VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

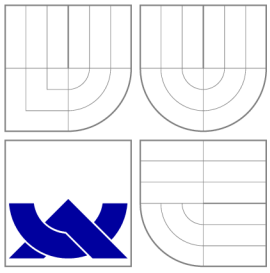
MONITORING APLIKACÍ BĚŽÍCÍCH V JVM POMOCÍ JVM TOOL INTERFACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

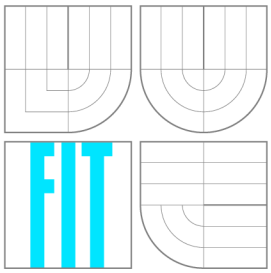
AUTOR PRÁCE
AUTHOR

PAVEL VOMÁČKA

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MONITORING APLIKACÍ BĚŽÍCÍCH V JVM POMOCÍ JVM TOOL INTERFACE

JAVA APPLICATIONS MONITORING USING JVM TOOL INTERFACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL VOMÁČKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2014

Abstrakt

Tato práce se zabývá návrhem a tvorbou agenta pro sledování běžící Java aplikace. Monitoruje obsazení zásobníku a haldy se zaměřením na lokalizaci alokace objektů. Zabývá se také vytvořením grafického uživatelského rozhraní pro zobrazení získaných informací. Je zde také popsáno použité rozhraní JVM TI a způsob práce s ním. Dále práce rozebírá virtuální stroj Javy a popisuje jeho důležité části s hlavním zaměřením na správu paměti.

Abstract

This thesis deals with the design and implementation of an agent with a purpose to watch over the stack and over the heap with focus on localization of object allocation. This information is then shown in a graphic user interface. A JVM TI interface is also described. Later parts of the thesis discuss the Java Virtual Machine and its important components with special attention to its memory management.

Klíčová slova

Virtuální stroj, monitorování, Java, JVM TI, sledování paměti, halda, zásobník.

Keywords

Virtual machine, monitoring, Java, JVM TI, memory monitoring, heap, stack.

Citace

Pavel Vomáčka: Monitoring aplikací běžících v JVM pomocí JVM Tool Interface, bakalářská práce, Brno, FIT VUT v Brně, 2014

Monitoring aplikací běžících v JVM pomocí JVM Tool Interface

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Pavla Tišnovského, Ph.D. a Ing. Radka Kočího, Ph.D.

.....
Pavel Vomáčka
20. května 2014

Poděkování

Mé poděkování patří mé rodině, která mne psychicky podporovala po celou dobu práce a měla řádnou dávku tolerance. Dále bych chtěl poděkovat panu Ing. Pavlu Tišnovskému, Ph.D. a Ing. Radku Kočímu, Ph.D, kteří mi pomohli v řešení technických otázek práce.

Obsah

Kapitola 1

Úvod

Ruku v ruce s vyššími a vyššími nároky na aplikace stoupá také náročnost jejich tvorby. S tím je samozřejmě spojena větší pravděpodobnost vytvoření chyby. Chyba se může projevovat různými způsoby, od překlepů až po špatný návrh celé aplikace. Jednou z možných chyb je nesprávná práce s pamětí a její velmi rychlé zahlcování. To je i přes dnešní velké kapacity operačních pamětí velmi nežádoucí. K rychlému zaplnění paměti, které je třeba detekovat, může také dojít například v aplikacích, které vytváří velmi dlouhé řetězce. Problémy mohou být také způsobeny zbytečně velkým množstvím vytvářených objektů. Aby programátoři nemuseli pouze staticky analyzovat zdrojový kód a zdlouhavě přemýšlet, kde asi mohli zvolit špatnou implementaci, jsou vytvářeny monitorovací a profilovací nástroje. Čím přesněji umí nástroj lokalizovat chybu, tím kratší může být doba potřebná pro odchycení chyb.

Velká část existujících aplikací monitoruje obsazení haldy zachycením obrazu paměti a její analýzou. Obraz haldy se získává buď periodicky a nebo vynuceně jako reakce na stisk příslušného tlačítka. Čímž nelze zachytit všechny alokované objekty. Najdeme i nástroje, které nabízí živé sledování alokací. Ve velké většině případů se obsazení haldy zobrazuje jako počet instancí jednotlivých tříd a suma jejich velikostí. V tomto zobrazení je problém najít nějaký konkrétní objekt a ještě těžší je lokalizovat místo jeho vzniku. Tento problém se snaží alespoň částečně vyřešit aplikace Memory.

Cílem mé bakalářské práce je vytvořit ve spolupráci s firmou RedHat Brno, s.r.o. nástroj pro tyto účely. Vytvořený nástroj by měl zvládnout odhalovat chyby, které se týkají vytváření velkých objektů nebo příliš časté vytváření objektů a pomoci lokalizovat místo vzniku těchto chyb.

Ve druhé kapitole tohoto dokumentu je rozebrán jazyk Java a důležité části virtuálního stroje, jejichž pochopení je pro tuto práci důležité. Třetí kapitola je věnována krátkému popisu existujících nástrojů pro profilování Java aplikací. Ve čtvrté kapitole je popsáno rozhraní JVM TI, které je základním stavebním kamenem vytvořeného nástroje. A konečně pátá kapitola popisuje návrh a implementaci nástroje Memory.

Kapitola 2

Java

Začátek vývoje tohoto jazyka je datován do roku 1991, kdy James Gosling začal pracovat na první verzi jazyka Java pojmenované Oak. V roce 1996 byl vydán první JDK (Java Development Kit) verze 1.0. V průběhu času vznikaly jednotlivé nové verze JDK, poslední verzí byla v roce 2011 JDK 7 (v tu dobu došlo k vydání OpenJDK a jeho uznání za oficiální referenční implementaci Javy SE 7 [22]) a v době psaní této práce Oracle vydalo JDK 8.

V dnešní době je Java druhým nejoblíbenějším programovacím jazykem [2]. Jedná se o objektově orientovaný jazyk - vše kromě osmi primitivních datových typů je objekt. Jeho nepřehlédnutelnou výhodou je velké rozpětí použití, od čipových karet (platforma JavaCard), přes mobilní telefony a různá vestavěná zařízení (platforma Java ME), osobní počítače (platforma Java SE) až po rozsáhlé distribuované systémy (platforma Java EE).

Další velkou výhodou je jednoduchá přenositelnost mezi jednotlivými operačními systémy a architekturami. Tato vlastnost je zařízena díky tomu, že zdrojový kód Javy není překládán přímo do strojového kódu, ale do tzv. bajtkódu, který je poté interpretován virtuálním strojem Javy (anglicky Java Virtual Machine). Z toho vyplývá, že na architektuře počítače je závislý pouze virtuální stroj, nikoliv napsaný zdrojový kód. Přenositelnost také podporuje řešení primitivních datových typů, kdy je u každého explicitně určena jeho velikost bez závislosti na dané architektuře. Problém s přenositelností může nastat pouze při přenášení mezi jednotlivými platformami Javy (např. z Java SE na Java ME), kdy "nižší"platforma nemusí podporovat veškeré funkce platformy "vyšší".

Za zmínku stojí také vlastnost, již uvítají všichni vývojáři aplikací, kteří byli zvyklí na nižší programovací jazyk (např. jazyk C). Virtuální stroj Javy obsahuje automatického správce paměti, díky kterému se programátor nemusí starat o to, kdy je potřeba paměť alokovat a kdy dealokovat. Zde se také dostáváme k jedné z největších nevýhod jazyka Java a tím je rychlost, která je způsobena interpretací jazyka a také prací správce paměti. Tyto nedostatky se vývojáři virtuálních strojů snaží odstranit metodami, které jsou popsány v kapitolách 2.1.2 a 2.2.4. Další nevýhodou jazyka Java je možné zahlcení paměti a také úniky paměti.

2.1 Virtuální stroj Javy

Virtuální stroj Javy (dále i JVM) je mezivrstva mezi operačním systémem počítače a aplikací napsanou nejen v jazyce Java (podporuje i např. Python, Ruby, Clojure), které zajišťuje běhové prostředí. Pro každou spuštěnou Java aplikaci je vytvořen nový proces virtuálního stroje. Jeho úkolem je zpracovávat mezikód, v Javě nazývaný bajtkód. Bajtkód vzniká po

překladu zdrojových kódů Javy do souborů s příponou .class obsahujících jednu třídu, rozhraní nebo výčet. JVM je takzvaný interpret, to znamená, že bajtkód, který mu je předložen, přímo vykonává a zprostředkovává tak komunikaci s operačním systémem, na kterém běží. Kvůli požadavkům na vyšší rychlost aplikací v Javě byly v průběhu vývoje JVM implementovány optimalizace. Kromě vylepšování implementace interpretace bajtkódu, vznikaly také nástroje jako JIT překladač popsaný v kapitole 2.1.2.

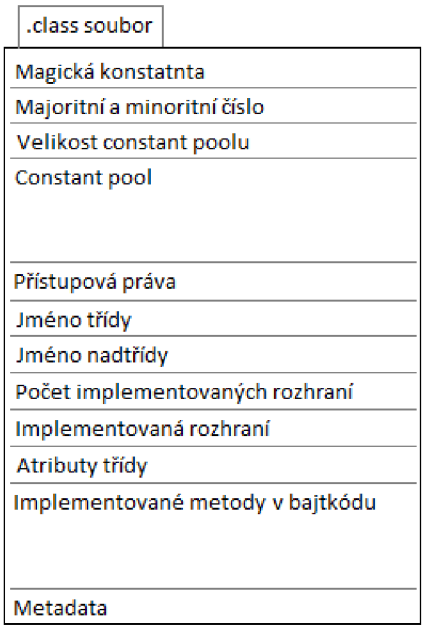
2.1.1 Soubory s příponou .class

Znalost .class souborů je důležitá pro lepší pochopení instrumentace bajkódu a z hlediska správy paměti je důležité pochopit, kde jsou ukládány konstanty a konstantní řetězce. Již bylo řečeno, že .class soubor vzniká překladem .java souboru, zároveň bylo uvedeno, že .class soubor obsahuje bajtkód pouze jedné třídy nebo rozhraní. Z možnosti implementovat v jednom .java soubrou více tříd vyplývá, že při překladu jednoho .java souboru může vzniknout i větší množství .class souborů. Pojmenování .class souborů se řídí přesnými pravidly. Samostatný .class soubor vznikne pro každou třídu, výčet a rozhraní. Způsob vytvoření názvu .class souboru je popsán v následujícím výčtu (čerpáno z[23]). Za pomlčkou je vždy sepsán způsob tvorby názvu pro uvedený typ .class souboru:

- běžnou třídu - stejně jako třída
- vnitřní třídu - jméno obalové třídy, znak dolaru a jméno
- lokální třídu - jméno obalové třídy, znak dolaru, generovaný index, jméno lokální třídy
- anonymní třídu - jméno obalové třídy, znak dolaru, generovaný index
- výčet - jako běžná třída
- rozhraní - stejně jako třída

Uvnitř virtuálního stroje jsou názvy jednotlivých tříd a datových typů ukládány speciálním způsobem. Jsou reprezentované takzvanými signaturami. Pro každý datový typ existuje speciální signatura - pro boolean jí je písmeno 'Z', pro byte 'B', pro char 'C', pro int 'I', pro long 'J', pro float 'F', pro double 'D' a pro třídu 'L'. Před každou signaturou je možné najít znak '[', který značí, že se jedná o pole, popřípadě dva pro dvourozměrné pole. Pomocí těchto signatur jsou identifikovány i metody, jejich návratové typy a parametry. Signatury tříd se skládají, jak bylo řečeno výše, ze znaku 'L' a plně kvalifikovaného jména třídy, tím je myšlen název třídy s umístěním v balíku, do kterého spadá. Jako oddělovač se místo znaku '.' používá '/' a signatura je ukončena znakem ';'. Přikladem může být třída *String*. Její signatura má tvar (bez uvozovek): *"Ljava/lang/String;"*. Tolik k názvům, nyní bude popsána struktura .class souborů.

Soubory .class jsou tvořeny sekvencí bajtů. Samotná struktura není příliš složitá a je znázorněna na obrázku 2.1. Na začátku každého souboru je takzvaná magická konstanta, jde o 32-bitovou hodnotu, která určuje, že se jedná o .class soubor a má hodnotu *0xCAFEBABE*. Následují dvě 16-bitové hodnoty, tzv. majoritní a minoritní číslo, které označují použitou verzi bajtkódu a slouží pro základní kontrolu kompatibility. Pro .class soubory vytvořené pomocí překladače pro verzi Java 7 SE bude majoritní číslo 51, pro verzi Java 8 to bude číslo 52. Dalších 16 bitů uchovává číselnou informaci o tom, kolik záznamů obsahuje constant pool. Tato hodnota je o jedničku vyšší než počet záznamů v constant poolu.



Obrázek 2.1: Struktura class souboru.

Nyní se dostáváme na nejzajímavější část .class soboru a tím je již zmíněný constant pool. Jedná se o tabulku struktur, která je indexována od jedničky do hodnoty o jedno nižší než ukládá předchozích 16 bitů. Každá položka v constant poolu začíná jednobytovým tagem, který značí typ záznamu. Seznam všech tagů i s hodnotami je uveden v tabulce 2.1.

Nyní rozebereme jaké položky může constant pool obsahovat a k čemu slouží. Půjdeme postupně, tak jak jsou jednotlivé tagy uvedené v tabulce 2.1. Jednotlivé položky s sebou kromě tagu nesou jeden až dva parametry, obvykle 16-bitové odkazy (tagy) do constant poolu, které musí být validní, tedy nesmí odkazovat mimo constant pool. V následujícím výčtu budou jednotlivé položky popsány:

- *CONSTANT_Class* nesoucí v prvním parametru odkaz na položku typu *CONSTANT_Utf8*, která reprezentuje validní název třídy nebo rozhraní. Druhý parametr je prázdný.
- *CONSTANT_Fieldref, CONSTANT_Methodref a CONSTANT_InterfaceMethodref* . Jsou reprezentovány podobnou strukturou, která využívá oba parametry. Prvním parametrem je odkaz na položku typu *CONSTANT_Class*. Rozdíly mezi jednotlivými položkami jsou ty, že *CONSTANT_Methodref* v prvním parametru musí odkazovat na třídu, ne na rozhraní, *CONSTANT_InterfaceMethodref* musí odkazovat na rozhraní, nesmí na metodu a *CONSTANT_Fieldref* může odkazovat jak na metodu tak na rozhraní. Druhý parametr musí být opět validní odkaz do constant poolu na položku typu *CONSTANT_NameAndType*, která obsahuje jméno a popis metody nebo pole.
- *CONSTANT_String* se používá k uložení konstantních řetězců dané třídy. Využívá pouze jeden parametr, ve kterém je odkaz na položku typu *CONSTANT_Utf8* obsahující daný řetězec.

Textový popis	Hodnota
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Tabulka 2.1: Naměřené hodnoty

- *CONSTANT_Integer* a *CONSTANT_Float* používají pouze jeden parametr, ve kterém je uložena přímo hodnota čísla.
- *CONSTANT_Long* a *CONSTANT_Double* ukládají 64-bitové číselné hodnoty. Používají dva parametry, jimiž jsou vyšší bity a ve druhém parametru nižší bity, které určují hodnotu čísla. Zvláštností je, že po těchto hodnotách v constant poolu vždy následuje jedna volná položka obsahující pouze tag.
- *CONSTANT_NameAndType* používá dva parametry. V prvním najdeme odkaz na *CONSTANT_Utf8*, ve kterém je uložen název metody nebo pole, a ve druhém odkaz opět na *CONSTANT_Utf8*, ve kterém je uložen deskriptor metody nebo pole.
- *CONSTANT_Utf8* je používán pro reprezentaci řetězců. Používá dva parametry - první parametr určuje počet bytů v bytovém poli obsazených řetězcem. Tato hodnota se neshoduje s délkou řetězce. Druhým parametrem je bytové pole, ve kterém jsou uloženy byty řetězce, řetězec není zakončen nulou.
- *CONSTANT_MethodHandle* je používán pro uložení informací o metodách. Používá dva parametry. Prvním je typ reference, podle které je určeno na jaký typ položky do constant poolu bude odkazovat druhý parametr. Druhý parametr je odkaz do constant poolu.
- *CONSTANT_MethodType* reprezentuje typ metody. Používá pouze jeden parametr, který odkazuje na *CONSTANT_Utf8*, který v sobě má deskriptor metody.
- *CONSTANT_InvokeDynamic* používaný speciální instrukcí invokedynamic. První parametr je odkaz do speciálního pole metod v class souboru. Druhým je odkaz na *CONSTANT_NameAndType*.

Pro představu, jaké položky může constant pool obsahovat v jednoduché reálné třídě doporučuji nahlédnout na internetovou stránku [21].

Po constant poolu v .class souboru následuje část obsahující přístupová práva - jedná se o 16 příznakových bitů, poté jméno této třídy a nadtříd, počet implementovaných rozhraní a samotná implementovaná rozhraní - pro každou z těchto položek je vyhrazeno 16 bitů, kromě pole implementovaných rozhraní, což je pole 16-bitových položek ukazujících do constant poolu. Následují atributy třídy, kódy jednotlivých metod v bajtkódu a celý soubor ukončují metadata o třídě. Konkrétní informace o instrukcích a detailnější informace o .class souborech jsou popsány v [18].

2.1.2 Just-in-time překladač

Vzhledem k tomu, že běh programů napsaných v jazyce Java nemůže z důvodů interpretace soupeřit v rychlosti s programy napsanými v nativních jazycích, začaly novější verze JVM využívat funkce JIT překladače (Just-in-time compiler), který za běhu překládá bajtkód do nativního strojového kódu. Zpočátku byla výhoda, kterou je možnost optimalizace pro právě používanou architekturu, stále zastíněna nevýhodou prodlevy při spuštění programu, která byla nutná k překladu a optimalizaci kódu. Z tohoto důvodu se u některých verzí JVM používá kompromis mezi těmito dvěma stránkami JIT compileru. Po spuštění aplikace dochází ke klasické interpretaci bez použití JIT compileru. Za běhu aplikace se zaznamenávají statistiky o tom, které metody byly kolikrát volány. Optimalizace překladem do nativního kódu se poté provádí pouze u těch částí, které jsou volány velmi často.

JVM režim klient a server

Jednou z implementací virtuálního stroje používající JIT compiler je běžně používaný HotSpot JVM. Ten v sobě má implementované dva řežimy, rozdíl mezi nimi je hlavně ve způsobu použití JIT compileru.

Jedním režimem je režim klient, ten je zaměřen na rychlý start aplikace a pomalejší provádění. Zde je použita optimalizace sledováním interpretace následována překladem často volaných metod. Dalším rozdílem je snaha alokovat menší haldu a tím ušetřit paměť pro operační systém a další aplikace. Z tohoto popisu je zřejmé, že využití klientského režimu bude hlavně v malých aplikacích a appletech.

Druhým režimem je server. Tento režim je vyladěn pro co možná nejrychlejší běh aplikace, z tohoto důvodu je zde přeložena do nativního strojového kódu a optimalizována mnohem větší část aplikace, což může způsobit znatelnější zpoždění při startu. Pro větší pružnost aplikace bývá v tomto režimu také alokována větší halda.

Při spouštění aplikace napsané v Javě, můžeme pomocí příkazové řádky s přepínačem *-client* popř. *-server* zvolit spuštění Java HotSpot Client Virtual Machine, popř. Java HotSpot Server Virtual Machine. Od verze J2SE 5.0 dochází při spouštění aplikace ke kontrole, zda je aplikace spouštěna na server-class počítači. Pokud ano, je zvolen režim server, pokud ne, je zvolen režim klient. Ve verzi Java SE 6 dochází k detekci i ve chvíli, kdy není zadán ani jeden z přepínačů a volba režimu se provádí podle tabulky 2.2. Od verze Java SE 7 nelze na 64-bitových systémech použít Java HotSpot Client Virtual Machine ani při spuštění aplikace s přepínačem *-client*. Automaticky se používá Java HotSpot Server Virtual Machine [4].

2.1.3 Instrumentace bajtkódu

Po spuštění JVM dochází pomocí takzvaného classloaderu k postupnému načítání .class souborů do operační paměti a jejich převedení do takového tvaru, aby bylo možné bajtkód

Architektura	OS	Klientský režim	Kontrola server-class	Serverový režim
SPARC 32bit	Solaris	V	X	V
i586	Solaris	V	X	V
i586	Linux	V	X	V
i586	MS Windows	X	O	V
SPARC 64bit	Solaris	O	O	X
AMD64	Solaris	O	O	X
AMD64	Linux	O	O	X
AMD64	MS Windows	O	O	X

Tabulka 2.2: Výběr režimu v závislosti na OS, X - implicitně vybráno, O - nepodporuje, V - lze explicitně zvolit.[5]

interpretovat. V průběhu načítání tříd je možné měnit jejich implementaci. Další možností je měnit těla metod již načtených tříd. Tato funkce nám dává možnost alespoň částečně změnit běh aplikace, od které nemáme zdrojové soubory, tudíž nepřichází v úvahu nový překlad. Hlavní použití ale spočívá v možnosti měnit bajtkód tak, abychom mohli sledovat volání určité metody, měřit strojový čas strávený v určité metodě a velké množství dalších informací. Nespornou výhodou tohoto procesu je, že je možné využít JIT compiler (kapitola 2.1.2) i pro instrumentovaný bajtkód a tím zrychlit sledování aplikace.

Od JDK verze 1.5 Java nabízí balík *java.lang.instrumentation*, který umožňuje instrumentaci bajtkódu bez využití nativních knihoven. Třídy z toho balíku dovolují psaní agenta, který bude provádět instrumentaci bajtkódu za chodu aplikace přímo v Javě. Od této verze je pro instrumentaci také možné využívat JVM TI (bude popsáno v kapitole 4), které úzce spolupracuje s JNI (Java Native Interface) 4.1. Pro instrumentaci je vyvinuta knihovna *java_crw_demo* [20], která je napsána v jazyce C. Dále existuje několik nástrojů, které nejsou přímo v JDK, ale také poskytují možnost instrumentace bajtkódu, například následující:

- BCEL - nástroj poskytující funkcionalitu pro vytváření, manipulaci a analýzu .class souborů. Nyní je častěji používán nástroj ASM. Více informací je k nalezení zde [6].
- ASM - nástroj, který umožňuje modifikovat existující třídní soubory, dynamicky generovat třídy, zároveň ale může být použit i staticky. Dále poskytuje algoritmy pro modifikaci a analýzu bajt kódu. Více informací na webu [7].
- Javassist - tato knihovna má dvě úrovně přístupu k instrumentaci bajtkódu. První je úroveň zdrojového kódu, zde je možné editovat bajtkód bez jeho přesné znalosti. Druhou je úroveň bajtkódu, kde je nutná znalost jeho struktury, protože se upravují data ve formátu v jakém jsou uloženy .class soubory. Další informace jsou k nalezení na webu [17].

2.2 Správa paměti v JVM

V této kapitole si popíšeme běhové paměťové části, které využívá JVM a v jejím závěru také správce paměti (anglicky Garbage Collector). Běhové paměťové části můžeme rozdělit do dvou skupin a to podle toho, zda jsou společné pro všechna vlákna Java aplikace nebo je má každé vlákno vlastní. Do první skupiny spadá prostor pro uložení informací o načtených

třídách - method area (kapitola 2.2.1) a halda (popsána v kapitole 2.2.3). Do druhé skupiny řadíme samostatný zásobník (popsán v kapitole 2.2.2) a samostatný PC čítač, který ukazuje na právě zpracovávanou instrukci, a posledním je zásobník pro nativní metody.

2.2.1 Method-area

Tato paměťová část je alokována při startu virtuálního stroje a slouží k ukládání informací o třídách. Při startu virtuálního stroje jsou do této paměťové části ukládány informace, mezi kterými je tzv. run-time constant pool, data polí a metod, kód metod a konstruktorů včetně speciálních metod používaných při inicializaci instancí, tříd a rozhraní. Velikost method-area nemusí být fixní a může se za běhu programu měnit. Stejně jako u haldy (method-area může být i součástí haldy) se o správu paměti může starat Garbage Collector, protože i zde se může stát, že některá třída již přestane být z aplikace dostupná.

Zde stojí za zmínku zejména run-time constant pool. Jedná se o běhovou reprezentaci constant poolu tak, jak byl popsán v kapitole 2.1.1. Díky použití constant poolu mohou mít instrukce, ve kterých dochází k volání metody krátký zápis, protože jejich parametrem není úplné kvalifikované jméno třídy a metoda, která se volá, ale pouze číselný odkaz do constant poolu. V constant poolu je již přes parametry položky, které tvoří odkaz opět do constant poolu, možné vše zjistit. Toto řešení může být u větších aplikací velmi úsporné z hlediska délky bajtkódu. Další využití je v přístupu k objektům na haldě, ke kterým nelze přistoupit přímo, ale obvykle se k ním přistupuje pomocí jména atributu, které je také uloženo v constant poolu. Tudíž instrukce pracující s těmito daty opět využívají pouze odkazy do constant poolu.

2.2.2 Zásobník

Pro každé vlákno aplikace je vytvořen nový zásobník. Pokud dojde ve vláknu k volání metody, je v zásobníku daného vlákna vytvořen zásobníkový rámec (anglicky stack frame). Protože je práce se zásobníkovými rámci zajištěna pouze funkcemi pop a push, a protože jednotlivé rámce nemusí být uloženy v paměti za sebou, je možné, aby i rámce byly uloženy na haldě. Rámec zaniká s koncem metody. V každém rámci je uložena informace o bodu návratu - místo návratu běhu aplikace po ukončení metody.

Každý rámec obsahuje parametry metod (u nestatických metod včetně odkazu na vlastní třídu this) a hned za nimi pole lokálních proměnných. Do pole lokálních proměnných je možné uložit operandy těchto typů: jednu položku zabírají boolean, byte, char, short, integer, float, reference a návratový kód, dvě položky zabírá long a double. K parametrům metod nebo lokálním proměnným se přistupuje pomocí indexů, tj. první má 0, druhý 1.

Dále rámec obsahuje zásobník operandů, jedná o klasický LIFO zásobník. Jeho velikost je určena už v čase překladu zdrojového kódu. Za běhu je tato velikost kontrolována a také je kontrolována správná práce s datovými typy. Například není možné použít instrukci pro integer nad datovými typy float i přesto, že integer a float mají stejnou bitovou šířku. Zásobník operandů se ještě využívá pro předávání parametrů volaným metodám a také pro uložení jejich návratových hodnot.

2.2.3 Halda

Halda (anglicky heap) slouží k ukládání veškerých instancí tříd, jejich parametrů a polí, které jsou používány za běhu aplikace. Její velikost může být určena implicitně nebo explicitně pomocí přepínačů při spouštění aplikace. Jejich popis je k nalezení v [4]. Velikost

haldy může být fixní nebo může být nastavena maximální, popřípadě minimální hodnota. Velikost alokované paměti pro haldu se může za běhu programu měnit a to v závislosti na potřebách běžící aplikace, halda může být jak rozšiřována tak zmenšována. Ve chvíli, kdy je nutné haldu expandovat a není pro tuto operaci dostatek volné operační paměti, dojde k chybě *OutOfMemoryError*. Určení hodnoty maximální a minimální velikosti haldy je velmi důležité zejména z důvodu rychlosti aplikace.

Ne vždy je halda využívána efektivně. K neefektivitě může docházet díky vlastnostem jazyka Java a nebo kvůli nevhodně napsanému zdrojovému kódu. Příkladem takové vlastnosti může být neměnnost řetězců. To znamená, že při složitějších operacích s řetězci, jako je konkatenace nebo hledání podřetězce v řetězci, se vytváří velké množství objektů a i dlouhé řetězce se musí kopírovat. A je zřejmé, že se to odrazí na výkonu aplikace. Další vlastností, která nepomáhá efektivnímu využití haldy je to, že řetězec je reprezentován objektem, který nese informace o objektu spolu s ukazatelem na pole, které obsahuje řetězec. Zde nastává problém s krátkými řetězci, protože se musí i u velmi krátkého řetězce ukládat informace o objektu, která může přesahovat velikost samotného řetězce.

2.2.4 Garbage Collector

Jak již bylo zmíněno výše, Garbage Collector se stará o uvolňování nepotřebných objektů. Náplní jeho činnosti je také zjišťování dostupnosti objektů. V případě, že objekt již není dostupný, je možné ho dealokovat. Dostupností objektu je myšlena existence alespoň jedné reference a to např. v zásobníkovém rámci, třídní proměnné, atributu objektu. Tento způsob správy paměti zpříjemňuje práci vývojářům, kteří se nemusí starat o to, kde objekt, který byl alokován, dealokují, jako je tomu u nižších programovacích jazyků (např. jazyk C).

První implementace správců paměti nebyly příliš výkonné a běh aplikace brzdily. V dnešní době je již tento problém menší, ovšem stále nezanedbatelný. Za dobu vývoje správců paměti vzniklo několik jejich implementací a bylo vystřídáno i několik přístupů.

Znalost implementace správců paměti je důležitá při spouštění aplikací a konfigurování virtuálního stroje pro určitý typ aplikace. Jiné požadavky budou pro aplikaci realtimovou, kde musí být poskytnuta okamžitá odezva. Zde se využijí správci inkrementálního typu, kteří nemusí projít celou haldou najednou a díky tomu zastavují aplikaci častěji, ale zato na velmi krátkou dobu. Dalším příkladem může být aplikace provádějící dlouhý výpočet, který může běžet i několik hodin. V tomto případě není problém, aby správce zastavil aplikaci i na několik sekund, proto je možné použít tzv. stop-the-world collector, který musí projít celou haldu najednou. Je nutné si uvědomit, že většina správců paměti musí běh aplikace alespoň na krátké okamžiky pozastavit. Další důvod k zamyšlení nad volbou určitého správce paměti, může být použití systému s více mikroprocesory nebo více jádry na jednom čipu. Existují totiž správci paměti, kteří běží paralelně s aplikací, tzv. concurrent collector. Nyní jsou ale nejčastěji používaní generační správci.

Generační algoritmy

Tyto algoritmy správce paměti vznikly díky myšlence, že u nově vzniklých objektů je větší pravděpodobnost, že budou uvolněny. V tomto případě se halda dělí na několik částí. Část nazývanou "young generation"(která může být ještě rozdělena na "eden space"a "survivor space") pro ukládání právě vzniklých a nebo krátkou dobu existujících objektů a část "tenured generation"pro objekty starší. Každý vytvořený objekt je automaticky ukládán v "young generation"části, do druhé části se dostává až po nějaké, předem specifikované

podmínce, například po tom, co "přežije"několik průchodů Garbage Collectoru. Na jednotlivých částech haldy se poté spouští i další různé algoritmy správců paměti, tak aby vyhovovaly způsobu použití jednotlivých částí.

Kapitola 3

Profilování aplikací

Profilování aplikací spočívá ve sledování běžících aplikací a získávání informací o nich. Tyto informace je pak možné využít při optimalizaci programu. Profilovací aplikace často sledují čas strávený v metodách nebo funkcích, počet provedení jednotlivých cyklů, počet přístupů na disk, obsazení haldy nebo jejich jednotlivých částí, atd. Využití profilovacích nástrojů roste s požadavky na velkou rychlost, propustnost a zároveň co možná nejmenší využití výpočetních zdrojů počítače.

Existuje více způsobů sledování aplikací. První je založeno na událostech. Profilovací nástroj získává informace o běžící aplikaci pomocí registrování událostí, ke kterým v této aplikaci dochází. Příkladem může být právě rozhraní JVM TI (kapitola 4). Toto řešení má výhodu v přesnosti sledovaní. Nevýhodou je to, že snižuje rychlost aplikace. Další možností je monitorování pomocí vzorků, kdy se periodicky získávají informace o běžící aplikaci a vytváří se statistická data. Toto monitorování není tak přesné, ale nezpomaluje monitorovanou aplikaci. Příkladem může být profiler Sysprof [13]. Další možností je statické sledování chování, kterým je čtení zdrojového kódu. Toto řešení je možné u drobných aplikací.

3.1 Přehled existujících nástrojů

Existuje poměrně velké množství aplikací, které se zabývají profilováním Java aplikací. Ty nejznámější, jako například JProfiler nebo YourKit Java Profiler, neexistují v bezplatné verzi. Nás ale budou zajímat aplikace, které jsou volně dostupné, protože výsledná aplikace bude také volně dostupná. Zároveň podmínkou pro vybrané aplikace je existence grafického uživatelského rozhraní. Do výběru se dostali nástroje VisualVM a Thermostat. Existují také nástroje spouštěné z terminálu jako je jmap, jstack, jhat, které jsou součástí JDK.

3.1.1 VisualVM

Tento nástroj nabízí různé možnosti sledování vzdálených i lokálních běžících virtuálních strojů. Mezi sledované veličiny patří načtené třídy, procesorový čas, vlákna a také paměť. Obsazení paměti je možné zobrazit grafem a nebo výpisem počtu objektů jednotlivých tříd a součtu jejich velikostí. To jsou jediné veličiny, které lze o objektech získat. Dále je možné nastavit filtr, pro sledování jen některých tříd a hodnotu, která určuje kolikátá každá alokace se bude sledovat. Nabízí se možnost sledování pomocí vzorků, tedy statistické sledování, nebo přesné sledování, pomocí událostí. Bližší informace o objektech, jako například to, kde vznikají tento nástroj neposkytuje. Zároveň není ani možné sledovat vytváření

jednotlivých objektů, ale pouze celkový počet objektů jednotlivých tříd. Oficiální stránky věnované nástroji VisualVM jsou [9].

3.1.2 Thermostat

Thermostat je nástroj, který poskytuje, tak jak uvádějí autoři na webových stránkách [16], možnost sledování více běžících virtuálních strojů a to zejména těch vzdálených. Souborem funkcí je Thermostat velmi blízko předchozímu nástroji, poskytuje zobrazení využití procesoru, vláken, načtených tříd, práce garbage collectoru a také obsazení paměti. Výsledky zachytávání alokací objektů prezentuje názvem třídy objektu, počtu jejich instancí objektů této třídy a sumě velikosti těchto objektů. Je možné zobrazit také objekty jednotlivě s informací o jejich velikosti. Další informací o objektu je pouze adresa, na které je alokován. Zde je již situace pro sledování jednotlivých objektů lepší, ovšem stále nejsou vypisovány žádné informace o místě alokace objektu.

Kapitola 4

Java Virtual Machine Tool Interface

Toto rozhraní [3] bylo navrženo a implementováno jako nástupce JVM Debug Interface [10] a JVM Profiling Interface [19]. Hlavní využití bude tedy v profilovacích a debugovacích nástrojích - v tomto případě agentech. Konkrétněji je možné nastavovat a využívat breakpointy, zjišťovat volané metody, přistupovat k objektům na haldě, sledovat práci garbage collectoru, získávat informace o třídách a vláknech a další. Toto rozhraní velmi úzce spolupracuje s rozhraním JNI, krátký přehled toho rozhraní bude uveden v následující kapitole 4.1.

Agenti využívající rozhraní JVM TI jsou závislí na platformě - vyskytují se jako sdílené knihovny. Ty jsou na různých systémech reprezentovány různě (např. na Windows *.dll, na Solaris systémech *.so) a jsou obvykle programovány v jazyce C nebo C++, protože pro použití JVM TI je nutné mít možnost volat funkce podle konvencí jazyka C. Zároveň je nutné moci do kódu připojit hlavičkový soubor jvmti.h. Komunikace může být obousměrná a probíhá pomocí volání funkcí rozhraní JVM TI (komunikace agenta s virtuálním strojem) a pomocí callback funkcí (komunikace virtuálního stroje s agentem).

4.1 Java Native Interface

Protože rozhraní JVM TI je v některý případech přímo závislé na funkcích rozhraní JNI, bude ve zkratce popsáno i toto rozhraní. JNI je rozhraní, které umožňuje aplikacím napsaným v jazyce Java používat a nebo spolupracovat s programy napsanými v jazycích, které pracují na nižší úrovni, jako je například jazyk C, C++ nebo assembler. Využití této možnosti lze spatřit v potřebě urychlit velmi často volanou část zdrojového kódu napsaného v Javě pomocí přepsání do nativního kódu, případně využívat speciální možnosti závislé na architektuře stroje, ke kterým není přes virtuální stroj jazyka možno přistupovat. Důvodem k vytvoření tohoto rozhraní byla odlišnost způsobu práce s nativním kódem v různých implementacích virtuálního stroje, což vedlo k velmi složité práci pro programátory, pokud měli za cíl širší využití jejich zdrojových kódů. V dalším odstavci bude popsán způsob vytvoření nativní metody, kterou lze volat z jazyka Java.

Pro vytvoření nativní metody je nutné udělat několik kroků. Prvním je vytvoření třídy, ve které se má nativní metoda volat a deklarovat v ní metodu s klíčovým slovem *native*. Pokud bude pro implementaci metody použit jazyk C nebo C++, je možné nechat vygenerovat odpovídající hlavičkový soubor pomocí příkazu `javah -jni` následovaným názvem

.class souboru, který vznikne překladem vytvořeného souboru s deklarovanou metodou. Poté definujeme v jazyce C nebo C++ funkci, která bude odpovídat deklaraci funkce ve vygenerovaném hlavičkovém souboru. Nyní stačí přeložit zdrojový kód napsaný v jazyce C nebo C++ jako sdílenou knihovnu. Tuto knihovnu je potom nutné načíst před voláním nativní metody ve zdrojovém kódu jazyka Java. To se provede zavoláním metody *loadlibrary("název_knihovny")* z balíku System. Nyní je možné zavolat ze zdrojového kódu jazyka Java nativní metodu, která spustí provádění kódu napsaného v jazyce C nebo C++.

Kromě volání nativních metod poskytuje rozhraní také velké množství funkcí interagujících s virtuálním strojem jazyka Java. Pro tyto funkce rozhraní JNI obsahuje speciální datové typy jako je *jthread* odpovídající vláknu, *jclass* odpovídající třídě, *jobject* odpovídající objektu a poté všechny primitivní datové typy, které podporuje jazyk Java ovšem s prefixovým písmenem 'j'. Tyto datové typy používá i rozhraní JVM TI. Mezi funkce, kterými toto rozhraní disponuje patří funkce pro práci se třídami, jako je vyhledávání tříd a nadtříd. Dále umožňuje práci s výjimkami - vyvolání určité výjimky. Existuje podpora pro práci s objekty, je možné vytváření nových objektů a zjišťování třídy objektu. Také je možné pracovat s objekty typu *String*, tudíž je lze vytvářet, zjišťovat jejich délku, obsah. V neposlední řade se nabízejí funkce pro práci s poli, pro vyhledávání identifikátorů metod, komunikaci se stavem virtuálního stroje a další. Konkrétní a kompletní popis lze najít v dokumentaci tohoto rozhraní [8].

4.2 Datové typy JVM TI

JVM TI definuje poměrně velké množství datových typů. Veškeré jsou definované v hlavičkovém souboru jvmti.h. Vzhledem k tomu, že je datových typů velké množství budeme se zabývat pouze jejich podmnožinou.

- Nejdříve se zaměříme na datový typ *jvmtiEnv**. Jedná se o ukazatel na samotné prostředí JVM TI a má v sobě veškeré informace o spojení. Položka v proměnné tohoto typu je ukazatel na tabulku funkcí, je to pole ukazatelů na funkce JVM TI. Tudíž pokud chceme volat nějakou funkci rozhraní JVM TI, je nutné jí volat pomocí proměnné typu *jvmtiEnv*.

- Další důležitý datový typ je *jvmtiError*. Funkce z rozhranní JVM TI mají návratový typ právě *jvmtiError*. Pomocí toho typu jsou předávána jednotlivá chybová hlášení funkcí.

- Struktura typu *jvmtiEventCallbacks* obsahuje ukazatele na všechny callback funkce. Tam, kde není hodnota ukaztele *NULL*, je ukazatel na callback funkci.

- Struktura typu *jvmtiCapabilities* obsahuje všechny schopnosti JVM TI. Každá schopnost může nabývat hodnoty 1 nebo 0, podle toho, zda se události odpovídající schopnosti mají registrovat nebo nemají.

- Struktura *jvmtiHeapCallbacks* obsahuje místo pro 15 callback funkcí. V době psaní této práce jich je využíváno pouze 5. Každá z nich reaguje na jiné objekty uložené na haldě. Jednotlivé callback funkce reagují popořadě na tyto objekty: běžné objekty, reference na objekty, atributy primitivního datového typu, pole primitivních datových typů a poslední pro řetězce.

4.3 Fáze běhu

Předtím než začneme rozebírat práci s funkcemi a jednotlivé funkce, bude dobré vysvětlit fáze běhu, se kterými JVM TI pracuje. JVM TI rozděluje běh virtuálního stroje do pěti fází takto:

- První je fáze OnLoad, takto je pojmenovaná část běhu, která začíná zavoláním funkce *Agent_OnLoad()* a končí návratem z této funkce.
- Druhá se nazývá Primordial fáze, začíná návratem z funkce *Agent_OnLoad()* a končí událostí VMStart a ta značí spuštění virtuálního stroje.
- Následující část se označuje Start fáze, její trvání je ohraničeno událostí VMStart a událostí VMInit, která značí úspěšnou inicializaci virtuálního stroje.
- Čtvrtou fází je Live fáze (neboli živá fáze), začíná událostí VMInit a končí událostí VMDeath. V této fázi je virtuální stroj v běhu a je možné připojit agenta za běhu pomocí *Agent_OnAttach()*.
- Poslední fází je Dead fáze, která je odstartována událostí VMDeath a nebo chybou při spouštění virtuálního stroje.

Důvod, proč je důležité mít povědomí o těchto fázích je, že některé funkce JVM TI lze volat pouze v některých fázích.

4.4 Připojení a odpojení agenta

Agenty je možné připojovat jak při spouštění virtuálního stroje, tak v době jeho běhu. Vždy je pro spuštění volána právě jedna funkce. V závislosti na tom, jak je agent spouštěn, je vitruálním strojem volána funkce *Agen_OnAttach()* nebo *Agent_OnLoad()* - alespoň jedna z těchto funkcí musí být implementována, jinak by nebylo možné agenta spustit. Dále může být implementována funkce *Agent_OnUnload()* sloužící k odpojení agenta. Spouštění za běhu způsobí inicializaci agenta, který po jejím dokončení informuje virtuální stroj o tom, zda proběhla inicializace správně. Je důležité si uvědomit, že při tomto způsobu volání agenta jsou některé schopnosti (capabilities) rozhraní JVM TI nedostupné. Například nastavení systémových vlastností.

Start agenta zároveň se startem virtuálního stroje dovoluje nastavení všech požadovaných schopností. K vysvětlení, proč je nutné je nastavovat, se dostaneme v další kapitole. Zde si pouze řekneme, že u některých je nezbytné, aby byly nastaveny právě před inicializací virtuálního stroje a toho dosáhneme pouze spuštěním agenta tímto způsobem. V této části je také důležité zjistit, zda virtuální stroj plně podporuje rozhraní JVM TI požadované verze. K tomu slouží funkce *GetEnv()* z rozhraní JNI, která akceptuje jako jeden z parametrů ukazatel na proměnnou typu *jvmtiEnv*, kterou v případě úspěšné kontroly, že virtuální stroj obsahuje správnou verzi rozhraní, naplní.

Při odpojování agenta a uvolňování knihovny virtuální stroj volá funkci *Agent_OnUnload()* (pokud je implementována). Odpojení je způsobeno obvykle ukončením běhu virtuálního stroje, a to jak chybovým, a nebo běžným, včetně chyb při startu virtuálního stroje.

4.5 Callback funkce

Jak již bylo řečeno komunikace virtuálního stroje s agentem probíhá pomocí callback funkcí. Callback funkce dovolují takzvané asynchronní volání funkcí. To znamená, že při implementaci zaregistrujeme určitou funkci k určité události a poté, při výskytu události je volána odpovídající funkce. V ní je implementována reakce na tuto událost. Abychom mohli používat callback funkce rozhraní JVM TI, respektive aby je mohl používat virtuální stroj, je nutné při implementaci provést registraci, nastavit schopnosti agenta a povolit zachytávání událostí.

Registrace probíhá voláním funkce *SetEventCallbacks()*, do níž se předá struktura typu *jvmtiEventCallbacks* obsahující ukazatele na jednotlivé funkce odpovídající událostem. Tuto funkci lze volat pouze v OnLoad fázi nebo v Live fázi. Všechny callback funkce musí být zaregistrovány v jednu chvíli.

Dostáváme k nastavení schopností agenta. K tomuto kroku nám poslouží funkce *AddCapabilities()*, která lze volat ve stejných fázích jako funkce předchozí, tj. OnLoad a Live fáze. Běžně se tato funkce používá v OnLoad fázi. Při použití v Live fázi některé virtuální stroje nepodporují všechny možné schopnosti. Nastavení určitých schopností se provádí předáním ukazatele na strukturu typu *jvmtiCapabilities*. Pokud nebude virtuální stroj podporovat hlášení některých událostí, které chceme aktivovat, dojde k navrácení chybové hodnoty.

Nyní je už jen nutné zachytávání událostí povolit. Tuto činnost je opět možné provádět během OnLoad fáze a Live fáze. Použití v Live fázi může být v tomto případě užitečné, protože to umožňuje reagovat na externí konfiguraci a za běhu zakazovat a povolovat sledování jednotlivých zaregistrovaných událostí. K povolení se používá funkce *SetEventNotificationMode()*, do které se předá typ události z proměnné typu *jvmtiEvent* a konstanta *JVMTI_ENABLE* pro povolení nebo *JVMTI_DISABLE* pro zakázání, která odpovídá popořadě hodnotě 1 a 0.

Callback funkce z rozhraní JVM TI použité ve vytvářené aplikaci jsou:

- *VMStart()* - funkce volána jako reakce na událost startu virtuálního stroje.
- *VMInit()* - reakce na inicializaci virtuálního stroje.
- *VMDeath()* - funkce, která je volána při přechodu do Dead fáze.
- *GarbageCollectionStart()* - tato funkce reaguje na začátek práce správce paměti.
- *GarbageCollectionFinish()* - opak předchozí funkce, je volána při ukončení práce správce paměti.
- *ObjectFree()* - funkce reagující na událost uvolnění objektu.
- *VMObjectAlloc()* - funkce reagující na některé typy alokací objektů.

4.6 Funkce JVM TI

Funkce rozhraní JVM TI slouží ke komunikaci agenta s virtuálním strojem. V této kapitole budou popsány právě ty funkce, které byly využity při implementaci. Všechny funkce jsou volány pomocí ukazatelů uložených v proměnné nesoucí běhové prostředí JVM TI.

Alokace a dealokace řetězců. Pro funkce JVM TI, které vrací přes ukazatel řetězec a nebo vyžadují řetězec, je potřebné alokovat nebo dealokovat tyto řetězce pomocí následujících funkcí. Tyto funkce se neshodují s obvyklými funkcemi pro alokaci a dealokaci jazyka C.

- *Allocate()* - alokuje paměť pro zadaný řetězec o zadané velikosti.
- *Deallocate()* - pro dealokaci řetězců alokovaných pomocí předchozí funkce a dealokaci řetězců, které jsou vrácené funkcemi JVM TI, musí být použita tato funkce.

Práce se třídami:

- *GetLoadedClass()* - funkce, která zjistí počet všech tříd načtených ve virtuálním stroji a zároveň jedním z parametrů vrátí pole těchto tříd. Může být volána pouze za Live fáze.
- *GetClassSignature()* - získá signaturu třídy (její název). Může být volána ve Start nebo Live fázi.
- *AddToBootstrapClassLoaderSearch()* - funkce umožňující agentovi přidat jednu nebo více tříd do rozsahu, ve kterém bude classloader hledat potřebné třídy. Běžně se jako parametr používá název souboru s koncovkou .jar nebo složka, ve které se třídy nacházejí.

Callback funkce pro práci s vlákny. Je důležité poznamenat, že všechny funkce, které vyžadují jako parametr vlákno, umí pracovat i s hodnotou *NULL* v tomto parametru - ta znamená aktuální vlákno. Výběr některých funkcí a jejich popis:

- *GetAllThreads()* - získá všechny živá vlákna z Java aplikace, která jsou právě připojeny k virtuálnímu stroji.
- *GetThreadInfo()* - získá informace o vlákně a naplní tak strukturu *jvmtiThreadInfo*, která obsahuje mimo jiné název vlákna. Její kompletní obsah je k nalezení v dokumentaci JVM TI [3].
- *RunAgentThread()* - tato funkce slouží pro spuštění vytvořené instance třídy java.lang.Thread, kterou je možné získat pomocí JNI funkcí. Jedním z parametrů této funkce je odkaz na funkci, která implementuje funkcionalitu vzniklého vlákna.
- *GetFrameCount()* - tato funkce souvisí více se zásobníkem než s vlákny, zjišťuje totiž kolik rámců je alokovaných pro vlákno předané parametrem funkce.

Funkce pro práci s heapem:

- *IterateThroughHeap()* - umožňuje průchod všemi objekty uloženými na haldě. Užitečnou vlastností této funkce je, že pomocí jejího druhého parametru lze nastavit první filtr - lze filtrovat, zda se budou sledovat pouze objekty s tagy, nebo bez nic. Co to jsou tagy bude vysvětleno dále v textu. Třetím parametrem lze specifikovat, která třída nás zajímá - budou vybírány pouze objekty zadané třídy. Nejzajímavějším je čtvrtý parametr, do kterého je předán ukazatel na strukturu typu *jvmtiHeapCallbacks*. Tuto funkci lze volat pouze v Live fázi.
- *SetTag()* - funkce, která dovoluje přiřadit objektu, který je předán jako parametr, číselnou hodnotu, podle které lze poté objekt rozpoznat. Lze volat ve Start nebo Live fázi.
- *GetTag()* - opak předchozí funkce, zjistí tag objektu. Lze volat ve Start nebo Live fázi.

Funkce pro práci s jednotlivými objekty:

- *GetObjectSize()* - tato funkce zjistí velikost objektu, akceptuje pouze dva parametry, objekt a ukazatel na proměnnou typu jlong, přes kterou vrátí velikost objektu.

Následně funkce pro získávání informací o virtuálním stroji:

- *GetSystemProperty()* - funkce dovolující získat informace o spuštěném virtuálním stroji, jako je jeho název, výrobce, verze, proměnná classpath nebo librarypath. Informace je nutné získávat po jednom, je ale možné získat více informací najednou pomocí podobné funkce *GetSystemProperties()*.

Funkce obsluhující monitory. Význam monitorů spočívá v tom, že callback funkce musí být reentrantní a JVM TI neudržuje žádnou frontu událostí. Jediný způsob jak zajistit výlučný přístup k datům uvnitř callback funkcí je právě pomocí monitorů.

- *CreateRawMonitor()* - vytvoření monitoru.
- *RawMonitorEnter()* - vstup do monitoru.
- *RawMonitorExit()* - výstup z monitoru.

Kapitola 5

Návrh a implementace

Tato kapitola se zabývá návrhem a vlastní implementací aplikace Memory a popisem protokolu, který je použit. Aplikace je rozvržena na dvě hlavní části. Pro agenta byl zvolen programovací jazyk C. Při jeho implementaci byly využity a upraveny části agenta heapTracker [1], který pracuje s knihovnou java_crw_demo a využívá jí pro instrumentaci bajtkódu. Tato knihovna pomocí instrumentace bajkódu zaregistruje nativní metody pro sledování alokací. Druhou částí je klient, který nese grafické uživatelské rozhraní. Uživatelské rozhraní je vytvořeno pomocí jazyka C++ a Qt frameworku verze 5.0.2 [12]. Pro další čtení je nutné se ujistit v rozdílu mezi slovy klient a agent. Klient je část aplikace nesoucí grafické uživatelské rozhraní a zastávající funkci klienta. Agent je druhá část aplikace zastávající serverovou část síťové komunikace a přímo komunikující s virtuálním strojem.

5.1 Návrh aplikace

V této kapitole bude zvlášť popsán návrh agenta - druhý odstavec, a návrh grafického uživatelského rozhraní - třetí odstavec. Komunikace mezi agentem a klientem probíhá pomocí BSD socketů s využitím protokolu TCP/IP.

První částí je agent, který se připojuje k virtuálnímu stroji a získává z něho informace o právě spuštěné aplikaci. Agent zachytává alokace instancí tříd (kromě alokací objektů pomocí volání JNI funkcí) a polí primitivních datových typů na haldě, počítá je a počítá sumu jejich velikostí. Dalšími informacemi, které jsou ukládané je signatura třídy objektu, velikost objektu a název vlákna, ve kterém byl objekt alokován. Pro ukládání takto velkého množství informací a časté vyhledávání v nich, jsou použity binární vyhledávací stromy. Toto řešení velmi urychluje vyhledávání oproti například lineárně vázaným seznamům. Pro sledování zásobníků je zvoleno sledování počtu alokovaných rámců ve všech zásobnících. Tato část aplikace zároveň zastává funkci serveru. Je spuštěna a očekává připojení na implitičním nebo explicitně zadaném portu. Toto řešení je zvoleno z důvodu vytvoření rozhraní, na které by se mohl připojovat i jiný klient, stačí aby podporoval vytvořený protokol popsaný v kapitole 5.4.

Grafické uživatelské rozhraní je rozvrženo na tři pomyslné části. První obsahuje informace o virtuálním stroji, kterými jsou název virtuálního stroje, jeho výrobce a verze. Druhá obsahuje grafy. Grafy jsou tři, první zobrazuje obsazení haldy alokovanými objekty v bytech, druhý zobrazuje počet alokovaných objektů a třetí počet alokovaných rámců v zásobnících. Poslední, třetí částí je tabulka zobrazující jednotlivé objekty a informace o nich. Těmi jsou velikost objektu, signatura třídy objektu (její tvar byl popsán v kapitole 2.1.1)

a název vlákna v němž objekt vznikl. Výpis objektů je možno omezit filtry. Do filtrů je možné zadávat celé názvy, nebo pouze podřetězce, které má název vlákna nebo signatura třídy obsahovat. Zároveň lze nastavit minimální velikost sledovaných objektů.

5.2 Agent

Práce agenta je závislá na volání callback funkcí a nativních metod. První takovou, která je zavolána při připojení je *Agent_OnLoad()*, následována callback funkcemi odpovídajícími jednotlivým fázím rozhraní JVM TI (5.2.1). Provádění dalších funkcí nemá dané pořadí a záleží vždy na událostech, ke kterým dojde. Další podkapitoly se věnují vláknu zastřešujícímu síťovou komunikaci (5.2.2), sledování zásobníku (5.2.3) a funkcím pro zachytávání alokací (5.2.4).

Hlavní datovou strukturou této aplikace je struktura typu *TGlobalData*, která obsahuje veškerá důležitá data, jako je suma počtů objektů, suma velikostí objektů, počítadlo tříd a vláken, ukazatele na kořeny stromů a další. Celkově aplikace využívá tři až pět binárních vyhledávacích stromů. Jeden strom pro uložení názvů tříd a číselných identifikátorů každé třídy. Druhý strom pro uložení názvů vláken a opět číselného identifikátoru každého vlákna. Třetí strom slouží k ukládání informací o objektech, těmi jsou tag, velikost, číselný identifikátor třídy a vlákna. Zbylé dva stromy jsou využity jen v případě, kdy je zadán filtr. Jsou v nich uloženy informace o vláknech a třídách, které odpovídají zadanému filtru. Vyhledávání v těchto stromech probíhá pomocí algoritmu preorder. Klíč po vkládání položek do stromu se bitově invertuje kvůli lepšímu rozložení binárních vyhledávacích stromů.

5.2.1 Reakce na události start a inicializace virtuálního stroje

Agent se k virtuálnímu stroji připojuje při jeho startu. Po němž dochází k inicializaci datových struktur a registraci callback funkcí. V tuto chvíli také probíhá kontrola, zda virtuální stroj obsahuje správnou verzi JVM TI a pomocí JNI funkce se získá běhové prostředí *jvmtiEnv*. Dále těsně po spuštění probíhá registrace callback funkcí a aktivace reakce na ně.

Následně se v callback funkci reagující na start virtuálního stroje provede registrace nativních metod třídy HeapTracker. A nastaví se hodnota atributu *engaged* na 1. Od této chvíle je možné volat nativní metody. Právě tento atribut zajišťuje, aby se aplikace při instrumentaci nedostala do nekonečné smyčky. Protože je provedena instrumentace konstruktoru třídy java.lang.Object, došlo by při jeho prvním zavolání k vyžádání načtení třídy HeapTracker, tím by se opět volal konstruktor třídy java.lang.Object a vše by se do vyčerpání místa pro zásobník opakovalo.

Poté se v reakci na událost inicializaci virtuálního stroje vytvoří vlákno, které zajišťuje síťovou komunikaci. Tvorba vlákna probíhá pomocí JNI funkce pro vytvoření nového objektu, do kterého se předá ID konstruktoru třídy java.lang.Thread. Takto vytvořené vlákno je spuštěno pomocí JVM TI funkce *RunAgentThread()*, implementace tohoto vlákna je popsána v podkapitole 5.2.2. Po inicializaci také dochází k uložení signatur všech aktuálně načtených tříd a přiřazení tagů ke každé třídě, podle kterých jsou poté třídy přiřazovány k objektům. Názvy tříd jsou uloženy do stromu názvů tříd. V této callback funkci je také volána funkce z prostředí JVM TI, která prochází celou haldu a sleduje objekty, kterým ještě nebyl nastaven tag. To, že budou vybírány pouze objekty, které chceme, je zařízeno parametrem JVMTI_HEAP_FILTER_TAGGED této funkce. Funkce při procházení haldou všem nalezeným objektům přiřadí tag, přičte jejich velikost ke globálnímu počítadlu obsazení paměti a inkrementuje počítadlo objektů. Nastavení tagů objektům je důležité zejména

při zjišťování velikosti uvolněných objektů. Pro objekty nalezené touto funkcí není možné zjistit název vlákna, ve kterém byly alokovány. Pro tyto situace byl zvolen název vlákna *beforeInit* a byl explicitně přidán do stromu názvů vláken. Stejně tak pro některé objekty zachycené funkcí *IterateThroughHeap()* není možné zjistit název třídy, zde byl zvolen název třídy *–UNKNOWN–*.

5.2.2 Vlákno pro síťovou komunikaci

Nyní bude blíže popsáno vlákno zajišťující komunikaci. Toto vlákno ihned po spuštění vstoupí do monitoru a vystoupí z něho až na svém konci. Na uvolnění tohoto monitoru čeká agent v callback funkci, po které je již virtuální stroj ve fázi Dead. Takto je ošetřen případ, kdy je spuštěna velmi krátká Java aplikace a mohlo by se stát, že se virtuální stroj ukončí ještě před připojením klienta. Tím pádem by nemohlo dojít k odeslání informací o objektech. Toto čekání lze vypnout příslušným parametrem, protože by v některých případech mohlo být nežádoucí. Prvně je vytvořen TCP socket a je mu nastaven port, na kterém bude naslouchat. Port je možné získat dvěma způsoby. Prvním je zadání portu jako parametr agenta, druhým je spuštěním agenta bez parametru. V tom případě se automaticky zvolí implicitní port (12345). Poté, jakmile dojde k připojení agenta, je zavolána funkce, která získá informace o běžícím virtuálním stroji pomocí JVM TI funkce *GetSystemProperty()* a rovnou je odešle na klientskou část aplikace. Následuje čtení z TCP socketu ve smyčce a reakce na jednotlivé zprávy přicházející z klienta. V případě, že dojde žádost o zaslaní dat pro vykreslování grafů, přečte aplikace hodnoty z globální struktury, ve které jsou uloženy počítadla obsazení a počtu objektů. V situaci, kdy dojde v grafickém uživatelském rozhraní k nastavení filtru pro zasílání informací o jednotlivých objektech přijde přes socket řetězec definující nastavení filtru (jeho tvar bude popsán v 5.4). Agent jej zpracuje a vytvoří dva nové stromy. Do prvního se zkopírují informace o třídách ze stromu tříd a do druhého informace ze stromu vláken, ale pouze těch tříd a vláken, které odpovídají filtru. Třída nebo vlákno odpovídá filtru, pokud je v jejím názvu obsažen podřetězec zadaný do filtru. Pokud byl filtr nastaven tak, aby byly posílány informace o objektech všech tříd nebo všech vláken, nevytváří se nový strom, ale použije se jeden nebo oba existující stromy. Dále je nastavení filtru uloženo do globální datové struktury. Následuje prohledání stromu všech objektů algoritmem preorder. Pokud se při prohledávání narazí na uzel, ve kterém je objekt odpovídající nastavení filtru, jsou informace o něm odeslány socketem (formát odesílání bude popsán v 5.4). Po ukončení spojení dojde k uzavření socketů, uvolnění monitoru a celý virtuální stroj včetně agenta se může ukončit.

5.2.3 Sledování zásobníku

Ke sledování obsahu zásobníku dochází ve funkci, která je volána v závislosti na příchod požadavku od klienta na data pro vykreslení grafů. V tu chvíli je zavolána funkce, která načte všechna aktuální vlákna a u každého vlákna spočítá počet zásobníkových rámců. Tento součet je poté vrácen, přidán k dalším informacím a odeslán na klientskou stranu aplikace. Zároveň je pro každé vlákno zkontrolováno, zda jeho název existuje ve stromu vláken. Pokud neexistuje, je tento název uložen.

5.2.4 Zachytávání alokace a uvolnění objektů

Následuje popis sledování alokace objektů. Pro tuto práci jsou implementovány čtyři rozdílné funkce. První z nich je callback funkce zaregistrovaná v JVM TI funkci *IterateThrou-*

ghHeap() zmíněné výše, která je spuštěna pouze jednou za celou dobu běhu klienta. Druhou je callback funkce JVM TI, která reaguje na alokace objektů. Problémem je, že nereaguje na alokace všech objektů, ale pouze pro objekty, které jsou vytvářené tak, že jejich metody nejsou uvedeny v bajtkódu. Tato událost nereaguje na vytvoření objektů klíčovými slovem *new* nebo instrukcí *newarray*, na alokace v průběhu inicializace virtuálního stroje a alokace pomocí JNI funkcí. Pro zachycení alokací pomocí instrukcí *new* a *newarray* slouží dvě nativní metody, které jsou volány díky provedené instrumentaci bajtkódu.

V každé z těchto funkcí dojde ke zjištění signatury třídy objektu pomocí JNI funkce *GetObjectClass()*. Tato signatura se vyhledá ve stromu tříd. Pokud je nalezena uloží se její identifikátor, pokud ne, je jí přiřazen nový identifikátor a je přidána do stromu tříd. Dále je zjištěn název vlákna pomocí JVM TI funkce *GetInfoThread()* a je provedeno hledání a nebo uložení názvu tak jako v předchozím případě. Tentokrát se ale pracuje se stromem názvů vláken. Ještě je nutné zjistit velikost objektu. To se provádí JVM TI funkcí *GetObjectSize()*, pokud není velikost předána parametrem. Dále je každému objektu přiřazen unikátní tag. Unikátní tag vzniká jako bitově invertována hodnota globálního čítače pro tagy objektů, který je při každém přidání objektu inkrementován. Je typu *jlong*, a ve chvíli kdy se inkrementuje na maximální hodnotu datového typu, je mu nastavena minimální hodnota a pokračuje v inkrementaci až do hodnoty -1. Při vyčerpání celého rozsahu počítadla odešle agent klientovi zprávu o ukončení a zastaví se ukládání informací o objektech. Jakmile jsou známy veškeré důležité hodnoty, je volána funkce pro vložení objektu do stromu objektů. V případě, že je v klientovi nastaven filtr na objekty, provede se ve všech funkcích, kromě callback funkce zaregistrované ve funkci *IterateThroughHeap()*, test pro každý objekt, zda odpovídá filtrům. A pokud ano, jsou přímo z těchto funkcí odeslány informace o právě vytvořeném objektu na klientskou stranu aplikace.

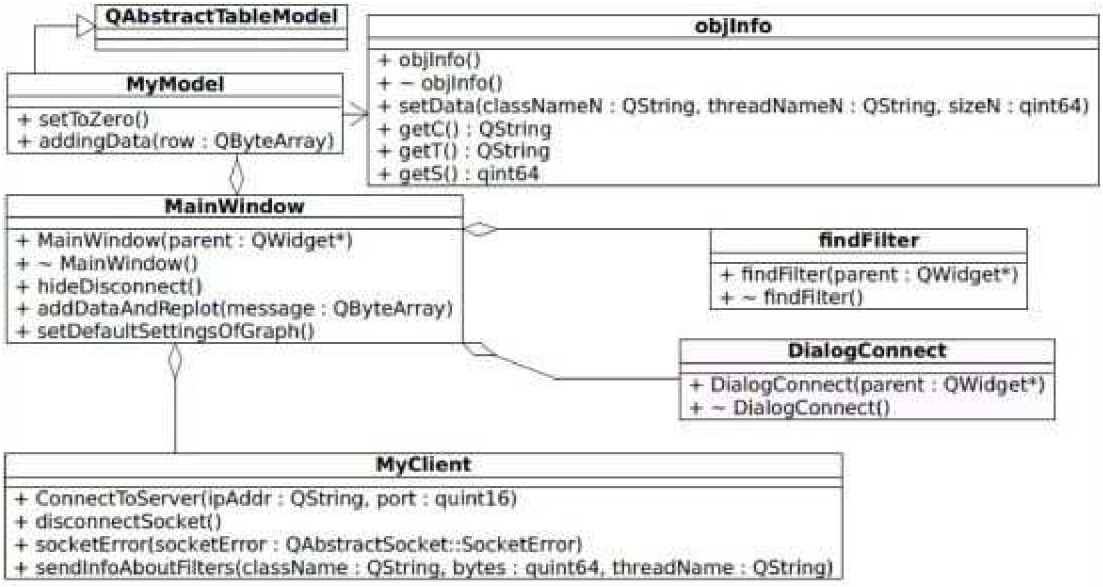
Pro sledování uvolňování objektů slouží pouze jedna callback funkce, která je volána pro každý uvolněný objekt. Jako parametr dostane tag objektu, který byl uvolněn. Její funkcionalita spočívá v dekrementaci počítadla objektů, odečtení velikosti objektu z celkové sumy uložené v globální datové struktuře. Pro tuto činnost je ale nutné najít ve stromě objektů uzel odpovídající tomuto objektu. Při této akci je nalezený uzel i uvolněn, protože nebude již dále potřeba.

Agent také sleduje začátek a konec práce správce paměti. Při jeho startu je v agentovi zakázáno odesílání informací o nových objektech, ty se po dobu jeho práce pouze ukládají do stromu objektů. Zároveň je do klientské aplikace odeslána zpráva o začátku jeho práce. Po ukončení je opět do klienta odeslána zpráva, která informuje o této události. Formát zpráv bude opět popsán v informacích o protokolu v kapitole 5.4.

5.3 Klientská část

Nejrozsáhlejší třídou aplikace je třída *MainWindow*, která stojí za celým hlavním oknem aplikace (rozložení uživatelského rozhraní definuje soubor *mainwindow.ui*, k nahlédnutí na obrázku 5.2). Její instance je vytvořená ve funkci *main*. V této třídě jsou vytvářeny a uloženy instance dalších tříd jako je třída *DialogConnect*, *MyClient*, *FindFilter* a *MyModel*, které budou popsány v dalších odstavcích. Pro lepší představu je uveden diagram tříd na obrázku 5.1. Tato třída obsluhuje jednotlivé části grafického uživatelského rozhraní. První částí jsou grafy, které jsou zobrazeny v *QTabWidget*u a jsou implementovány pomocí třídy *QCustomPlot*, která je dostupná na internetových stránkách [11]. Zde je nastavena barva křivky grafu za pomoci třídy *QPen*. Dále jsou osám grafů přiřazeny popisy. Mezi základní funkcionalitu třídy *QCustomPlot* patří možnost manipulovat s grafem pomocí myši. Z těchto

funkcí je využita možnost přibližovat a oddalovat osou y pomocí kolečka myši a posouvat grafem táhnutím myši. Dále graf využívá možnost reakce na dvojklik, při kterém se rozsah osy x změní na aktuální počet sekund sledování zvýšený o hodnotu 20. Také se zde nastavují rozsahy os. Pomocí slotu *addDataAndReplot()* jsou grafy aktualizovány. V tomto slotu se také zpracovává přijatý řetězec, který nese informaci o nových hodnotách grafu. Další částí grafického rozhraní, která je zde inicializována, je tabulka, které je nastavena možnost interaktivní změny šířky sloupců. Je jí také přiřazena instance modelu (implementován třídou *MyModel*), který zařizuje práci s daty zobrazovanými v tabulce. V této třídě je také implementováno zobrazování a skrývání tlačítek "Connect"a "Disconnect". Zároveň povolení a zakázání tlačítka "Set Filter"při připojení a odpojení klienta k serveru. Ve spodní části aplikace je přidána stavová lišta, ve které se zobrazují informace o stavu připojení.

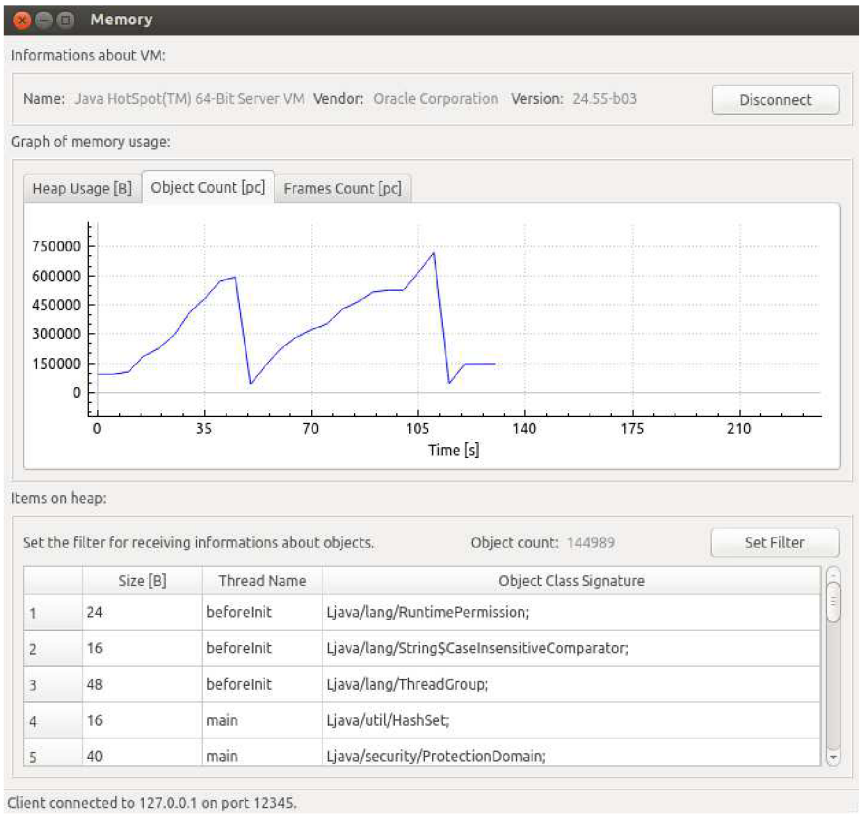


Obrázek 5.1: Diagram tříd, zachyceny jsou pouze public metody, sloty a signály jsou vynechány.

Po kliknutí na tlačítko "Set Filter"je zobrazena instance třídy *findFilter*, obrázek 5.3. V této třídě je implementovaný jeden slot, který reaguje na stisk tlačítka "OK". Jeho úkolem je pomocí regulárního výrazu zkontrolovat, zda do pole pro minimální velikost objektu byly zadány pouze číslice. Pokud ano, načtou se řetězce zadané do jednotlivých *TextEdit* položek a pomocí signálu *sendFilterData()* se odešlou do instance třídy *MyClient*, kde se z nich vytvoří zpráva (formát bude popsán v kapitole 5.4) a odešle se na server.

Třídou, která stojí za funkčností okna pro nastavení IP adresy a portu, je *DialogConnect*. Tato třída je na pozadí okna zobrazeného na obrázku 5.4. Její prioritní funkcionalitu zajišťuje slot *MakeConnection()*, který reaguje na stisk tlačítka "OK". V tu chvíli se pomocí regulárního výrazu zkontroluje, zda byla zadaná IPv4 adresa ve validním formátu, nebo bylo zadáno slovo "localhost", které je poté nahrazeno IP adresou 127.0.0.1. Dále je kontrolováno, zda je zadáno validní číslo portu. V případě, že kontroly proběhnou v pořádku dojde k emitování signálu *DoAConnection()*, na který reaguje slot *ConnectToServer()* ze třídy *MyClient*.

Zmíněná třída *MyClient* zastřešuje práci se síťovou komunikací. Je zde vytvořen socket,



Obrázek 5.2: Hlavní okno aplikace.

který je poté ve slotu *ConnectToServer()* připojen k serveru. Byla zvolena asynchronní práce se sockety. To znamená, že je vždy, když přijdou na socket data, emitován signál *readyRead()*. Na tento signál reaguje slot, který data přečte a odešle je do metody, pro zpracování zprávy přijaté od serveru. Tato metoda poté emituje signály, kterými předává přijaté informace dalším objektům. V instanci této třídy běží časovač, který se odstartuje připojením k serverové části a každých 5 sekund odesílá žádost na server. Odpovědí na tuto žádost je zpráva s hodnotami pro vykreslování grafů. Ty se tedy překreslují vždy po 5-ti sekundách. Tato třída má také implementovaný slot pro zpracování kliknutí na tlačítko "Disconnect", kterým odešle na server informaci o ukončení a ten má možnost korektně ukončit spojení. Formát zpráv bude uveden v kapitole 5.4. Za situace, kdy dochází k dealokaci objektů v Java aplikci, je přijata zpráva nesoucí tuto informaci, která zastaví přijímání nových objektů. Jakmile práce správce paměti skončí, je přijata zpráva, která iniciuje opětovné odeslání nastaveného filtru. Tudíž je na klientskou aplikaci doručen aktuální seznam objektů. Toto řešení bylo zvoleno z důvodu časové a paměťové úspory. Protože vyhledávání určitého objektu v *QVector*u je časově velmi náročné a ukládání všech informací do binárního vyhledávacího stromu je nepraktické z důvodu procházení stromu pro každý vypisovaný řádek.

V případě, že je nastaven filtr a na klientskou část aplikace přijdou informace o objektech, jsou odesílány do instance třídy *MyModel*, která dědí ze třídy *QAbstractTableModel*. V instanci této třídy dojde ke zpracování dat a jejich uložení. Informace se ukládají do instance *QVector*u, který obsahuje ukazatele na instance třídy *objInfo*. Obsah *QVector*u

Obrázek 5.3: Okno pro nastavení filtru.

Obrázek 5.4: Okno pro nastavení připojení.

je poté zobrazen v tabulce. Při doručení informací o nových objektech se tyto informace ukládají na konec vytvořeného *QVector*u a inkrementuje se počet zobrazených řádků.

Zbývá již jen třída *objInfo*. Tato třída implementuje pouze metody pro nastavení a získání třídních proměnných, kterými jsou název třídy, název vlákna, velikost a pozice.

5.4 Komunikační protokol

Veškerá síťová komunikace na aplikační vrstvě mezi klientem a agentem probíhá přes vytvořený protokol. Ve druhém odstavci se zaměříme na zprávy odesílané ze serveru na klienta, ve třetím na zprávy zasílané v opačném směru.

Každá odeslaná zpráva začíná znakem konce řádku, tedy znakem '\n'. Jednotlivé části zprávy jsou oddělovány znakem '\t'. Uvozovky u příkladů zpráv jsou vždy uvedeny pro lepší čitelnost, protokol je nepoužívá. Prvními zprávami, které klient obdrží jsou zprávy s informacemi o virtuálním stroji. Název virtuálního stroje je označen znakem '$', výrobce znakem '#' a verze virtuálního stroje znakem '!'. Výsledné zprávy tedy mohou vypadat takto, název virtuálního stroje: *"\n$Java Virtual Machine HotSpot Server 64-bit"*, výrobce: *"\n#Oracle Corporation"* a verze: *"\n!25.04-b"*. Ze serveru na klienta jsou také odesílány informace o počtu rámců, objektů a jejich celkové velikosti. Pro tuto zprávu slouží identifikační znak 'S', pro identifikaci části obsahující číselnou informaci o obsazení paměti v bajtech slouží písmeno 'U', pro počet objektů písmeno 'C' a pro počet rámců písmeno 'F'. Výsledná zpráva tedy může vypadat takto: *"\nSU356668\tC2911\tF20"*. Dále se odesílají informace o objektech. Pro identifikaci, že se jedná o informaci o objektu slouží písmeno

'O'. Poté jsou data uložena do řetězce v pořadí název třídy, název vlákna a velikost objektu. Výsledná zpráva tedy může vypadat takto: *"\nOjava/lang/String;\tmain\t24"*. Další zprávou, která může být do klienta přijata, je informace o tom, že byl spuštěn správce paměti. Tuto situaci označuje řetězec "DELETE", takže zpráva vypadá takto: *"\nDELETE"*. Opačnou událost popisuje zpráva o ukončení práce správce paměti, která iniciuje nové načtení objektů. Tato událost je rozpoznána pomocí řetězce "RESEND", zpráva tedy vypadá takto: *"\nRESEND"*. Poslední desílanou zprávou, je řetězec *"\nQUIT"*, ten je odeslán na klientskou stranu ve chvíli, kdy počítadlo objektů vyčerpá svůj rozsah.

Z klienta na server chodí pouze tři typy zpráv. První je žádost o zaslání statistických dat pro aktualizaci grafů, která má tvar: *"GET STATS"*. Druhým typem zprávy je informace o nastavení filtru objektů, ta začíná slovem *"FILTER"* a následují popořadě název třídy, minimální velikost a název vlákna, oddělené znakem '\t'. Opět následuje příklad: *"FILTER\tStr\t20\tmain"*. Zde je ještě nutné zmínit, že v případech, kdy není zadán filtr a jsou při odeslání filtru ponechána některá pole pro třídu nebo vlákno prázdná, bude odeslán místo prázdného řetězce, řetězec "$$$$"a nevyplněná velikost objektu bude automaticky nahrazena hodnotou 0. Výsledný řetězec při nezadání filtrů může vypadat takto: *"FILTER\t$$$$\t0\t$$$$"* Poslední možnou zprávou je informace o odpojení klienta a ta má tvar: *"DISC"*.

5.5 Testování aplikace

Pro testování aplikace byl vytvořen jednoduchý testovací soubor, který obsahoval dvě třídy a byla v něm vytvořena dvě vlákna. V různých místech byly vytvářeny nové objekty různých tříd. Vytváření objektu bylo pozdrženo požadavkem na vstup z terminálu. Tím se získal čas na připojení klienta a nastavení příslušných filtrů. Poté bylo sledováno, zda se objekt vytvořený po zadání vstupu na terminál objevil v seznamu objektů. Dále bylo testováno vytváření objektů v jednotlivých vláknech s nastavením filtru na právě to vlákno, ve kterém se měl objekt vytvořit.

Další testování proběhlo na volně dostupné aplikaci GiftedMotion verze 1.23 (ke stažení na webových stránkách [14]). Byla na ní otestována funkčnost reakce na správce paměti a filtrování při větším množství objektů. První testování probíhalo pomocí nástroje VisualVM, popsaného v kapitole 3.1.1. Zde proběhlo pouze porovnání grafů obsazení haldy. Při startu aplikace byly grafy na řádově stejných hodnotách (přibližně 6MB). Přesnou hodnotu nelze zjistit, protože grafické uživatelské rozhraní výše jmenované aplikace je velmi náchylné na změnu polohy kurzoru myši nebo překrytí oknem, při čemž se vytváří nové objekty, tudíž nelze zajistit naprosto stejné podmínky. Dále se graf obsazení haldy lišil, což je způsobeno periodickým spouštěním správce paměti ve VisualVM. Tato skutečnost byla ověřena sledováním jiné aplikace pomocí tohoto nástroje, kde byl správce paměti spouštěn v přibližně stejných periodách jako ve výše zmíněné aplikaci. Tento test byl proveden i na jiných nástrojích, kde se výsledky mezi touto aplikací a i mezi nástroji samotnými velmi lišily. Po konzultaci s odborným vedoucím jsme došli k závěru, že rozdílné výsledky budou pravděpodobně způsobeny sledováním kompletního obsazení haldy a ne pouze sledováním alokovaných objektů. Na haldě se mohou objevit i různé další informace tak, jak bylo popsáno v podkapitolách kapitoly o správě paměti virtuálního stroje 2.2. Dalším důvodem může být, že tato aplikace nesleduje alokace objektů prováděné pomocí rozhraní JNI.

Další aplikací, která byla použita pro otestování správně funkčnosti sledování vytváření objektů, byl nástroj JProfiler [15]. Tato aplikace zvládá zobrazení počtu aktuálně alokovaných objektů. Testování bylo opět prováděno na aplikaci GiftedMotion verze 1.23. Celý

test probíhal tak, že byla aplikace sledovaná JProfilerem a v aplikaci byla provedena určitá činnost, jako například výběr barvy, stisk určitého tlačítka a podobně. Stejná činnost byla provedena při sledování vytvořenou aplikací Memory a výsledek byl řádově porovnáván s prvním výsledkem. Například pro výběr barvy bylo v obou případech alokováno přibližně 2000 nových objektů. Toto číslo nelze opět určit přesně, protože i zde stačí malý pohyb ukazatelem myši a ihned dochází k alokaci dalších objektů. Řádově si výsledná čísla odpovídala.

Testování na zmíněné aplikaci také ukázalo jak velký vliv má vytvořený agent na rychlost aplikace. Bylo zjištěno, že zpomalení aplikace je viditelné a zejména práce správce paměti aplikaci může na krátkou chvíli i pozastavit. Toto zpomalení bylo po konzultaci s odborným vedoucím označeno jako odpovídající způsobu sledování.

Kapitola 6

Závěr

Cílem této práce bylo nastudovat rozhraní JVM TI a vytvořit aplikaci s grafickým uživatelským rozhraním, která bude sledovat obsazení haldy a zásobníku. Pro dosažení cíle bylo nutné nastudovat rozhraní JVM TI, které je spolu s funkcemi důležitými pro tuto práci popsáno v kapitole 4. Pro snadnější pochopení některých částí aplikace byly v kapitole 2 popsány základní části paměťového struktury virtuálního stoje. Výsledná aplikace nesoucí název Memory je popsána v kapitole 5. V kapitole 3 jsou zmíněny dva existující nástroje a provedeno porovnání s nástrojem implementovaným.

Cílem implementační bylo vytvořit takový nástroj, který bude sledovat vytváření objektů a zobrazovat množství zabrané paměti těmito objekty a zároveň poskytne informace pro jednodušší lokalizaci vytváření problémových objektů s možností filtrovat sledované objekty. Tento cíl byl splněn návrhem a implementací aplikace Memory, která poskytuje potřebné informace ve formě grafů zobrazujících obsazení paměti, počet alokovaných objektů a počet rámců. Dále umožňuje filtrovat objekty a jejich výpis zobrazovat do přehledné tabulky. Pro lepší lokalizaci místa, kde dochází k vytváření problémových objektů poskytuje aplikace informaci o vlákně, ve kterém byl objekt vytvořen. Aplikace, taková jaká je, má velký potenciál pro budoucí rozšiřování. Mezi plánované změny patří zejména zjištění metody, ve které došlo k alokaci, čímž se ještě zpřesní místo, které iniciuje alokace. Dále možnost ručního spuštění správce paměti, dalším možným rozšířením je implementace výpisu procesorového času stráveného v jednotlivých vláknech. Rozšíření se budou týkat také grafického uživatelského rozhraní. Zde je plánována možnost nastavení intervalu obnovy grafů, možnost fulltextového vyhledávání v seznamu objektů, řazení seznamu objektů a také implementace ovládacích prvků pro výše zmíněné vylepšení.

Literatura

[1] JVM TI Demonstration Code [online]. http://acm2013.cct.lsu.edu/localdoc/java/6-jdk/demo/jvmti/, 2013 [cit. 2014-04-22].

[2] TIOBE Software: Tiobe Index [online]. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, [cit. 2014-03-20].

[3] JVM(TM) Tool Interface 1.2.1 [online]. http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html, [cit. 2014-03-22].

[4] Java SE Documentation [online]. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html, [cit. 2014-03-23].

[5] Server-Class machine detection [online]. http://docs.oracle.com/javase/7/docs/technotes/guides/vm/server-class.html, [cit. 2014-03-23].

[6] Apache Commons BCEL [online]. http://commons.apache.org/proper/commons-bcel/, [cit. 2014-04-01].

[7] ASM - Home page [online]. http://asm.ow2.org/, [cit. 2014-04-01].

[8] Java SE 7 Java Native Interface - Related APIs and Developer Guides [online]. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html, [cit. 2014-04-02].

[9] Home – Project Kenai [online]. http://visualvm.java.net/, [cit. 2014-04-22].

[10] Java Virtual Machine Debug Interface [online]. http://www.oracle.com/technetwork/java/javase/jvmdi-spec-135507.html, [cit. 2014-04-22].

[11] Qt Plotting Widget. Introduction [online]. http://www.qcustomplot.com/, [cit. 2014-04-28].

[12] Qt Project [online]. http://qt-project.org/, [cit. 2014-04-28].

[13] Sysprof - Statistical, system-wide Profiler for Linux [online]. http://sysprof.com/, [cit. 2014-05-08].

[14] GiftedMotion — Onyxbits [online]. http://www.onyxbits.de/giftedmotion, [cit. 2014-05-15].

[15] Java Profiler - JProfiler [online]. https://www.ej-technologies.com/products/jprofiler/overview.html, [cit. 2014-05-17].

[16] Thermostat [online]. http://icedtea.classpath.org/thermostat/, [cit. 2014-05-8].

[17] Chiba, S.: Javassist [online]. http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/, 2006-06-03 [cit. 2014-04-02].

[18] Lindholm, T.; Yellin, F.: *The Java virtual machine specification Second Edition.* Boston: Addison-Wesley, 2003, ISBN 0-201-43294-3, xv 473 s.

[19] O'Hair, K. T.: The JVMPI transition to JVMTI [online]. http://www.oracle.com/technetwork/articles/javase/jvmpitransition-138768.html, 2004-07-01 [cit. 2014-04-22].

[20] O'Hair, K. T.: Bytecode instrumentation (BCI) — Java.net [online]. https://weblogs.java.net/blog/kellyohair/archive/2005/05/bytecode_instru.html, 2005-05-13 [cit. 2014-04-01].

[21] Richards, N.: Hello World! in 70 Bytes [online]. http://www2.sys-con.com/itsg/virtualcd/java/archives/0707/richards/index.html, 2004 [cit. 2014-05-18].

[22] Stahl, H.: Moving to OpenJDK as the official Java SE 7 Reference Implementation (Henrik on Java) [online]. https://blogs.oracle.com/henrik/entry/moving_to_openjdk_as_the, 2011-06-08 [cit. 2014-03-20].

[23] Tišnovský, P.: Pohled pod kapotu JVM (1.část - prohlížení a modifikace bajtkódu) [online]. http://www.root.cz/clanky/pohled-pod-kapotu-jvm-1-cast-prohlizeni-a-modifikace-bajtkodu/#k05, 2011-12-13 [cit. 2014-03-26].