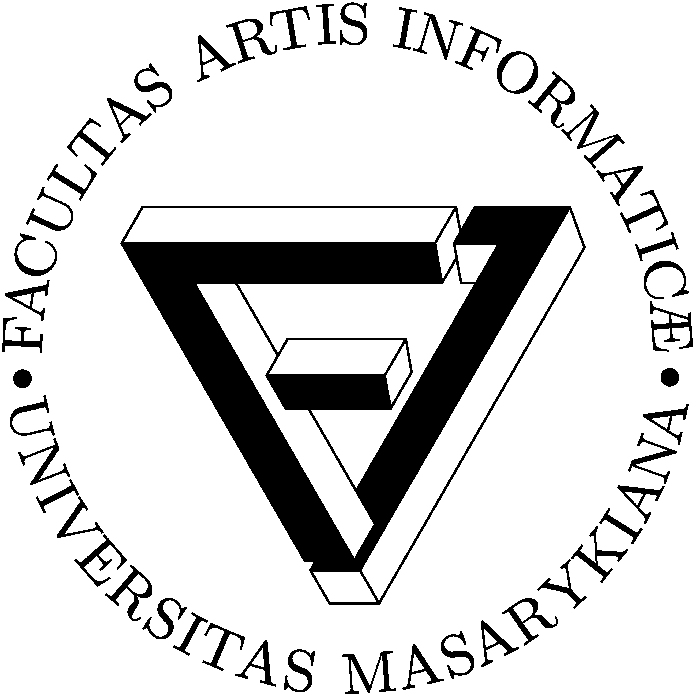
MASARYKOVA UNIVERZITA

FAKULTA INFORMATIKY

****

**Návrh a implementace rozhraní**

**pro monitorování komponenty Engine systému Perun**

BAKALÁŘSKÁ PRÁCE

**Jana Čecháčková**

Brno, jaro 2014

**Zadání:**

Engine je komponenta systému Perun, která slouží pro generování konfiguračních souborů a jejich následné rozesílání koncovým službám. Engine funguje masivně paralelně, špičkovou zátěž rozkládá v čase pomocí interních front a dalších vnitřních stavů.

Úkolem práce bude seznámit se s Engine a následně navrhnout a implementovat rozhraní pro monitorování aktuálního vytížení a vnitřních procesů Engine. Rozhraní musí poskytovat nejen statistické údaje, ale musí umožňovat přístup i k detailům konkrétních úloh, které jsou právě zpracovávány nebo čekají ve frontách. Dále toto rozhraní musí umožňovat sběr statistických dat o průběhu zpracovávaných úloh, například délky běhů a počty dokončených úloh v čase.

Osnova

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Úvod

Představení systému Perun

* Virtuální organizace
* Správa identit uživatelů
* Správa služeb
  + - Facility
    - Resource
* Hlavní složky systému Perun

Představení Perun Engine

Model Perun Engine

Task Executor

Návrh řešení

Implementace

**Úvod**

**Představení systému Perun**

Systém Perun vznikl jako projekt sdružení CESNET[[1]](#footnote-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[[2]](#footnote-2). Metacentrum 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.

**Virtuální organizace**

Virtuální organizace je jednoduchá skupina skládájící 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.

**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]](#footnote-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]](#footnote-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.

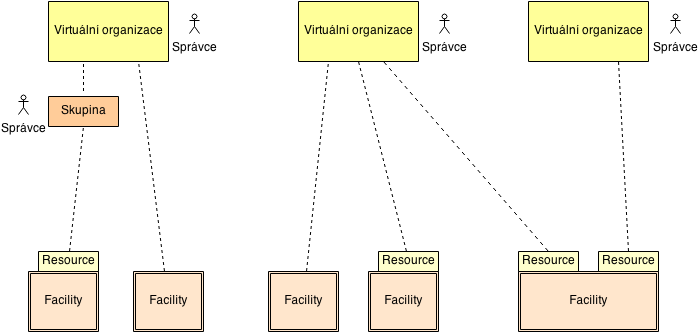
Perun je vytvořen i k nasazení do existujících infrastruktur, kde přináší robustní a škálovatelné řešení.

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 organizace 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.

**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.



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.

**­**

**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.

**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.

**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.

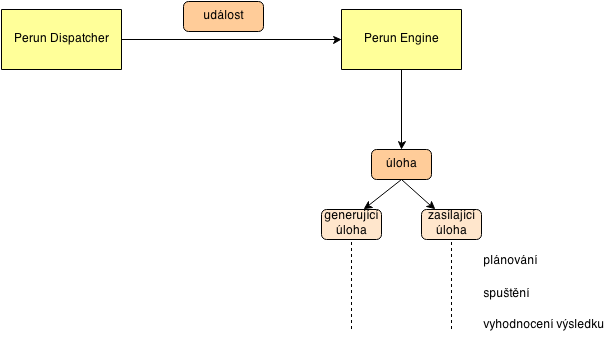
Každá událost se po přijetí do Perun Engine rozdělí na dvě podúlohy, a to generující úlohu a zasílající úlohu. Generující úloha slouží k vygenerování skriptu, který bude spuštěn na cílovém stroji. Zasílající úloha pak slouží k přenesení tohoto skriptu k samotné službě. Je zřejmé, že zasílající úloha 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.

**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 paralelně, 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.

**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.

**Task Executor**

K samotnému spouštění úloh používá Engine externí rozhraní Task Executor, které se řadí pod framework Spring[[5]](#footnote-5). 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 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 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.

**Task Executor**

Obecně je Task Executor definován jako rozhraní, které poskytuje vyšší úroveň abstrakce třídy Runnable[[6]](#footnote-6). 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[[7]](#footnote-7). 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.

**Thread pool**

Většina Task Executorů, které jsou určeny pro asynchronní spouštění, využívají Thread pool. Ten obsahuje úlohy, které jsou momentálně spuštěny. Úlohy, které přicházejí do Task Executoru, si thread pool shromažďuje a spouští až do určitého limitu. Tento limit je daný jeho maximální velikostí. 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. Můžeme tedy říci, že maximální velikost Thread poolu je shodná s maximálním možným počtem úloh, které mohou být najednou spuštěny.

**Thread Pool Task Executor**

Thread Pool Task Executor je třída frameworku Spring, která umožňuje konfiguraci asynchronního spouštění úloh, které využívá thread pool. 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 umožňuje viditelnost konfigurace parametrů pro Thread Pool Executor[[8]](#footnote-8)

ne zodpovídá za zpracování Events ODKAZ, správu Tasků ODKAZ, jejich plánování, odesílání a spuštění na daných strojích. Posílání konfiguračních souborů z Perun Engine napříč sítí k jednotlivým strojům je časově nejdražší aktivita. Proto existuje z důvodu efektivnosti několik instancí Engine a každá z nich obsluhuje pouze omezené množství konkrétních strojů.

Pro přehlednost a lepší pochopení Engine je nutné uvést několik pojmů, které jsou pro tuto komponentu klíčové.

Event

Event neboli událost v sobě obsahuje strukturované informace o změnách, které se mají provést, v podobě textového řetězce. Eventy vznikají v Perun Dispatcher ODKAZ pomocí parsování z databázových dat a po zaslání do Perun Engine se tyto strukturované informace využívají k vytvoření tzv. Tasků ODKAZ, které má na starost EventParser ODKAZ. Jeden event v sobě obsahuje informace o jedné změně na jednom konkrétním stroji.

Task

Task je typický objekt Perun Engine, který reprezentuje službu (ExecService[[9]](#footnote-9)), která je naplánováná k propagaci na určitém stroji (Facility[[10]](#footnote-10)). Tyto tasky se v Perun Engine plánují, spouštějí, podávají o sobě informaci o výsledku a mohou se také přeplánovat. Task vzniká v Perun Engine a je výsledkem rozparsování textové zprávy Eventu v Event Parseru[[11]](#footnote-11).

Worker

Workery vznikají z tasků a jsou primárně objekty Task Executoru ODKAZ. Každý worker reprezentuje jednu akci, která spouští odpovídající skript.

Funkcionalita Perun Engine zahrnuje převody mezi těmito typy objektu a to ve svých třídách k tomu určených. Worker je poslední a strukturálně nejnižší jednotkou Engine a ostatní instance jsou určeny především pro přístup k samotnému workeru, či jeho obalení a lepší srozumitelnost. Pro přístup k informacím, aktuálním stavu i dokončení úkolů je používán primárně Task.

Hlavní složky Engine, které zodpovídají za postupné systematické zpracování všech tasků jsou spouštěny v pravidelných intervalech a samotné tasky jsou zpracovávány paralelně. To výrazně ovlivňuje efektivitu a časovou náročnost Engine, pokud vezmeme v úvahu, že by tato komponenta měla mít v jedné své instanci potenciál zpracovat i tisíce tasků.

**Model Perun Engine**

Diagram Perun Engine se skládá z několika důležitých částí. Pojďme se podívat, k čemu jednotlivé komponenty slouží a jaký je jejich význam v Perun Engine.

**Message Reciever**

Message Reciever zpracovává JMS messages přicházející z Perun Dispatcher, které dále rozděluje podle typu na:

* Command messages, které reprezentují příkaz a jsou dále předány Command Processoru
* Event messages, které obsahují alespoň jeden Event, a dále jej předávájí Event Processoru

**Event Processor**

V Event Processoru jsou po určitou krátkou dobu drženy všechny příchozí Eventy. Důvodem je především efektivita Engine. Uveďme si příklad: pokud dva uživatelé, kteří využívají stejnou Facility ODKAZ, změní své údaje, byl by pro každou takovouto změnu vytvořen nový Task. V Event Processoru je tzv. Pooling, který si tyto Eventy před vytvořením Tasků drží a pokud tento případ rozpozná, z těchto dvou Eventů vznikne pouze jeden Task.

**Event to ExecService Resolver**

Každý Event, který obdrží Event Proccesor je zde porovnáván s Processing Rules. Jako výsledek je do Event Processoru navrácena dvojice ExecService a Facility. Jak již bylo zmíněno u Event Processoru, pokud dva uživatelé zašlou požadavek na stejný stroj, tato komponenta pro obě tyto akce vrátí pouze jednu dvojici. ExecService a Facility.

*ExecService*

ExecService reprezentuje entitu, kterou je třeba naplánovat k propagaci, to znamená vytvořit z ní Task ODKAZ.

Existují dva typy ExecService:

* Generating ExecService, která reprezentuje konfigurační soubory generující skripty. Zajišťuje generování aktuálních konfiguračních souborů pro celou Facility ODKAZ, které jsou později na dané místo zaslány. Tento typ může existovat jako samostatný objekt, či v páru se Sending ExecService.
* SendingExecService, která reprezentuje konfigurační soubory zasílající skripty a má na starost zasílání konfiguračních souborů napříč sítí. Tento typ může existovat pouze v páru s Generating ExecService.

*Facility*

Facility reprezentuje aktuální zařízení, na kterou chceme propagovat změnu či aktualizaci. Typicky se jedná o úložiště dat.

**Task Scheduler**

Task Scheduler je zodpovědný za proces plánování Tasků. Daná ExecService zde bude procházet komponentami Denials Resolver a Dependencies Resolver.

**Denials Resolver a Dependencies Resolver**

Denials Resolver zjišťuje, zda pro danou ExecService existují nějaké stroje nebo hosté, kde je tato ExecService zakázána. Oproti tomu komponenta Dependencies Resolver slouží ke zmapování tzv. Stromu závislostí dané ExecService. Uveďme si příklad: pokud je ExecService A závislá na ExecService B, je nutné nejprve spustit ExecService B, přičemž ExecService A čeká na dokončení a až poté, co jsou její závislosti splněny, může být také spuštěna.

**Message Transmitter**

Message Transmitter je obdobnou součástí Engine jako Message Reciever ODKAZ a slouží jako výstup Engine, ve kterém o sobě podává data. Výstupní komunikace probíhá opět formou JMS zpráv ODKAZ.

**Propagation Maintainer**

Pokud nějaký z Tasků při běhu Engine neskončí úspěšně, Propagation Maintainer má za úkol tyto Tasky přeplánovat pro nové spuštění.

**Command Processor**

Command Processor slouží k interpretaci příkazových zpráv, které mohou být za účelem získání statistik nebo násilného spuštění ExecService.

**Task Executor**

Task Executor je určen k samotnému spuštění naplánovaných Tasků. Podrobný popis této komponenty i její funkcionality je popsán v následující kapitole.

**Návrh řešení**

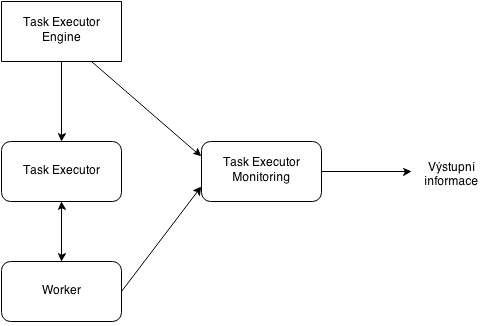
U různých návrhů se můžeme v diagramech i v textu setkat s označením Task Executor Engine a Task Executor.

Rozlišujme je striktně podle tohoto pojmenování:

* Task Executor Engine jako komponentu systému Perun Engine
* Task Executor jako ThreadPoolTaskExecutor ODKAZ, který ma na starost samotné spouštění Workerů ODKAZ

**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.



Task Executor Engine vytvoří nové instance Workerů ODKAZ a zasílá je na spuštění do Task Executor, kde probíhá samotný běh a celá organizace spouštění Workerů.

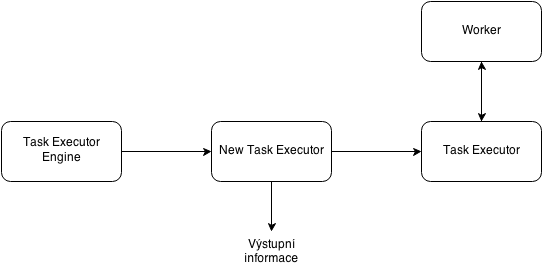
Task Executor Monitoring

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. Poté o sobě podal každý Worker oznámení o začátku a konci běhu včetně jeho úspěšnosti. Monitorovací komponenta tyto údaje ukládala a Workery vhodně rozdělovala do kolekcí podle jejich průběhu zpracování (právě probíhající, čekající ve frontě, dokončeno). Z těchto kolekcí byla pak schopna poskytnout veškeré dostupné informace o Workerech i jejich aktuálním zpracování.

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 Workerů i Task Executor Engine, ale nejedná se o monitoring samotného Task Executoru.

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

**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.

Nový návrh představoval vytvoření nové komponenty New Task Executor, která rozšiřuje třídu Task Executor (je tedy podtřídou[[12]](#footnote-12) 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.

Prvním krokem bylo využití interní fronty Task Executoru, kterou je možné v původním návrhu získat. Tato fronta v sobě obsahuje Workery ODKAZ, kteří čekají na uvolnění místa v Thread Poolu ODKAZ a následné spuštění. Problémem zůstala neviditelnost Thread Poolu ODKAZ, který obsahuje běžící Workery.

Protože jsou běžící Workery uloženy v kolekci Task Executoru jako privátní atribut, přišla otázka, jakým způsobem se k dané kolekci dostat a získat z ní potřebné informace. Prvním návrhem na řešení tohoto problému byla tzv. Reflexe.

**Reflexe**

Termínem reflexe označujeme schopnost získat za běhu informace o typu objektu s nímž program pracuje. <http://cs.wikipedia.org/wiki/Java_Messaging_Services>

<http://docs.spring.io/spring/docs/2.5.x/api/org/springframework/core/task/TaskExecutor.html>

<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>

Změna statusu tasku

Za dobu běhu instance Perun Engine každý task několikrát změní svůj status. Za tuto aktivitu jsou odpovědní především:

* Task Scheduler: po spuštění nastavuje stav tasků na „plánovaný“
* Task Executor: po spuštění nastavuje stav „plánovaných“ tasků na „probíhající“ a po svém dokončení opět nastavuje stav těchto tasků na „ukončené“ nebo „chybné“

Status tasku primárně rozhoduje, jak bude task dále zpracováván.

Scheduling Pool

Scheduling Pool je nezbytnou součástí Perun Engine, ve které jsou uloženy všechny tasky a také informace o nich. Ostatní komponenty Perun Engine přistupují k taskům právě přes Scheduling Pool. Do funkcionality této komponenty patří přidávat tasky, měnit jejich status, získávat seznamy tasků rozdělených podle statusu a také tasky odstraňovat.

Můžeme tedy říci, že Scheduling Pool nám je schopný poskytnout aktuální informace o stavu všech tasků, které ještě nebyly zpracovány (po zpracování jsou tasky z Scheduling Poolu vymazány).

1. [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. [↑](#footnote-ref-5)
6. [↑](#footnote-ref-6)
7. [↑](#footnote-ref-7)
8. [↑](#footnote-ref-8)
9. [↑](#footnote-ref-9)
10. [↑](#footnote-ref-10)
11. [↑](#footnote-ref-11)
12. [↑](#footnote-ref-12)