



AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

Praca dyplomowa magisterska

Debuggowanie aplikacji komunikujących się asynchronicznie oparte o historię komunikatów.

History-based approach for debugging applications using asynchronous communication.

Autor:

Krzysztof Romanowski

Kierunek studiów:

Informatyka

Opiekun pracy:

dr hab. Arkadiusz Janik, dr inż

Kraków, 2015

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję ... tu ciąg dalszych podziękowań np. dla promotora, żony, sąsiada itp.

Spis treści

1. Wprowadzenie	7
1.1. Motywacja	7
1.2. Cele pracy	7
2. Asynchroniczny debugger	9
2.1. Techniki i narzędzia służących do debuggowania	9
2.1.1. Wyjątki i ich analiza	9
2.1.2. Tracing	10
2.1.3. Instrumentacja kodu	10
2.1.4. Debugger	10
2.2. debugowanie na JVM	11
2.2.1. JVM: wątki	11
2.2.2. Architektura JDBI	11
2.2.3. Narzędzia udostępniane przez JDBI	12
2.3. Zasada działania	12
2.3.1. Asynchroniczna historia wywołań	12
2.3.2. Historia komunikatów	12
2.3.3. Budowanie historii komunikatów	12
2.3.4. Asynchroniczne debuggowanie aplikacji aktorowych: Akka	13
3. Aplikacje testowe	15
3.1. Opis technologii	15
3.1.1. Model aktora	15
3.1.2. Framework Akka	15
3.1.3. Integracja z Asynchronicznym debuggerem	15
3.2. Aplikacja 1	15
3.2.1. Motywacja	15
3.2.2. Opis działania	15
3.2.3. Opis debuggowania	15

3.3.	Aplikacja 2.....	15
3.3.1.	Motywacja.....	15
3.3.2.	Opis działania	16
3.3.3.	Opis debuggowania	16
4.	Sposoby tworzenie historii komunikatów	17
4.1.	Dwa etapy: zbieranie i wysyłanie danych	17
4.2.	Metody zbierania danych.....	17
4.2.1.	Plain JDI.....	17
4.2.2.	JVM TI: Java Agent	17
4.2.3.	JDI: instrumentacja kodu	17
4.3.	Metody przesyłania danych	17
4.3.1.	Plain JDI.....	17
4.3.2.	Plain JDI: lazy mode	17
4.3.3.	Filesystem	17
4.3.4.	Socets	17
4.4.	Testowane złożenia.....	17
5.	Wyniki oraz ich a analiza	19
5.1.	Aplikacja 1	19
5.1.1.	Test 1	19
5.1.2.	Test 2	19
5.1.3.	Test 3	19
5.1.4.	Analiza	19
5.2.	Aplikacja 2.....	19
5.2.1.	Test 1	19
5.2.2.	Test 2	19
5.2.3.	Test 3	19
5.2.4.	Analiza	19
5.3.	Zestawienie zbiorcze	19
5.3.1.	Test 1	19
5.3.2.	Test 2	19
5.3.3.	Test 3	19
5.3.4.	Analiza	19
6.	Analiza oraz wnioski.....	21
6.1.	Dalsze możliwości rozwoju	21

1. Wprowadzenie

1.1. Motywacja

Podczas tworzenia aplikacji asynchronicznych twórcy wielokrotnie napotykają ograniczenia narzędzi, które nie są przystosowane do pracy z tą klasą aplikacji. Podstawowe techniki, takie jak analiza wyjątków czy klasyczne debuggery, przeważnie nie dają wystarczających informacji o naturze problemów. Iulian Dragos w swojej prezentacji [Dra14a] przedstawił koncepcję asynchronicznego debuggera, przeznaczonego do debugowania aplikacji stworzonych przy wykorzystaniu technologii Akka oraz mechanizmu Feature'ów z języka Scala. Po wysłuchaniu prelekcji uznałem, że przedstawiona przez Dragos'a koncepcja jest dobra, jednakże wykorzystane tutaj sposoby persystencji komunikatów będą miały zbyt duży wpływ na działania debuggowanej aplikacji.

1.2. Cele pracy

Celem poniższej pracy jest zbadanie możliwości oraz efektywności debugowania aplikacji komunikujących się asynchronicznie w oparciu o historię komunikatów. Głównym obszarem zainteresowań pracy będą pomiar oraz zmniejszenie narzutu sposobu persystowania i analizy komunikatów na czas wykonywania poszczególnych części aplikacji. Zamierzam zaimplementować i przetestować różne podejścia, a następnie zestawić wyniki z ograniczeniami danej metody. Ponieważ jednak przedmiotem tej pracy nie jest stworzenie asynchronicznego debuggera, wykorzystam pracę Iuliana [Dra14b]. Zaimplementowany debugger jest częścią ScalaIDE – IDE dedykowanego Scali. Przetestuję wydajność, wykorzystując aplikacje napisane w frameworku Akka – najpopularniejszej technologii do pisania aplikacji asynchronicznych, opartych o wymianę komunikatów w ekosystemie Scali.

2. Asynchroniczny debugger

W tym rozdziale zamierzam przedstawić zasadę działania wykorzystanego asynchronicznego debugera na tle narzędzi oferowanych przez JVM. Zamierzam nakreślić problemy oraz sposoby ich rozwiązywania oraz pokazać dlaczego sposób persystencji komunikatów jest kluczowy dla minimalizacji wpływu debugera na debugowaną aplikację.

2.1. Techniki i narzędzia służących do debuggowania

W tym podrozdziale przedstawię techniki oraz narzędzia służące analizie i debugowaniu aplikacji które udostępnia ekosystem JVM. Zamierzam również pokazać dlaczego narzędzia te są niewystarczające do debugowania programów asynchronicznych.

2.1.1. Wyjątki i ich analiza

Duża część błędów w programie kończy się rzuceniem wyjątku. Poza informacją o rodzaju wyjątku oraz komunikatem błędu JVM udostępnia nam stack trace.

A Stack Trace is a list of method calls from the point when the application was started to the point where the exception was thrown. The most recent method calls are at the top. [Wik]

Stack trace jest niezwykle przydatny przy śledzeniu wywołania gdyż błędy często biorą się ze złego kontekstu wywołania metody (np. wykorzystanie niewłaściwej metody czy nieskończona rekursja). Dzięki jego analizie jesteśmy w stanie odtworzyć historie wywołań oraz znaleźć kontekst w którym wyjątek został rzucony. Co więcej analiza stack trace'ów pozwala nam na odtworzenie w pewnych sytuacjach parametrów metody czy wartości pól w obiektach. Niestety stack trace udostępnia nam tylko stos wywołań metod w pojedynczym wątku. Logika (potok logiki) w aplikacjach asynchronicznych jest przeważnie rozrzucony po wielu wątkach. Przykładowy stack trace z aplikacji w frameworku Akka:

TODO wyjątek

Załączony listing daje nam informacje o stosie wywołań od odebrania ostatniej wiadomości. W przeważającej liczbie przypadków są to informacje niewystarczające gdyż przeważnie nie pozwalają nam to na odtworzenie cyklu wiadomości które doprowadziły do powstania wyjątku.

2.1.2. Tracing

Jest to dość proste i naiwne podejście do debuggowania polegające na logowaniu punktów w aplikacji (na standardowe wyjście lub w inne miejsce, np. do pliku) i późniejszej analizie (przeważnie post mortem). Podstawową wadą tego rozwiązania jest konieczność ponownej kompilacji kodu dla każdej fali dodawania kolejnych logów. Po poprawieniu błędów programiści często zapominają o usunięciu wszystkich logowanych komunikatów co zaśmieca logi czy kod źródłowy a nawet potrafi być przyczyną problemów. Rozwiązanie mimo swojej prostoty ma szereg zalet. Nie wymaga żadnych specjalistycznych narzędzi ani umiejętności. Posiada także całkiem mały wpływ na wykonanie aplikacji lecz w przypadku aplikacji wielowątkowych wymaga zastosowania bardziej zaawansowanych technik (takich jak dedykowany plik dla każdego wątku). Mimo swej prostoty rozwiązanie to jest szeroko stosowane przez programistów. W przypadku aplikacji asynchronicznych często jest jednym dostępnym sposobem debuggowania.

2.1.3. Instrumentacja kodu

TODO Pisać o tym??? Instrumentacja kodu jest przeważnie wykorzystywana do zautomatyzowanego trawingu.

2.1.4. Debugger

Debugger jest programem do dynamicznej analizy wykonania innych programów.

With the magic of including debugging symbols in the executable, the debugger gives the illusion of executing the program line by line of source code, instead of instruction by instruction of compiled machine code [NSM08]

Odwzorowanie kodu maszynowego na kod źródłowy jest skomplikowanym zadaniem, zwłaszcza dla języków wysokopoziomowych. Współczesne debuggery opierają swoje działanie o mechanizm breakpointów czyli definicji miejsc w których wykonanie programu powinno zostać wstrzymane (czasem tylko w celu zalogowania komunikatu - debugger może być także wykorzystany do trawingu). Po wstrzymaniu działania użytkownik może przeglądać wartości zmiennych, pól czy w bardziej zaawansowanych debuggerach ewaluować wyrażenia. Debugger pozwala także na sterowanie wykonaniem programu. Poza komendami które nie ingerują w normalne wykonanie programu (Step in, Step out, Step into itp.) pozwala także na np. ponowne wykonanie funkcji (drop frame).

Niestety debuggery dostępne dla aplikacji działających na JVM mają tę samą wadę co analiza wyjątków. Są skuteczne tylko do analizy logiki w ramach pojedynczego wątku. Jest to spowodowane architekturą JVM oraz udostępnianymi przez nią narzędziami.

2.2. debugowanie na JVM

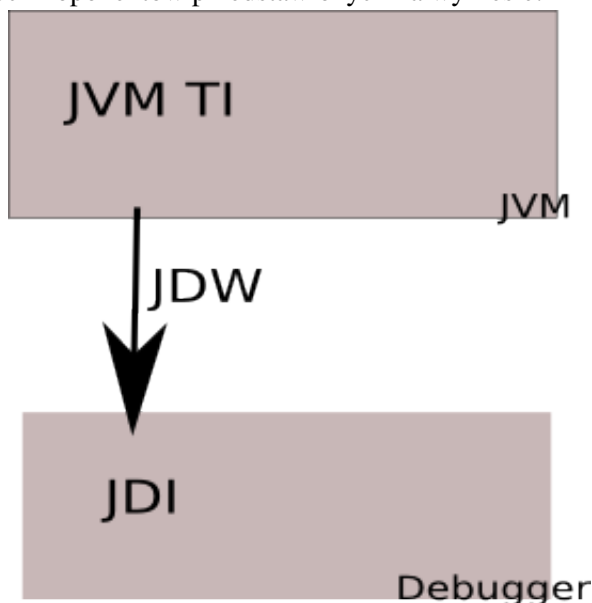
W tym podrozdziale zamierzam przedstawić JVM w kontekście debugowania aplikacji asynchronicznych. Zmierzam pokazać że potok wywoływania kodu maszynowego różni się koncepcyjnie od potoku przetwarzania komunikatów przez aplikację asynchroniczną. Zamierzam także przestawić właściwości Java Platform Debugger Architecture na których opiera się działania Asynchronicznego Debuggera.

2.2.1. JVM: wątki

TODO - opisać stosowość i przypisanie do wątku. Pokazać dlaczego rozdział między logiką opartą na komunikatach a na wywołaniach metody

2.2.2. Architektura JDPA

Jako że każdy debugger jest oddzielną aplikacją, Java Platform Debugger Architecture składa się z trzech koponentów przedstawionych na wykresie:



1. JVM TI: instrumentacja JVM

Java™ Virtual Machine Tool Interface służy instrumentacji debuggej JVM.

The JVMTM Tool Interface (JVM TI) is a programming interface used by development and monitoring tools. It provides both a way to inspect the state and to control the execution of applications running in the Java™ virtual machine (VM). [?14]

Nie jest to tylko narzędzie do debugowania ale może być także podstawą dla tworzenia innych narzędzi takich jak np. profilery.

Aspektem który nas najbardziej interesuje jest niskopoziomowa obsługa breakpointów. Mechanizm działania jest podzielony na 2 etapy:

1. Deklaracja breakpointu 2. Obsługa eventu wygenerowanego przez breakpoint (tu następuje decyzja czy zatrzymać wykonywanie czy nie)

JVM TI pozwala nam także na dostęp do zmiennych lokalnych oraz parametrów metod. Pozwoli nam to na przechwytywanie przesłanych komunikatów bez udziału debbugera a co za tym idzie przyspieszenie całego procesu, gdyż komunikacja między dwiema aplikacjami jest kosztowna.

2. JDWP: protokół transportowy

Java Debug Wire Protocol: protokół transportowy wykorzystywany do komunikacji między maszyną wirtualną (JVM TI) a debuggerem (lub innym programem takim jak profiler). W asynchronicznym debuggerze korzystamy z implementacji dostarczanej przez JDT. W większości przypadków komunikacja jest schowana za warstwą abstrakcji implementując JDI. Więcej informacji możemy znaleźć w specyfikacji[?]

3. JDI: wysokopoziomowe API

Wysokopoziomowe API służące do tworzenia debuggerów i innych narzędzi programistycznych (np. profilerów). Podobnie jak JDWP jest zaimplementowany przez JDT i stanowi wysokopoziomowe API które jest podstawą debuggera. Zawiera metody pozwalające na: - deklarowanie breakpointów - obsługę eventów (np. zatrzymanie na breakpoint'cie) - dostęp do obiektów (parametrów metod, zmiennych lokalnych czy statycznych pól), - wywoływanie metod i tworzenie nowych instancji - sterowanie wywołaniem aplikacji

Większość wymienionych metod zostanie wykorzystana do stworzenia asynchronicznego debuggera.

2.2.3. Narzędzia udostępniane przez JDPA

TODO: opisać breakpoint, code evaluation itp. - to co będziemy wykorzystywać.

2.3. Zasada działania

W tym podrozdziale zamierzam przedstawić zasadę działania asynchronicznego debuggera oraz określić miejsca które będą przedmiotem tej pracy.

2.3.1. Asynchroniczna historia wywołań

TODO

2.3.2. Historia komunikatów

TODO

2.3.3. Budowanie historii komunikatów

TODO

2.3.4. Asynchroniczne debuggowanie aplikacji aktorowych: Akka

3. Aplikacje testowe

W tym rozdziale zamierzam przedstawić aplikacje które posłużą do testowania debuggera. Opiszę technologię oraz algorytmy w nich zastosowane.

3.1. Opis technologii

3.1.1. Model aktora

3.1.2. Framework Akka

3.1.3. Integracja z Asynchronicznym debuggerem

3.2. Aplikacja 1

3.2.1. Motywacja

Co chcemy zbadać, po co itp.

3.2.2. Opis działania

Opis algorytmu, architektury itp.

3.2.3. Opis debuggowania

W jaki sposób aplikacja będzie debuggowana, opis breakpointów, testów wydajności oraz walidacji wyników.

3.3. Aplikacja 2

3.3.1. Motywacja

Co chcemy zbadać, po co itp.

3.3.2. Opis działania

Opis algorytmu, architektury itp.

3.3.3. Opis debuggowania

W jaki sposób aplikacja będzie debuggowana, opis breakpointów, testów wydajności oraz walidacji wyników.

4. Sposoby tworzenie historii komunikatów

W tym rozdziale zamierzam przedstawić sposoby tworzenia historii komunikatów. Zamierzam podzielić ten proces na dwa etapy i przedstawić sposoby ich implementacji. W ostatnim podrozdziale zamierzam przedstawić testowane sposoby (złożenia).

4.1. Dwa etapy: zbieranie i wysyłanie danych

4.2. Metody zbierania danych

4.2.1. Plain JDI

4.2.2. JVM TI: Java Agent

4.2.3. JDI: instrumentacja kodu

4.3. Metody przesyłania danych

4.3.1. Plain JDI

4.3.2. Plain JDI: lazy mode

4.3.3. Filesystem

4.3.4. Sockets

4.4. Testowane złożenia

TODO: wyjdzie podczas implementacji

5. Wyniki oraz ich analiza

5.1. Aplikacja 1

5.1.1. Test 1

5.1.2. Test 2

5.1.3. Test 3

5.1.4. Analiza

5.2. Aplikacja 2

5.2.1. Test 1

5.2.2. Test 2

5.2.3. Test 3

5.2.4. Analiza

5.3. Zestawienie zbiorcze

5.3.1. Test 1

5.3.2. Test 2

5.3.3. Test 3

5.3.4. Analiza

6. Analiza oraz wnioski

TODO: w zależności co wyjdzie

6.1. Dalsze możliwości rozwoju

Bibliografia

- [?14] Oracle ? Java™ virtual machine tool interface (jvm ti). <http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html>, 2014.
- [Dra14a] Iulian Dragos. Rethinking the debugger. <http://scalacamp.pl/data/async-debugger-slides/index.html#/>, 2014.
- [Dra14b] Iulian Dragos. Rethinking the debugger. <http://scalacamp.pl/data/async-debugger-slides/index.html#/>, 2014.
- [NSM08] Peter Jay Salzman Norman S. Matloff. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, 2008.
- [Wik] Wikibooks. *Java Programming*. Wikibooks ?, ???