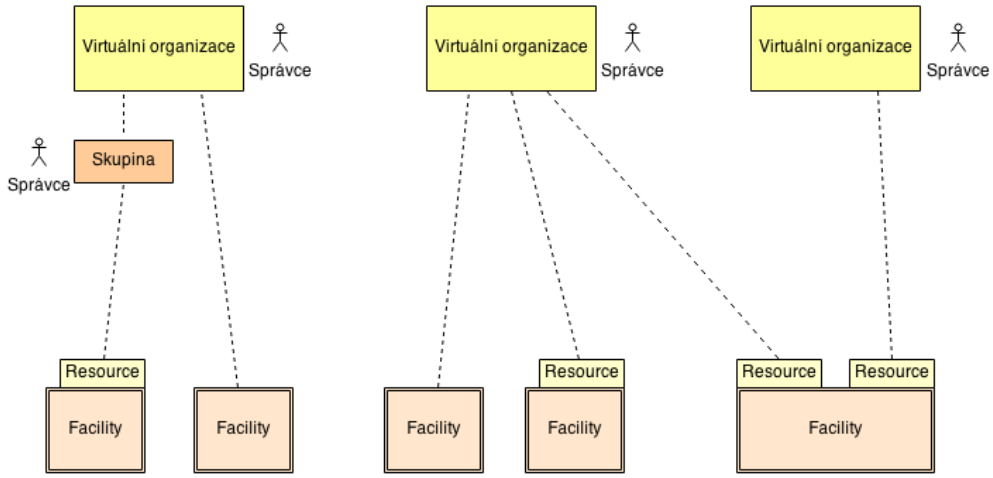


Figure 2.1: Schéma systému Perun

Facility Facility je homogenní entita, která poskytuje služby. Může představovat libovolnou službu, např. datové úložiště, tiskárnu, učebnu atd. Jediná podmínka pro facility je, že nastavení zůstane pro celou tuto jednotku neměnné. Poskytovatel služeb provádí konfiguraci této facility a po dokončení k ní poskytuje přístup virtuálním organizacím. Pokud spolu uzavřou poskytovatel služeb a správce virtuální organizace dohodu o používání služeb, měla by tato dohoda zahrnovat podmínky, pod kterými budou členové virtuální organizace tuto službu využívat. Poskytovatel služeb může pro virtuální organizace vytvořit tzv. Resource.

Resource Resource definuje technické podmínky a omezení používání facility virtuálními organizacemi. Správce virtuální organizace se poté může rozhodnout, kteří členové z virtuální organizace mohou používat resource nebo mohou tímto právem pověřit některého ze správců skupin. Základní komponenty systému Perun Perun je skládá z několika důležitých komponent, které mají přesně definovanou funkcionalitu. Mezi ně patří jádro, RCP, Registrar a Dispatcher s Engine. Jádro Perunu má na starost data a operace s uživateli, virtuálními organizacemi, službami a zdroji. Komponenta RPC je hlavní programovatelné prostředí systému Perun. RPC zprostředkovává komunikaci ostatním komponentám nebo i externím systémům se systémem Perun. Registrar je komponenta určená k zápisu uživatele a správě registračních formulářů. Dispatcher a Engine jsou zodpovědni za distribuování seznamů přístupů a konfigurací dále konečným službám.

2.4 Základní komponenty systému Perun

Perun je skládá z několika důležitých komponent[5], které mají přesně definovanou funkcionalitu. Mezi ně patří jádro, RCP, Registrar a Dispatcher s Engine. Jádro Perunu má na starost data a operace s uživateli, virtuálními organizacemi, službami a zdroji. Komponenta RPC je hlavní programovatelné prostředí systému Perun. RPC zprostředkovává komunikaci ostatním komponentám nebo i externím systémům se systémem Perun. Registrar je komponenta určená k zápisu uživatele a správě registračních for-

5. komponenty Peruna

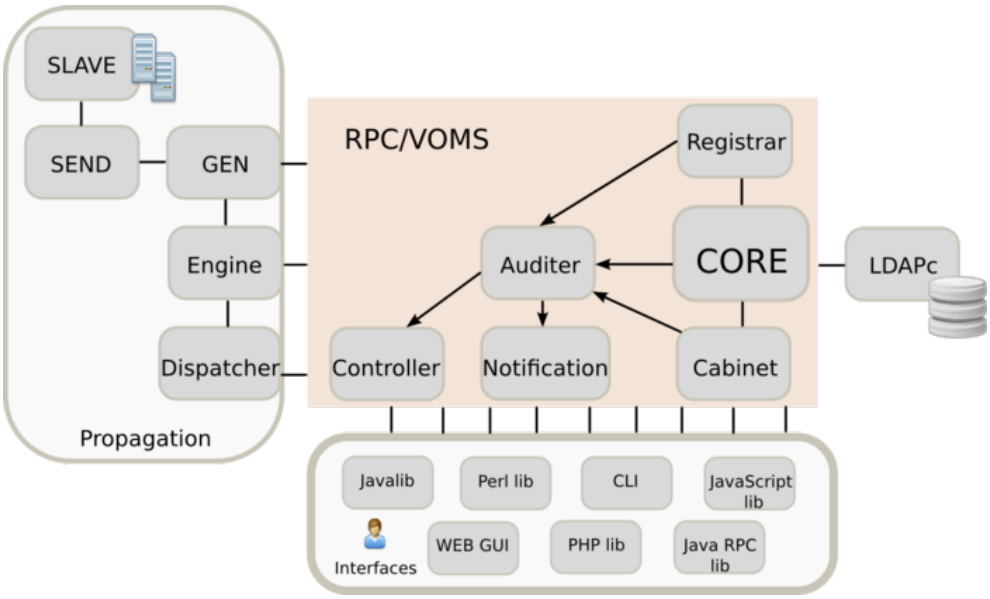


Figure 2.2: Struktura systému Perun

mulářů. Dispatcher a Engine jsou zodpovědni za distribuování seznamů přístupů a konfigurací dále konečným službám.

3 Představení Perun Engine

Perun Engine je součást systému Perun, která zpracovává přijaté události a propaguje nový stav do vybraných destinací. Pod událostmi si můžeme představit např. přidání nového uživatele do virtuální organizace, zahájení využívání služby nějakou virtuální organizací, atd. Perun Engine zodpovídá za to, aby byly všechny změny propagovány na samotné služby. Události do Perun Engine zasílá komponenta systému Perun s názvem Perun Dispatcher.

3.1 Perun Dispatcher

Tato komponenta je důležitou součástí systému Perun, která zpracovává databázová data a vytváří z nich události. Pokud je rozpoznána událost, která je spojena s nějakou ze služeb, je tato událost zaslána do Engine.

Událostí může být v jednom čase i velké množství a mohou způsobit velké vytížení Engine, což není žádoucí. Proto je další funkcí Perun Dispatcher vhodně události rozdělovat mezi několik instancí Engine. Rozdělení může být provedeno podle různých kriterií, např. geografické vzdálenosti strojů nebo podle toho, který Engine momentálně žádné události nezpracovává.

V současné době je pro zpracování událostí využíván Engine pouze jeden, v budoucnu se předpokládá rozšíření systému Perun a nasazení více instancí Engine.

3.2 Zpracování událostí

Perun Engine zodpovídá za zpracování událostí, které získá. Událost má při příchodu do Engine podobu textového řetězce a obsahuje strukturované informace o změnách, které se mají provést. Engine pomocí parsování této zprávy získává informace o změnách, které má za úkol propagovat.

Engine je při tomto zpracování schopen vyhledat v událostech duplicity. Uveďme si příklad vzniku duplicity: Do virtuální organizace, která využívá nějakou službu, se přidají dva noví uživatelé. Perun Dispatcher zprávy o těchto změnách zašle do Engine a ten zaregistruje, že se jedná o stejný typ změny, který má být propagován na stejnou službu. Dále tedy zpracovává tyto dvě události jako jednu úlohu.

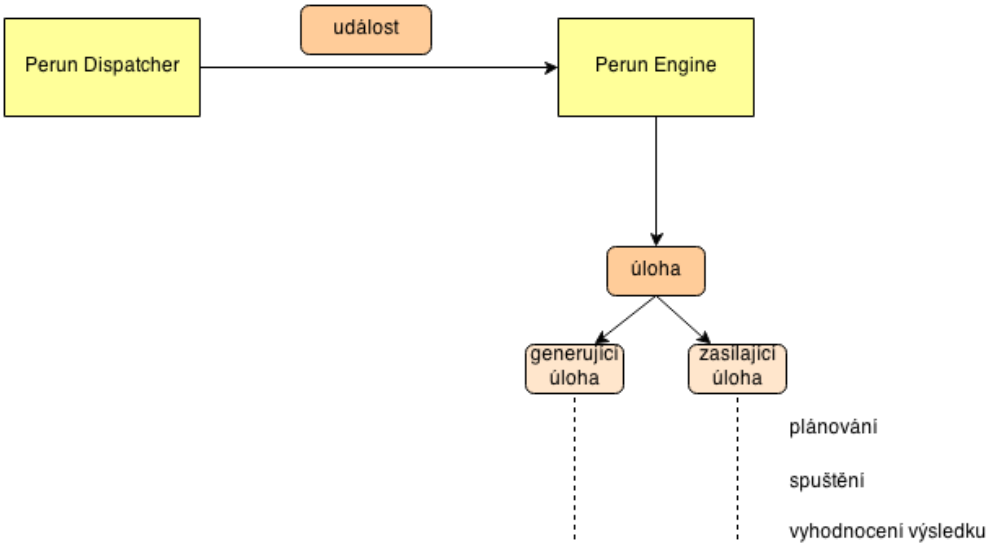


Figure 3.1: Proces zpracování události

Každá událost se po přijetí do Perun Engine rozdělí na dvě podúlohy, a to generující úlohu a zasílající úlohu. Generující úloha slouží k vygenerování skriptu, který bude spuštěn na cílovém stroji. Zasílající úloha pak slouží k přenesení tohoto skriptu k samotné službě. Je zřejmé, že zasílající úloha může plnit svou funkci jen za předpokladu, že generující úloha už daný skript vytvořila. Tuto závislost je nutné při plánování úloh ke spuštění vzít v úvahu.

3.3 Plánování a spouštění úloh

Všechny získané úlohy se v Engine nejprve naplánují ke spuštění. Při plánování úloh se zjišťuje, zda není daná propagace změny na této službě zakázána – v tom případě není možné propagaci změny provést. Také je nutné zmapovat strom závislostí – pokud Engine zjistí, že je úloha závislá na některé z jiných úloh, které dosud nebyly spuštěny, musí je naplánovat a spustit přednostně. Pokud má úloha všechny závislosti splněny, tzn. všechny úlohy, na kterých je závislá, už byly spuštěny a úspěšně dokončeny, pak ji Engine naplánuje ke spuštění.

Po těchto procesech plánování jsou úlohy připravené k samotnému odeslání a následné propagaci na jednotlivých strojích. Úlohy jsou spouštěny par-

alelně, tzn. Engine jich spouští několik najednou. Protože množství naplánovaných úloh může být opravdu velké, omezuje Engine spouštění úloh pouze do určitého limitu – zbývající úlohy čekají na spuštění ve frontě na uvolnění místa.

Jakmile jsou všechny úlohy, které bylo potřeba spustit, dokončeny, je provedena kontrola o úspěšnosti úloh – zda jejich spuštění proběhlo bez problémů či nikoliv. Pokud je nalezena úloha, při které vznikly v průběhu jejího spuštění problémy, Engine tuto úlohu naplánuje na opětovné spuštění.

Pokud Engine obslouží události, které mu Dispatcher zaslal, tzn. úlohy naplánuje, spustí a počká na jejich dokončení, komunikuje s další komponentou systému Perun a to Perun Controller. Engine informuje Controllera o dokončení jednotlivých úloh.

3.4 Perun Controller

Perun Controller komunikuje s komponentami Perun Dispatcher a Perun Engine skrz databázi a koordinuje jejich správu propagací. Jedná se o knihovnu, která také umožňuje přístup k těmto komponentám. Controller je navržen tak, aby poskytl prostředky pro plnění administrativních úloh a dohlížel na aktuální statistiky.

3.5 Task Executor

K samotnému spouštění úloh používá Engine externí rozhraní Task Executor, které se řadí pod framework Spring[1]. Task Executor neboli „spouštěč úloh“ má na starost celkovou organizaci aktuálně běžících úloh a také spravuje úlohy čekající na spuštění, které ukládá do interní fronty v pořadí, v jakém budou úlohy později spuštěny.

Vnitřní procesy Task Executoru jsou pro Engine neviditelné a nepřístupné, takže z nich není možné získat informace o aktuálním vytížení Task Executoru, jeho stavu ani úspěšnosti úloh. To výrazně snižuje přehlednost práce Engine.

Tato práce se zabývá tématem vylepšení rozhraní Task Executor a přináší možnost detailního monitoringu jeho vnitřních procesů. V následujících kapitolách si podrobně ukážeme funkcionalitu Task Executoru i průběh

1. Spring

jeho vnitřních procesů. Seznámíme se s údaji, které jsou pro nás v Task Executoru klíčové a poté si představíme návrhy na řešení monitoringu tohoto rozhraní, které byly v průběhu vývoje uváženy.

4 Task Executor

Obecně je Task Executor definován jako rozhraní, které poskytuje vyšší úroveň abstrakce třídy Runnable[1]. Implementace tohoto rozhraní může využít všechny typy odlišných strategií spouštění, např. synchronní, asynchronní, spouštění s využitím thread poolu a další.

Perun Engine k samotnému spouštění úloh využívá externího Task Executoru, který je poskytován frameworkem Spring[2]. Spring poskytuje abstrakci pro asynchronní spouštění, tzn. Task Executor umí spouštět více úloh najednou. Spring také kromě jiného poskytuje implementaci Task Executoru, která podporuje tzv. Thread pool a tato implementace nese stejnojmenný název Thread Pool Task Executor.

4.1 Thread pool

Většina Task Executorů, které jsou určeny pro asynchronní spouštění, využívají tzv. thread pool. Ten obsahuje úlohy, které jsou momentálně spuštěny. Úlohy, které přicházejí do Task Executoru, si thread pool shromažďuje a spouští až do určitého početního limitu. Pokud se thread pool naplní, ostatní úlohy typicky čekají v nějaké interní frontě Task Executoru na dokončení některé z běžích úloh a následného uvolnění místa v thread poolu. Uvolněné místo posléze zabere úloha, která čekala ve frontě nejdéle.

Můžeme tedy říci, že maximální velikost Thread poolu je shodná s maximálním možným počtem úloh, které mohou být najednou spuštěny.

4.2 Thread Pool Task Executor

Thread Pool Task Executor je třída frameworku Spring, která využívá thread pool a umožňuje konfiguraci asynchronního spouštění úloh. Tato implementace rozhraní Task Executor může být použita pouze v prostředí Java 5 a v tomto prostředí se jedná o nejvíce využívanou implementaci.

Thread Pool Task Executor má tři základní parametry, na základě kterých řídí spouštění úloh. Jedná se konkrétně o Core Pool Size, Queue Capacity a Maximum Pool Size.

---

1. class Runnable
2. Task Executor Spring

- *Core Pool Size*: číselná konstanta, která reprezentuje počet úloh, se kterými Task Executor nastartuje. Core Pool Size také definuje velikost thread poolu. Pokud je počet úloh vyšší než Core Pool Size, jsou přebývající úlohy drženy v interní frontě

- *Queue Capacity*: číselná konstanta, která reprezentuje maximální počet úloh, které je možné držet v interní frontě, dokud se neuvolní místo v thread poolu.

- *Maximum Pool Size*: číselná konstanta, která se stává klíčovou, pokud je již fronta zcela zaplněna a nelze tedy přidávat další čekající úlohy. V tomto případě se kapacita thread poolu může zvýšit z Core Pool Size na Maximum Pool Size a tím je Task Executor schopný obsloužit více úloh zároveň. Tento parametr reprezentuje maximální počet běžících úloh a upravuje velikost thread poolu.

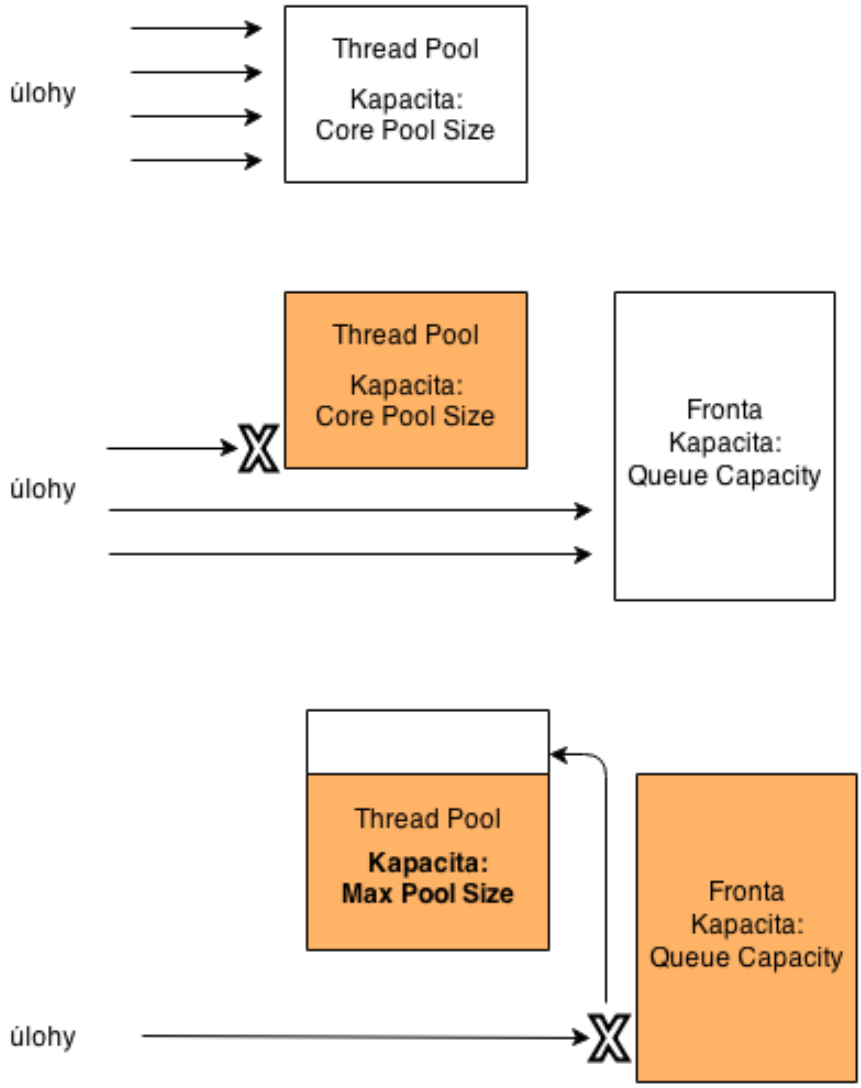


Figure 4.1: Zpracování Workerů Task Executorem

4.3 Srovnání s Thread Pool Executorem

Thread Pool Task Executor ve svých vnitřních procesech používá ke spouštění a organizaci běžících úloh třídu z balíku java.util.concurrent[3] a to třídu Thread Pool Executor[4]. Tato třída funguje se stejnými principy jako výše definovaný Thread Pool Task Executor. Jejich rozdíl spočívá pouze v tom, že Thread Pool Task Executor umožňuje programátorům přehledný přístup ke konfiguraci parametrů Core Pool Size, Maximum Pool Size a Queue Capacity. Systém Perun navíc masivně využívá ke svému fungování framework Spring, proto je z důvodu přehlednosti a srozumitelnosti vhodné při vývoji udržet programátorskou jednotnost.

4.4 Monitoring Task Executoru

Při návrhu na monitoring Task Executoru bylo v první řadě uváženo, o čem vlastně chceme získat informace. Byly zneseny požadavky na informace, které musí monitoring poskytovat, a to:

- Veškeré informace o úlohách, např. typ úlohy (generující nebo zasílající), ID služby, pro kterou byla úloha naplánovana, stav úlohy (čekající na spuštění, právě probíhající nebo dokončena), atd.
- Počet úloh, které jsou právě umístěny v thread poolu a informace o nich
- Počet úloh, které čekají na spuštění ve frontě a jejich pořadí a informace o nich
- Informace o dokončených úlohách, jejich úspěšnosti a také doby trvání běhu

V následující kapitole bude zmíněno několik návrhů na získání těchto informací. Pojd'me se na ně tedy podívat.

3. java.util.concurrent
4. Thread Pool Executor Java

5 Návrh řešení

U různých návrhů řešení se můžeme v diagramech i v textu setkat s pojmy, které je třeba pro ujasnění předem definovat a případně odlišit. Rozlišujme je striktně podle tohoto pojmenování:

- *Task Executor Engine* - Součást komponenty Perun Engine, která spravuje Task Executor.

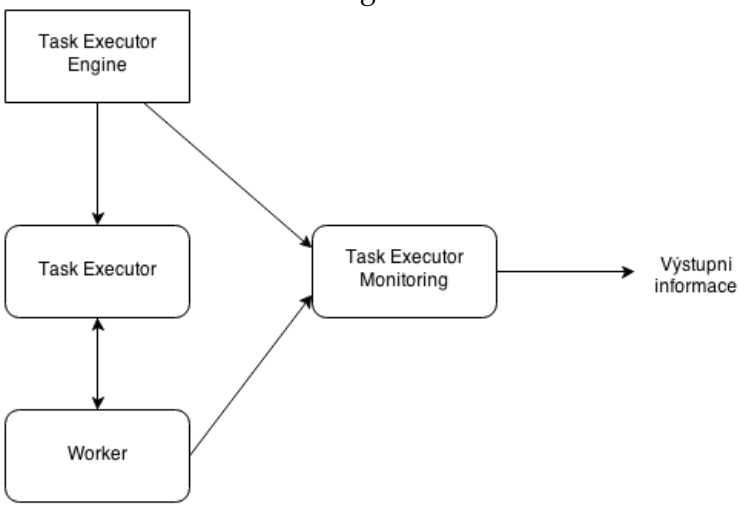
- *Task Executor* - Rozhraní pro třídu Thread Pool Task Executor, která má na starost samotné spouštění úloh.

- *Worker* - Typický objekt Task Executoru, který reprezentuje úlohu. Worker vzniká v Task Executor Engine a je vytvořen z úlohy neboli Tasku . Z jedné úlohy může vznikat více Workerů. Worker je v Perun Engine definovaný jako samostatná třída.

5.1 Verze 1

Jako první myšlenka se naskytl návrh na vývoj samostatné komponenty Task Executor Monitoring, která bude komunikovat s Task Executor Engine a samotnými Workery.

Figure 5.1: První návrh



Task Executor Engine vytvoří nové instance Workerů a zasílá je na spuštění do Task Executoru, kde probíhá samotný běh a celá organizace spouštění Workerů. Task Executor komunikuje se třídou Worker a dává jí signály ke spuštění skriptů. Celý běh se tedy odehrává ve třídě Worker, kde se po dokončení běhu zjišťují informace o tom, zda bylo spuštění úspěšné a v jakém čase skončilo. Tyto informace se ukládají do úlohy neboli Tasku, ze které byl tento Worker vytvořen.

Task Executor Monitoring

Mimo Task Executor byla v návrhu postavena nová samostatná komponenta Task Executor Monitoring, která dostávala oznámení z Task Executor Engine o zaslání Workeru do Task Executoru.

1. Přijímání workerů ke spuštění

   Při každém příchodu nového Workeru do Task Executor Monitoring byla tato instance uložena do jedné z kolekcí (právě probíhající, čekající ve frontě, dokončeno), o jejím rozdělení rozhodovalo prvotní ověření limitů. Jako limity pro jednotlivé kolekce byly zvoleny konstanty Core Pool Size a Queue Capacity. Pokud dosáhla kolekce, ve které byly uloženy běžící Workery, velikosti Core Pool Size, byl tento Worker uložen do kolekce, která představovala frontu, ve které Workers čekají na spuštění. Pokud dosáhla také kolekce reprezentující čekající workery svého maximálního limitu, byl worker odmítnut.

2. Správa workerů při změně stavu

   Při skončení běhu o sobě podal každý Worker oznámení o svém konci. Monitorovací komponenta tento Worker ukládala do kolekce dokončených úloh. To bylo vhodné především proto, že jsou po monitorovací komponentě požadovány informace i o dokončených Workerech a informacích o nich. Worker byl následně ihned smazán z kolekce běžících úloh.

   Po uvolnění místa v kolekci běžících úloh bylo toto místo ihned zaplněno dalším Workerem z kolekce čekajících, pokud zde nějaký byl. Task Executor Monitoring byl schopný v jakémkoliv čase podat veškeré požadované informace o Workerech, které právě běží, čekají ve frontě nebo jsou již dokončeny.

Nedostatky

Tento první návrh monitoringu se při testování ukázal jako velmi náchylný k chybám či k rozdílnostem s původními procesy v Task Executoru. Nemonitoruje totiž samotný Task Executor, ale pouze se snaží nasimulovat jeho chování. V tom mu pomáhají hlášení Task Executor Engine o zaslání Workeru ke spouštění a také hlášení samotných Workerů o dokončení svého běhu, ale nejedná se o monitoring samotného Task Executoru.

Druhým nedostatkem bylo neuvážení možnosti zaplnění celé fronty a proto tento návrh neobsahoval ošetření případu, kdy je třeba zvýšit kapacitu thread poolu, v tomto případě kapacitu kolekce, do níž ukládáme běžící Workery, na velikost Maximum Pool Size.

V další verzi byly uváženy způsoby, jak by bylo možné monitorovat samotný Task Executor s využitím jeho vlastních prostředků.

5.2 Verze 2

V druhém návrhu na řešení monitoringu bylo uváženo rozšíření Task Executoru, které by o sobě dokázalo podávat požadované informace o aktuálním i dlouhodobém stavu.

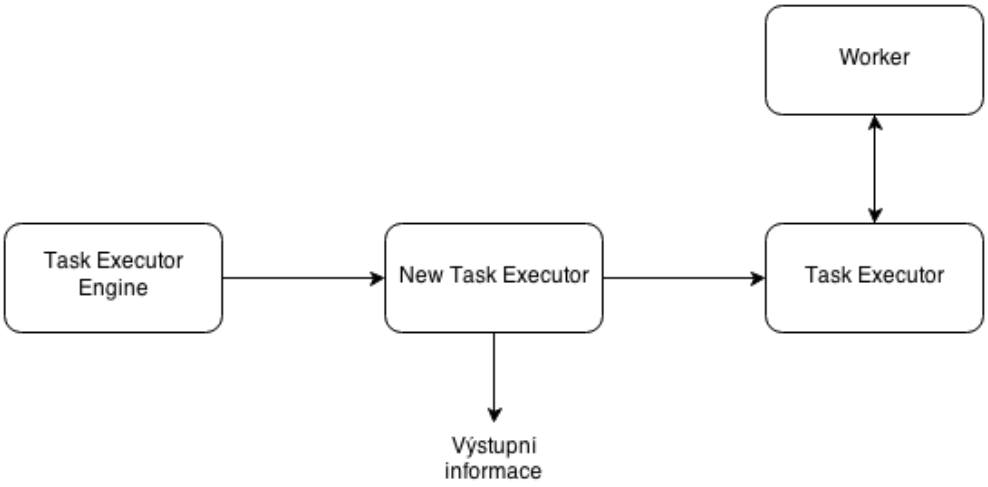


Figure 5.2: Druhý návrh

Nový návrh představoval vytvoření nové komponenty Task Executor Monitoring, která rozšiřuje třídu Task Executor (je tedy podtřídou[1] Task Executor). Hlavním cílem této verze bylo do New Task Executoru přidat vhodné metody, které by byly schopny v daný čas vrátit potřebné informace o vnitřních procesech.

Task Executor Monitoring fungoval jako prostředník mezi Task Executor Engine a Task Executor. Každý Worker, který zašle Task Executor Engine ke spuštění, je tedy nejprve zpracován komponentou Task Executor Monitoring, která potom Worker dále předává ke zpracování své nadtřídě[2] a to puvodnímu Task Executoru. Záměrem implementace monitorovací komponenty bylo získat snadný přístup k metodám a atributům Task Executoru a získat z nich požadované informace o Workerech, které bude Task Executor Monitoring schopný vhodně interpretovat.

1. podtřída
2. nadtřída

Získání informací z Task Executoru

Získání informací, které z Task Executoru potřebujeme, můžeme rozdělit na tři podúlohy, a to informace o běžících Workerech, dále čekajících Workerech ve frontě a dokončených Workerech.

1. Získání informací o Workerech čekajících ve frontě:

   Prvním krokem k získání informací o Workerech bylo využití interní fronty Task Executoru, kterou je možné z jeho původní implementace získat. Tato fronta v sobě obsahuje Workery, kteří čekají na uvolnění místa v thread poolu a následné spuštění. Tyto informace stačilo později jen přehledně zpracovat a ve vhodném formátu předat na výstup.

2. Získání informací o dokončených Workerech:

   K přístupu ke všem Workerům, kteří ukončili svůj běh, byla vytvořena jednoduchá kolekce, do které se přidal každý Worker, který oznámil komponentě Task Executor Monitoring svůj konec. Z této kolekce bylo opět možné jednoduše získat všechny požadované informace.

3. Získání informací o běžících Workerech:

   Problémem zůstala neviditelnost thread poolu, který obsahuje běžící Workery. V Task Executoru neexistuje žádná veřejná metoda, která by nám poskytla informace o thread poolu nebo Workerech, které obsahuje. Protože jsou běžící Workery uloženy v kolekci Task Executoru jako privátní atribut, vyvstala otázka, jakým způsobem se k dané kolekci dostat a získat z ní potřebné informace. Návrhem na řešení tohoto problému bylo použití tzv. reflexe.

Reflexe

Termínem reflexe označujeme schopnost získat za běhu informace o typu objektu s nímž program pracuje. Další schopností reflexe je možnost tyto data také modifikovat. Reflexi používáme v případě, že chceme získat vzdálenou třídu, atribut třídy či metodu. Zároveň ji můžeme použít jenom tehdy, pokud známe přesné jméno objektu (třída, atribut, metoda), který chceme získat.

V našem případě chceme získat kolekci běžících Workerů ze třídy Task Executor. K této třídě máme přístup, protože máme plně pod správou její podtřídu Task Executor Monitoring a jméno hledané kolekce je veřejně známo. Tímto jsme splnili předpoklad k úspěšnému provedení reflexe.

Problémy

Při realizaci tohoto návrhu bylo objeveno několik problémů a komplikací.

Prvním problémem bylo, že hledaná kolekce běžících Workerů je typu HashSet <Worker> a jejími objekty jsou instance třídy Worker. Třída Worker je zde privátní vnořenou třídou Task Executoru, proto by při realizaci reflexe bylo nutné nejprve pomocí reflexe získat odkaz na tuto třídu – jinak by nebylo možné získat informace o jednotlivých Workerech z kolekce. Získání kolekce by bylo možné až v druhém kroce. Bylo by tedy nutné použít jednu reflexi jako součást druhé reflexe, což bylo uváženo jako velmi složitá technika k získání běžících Workerů a informacích o nich. Začalo se tedy přemýšlet o jednoduším způsobu, jak se k požadovaným datům dostat.

Druhým problém byla samotná privátní vnořená třída Worker. Jedná se o naprosto odlišný objekt v porovnání s Workerem, se kterým pracuje Task Executor Engine a Task Executor Monitoring. V tomto případě obsahuje třída Worker metody s pokročilými technikami a z jejich atributů, které jsou také privátní, již nelze vyčíst informace, které jsou pro nás důležité. Tyto problémy byly hlavní příčinou od opuštění od tohoto návrhu na řešení.

Využití

Ačkoliv bylo od tohoto návrhu na řešení monitoringu nakonec upuštěno, byly v dalším návrhu využity navržené způsoby k získání čekajících Workerů ve frontě a dokončených Workerů.

5.3 Verze 3

Ve třetí verzi bylo využito poznatků z předchozích verzí, které byly propojeny a závěrem vzniklo jednoduché, přehledné a funkční řešení pro monitoring Task Executoru.

Využití první verze návrhu

Z první verze návrhu na monitoring Task Executoru byl převzat nápad na vytvoření nějaké pomocné kolekce, která bude určeným způsobem simulovat chování Task Executoru. Vylepšení této verze oproti předchozí je především v tom, že tato kolekce již nebude součástí nějaké externí třídy, ale bude umístěna přímo v Task Executoru Monitoring. Rozšíření třídy, kterou používáme, v tomto případě Task Executor Monitoring, je výrazně přehlednější i intuitivnější než původní implementace nové třídy, která získávala pouze signály z ostatních komponent a ty sama zpracovávala. Nově zavedená kolekce v Task Executoru Monitoring bude využívaná k ukládání Workerů, kteří právě probíhají.

Využití druhé verze návrhu

Z druhé verze bylo ihned použito několik nápadů, které se ukázaly jako jednoduché, přehledné a efektivní, ale především plně funkční.

První nápad, který byl využit, byl směrován na způsob ukládání informací o Workerech, kteří již dokončili svůj běh. Task Executor po dokončení běhu Workeru tento Worker zahazuje a dále si jej nepamatuje. Proto bylo nutné Task Executor rozšířit o schopnost pamatovat si všechny Workery, které jím prošly. Za tímto účelem byla v předchozí verzi vytvořena jednoduchá kolekce, do které se Workery po dokončení svého běhu uloží.

Druhé využití předchozí verze bylo ve způsobu získání interní fronty Task Executoru. Z této fronty potom snadno obdržíme všechny potřebné informace o Workerech, kteří čekají v této frontě na uvolnění místa v thread poolu.

Monitoring Task Executoru má za úkol podat informace o běžících, čekajících a dokončených úlohách. Jeho funkcionalitu nyní rozdělíme na tři podúkoly, a to na:

- monitoring běžících úloh
- monitoring čekajících úloh
- monitoring dokončených úloh

Pojďme si ukázat návrhy na řešení monitoringu pro jednotlivé podúlohy.

5.3.1 Monitoring běžících úloh

Získání informací o běžících Workerech se v druhé verzi návrhu ukázalo jako nejobtížnější část monitoringu. V této verzi řešení bylo k jejich získání využito veřejných metod Task Executoru a to metod beforeExecute() a afterExecute()[3].

Z dokumentace k těmto metodám, které jsou součástí Task Executoru, můžeme zjistit, že metody jsou volány bezprostředně před spuštěním Workeru a také bezprostředně jeho dokončení jeho běhu, jak už vyplývá také z názvů těchto metod.

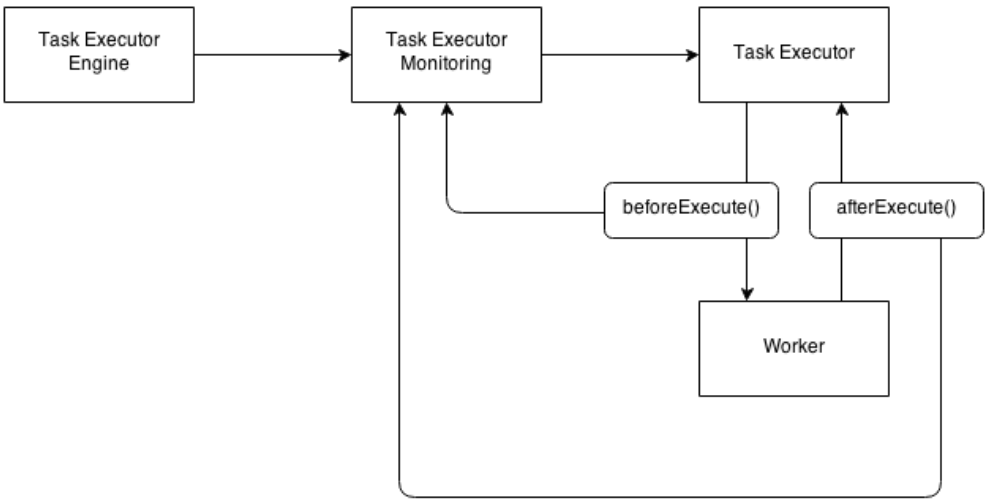


Figure 5.3: Třetí návrh

3. metody beforeExecute() a afterExecute()

Metody beforeExecute() a afterExecute() jsou v původní implementaci Task Executoru prázdné - byly vytvořeny především za účelem dalšího rozšíření Task Executoru. Využití těchto metod tedy bylo vcelku jednoduché. V podstatě stačilo zaznamenat průchod Workera těmito metodami a následně ho podle toho přiřadit do odpovídající kolekce, ze které o něm budou vypisovány informace. Po zavolání funkce beforeExecute() se Worker ihned přidá do nové kolekce pro běžící úlohy a po zavolání funkce afterExecute() tuto kolekci opouští.

5.3.2 Monitoring čekajících úloh

Nápad na monitorování úloh, které čekají ve frontě na uvolnění místa, byl převzat z druhé verze návrhu. Task Executor umožňuje získání interní fronty pomocí metody getQueue()[4] a z té jsme schopni v případě potřeby získat veškeré požadované informace. Pokud se navíc informace z interní fronty Task Executoru shodují s informacemi, které očekáváme a nevyskytují se žádné duplicity úloh atd. v jiných kolekcích, které jsme v Task Executoru dodatečně vytvořili, máme dobrou kontrolu správnosti naší práce. Interní fronta nám je schopna poskytnout veškeré informace, které potřebujeme, proto zde není třeba žádných dalších doplňků.

5.3.3 Monitoring dokončených úloh

Úlohy, které dokončí v Task Executoru svůj běh, je nutné nějakým způsobem uchovat. Task Executor dokončené Workery zahazuje a dále si je již nepamatuje. Proto byla na základě nápadu z předchozí verze vytvořena jednoduchá kolekce, do které se Workery po svém dokončení ukládají. Rozdíl oproti ostatním kolekcím je v tom, že Workeři v této třídě mají informaci také o délce svého běhu a také své úspěšnosti.

K přidání dokončeného Workeru bylo opět, stejně jako u běžících úloh, využito metody afterExecute(). Jak již bylo řečeno, metoda afterExecute() je volána bezprostředně po dokončení běhu Workeru. Proto je tato metoda doplněna o novou funkcionalitu, kdy se každý Worker, který touto metodou projde, uloží do kolekce dokončených Workerů.

4. getQueue

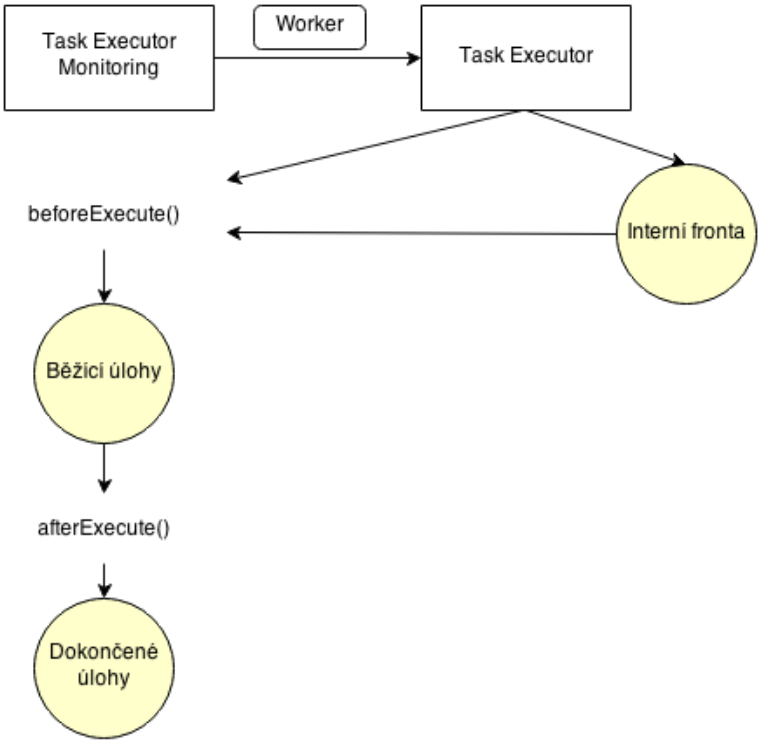


Figure 5.4: Průchod Workeru jednotlivými kolekcemi

Třetí návrh se ukázal jako vhodné řešení monitoringu. Je jednoduše koncipovaný, efektivní a měl by zvládat i velkou zátěž, která může být způsobena velkým počtem úloh ke zpracování. Proto nyní přejdeme k samotné implementaci tohoto návrhu, kde si ukážeme zajímavosti, ale také problémy, které bylo třeba řešit.

6 Implementace

V této kapitole přejdeme k samotné implementaci návrhu na monitoring Perun Engine a ukážeme si problémy, se kterými se bylo třeba v průběhu implementace vypořádat.

6.1 Parametry Workeru

Každý Worker obsahuje informace, které chceme monitorovat. Mezi tyto informace patří:

- identifikační číslo a typ úlohy (generující nebo zasílající)
- facility, na kterou má být změna propagována (identifikační číslo, název a typ)
- status úlohy (zpracovávající se, dokončena)
- pokud je úloha již dokončena, potom obsahuje informace také o času spuštění a času dokončení běhu

Při implementaci monitoringu jsem narazila na problém, kterým bylo získání duplicitních a nejasných informací o Workerech. Jak už bylo řečeno v návrhové části práce, z jedné úlohy může vzniknout více Workerů. Problémem je, že z údajů, které nám Worker poskytuje, není občas možné Workery odlišit, protože vznikli ze stejné úlohy a jsou naplánovani na stejnou facility.

Řešením tohoto problému bylo zavedení nového parametru, který bude reprezentovat identifikační číslo Workeru. Každému Workeru je v Task Executoru Engine přiděleno jedinečné číslo, na základě kterého jej můžeme odlišit od ostatních Workerů. Číslo není generováno náhodně, prvnímu Workeru, který prochází Task Executorem Engine je přidělena jednička a s příbývajícím počtem Workerů toto číslo postupně roste vždy o jedna.

6.2 Nový status úlohy

Status úlohy se průchodem Perun Engine několikrát mění. Mohou nastat čtyři různé stavy, ve kterých se úloha nachází, a to:

- planned - úloha se stává plánovanou po příchodu do Perun Engine

- processing - úloha se začíná zpracovávat po vstupu do Task Executoru Engine
- done - úloha prošla Task Executorem a je dokončena
- error - úloha prošla Task Executorem a v průběhu jejího spuštění se vyskytla chyba, proto nebyla řádně dokončena

Při implementaci monitorovací komponenty jsem narazila na jistou nepřehlednost v informacích o Workerech. Parametry Workerů se od sebe dostatečně nelišili a nebylo možné rozpoznat, zda daný Worker čeká v interní frontě nebo zda právě probíhá.

Proto byl zavedený nový status úlohy, a to status running neboli běžící. Úloha, která má status running, o sobě vypovídá, že je právě v daném okamžiku spuštěna a zatím její běh nebyl dokončen. Změna statusu je realizována v třídě Worker a to ihned při zahájení spuštění Workeru zavoláním metody run().

6.3 Změna statusu úlohy

Jedním z parametrů úlohy je její status, který se při průchodu Perun Engine několikrát změní, jak bylo zmíněno výše. Pokud chce monitorovací komponenta v určitém čase získat všechny informace o všech Workerech, kteří byli a jsou spuštěny, získávání těchto informací zabere také nějaký čas. Je nutné si uvědomit, že délka běhu Workeru se pohybuje v řádu milisekund a Workery je možné pouštět paralelně.

Při implementaci monitoringu bylo zjištěno, že některé úlohy mohou v průběhu výpisu informací změnit svůj stav a tím se přesunout do jiné kolekce. To nemůžeme dopustit, protože potom dochází k velmi snadnému rozbití informací a získané údaje s největší pravděpodobností nebudu pravdivé. Operace s Workery proto bylo nutné v průběhu výpisu uzamknout, aby se tomuto problému předešlo.

Zámek

Pro uzamknutí Task Executoru v průběhu získávání informací o něm byla využita třída zámku

http://cs.wikipedia.org/wiki/Java_Messaging_Services
http://docs.spring.io/spring/docs/2.5.x/api/org/springframework/core/task/TaskExecuto
http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html
http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/scheduling.html
https://wiki.metacentrum.cz/wiki/Perun_component_schema#Engine