AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

Praca dyplomowa magisterska

Debuggowanie aplikacji kominukujących 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

1.1. Motywacja

Podczas tworzenia aplikacji asychronicznych 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 koncepję asychronicznego 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 efektywaności debugowania aplikacji komunikujących się asynchronicznie w oparciu o historię komunikatów. Głównym obszarem zainteresowań pracy będą pomiar oraz zmieniejszenie 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]. Zaimpelementowany debugger jest częścią ScalaIDE – IDE dedykowanego Scali. Przetestuję wydajność, wykorzystując aplikacje napisane w frameworku Akka – najpopularniejszej technologii do pisania aplikacji asychronicznych, opartych o wymianę komunikatów w ekosystemie Scali.

2. Asychroniczny debugger

W tym rozdziale zamierzam opisać techniki służące debuggowaniu oraz pokazać dlaczego mogą być niewsytarczjące do debuggowania aplikacji asychronicznych. Przedstawie także zasadę działania wykorzystanego asynchronicznego debuggera oraz pokaże dlaczego efektywaność persystencji komunikatów jest kluczowa.

2.1. Aplikacje asynchroniczne oraz ich problemy

W tym podrozdziale scharakteryzuje asplikacje asychroniczne oraz przedstawie dodatkowe problemy które się w nich pojawiają.

2.1.1. Aplikacje komunikujące się asynchronicznie

TODO definicja

2.1.2. Problemy

Vipindeep i Pankaj w swoim artykule [VV14] przedstawili powszechne problemy które pojawiają się podczas tworzenia applikacji. Większość z nich daje sie łatwo rowiązać przy zwałaszcza przy wykorzystaniu narzędzi takich jak debbugger. W przypaku prostycj błędów aktorytmicznych analiza kodu jest wystarczającym kontekstem, lecz w przypadku skomplikowanych i dynamicznych aplikacji musimy posiłkować się wsparciem narzędzi. Przykładowo problem podwójnego zwalniania zasobów możemy względie prosto rozwiązać korzystając z debuggara. Zatrzymując aplikację w miejscach zwalniania problematycznego zasobu i analizjąc historię oraz kotekst wywołania możemy wyeliminować błędny kod.

Powyższy przykład zwraca uwagę na wagę historii wywołań jako kluczowej do rowziązania wielu problemów. Żeby znaleść przyczyne powstania błędu musimy wiedzieć jak do niego doszło. Cowięcej błedy w programie manifestują się daleko od miejsca w którym zostały popełnione. Przykładowo błąd w konifugracji programu podczas startu może obiwaić się dopiero przy próbie użycia danej funkcjonalności.

2.1.3. Histia wywołań, wątki a aplikacje asychroniczne

opisać dlaczego historia nie jest prosta tu

2.1.4. Asychrnoniczność oraz czas jako dodatkowy czynnik

Applikacje asychrnoniczne są szczególnie narażone w błedy opisane przez Vipindeep i Pankaj w podroziale 2.3. opisali klasę problemów szczególnie ważnych dla aplikacji asychrnonicznych. Programiści tworząc aplikację asychrnoniczne często pomijają fakt że dany fragment kodu w pewnych sytuacjach może wykonywać się znacząco wolniej przez co spowoduje że poprawne działanie zostanie zakłucone i np. inny komunikat zepsuje stan (TODO: corrupt + lepsza ilustracja). Przedstawiona klasa problemów wymaga od narzędzi jaknajmniejszej ingerencji w debuggowaną applikacje.

2.1.5. Klasy problemów w kontekscie aplikacji asychronicznych

Podsuowując powyższe podrodziały możemy podzielić problemy na 3 klasy:

1. Problemy wymagające konteksu historycznego od momentu odebrania wiadmości
2. Problemy wymagające historii wysłanych wiadomości
3. Problemy wymagjące minimalnej ingerencji w debuggowana aplikację.

2.2. 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 asychronicznych.

2.2.1. Histria działania jako klucz do rozwiązania problemu

2.2.2. Historia wywołania a wątki

//TODO opisac mechanizm analizy stosu wywołań. Pokazać że poza stosem nie mamy innych informacji

2.2.3. 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 rekurusja). 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 asychronicznych jest przeważnie rozrzucony po wielu wątkach. Przykładowy stack trace z applikacji w fameworku 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 pozawala nam to na odtworzenie cyklu wiadomości które doprowadziły do powstania wyjątku.

2.2.4. Tracing

Jest to dość proste i naiwne podejście do debugowania polegające logowaniu tekstowych komunikatów zawierających informacje o stanie aplikacji w danym punkcie oraz późniejszej analizie (przeważnie post mortem). Podstawową wadą tego rozwiązania jest konieczność ponownej kompilacji kodu dla każdej fali dodawania kolejnych logów. Często informacje potrzebne do diagnozy problemu nie dają się łatwo zalogować czy to z powodu swojego rozmiary czy struktury. 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 prosty 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).

Umiejętnie stosowany tracing pozwala nam śledzić historię działania aplijakacji. Niestety jest to metoda dość prymitywna i w przypdaku bardziej zaawansowowanych problemów jest niewystarczająca.

Mimo swej prostoty rozwiązanie to jest szeroko stosowane przez programistów. W przypadku aplikacji asychronicznych często jest jednym dostępnym sposobem debugowania.

2.2.5. Instrumentacja kodu

TODO Pisać o tym??? Instrumentacja kodu jest przeważnie wykorzystywana do zautomatyowanego tracingu.

2.2.6. 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 tracing'u). 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. pownowe wykonanie funkcji (drop frame). Debugger jest doskonałym narzędziem zarówno do naprawiania problemów jak i do zapoznania się z instniejącym kodem.

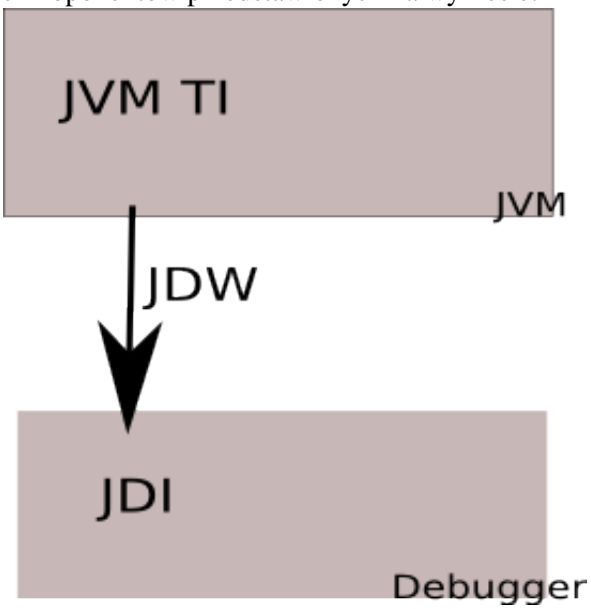
Niestety debuggery dostępne dla aplikacji działających na JVM posiadają to samo ograniczenie co analiza wyjątków. Są skuteczne tylko do analizy logiki w ramach pojedynczego wątku. Jest to spowodowane architekturą JVM oraz udostępnianymi przez nią narzędziami.

2.3. Java Debug Platform Achitecture

W tym podroździale zamierzam przedstawić JDPA którę jest podstawowym narzędziem do budowy debuggerów dla aplikacji działących na JVM. Zamierzam również przedstawić narzędzia i właściwości które zamierzam wykorzystać w dalszej części swojej pracy.

2.3.1. Architektura

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



2.3.1.1. 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 JavaTM virtual machine (VM). [?14]

Nie jest to tylko narzędzie do debugowania ale może być także podstawą dla tworzenia innych narzedzi 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.

2.3.1.2. JDWP: protokół transportowy

Java Debug Wire Protocol: protokół transportowy wykożystywany do komunikacji między maszyną wirtualną (JVM TI) a debuggerem (lub innym programem takim jak profiler). W asychronicznym debuggerze korzystamy z imlementacji dostarczanej przez Java Debug T. W wiekszości przypadków komunikacja jest schowana za warstwą abstrakcji implementują JDI. Więcej informacji możemy znaleść w specyfikacji[**?**]

2.3.1.3. JDI: wysokopoziomowe API

Wysokopoziomowe API służace do tworzenia debuggerów i innych narzędzi programistycych (np. profilerów). JDI jest interfesjem udostępnianym przez implementacje JDWP i służy do wysokopoziomowej komunikacji z debuggowaną maszyną wirtualną. Podobnie jak JDWP jest zaimplementowany przez JDT i będzie stanowić podstawę dla omawianego asynchronicznego debuggera. Zawiera metody pozwalające na:

1. deklarowanie breakpointów
2. obsługę eventów (np. zatrzymanie na breakpoint'cie)
3. dostęp do obiektów (parametrów metod, zmiennych lokalnych czy statycnych pól),
4. wywoływanie metod i tworzenie nowych instancji

5. sterowanie wywołaniem aplikacji

Większość wymienionych metod zostanie wykorzystana do stworzenia asychronicznego debugera.

2.4. Zasada dzialania

W tym podrozdziale zamierzam przedstawić zasadedziałania asychronicznego debuggera oraz nakreślić miejsca które będą przedmiotem tej pracy.

2.4.1. Asychrnoniczna historia wywołań

TODO

2.4.2. Historia komunikatów

TODO

2.4.3. Budowanie historii komunikatów

TODO

2.4.4. Asychroniczne debuggowanie aplikacji aktorowych: Akka

2.5. 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 asynchroniczne. Zamierzam także przestawić właściwości Java Platform Debugger Architecture na których opiera się działania Asychronicznego Debuggera.

2.5.1. JVM: wątki

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

3. Aplikacje testowe

W tym rozdiale 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 Asychronicznym debuggerem

3.2. Aplikacja 1

3.2.1. Motywacja

Co chcemy zbadać, po co itp.

3.2.2. Opis diałania

Opis algorytmu, architektury itp.

3.2.3. Opis debuggowania

W jaki sposób aplikacja bedzie 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 diałania

Opis algorytmu, architektury itp.

3.3.3. Opis debuggowania

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

4. Sposoby tworzenie historii komunikatów

W tym rozdiale zamierzem przedstawić sposoby tworzenia historii komunikatów. Zamierzam podzielić ten proces na dwa etapy i przedstawić sposoby ich implementacji. W ostatnim podrozdiale 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. Socets

4.4. Testowane złożenia

TODO: wyjdzie podczas implementacji

5. Wyniki oraz ich a analiza

5.1. Aplickacja 1

5.1.1. Test 1

5.1.2. Test 2

5.1.3. Test 3

5.1.4. Analiza

5.2. Aplickacja 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 możlowości 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.

[VV14] Pankaj Jalote Vipindeep V. List of common bugs and programming practices to avoid them. `some-article`, 2014.

[Wik] Wikibooks. *Java Programming*. Wikibooks ?, ???