STUDIA INFORMATICA 2012
Volume 33 Number 2B (106)

Łukasz WYCIŚLIK, Maciej MILCZAREK
Politechnika Śląska, Instytut Informatyki

ZASTOSOWANIE TECHNOLOGII OSGI W IMPLEMENTACJI SYSTEMU REALIZACJI ZADAŃ ROZPROSZONYCH W ARCHITEKTURZE NADZORCA-WYKONAWCA DLA ŚRODOWISKA GRID

Streszczenie. Artykuł opisuje wykorzystanie koncepcji GRID do implementacji szkieletu obliczeniowego wspomagającego rozpraszanie przetwarzania, na podstawie modelu nadzorca-wykonawca. Zastosowanie technologii Java pozwoliło na uniezależnienie się od systemu operacyjnego konkretnego węzła sieci GRID. Dzięki użyciu infrastruktury OSGi otrzymano możliwość modułowej budowy całego rozwiązania wraz z takimi zaletami, jak np. włączanie funkcjonalności modułów bez zatrzymywania systemu (ang. *hot deploy*). Zaimplementowany system jest elastyczny i umożliwia łatwą rozszerzalność o możliwość obliczeń dla dowolnych dziedzin problemowych.

Słowa kluczowe: przetwarzanie rozproszone, grid, OSGi, nadzorca, wykonawca

MASTER-SLAVE DISTRIBUTED TASK EXECUTION SYSTEM IMPLEMENTATION WITH OSGI IN A GRID ENVIROMENT

Summary. The article describes implementation of a computing framework based on the master-slave model in a GRID environment. Java technology gives the possibility of operating systems independence while usage of OSGi technology gives the possibility of modular architecture with such feature as hot deploy. The implemented system is flexible and enables extensions for various computing problems.

Keywords: distributed computing, grid, OSGi, master, slave

1. Wstęp

W ostatniej dekadzie zmniejszyło się tempo wzrostu wydajności nowo projektowanych i wytwarzanych jednostek obliczeniowych (przyczyną są przede wszystkim ograniczenia

natury technologicznej, np. problemy w dalszej miniaturyzacji tranzystorów). Zmusiło to środowiska naukowe do poszukiwania innych sposobów zwiększenia wydajności systemów obliczeniowych. Najbardziej popularną i najczęściej stosowaną metodą jest rozkładanie obliczeń na wiele procesorów lub komputerów.

Oczywiście istnieją przeznaczone gałęzie produktowe wyposażane w wiele wydajnych jednostek obliczeniowych, które dedykowane są do szczególnie złożonych obliczeń, jednak przyrost kosztu takich systemów jest nieproporcjonalny do przyrostu oferowanej mocy obliczeniowej. Skutkuje to ograniczoną możliwością powszechnego wykorzystania takich systemów, gdyż z punktu widzenia rachunku ekonomicznego zdecydowanie lepsze efekty daje zastosowanie systemów rozproszonych, złożonych z wielu tańszych, standardowych komputerów, wspólnie się ze sobą komunikujących.

Dodatkową elastyczność można osiągnąć, wykorzystując rozproszone środowiska heterogeniczne, w którym każdy komputer może być zbudowany w odmiennej architekturze i działać pod kontrolą innego systemu operacyjnego. Takie środowisko obliczeniowe wymaga wykorzystania odpowiedniej platformy, która umożliwia definiowanie, rozpraszanie i wykonywanie zadań na odpowiednim poziomie abstrakcji, który pozwoli uniezależnić się od konkretnej architektury czy też systemu operacyjnego. Dzięki temu osoba definiująca dane zadanie i dostarczająca dane wejściowe nie musi być nawet świadoma rodzajów platform sprzętowych i systemowych, na których oparta jest sieć obliczeniowa.

Przedmiotem niniejszego artykułu jest implementacja systemu wykonywania zadań rozproszonych w architekturze nadzorca-wykonawca (ang. *master/slave*). System został zbudowany na podstawie technologii OSGi, należącej do ekosystemu platformy Java, dzięki czemu może być wykorzystywany w środowiskach heterogenicznych. Rozwiązanie stanowić ma elastyczny szkielet obliczeniowy, dający możliwość implementacji dedykowanych domen dziedzinowych. Dzięki temu, po zdefiniowaniu dedykowanych typów danych czy też metod znajdzie on zastosowanie w danej dziedzinie, natomiast do wspomnianej implementacji nie będzie potrzebna wiedza o funkcjonowaniu sieci komputerowej. Zastosowanie stworzonego szkieletu będzie stanowić również niejako wspólny mianownik dla wielu domen obliczeniowych, pozwalając powtórnie używać (ang. *reuse*) funkcjonalności stworzonych na potrzeby innych domen. Dla potrzeb systemu został również zaproponowany dedykowany język definiowania i sterowania przepływem zadań rozproszonych, który, dzięki zastosowaniu narzędzia ANTLR [1], może również być rozszerzany w celu dostosowania go do specyfiki konkretnych dziedzin.

2. Koncepcja i architektura

System oparty jest na koncepcji nadzorca-wykonawca. Jest to jeden z najbardziej klasycznych modeli kontroli procesów i urządzeń, w którym jeden byt główny, zwany nadzorcą, kontroluje jeden lub wiele bytów podrzędnych, zwanych wykonawcami. Komunikacja między nadzorcą i wykonawcami oparta jest na zasadzie wymiany komunikatów, zaś sami wykonawcy nie komunikują się między sobą.

Technologia OSGi [2] stanowi platformę i szkielet aplikacji języka programowania oraz platformy Java. Zapewnia ona modułowość systemu i daje doskonałą możliwość zastosowania koncepcji luźnego powiązania (ang. *loose coupling*) [3]. W systemie wykonywania zadań rozproszonych zastosowana została referencyjna implementacja Equinox, rozwijana pod parasolem fundacji Eclipse [4].

System posiada cechy rozwiązań gridowych [5]. Wykonywanie zadań rozproszonych jest dla użytkownika „przezroczyste". Użytkownik nie musi wiedzieć ani dbać o to, ile i którzy wykonawcy biorą udział w wykonywaniu zleconego zadania.

2.1. OSGi

OSGi (ang. *Open Services Gateway initiative*) narzuca modułową budowę aplikacji. Moduły OSGi (ang. *OSGi bundles*) posiadają własny cykl życia, który znajduje się pod kontrolą kontenera OSGi.

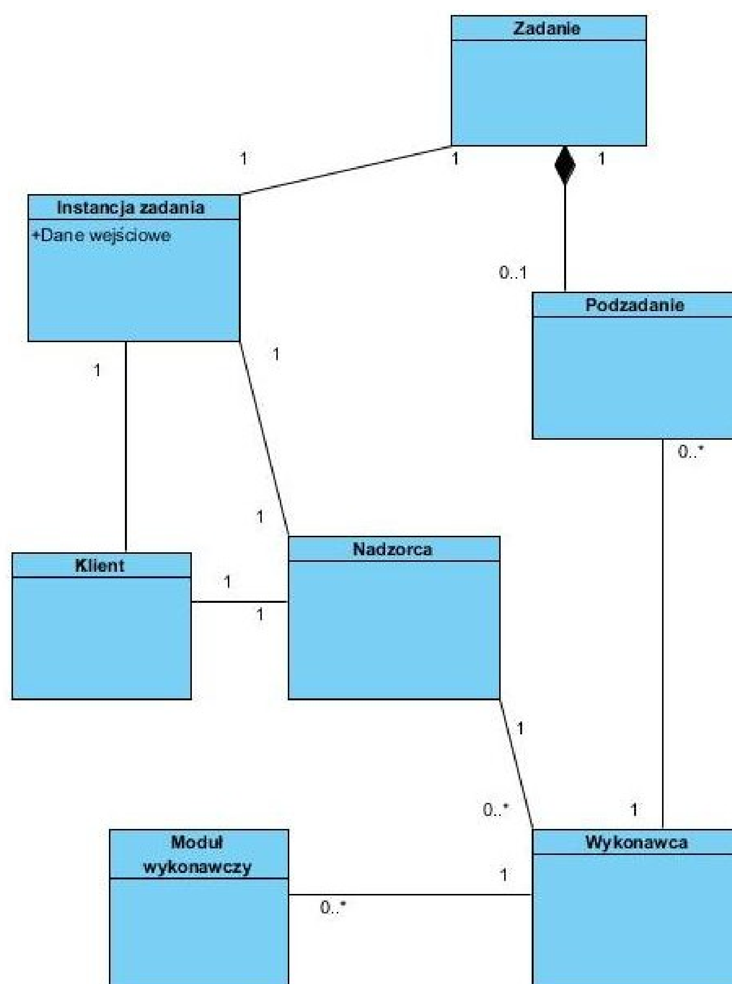
Moduł OSGi, w sensie jednostki wdrożeniowej, znajduje się w archiwum JAR, jednak nie udostępnia wszystkich swoich klas innym modułom. Aby klasa była dostępna dla innych modułów aplikacji, musi zostać wyspecyfikowana w deklaracji eksportu w pliku manifestu archiwum.

Kontener OSGi posiada również rejestr usług. Moduł może udostępniać usługę poprzez interfejs języka Java – w takim przypadku szczegóły implementacyjne usługi zostają zamknięte w definicji modułu i są niewidoczne dla innych modułów. Inne moduły zyskują dostęp do obiektu realizującego usługę przez mechanizm wstrzykiwania zależności (ang. *dependency injection*). Zarówno produkcja, jak i konsumpcja usług definiowane są w odpowiednich plikach XML.

OSGi zapewnia również kontrolę wersji modułów. Każdy moduł identyfikowany jest w kontenerze poprzez unikalną parę: nazwa i wersja. W systemie w danym momencie czasu może pracować wiele modułów o tej samej nazwie, ale innym numerze wersji. Dostępnych może być również wiele klas języka Java o tej samej, kwalifikowanej nazwie. Wynika to z faktu, że kontener OSGi tworzy osobny classloader JVM dla każdego modułu.

2.2. Model pojęciowy i koncepcja rozwiązania

Poniższy diagram klas stanowi uproszczony model pojęciowy systemu.



Rys. 1. Diagram klas
Fig. 1. Class diagram

Podstawowymi klasami, stanowiącymi jednocześnie aplikacje składające się na system, są klasy Nadzorcy i Wykonawcy. Nadzorca prowadzi rejestr Wykonawców, którzy za pomocą odpowiednich komunikatów zgłaszają mu swoją obecność w systemie. Wykonawca niezgłaszający się przez dany okres czasu jest z rejestru usuwany, co zapewnia odporność systemu na awarię pojedynczych węzłów.

Definicja zadania rozproszonego jest reprezentowana przez klasę Zadanie. Składa się ono, między innymi, z definicji Podzadań, które stanowią elementarną rozpraszaną w systemie jednostkę. Nadzorca rozsyła zlecenie wykonania Podzadania z odpowiednimi danymi wejściowymi do dostępnych Wykonawców, którzy po ich wykonaniu zwracają Nadzorcy odpowiednie wyniki.

Niezbędną do rozpoczęcia wykonywania Zadania jest tzw. Instancja zadania, która stanowi dla niego zbiór danych wejściowych.

Moduł wykonawczy stanowi zbiór komend, tj. funkcji wywoływanych w definicjach Zadania i Podzadania, zgrupowanych charakterystycznie dla dziedziny problemowej, w której system jest wykorzystywany. Każda komenda może przyjmować argumenty i zwracać wyniki. Rozszerzalność systemu jest zapewniona przede wszystkim przez możliwość tworzenia i wdrażania nowych Modułów wykonawczych, co czyni system adaptowalny dla nowych dziedzin obliczeniowych.

Klasa Klienta reprezentuje aplikację kliencką systemu. Służy ona użytkownikowi przede wszystkim do zlecania wykonywania Podzadań z odpowiednimi danymi wejściowymi, określonymi w definicji Instancji.

2.3. Przykładowy Moduł wykonawczy

Implementacja nowego Modułu wykonawczego jest równoznaczna ze stworzeniem modułu OSGi. Moduł taki udostępnia serwis OSGi przez interfejs IModuleService, wspólny dla wszystkich Modułów wykonawczych. Klasa implementacyjna Modułu musi dziedziczyć po klasie ModuleBase, która implementuje interfejs IModuleService.

Poniżej znajduje się początek definicji klasy implementacyjnej dla przykładowego Modułu operacji macierzowych:

```
@Module(id = "MatrixOperations", version = "1.0.0", type = ModuleType.BOTH)
public class MatrixOperations extends ModuleBase
```

Należy zwrócić uwagę na dodatkową adnotację Module, którą oznaczona została klasa implementacyjna. Pobiera ona w parametrach identyfikator modułu, jego wersję oraz typ. Typ modułu określa, czy może być on wykorzystywany w aplikacji Nadzorcy, aplikacji Wykonawcy bądź obu. Identyfikator i wersja modułu tworzą z kolei unikalną w ramach systemu parę.

Komenda Modułu wykonawczego implementowana jest za pomocą metody języka Java. Przykładowa metoda, służąca do wyliczania sumy iloczynów wektorów, ma postać:

```
@Command(id = "multiplyVectors", params = {
      @Param(id = "firstVector", type = VArray.class, role = VarRole.INPUT),
      @Param(id = "secondVector", type= VArray.class, role = VarRole.INPUT),
      @Param(id = "result", type =VInteger.class, role = VarRole.OUTPUT), })
public void multiplyVectors(VArray firstVector, VArray secondVector,
      VInteger result) {

   long longResult = 0;

   for (int i = 0; i < firstVector.size(); i++) {
      longResult += ((VInteger) firstVector.getElementAt(i)).longValue()
            * ((VInteger) secondVector.getElementAt(i)).longValue();
   }

   result.setValue(new VInteger(longResult));
}
```

Metoda oznaczona jest adnotacją Command. W parametrach adnotacji należy podać unikalny w ramach Modułu identyfikator komendy oraz listę argumentów. Każdy argument definiuje się za pomocą adnotacji Param, która przyjmuje unikalny w ramach Komendy identyfikator argumentu, jego typ (z zestawu typów systemu realizacji zadań rozproszonych) oraz rolę. Rola argumentu określa, czy mamy do czynienia z argumentem wejściowym, wyjściowym bądź argumentem pełniącym funkcję zarówno wejściową, jak i wyjściową.

Dzięki zastosowaniu powyższej techniki przekazywania argumentów możliwe jest, by komenda zwracała więcej niż jedną wartość.

Klasa ModuleBase zawiera kod, który za pomocą mechanizmów polimorfizmu i refleksji (pakiet java.lang.reflect) odczytuje informacje podane w adnotacjach klasy Modułu i metod reprezentujących Komendy, a następnie udostępnia je innym częściom systemu.

2.4. Przykładowa definicja zadania

Zadania rozproszone definiowane są w języku stworzonym specjalnie dla potrzeb systemu. Jest to język imperatywny z elementami proceduralnymi. Pełna gramatyka języka jest zdefiniowana za pomocą rozszerzonej notacji Backusa-Naura.

Poniżej przedstawiona została definicja przykładowego zadania mnożenia dwóch macierzy.

```
TASK MatrixIntMultiplying : 1.0.0 BEGIN
   MODULES BEGIN
      mo -> MatrixOperations: 1.0.0;
   END;
   IN BEGIN
      matrixA: MATRIX OF INTEGER;
      matrixB: MATRIX OF INTEGER;
   END;
   OUT BEGIN
      matrixRes: MATRIX OF INTEGER;
   END;
   VARS BEGIN
      vectorsToMultA: ARRAY OF ARRAY OF INTEGER;
      vectorsToMultB: ARRAY OF ARRAY OF INTEGER;
      matrixSize: INTEGER;
      slaveTasks: ARRAY OF SLAVETASKOBJ;
      sTask: SLAVETASKOBJ;
      nextSlave: SLAVEOBJ;
      i, j: INTEGER;
      result: INTEGER;
   END;
   FLOW BEGIN
      CALL mo.verifyInput firstMatrix -> matrixA, secondMatrix -> matrixB,
         resultMatrixSize -> matrixSize;
      CALL mo.sliceMatrixToMultiply firstMatrix      -> matrixA,
         secondMatrix     -> matrixB, vectorsFromFirst  -> vectorsToMultA,
         vectorsFromSecond   -> vectorsToMultB;
      CALL INIT_MATRIX matrix -> matrixRes, height -> matrixSize,
         width -> matrixSize;
      FOR i = 0; i < matrixSize; i++ BEGIN
         FOR j = 0; j < matrixSize; j++ BEGIN
            CALL GET_FREE_SLAVE_AND_BLOCK slave -> nextSlave;
```

```
                WHILE nextSlave == NULL BEGIN
                    CALL WAIT inMilis -> 500;
                    CALL GET_FREE_SLAVE_AND_BLOCK slave -> nextSlave;
                END;
                CALL START_SLAVETASK
                    slave -> nextSlave, slaveTask -> "vectorMultiplying",
                    slaveTaskObj -> sTask, vectorA -> vectorsToMultA[i],
                    vectorB -> vectorsToMultB[j], result -> matrixRes[i, j];
                CALL ADD_TO_ARRAY array -> slaveTasks, value -> sTask;
            END;
        END;
        CALL JOIN_ALL tasks -> slaveTasks;
    END;
    SLAVETASK vectorMultiplying BEGIN
        MODULES BEGIN
            mo -> MatrixOperations: 1.0.0;
        END;
        IN BEGIN
            vectorA: ARRAY OF INTEGER;
            vectorB: ARRAY OF INTEGER;
        END;
        OUT BEGIN
            result: INTEGER;
        END;
        FLOW BEGIN
            CALL mo.multiplyVectors
            firstVector      -> vectorA,
            secondVector  -> vectorB,
            result           -> result;
        END;
    END;
END;
```

Definicja Zadania rozpoczyna się od identyfikatora i wersji (para ta jest unikalna w obrębie systemu). Następnie znajdują się bloki definiujące wykorzystywane Moduły wykonawcze (wraz z określeniem „aliasów" wykorzystywanych w bloku przepływu) oraz zmienne wejściowe, wyjściowe i lokalne, wykorzystywane jedynie wewnątrz Zadania.

Słowo kluczowe FLOW rozpoczyna blok przepływu zadania. Jak widać na przykładzie, język implementuje struktury sterujące, takie jak pętle FOR i WHILE.

Wywołanie Komendy odbywa się za pomocą słowa kluczowego CALL, po którym znajduje się jej nazwa, poprzedzona aliasem Modułu wykonawczego. Następnie znajduje się lista argumentów Komendy oddzielona przecinkami. Dzięki zastosowaniu parametrów nazywanych, kolejność, w której występują one w wywołaniu Komendy, nie ma znaczenia.

Oprócz wywołania Komend zdefiniowanych w Modułach wykonawczych, wywołuje się również tzw. Komendy systemowe. Są to Komendy podstawowe dla działania systemu, a ich wywołanie można rozpoznać po braku aliasu przed nazwą Komendy. Przykładem może być komenda GET_FREE_SLAVE_AND_BLOCK, która zwraca zmienną reprezentującą dostępnego w danym momencie wykonawcę, jednocześnie blokując go dla potrzeb Zadania.

Ostatnim elementem definicji Zadania rozproszonego jest definicja Podzadania. Podzadanie stanowi rozpraszaną jednostkę Zadania i jest wykonywane na aplikacjach Wykonawców. Jego struktura jest niemal identyczna z definicją Zadania (jedyną różnicą jest brak możliwości definiowania kolejnych Podzadań wewnątrz Podzadania).

Podzadanie, wraz z listą argumentów i zmienną reprezentującą Wykonawcę, jest uruchamiane przez wywołanie Komendy START_SLAVETASK.

Zadanie może definiować, teoretycznie, dowolną liczbę Podzadań. W szczególności może nie definiować żadnego – takie Zadanie będzie jednak wykonywane jedynie na Nadzorcy, co oznacza, że jego realizacja nie zostanie rozproszona.

3. Podsumowanie

Technologia OSGi doskonale sprawdza się w projektach, w których liczba modułów nie jest z góry określona i z założenia może w przyszłości rosnąć. Umożliwia również maksymalizację skuteczności wielokrotnego wykorzystania tego samego kodu – w przypadku systemu będącego tematem artykułu, aplikacje Nadzorcy i Wykonawcy dzielą kod źródłowy w ok. 90%.

Wykorzystana technologia ułatwia również tworzenie nowych Modułów Wykonawczych, dzięki czemu pole potencjalnego zastosowania systemu jest bardzo szerokie. Każdy z tych modułów jest tożsamy z modułem OSGi, zatem kontener automatycznie zajmuje się ich wersjonowaniem i zarządzaniem cyklem życia.

Dzięki zdefiniowaniu odpowiedniego języka DSL (ang. *Domain-Specific Language*), służącego do opisu zadań rozproszonych, możliwe jest ich definiowanie przez osoby niebędące programistami. Jest to ważna cecha, która pozwala na szersze wykorzystanie systemu w różnych dziedzinach problemowych.

System może zostać w przyszłości zaadaptowany do architektury agentowej. Oznacza to odejście od modelu nadzorca-wykonawca, jednak, dzięki istniejącemu już w systemie podziałowi modułowemu większość kodu źródłowego, może zostać ponownie wykorzystana.

W przyszłości można również rozważyć udostępnienie możliwości oferowanych przez współczesne akceleratory oparte na architekturach CUDA (ang. *Compute Unified Device Architecture*) i podobnych, które mogłyby być wykorzystywane przy implementacji podzadań dla modułów wykonawczych. Pozwoliłoby to łączyć tak popularne obecnie źródła „tanich” dużych mocy obliczeniowych w infrastrukturę o architekturze GRID.

BIBLIOGRAFIA

1. Parr T.: The Definitive Antlr Reference. 2007.
2. Hall R. S., Paulus K., McCulloch S., Savage D.: OSGi In Action. Creating Modular Applications in Java. 2011.

3. Fowler M.: Inversion of Control Containers and the Dependency Injection pattern. 23.01.2004
4. D'Anjou J., Fairbrother S., Kehn D., Kellerman J., McCarthy P.: The Java Developer's Guide to Eclipse. 2nd Edition, 2005.
5. Magoules F.: Fundamentals of Grid Computing. 2010.

Wpłynęło do Redakcji 13 stycznia 2012 r.

Abstract

The article describes the concept and the implementation of a computing framework based on the master-slave model in a GRID environment. Java technology gives the possibility of operating systems independence while usage of OSGi technology gives the possibility of modular architecture with such feature as hot deploy, bundles versioning etc.

The implemented system is flexible and enables extensions for various computing problems. To build modules dedicated for specific computing domain one need only implement (additionally to domain operators) simple communicating interface but needn't know about network protocols, operating systems etc.

Adresy

Łukasz WYCIŚLIK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, lwycislik@polsl.pl.

Maciej MILCZAREK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska.