VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÁ KOORDINACE A ŘÍZENÍ PROCESŮ NA PLATFORMĚ JAVA

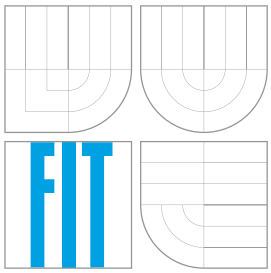
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN JANYŠ

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÁ KOORDINACE A ŘÍZENÍ PROCESŮ NA PLATFORMĚ JAVA

AUTOMATED ARRANGEMENT AND COORDINATION OF PROCESSES ON THE JAVA PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MARTIN JANYŠ
AUTHOR

VEDOUCÍ PRÁCE — Ing. KOČÍ RADEK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Předmětem diplomové práce je téma odolnosti a stability webových aplikací se zaměřením na platformu Java. Řada existujících informačních systémů postavených nejen nad touto platformou se potýká s problémy, které narušují stabilitu aplikace. Tyto problémy pak mohou vyústit ve výpadek, odstávku a následně i finanční nebo obchodní ztrátu v důsledku nefunkčnosti celé služby. Cílem bude ukázat problémy, se kterými se aplikace potýkají v provozním prostředí, a jak je proaktivně řešit. Jako možná dílčí řešení zvýšení stability mohou být vhodná konfigurace JVM (Java Virtual Machine), analýza a oprava odhalených chyb anebo technika na zvýšení stability nazývaná Sandboxing, které se věnuje tato práce. Pomocí této techniky je možné rozdělit aplikace do samostatných částí, které se nemohou ovlivnit. Zamezí se tak šíření chyb mezi částmi aplikace a tím zvýšíme stabilitu celé aplikace. Mezi cílové aplikace patří Java aplikace realizované za pomoci aplikačního rámce Spring. Do takto postavených aplikací lze zavést techniku Sanboxing vhodnou konfigurací, která zajistí, že běh aplikace bude rozdělen do určených částí, které budou automaticky testovány a případně restartovány. Aplikace se tak sama zotaví v postižených částech bez kompletního výpadku. Projekt nese jméno Java Capsules.

Abstract

The subject of this thesis is the topic of the resilience and stability of web applications with a focus on the Java platform. Many existing information systems based not only upon this platform face problems that disturb the stability of applications. These problems may result in the failure, downtime and, consequently, financial or business loss due to the malfunction of the whole service. The aim is to show the problems that the applications face in a production environment and to show how to address them proactively. A possible partial solution to increase the stability may be an appropriate configuration of JVM (Java Virtual Machine), an analysis and corrections of detected errors, or a technique called Sandboxing to increase the stability, which this thesis deals with. Using this technique, it is possible to divide the application into separate parts that cannot influence each other. This prevents the propagation of errors among the parts of the application and thereby increases the stability of the entire application. The target applications include the Java applications made with the help of Spring framework. The Sanboxing technique can be implemented into the applications built this way by means of suitable configuration, which ensures that the application run will be divided into specified parts that will be automatically tested and possibly restarted. The application then recovers itself in the affected areas without a complete failure. The project is called Java Capsules.

Klíčová slova

Java, Sandbox, Oddělování procesů, JVM, AOP, Webová aplikace, Spring, Stabilita aplikací

Keywords

Java, Sandbox, JVM, AOP, Web application, Spring, Application resilience

Citace

Martin Janyš: Automatická koordinace a řízení procesů
na platformě Java, diplomová práce, Brno, FIT VUT v Brně, 2015

Automatická koordinace a řízení procesů na platformě Java

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Kočího Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Janyš
19. května 2015

Poděkování

Děkuji vedoucímu diplomové práce Ing. Radku Kočímu Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce. Dále děkuji za odbornou a technickou pomoc při řešení problémů Mgr. Miloši Zikmundovi.

Obsah

Kapitola 1

Úvod

Diplomová práce se zabývá problémy stability, které se objevují ve webových aplikacích, jakými mohou být informační systémy a podnikové portály. Tyto systémy jsou často v podobě monolitické aplikace, běžící na jednom aplikačním serveru a v jednom procesu, i když se jedná o několik samostatně nasazených aplikací. Nasazeným aplikacím není možno přidělovat systémové prostředky jednotlivě. Funkcionalita, která je z důvodu chyby schopna vyčerpat všechny přidělené systémové zdroje, může způsobit výpadek celé aplikace. Pokud tento problém oddělíme od zbytku aplikace, chyba nepostihne celý systém a zotavení při výpadku proběhne pouze v rámci oddělené části.

Navrhovaným řešením těchto problémů je koncept oddělování procesů, který je též znám pod anglickým názvem *Sandboxing*[1]. Tato technika je schopna přispět k prevenci zmiňovaných problémů. Implementace techniky *Sandboxing* bude v této práci navržena na podnikové aplikace (webové a portálové informační systémy) a jejich komponenty běžící na platformě Java. U takových aplikací jsou používány existující knihovny, aplikační rámce a řešení. Proto je potřeba pro dobrou integraci implementované techniky *Sandboxing* zvolit jeden z existujících a používaných rámců. Pro implementaci je zvolen aplikační rámec Spring, který je hojně používán pro vývoj podnikových aplikací.

Na poli podnikových aplikací, přesněji řečeno portálů, se již objevil princip *Sandboxingu*, a to v podání firmy Liferay [6]. Liferay poskytuje formou pluginu možnost využívat sandboxing portletů ve webové aplikaci. Tuto podporu lze využít pouze v kombinaci s touto platformou a v placené edici, která je určena firmám a podnikům (Enterprise edition). Proto na ně většina institucí nemusí finančně dosáhnout. Praktická část této práce staví na odlišných idejích. Volbou otevřené platformy Java a Spring si hledá místo a uplatnění v širší škále existujících aplikací mezi systémy využívající aplikační rámec Spring. Dále je kladen důraz na oddělení menší části aplikace než celé portletové aplikace. Implementace je navržena na oddělení libovolné části aplikace a nejmenším oddělitelným celkem je jeden objekt (Java *bean*).

Práce je členěna následovně. Druhá kapitola bude zaměřena na diskuzi problémů, které se u podnikových aplikací vyskytují a kterým budeme zamezovat. Ve třetí kapitole budou ukázány nástroje, které slouží na řešení a prevenci těchto problémů. Další kapitoly se věnují vlastní realizaci a implementaci Java Capsules. Java Capsules je pojmenování projektu, který bude praktickou částí této práce. Po výkladu návrhu architektury a implementace řešení bude vyložen způsob integrace výsledného produktu. Přehled dosažených výsledků

[1] Sandbox je bezpečnostní mechanizmus, jehož principem je oddělení některých programů do samostatného prostředí tak, aby se zamezilo šíření chyb do ostatních oblastí systému. Je tak odstraněna hrozba pro okolní programy [12].

z hlediska výkonnosti, které porovnávají odezvu a propustnost aplikace s ochranou rámce Java Capsules a bez sandboxu je uveden v kapitole 7. Přínos ochrany a zvýšení stability oddělením částí aplikace je diskutován především v kapitole 2.4. Následuje samotný závěr práce.

Kapitola 2

Problémy podnikových aplikací

V této kapitole se zaměříme na klíčové problémy aplikací, na které budeme hledat řešení. Některá probíraná témata budou obecného charakteru. Používaná terminologie je vztažena k systémům na platformě Java, a to zejména na její část zvanou Java Enterprise Edition, která se používá pro implementaci informačních systémů. V momentě, kdy budeme hovořit o aplikaci nebo systému, je zpravidla myšlena aplikace podniková nebo webová nebo informační systém běžící na Java Virtual Machine.

2.1 Stabilita

Vlastnosti každého nedistribuovaného systému jsou do značné míry ovlivněny nejslabším článkem. Proto může zejména výkon a stabilita jednoho systému záviset na úzkém místě takového systému. Pokud se v aplikaci vyskytuje část, která odebírá pro svůj běh neúměrně velké množství výpočetního času nebo paměťového prostoru, zpravidla se tak děje na úkor ostatních subsystémů. Pokud tato problematická část vyčerpá zdroje, dostává se systém do potíží. V krajních případech může tato situace vést až k zastavení nebo pádu celého systému. Systém přestane komunikovat s okolím a je pro svůj účel nepoužitelný.

Pod pojmem stabilní aplikace budeme uvažovat aplikaci, která odpovídá na příchozí požadavky, a dokonce i za nespecifikovaných podmínek se umí zotavit. Do nedefinovaného stavu se může aplikace dostat z několika důvodů. Jako nejběžnější příklad uvedu vyčerpání paměti (`Out of Memory Error`), dále se může jednat o výpadek sítě a rozpojení komunikace, hardwarovou poruchu, přetečení zásobníku, vyčerpání počtu otevřených souboru nebo dosažení maximálního počtu vláken.

Tyto stavy nelze v naprosté většině případu řešit reaktivně, protože zpravidla nelze zabrané prostředky do systému vrátit nebo obnovit stav před poruchou. Aplikace v takovém chybovém stavu zpravidla není schopna plnit svůj účel. Příčiny a řešení problémů jsou komplexní a časově náročné. Pokud v systému navíc neužíváme prostředky pro analýzu, může být odhalení příčiny téměř nemožné (více v kapitole 3.1). Stabilitu mohou ovlivňovat i produkty třetích stran, které nejsou pod naší správou. Může se jednat o dodavatele do stejné aplikace, ale i o využité aplikační rámce a knihovny.

Pokud má chybový stav za následek selhání systému, mluvíme o tzv. pádu aplikace. Kritický pád aplikace je zpravidla řešen jako incident v ostrém provozu, kde není možné tolerovat dlouhodobý výpadek. Nejjednodušším známým řešením pak může být restart celé aplikace, a to i přesto, že problém může být lokálního charakteru a nemusí zapříčinit snížení dostupnosti všech částí systému. Restart celé aplikace potom vede k odstávce systému.

Tato odstávka, ať už se jedná o manuální, či automatické zotavení systému, může být pro zákazníka kritická a tvůrci aplikace mohou být vystaveni sankcím v rámci dohody o úrovni poskytovaných služeb (SLA - Service Level Agreement)[1].

Jedna z nejčastějších příčin pádu aplikace je způsobena vyčerpáním paměti. Není to ovšem případ, kdy bychom systému zapomněli nastavit dostatečný příděl operační paměti, ale jde o únik paměti (tzv. *Memory Leak*). Pády na nedostatek paměti bych ze své zkušenosti z praxe rozdělil na dvě skupiny. V prvním případě se jedná o *chybu na straně aplikace*, ať už naši nebo využívaného produktu. V druhém případě může být část systému dočasně přetížena aktuálním provozem. Příkladem takového případu může být rozesílání měsíčních vyúčtování, SMS kampaně apod., ale i v těchto případech není chyba v implementaci aplikace vyloučena. V obou případech budeme mít za cíl prostředí v co nejkratší době zotavit. V práci budu hledat řešení na předcházení pádu celé aplikace a na zotavení bez nutnosti odstávky celého systému.

2.2 Přidělování prostředků

Předchozí případ, který poukazuje na slabý článek v systému, může být doplněn dalším negativním faktorem. Tím je nemožnost nastavovat konfiguračně na vysoké úrovni abstrakce prostředky, které tento článek smí využít.

V rozsáhlém systému bývá také obvykle více dodavatelů, kteří se dělí o stejné zdroje, které jim jsou v rámci jedné aplikace poskytnuty. Systém pak tyto prostředky přiděluje všem, kdo si o ně požádá. Je tedy velice obtížné přesně vymezit a přidělit zdroje jednotlivým aplikacím v jednom systému. Pokud aplikace jednoho dodavatele vyčerpá všechny zdroje, bude tím postižen celý systém bez rozdílu. Není tak možné například pro dva dodavatele rozdělit operační paměť a čas procesoru na poloviny. V některých případech to ani nebude vhodné.

Prakticky bude v takovém případě rozumné shora omezit prostředky pro jednotlivé části aplikace různých dodavatelů. Tím dosáhneme zejména toho, že aplikace nebude mít možnost neúměrně čerpat prostředky na úkor jiné své části.

2.3 Škálovatelnost

Požadavky systému se při jeho rozvoji a přidávaní nové funkcionality mohou přirozeně zvýšit. Takto zvyšující se požadavky nemusí být možné z technických důvodů uspokojit v rámci jedné výpočetní jednotky. V takovém případě by bylo vhodné aplikaci rozdělit do vzájemně komunikujících celků. Na takový zásah není zpravidla architektura současného řešení připravena. Při vysoké složitosti kódu aplikace ani nemusí být takový úkon efektivně proveditelný.

V případě, že by bylo možné aplikaci rozdělit na komunikující celky, máme řešení na právě zmíněné problémy. Vznikl by tak komunikující distribuovaný systém, jehož odolnost, stabilita a přidělování prostředků je dáno vlastnostmi nezávislých uzlů. Toho lze dosáhnout pomocí existujících řešení. Příkladem může být použití Java API for XML Web Services (JAX-WS), Java API for RESTful Web Services (JAX-RS) nebo Java Messaging Services (JMS). Tyto technologie definují komunikační rozhraní mezi systémy. Vyžadují

[1] „SLA je portfolio metrik je stěžejním parametrem rozhrani mezi odběratelem a externím poskytovatelem infomatických služeb“ [16]

však implementační zásah a znalost dané technologie. Tyto technologie musí být integrovány na úrovni architektury a jejich dodatečné zavedení do komplexního systému nemusí být vždy jednoduché.

2.4 Princip zvýšení stability

Pokud bychom byli schopni transparentně a konfiguračně aplikaci rozdělit na samostatné celky, které spolu komunikují, byly by vlastnosti celého systému určeny parametry jednotlivých částí nezávisle na ostatních. Pro označení takovéhoto oddělení aplikace existuje pojem *Sandbox* [12]. Princip využívání toho mechanismu se nazývá *Sandboxing*.

Sandbox je bezpečnostní mechanizmus, jehož principem je oddělení některých programů do samostatného prostředí tak, aby se zamezilo šíření chyb do ostatních oblastí systému. Je tak odstraněna hrozba pro okolní programy [12]. Použití Sandboxu je vhodné i v případě, kdy chceme použít nedůvěryhodný software.

V podání této práce a její implementační části se tedy bude jednat o Sandboxing na platformě Java. Sandbox bude představován pomocí samostatného procesu. V případě pádu části aplikace není ohrožen celý systém a je možné tento subsystém resetovat samostatně. Dosáhneme tak řešení problémů stability a zotavení (kapitola 2.1). Sandbox představovaný Java procesem je možné samostatně konfigurovat a omezit tak přístup ke zdrojům (kapitola 2.2). Konfiguraci lze provést pomocí argumentu Java Virtual Machine. Argumenty, které jsou spojeny se stabilitou, budou zmíněny v sekci 3.2. Dalším produktem řešení bude možnost definovat Sandboxy na různých strojích. Je to však pouze nadstavba nad samotným Sandboxem. Takovým rozdělením lze zvýšit škálovatelnost aplikace (kapitola 2.3).

Pomocí oddělení běhu Java aplikace do samostatných procesů získáme vlastnosti systému, které řeší předchozí problémy. V systému se bude vyskytovat jeden hlavní uzel, který bude rozdělovat požadavky vydefinovaným odděleným částem částem.

2.4.1 Existující řešení zajištění stability

Koncept oddělování procesů za účelem zvýšení odolnosti aplikací se v prostředí Java již objevil. Jde o řešení společnosti Liferay, která ve své podnikové verzi portálu nabízí možnosti oddělit aplikace. Tato možnost je dostupná pouze v placené distribuci. [6]

Po zapnutí podpory sandboxu je v administraci portálu dostupné uživatelské rozhraní, pomocí kterého lze sandbox vytvořit. Mezi nevýhody Sandboxingu na platformě Liferay bych zařadil zejména jeho dostupnost v placené verzi ve formě rozšíření. Dále ho lze využít až od verze 6.2[2], jeho podpora je závislá na dané verzi portálu Liferay. Poskytuje možnost oddělení pouze celé aplikace. Aplikace však může být složena z více portletových aplikací, které takto oddělit nejde. Naproti tomu řešení, které je popsáno v této práci, bude možné využívat možnosti oddělení procesů napříč platformami a také na více úrovních granularity.

[2]Liferay verze 6.2 je právě teď (2015) nejvyšší produkční verzí tohoto produktu. Nad rámec této verze se objevují pouze opravné balíčky.

Kapitola 3

Zajištění stability Java aplikací

V následující kapitole se budeme blíže zabývat stabilitou aplikací na platformě Java z jiného úhlu pohledu, a sice z hlediska jejich monitorování, přecházení, analýzy a řešení. Zaměříme se na webové a podnikové aplikace, protože jsou hlavním produktem na této platformě.

Příčiny porušení stability tkví v širokém spektru možných scénářů. Tyto případy lze popsat jako neošetřené či nečekané stavy, do kterých se aplikace dostala. Příkladem může být hardwarové selhání, výpadek na síti, neošetřené vstupy aplikace, vyčerpání vláken nebo paměti a mnoho dalších. Uniformní odpovědí na řešení takových problémů je návrat aplikace do známého stavu (např. restart). Tento krok může v krajním případě znamenat obnovení záloh systému (tzv. revert).

Neočekávaný stav jedné části systému může postihnout stabilitu systému jako celku. Zotavení formou restartu pak tento výpadek ještě prodlouží v důsledku doby startování a inicializace. Tento krok po sobě, naneštěstí, uklidí většinu informací pro následnou analýzu. Proto je vhodné při pádu aplikace odebírat z prostředí co nejvíce informací o příčině. Na to je nutné se náležitě připravit, ať už se jedná o znalost nástrojů, nastavení nebo dostatečné místo na disku pro otisky paměti. Této problematice se bude věnovat sekce 3.1.

Problém vyčerpání paměti je nejběžnější příčinou incidentu na systému. Vyčerpání paměti se na platformě Java může projevovat jako neustále se opakující uvolňování nepotřebné paměti (Garbage Collecting), které je doprovázeno sníženou odezvou aplikace. Aplikace nemá prostor pro alokaci zdrojů pro nově příchozí požadavky, přestane odpovídat a stane se nedostupnou. Dalším případem pádu může být zastavení aplikace prostředky operačního systému, a to při vyčerpání paměti nad meze operačního systému, což samozřejmě vede k nedostupnosti.

3.1 Analýza a řešení problémů stability

V textu bude následovat definování pojmů z oblasti Java platformy, ke kterým se pojí nástroje pro monitoring a analýzu problémů, jež budou popsány dále.

3.1.1 Garbage collector

Gargage collector (dále GC) je anglický název mechanizmu pro uvolňování paměti, který je také využíván na platformě Java. Uvolnění paměti je založeno na generačních algoritmech, které vycházejí z předpokladu, že většina objektů žije v aplikaci po krátkou dobu. U každého objektu si na základě počtu spuštěných GC pamatuje věk objektu. Pokud je objekt využíván dlouhodobě, dostane se do prostoru starší generace, který není tak často uvolňován.

V případě zaplnění některého z paměťových prostorů GC automaticky prohledává paměť, vybírá a uvolňuje nepotřebné objekty, na které již neexistuje reference. I když GC automaticky uvolňuje paměť, nejedná se o mechanizmus, který by zabránil únikům paměti (`Memory Leak`), těm musí programátor předejít v kódu aplikace. Pokud se objeví objekt k odstranění, GC odstraňuje celý řetězec závislostí na něm a také umí odhalit cyklické shluky nepotřebných objektů. Proces GC lze volat i explicitně, ale důrazně se to nedoporučuje kvůli dopadu na výkon systému [10].

Výběr typu GC a jeho parametrů má přímý vliv na stabilitu a odezvu systému. Pokud máme velkou paměť, její Garbage collecting trvá déle. Přidání paměti JVM tak často není řešením problému „`Out of Memory`“, protože leak se může zvětšovat neustále. Přidání paměti zhoršuje odezvu aplikace při uvolňování paměti.

V kontextu Garbage collectoru lze narazit na následující pojmy, které používá většina typů GC:

- **Mark & Sweep** — toto označení nesou dvě fáze uvolňování paměti na haldě. Obě fáze probíhají během zastavení aplikace.

  Během fáze Mark se označí veškeré objekty, které jsou využívány, a také objekty, jež jsou skrze ně dostupné. Výchozím bodem pro označení je kořenový seznam referencí, skrz který se označí všechny odkazované objekty. Během fáze Sweep je procházena halda, aby se nalezly prostory mezi používanými objekty (neoznačené instance). Tyto prostory jsou označeny jako volné a jsou dostupné pro další alokaci. [15]

- **GC Log** — Java Virtual Machine poskytuje možnost zapnout GC Log. Při této volbě se budou zaznamenávat údaje o uvolňování paměti do souboru. Volby pro zapnutí a konfiguraci GC Logu budou zmíněny v sekci 3.2.

  Záznam je relativně čitelný. Pokud ale budeme analyzovat záznamy z několikadenního provozu při zátěži, může se jednat o soubory s desetitisíci řádků na hodinový záznam. Existují nástroje, které tento záznam převádějí do grafické podoby.

Typy GC

GC je možné pomocí několika aspektů [5] rozdělit na:

- **Sériové a paralelní** — V případě sériového GC je uvolnění paměti provedeno v hlavním vlákně, kdežto u paralelního GC v samostatných vláknech.

- **Konkurentní a „Stop-the-world“** — konkurentní typ GC se snaží spouštět GC a nezastavovat běh aplikace. „Stop-the-world“ GC má za následek zastavení celé aplikace až do dokončení uvolňování paměti. K pozastavení aplikace musí dojít vždy. Důvodem je označení objektů k uvolnění, které musí být označeny za konzistentního a neměnného stavu.

- **Minor a Major** — při zaplnění *eden* se provede Minor GC a je velmi rychlý. Major, také Full, GC je spuštěn při zaplnění *Turnered* a je řádově časově náročnější. (Zmíněné paměťové prostory budou vyloženy v 3.2.1).

- **Kompaktní, nekompaktní a kopírovací** — tento typ GC určuje přístup k fragmentaci. Všechny kompaktní objekty po dokončení GCing defragmentuje do jednoho celku. Nekompaktní objekty zanechá na svém místě a ponechá je tak fragmentované.

Kopírovací přístup je defragmentace, kde jsou objekty kopírovány do nového paměťového prostoru. Přínosem tohoto přístupu je fakt, že zdrojová oblast je připravena k alokaci. Na druhou stranu kopírování vyžaduje delší výpočetní čas.

3.1.2 Nástroje

Java poskytuje několik standardních mechanizmů pro kontrolu, analýzu a řešení problémů (nejen problémů stability, na které se zaměříme), které se v rámci Java Virtual Machine objeví. Užitečnou sadu nástrojů také doplňují produkty třetích stran.

Nástroje součástí JDK[1]

`jps` Jedná o nejjednodušší nástroj v této sekci. Jeho použití však bude pravděpodobně předcházet použití většiny následujících. Zobrazí Process identifier (PID) všech Java procesů.

`jinfo` Pokud máme PID procesu, je možné prohlížet argumenty JVM. V prostředí, kde se používá kaskáda startovacích skriptů, se jedná o nejspolehlivější zjištění aktuální konfigurace JVM.

`jmap` Pomocí nástroje jmap je možné odebrat *Heap dump*. Jde o obraz paměti v aktuálním stavu. Tento obraz může být klíčovým materiálem pro zjištění úniku paměti. Jeho velikost je přímo úměrná velikosti paměťového prostoru haldy.

`jstack` Výstupem tohoto nástroje je textový soubor zvaný thread dump. Obsahem je výpis všech vláken, které se v JVM v aktuální moment vyskytují. Z tohoto výpisu je pak možné zjistit řádku kódu, kde se vlákno vyskytuje. Agregovanou informací z tohoto záznamu může být počet vláken se stejným jménem nebo ve stejném místě volání.

Vlákna mají přímý vliv na velikost obsazené paměti. Pokud mají vlákna velký zásobník, může dojít k vyčerpání paměti jejich zvýšeným počtem.

`jconsole` Na rozdíl od předchozích nástrojů je jconsole opatřena grafickým rozhraním. Poskytuje informace z části srovnatelné se všemi předchozími nástroji. Po připojení se k procesu je možné nahlížet na využití paměti, počet vláken, využití CPU a počet zavedených tříd. Nalezneme zde i přehled argumentů JVM podobně jako v nástroji `jinfo`.

`VisualVM` Nástroj VisualVM je velmi podobný nástroji jconsole. Je rozšiřitelný pomocí řady zásuvných modulů a poskytuje tak detailnější a komplexnější analytické informace o virtuálním stroji Javy. Součástí Java Development Kit, Standard Edition je od verze 6, update 7.

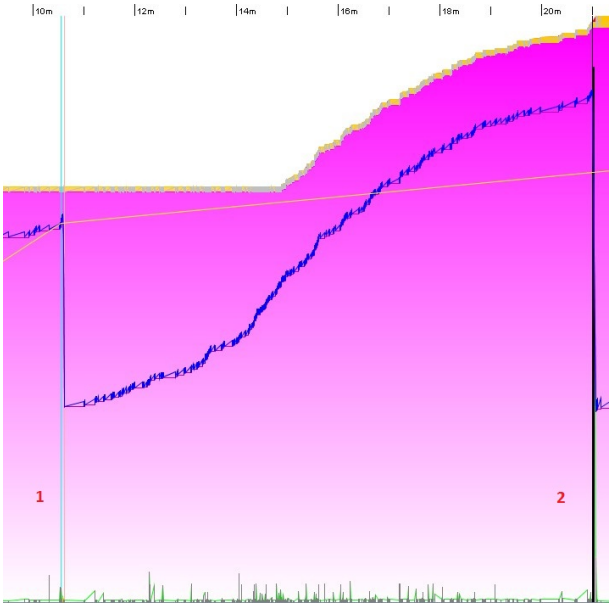
Ostatní nástroje

`GCViewer` Jedná se o software[2], který je schopný vizualizovat GC Log výstup JVM. Obrázek 3.1 ukazuje příklad reálného GC Logu. Vertikální osa y udává velikost haldy (2 GB) a v ose x je zaznamenán čas od startu systému. Pomocí barev potom průběhy Garbage

[1] http://docs.oracle.com/javase/7/docs/webnotes/tsg/TSG-VM/html/tooldescr.html
[2] http://sourceforge.net/projects/gcviewer

collectingu. Světle modrá barva u čísla 1 znamená Minor GC, který sníží úroveň zabrané paměti (tmavě modrá křivka). Dalším význačným bodem je černá barva v okolí čísla 2. Zde probíhá tzv. „Full Garbage collecting“, který je doprovázen zastavením JVM. Pokud se černá barva vyskytuje opakovaně, znamená to, že JVM stojí a neodpovídá. Příčinou je zaplněná paměť, kterou nelze uvolnit. Důsledkem může být pád aplikace.

Ostatní barvy a křivky zachycují stav odlišných paměťových prostorů a práce s jejich pamětí.



Obrázek 3.1: Vizualizace gc logu pomocí GCViewer

`MAT` Pod touto zkratkou se skrývá nástroj Memory Analyzer[3]. Tento nástroj je velmi užitečným pomocníkem pro analýzu problémů způsobených chybou vyčerpání paměti. Vstupem je soubor *Heap Dump*, který získáme například pomocí nástroje `jmap`. Aplikace nám poskytne agregovaný výstup nad tímto otiskem paměti. Z přehledu lze odhalit příčinu vyčerpání paměti.

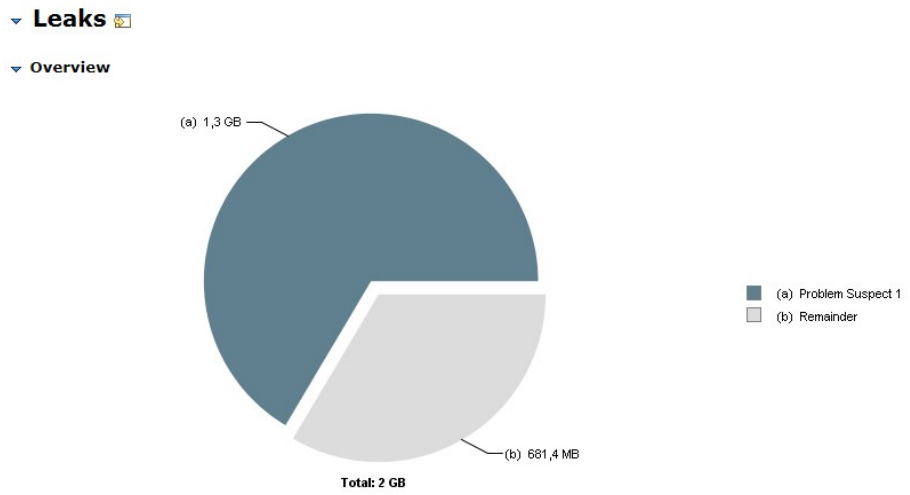
MAT poskytne uživateli kompletní přehled nad všemi objekty, které se nacházely v paměti v době odebrání otisku. Zároveň agreguje stejné třídy a tak dokáže označit ty třídy, které zabírají velké množství paměti, a to i v případě, že se jedná o velký počet malých objektů nebo kaskádu navázaných objektů.

Obrázek 3.2 ukazuje, jak vypadá reálný Memory Leak. Jedná se o případ, kdy je v jedné třídě drženo 66,46 % paměti, což je 1,3 GB. Dále v nástroji zjistíme typ objektů, které paměť zabírají, což nás dovede k příčině problému. Tento případ měl za následek pád aplikačního serveru a nutnost restartovat aplikaci.

3.1.3 Java Management Extensions

Java Management Extensions (zkráceně JMX) je označení pro standardní Java technologii, která doplňuje spektrum dostupných nástrojů. JMX zpřístupňuje rozhraní aplikačních tříd.

[3] http://www.eclipse.org/mat



Obrázek 3.2: Odhalení Memory Leaks v programu MAT

Nedílnou součástí je tedy implementace takového rozhraní na straně aplikace.

Tento mechanizmus implementuje mnoho existujících aplikačních serverů a aplikačních rámců, aby vystavily své interní třídy pro správu prostřednictvím JMX. Pomocí takového rozhraní, lze za běhu zasahovat do objektů aplikace. Systém může také posloužit pro analýzu, monitoring a také případné odvrácení zdánlivě nevyhnutelného pádu aplikace. Technologicky je toto rozhraní řešeno pomocí RMI (Remote Method Invocation).

Pomocí JMX pouze vystavujeme rozhraní objektů. Objekt, který vystavuje své rozhraní, se nazývá `MBean` (Manager Bean[4]). K připojení na MBean slouží například nástroj `jconsole` [11]. Pro přístup k MBean lze využít i jiné metody. Mezi jednoduchou metodu patří webové rozhraní. Takové rozhraní poskytují aplikační servery (např. JBoss, WebLogic) v podobě administrační konzole. Webové rozhraní JMX implementují i nezávislé projekty. Jedním je `jminix`[5].

Využití JMX

JMX implementuje mnoho komponent aplikačních serverů a rámců. Získáme tak přístup a možnost nahlížet do jejich prostředků. Máme tak pod kontrolou cache, Java Messaging Services (JMS) fronty, logování, Garbage Collector, thread pools a mnoho dalších. Tyto komponenty lze monitorovat a spravovat pomocí JMX a zásahem do těchto komponent můžeme odvrátit pád aplikace. Například ručním obnovením JMS spojení, zničením vláken, která uvízla v části systému apod.

Pokud budeme chtít využít JMX nad svým řešením, třída musí implementovat jakékoliv rozhraní, pomocí kterého bude zpřístupněno v JMX. Pod tímto rozhraním je instance zaregistrována jako MBean a její implementace provádí námi definovanou činnost. Lze například nahlížet na velikost proměnných nebo odstraňovat záznamy z kolekcí. Možnosti jsou omezeny pouze naší implementací.

[4]Bean je označení pro instanci objektu na platformě Java, která je pojmenovaná a lze ji referencovat pomocí daného identifikátoru.

[5]http://code.google.com/p/jminix

3.2 Konfigurace Java Virtual Machine

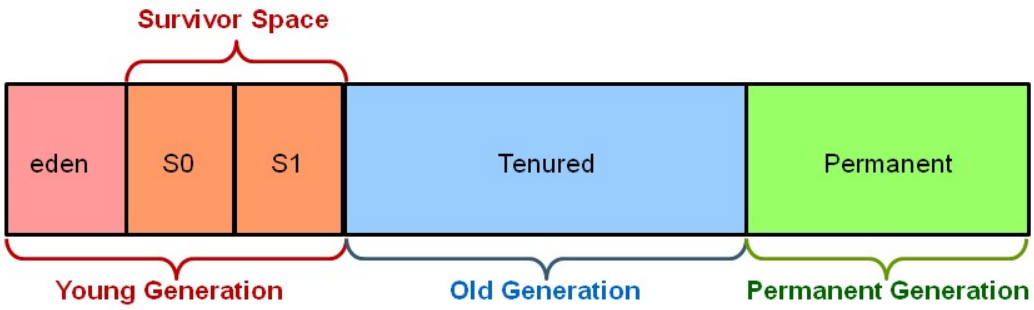
Doteď jsme se zabývali spíše monitorováním a reaktivním řešením problémů stability na JVM. Nyní si představíme konfigurační parametry JVM, pomocí kterých můžeme ovlivnit stabilitu JVM proaktivně. Tyto parametry mohou mít vliv na stabilitu a odezvu systému. Správným použitím se dají zlepšit i vlastnosti dobře fungujícího systému.

Některé parametry, které budou uvedeny, jsou praktické pro monitoring a analýzu problému na JVM, zejména problémů spojených s pamětí, protože tento problém je častou příčinou pádu Java aplikace. Přidání paměti, kterou bude mít JVM k dispozici, často nemusí být řešením. Naopak to bude mít za následek horší odezvu při Garbage Collectingu. Navyšování paměti zpravidla není nutné, protože volání aplikace by mělo být bezstavové a nemělo by zanechávat v průběhu času narůstající alokovaný prostor.

3.2.1 Paměťové prostory JVM

Pro další výklad je nutné blíže se podívat na paměťový prostor haldy JVM.

Paměťový prostor JVM je pro účely Garbage collectingu rozdělen do několika částí, jak ukazuje obrázek 3.3. Tyto části představují generace. Objekt na základě doby své existence v paměti prochází těmito generacemi. Tyto generace jsou rozděleny a posány na základě [3], [13].



Obrázek 3.3: Struktura paměti JVM
(Java Garbage Collection Basics[6])

- **Young Generation:** Paměťový prostor plní nově vytvořené objekty. Většina z nich se v krátkém čase stane nedosažitelnými. Jsou uvolněny tzv. minor GC.
  - **Eden:** Pokud je objekt vytvořen, je umístěn v této části. Objekty opouští tento prostor po prvním uvolňování.
  - **Survivor Space:** Prostor přeživších se plní takovými objekty z *Eden*, které nejsou uvolněny nejbližším během GC. Tato paměťová oblast je rozdělena na dvě. Pokud je některý z nově vytvořených objektů potřebný i po prvním spuštěním GC, dostává se do *Survivor Space S0*. Pokud při dalším GC nemůže být uvolněn, dostává se do sekce *S1*. Obsazenost těchto prostorů se střídá a jeden je tedy vždy prázdný, je to dáno funkcí GC.
- **Old Generation:** (také *Tenured*) Pokud nemohou být objekty nadále uvolněny z *Young Generatiron*, dostávají se do tohoto prostoru. Ten je zpravidla větší a jeho

[6] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

uvolňování probíhá méně často a trvá dále. Jeho GC je označován jako Major GC nebo Full GC.

Spolu s *Young Generation* tvoří prostor zvaný jako *Heap*.

- **Permanent Generation:** Jiným označením prostor metod (method area) ukládá např. instance tříd typu `Class` (metadata, která popisují třídy) nebo interní řetězce. Do tohoto prostoru se nedostávají objekty z *Old Generation*. Prostor je plněn na základě používaných tříd v aplikaci. Jeho uvolňování také spadá pod Major GC.

3.2.2 Konfigurace JVM

Java Virtual Machine (JVM) lze konfigurovat pomocí parametrů, které jsou Java procesu předány v době spuštění. Několik z nich bude popsáno.

Garbage Collector

Při konfiguraci je nutné si uvědomit, že při aktivaci GC dochází vždy k zastavení běhu aplikace. Některé zmíněné nastavení se pouze snaží minimalizovat jejich dobu pozastavení, použitím více vláken nebo označováním za běhu. Finální výběr kandidátů na uvolnění ale musí být proveden nad konzistentním stavem paměti, proto se aplikace zastaví.

Volba parametrů GC spolu s natavením paměti ovlivňuje odezvu systému a nevhodné parametry mohou aplikaci za určitých podmínek shodit.

`-XX:+DisableExplicitGC` — Tímto příkazem lze zakázat explicitní volání Garbage collectingu. Ve výchozím nastavení je toto volání povoleno.

`-XX:+UseSerialGC` — GC použije algoritmus *Mark–Sweep–Compact*. První dvě fáze proběhnou tak, jak je popsáno v sekci 3.1.1. Třetí je označení pro fragmentaci. Jde o nejjednodušší typ GC, který sériově prochází paměť. Je vhodný pro jednoduché jednovláknové aplikace s malým využíváním paměti cca do 100 MB. [4]

`-XX:+UseParallelGC` — Tento typ GC provádí svou činnost na *Young Generation* jako sériový GC pomocí více vláken. Je vhodný pro systém s větším počtem jader CPU a větší pamětí. GC uvolňuje paměťový prostor *Young Generation* paralelně v samostatných vláknech. Těchto vláken je ve výchozím nastavení `N`, kde `N` je počet procesorů. Tuto hodnotu lze měnit, viz dále. GC proto není vhodný pro jednoprocesorové stroje. [4]

`-XX:+UseParallelOldGC` — Jedná se o stejný přístup jako v předchozí volbě, ale cílovým prostorem je *Old Generation*.

`-XX:+UseConcMarkSweepGC` — Concurrent Mark Sweep (CMS) GC se snaží minimalizovat pauzy způsobené GC tak, že provádí uvolňování za běhu. Tím se docílí kratšího trvání při jeho vyvolání GC. [4]

`-XX:+ExplicitGCInvokesConcurrent` — Ve spojení CMS aplikuje paralelní uvolňování při explicitním volání.

`-XX:+UseG1GC`[7] — GC je principem podobný CMS. Má sloužit jako jeho náhrada. Je určen pro velké paměťové prostory na multiprocesorových strojích. G1 vykazuje stabilní výsledky při paměti 6 GB a větší s pauzami nižšími než 0,5 s. [1]

[7] od Java 7

`-XX:ParallelGCThreads=<n>` a `-XX:ParallelCMSThreads=<n>` — Možnost specifikovat počet vláken, které provádějí CMS a ParallelGC.

Pro bližší srovnání je možné navštívit zdroj [14].

Nastavení paměťových prostorů

Následující parametry označené * jsou sufixovány pomocí jednotky množství dat, např.: 16b, 32k a 64m pro (16 bajtů, 32 kilobajtů a 64 megabajtů). V případě výchozí a maximální hodnoty se po startu JVM alokuje výchozí hodnota velikosti paměti a roste do jejího definovaného maxima.

`-XX:NewSize*` — Výchozí hodnota pro velikost *Young Generation*.

`-XX:MaxNewSize*` — Maximální hodnota pro velikost *Young Generation*. Tato oblast je nejčastěji využívanou a probíhá nad ní častý úklid paměti.

`-XX:NewRatio=<n>` — Poměr rozdělení *Young Generation* a *Old Generation*

`-XX:SurvivorRatio=<n>` — Poměr rozdělení *Young Generation* na oblasti *Eden* a *Survivor Space*

`-Xms*` — Výchozí hodnota pro alokaci paměti pro *Young Generation* + *Old Generation* (výchozí velikost).

`-Xmx*` — Maximální hodnota pro alokaci paměti na *Young Generation* + *Old Generation*. Zvětšením této hodnoty nemusíme vůbec vyřešit `Memory Leak`, problém se tím pouze oddálí. Navíc se zhorší odezva systému v důsledku delšího uvolňování.

`-XX:PermSize=*` — Výchozí hodnota prostoru *Permanent Space*.

`-XX:MaxPermSize=*` — Maximální hodnota prostoru *Permanent Space*. Hodnota závisí na počtu použitých tříd v aplikaci. Pro velké projekty nebo časté přenasazování musí být větší. Pokud nastavíme maximální hodnotu stejnou jako výchozí, nebude docházet ke zvětšování paměti, které je doprovázeno *Full GC*. To je vhodné pro zvýšení výkonnosti.

`-Xss*` — Nastaví velikost zásobníku. Pokud je tato hodnota příliš malá, bude nastávat přetečení, při velké hodnotě a velkém počtu threadů dochází k rychlejšímu vyčerpání paměti.

Ladění aplikací

Nastavení pro ladění aplikací, která ovlivňují chovaní JVM z hlediska ladění aplikací.

`-XX:-HeapDumpOnOutOfMemoryError` — Tento parametr zapne automatické vytváření otisku paměti při chybě na nedostatek paměti. Výstup v takovém případě odpovídá `jmap` a může být analyzován pomocí `MAT`.

`-XX:HeapDumpPath=<path>.hprof` — Definice místa uložení otisku paměti.

`-XX:OnOutOfMemoryError=<cmd args;cmd args;...>` — Příkazy, které se provedou při chybě na nedostatek paměti.

`-XX:OnError=<cmd args;cmd args;...>` — Příkazy při fatální chybě.

`-verbose:gc` — Zapne logování garbage collectingu do GC Logu (3.1.1).

`-Xloggc:<gc.log>` — Definice umístění gc logu.

`-XX:+PrintGCDetails` — Zapne detailní výpis gc.

`-XX:+PrintGCTimeStamps` — Zapne tisk časových známek gc.

Kapitola 4

Návrh

4.1 Analýza požadavků

V následujících podkapitolách bude provedena analýza požadavků, které musí výsledná implementace splňovat. Budou specifikovány základní požadavky a požadavky na jednotlivé úrovně oddělení aplikací.

4.1.1 Základní požadavky

Implementace Sandboxingu bude podléhat několika kriteriím, aby ho bylo možné efektivně využít v existující aplikaci.

1. Respektovat specifikaci v kapitole 4.3, tj. vytvoření nezávislých oddělených částí, konfigurační charakter, automatické oddělení a návrat do konzistentního stavu
2. Integrace musí být v maximální míře neinvazivní, tj. že musí klást co nejmenší nároky na úpravy existující aplikace při zavedení či odstranění.
3. Integrace musí být dostupná pomocí rozhraní vysoké úrovně.
4. Všechny důležité části aplikace musí podléhat možnosti konfigurace.
5. Zároveň však musí tam, kde je to možné, poskytovat výchozí hodnotu tak, aby uživatele zbytečně konfigurace neobtěžovala nebo se nedopustil chyb.
6. Možné konfigurace aplikace odvodí z prostředí programu, kam je integrována.
7. Aplikace musí počítat se zavedením do různých prostředí a různých verzí knihoven, např. Spring Frameworku.
8. Musí být použity co nejnižší a stabilní verze knihoven a dopředná kompatibilita, aby se nesnižovala škála aplikací, kde bude možné řešení využít.
9. Aplikace musí v maximální míře využívat existující funkcionalitu z aplikačního rámce Spring.
10. Aplikace musí vystavovat rozhraní pro administrativní zásah do její správy.
11. Aplikace musí klást důraz na rozšiřitelnost.

4.1.2 Spring bean

Oddělování instancí Spring bean má svá specifika nad rámec již zmíněných. Jedná se hlavně o:

1. Nezávislost na kontejneru webových aplikací (Servlet API).
2. Vypořádání se s některými problémy, kterým podléhá Remote Method Invocation (RMI), potažmo Remote Procedure Call (RPC).
   (a) Vzdálené předání referencí.
   (b) Výpadky na síti a rozpojení komunikace.

4.2 Problémy a omezení

Při návrhu a analýze řešení bylo nutné odhalit a definovat jistá omezení, která z problematiky plynou. Jelikož se bude jednat o oddělování procesů nad existující aplikací, tato aplikace musí nadále plnit svůj účel. Jejím účelem je obsluhovat příchozí požadavky, a proto se automaticky v navrhovaném řešení i tato aplikace stává hlavním uzlem, který bude rozesílat požadavky na podřízené uzly.

Omezení a problémy lze rozdělit na:

1. Systémové problémy:
   - Je nutné mít na systému povolené firewall porty, protože sandboxy ke komunikaci využívají sockety.
2. Problémy architektury:
   - Nelze předávat reference, jelikož jsou procesy odděleny a mají svůj adresový prostor. Instance a odkazy na ně budou řešeny a předávány pomocí unikátního identifikátoru.
   - Nelze používat statické proměnné v oddělených procesech, protože budou dostupné pouze v jednom procesu.
   - Nelze vyhledávat systémové zdroje pomocí Java Naming and Directory Interface (JNDI).
3. použití:
   - Předávané argumenty do sandboxu musí být serializovatelné, aby je bylo možné odeslat pomocí socketu. V Javě to pak znamená nutnost u typů argumentů a návratové hodnoty implementovat rozhraní `java.io.Serializable`.

4.3 Návrh Java Capsules

Tato kapitola bude popisovat návrh implementace řešení stability aplikací na platformě Java. Pokud bude zmíněn pojem Java Virtual Machine (JVM), bude vždy myšlena distribuce of firmy Oracle.

Řešení problému stability bude spočívat v rozdělení aplikace na celky, které budou běžet v odděleném procesu. Tím dosáhneme, že v případě výskytu problému zůstane zbytek aplikace neohrožen.

Rozdělení aplikace na takové celky musí splňovat některá základní specifika.

1. Oddělené části aplikace musí být na sobě nezávislé.
2. Navržené řešení musí poskytovat mechanizmus pro automatické uvedení oddělených části do konzistentního stavu (restart).
3. Zavedení mechanizmu oddělení musí být konfiguračního charakteru v rámci aplikace.

Pro účely oddělení připadá v úvahu vlákno, proces nebo samostatný hardwarový prvek. Vzhledem k tomu, že vlákno tvoří samostatný celek pouze z pohledu výpočtu na procesoru, není oddělení pomocí vláken vhodné. Použití samostatných hardwarových jednotek by porušilo bod 3, protože bychom, kromě samotné aplikace museli vytvořit architekturu výpočetních uzlů. Vhodným prostředkem pro oddělení celků je tedy jen proces. Pokud oddělenému celku dojde paměť, neohrozí zbylou funkcionalitu, protože se chyba projeví pouze v rámci jednoho procesu.

Vybraným řešením bude tedy *rozdělení aplikace do více procesů a jejich automatická správa a koordinace* (*Sandboxing*). Pokud v aplikaci oddělíme části do samostatných procesů, nutně nás to zavede k nutnosti meziprocesové komunikace (Inter-Process Communication — IPC). Mezi základní IPC patří signál, socket, roura, sdílená paměť.

Většina IPC je použitelná pouze v rámci jednoho stroje. Vzhledem k tomu, že z řešení nechceme vylučovat možnost běhu Sandboxu na jiném stroji, připadá v úvahu pouze *socket*. Stav, kdy aplikace v odděleném procesu přestane fungovat, bude možné detekovat a podsystém zotavit bez nutnosti výpadku. Jednotlivé procesy bude možné konfigurovat samostatně a tím je omezit. Takto dosáhneme jemnějšího dělení prostředků, než bylo možné v rámci monolitické aplikace, a nakonec navrhované řešení bude možné použít i pro „vzdálené“ sandboxy, kdy hlavní systém a jeho podčást nemusí nutně běžet na stejném stroji.

4.4 Možnosti nasazení

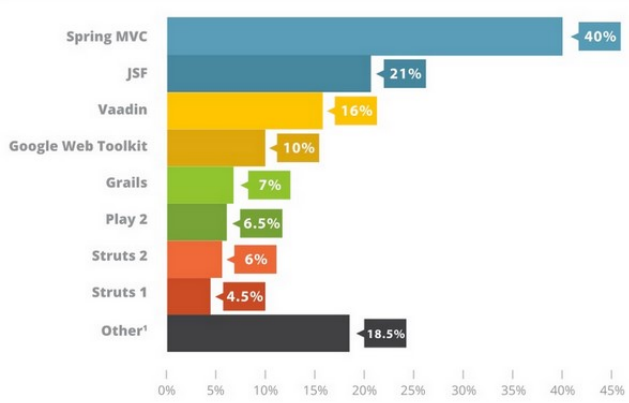
Možnosti použití jsou úzce spojeny s úsilím, které je nutné vynaložit k nasazení takových Sandboxů. Samotný mechanizmus zavedení a udržování Sandoxů musí být sám o sobě velice stabilní, jinak by nemohl řešit problémy stability.

Z těchto důvodů je vhodné zvolit v maximální míře existující řešení, která již poskytují náležitý standard. Možnosti nasazení musí pokrývat širokou škálu aplikačních domén. Tyto prerekvizity mě vedly k výběru prostředků pro realizaci mechanizmu Sandboxu. Implementace bude vedena v maximální míře ve spolupráci a rozšíření aplikačního rámce `Spring`[1].

Podnikové aplikace jsou zpravidla postaveny nad vybraným aplikačním rámcem a právě Spring patří k jedněm z nejpoužívanějších z nich, jak dokládá obrázek 4.1. Nutno podotknout, že Spring nenabízí prostředky pouze pro webové aplikace. Jeho spektrum užití je mnohem širší. Tento rámec je také neinvazivním způsobem integrovatelný do většiny ostatních webových rámců a technologií Java Enterprise Edition. Proto byl Spring vybrán jako dostatečně schopný a rozšířený aplikační rámec pro realizaci použitelného řešení Sandboxingu. Řešení bude tedy možné použít všude tam, kde je použit tento aplikační rámec.

Projekt realizovaný v implementační části ponese název **Java Capsules**. Druhé slovo názvu bylo vybráno jakožto anglický překlad ochranné tobolky léků, jejímž účelem je oddělit a chránit obsah před prostředím a naopak.

[1] http://www.spring.io



Obrázek 4.1: Zastoupení webových aplikačních rámců dle serveru zeroturnaround.com

4.5 Architektura

Architektura řešení, které zajistí sandbox (oddělený proces), bude spočívat ve vytvoření procesu dle současné aplikace. Procesy budou komunikovat pomocí síťových prostředků. Tato komunikace bude synchronní a bude odpovídat modelu *client–server*. Aplikace, kterou je řešení nasazeno, bude plnit úlohy arbitra, který bude přijímat požadavky jako doposud a bude je dále delegovat na oddělený proces. Z toho důvodu nebude možné řešením ochránit jeho funkčnost. Bude mít za úkol:

1. Vytvořit procesy pro sandbox(y)[2].
2. Rozhodnout, jakým způsobem sandbox požadavek obslouží.
3. Serializovat kontext volání metody a odeslat ho na sandbox.
4. Kontrolovat životaschopnost jednotlivých uzlů.

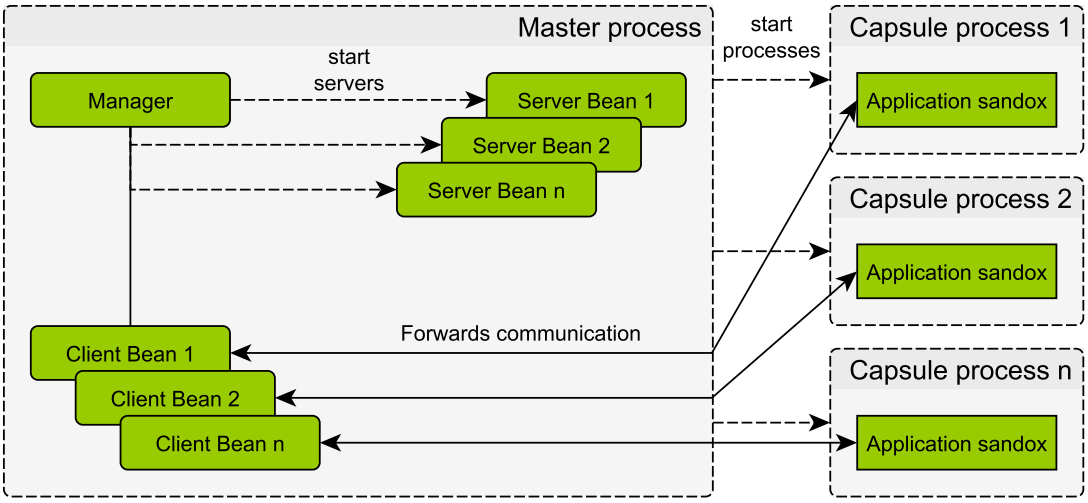
4.5.1 Základní komponenty

Náhled architektury na úrovni Java procesů ukazuje obrázek 4.2 (str. 22). Oddělené části aplikace bude možné definovat rozličným způsobem, aby bylo možné sandbox nasadit na různé aplikační domény (viz. kapitola 4.5.2). Rozdělení aplikace na jednotlivé architektonické vrstvy je zachyceno pomocí obrázku 4.3 (str. 23). Vrstvy jsou rozloženy mezi dva procesy (hlavní a sandbox), komunikující prostřednictvím socketu. Zdroje označené `Sandbox resources` jsou zduplikovány z hlavního procesu a jejich běh je prováděn v rámci sandboxu.

V navrženém řešení budou figurovat 3 základní komponenty:

- **Manager** — Jde o klíčovou komponentu, která uvádí vše do chodu. Jeho úkolem je kontrolovat životaschopnost sandboxů a v případě potřeby je restartovat. Veškerou činnost zaznamenává do logů a publikuje aplikační události, na které je možné libovolně registrovat odběratele.

[2]Pokud se nebude jednat o případ vzdálených sandboxů.



Obrázek 4.2: Oddělení procesů na platformě Java

- **Client** (Master process) — Klientská část je tvořena v rámci existující aplikace (jejího procesu). Klient se po vytvoření samostatného procesu připojí na rozhraní oddělené části aplikace a přeposílá mu požadavky.
- **Server** (Sandbox) — Server je vytvořen v sandboxovaném procesu a obsluhuje požadavky ze strany klienta/klientů. Periodicky také odpovídá na požadavky managera, aby se potvrdilo, že sandbox je schopný odpovídat.

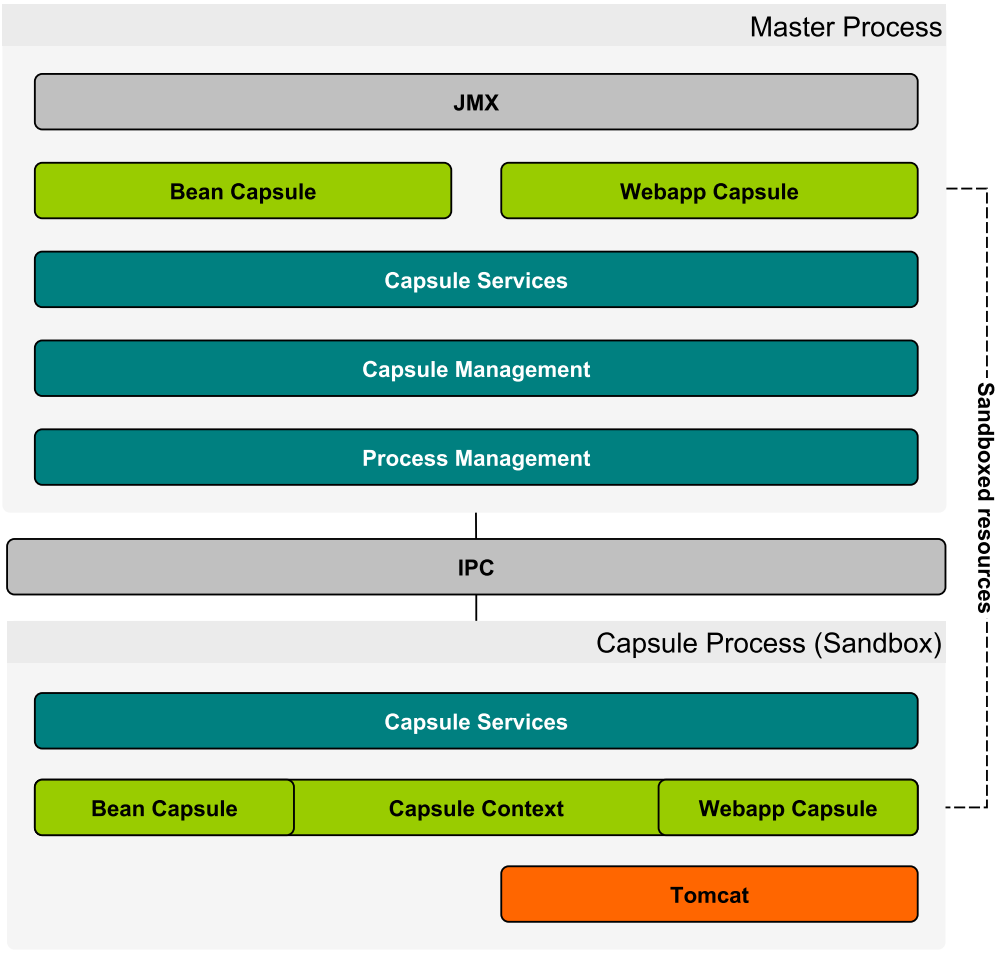
V diagramu 4.6 (str. 25) je znázorněna sekvence volání při zavedení a použití sandboxu. Pomocí aplikace je vytvořen sandbox a manager, který kontroluje životaschopnost sandboxu. Manager periodicky zkouší kontaktovat klienta. Pokud se mu v daném čase nedostane odpovědi, je sandbox zastaven a je vytvořen nový. Všechna volání, která nejsou určena definicí, jako volání do sandboxu, jsou provedena v rámci aplikace. V opačném případe požadavek putuje na sandbox.

4.5.2 Úrovně granularity

Navržená architektura umožní oddělení různých částí či vrstev aplikace. Tyto oddělené části aplikace, které budou chráněny v samostatném procesu, jsou například: beany či skupina bean aplikace, samostatné webové aplikace či jejich části na základě adresy požadavku, jak si ukážeme dále. Pro použití jednotlivých úrovní bude připraveno konfigurační rozhraní v prostředcích aplikačního rámce Spring.

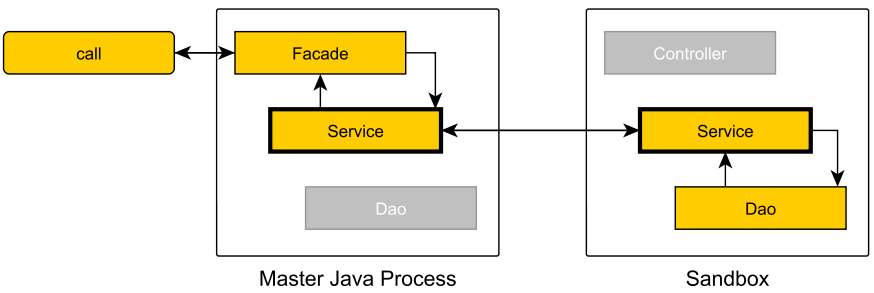
Spring bean

Ve vybraném aplikačním rámci se o aplikační instance objektů stará kontejner, který spravuje jejich životní cyklus a v případě potřeby je dává k dispozici aplikaci. Množinu těchto instancí bude v řešení možné vydefinovat a všechna volání z této množiny budou provedena v rámci sandboxu. Tuto situaci popisuje obrázek 4.4 (str. 23). Na něm je zvýrazněna množina těchto instancí, jejichž volání je provedeno na sandboxu, šedá barva označuje objekty, které nejsou volány, protože se jejich volání neprovádí v daném procesu. Tato množina



Obrázek 4.3: Architektura aplikace pro oddělení procesu (sandboxing)

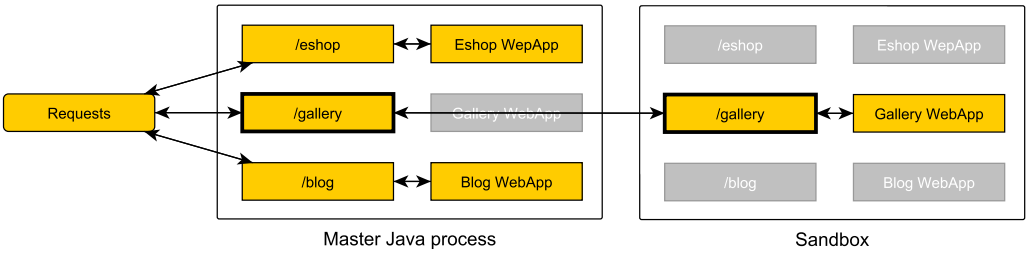
je definována jako $\{x|x.name = Service\}$. Volání této třídy nebude provedeno v původní aplikaci. Instance zvýrazněny šedou barvou zůstanou nepoužity.



Obrázek 4.4: Sandbox/Capsule na úrovní bean

Webové aplikace

Ve srovnání se samotnými instancemi jsou větším celkem podnikové a webové aplikace. Pro ty platí po stránce vytvoření odděleného procesu stejná pravidla, ale jeho samotné nastartování je nutné provést jinak. Webové aplikace jsou závislé svými třídami na knihovnách webových kontejnerů. V případě webové aplikace se v sanboxu nastartuje její protějšek, na který budou přeposílány požadavky. Ostatní nadbytečné aplikace není nutné do sandboxu zavádět (označeny šedou barvou). Tento případ popisuje obrázek 4.5. Referenčním webovým kontejnerem bude `Tomcat`.

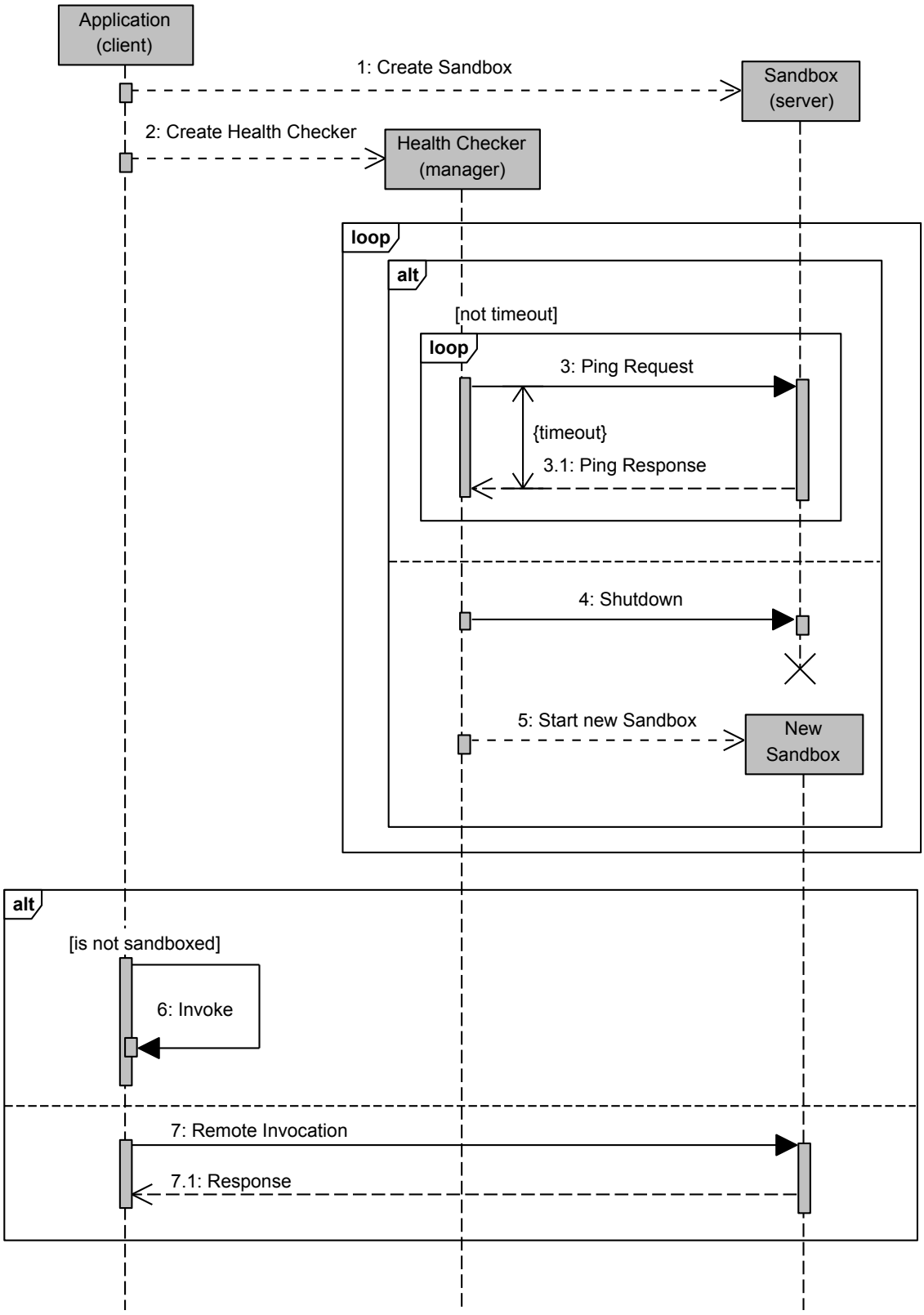


Obrázek 4.5: Sandbox/Capsule na úrovní webových aplikací.

4.6 Správa procesů

Řešení musí být možno spravovat i jinak než automaticky, pomocí manageru. To znamená, že výsledná aplikace musí uživateli umožnit do životního cyklu zasáhnout, dle jeho potřeb, ručně. Pro tento účel bude řešení opatřeno JMX rozhraním pro jeho správu a monitoring. Bude nabízet kompletní škálu operací pro zjištění stavu a operace životního cyklu každého sandboxu.

Pro přístup do JMX rozhraní existuje řada aplikací s grafickým uživatelským rozhraním. Zástupci jako `jconsole` nebo `VisualVM` byly zmíněni v předchozí kapitole. Stejně tak, že existují nadstavby pro přístup pomocí internetového prohlížeče, které poskytují aplikační servery. Také existují samostatná řešení.



Obrázek 4.6: Návrh práce oddělených procesů

Kapitola 5

Prostředky a nástroje

5.1 Platforma

Realizace cílového produktu na sandboxing aplikací (Java Capsules) je od začátku mířena na platformu Java, konkrétně Java Enterprise Edition. Aplikacemi na této platformě jsou typicky webové aplikace, tedy aplikace na serveru, které mohou být dostupné prostřednictvím webového prohlížeče. Dalšími aplikacemi mohou být dávkové úlohy, aplikace pro práci s databází (např. Extract, Transform and Load (ETL)), nebo konzumenti či producenti integračních služeb (Web Services — SOAP, REST). Výsledkem bude možnost konfiguračně zavést sandboxing do existující aplikace na výše popsané platformě. Nejjednodušeji pak lze sandbox využít v aplikacích, které již využívají aplikačního rámce Spring, nad kterým je postaven projekt Java Capsules.

Přehled jednotlivých použitých knihoven a aplikačních rámců je součástí přílohy B.

5.2 Vybrané technologie

5.2.1 Aplikační rámec Spring

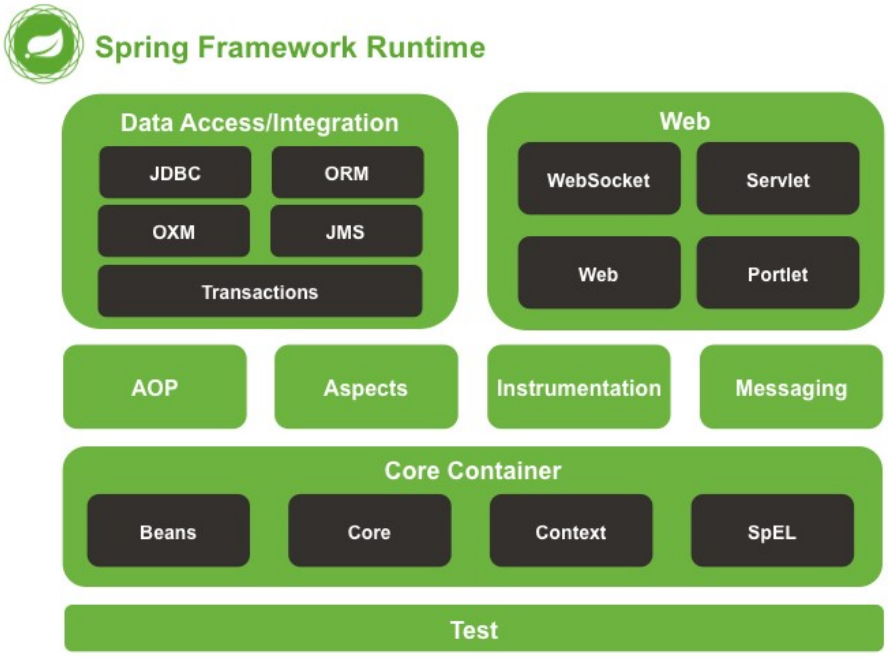
Aplikační rámec Spring je soubor několika aplikačních rámců a knihoven, které integruje do jednotného API vyšší úrovně. Dle slov autorů se jedná o Java platformu, která poskytuje prostředky pro vývoj Java aplikací a dovoluje autorovi se soustředit na implementaci samotné aplikace. Mezi nejdůležitější součásti Springu patří implementace Inversion of Control (IoC) kontejneru a Dependency Injection. Mezi další pak moduly pro podporu testování, objektově relační mapování, webové a portletové aplikace, Aspect Oriented Programing (AOP) a mnoho dalších. [7]

Mezi nejvyužívanější moduly z Spring (obrázek: 5.1, str. 27) v praktické části této práce patří: Core, Context, Web, Servlet, AOP, Test a další.

5.2.2 Aspektově orientované programování

Aspektově orientované programování přináší do aplikací nový rozměr implementace. Většina moderních programovacích jazyků je modulárních a člení svoji logiku do modulů. Aspekty přináší koncept, který aplikuje svoji funkcionalitu napříč moduly přes různé typy a objekty. Tento princip může být také označován jako „crosscutting concerns“.

Spring pro zajištění AOP využívá a zapouzdřuje knihovny jako *CGLIB* a *JDK Dynamic Proxy* a vychází z *AspectJ*.



Obrázek 5.1: Spring Framework (http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html)

Spring definuje názvosloví důležitých částí AOP [8]:

- Aspekt — je označení funkcionality, kterou je možné využívat napříč moduly. Například: logování, transakce, kontrola oprávnění.
- Join point — reprezentuje místo provádění programu (metody), kde je prováděn Aspekt.
- Advice — je implementace aspektu. Může být realizována několika typy jako před, za, okolo volání nebo při výjimce.
- Pointcut — je predikát, pomocí kterého je definována množina modulů, tříd, typů, ... nad kterými bude proveden aspekt. Pointcut bude hrát klíčovou roli v dalším textu.
- Target object — je cílový objekt, nad kterým je prováděn Aspekt.
- AOP proxy — je obalující objekt cílového objektu. Proxy provádí funkcionalitu aspektu. Jedná se o JDK dynamic proxy nebo CGLIB proxy.

Aspektově orientované programování neboli AOP je využito jako základní kámen oddělování Spring bean.

Pointcut

Jak již bylo řečeno Pointcut slouží k definici prvků, které bude pokrývat daný aspekt. Spolu s Advice tvoří základní kameny AOP. Advice definuje činnost a Pointcut, kde je ji třeba provést.

V aplikačním rámci Spring existuje více druhů této definice. Mezi ty obecnější a tím i silnější patří AspectJ Expression Pointcut. Kompletní přehled a příklady jsou k vidění v referenční dokumentaci Spring [8]. Pro ilustraci uvedu některé z nich:

- `execution(public * *(..))` — všechny public metody
- `within(com.service..*)` — vše v balíku a podbalíkách `com.service`
- `@target(org.springframework.transaction.annotation.Transactional)` — všechny cílové objekty, které mají anotaci `Transactional`.

5.2.3 Apache Tomcat

Apache Tomcat je open source projekt, který implementuje Servlety a Java Server Pages z Java platformy. Jedná se servletový kontejner (nikoli o aplikační server).

Tomcat v kombinaci se Spring Frameworkem poskytuje škálu prostředků pro realizaci webových aplikací. Oproti ostatním podobným dostupným produktům vyniká tím, že je otevřený, zdarma a s menšími nároky na zdroje. Toto jsou důvody, proč byl Apache Tomcat zvolen jako referenční platforma pro webové aplikace v této části.

Tomcat je možné využít také v takzvané „embedded“ verzi. V praxi to znamená, že není nutné, aby tento kontejner byl spouštěn jako vlastní proces. Lze ho také vytvořit a využít uvnitř aplikace. Jinými slovy, lze si programově za běhu vytvořit a nakonfigurovat servletový kontejner. Embedded Tomcat je proto použit jako kontejner v procesu, který vytváří sandbox.

5.2.4 Maven

Projekt využívá nástroj Maven na kompilaci a sestavení jar archívu. Následující příkazy předpokládají soubor `mvn` na PATH systému. Pro vytvoření archívu stačí zadat: `mvn clean install`. Vytvořený archív bude dostupný jako maven závislost nebo v maven repozitáři jako jar soubor. Pro spuštění rychlé kompilace bez testů pak `mvn clean install -DskipTests=true`.

Kapitola 6

Prvky Java Capsules

Integrace do existujících aplikací využívající aplikační rámec Spring je možná, řekl bych až snadná. Jediný povinný parametr je port nebo url adresa. Dále stačí zadefinovat manager a vše je připraveno k použití.

Protože se pod třemi základními komponentami klient, server a manager skrývá více než pouhé tři instance bean, je knihovna Java Capsules opatřena vlastním jmenným prostorem `capsule` pro konfiguraci Springu pomocí XML. To znamená, že pokud je funkce základní komponenty složena z více objektů, které mezi sebou mají asociaci tak je reprezentuje jeden konfigurační XML element. Předpis pro definici je v souboru `capsule-1.0.xsd`.

V následujících kapitolách bude vždy uveden minimalistický příklad konfigurace a v příloze C potom všechny konfigurace parametrů s jejich výchozí hodnotou.

Základem každé XML konfigurace Springu je kořenový element `beans`, který také nese jména jmenných prostorů a jejich umístění. Tam si také zadefinujeme jmenný prostor `capsule`. Element `beans` pak musí obsahovat kořenový element Java Capsules. V našem případě `capsule:capsules`, který je nutný, protože obsahuje konfiguraci profilu[1]. V dalším textu budou tyto kořenové elementy (Spring - `beans` a Java Capsules - `capsule:capsules`) vynechány a bude ukázána konfigurace dané komponenty, která by se nacházela uvnitř. Konfigurace by vypadala tedy následovně:

Zdrojový kód 6.1: Kořenová XML konfigurace

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:capsule="http://www.ibacz.eu/schema/capsule"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.ibacz.eu/schema/capsule
      http://www.ibacz.eu/schema/capsule/capsule-1.0.xsd">

    <capsule:capsules>
        <!-- client, server, manager, config -->
    </capsule:capsules>
</beans>
```

[1] Profil je prostředek Springu, kterým lze odlišit, kdy se jaká část konfigurace má/nemá spouštět. V případě Java Capsules se pomocí něj rozlišuje, zda Spring běží jako sandbox proces nebo ne.

6.1 Přínos zavedení

Zavedením Java Capsules do existující aplikace získáme možnost oddělit její část do separátního procesu. Tím chrání veškeré zdroje jak procesu existujícího tak nově odděleného. V případě chyby aplikace, ze které se nelze zotavit, pak není nutné odstavit a restartovat celou aplikaci. Z chyby se lze zotavit restartováním oddělené části a tím způsobit pouze částečnou nedostupnost.

Mezi časté chyby, které vznikají za běhu v Javě a je možné omezit jejich následky pomocí sandboxing, patří:

- Memory Leak
- Vyčerpání vláken z thread pool nebo vyčerpání paměti v důsledku velkého množství vláken
- Deadlock
- Race condition
- Rozpojení komunikace
- Vyčerpání počtu spojení např. do databáze
- ...

6.2 Společná konfigurace

Klient a server mají některé společná konfigurace. Jedná se o komunikační protokol, profil (rozlišení sandbox aplikace od původní) a kontext (adresa). Následující XML (6.2) by se nacházelo v elementu `capsule:capsules` na řádku 11 v 6.1 stejně tak jako definice managera a sandboxu v následujícím textu.

Zdrojový kód 6.2: Klient-server konfigurace v XML

```
1 <capsule:config id="capsuleConfig">
      <capsule:protocol value="rmi"/>
      <capsule:context-path value="capsuleService"/>
      <capsule:capsule-profile value="capsule.profile.remote"/>
5 </capsule:config>
```

6.3 Manager

Manager je hlavní ze tří komponent Java Capsules. Tato komponenta má na starosti správu všech souvisejících objektů. Jak ukazuje obrázek 4.2, každý klient a server jsou identifikováni unikátním jménem v rámci běhového prostředí Spring. Tyto identifikátory slouží pro odlišení v rámci manageru.

Stěžejní je implementace rozhraní `CapsuleManager`, které má několik základních implementací. API poskytuje metody pro:

- Spuštění serveru a tím procesu se sandboxem

- Zastavení serveru a tím procesu se sandboxem
- Vynucené zastavení serveru a tím procesu se sanboxem
- Test, zda server běží
- Restart serveru
- Získání jmen serverů, které manager spravuje
- Získání jmen klientů, které manager spravuje
- Získání instance serveru dle jména
- Získání instance klienta dle jména

6.3.1 Správa pomocí JMX

Klíčovou správa managera lze vystavit pomocí JMX. Získáme tak možnost administrátorského přístupu do běhu aplikace. Na rozhraní JMX je možné přistupovat pomocí řady programů, jak již bylo zmíněno v kapitole 3.1.3. Pro konfiguraci JMX není Java Capsules neposkytuje žádné rozhraní vyšší úrovně (XML namespace), a tak je nutné nastavení provést čistě pomocí komponent Springu. Důvodem je, že jednotlivé komponenty nelze zapouzdřit bez vedlejších efektů. Konfigurace je v příloze C.

6.4 Process manager

Process manager (třída `ProcessManager`) je základní implementací manageru. Třída naslouchá aplikačním událostem, na které reaguje a které také sama publikuje. Při startu aplikace detekuje všechny třídy reprezentující klienta nebo servery přítomné v aplikačním kontextu a zařadí si je do své struktury. A spustí svůj běh, na jehož začátku spouští servery a ti následně procesy sandboxu a klienty. Většinu ostatní funkcionality pouze deleguje daným objektům. Procesy umí také zastavit a restartovat. Při konci své činnosti všem odešle příkazy k ukončení.

Zdrojový kód 6.3: XML konfigurace: process manager

```
1 <capsule:manager/>
```

6.4.1 Vytvoření odděleného procesu

Pro dostatečné oddělení části aplikace, aby neohrožovala ostatní funkcionalitu, byl vybrán proces. Proto je nutné v rámci oddělení aplikace spustit proces, který obstará komunikační rozhraní a spustí v sobě samotnou oddělenou část.

Knihovny jazyka Java neposkytují pro práci s procesy tak dobré prostředky jako například pro práci s vlákny. Java ani knihovny třetích stran neposkytují potřebné prostředky pro komunikaci (například zasílání zpráv) ani synchronizaci (semafory) v rámci dvou procesů, dokonce ani test, zda proces běží.

Pro potřeby Java Capsule bylo nezbytné vytvořit abstrakci nad minimalistickým API, které Java poskytuje. Jedná se o implementace rozhraní `ProcessExecutor`, konkrétně o `SingleProcessExecutor` a jeho potomka `LoggingProcessExecutor`. Tyto třídy mají

na starosti spuštění Java procesu. Zajistí předání argumentů a *classpath* a spustí třídu, která obsahuje metodu `main()` na obsluhu odděleného procesu. Implementace poskytuje API pro předává příkazů do procesu. Logovací rozšíření také přenáší standardní výstup do rodičovského procesu.

6.5 Health-check manager

`HealthCheckManager` je rozšíření předcházejícího typu o jednu důležitou vlastnost a to zasílání dotazů, které nenesou žádná data a očekávají kladnou odpověď (ping). Mezi základní konfigurace patří:

- interval mezi dotazy (ping)
- počet opakování při neúspěchu, pro vyrovnání se s krátkodobými výpadky na síti

Další důležité parametry odvozuje od třídy klienta:

- timeout na odpověď
- timeout na vypnutí
- příznak, zda při neúspěchu restartovat automaticky

Proces ověření životaschopnosti je řízen stavovým automatem vyobrazeným na obrázku 6.1 (str. 33), který provede jedno ověření a svoji činnost končí. Nový automat je volán periodicky pro každého klienta, který se nalézá ve stejném aplikačním kontextu. Každý stav také publikuje aplikační událost o tom, co se s jakým klientem a potažmo serverem, ke kterému se klient připojuje, stalo. Na tyto události lze volitelně registrovat libovolné množství posluchačů a reagovat na ně dle libosti. Například na `FAIL` zaregistrovat posluchače, který odešle email administrátorovi.

Zdrojový kód 6.4: XML konfigurace: healt-check manager

```
1 <capsule:health-check-manager/>
```

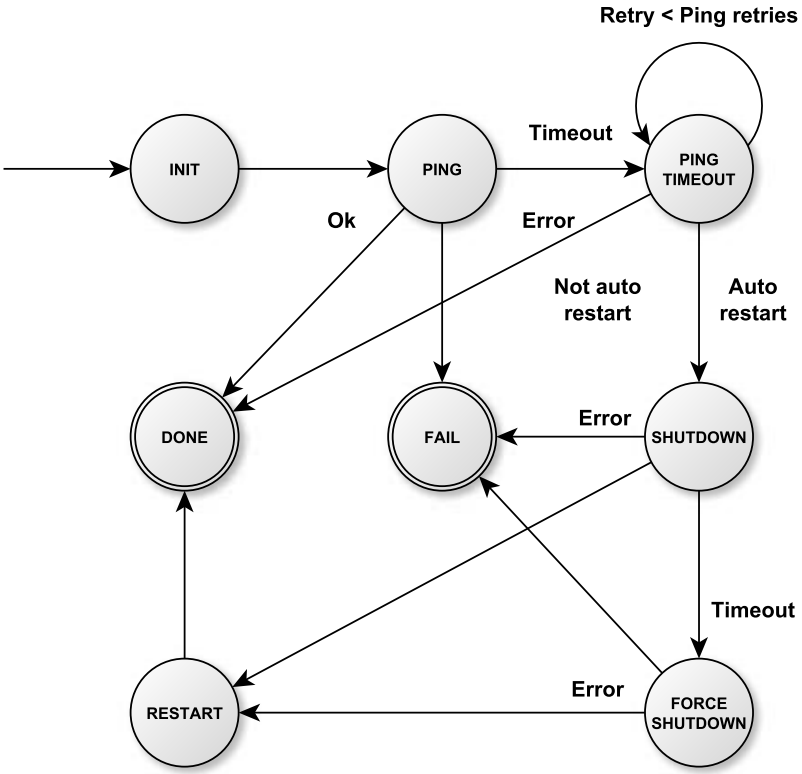
6.6 Distributed manager

Distribuovaný manager, který je postaven tak, aby implementoval požadavek na distribuci zdrojů z kapitoly 2.3, tedy případ, kdy bude cílem rozdělit aplikaci do několika procesů, které nemusí běžet na stejném fyzickém zařízení.

Na rozdíl od *clusteru* se nesnaží o zvýšení výpočetní síly, rychlosti a tím také spolehlivost duplikací celých serverů. Jde pouze o ochranu a oddělených částí aplikace. Také z pohledu architektury se nejedná o rovnocenné aplikace. Jedna je vždy nadřazena (master process) a řídí druhou (sandbox process).

Architektura klient-server, kterou implementují dvě z hlavních částí Java Capsules, splňuje předpoklady k naplnění tohoto cíle již ze své podstaty, jedná se jen o zadání správné URL. Problémy nastávají u správy, kde manager potřebuje běžet na stejném stroji jako server, aby ho měl pod kontrolou a mohl ho kdykoliv ukončit a restartovat.

Řešení spočívá v tom, že pro tento způsob existují dva druhy managerů, jak ukazuje 6.2 (str. 34). První vystavuje své rozhraní a druhý v master procesu se s ostatními po startu



Obrázek 6.1: Stavový automat - Health-check manager

spojí a zaregistruje si jména jejich serverů. Požadavky na instance, které nejsou přítomny lokálně, se pak předávají vzdáleným managerům. Unikátnost jmen je čistě v rukou uživatele.

Zdrojový kód 6.5: XML konfigurace: Distribute manager

```
<capsule:distribute-health-check-manager>
    <capsule:remote-manager name="remoteManager">
        <capsule:url value="url:port"/>
        <capsule:protocol value="rmi"/>
    </capsule:remote-manager>
</capsule:distribute-health-check-manager>
```

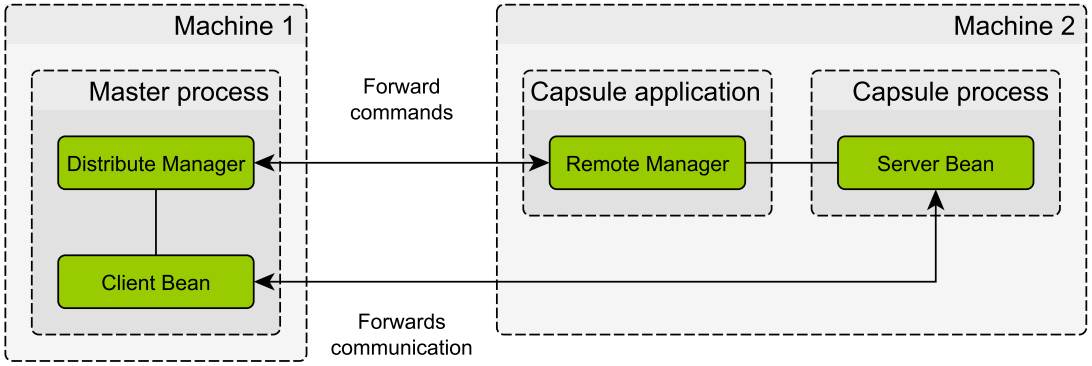
Zdrojový kód 6.6: XML konfigurace: Remote manager

```
<capsule:expose-manager name="remoteManager" port="port"/>
```

6.7 Spring beans

Aplikační rámec Spring pro svůj běh vytváří tzv. aplikační kontext, dále pouze kontext. V tomto kontextu jsou všechny instance uživatelských tříd pojmenovány (uživatelem nebo automaticky) a splňují tak podmínku na to, aby mohli být v Java terminologii označeny jako beany.

Spring beans je první a nejjemnější úroveň granularity oddělení části aplikace. V implementaci tohoto kontextu lze oddělit i volání jediné metody a to je pravděpodobně nejmenší



Obrázek 6.2: Oddělení procesů na platformě Java

smysluplně oddělitelný celek. Možnosti, jak a kolik volání instancí bean bude odděleno do samostatného procesu, jsou čistě v rukou uživatele a jeho konfigurace. Sandboxing bean je určen pro singleton beany.

6.7.1 Oddělení bean

V oddělení části aplikace na této úrovni granularity jsou potřeba dvě části, jak už se dá předpokládat, bude se jednat o klienta a server:

Beans je klientská část aplikace a vytváří komunikační stub a předává invokaci veškerých metod, které ho zavolají při svém běhu, do sandboxu. V konfiguraci je tato definice označena jako `beans`. Po obdržení odpovědi jsou zapsány změny do navrácených objektů.

Zdrojový kód 6.7: XML konfigurace: Beans

```
1 <capsule:beans id="capsuleBeans">
      <capsule:pointcut value="within(com.package..*)"/>
  </capsule:beans>
```

Tyto části stojí na konceptech AOP. Základem klientské části je, kromě podpůrných konfigurací, samotný klient a Advisor. Advisor se skládá z Pointcut a Advice (viz 5.2.2). Pomocí pointcut lze definovat libovolný výraz (AspectJ expression) na označení množiny balíčků, typů a metod, které Pointcutu odpovídají. Pokud se v aplikačním kontextu nacházejí odpovídající beany, jsou na ně aplikovány Aspekty, které se postarají o oddělení.

Základem konfigurace a vymezení oddělené části aplikace je tedy Pointcut. Pro oddělení nebo úpravu chování chování jsou definovány následující anotace:

- `@Capsule` — Jedná se o pomocnou anotaci na označení tříd, které jsou odděleny od stávající aplikace do nového procesu. Pokud není nastaven Pointcut výraz, je výchozí chování definováno na třídy, které obsahují tuto anotaci. Pokud uživatel využije vlastní AspectJ expression a zároveň tuto anotaci, musí podle toho definovat výraz přidáním podvýrazu s propojením pomocí logického součtu.
- `@NoCapsule` — Tato anotace způsobuje negaci předchozí definice oddělení. Volání třídy nebo metody s touto anotací je provedeno v rámci stejného procesu.
- `@CapsuleParam` — Poslední anotace slouží ke konfiguraci parametrů. Je možné tak označit třídy, metody nebo parametry, které jsou pouze vstupní anebo vstupně-

výstupní. Slouží pro zlepšení výkonnosti při zpětném odesílání a zapisování. Výchozí chování je zpětný zápis argumentů po dokončení invokace metody.

Základní konfigurace:

- `id` — Unikátní identifikátor beany
- `class` — Třída beany
- `config-name` — Identifikátor konfigurace
- `url` — Url ve tvaru `host:port`
- `pointcut/poincut-ref` — AspectJ výraz
- `ping-timeout` — Čas vypršení čekání na odpověď
- `connection-timeout` — Čas vypršení čekání na připojení po startu aplikace
- `shutdown-timeout` — Čas vypršení čekání na vypnutí aplikace
- `reconnect-interval` — Interval znovu-připojování k serverové části
- `restart-after-ping-timeout` — Příznak, zda provést automatický restart při nemožnosti kontaktovat sandbox

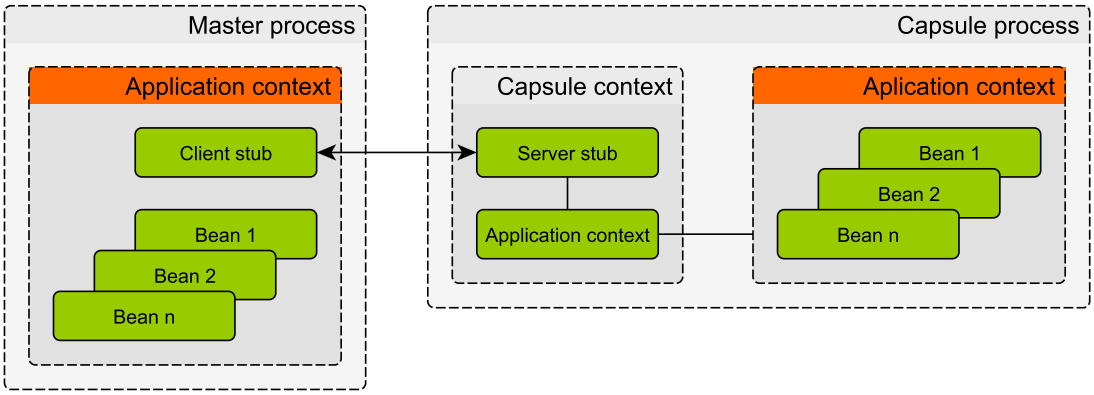
Beans Sandbox je serverová část aplikace a stará se o aplikační sandbox, přijímá požadavky z klientských bean (kontext invokace) a provádí je v rámci svého procesu. Má jeden povinný parametr: definici kontextů, které budou v rámci sanboxu, a řadu volitelných, jako: prefix logu, název java procesu, port, argumenty JVM, jméno společné konfigurace a příznak, zda při chybě instanci automaticky restartovat (ten ma samozřejmě smysl pouze v případě Health-check Manageru). Strukturu kontextů ukazuje obrázek 6.3 (str. 36). Vyznačené kontexty si sobě odpovídají.

Zdrojový kód 6.8: XML konfigurace: Beans Sandbox

```
1 <capsule:beans-sandbox id="capsuleSandbox">
      <capsule:spring-contexts>
        config.xml
      </capsule:spring-contexts>
5 </capsule:beans-sandbox>
```

Základní konfigurace:

- `id, class, config-name` — Stejná sémantika jako u klienta
- `auto-start` — Příznak, který definuje, zda se proces sandboxu spustí automaticky při startu
- `port` — Port komunikace
- `jvm-arguments` — Argumenty JVM
- `output-prefix, error-prefix` — Prefix logu ze sandbox proces
- `java` — Program java dostupný pro spuštění v systému
- `spring-contexts` — Kontexty spuštěné v sandboxu



Obrázek 6.3: Oddělení bean — kontexty

6.7.2 Vyhledávání a používání prostředků odděleného procesu

Pokud budeme chtít přenášet invokaci metody objektu do jiného procesu, narazíme na problém, kterou instanci v kontextu v druhém procesu invokovat, neboli které instance si ve dvou stejně vytvořených kontextech odpovídají. Jelikož se pohybujeme v aplikačním kontextu, kde jsou instance beany, je každá instance pojmenována. To celý problém zjednodušuje. Předání odkazu na objekt, nad kterým se má provést volání metody, bude tedy pomocí jména beany.

Aplikační kontext poskytuje řadu metod na získání instance dle jména nebo typu, ale neexistuje žádná metoda na získání jména. Tento problém je řešen tak, že se při startu aplikace všechny instance v kontextu zaregistrují do hashovací tabulky s porovnáním na referenci, kde je klíčem obalovací objekt s referencí. Poté je možné z předané reference zpětně získat jméno. Naplnění hashovací tabulky se děje pouze při startu, proto je jeho složitost z pohledy běhu nepodstatná. Samotné vyhledání se děje s konstantní složitostí hashovací tabulky.

6.7.3 Popis práce oddělení bean

Nakonec této kapitoly je uvedena sekvence kroků, která se provede při využití oddělení aplikací ve výchozím nastavení na úrovni Spring bean.

1. V době inicializace aplikace
   (a) Nastavení automatického vytváření proxy nad objekty
   (b) Vytváření proxy pro komunikaci se sandboxem nad objekty vymezenými Pointcut výrazem
   (c) Inicializace inverzního mapování referencí na jména bean
2. Po zinicalizování existující aplikace
   (a) Spuštění druhého procesu
   (b) Čekání na spuštění
3. Běh aplikace

(a) Invokace metody, která je součástí oddělení aplikace

(b) Nalezení jména instance

(c) Serializace parametrů a metody

(d) Odeslání požadavku na druhý proces

(e) Provedení v druhém procesu

(f) Zapsání stavu po invokaci do parametrů metody (může se stát, že metoda modifikuje vnitřní data objektu)

(g) Navrácení hodnoty

6.8 Webové aplikace

Na úvod této kapitoly bude velice stručně vysvětleno, co jsou webové aplikace v Javě a na jakých konceptech a technologií stojí a ukážeme si, jak se k Java standardům staví aplikační rámec Spring.

6.8.1 Java Enterprise Edition

Java EE je edice Java, která pokrývá podporu na tvorbu webových a podnikových aplikací. Takové aplikace jsou zpravidla serverové, využívají přístup do databáze a mohou svůj výstup posílat klientskému prohlížeči webových stránek. Aplikace potřebují pro svůj běh servletový kontejner nebo aplikační server, což jsou rozsáhlé programy, které implementují API Java EE a aplikace se do nich nasazují.

Aplikace můžeme rozdělit na dvě skupiny: prezentačně orientované a servisně orientované [2]. Prezentačně orientovaná aplikace provádí svůj výstup na webové stránky pomocí HTML aj. Aplikace obsluhuje požadavky a zasílá odpovědi (request-response). Mezi technologie prezentační vrstvy patří například Java Server Pages nebo Java Server Faces. Naproti tomu servisně orientované aplikace vystavují své rozhraní pomocí webových služeb, se kterými komunikují ostatní klienti. Může se jednat o JAX-WS (implementace SOA) nebo JAX-RS (referenční implementace REST).

Základem každé takové aplikace jsou Servlety. Servlet je třída implementující rozhraní Servletu. Ta pak umí obsluhovat příchozí síťové, často HTTP, požadavky.

6.8.2 Spring Web MVC

Spring Web MVC je modul aplikačního rámce Spring, který se zabývá podporou tvorby webových aplikací za pomoci architektonického vzoru Model-View-Controller (MVC), který je „request-driven“, tedy řízen příchozími požadavky, na základě kterých jeden centrální Servlet obsluhuje požadavky [9]. V případě Springu se jedná o `DispatcherServlet`.

Tato práce není primárně zaměřena na tvorbu webových aplikací, proto pro další informace doporučuji jiné zdroje jako: [2] a [9].

6.8.3 Oddělení webových aplikací

Oddělování celých webových aplikací je znatelně komplexnější záležitost, než tomu bylo u samotného Springu a jeho bean. Kromě prostého vytvoření procesu (viz. kapitola 6.4.1) je potřeba řešit několik dalších věcí:

1. Proces sandboxu potřebuje pro svůj běh servletový kontejner
2. Nasazení webových aplikací do sandboxu
3. V případě RMI je potřeba dvou portů

Oddělení webové aplikace poskytuje 2 různé funkcionality označené jako:

1. **Servlet** — Jedná se zachytávání požadavků na Servlet hlavní aplikace a následný přenos požadavku do sanboxu, kde je proveden, a zpropaguje se odpověď zpět.
2. **Bean** — Je podpora funkcionality Spring beans, která již byla zmíněna, v prostředí webových aplikací.

Stejně jako předtím je aplikace rozdělena na klientskou (hlavní proces) a serverovou část (oddělený proces).

Web App je označení pro klientskou část a vytváří komunikační stub pro přeposílání servletových požadavků na sandbox. Pro využití této funkcionality jsou nezbytné konfigurace na dvou místech.

1. Spring xml konfigurace

Zdrojový kód 6.9: XML konfigurace: Web App

```
1 <capsule:web-app id="webAppClient"/>
```

2. web.xml deskriptor

Zdrojový kód 6.10: XML konfigurace: Web App

```
1 <servlet>
      ...
      <servlet-class>
        eu.ibacz.capsules.webapps.servlet.CapsuleDispatcherServlet
5     </servlet-class>
      ...
  </servlet>
```

Kompletní konfigurace je součástí přílohy C. Z konfigurace je vidět, že odchytávání a přeposílání požadavků na existující aplikaci je provedeno pomocí Servletu typu `CapsuleDispatcherServlet`, která je hlavní částí Spring MVC.

Mezi hlavní konfigurace patří:

- `profile` — profil sandbox procesu
- `capsule-excludes` — Regulární výrazy požadavků, které budou obslouženy lokálně, tedy nebudou přeposílány na sandbox. Vhodné pro použití na CSS soubory.
- `capsule-includes` — Regulární výrazy požadavků, které budou obslouženy v sandboxu. Pokud je nezadáno include, je každý požadavek a je přeposláno vše, co je v include a není v exclude.
- `restful` — Příznak přepne komunikaci do režimu restful. Je možné využít všechny prostředky REST, které poskytuje Spring (Https, vlastní konverze zpráv, aj.).

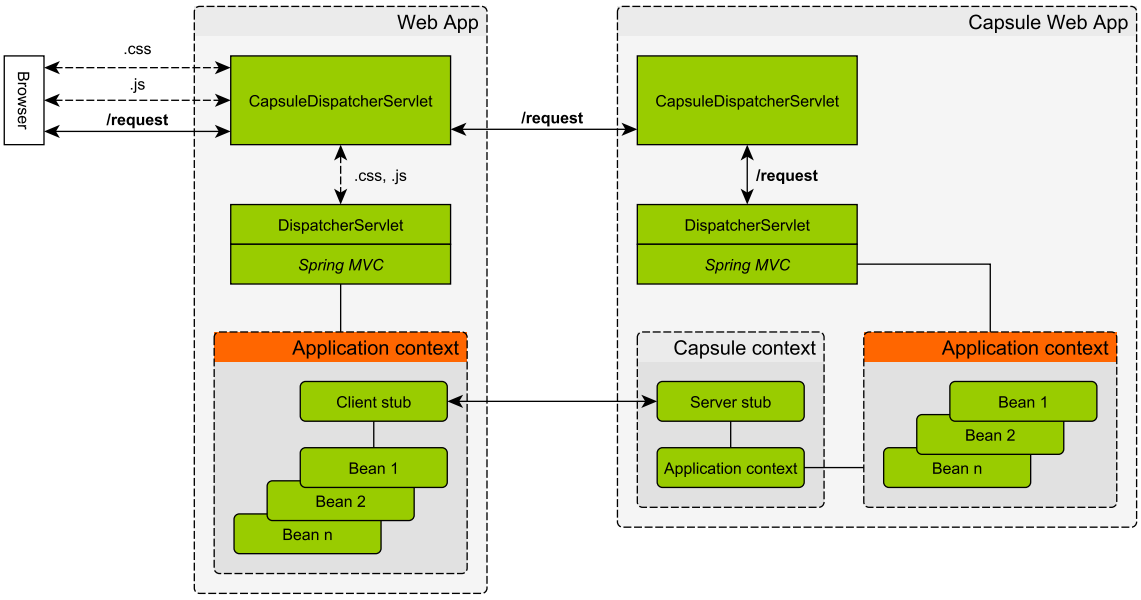
- `content-type` — Typ obsahu pro komunikaci mezi Servlety při přeposílání příchozích požadavků. Konfigurovat má smysl pouze ve spojení s restful.
- Další konfigurace jsou totožné s předchozí úrovní granularity

Web App Sandbox označuje komponentu, která je abstrakcí sandbox procesu v rámci kterého je spuštěna webová aplikace. Oddělená aplikace se automaticky odvodí z aktuální, ve které je komponenta spuštěna. Součástí konfigurace sandboxu webových aplikací jsou vhodně nakonfigurované knihovny pro Embedded Tomcat, ty nelze získat z existující aplikace. Sada knihoven v referenční verzi je součástí zdrojového kódu.

Zdrojový kód 6.11: XML konfigurace: Web App Sandbox

```
1 <capsule:web-app-sandbox id="webAppServer"/>
```

Obrázek 6.4 ukazuje architekturu implementace úrovně oddělování aplikací. Z obrázku je vidět, že je velmi podobný předchozí úrovni oddělování. Je zde naznačeno zpracování požadavků */request* s konfigurací, kde je statický obsah stránek rovnou obsluhován hlavním procesem. Webové aplikace poskytují podporu i pro využití sandboxingu bean. Pokud uživatel chce nakonfigurovat tuto kombinaci, stačí definovat jako serverovou část webovou aplikaci a jako klienta Spring beans.

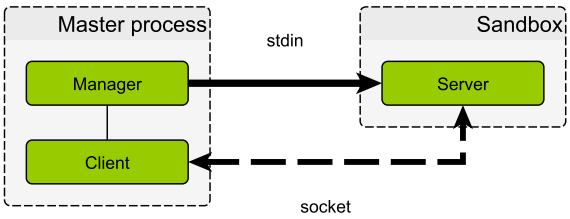


Obrázek 6.4: Oddělení webových aplikací — architektura

6.9 Meziprocesová komunikace

Komunikaci mezi procesy je možné rozdělit na dva hlavní typy. První skupinou je servisní rozhraní, pomocí kterého komunikuje manager s procesem sandboxu. Tato komunikace je jednosměrná a manager tak zasílá příkazy pomocí standardního vstupu procesu. Druhým je obousměrná komunikace, ve které probíhá volání a odpověď na daný požadavek, který

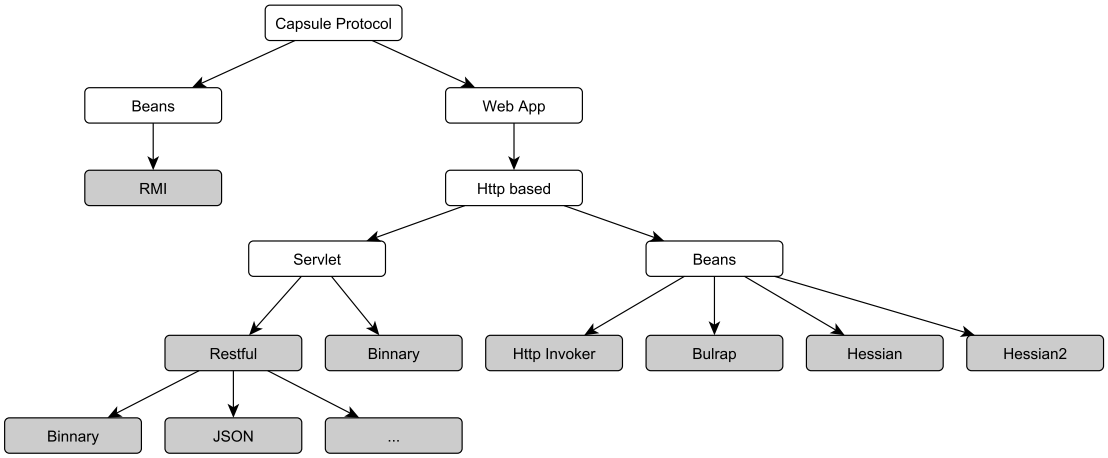
proces obsluhuje. Tímto způsobem se posílají požadavky oddělené aplikace. Tato komunikace je realizována pomocí soketů a implementuje model klient-server, jeden proces se vždy dotazuje a druhý odpovídá. V případě implementace této práce je klientem původní aplikace (hlavní proces) a serverem je proces sandboxované aplikace. Jak znázorňuje obrázek 6.5.



Obrázek 6.5: Meziprocesová komunikace

Procesy musí komunikovat dohodnutým způsobem, stejným protokolem. Otázkou zůstává jakým protokolem. Hlavním kritériem pro výběr protokolu bude odezva. V tomto ohledu je aplikace variabilní a je možné si protokoly volit. Porovnání protokolů a jejich výkonnost bude součástí kapitoly 7. Podporovaných protokolů je několik, jak ukazuje obrázek 6.6. Nutno však podotknout, že u restful implementace a Spring bean jsou možnosti neomezené a lze si zvolit vlastní protokol pro komunikaci. Stejně tak pro beany lze použít vlastní implementaci rozhraní.

Použité protokoly se liší, dle použití dané úrovně granularity sandboxingu. Pro celé webové aplikace je možné využít protokoly z podstromu „**Servlet**“, protože se jedná o servletově založenou komunikaci. Pro úroveň instancí bean je možné využívat protokoly z podstromů „**Beans**“. Možnosti se pak liší podle toho v jakém kontextu je Capsule sandbox využit. Samotný sandboxing bean podporuje pouze RMI a není tak závislý na servletové implementaci. Naproti tomu ve webových aplikacích je možné využít protokoly, které se opírají o Servlet API a HTTP.



Obrázek 6.6: Dostupné komunikační protokoly

- RMI — je zkratka pro Remote Method Invocation. Jedná se o technologii, která je sou-

částí Java od verze 1.1. Implementuje architekturu klient-server doplněnou o registr služeb. Jedná se o protokol specifický pouze pro Java a nepotřebuje další knihovny. Je součástí knihoven jazyka Java.

- Http Invoker — pod označením Http Invoker se skrývá implementace, která je součástí aplikačního rámce Spring[2]. Jedná se o doporučovaný způsob Java-to-Java komunikace.
- Hessian — je binární protokol nad http pro výměnu zpráv. Na rozdíl od předchozích protokolů se nemusí nutně jednat o Java-to-Java komunikaci. Je dostupný v knihovnách od společnosti Caucho.
- Hessian2 — jde o implementaci předchozího v druhé verzi.
- Bulrap — je protokol také od Caucho. Jedná se o protokol postavený nad http. Zprávy jsou serializovány do XML zápisu, na rozdíl od binárního.
- Http-binnary — poslední dva protokoly nebo spíše styly se pojí k servletové komunikaci oddělených procesů, jsou pomocí nich vyměňovány požadavky a odpovědi servletů v dvou procesech. Tato implementace je čistě nad standardními knihovnami Java a je postavena na binárních streamech, které si vyměňují objekty.
- Http-resful — implementace restful výměny zpráv umožňuje využít veškerých prostředků, které nabízí REST implementace[3] v aplikačním rámci Spring. Uživatel si volí způsoby serializace a deserializace objektů i typ obsahu. Implementace je možné konfiguračně měnit nebo doplnit o nové. V této variantě lze http obohatit například i o SSL.

Vytvořené procesy je možné synchronizovat tak, aby klient nebo klienti věděli, že server nebo servery jsou připraveni. Synchronizace je nepovinná a lze ji vypnout. Hlavní proces pak nečeká na inicializaci sandboxu.

6.10 Injekce závislostí

Pokud nahlédneme do implementace samotného Spring frameworku můžeme si všimnou zajímavé věci. Spring nevytváří žádný aplikační kontext a zároveň ani nespoléhá na existenci jiných závislostí. Jinými slovy injekce závislostí nepoužívá autowire[4]. Naopak třídy spoléhají na životní cyklus Spring komponenty. Z toho plyne, že pokud vytváříme aplikační rámec nebo knihovnu, musíme řešení implementovat jiným způsobem než koncovou aplikaci.

Java Capsules pro svůj běh potřebuje některé závislosti. Zároveň není vhodné využívat autowire. Řešení závislostí spočívá v použití rozhraní Aware. Aware rozhraní je rozhraní, které obsahuje nastavení požadované komponenty do dané třídy. Druhým krokem je *bean postprocesor*, který každé třídě, která implementuje dané rozhraní nastaví, požadované objekty.

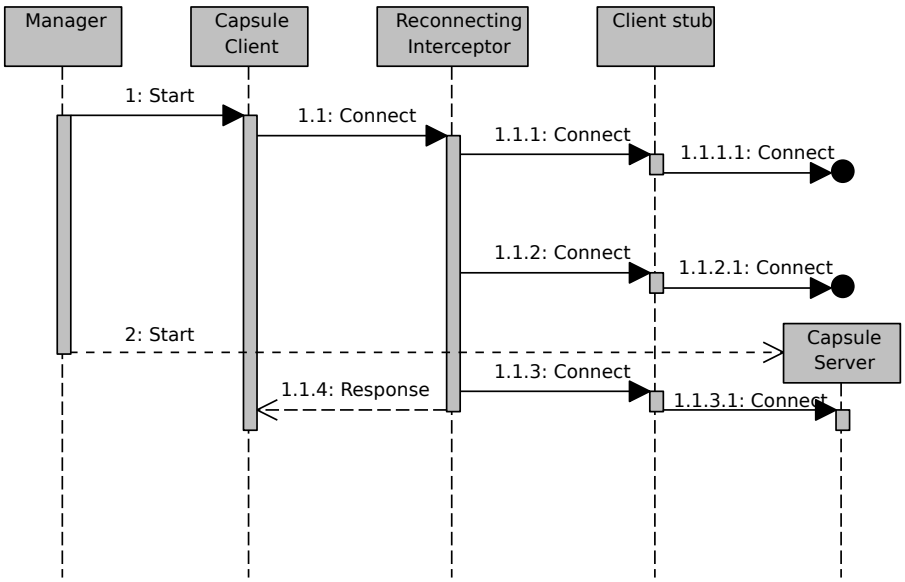
Pokud definujeme komponentu Java Capsules, tak sama o sobě neprovádí funkcionalitu. Komponenty „ožívají“ ve chvíli, kdy je objeví manager. Manager vytvoří požadované závislosti a zároveň se postará o jejich distribuci do ostatních komponent. Dále se postará o spuštění sandboxů a jejich spojení s existující aplikací.

[2] balík `org.springframework.remoting.httpinvoker`
[3] balík `org.springframework.web.client.RestTemplate`
[4] Automatická injekce závislostí

6.11 Vzdálené připojení

V existující aplikaci je klientská část vytvořena před sandboxem. Proto v době vytvoření nelze spojit obě části, jak to některé z protokolů dělají a vyžadují. Zároveň se některé protokoly (jako RMI) neumí vypořádat s rozpojením komunikace a znovu připojením. Tuto situaci bylo potřeba vyřešit.

Jelikož Java Capsules poskytuje rozmanité množství protokolů, nelze tento problém řešit procedurálně. Znamenalo by to totiž implementace této funkcionality do rozšíření každého z protokolů a tím i zbytečný nárůst kódu a přizpůsobení vlastním způsobem. Proto jsem použil pro navázání spojení proxy objekt, tedy prostředník, který se o spojení postará. Pro spojení se zavolá jedna metoda a ta se opakuje v definovaných intervalech do vypršení časovače nebo připojení. Navázání spojení je vyznačeno na obrázku 6.7.



Obrázek 6.7: Navázání spojení

Kapitola 7

Dosažené výsledky

7.1 Testování výkonnosti

Testování výkonnosti (performance testing) Java aplikací má svá specifika daná Java Virtual Machine. Jedná se o *Garbage collecting* a *Just in time compilation*, které přinášejí do aplikace zpoždění v neočekávanou chvíli a do měření zpoždění. Během měření tedy potřebujeme dodržet následující podmínky:

- Nesmí nastat (major) Garbage Collection — toho docílíme tak, že aplikace má po celou dobu běhu přiřazen dostatek paměti a vláken, se kterými zátěžové testy provádí. Jde nám hlavně o eliminaci major GC, které je doprovázeno zastavením JVM.
- Aplikace musí minimalizovat provádění JIT — JVM využívá statistiky, na základě kterých provádí JIT kompilace a optimalizace. Pokud kód opakovaně provádí, bude zkompilován do nativního. Pro přesnější měření provede akce opakovaně před spuštěním měření.

Pro testování byl použit testovací rámec JUnit a nástroj Gatling[1]. Testy probíhaly na netriviálních aplikacích, případech a objektech. Testovanou aplikací bylo především *Simple CMS*[2]. Aplikace Simple CMS představuje menší webovou aplikaci s několika stránkami. Každé volání zahrnuje nejméně dva dotazy do databáze a přístup pomocí transakcí, architekturu MVC a s tím spojené úlohy, nakonec samotné vykreslení stránky. Změřené hodnoty jsou porovnány jako výsledky s Java Capsules a čisté aplikace bez Java Capsules.

7.1.1 Typy měření

Java Capsules byla testována v režimech a v podporovaných protokolech, dle obrázku 6.6 (str. 40).

1. Režim — Java Beans
   (a) Samotné beans (RMI)
   (b) Beans ve webovém prostředí (Http Invoker, Bulrap, Hessian, Hessian2)
2. Režim — Webapp (Binnary, Restful – binnary, Restful – JSON)

[1] http://gatling.io/
[2] http://github.com/mjanys/Simple-CMS

7.1.2 Odezva

Měření odezvy probíhalo v sekvenci stovek opakování a poté byla vypočítána průměrná hodnota. Výsledky jsou v milisekundách (ms). Tabulka 7.1 ukazuje tři sady testů Java Capsules:

(1) Oddělení Java Beans ve Springové aplikaci bez servletového kontejneru

(2) Oddělení Java Beans ve Webové aplikaci se servletovým kontejnerem

(3) Oddělení celé webové aplikace na úrovni HttpServletRequest

Každá sekce obsahuje řádek bez protokolu, který udává odezvu bez využití Java Capsules. Srovnávat lze pouze výsledky v rámci jednoho typu použití v jedné sekce tabulky. Zavedení Java Capsules přináší do aplikace pochopitelnou režii při volání odděleného procesu. Z profilování běhu v Java Capsules kódu bylo zřejmé, že aplikace tráví 90–95% času právě síťovou komunikací. Výsledky odezvy zůstaly ve stejném řádu a například pro oddělení celé webové aplikace (jeden z režimů oddělení, aplikace komunikují na úrovni http požadavků) byla odezva zvýšena z 49 ms (aplikace bez Java Capsules) na 67 ms. Pro oddělení na úrovni objektů (bean, komunikace na úrovni invokaci metod) pak z 49 ms na 87 ms. Tyto části mezi sebou nelze porovnávat, protože jsou zde vyměňována úplně jiná data.

Z výsledků můžeme říci, který protokol je pro modelový případ nejrychlejším. Dále je vidět, jakou změnu odezvy přináší zavedení Java Capsules. Jelikož se pohybujeme ve stejném řádu, považuji výsledky za uspokojivé v rámci webových aplikací. V samostatném testu bean jsou výsledky pod 1ms.

		Protokol	Čas jednoho volání v ms
(1)	Java Beans	—	0,002
	Java Capsules + Java Beans	RMI	0,546
(2)	WebApp + Java Beans	—	49
	Java Capsules + WebApp + Java Beans	Http Invoker	95
		Burlap	108
		Hessian	**87**
		Hessian2	97
(3)	WebApp	—	49
	Java Capsules + WebApp	**Binnary**	**67**
		Restful/binnary	79
		Restful/json	84

Tabulka 7.1: Odezva — Java Capsules

7.1.3 Propustnost

Měření propustnosti proběhlo zatížením aplikace stovkami paralelních požadavků s dostatkem paměti a vysokých limitů (počet vláken v Tomcat a JVM). Výsledky jsou v počtu požadavků za sekundu (req/s). Tabulka 7.2 opět ukazuje výsledky propustnosti aplikace pod velkou zátěží. Pomocí nástroje Gatling bylo vygenerováno 500 uživatelů, kteří lineárně přicházeli po dobu 15 sekund, v systému se zdrželi 8 sec při prohlížení různých stránek.

Příčina dopadu snížení propustnosti u bean ve webovém prostředí je větší, protože zde se na rozdíl od sandboxu webové aplikace neprovádělo pouze jedno volání do sandboxu. Na měření je pozitivní, že aplikace i sandbox ve stres testu obstály a všechny požadavky obsloužily i za cenu zhoršení propustnosti. Testů proběhlo více a tyto výsledky byly vybrány jako demonstrace. Na přiloženém mediu jsou dostupné kompletní reporty i s grafy (viz. příloha F).

Prostorová závislost řešení je řádově stejná jako původní aplikace. Proto při zavedení Java Capsules bude potřeba dvojnásobek paměti pro oddělenou část.

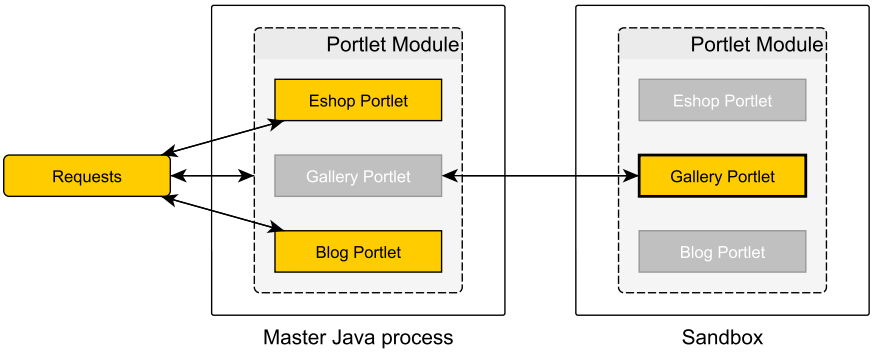
		Protokol	Propustnost v req/s
(2)	WebApp + Java Beans	—	101,0
	Java Capsules + WebApp + Java Beans	**Http Invoker**	**31,2**
		Burlap	29,9
		Hessian	29,7
		Hessian2	30,7
(3)	WebApp	—	101,0
	Java Capsules + WebApp	**Binnary**	**58,6**
		Restful/binnary	54,9
		Restful/json	52,6

Tabulka 7.2: Propustnost — Java Capsules

7.2 Možná rozšíření

7.2.1 Portletové aplikace

Portletové aplikace jsou nadmnožinou webových aplikací. Jedna aplikace může obsahovat několik samostatných celků jménem portlety. Definice sandboxu bude obsahovat množinu portletů, které budou chráněny sandboxem, jak znázorňuje obrázek 7.1.



Obrázek 7.1: Sandbox/Capsule na portletových aplikacích.

7.2.2 Integrace dalších technologii

Mezi dalšími rozšířeními může být například integrace s prezentační vrstvou JSF (Java Server Faces), která je také servletově založená.

Kromě prezentačních technologii by Java Capsules mohla obsahovat volbu servletových kontejnerů. Mezi servletové kontejnery, které by bylo možné doimplementovat jako aplikační sandbox, patří Jetty nebo Spring Boot opět s Tomcat nebo Jetty.

7.2.3 Nová funkcionalita

Nápad na další funkcionalitu by mohl představovat možnost jistého rozkladu zátěže mezi sandboxy. To by znamenalo, že hlavní aplikace může dotazovat více různých sandboxů. Opačný případ je podporován již teď (1 sandbox, více klientů). V podobném režimu by mohlo být potencionálně praktičtější rozšíření a to možnost vyměnit v případě výpadku sandboxu druhý, který by aplikace měla k dispozici a snížit tak dobu výpadku sandboxu.

Kapitola 8

Závěr

Práce se věnovala problému stability na platformě Java. Byly vyloženy varianty přístupu k řešení stability, a to zejména prevence pomocí konfigurace JVM a představeny nástroje pro analýzu vzniklých problémů.

Jako jedno z řešení problémů se stabilitou bylo představeno oddělování procesů (Sandboxing). Pád v odděleném procesu nebude mít za následek pád celé aplikace a následky jeho chyb se tedy nebudou šířit do zbytku systému. Tato technika byla následně implementována. Zavedením Sandboxing do existující aplikace přináší zvýšení odolnosti a vyšší stabilitu aplikace. Implementovaný rámec, který Sandoxing zajišťuje, byl nazván rámce Java Capsules. Aplikaci lze ručně či automaticky spravovat po definovaných částech. Stabilita aplikace jako celku se tedy zvýší, protože jedna chyba nezpůsobí výpadek celého systému. V konfiguraci lze definovat libovolné uskupení objektů, tříd, balíčků nebo typů a jejich provádění přenášet do volání pod sandboxem. V případě, že tato volání způsobí chybu, správa sandboxu se o tom dozví a může tuto instanci sandboxu restartovat. Po startu je obnovena původní komunikace a systém funguje jako před chybou.

Na příkladech, které jsou k dispozici na přiloženém mediu lze pozorovat nejčastější chyby Java aplikací, jako je vyčerpání paměti, vyčerpání počtu vláken nebo ukončení aplikace (v reálném případě např. systémem). Jakmile je automaticky detekován výpadek, aplikace je zotavena restartem sandboxu. Ve své praxi jsem se již setkal se všemi druhy chyb, které byly v textu zmíněny. Příkladem mohou být případy, kdy jedna služba na obsluhu zpráv obsahovala těžko odhalitelnou race condition, která znemožnila funkčnost služby. Dále mohu jmenovat situaci, kdy marketingová kampaň pravidelně přetížila systém natolik, že nemohl dále pracovat. V obou případech pak bylo nutné celý systém restartovat. Start takového systému pak znamenal nedostupnost okolo jedné hodiny ve špičce. Oba tyto případy by bylo možné řešit pomocí Java Capsules a vyhnout se tak odstávce celého systému.

Řešení je v závěru vystaveno zátěžovým testům, které obslouží všechny požadavky pouze se sníženou odezvou a propustností. Nenastal tedy případ, že by aplikace přestala odpovídat. Dosažené výsledky (kapitola 7) pro webové aplikace se pohybují ve stejném řádu odezvy i propustnosti. Výsledky pro beany mají vyšší rozdíl, ale odezva do 1 milisekundy je únosnou hodnotou v případě aplikace pro koncového uživatele. Odezva systému a propustnost se sníží hlavně v důsledku režie, serializace objektů a poslání pomocí socketu.

Java Capsules je aplikační rámec, který uživateli poskytne prostředky pro zvýšení stability a pro tento účel si jistě najde místo v problémových částech existujících řešení. Usuzuji tak i z ohlasu, který byl v komunitě po představení řešení týkajících se stejného tématu od firmy Liferay.

Literatura

[1] Garbage-First Collector. [online]. 2013 [cit. 2014-10-19], Oracle.
URL http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html

[2] The Java EE 6 Tutorial. [online]. 2013 [cit. 2015-02-26], Oracle.
URL http://docs.oracle.com/javaee/6/tutorial/doc

[3] Java Garbage Collection Basics. [online]. [cit. 2014-10-07], Oracle.
URL http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[4] *Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning*, kapitola Available Collectors. [online]. 2013 [cit. 2014-10-19].
URL http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

[5] Memory Management in the Java HotSpot Virtual Machine. Technická zpráva, Sun microsystems, 04, [online]. 2006 [cit. 2014-10-19], Oracle.
URL http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

[6] Sandboxing Portlets to Ensure Portal Resiliency. [online]. 2014 [cit. 2014-11-30], Liferay.
URL https://www.liferay.com/documentation/liferay-portal/6.2/user-guide/-/ai/sandboxing-portlets-to-ensure-portal-re-liferay-portal-6-2-user-guide-18-en

[7] *Spring Framework Reference Documentation*, kapitola Introduction to the Spring Framework. [online]. 2014 [cit. 2015-02-15], Pivotal Software.
URL http://docs.spring.io/spring/docs/current/spring-framework-reference/html

[8] *Spring Framework Reference Documentation*, kapitola Aspect Oriented Programming with Spring. [online]. 2014 [cit. 2015-02-15], Pivotal Software.
URL http://docs.spring.io/spring/docs/current/spring-framework-reference/html

[9] *Spring Framework Reference Documentation*, kapitola Web MVC framework. [online]. 2014 [cit. 2015-02-15], Pivotal Software.
URL http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[10] Understanding Java Garbage Collection. [online]. 2013 [cit. 2014-10-05].
URL http://www.azulsystems.com/sites/default/files/images/Understanding_Java_Garbage_Collection_v3.pdf

[11] Goetz, B.: Java Garbage Collection Basics. [online]. 2006 [cit. 2014-10-08], IBM.
URL http://www.ibm.com/developerworks/library/j-jtp09196

[12] Janssen, C.: Sandboxing. [online]. 2010 – 2014 [cit. 2014-10-15].
URL http://www.techopedia.com/definition/25266/sandboxing

[13] Lee, S.: Understanding Java Garbage Collection. [online]. 2012 [cit. 2014-10-08].
URL http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection

[14] Müller, A.: Benchmarking G1 and other Java 7 Garbage Collectors. [online]. 2013 [cit. 2014-10-20].
URL http://blog.mgm-tp.com/2013/12/benchmarking-g1-and-other-java-7-garbage-collectors

[15] Preiss, B. R.; Eng, P.: *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, kapitola Mark-and-Sweep Garbage Collection. [online]. 1999 [cit. 2014-10-08].
URL http://www.brpreiss.com/books/opus5

[16] Učeň, P.: *Metriky v informatice: jak objektivně zjistit přínosy informačního systému.* Grada, 2001, ISBN 80-247-0080-8.

Příloha A

Slovík pojmů

A.1 Seznam zkratek

AOP Aspect Oriented Programing. 26, 27

CMS Concurrent Mark Sweep. 15

ETL Extract, Transform and Load. 26

GC Garbage Collection. 9, 10, 12, 15, 16

GCing Garbage Collecting. 10

IoC Inversion of Control. 26

IPC Inter-Process Communication. 20

JAX-RS Java API for RESTful Web Services. 7

JAX-WS Java API for XML Web Services. 7

JDK Java Development Kit. 11

JMS Java Messaging Services. 7, 13

JMX Java Management Extensions. 12, 13

JNDI Java Naming and Directory Interface. 19

JVM Java Virtual Machine. 1, 10–12, 14, 15, 19

PID Process identifier. 11

RMI Remote Method Invocation. 13, 19

RPC Remote Procedure Call. 19

SLA Service Level Agreement. 7

A.2 Seznam pojmů

Heap Paměťový prostor, halda. 15

Java Capsules Pojmenování projektu realizovaného v rámci této práce. 20

Java Enterprise Edition Platforma pro vývoj podnikových aplikací. 6, 20, 26

Java Virtual Machine Běhové prostředí každého Java programu. 6

Memory Leak Únik paměti, tato paměť se stává pro aplikaci nedostupnou. 7, 10, 16

Sandbox Sandbox. 8

Sandboxing Sandboxing. 4, 8

Spring Soubor aplikačních rámců pro tvorbu podnikových aplikací. 20

Příloha B

Kompatibilita verzí aplikace

V následující tabulce jsou uvedeny referenční verze použitých softwarových produktů. Většina knihoven a aplikačních rámce jsou dostupné v Maven Central Repository[1]. V případě, že tomu tak není, je uveden zdroj jako odkaz, kde je možné software nalézt.

Referenční verze představují verze, se kterými byl produkt testován a vyvíjen. Podporované pak verze, které mají kompatibilní API a je možné je využít.

Referenční verze	
Spring Framework	3.2.9.RELEASE
AspectJ	1.7.4
CGLib	2.2
Log4j	1.2.17
Hessian	4.0.38
Tomcat	7.0.55
Embedded Tomcat	7.0.55[2]

[1] http://search.maven.org
[2] http://tomcat.apache.org/download-70.cgi

Příloha C

XML konfigurace

V této části je uvedena plná konfigurace jednotlivých komponent. Každý element jehož atribut je `value` čili `<element value="value"/>`, lze psát ve tvaru `<element>value</element>`.

C.1 Process Manager

Zdrojový kód C.1: XML konfigurace: Process Manager

```
<capsule:manager
         id="capsule.manager.bean"
         class="eu.ibacz.capsules.framework.management.
         ProcessManager">
</capsule:manager>
```

C.2 Health-check Manager

Zdrojový kód C.2: XML konfigurace: Healt-check Manager

```
<capsule:health-check-manager
         id="capsule.manager.bean"
         class="eu.ibacz.capsules.framework.management.healthcheck.
                HealthCheckManager"
         pool-size="20">

    <capsule:ping-interval value="60"/>
    <capsule:ping-retries value="3"/>

</capsule:health-check-manager>
```

C.3 Distribute manager

Zdrojový kód C.3: XML konfigurace: Distribute Manager

```
<capsule:distribute-health-check-manager>
    <capsule:remote-manager name="remoteManager">
        <capsule:url value="url:port"/>
        <capsule:protocol value="rmi"/>
    </capsule:remote-manager>
</capsule:distribute-health-check-manager>
```

Zdrojový kód C.4: XML konfigurace: Remote Manager

```
<capsule:expose-manager name="remoteManager" port="port"/>
```

C.4 Beans

Zdrojový kód C.5: XML konfigurace: Beans

```
<capsule:beans
         id="capsuleClient"
        class="eu.ibacz.capsules.framework.bean.BeanClient">

    <capsule:config-name value="capsuleConfig"/>
    <capsule:url value="localhost:1099"/>
    <capsule:pointcut value="@annotation(Capsule)"/>
    <capsule:ping-timeout value="1000"/><!--1000ms-->
    <capsule:connect-timeout value="5000"/><!--5000ms-->
    <capsule:shutdown-timeout value="5000"/><!--5000ms-->
    <capsule:reconnect-interval value="250"/><!--250ms-->
    <capsule:restart-after-ping-timeout value="true"/>
</capsule:beans>
```

C.5 Beans Sandbox

Zdrojový kód C.6: XML konfigurace: Beans Sandbox

```
<capsule:beans-sandbox
         id="capsuleServer"
         class="eu.ibacz.capsules.framework.bean.BeanServer">
    <capsule:config-name value="capsuleConfig"/>
    <capsule:auto-start value="true"/>
    <capsule:port value="1099"/>
    <capsule:jvm-arguments>
    <!-- argumenty -->
    </capsule:jvm-arguments>
    <capsule:output-prefix value="out"/>
    <capsule:error-prefix value="err"/>
    <capsule:java value="java"/>
    <capsule:spring-contexts>
        context.xml
    </capsule:spring-contexts>
</capsule:beans-sandbox>
```

C.6 Web App

Zdrojový kód C.7: XML konfigurace: Web App - Spring

```
<capsule:web-app
         id="webAppClient"
         class="eu.ibacz.capsules.framework.webapp.WebAppClient">

    <capsule:config-name value="capsuleConfig"/>

    <capsule:content-type value="application/octet-stream"/>
```

```
    <capsule:restful value="false"/>

    <capsule:url value="localhost:8383"/>
    <capsule:ping-timeout value="1000"/><!--1000ms-->
    <capsule:connect-timeout value="5000"/><!--5000ms-->
    <capsule:shutdown-timeout value="5000"/><!--5000ms-->
    <capsule:reconnect-interval value="250"/><!--250ms-->
    <capsule:restart-after-ping-timeout value="true"/>
</capsule:web-app>
```

Zdrojový kód C.8: XML konfigurace: Web App - web.xml

```
<servlet>
    <servlet-name>main-servlet</servlet-name>
    <servlet-class>
      eu.ibacz.capsules.webapps.servlet.CapsuleDispatcherServlet
    </servlet-class>
    ...
    <init-param>
        <param-name>clientBean</param-name>
        <param-value>webAppClient</param-value>
    </init-param>

    <init-param>
        <param-name>profile</param-name>
        <param-value>!capsule.profile.remote</param-value>
    </init-param>
    <init-param>
        <param-name>capsule-excludes</param-name>
        <param-value>
            /css/.*
            /js/.*
            /fonts/.*
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

C.7 JMX

Zdrojový kód C.9: XML konfigurace: JMX rozhraní

```
<beans ...
    profile="!capsule.profile.remote">
    <!-- definice profilu !!!-->
    <bean id="mbeanServer"
          class="org.springframework.jmx.support.
                 MBeanServerFactoryBean"/>

    <bean id="exporter" class="org.springframework.jmx.export.
                               MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
```

```
    </bean>

    <bean id="assembler"
          class="org.springframework.jmx.export.assembler.
                 MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="namingStrategy"
          class="org.springframework.jmx.export.naming.
                 MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="jmxAttributeSource"
          class="org.springframework.jmx.export.annotation.
                 AnnotationJmxAttributeSource"/>

    <bean class="eu.ibacz.capsules.jmx.management.
                 JmxHealthCheckCapsuleManagerWrapper">
        <constructor-arg ref="capsule.manager.bean"/>
    </bean>
</beans>
```

Příloha D

Doporučený způsob použití

Doporučeným způsobem oddělení větší části je vytvoření fasády (Návrhový vzor Facade), která bude prostředníkem mezi rozhraním Capsule a existující aplikací. Tuto implementaci je nutné anotovat pomocí `@Capsule`. Rozhraní fasády tak bude na jednom místě a lze tak udržet na jednom místě i konfiguraci rozhraní Java Capsule. Následující demonstrativní příklad je pro jednoduchost bez využití fasády.

D.1 Modelová situace

D.1.1 Existující aplikace

1. Máme existující aplikaci
2. Aplikace obsahuje stránku dostupnou na `/sendMessage`, která má na starosti rozesílání zpráv všem zákazníkům a jednou týdně způsobí pád aplikace.
3. Aplikace obsahuje implementaci služby `DocumentService`, která zpracovává dokumenty pomocí knihovny, která způsobuje memory leak. S předaným dokumentem se dále nepracuje, pouze se předá ke zpracování.

D.1.2 Řešení

1. Existující aplikaci doplníme o soubor `capsules.xml` s konfiguracemi Java Capsules.
2. Současný `DispatcherServlet` nahradíme `CapsuleDispatcherServlet`, který nastavíme na zpracování pouze požadavků `/sendMessage/*` (parametr include).
3. Nakonfigurujeme Java Capsules `beans` na sandboxing `DocumentServiceImpl`.
4. Nesmí chybět manager, který vše uvede do pohybu.

D.1.3 Výsledná konfigurace

Výsledného chování docílíme i s pomocí výchozích nastavení, proto je potřeba pouze deklarovat komponenty. Klíčové části konfigurace budou vypadat následovně:

web.xml

Zdrojový kód D.1: XML konfigurace: web.xml

```
<servlet>
    ...
    <servlet-class>
        eu.ibacz.capsules.webapps.servlet.CapsuleDispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>include</param-name>
        <param-value>/sendMessage/.*</param-value>
    </init-param>
    <init-param>
        <param-name>clientBean</param-name>
        <param-value>webApp</param-value>
    </init-param>
    ...
</servlet>
```

Spring xml config

Zdrojový kód D.2: XML konfigurace: Spring

```
<capsule:capsules>
    <capsule:health-check-manager id="capsuleManager"/>
    <capsule:web-app-sandbox id="webAppSandbox"/>
    <capsule:web-app id="webApp"/>
    <capsule:beans id="beans"/>
</capsule:capsules>
```

Java

Zdrojový kód D.3: XML konfigurace: Java

```
@Capsule
@Service
public class DocumentServiceImpl implements DocumentService {
    // code
}
```

Příloha E

Struktura projektu

```
└── capsules
    ├── annotation ... Anotace dostupné pro integraci s Java Capsules
    ├── beans ... Implementace všech podpůrných tříd pro práci s objekty
    │            v sandboxu
    ├── config.parser ... Balíček pro práci s Spring XML konfigurací
    ├── context ... Konfigurační třídy všech typů Java Capsules sandboxů
    │   ├── bean ... Konfigurace Bean v Java Capsules sandboxu
    │   ├── capsule ... Konfigurace služeb v Java Capsules sandboxu
    │   ├── configuration ... Společná konfigurace Java Capsules sandboxu
    │   └── webapp ... Konfigurace Webových aplikací v Java Capsules sandboxu
    ├── event ... Události v Java Capsules
    ├── framework ... Jedná se o hlavní balíček frameworku Java Capsules,
    │   │             obsahuje abstrakce všech typů a všech částí celého
    │   │             řešení
    │   ├── bean ... Klient, server, MainClass (spustitelná třída) a
    │   │           implementace služeb pro oddělování bean
    │   ├── management ... Implementace tříd pro automatickou správu sandboxů
    │   └── webapp ... Klient, server, MainClass (spustitelná třída)
    │                  a implementace služeb pro oddělování webových aplikací
    ├── http ... Balíček s podporou pro HTTP komunikaci, kterou
    │           využívají WebApp Java Capsules
    ├── jmx ... Implementace pro podporu JMX
    ├── process ... Implementace tříd a podpůrných prostředků pro
    │               spouštění a správu Java procesů
    ├── service ... Základní implementace služeb pro správu a komunikaci s
    │               odděleným procesem
    ├── webapps ... Klient, server, MainClass (spustitelná třída) a
    │   │           implementace služeb pro oddělování webových aplikací
    │   ├── builder ... Rozhraní a jeho jediná implementace (Tomcat), které se
    │   │               starají o konfiguraci aplikačního serveru
    │   ├── request ... Rozšíření tříd ServletRequest o možnost serializace a
    │   │               deserializace mezi procesy
    │   ├── response ... Rozšíření tříd ServletResponse o možnost serializace a
    │   │                deserializace mezi procesy
    │   ├── service ... Služby pro obsluhu WebApp Java Capsules
    │   └── servlet ... Servlety pro podporu a integraci s Java Capsules
```

Příloha F

Obsah media

Přiložené medium obsahuje následující adresářovou strukturu.

```
├── Latex ... Zdrojové texty diplomové práce
├── Performance results ... Výsledky jednotlivých výkonnostních testů, grafy,
│   │                       tabulky
│   ├── java-beans
│   ├── raw-app
│   └── web-app
├── Project ... Zdrojové kódy Java Capsules
├── Simple CMS (test app) ... Zdrojové kódy aplikace Simple CMS, která byla
│                             využita pro testování
├── Martin Janys DP.pdf
└── Martin Janys DP - print.pdf
```