

Uniwersytet Jagielloński
Wydział Matematyki i Informatyki
INSTYTUT INFORMATYKI i MATEMATYKI KOMPUTEROWEJ
Studia stacjonarne

Numery indeksów: 1076011, 1076556

Alan Hawrot, Michał Semik

jNode – framework dla aplikacji rozproszonych opierający się o nieblokujący model komunikacji

Opiekun pracy magisterskiej:
dr Piotr Kalita

Kraków 2016

Spis treści

| | |
|---|-----------|
| Wstęp..... | 5 |
| Podstawowe pojęcia..... | 5 |
| Odwrócenie sterowania..... | 5 |
| Wywołanie zwrotne..... | 6 |
| Wstrzykiwanie zależności | 7 |
| Classpath | 8 |
| Class loader | 9 |
| Classpath scanning..... | 10 |
| Programowanie współbieżne | 11 |
| Wybrane problemy programowania współbieżnego | 11 |
| Race Condition | 11 |
| Własność żywotności..... | 11 |
| Własność uczciwości..... | 12 |
| Programowanie synchroniczne | 12 |
| Programowanie rozproszone | 14 |
| System rozproszony | 14 |
| Model obliczeniowy..... | 16 |
| Programowanie asynchroniczne..... | 16 |
| Framework | 16 |
| Open Source..... | 17 |
| Mechanizm refleksji..... | 17 |
| Mechanizm serializacji..... | 18 |
| Skalowanie horyzontalne i wertykalne | 18 |
| Programowanie asynchroniczne | 19 |
| Node.js | 21 |
| Apache Hadoop..... | 22 |
| Hadoop Distributed File System | 23 |
| MapReduce..... | 24 |
| jNode..... | 26 |
| Model obliczeniowy | 27 |
| Skalowalność | 28 |
| Dependency Injection..... | 28 |
| Porównanie z Node.js..... | 29 |
| Porównanie z Apache Hadoop | 30 |
| Zastosowane technologie, frameworki i biblioteki | 30 |
| Spring Framework..... | 30 |

| | |
|---|-----------|
| Maven, czyli zautomatyzowane zarządzanie projektem | 31 |
| AOP | 32 |
| Aspect Weaving | 32 |
| JGroups | 33 |
| Non-Blocking I/O | 34 |
| WatchService | 35 |
| java.util.concurrent | 35 |
| Architektura systemu i algorytmy | 36 |
| Moduł client | 38 |
| Tworzenie zadań i wywołania zwrotne | 39 |
| Rejestrowanie zadań | 39 |
| Adnotacje | 40 |
| Moduł engine | 41 |
| Rozpoczynanie obliczeń | 41 |
| Najważniejsze klasy | 41 |
| TaskCoordinator | 42 |
| EventLoopThread i EventLoopThreadRegistry | 42 |
| WorkerPool | 43 |
| Modyfikacja implementacji puli wątków dostarczonej przez Spring Framework | 44 |
| Struktura tasków na poziomie modułu engine | 45 |
| JarPath, czyli katalog na archiwa | 46 |
| Zależności modułu engine | 47 |
| Moduł cluster | 47 |
| Wstęp | 47 |
| Główna abstrakcja, czyli Distributor oraz integracja z biblioteką JGroups | 49 |
| Struktura tasków na poziomie modułu cluster | 51 |
| Rejestry tasków, czyli nadzorowanie stanu systemu | 52 |
| Mechanizm wyboru węzła | 52 |
| Komunikacja między węzłami, czyli nurkując w Distributora | 56 |
| HeartBeat | 56 |
| Dostarczanie zadań pomiędzy węzłami i rozsyłanie wyników | 56 |
| Dostarczanie zadań pomiędzy węzłami | 59 |
| Procedura ponownego przenoszenia zadania na inny węzeł | 60 |
| Podejście pierwsze | 61 |
| Podejście drugie | 62 |
| Podejście trzecie | 62 |
| Podejście czwarte | 63 |

| | |
|---|------------|
| Podejście piąte..... | 65 |
| Podejście szóste..... | 66 |
| Rozsyłanie wyników..... | 67 |
| Wykrywanie i rejestracja podzadań powstałych na innym węźle | 68 |
| DelegationHandler..... | 69 |
| Propagowanie archiwum jar pomiędzy węzłami..... | 74 |
| Podejście 1 | 74 |
| Podejście 2 | 74 |
| Obsługa błędów | 75 |
| Zwalnianie zasobów przy zamykaniu aplikacji | 77 |
| Spis komunikatów w systemie..... | 78 |
| Moduł crosscutting | 79 |
| Moduł context | 80 |
| Moduł main | 83 |
| Moduł monitor | 85 |
| Podsumowanie działania systemu jNode..... | 85 |
| Biblioteka użytkownika i przykładowe programy..... | 89 |
| Program: Quicksort | 89 |
| Program: Password Cracker..... | 96 |
| Przetestowane scenariusze | 99 |
| Możliwości rozwoju jNode | 100 |
| Podsumowanie..... | 102 |
| Bibliografia | 102 |

Wstęp

W dzisiejszych czasach ze względu na rosnącą moc obliczeniową komputerów i malejące koszty poszczególnych podzespołów zwraca się coraz mniejszą uwagę na tworzenie wydajnego oprogramowania. Wiele dostępnych na rynku języków programowania wspiera paradygmaty programowania współbieżnego, rozproszonego czy asynchronicznego. Pomimo tego, że języki te umożliwiają pisanie takich programów, które w efektywny sposób korzystają z zasobów sprzętowych, to jednak użycie udostępnianych mechanizmów i bibliotek programistycznych jest skomplikowane, a powstały kod - ciężki w utrzymaniu. Celem tej pracy było opracowanie frameworka przeznaczonego dla języka programowania Java, który ułatwia tworzenie aplikacji lepiej wykorzystujących dostępną moc obliczeniową. Powstałe oprogramowanie – jNode – opiera się na modelu asynchronicznym, który zrealizowany jest w oparciu o zasady programowania sterowanego zdarzeniami. Ponadto głównym założeniem było umożliwienie skalowania wertykalnego i horyzontalnego. Framework jNode tworzony był we współpracy i jego autorami są: Michał Semik oraz Alan Hawrot. System został opisany w niniejszym dokumencie. Każdy z kolejnych fragmentów pracy napisany jest przez jednego z autorów i pod każdym widnieje podpis. W rozdziale pierwszym zostały przedstawione i omówione najważniejsze pojęcia związane z systemem. Znajduje się tam również krótki opis najważniejszych technologii, frameworków i bibliotek, które zostały wykorzystane przy tworzeniu systemu. W kolejnym rozdziale opisana została szczegółowo architektura systemu jNode i zaimplementowane algorytmy. Trzeci rozdział dotyczy biblioteki użytkownika jNode, gdzie przedstawione zostały przykładowe programy napisane przy jej użyciu. Na końcu pracy znajduje się podsumowanie i dyskusja na temat możliwości rozwoju systemu.

[Alan Hawrot]

Podstawowe pojęcia

Odwrócenie sterowania

Odwrócenie sterowania (ang. Inversion of Control), jest to mechanizm opierający się o zmianę przepływu sterowania w programie w stosunku do tradycyjnego. Pojęcie odwrócenia przepływu sterowania stosuje się do dwóch komunikujących się porcji programu. Porcje te mogą być przykładowo dwiema komunikującymi się bibliotekami, fragmentem tworzonego

programu komunikującego się z biblioteką, a także dwoma komunikującymi się obiektami w języku obiektowo orientowanym w ramach jednego programu.

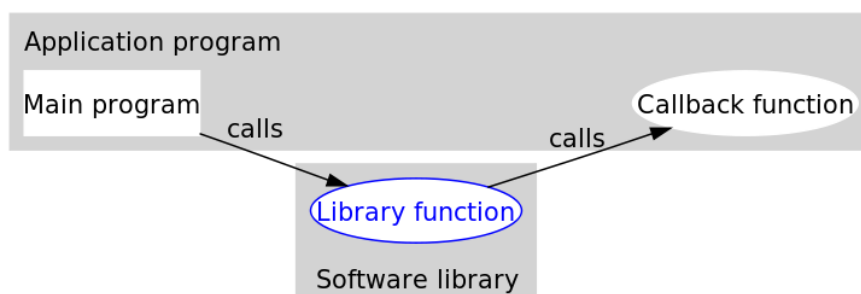
Zakładamy, że mamy dwie porcje programu A i B, w których A korzysta z B (przykładowo biblioteka A zawiera w sobie bibliotekę B). W tradycyjnym przepływie sterowania komunikacja polega na wywoływaniu metod/funkcji/procedur porcji B przez A. Po odwróceniu przepływu sterowania, B wywołuje instrukcje A, natomiast w dalszym ciągu A korzysta z B. Zasada ta jest też znana jako Hollywood Principle: Don't call us, we'll call you. Może ona być zrealizowana m.in. poprzez opisane poniżej frameworki, wywołania zwrotne, oraz wstrzykiwanie zależności [1,2].

[Michał Semik]

Wywołanie zwrotne

Wywołanie zwrotne (ang. Callback), jest to technika komunikacji pomiędzy dwiema porcjami programu, która polega na przekazaniu funkcji do wywołania z porcji A do porcji B, natomiast porcja B w dowolnym momencie ją wywołuje. Wywołania zwrotne można podzielić na synchroniczne oraz asynchroniczne. Pierwsze mają miejsce, gdy wywołanie zwrotne następuje podczas przekazywania funkcji pomiędzy porcjami, natomiast drugie, gdy wywołanie następuje w dowolnym czasie [3].

Wywołania zwrotne są bardzo często stosowane w komunikacji z bibliotekami. Przykładowo, w bibliotece realizującej okienkowy interfejs graficzny, funkcje biblioteczne mogą przyjmować wywołania zwrotne w celu przekazania do aplikacji korzystającej z biblioteki informacji o akcjach użytkownika.



Rysunek 1. Przykład wywołania zwrotnego przy komunikacji programu użytkownika wraz z biblioteką [3].

W języku Java wywołanie zwrotne jest możliwe do zrealizowania poprzez polimorfizm. Jest to typowe podejście języków obiektowo orientowanych. W językach skryptowych (przykładowo JavaScript), w których nie jest stosowana kontrola typów, do osiągnięcia wywołań zwrotnych wykorzystywane jest tzw. duck typing, czyli wywoływanie metody obiektu na podstawie jej sygnatury (nazwy metody i ilości przyjmowanych argumentów).

[Michał Semik]

Wstrzykiwanie zależności

Wstrzykiwanie zależności (ang. dependency injection) jest to technika często stosowana w językach obiektowo orientowanych, polegająca na separacji konfiguracji obiektów od ich użycia. Kieruje się tym, że obiekty nie powinny tworzyć innych obiektów. Obiekty zawierane powinny być podawane (wstrzykiwane) do obiektów zawierających podczas ich konstrukcji. Dzięki takiemu podejściu tworzone oprogramowanie posiada lepszą własność ponownego użycia oraz jest prostsze w testowaniu.

Załóżmy, że chcemy napisać komponent, który pozwala na wyszukiwanie filmów, filtrując je względem reżysera. W takim wypadku możemy stworzyć klasę MovieLister, która posiada funkcję moviesDirectedBy przyjmującą jako argument imię i nazwisko reżysera.

```
1  import java.util.*;
2
3  public class MovieLister {
4
5      private MovieFinder finder;
6      public MovieLister() { finder = new ColonDelimitedMovieFinder("movies1.txt"); }
7
8
9
10     public Movie[] moviesDirectedBy(String arg) {
11         List allMovies = finder.findAll();
12         for (Iterator it = allMovies.iterator(); it.hasNext(); ) {
13             Movie movie = (Movie) it.next();
14             if (!movie.getDirector().equals(arg)) it.remove();
15         }
16         return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
17     }
18 }
19
```

Kod źródłowy 1. Klasa Movie Lister [4].

Funkcja ta korzysta z interfejsu MovieFinder, która posiada funkcję findAll, udostępniającą wszystkie filmy.

```

1  import java.util.List;
2
3  public interface MovieFinder {
4      List findAll();
5  }
6

```

Kod źródłowy 2. Interfejs MovieFinder [4].

Tworzeniem obiektu implementującego interfejs `MovieFinder` zajmuje się klasa `MovieLister`, w naszym przykładzie jest to `ColonDelimitedMovieFinder`.

Implementacja interfejsu `MovieFinder` korzysta z źródła informacji o filmach w postaci pliku `movie1.txt`, w którym filmy są oddzielone od siebie przecinkiem. W przypadku, gdybyśmy ponownie chcieli wykorzystać komponent w innym programie, natomiast źródłem informacji byłby plik `xml`, filmy znajdowałyby się w bazie danych lub były dostępne poprzez `web service` to nie możemy tego zrobić, ponieważ obiekt `MovieLister` jest na stałe powiązany z obiektem klasy `ColonDelimitedMovieFinder`. Rozwiązaniem tego problemu jest `dependency injection`, które można zrealizować poprzez podanie obiektu implementującego interfejs `MovieFinder` jako argument konstruktora klasy `MovieLister`.

Rozszerzając ten problem do architektury całego systemu, tzn. zakładając, że każda klasa korzysta z `dependency injection` powstaje nowy problem: w jaki sposób połączyć razem wykorzystywane obiekty? Aby rozwiązać ten problem powstały tzw. kontenery wstrzykiwania zależności. Jedną z implementacji takiego podejścia jest omawiana w osobnym rozdziale *Spring Framework* [4,5].

[Michał Semik]

Classpath

`Classpath` jest to zmienna wykorzystywana przez wirtualną maszynę Javy w celu załadowania klas uruchamianej aplikacji użytkownika. W zmiennej tej mogą być podawane ścieżki bezpośrednio do skompilowanych klas (o rozszerzeniu `.class`), do katalogu będącego korzeniem pakietów skompilowanych klas lub do archiwów `zip` oraz `jar`. Wykorzystywana jest także dodatkowa wartość `"*"` (tzw. `wildcard`), którą można umieścić jedynie na końcu ścieżki katalogu. W takiej formie zmienna oznacza, że powinny zostać uwzględnione wszystkie archiwa `jar` i `zip` w tym katalogu. Ścieżki powinny być oddzielane dwukropkiem lub

średnikiem, w zależności od wykorzystywanego systemu operacyjnego. Zmienna może być zdefiniowana przy pomocy zmiennej środowiskowej CLASSPATH lub przy pomocy argumentu -cp, podawanego podczas uruchamiania wirtualnej maszyny Javy. Argument -cp ma wyższy priorytet, tzn. jeżeli został podany, to zmienna środowiskowa CLASSPATH zostanie zignorowana. Zawarte w tej zmiennej klasy są dostępne podczas wykonywania aplikacji z poziomu tzw. system class loadera, omawianego przy okazji class loaderów [6].

[Michał Semik]

Class loader

Class loader to oprogramowanie wchodzące w skład Java Runtime Environment (JRE), które służy do dynamicznego ładowania do maszyny wirtualnej Javy klas w postaci plików .class zawierających kod bajtowy. Pliki te tworzone są przez kompilator Javy, który kompiluje kod źródłowy do kodu bajtowego. Klasy ładowane są „na życzenie” tzn. wtedy, gdy są potrzebne i wywoływane w kodzie. Każda klasa musi zostać załadowana przez class loader i jest powiązana z tym class loaderem, który ją załadował. Podczas uruchamiania maszyny wirtualnej trzy rodzaje class loaderów są używane:

- Bootstrap class loader
- Extensions class loader
- System class loader

Bootstrap class loader jest odpowiedzialny za załadowanie podstawowych, standardowych klas Javy, które znajdują się pod ścieżką <JAVA_HOME>/jre/lib. Class loader ten jest częścią maszyny wirtualnej Javy i napisany jest w kodzie natywnym.

Extensions class loader ładuje klasy znajdujące się w katalogu <JAVA_HOME>/jre/lib/ext lub w każdym innym określonym przez zmienną java.ext.dirs (system property).

System class loader ładuje klasy znajdujące się w ścieżce CLASSPATH.

Występuje tutaj zjawisko delegacji i hierarchii class loaderów tzn. każdy z class loaderów posiada „ojca”. Wyróżnione zostały dwa typy class loaderów: parent-first i child-

first. Domyślnie stosowany jest typ parent-first. W jego przypadku podczas próby załadowania klasy następuje delegacja tego zadania do „ojca”, który z kolei deleguje to zadanie do swojego „ojca”. Na samym szczycie hierarchii znajduje się Bootstrap class loader. Jeżeli dana klasa nie została załadowana przez „ojca”, to wtedy następuje załadowanie klasy przez dany class loader.

Należy również zaznaczyć, że gdy zaistnieje potrzeba rozszerzenia funkcjonalności lub zmiany sposobu działania domyślnych class loaderów – jest taka możliwość. Każdy z class loaderów jest napisany w Javie. Tworzenie własnej implementacji opiera się o napisanie klasy dziedziczącej z klasy abstrakcyjnej o nazwie `ClassLoader` znajdującej się w pakiecie `java.lang` [7].

[Alan Hawrot]

Classpath scanning

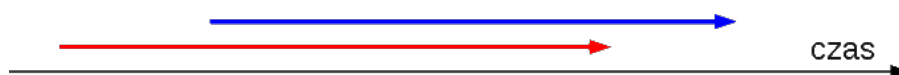
Classpath scanning jest to proces iterowania poprzez zasoby dostępne dla zadanych class loaderów. Głównie są to skompilowane klasy, natomiast mogą to być także dowolne inne pliki, w tym archiwa zawierające klasy. Skanowanie wewnętrznych archiwów jest realizowane rekurencyjnie. Gdy zlecimy wykonanie skanowania na systemowym class loaderze, uwzględniane będą zasoby we wszystkich zadanych archiwach, katalogach i ich podkatalogach podanych w zmiennej `classpath`, a także zasoby udostępniane przez przodków class loadera, czyli `extension` i `bootstrap`. Classpath scanning na systemowym class loaderze potrafi być dość długim procesem, gdy zechcemy skanować od korzenia, tzn. bez podania pakietu/folderu początkowego. Dlatego też w praktyce skanowanie rozpoczyna się odadanego pakietu początkowego. Classpath scanning jest zazwyczaj stosowany w scenariuszach, w których nie można skorzystać z mechanizmu refleksji. Przykładowo za pomocą refleksji możemy znaleźć wszystkie adnotacje, które używa dana klasa, natomiast nie możemy znaleźć wszystkich klas, które korzystają z danej adnotacji. Za pomocą refleksji jest także bardzo łatwo znaleźć wszystkie klasy bazowe dla zadanej klasy, natomiast znalezienie wszystkich klas pochodnych nie jest możliwe. W obu tych przypadkach można wykorzystać classpath scanning. Oba te scenariusze są szczególnie przydatne przy tworzeniu frameworków, takich jak kontener `dependency injection` [8].

[Michał Semik]

Programowanie współbieżne

Pojęcie programowania współbieżnego początkowo dotyczyło wielozadaniowych systemów operacyjnych, jednak dzisiaj pojęcie to jest wykorzystywane na szerszą skalę i dotyczy różnych aspektów programowania, takich jak programowanie równoległe, bazy danych, systemy czasu rzeczywistego, monitorowanie procesów technologicznych.

Z programowaniem współbieżnym mamy do czynienia, gdy jeden proces (lub wątek) rozpoczął się w trakcie działania drugiego procesu.



Rysunek 2. Reprezentacja dwóch procesów postępujących względem osi czasu [9].

Program współbieżny cechuje się posiadaniem współdzielonej pamięci głównej pomiędzy procesami/wątkami, zatem jest on wykonywany w ramach jednego komputera [9].

[Michał Semik]

Wybrane problemy programowania współbieżnego

Race Condition

Race condition jest jednym z fundamentalnych problemów programowania współbieżnego. Z zajęciem race condition mamy do czynienia, gdy poprawność wykonania programu zależy od kolejności wykonanych instrukcji w wątkach czy też w procesach. Inaczej mówiąc, gdy dojdzie do nieprzewidzianego przez programistę przeplotu instrukcji to w programie wystąpi błąd. Przykładowo zapis danych współdzielonych powinien być wzajemnie wykluczony z każdą inną próbą odczytu i zapisu tych danych. Jeżeli własność wzajemnego wykluczenia nie zostanie spełniona, to w programie mogą powstać uszkodzone dane lub odczytane dane mogą być nieprawidłowe [10].

[Michał Semik]

Własność żywotności

Własność żywotności można w najprostszy sposób opisać w następujący sposób: *jeśli proces chce coś zrobić, to w końcu mu się to uda*. W przypadku braku żywotności dochodzi do jednego z poniżej opisanych błędów:

1. Zakleszczenie (ang. *Deadlock*) – jest to globalny brak żywotności. Wszystkie procesy oczekują na zdarzenie, które nigdy nie nastąpi.
2. Zagłodzenie (ang. *Starvation*) – lokalny brak żywotności. Jeden z procesów zgłosił żądanie dostępu do jakichś zasobów, ale nigdy ich nie otrzyma [9].

[Alan Hawrot]

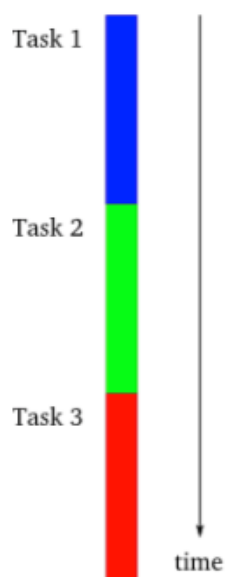
Własność uczciwości

W przypadku wcześniej opisanej własności żywotności akceptowalne są sytuacje, w których żądanie zgłoszone przez proces będzie obsłużone po kilku miesiącach lub spośród dwóch procesów jeden jest ciągle faworyzowany. Ciężko jednak uznać takie rozwiązania za sprawiedliwe. Własność uczciwości jest trudna do zdefiniowania i zależna od konkretnej sytuacji. Rozważa się następujące wersje:

- Uczciwość słaba – jeśli proces nieprzerwanie zgłasza żądanie, to w końcu zostanie ono obsłużone.
- Uczciwość mocna – jeśli proces nieskończenie wiele razy zgłasza żądanie, to w końcu zostanie ono obsłużone.
- Oczekiwanie liniowe – jeśli proces zgłosił żądanie, to zostanie ono obsłużone zanim inne procesy zostaną obsłużone więcej niż jeden raz.
- FIFO – jeśli proces zgłosił żądanie, to zostanie ono obsłużone przed żądaniami innych procesów zgłoszonych później [9].

[Alan Hawrot]

Programowanie synchroniczne



Rysunek 3. Jednowątkowy synchroniczny model [11].

Na powyższym rysunku został przedstawiony przykładowy program składający się z trzech instrukcji („zadań”), który wykonywany jest w oparciu o jednowątkowy synchroniczny model. W programowaniu synchronicznym instrukcje wykonywane są sekwencyjnie, w porządku w jakim zostały zapisane w programie, a w danej chwili wykonywana jest tylko jedna instrukcja. Rozpoczęcie wykonywania kolejnej instrukcji może się rozpocząć pod warunkiem, że poprzednia została w całości zakończona.



Rysunek 4. Wielowątkowy synchroniczny model [11].

Przy zastosowaniu wielowątkowego modelu synchronicznego wykonywanie instrukcji może być oddelegowane do osobnych wątków. Wątki zarządzane są przez system operacyjny lub pośrednio przez maszynę wirtualną i w przypadku wielordzeniowych procesorów, wielowątkowych procesorów lub komputerów z wieloma procesorami wątki te, a więc instrukcje programu mogą być wykonywane jednocześnie. Głównym celem zastosowania tego

modelu jest to, że programista koncentruje się na definiowaniu zestawu instrukcji, które mogą zostać wykonane niezależnie, w tym samym czasie, a sama egzekucja tych instrukcji jest obsługiwana przez system operacyjny lub pośrednio przez maszynę wirtualną. Niestety w praktyce zazwyczaj bywa, że aplikacje wielowątkowe są skomplikowane i konieczna jest dodatkowa koordynacja pomiędzy wątkami. Mówimy tutaj o wcześniej wspomnianym *Programowaniu współbieżnym*, które niesie ze sobą wiele trudności. Niektóre z nich zostały przybliżone w poprzednich punktach pracy. Przykładowo jeżeli wątki jednocześnie korzystają z zasobu współdzielonego jakim może być pamięć operacyjna - konieczna jest dodatkowa synchronizacja. Konieczność wprowadzenia synchronizacji może spowodować spadek wydajności, a tym samym narzut użycia modelu współbieżnego może być tak duży, że jego zastosowanie będzie nieopłacalne [11].

[Alan Hawrot]

Programowanie rozproszone

Program rozproszony to zbiór współbieżnie wykonywanych procesów w modelu rozproszonym. O modelu rozproszonym mówi się wtedy, gdy program współbieżny wykonywany jest na różnych komputerach. Model rozproszony stosuje się również w sytuacjach, gdy procesy działają na tym samym komputerze, ale ich przestrzenie adresowe tworzą odrębne jednostki. Podsumowując cechą charakterystyczną jest brak wspólnej pamięci, a do komunikacji używana jest wymiana komunikatów [9].

[Alan Hawrot]

System rozproszony

System rozproszony to zbiór niezależnych urządzeń technicznych połączonych w jedną, spójną logicznie całość. System rozproszony posiada następujące cechy:

- Dzielenie zasobów – wielu użytkowników systemu może korzystać z danego zasobu.
- Otwartość – podatność na rozszerzenia, możliwość rozbudowy systemu zarówno pod względem sprzętowym, jak i oprogramowania.
- Współbieżność – zdolność do przetwarzania wielu zadań jednocześnie.
- Skalowalność – cecha systemu umożliwiająca zachowanie podobnej wydajności systemu przy zwiększaniu skali systemu.

- Tolerowanie awarii – właściwość systemu umożliwiająca działanie systemu mimo pojawiania się błędów i (lub) uszkodzeń.
- Przezroczystość – właściwość systemu powodująca postrzeganie systemu przez użytkownika jako całości, a nie poszczególnych składowych.
- Brak wspólnego zegara fizycznego.
- Brak wspólnej pamięci – do komunikacji używana jest wymiana komunikatów.
- Rozproszenie geograficzne.
- Niejednorodność – inaczej heterogeniczny system, poszczególne składowe systemu mogą się od siebie różnić, np. różne procesory, różny sprzęt, różne systemy operacyjne.

Współczesne systemy rozproszone mają zazwyczaj postać klastra komputerowego – grupy różnych komputerów połączonych siecią komputerową, które współpracują ze sobą w celu udostępnienia zintegrowanego środowiska pracy. Jednostki komputerowe wchodzące w skład klastra nazywane są węzłami.

Bardzo ważną cechą systemu rozproszonego jest jego niezawodność. System rozproszony powinien być wysoce odporny na błędy lub awarie. Awarie systemu mogą wystąpić na różnych poziomach systemu rozproszonego:

- Błąd węzła – węzeł może nie wysyłać komunikatu lub wysyłać niepoprawne komunikaty. Węzeł czasowo lub trwale może przestać działać.
- Błąd łącza – łącze między dwoma węzłami może nie przekazywać komunikatu lub przekazywać niepoprawne komunikaty.
- Błąd wynikający z połączenia dwóch poprzednich.

Niestety obecnie całkowita odporność na awarie nie jest możliwa, lecz należy uwzględnić wspomniane sytuacje przy projektowaniu systemu rozproszonego i zapewnić ich odpowiednią obsługę, aby pomimo ich wystąpienia użytkownicy mogli nadal korzystać z udostępnianej przez system usługi/zasobu. Ze względu na to, iż jednostki wchodzące w skład systemu rozproszonego są zazwyczaj połączone siecią komputerową, na poziomie oprogramowania można przykładowo rozważyć zastosowanie protokołu zapewniającego niezawodność dostarczenia komunikatów [12,13].

[Alan Hawrot]

Model obliczeniowy

Model obliczeniowy (ang. computational model) określa, w jaki sposób będą programowane i wykonywane obliczenia w programie. Na model obliczeniowy składa się model architektoniczny systemu i język programowania [14].

[Michał Semik]

Programowanie asynchroniczne

Z programowaniem asynchronicznym w przeciwieństwie do programowania synchronicznego mamy do czynienia, gdy w komunikacji dwóch porcji programu nadawca komunikatów nie czeka na odpowiedź od odbiorcy. Przykładowo może to nastąpić przy wywołaniu procedury, wtedy mówimy o asynchronicznym wywołaniu procedury. W komunikacji takiej nadawca zapisuje dane do bufora (przykładowo może to być gniazdo TCP/UDP), natomiast odbiorca odbiera dane w dowolnej chwili. Może to nastąpić w innym wątku, procesie lub nawet na innym komputerze. Komunikacja asynchroniczna jest typowo stosowana w programowaniu rozproszonym [15,9].

[Michał Semik]

Framework

Framework – szkielet do budowy aplikacji. To abstrakcja, która definiuje strukturę aplikacji oraz zapewnia ogólny mechanizm jej działania. Framework dostarcza zestaw komponentów i bibliotek ogólnego przeznaczenia do wykonywania określonych zadań. Komponentami mogą być przykładowo: programy, kompilatory, interfejsy programistyczne aplikacji (API), biblioteki, czy też różnego rodzaju narzędzia programistyczne. Programista tworząc aplikację, rozbudowuje i dostosowuje poszczególne komponenty do wymagań realizowanego projektu.

Cechy charakterystyczne:

- odwrócenie sterowania – w odróżnieniu od aplikacji oraz bibliotek, przepływ sterowania jest narzucany przez framework, a nie przez użytkownika.

- domyślne zachowanie – framework posiada domyślną konfigurację, która musi być użyteczna i dawać sensowny wynik, zamiast być zbiorem pustych operacji do nadpisania przez programistę.
- rozszerzalność – poszczególne komponenty frameworka powinny być rozszerzalne przez programistę, jeśli ten chce rozbudować je o niezbędną mu dodatkową funkcjonalność.
- zamknięta struktura wewnętrzna – programista może rozbudowywać framework, ale nie poprzez modyfikację domyślnego kodu [16].

[Alan Hawrot]

Open Source

O oprogramowaniu Open Source mówimy, gdy jego kod źródłowy jest udostępniany na licencji, która pozwala na czytanie, zmianę i udostępnianie kodu źródłowego komukolwiek w dowolnym celu [17].

[Michał Semik]

Mechanizm refleksji

Mechanizm refleksji, znany też jako paradygmat programowania refleksyjnego, polega na tym, że program podczas wykonywania ma możliwość analizy i modyfikacji własnego kodu wykonywalnego. Mechanizm ten jest szeroko stosowany w językach wysokiego poziomu jak C# oraz Java.

Język Java pozwala na inspekcję klas, interfejsów, adnotacji, pól, metod oraz konstruktorów w czasie wykonywania programu bez znajomości ich nazw podczas kompilacji. Na tej samej zasadzie umożliwia tworzenie obiektów oraz wywoływanie metod. Istnieje możliwość zmiany modyfikatorów dostępu pól oraz metod, dzięki czemu można uzyskać dostęp do metod i pól prywatnych.

W Javie mechanizm refleksji wykorzystywany jest bardzo często przy użyciu adnotacji, które dostarczają dodatkowych informacji podczas analizy programu.

Mechanizm refleksji jest potężnym narzędziem; pozwala na wykonywanie operacji, które nie mogłyby być bez niego możliwe. Mechanizm ten jest mniej wydajny od nierefleksyjnych odpowiedników (przykładowo wywołanie metod, tworzenie obiektów). Jest to spowodowane tym, że nie można dokonać niektórych optymalizacji, które normalnie dokonuje kompilator. Mechanizm ten jest również podatny na błędy i trudny w debugowaniu [18,19,20].

[Michał Semik]

Mechanizm serializacji

Proces przekształcenia obiektów z zachowaniem ich aktualnego stanu na postać binarną lub znakową w sposób, który umożliwia ich późniejsze odtworzenie nazywa się serializacją. Proces odwrotny określa się jako deserializację. Mechanizm serializacji stosowany jest najczęściej w celu utrwalenia obiektu w pliku dyskowym, przesłania do innego procesu lub innego komputera poprzez sieć. W przypadku języka programistycznego Java, serializacja obiektów danej klasy jest możliwa po implementacji interfejsu `Serializable` z pakietu `java.io`. Obiekty klas, które nie implementują tego interfejsu nie będą mogły być poddane serializacji (ich stan nie będzie serializowany ani deserializowany). Wprawdzie powstały biblioteki umożliwiające serializację obiektów klas, które nie implementują interfejsu `Serializable`, jednak w przypadku wbudowanego mechanizmu serializacji Javy jest to wymagane [21].

[Alan Hawrot]

Skalowanie horyzontalne i wertykalne

Są to terminy związane z systemami rozproszonymi i dotyczą możliwości rozbudowy systemu w przypadku zwiększonego zapotrzebowania na zasoby sprzętowe lub zasoby programu.

- Skalowanie horyzontalne polega na dołączeniu do systemu kolejnego węzła, którym może być komputer w przypadku klastra komputerowego. Wydajność klastra w większości wypadków przewyższa wydajność pojedynczego komputera. Dzięki zastosowaniu klastrów i skalowania horyzontalnego możliwe jest wykonywanie obliczeń, które kiedyś mogły być wykonane tylko na superkomputerach.

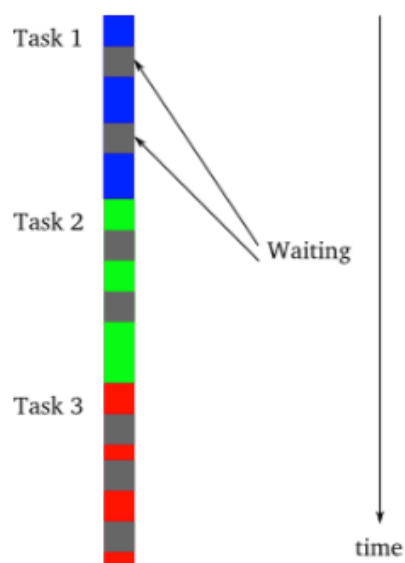
- W przypadku mniejszego zapotrzebowania na zasoby skalowanie polega na odłączeniu węzłów lub zasobów pojedynczego węzła [22].

Programowanie asynchroniczne



19

instrukcji, synchronizacji lub różnego rodzaju blokad, programista może kontrolować wykonywanie wątków, lecz to znacznie komplikuje program. Należy jednak zaznaczyć, iż w modelu asynchronicznym programista jest odpowiedzialny za podział instrukcji na mniejsze części, które mogą być wykonywane naprzemiennie. Jeżeli dana instrukcja korzysta z rezultatów innej - musi być napisana w taki sposób, aby przyjmowała rezultat w postaci wyników cząstkowych.



Rysunek 6. Blokowanie w synchronicznym programie [11].

Wydawać by się mogło, iż zastosowanie modelu asynchronicznego również niepotrzebnie komplikuje program, gdyż czas potrzebny na wykonanie wszystkich instrukcji jest taki sam jak w przypadku jednowątkowego modelu synchronicznego. Jest jednak sytuacja, w której można zauważyć jego znaczną przewagę. Jeżeli w danym programie synchronicznym wykonywanych jest wiele operacji blokujących, to program ten większość czasu spędzi na oczekiwaniu. Wykonywanie danej instrukcji nie może zostać kontynuowane, gdyż oczekuje ona na spełnienie jakiegoś zewnętrznego warunku. Często jest to zakończenie operacji wejścia-wyjścia. Na czas operacji wejścia-wyjścia, program jest zablokowany, a procesor w tym czasie nie wykonuje żadnych instrukcji. Dzięki zastosowaniu modelu asynchronicznego, gdy dochodzi do operacji blokującej i oczekiwania, procesor wykona inną instrukcję programu, która w danej chwili może być kontynuowana. Program zostanie zablokowany tylko wtedy, gdy żadna z instrukcji nie może być kontynuowana. Powracając jeszcze do porównania z wielowątkowym modelem – dlaczego dla każdej operacji blokującej nie stworzyć osobnego wątku? Jeżeli dany wątek jest zablokowany – inny może zostać wznowiony. Poza wcześniej

wspomnianymi trudnościami niestety istnieje dodatkowy narzut związany z tworzeniem nowych wątków, jak również zmianą kontekstu. Dla każdego nowego wątku musi zostać przydzielona pamięć związana z utrzymywaniem jego stanu. Ponadto przy zatrzymaniu i wznowieniu wątków następuje zmiana kontekstu związana np. z zapisem licznika rozkazów, stanu rejestrów procesora, stosu itp.

Jak widać zastosowanie modelu asynchronicznego przy pisaniu programów może przynieść znaczne korzyści. W porównaniu do modelu synchronicznego i wielowątkowego model asynchroniczny sprawuje się najlepiej, gdy:

- W programie istnieje duża liczba zadań, spośród których wykonywanie przynajmniej jednego może być kontynuowane w danej chwili.
- Zadania wykonują wiele operacji blokujących np. wejścia-wyjścia.
- Zadania są niezależne od siebie tzn. nie ma potrzeby wprowadzania dodatkowej komunikacji pomiędzy nimi lub jest ona niewielka, aby zadania nie czekały na siebie nawzajem i mogły być niezależnie wykonywane [11].

[Alan Hawrot]

Node.js

Node.js jest to środowisko uruchomieniowe JavaScript, które pozwala na uruchamianie skryptów poprzez wywołanie polecenia wraz z skrypcem podanym jako parametr. Node.js udostępnia szereg API, dostępnych z poziomu wykonywanych za pomocą środowiska skryptów. Niektóre z dostarczanych API umożliwiają dostęp do funkcjonalności systemu operacyjnego takich jak API systemu plików oraz obsługi procesów. Node.js udostępnia szereg API wspomagających tworzenie aplikacji serwerowych, czyli przede wszystkim API serwera http, https oraz inne pomocnicze, przykładowo do parsowania adresów URL.

Jednym z fundamentalnych elementów Node.js jest system zdarzeń, umożliwiający programowanie asynchroniczne. Duża liczba udostępnianych API, przede wszystkim te, które polegają na operacjach wejścia-wyjścia jest zbudowana w oparciu o system zdarzeń, dlatego też można powiedzieć, że Node.js wysoce wspiera programowanie asynchroniczne poprzez udostępnienie nieblokujących operacji wejścia-wyjścia.

System zdarzeń polega na emisji zdarzeń oraz obsługujących je wywołaniach zwrotnych. Zdarzenia obsługiwane są w jednym wątku przez wywołania zwrotne, znanym jako pętla zdarzeń. Odpowiedzialnością użytkownika środowiska jest implementacja wywołań zwrotnych, natomiast implementacja udostępnianych API dokonuje emisji zdarzeń. Niektóre z udostępnianych asynchronicznych operacji stosują wewnętrzną pulę wątków w celu ich realizacji, natomiast dostęp do tworzenia asynchronicznych zadań do wykonania w puli nie jest możliwy z poziomu użytkownika platformy [23].

Jedną z głównych zalet Node.js jest możliwość obsługi wielu żądań http poprzez zastosowanie tylko jednego wątku pętli zdarzeń. Dlatego też Node.js jest przeznaczone głównie do tworzenia aplikacji serwerowych. Możliwość obsługi wielu żądań zachodzi przede wszystkim dzięki wykorzystaniu nieblokujących bibliotek. W praktyce aplikacja użytkownika powinna zlecać wszystkie długo trwające operacje do wykonania przez asynchroniczne operacje. Główną wadą Node.js jest brak tolerancji na operacje ograniczone czasem procesora, które po prostu blokują jedyny wątek. Node.js umożliwia skalowanie wertykalne serwerów poprzez wykorzystanie modułu cluster. Moduł ten pozwala na tworzenie kopii procesów poprzez operację fork. Umożliwia również nasłuchiwanie żądań http na każdym z procesów jednocześnie, co pozwala na lepsze wykorzystanie zasobów maszyn wieloprocessorowych. Moduł ten nie umożliwia wykorzystania pamięci współdzielonej [24]. Domyślnie Node.js stosuje jeden wątek pętli zdarzeń także skalowanie wertykalne jest wyłączone [25,26,27,28].

[Michał Semik]

Apache Hadoop

Apache Hadoop jest to platforma służąca do rozproszonego składowania i przetwarzania dużej ilości danych. Jest napisana w języku Java. Składa się z następujących modułów:

- Hadoop Common – biblioteki i narzędzia używane przez pozostałe moduły.
- Hadoop Distributed File System (HDFS) – rozproszony system plików.
- Hadoop YARN – platforma do zarządzania zasobami klastra.
- Hadoop MapReduce – implementacja modelu MapReduce do przetwarzania dużej ilości danych.

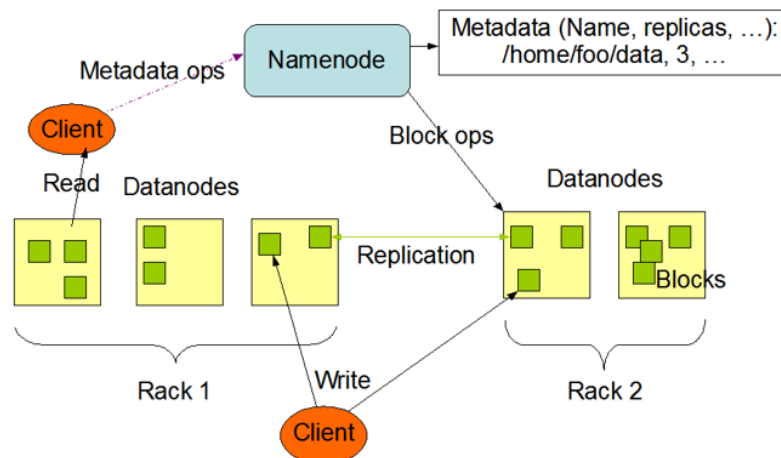
W tym rozdziale skupię się na opisie MapReduce i HDFS, głównie pod kątem zasady działania i architektury systemu [29].

[Michał Semik]

Hadoop Distributed File System

HDFS jest to implementacja rozproszonego systemu plików bazująca na Google File System. System składa się z węzłów działających w architekturze master/slave [30]. Z założenia może przetrzymywać pliki ogromnej wielkości; każdy plik jest dzielony na bloki. Bloki jednego pliku są równej wielkości (poza wielkością ostatniego bloku). Każdy z bloków znajduje się na co najmniej jednym węźle - system posiada mechanizm replikacji bloków. Użytkownik może określić ilość replik i wielkość bloku dla każdego pliku. Wyróżniane są dwa typy węzłów: NameNode(master) oraz DataNode(slave).

W klastrze istnieje dokładnie jeden węzeł NameNode, który jest odpowiedzialny za przetrzymywanie metadanych; przechowuje informacje na temat wszystkich plików w systemie i ich odwzorowania w bloki oraz informacje, na których węzłach znajdują się bloki. NameNode jest jedynym punktem awarii całego systemu [31]. Nie przechowuje bloków i dane plików użytkownika nigdy nie są przez niego przesyłane, tym zajmują się węzły typu DataNode. Węzły te okresowo wysyłają komunikat HeartBeat [32] do NameNode. Do NameNode wysyłają także komunikaty Blockreport, które informują o posiadanych blokach. Komunikacja pomiędzy węzłami oraz klientem systemu jest oparta o abstrakcję zdalnego wywołania procedury [33] bazującego na stosie protokołów TCP/IP. System zaprojektowany jest tak, że NameNode nie wysyła żądań do innych węzłów ani klientów, lecz nasłuchuje na przychodzące od nich żądania.



Rysunek 7. Architektura HDFS [34].

Użytkownikowi systemu udostępnione są operacje takie jak w tradycyjnym systemie plików, zapewniając przezroczystość przy dostępie do systemu. Ciekawą funkcjonalnością jest możliwość wykonania kodu programu na węzłach zawierających wymagane dane. Jest to oparte o ideę, że przenoszenie obliczeń jest szybsze niż przenoszenie danych [35,34].

[Michał Semik]

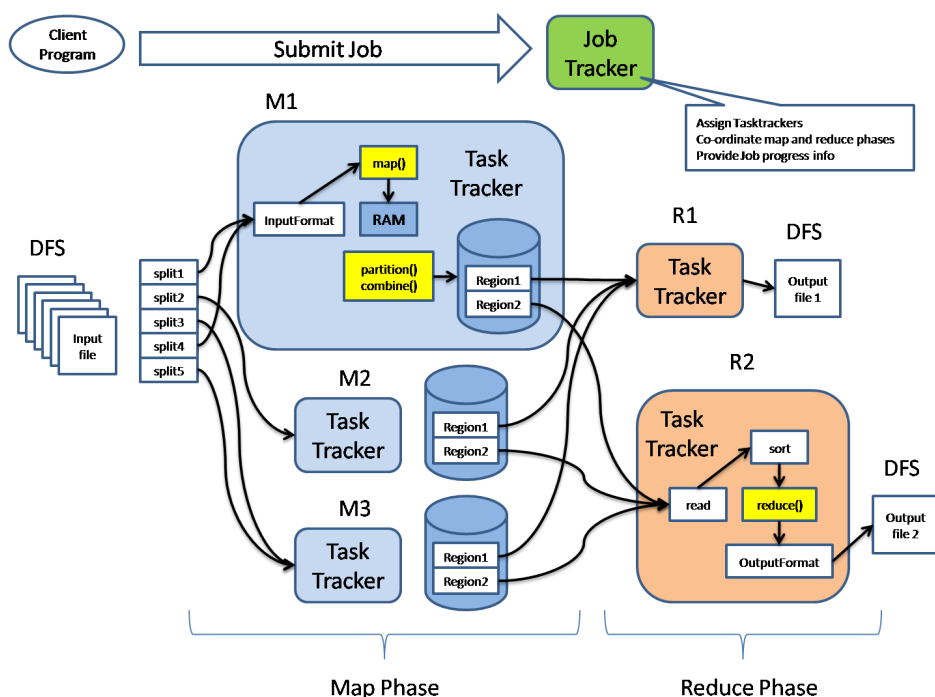
MapReduce

MapReduce jest to model obliczeniowy umożliwiający rozproszone przetwarzanie dużej ilości danych. Inspiracją modelu są powszechnie wykorzystywane w językach funkcyjnych operacje map i reduce. MapReduce wymyślono w Google natomiast implementacja w Hadoop jest jedną z najbardziej popularnych implementacji Open Source i na niej skupia się ten rozdział [36].

W najprostszym i najbardziej popularnym scenariuszu MapReduce na wejściu przyjmuje plik lub katalog plików, przetwarza je i na wyjściu otrzymujemy pliki. Przetwarzanie następuje przez operacje map, combine i reduce, które dostarczane są przez użytkownika. Dostarczanie operacji polega na implementacji odpowiednich interfejsów. Operacja map przyjmuje na wejściu parę składającą się z klucza i wartości i zwraca parę klucz, lista wartości. Parę klucz, lista wartości przyjmują i zwracają operacje combine i reduce.

System jest o bardzo podobnej architekturze jak HDFS, jest również oparty o technikę master/slave. Węzeł główny jest typu Job Tracker, węzły pomocnicze są typu Task Tracker. W klastrze powinien istnieć jeden węzeł główny oraz co najmniej jeden węzeł pomocniczy. Węzeł główny jest pojedynczym punktem awarii klastra. Przechowuje informacje o wolnych zasobach obliczeniowych (slotach), które znajdują się w węzłach pomocniczych. W slotach mogą być wykonywane taski. Zarówno operacja map, combine, jak i reduce jest taskiem. Węzeł główny zleca zadania do wykonania i ponawia w przypadku błędu wykonania lub awarii węzła pomocniczego. Węzły pomocnicze udostępniają informacje o ukończonych zadaniach węzłowi głównemu. Wysyłają również komunikat HeartBeat.

Użytkownik dostarcza operacje map, combine i reduce w archiwum jar do węzła Job Tracker zlecając prace do wykonania (job). Węzeł ten rozpropagowuje archiwum na wszystkie węzły Task Tracker. Przy zlecaniu pracy określa się w jaki sposób podzielić dane wejściowe na pary klucz-wartość. Przykładowy podział to wartość będąca linią pliku i klucz będący numerem linii. Inny podział może polegać na odseparowaniu n linii lub podzielenie plik po n bajtów na krotkę. Istnieje wiele typów podziałów danych wejściowych zaimplementowanych przez framework. Po podziale danych rozpoczyna się faza map, w której wykonywane są na parach operacje map. Gdy korzystamy z HDFS operacje map wykonywane są na węzłach „blisko danych” [37]. Można skorzystać z wejścia będącego tabelą w relacyjnej bazie danych, jednak z tego względu wydajność będzie mniejsza [38]. W trakcie fazy map периодически wykonują się operacje combine dla wyników z dotychczas wykonanych operacji map. Wynik operacji combine wysyłany jest od razu do węzła, w którym nastąpi operacja reduce dla danego klucza. Operacja combine ma służyć do wstępnej, lokalnej redukcji danych przed ich rozpropagowaniem. Po ukończeniu wszystkich operacji map i combine węzeł główny zleca rozpoczęcie fazy reduce. Wyniki tasków typu reduce zapisywane są do plików wyjściowych [39,40,41,42,43].



Rysunek 8. Węzły i ich odpowiedzialności w Apache Hadoop MapReduce [43].

[Michał Semik]

jNode

Przedmiot tej pracy jest systemem, który powstawał od 21 października 2015 roku, kiedy to zostało stworzone dedykowane repozytorium kodu źródłowego w serwisie GitHub. Repozytorium to znajduje się pod adresem <https://github.com/msemik/jNode>, gdzie kod źródłowy programu jest dostępny publicznie.

Głównym celem systemu jest umożliwienie użytkownikowi tworzenia i wykonywania programów w języku Java opartych o nieblokujące, asynchroniczne podejście. Dzięki zastosowaniu programowania asynchronicznego, programy działające na platformie jNode są bardziej wydajne od ich synchronicznych odpowiedników. jNode cechuje się zaawansowaną architekturą, która niesie za sobą zalety w postaci bardzo dobrej skalowalności zarówno wertykalnej, jak i horyzontalnej. Równocześnie stworzony system jest w pełni zautomatyzowany i bardzo wygodny w obsłudze.

System składa się z dwóch fundamentalnych elementów. Pierwszym jest środowisko uruchomieniowe, które umożliwia wykonywanie programów użytkownika. Drugim elementem jest biblioteka użytkownika, która służy aplikacji użytkownika do komunikacji z platformą.

Model obliczeniowy

Tworząc aplikację działającą z środowiskiem jNode, podstawowym pojęciem, którym się posługujemy są zadania, które są wykonywane asynchronicznie poprzez środowisko uruchomieniowe. Użytkownik poprzez bibliotekę użytkownika może zlecać zadania do wykonania. Aplikacja użytkownika jest udostępniana w postaci archiwum JAR (Java Archive) [44]. Aby uruchomić aplikację w platformie najpierw należy uruchomić platformę, co można zrobić poleceniem „jNode” z linii poleceń. Następnie należy skopiować archiwum do katalogu, nazywanego od tej pory JarPath. Jest to katalog domyślnie znajdujący się w katalogu głównym platformy. Archiwum, które zostanie skopiowane do tego katalogu zostaje automatycznie uruchomione. Dokładniej, zostaje uruchomiona funkcja main klasy zdefiniowanej w pliku manifest znajdującym się w archiwum [44]. Wywoływana funkcja main jest pierwszym zadaniem aplikacji użytkownika.

Każde zadanie po ukończeniu zwraca rezultat, który może być obiektem dowolnej klasy serializowalnej. Rezultat jest zlecony do obsłużenia przez aplikację użytkownika jako wywołanie zwrotne (ang. callback). W jNode każde zlecane zadanie dostarczane jest wraz z wywołaniem zwrotnym, pierwsze zadanie będące wywołaniem funkcji main zlecane jest z pustym wywołaniem zwrotnym.

Środowisko jNode posiada pętlę zdarzeń obsługującą (wykonującą) wywołania zwrotne, obsługiwaną przez jeden wątek. Rozwiązanie takie pozwala na bezpieczne wykonywanie wywołań zwrotnych, które mogą korzystać z współdzielonej pamięci bez jakiegokolwiek synchronizacji. Podejście to jest również znane jako wzorzec projektowy Reactor [45]. Zlecone zadania natomiast, wykonują się ze sobą współbieżnie, w związku z czym korzystanie przez nie z współdzielonej pamięci wymaga większej uwagi. Wykonywanie zadań następuje wewnątrz platformy poprzez pulę wątków, której ilość wątków można definiować przy uruchamianiu platformy.

jNode obsługuje dowolną ilość aplikacji użytkownika, przy czym każda z nich musi mieć inną nazwę. Aby uruchomić aplikację należy skopiować odpowiednie archiwum do

katalogu JarPath. Przy wykonywaniu wielu aplikacji, każda z nich otrzymuje odrębną pętlę zdarzeń do wykonywania wywołań zwrotnych, jednak wszystkie aplikacje korzystają z wspólnego silnika do wykonywania zadań.

[Michał Semik]

Skalowalność

jNode posiada cechę wyróżniającą go pośród systemów o zbliżonych funkcjonalnościach, którą jest możliwość skalowania zarówno wertykalnego, jak i horyzontalnego. Skalowalność wertykalną aplikacji w Javie jest stosunkowo łatwo osiągnąć, ponieważ wątki Javy korzystają z zasobów maszyn wieloprocessorowych [46]. Zatem wystarczy napisać dobrze aplikację wielowątkową, aby osiągnąć dobrą własność skalowalności wertykalnej. Natomiast jNode posiada także natywne wsparcie do skalowania horyzontalnego. Skalowanie można osiągnąć poprzez uruchomienie dodatkowych instancji platformy. Instancje te nazywamy węzłami. Wszystkie węzły w sieci łączą się w jeden klaster i współdzielą zasoby obliczeniowe. Ustanawianie połączenia pomiędzy węzłami zachodzi automatycznie bez ingerencji użytkownika. Współdzielenie czasu maszyn jest realizowane poprzez możliwość zlecania zadań do wykonywania innym węzłom. Zlecenie nazywamy delegowaniem zadań i również zachodzi w pełni automatycznie. Gdy węzeł posiada zaległe zadania w kolejce dokonuje wyboru najlepszych kandydatów, do których może delegować zadania. Wybór jest dokonywany za pomocą algorytmu bazującego na przybliżonym obciążeniu innych węzłów.

Skalowanie horyzontalne wprowadza do systemu programowanie rozproszone, które z kolei wprowadza problem zapewnienia niezawodności systemu. Mimo to jNode zapewnia własność wykonania wszystkich zadań składających się na program użytkownika. Jest ona zrealizowana poprzez wprowadzenie odpowiednich mechanizmów kontroli stanu i ponawiania zadań w całym klastrze. Własność ta jest zachowana pod warunkiem, że awarii nie ulegnie węzeł, na którym została uruchomiana aplikacja użytkownika. W takiej sytuacji pozostałe węzły anulują zadania pochodzące z aplikacji uruchomionych na węźle, który uległ awarii.

[Michał Semik]

Dependency Injection

W celu umożliwienia wygodnego tworzenia obiektowo orientowanych aplikacji użytkownika, system wspiera wzorzec dependency injection. Środowisko uruchomieniowe realizuje funkcjonalność kontenera dependency injection. Funkcjonalność ta jest analogiczna do udostępnianej przez Spring Framework oraz standard Context Dependency Injection w API Java Enterprise Edition, jednak jest bardzo okrojona z powodu dużej ilości pracy potrzebnej w celu realizacji pełnej funkcjonalności kontenera. W każdym razie istnieje możliwość automatycznego podłączenia przez platformę obiektów zarówno do zadań, jak i wywołań zwrotnych po zastosowaniu odpowiednich adnotacji przed typem pola. Funkcjonalność ta jest przydatna szczególnie do współdzielenia stanu pomiędzy zadaniami oraz wywołaniami zwrotnymi.

[Michał Semik]

Porównanie z Node.js

jNode był głównie inspirowany platformą Node.js. Podobieństwo pomiędzy systemami jest zachowane na poziomie modelu obliczeniowego, czyli również zostało zastosowane podejście oparte o jednowątkową pętlę obsługi zdarzeń, wraz z wykorzystaniem centralnej puli wątków do obliczeń asynchronicznych. Platformy są implementowane w zupełnie innych technologiach, umożliwiając wykonywanie aplikacji w innych technologiach oraz zastosowania platform również się nie pokrywają. jNode jest napisany całkowicie w języku Java i obsługuje aplikacje użytkownika w tym języku, natomiast Node.js jest napisany w JavaScript oraz C/C++ i umożliwia wykonywanie skryptów JavaScript. Jedna instancja platformy Node.js może wykonywać jeden skrypt użytkownika, natomiast jedna instancja jNode obsługuje ilość aplikacji ograniczoną możliwościami sprzętowymi. jNode udostępnia użytkownikowi możliwość wykonywania dowolnych zadań w scentralizowanej puli, natomiast Node.js udostępnia szereg wyspecjalizowanych operacji, a dostęp do scentralizowanej puli nie jest możliwy, jak było wspomniane przy okazji omawiania Node.js. jNode udostępnia wbudowany mechanizm skalowania horyzontalnego natomiast Node.js nie posiada takiego wsparcia. W Node.js sugerowane jest wykorzystanie techniki load balancing [47] w celu uzyskania skalowania horyzontalnego, jednak w praktyce umożliwia to skalowanie żądań http, natomiast jNode może skalować natywnie wykonywanie zadań, co kreuje potencjał do zastosowania platformy jNode również przy rozwiązywaniu problemów obliczeniowych, pozostawiając zdolność do obsługi dużej ilości operacji wejścia/wyjścia.

Porównanie z Apache Hadoop

jNode podobnej funkcjonalności do Hadoop Distributed File System nie posiada, dlatego też skupię się na MapReduce. Architektura jNode nie ma pojedynczego punktu awarii, w przeciwieństwie do MapReduce system może funkcjonować po utracie dowolnego węzła. W jNode zatrzymają się jedynie aplikacje uruchomione z węzła, który uległ awarii. W obecnej wersji systemu jest możliwe dodanie funkcjonalności wznawiania aplikacji na innym węźle. Kluczową różnicą jest, że MapReduce służy do przeanalizowania dużego zbioru danych w celu otrzymania wyjścia programu. jNode ma właściwości pozwalające na pisanie aplikacji, które mają za zadanie pracować nieustannie. W kontekście Big Data [48], istnieje podział systemów na operacyjne i analityczne. Systemy operacyjne to takie, które służą do przetwarzania danych w czasie rzeczywistym. Przykładowo są to bazy NoSQL. W drugim typie systemów wykonywanie programu może trwać długo. MapReduce jest systemem klasyfikowanym jako analityczny. Jak się do tego ma jNode? jNode można wykorzystać do pisania aplikacji opartych o architekturę klient-serwer [49], stąd można wyciągnąć wnioski, że jNode ma silne właściwości operacyjne. Jednocześnie posiada możliwość przetwarzania równoległego jak MapReduce. Być może po integracji jNode z rozproszonym systemem plików - możliwości analityczne byłyby porównywalne z Apache Hadoop [50].

[Michał Semik]

Zastosowane technologie, frameworki i biblioteki

Spring Framework

Spring Framework wspiera tworzenie aplikacji w języku Java. Framework ten posiada wiele funkcjonalności, a wykorzystywane w naszej aplikacji są jego skromną częścią. Mimo to wykorzystane funkcjonalności były kluczowe i bez nich stworzenie podobnego systemu wymagałoby dużego nakładu pracy. Fundamentalnym elementem jNode dostarczanym przez Spring Framework jest zastosowanie odwrócenia sterowania poprzez dependency injection. W tym celu Spring Framework udostępnia moduł implementujący funkcjonalność kontenera dependency injection [51]. Drugą znaną funkcjonalnością jest dostarczana implementacja

wzorca obserwator [52]. Implementacja ta jest bardzo wygodna, ponieważ pozwala na odwrócenie zależności [53] z praktycznie zerowym nakładem pracy. Poprzez wykorzystanie tych dwóch mechanizmów udało się nam stworzyć aplikację modułarną [54].

[Michał Semik]

Maven, czyli zautomatyzowane zarządzanie projektem

Maven, jest narzędziem służącym do zarządzania projektem. Umożliwia przede wszystkim automatyzację i zarządzanie procesem budowania aplikacji na platformę Java oraz udostępnia mechanizm zarządzania zależnościami projektu.

Maven wprowadza istotne pojęcie modułu. Moduł jest to zbiór klas, który posiada własne zależności i moduły. W odróżnieniu od modułu zawierającego kod źródłowy, zależność jest zbudowaną aplikacją, nazywaną artefaktem. Aby zbudować całą aplikację należy zbudować wszystkie moduły w odpowiedniej kolejności, zaczynając od tych, dla których wszystkie podmoduły są już zbudowane. Maven automatycznie realizuje ten proces. Podczas budowania aplikacji, klasy zależnych podprojektów są przechodnio udostępniane, jednak istnieją mechanizmy, które pozwalają na kontrolę przechodniości [55].

Modularyzacja aplikacji jest jednym ze sposobów na osiągnięcie programowania komponentowego [56]. Obecnie nie ma wbudowanego w tym języku podobnego mechanizmu, natomiast istnieją plany wprowadzenia pojęcia modułów [57]. Modularyzacja aplikacji pozwala na izolację grup klas, co posiada różne zalety. Dla nas najważniejsze jest, że ułatwia rozumienie programów. W pracy wykorzystaliśmy ten fakt generując za pomocą środowiska deweloperskiego diagramy modułów/komponentów i ich zależności. Z pomocą wygenerowanych diagramów zostanie wytłumaczona zasada działania systemu. Do innych zalet należy zachowanie luźnych powiązań [58], tzn. zmniejszenie wzajemnej wiedzy klas. Relacje pomiędzy klasami są kontrolowane poprzez granice wyznaczone modułami. Polepszona jest własność do ponownego użycia kodu. Ułatwione jest wprowadzenie zmian w systemie poprzez możliwość podmiany całych komponentów.

[Michał Semik]

Programowanie aspektowe (AOP) to paradygmat programowania wspomagający separację zagadnień i wydzielenie w programie niezwiązanych ze sobą funkcjonalnie części. Do typowych mechanizmów obiektowych dodaje koncepcję aspektu, który reprezentuje pojedyncze zagadnienie, grupuje kod i reguły jego łączenia z innymi fragmentami programu. Programowanie aspektowe umożliwia dodanie funkcjonalności do programu bez modyfikacji samego kodu. Dzięki temu takie funkcje jak np. autoryzacja, autentykacja, walidacja, logowanie, transakcje, caching czy inne, których kod jest powtarzany i wywoływany w wielu miejscach systemu, mogą zostać wyodrębnione do osobnych jednostek modularyzacji. Pozwala to zachować czytelność kodu systemu. Klasy reprezentujące zagadnienia biznesowe nadal zawierają tylko to, za co odpowiadają. Aspekty, stosowane obok klas, umożliwiają zatem lepszą, wielokryterialną strukturalizację systemu informatycznego niż użycie samych tylko klas [59].

[Alan Hawrot]

Aspect Weaving

Proces tkania (ang. *weaving*) jest to proces, w którym zachodzi łączenie kodu aspektów z pozostałymi częściami systemu. Proces ten może zachodzić podczas trzech różnych etapów:

- Compile-time – to najprostsze podejście. Specjalny kompilator otrzymuje na wejściu kod źródłowy klas i na wyjściu produkuje skompilowane, zmodyfikowane definicje klas. Same aspekty mogą być w postaci binarnej lub kodu źródłowego.
- Post-compile-time – stosowany do łączenia aspektów z już skompilowanymi klasami, archiwami jar. Proces łączenia zachodzi poprzez modyfikację bajtkodu. Łączone aspekty nadal mogą być w postaci kodu źródłowego lub binarnej.
- Load-time – zachodzi podczas ładowania definicji klas do maszyny wirtualnej. W tym celu musi zostać wykorzystany specjalny class loader.

W jNode zostało wykorzystane Compile Time Weaving (CTW) przy użyciu wtyczki Maven, która korzysta z kompilatora udostępnianego przez AspectJ (ajc). Było to konieczne, aby zapewnić lepszą komunikację między aplikacją kliencką a platformą. Niektóre z obiektów w aplikacji klienckiej są tworzone poprzez bezpośrednie wywołanie konstruktora, poza

kontrolą kontenera Spring IoC. W celu zapewnienia komunikacji tych obiektów z obiektami platformy zachodzi jednak potrzeba potraktowania ich jako beany i umożliwienie przeprowadzenia wstrzykiwania zależności. Dzięki aspektom i compile time weaving zostało to osiągnięte [60].

[Alan Hawrot]

JGroups

JGroups to narzędzie do zapewnienia niezawodnej komunikacji w sieci komputerowej. Zostało napisane w całości w języku Java. Narzędzie to może zostać wykorzystane do tworzenia klastrów, w których węzły komunikują się ze sobą poprzez wysyłanie komunikatów. Jego najważniejsze funkcjonalności to:

- Tworzenie/usuwanie klastrów w sieci LAN lub WAN.
- Możliwość dołączenia do klastra lub jego opuszczenia.
- Wykrywanie członkostwa i wysyłanie powiadomień na temat węzłów, które dołączyły do klastra lub go opuściły.
- Detekcja awarii węzłów. Automatyczne usuwanie węzłów, które uległy awarii i wysyłanie powiadomień do pozostałych członków klastra na temat usuniętego węzła.
- Wysyłanie i odbieranie komunikatów typu jeden-do-jeden.
- Wysyłanie i odbieranie komunikatów typu jeden-do-wielu.

Główną zaletą narzędzia JGroups jest jego elastyczny stos protokołów, który pozwala deweloperom na jego dokładne dostosowanie do wymagań aplikacji lub charakterystyk sieci komputerowej. JGroups udostępnia duży wybór różnych protokołów, które zostały już zaimplementowane. Niektóre z nich to:

- Protokoły transmisji danych: UDP lub TCP.
- Fragmentacja dużych komunikatów.
- Niezawodna transmisja komunikatów typu unicast lub multicast. Zagubione komunikaty są ponownie wysyłane.
- Wykrywanie awarii: węzły, które uległy awarii są wykluczone z członkostwa.
- Porządkowanie przesyłanych komunikatów: FIFO, Total Order.
- Protokoły dotyczące członkostwa, szyfrowania, czy też kompresji.

Do transmisji komunikatów domyślnie stosowany jest protokół UDP w przypadku komunikatów typu jeden-do-jeden lub UDP oparty o IP multicast w przypadku komunikatów typu jeden-do-wielu. Zwykle stosuje się UDP dla sieci LAN, natomiast dla sieci WAN używa się protokołu TCP, gdyż nie wszystkie sieci umożliwiają IP multicasting. Protokół UDP jest jednak zawodny – pakiety mogą zostać zagubione, zduplikowane, mogą zostać odebrane w złej kolejności, czy też istnieje limit na wielkość komunikatu. Z pomocą przychodzi wspomniany wcześniej elastyczny, w pełni konfigurowalny stos protokołów. Protokół transmisji danych znajduje się na samym dole stosu. By zapewnić niezawodność i spełnić inne wymagania do stosu przykładowo mogą zostać dodane protokoły:

- NAKACK – retransmisja zagubionych komunikatów, usuwanie duplikatów.
- FIFO – utrzymanie porządku typu FIFO przesyłanych komunikatów.
- GMS – wykrywanie członkostwa.
- FRAG2 – fragmentacja dużych komunikatów.
- FD/FD_ALL – detekcja awarii, otrzymywanie powiadomień o awariach.
- ENCRYPT – w celu zapewnienia szyfrowania.

Wysyłane i odbierane komunikaty przechodzą przez każdą warstwę stosu, gdzie są odpowiednio przetwarzane. Dostępnych protokołów i możliwości konfiguracji jest naprawdę wiele. Programista ma również możliwość napisania własnego protokołu, który może zostać dodany do stosu. Zatem zastosowanie narzędzia JGroups w celu konfiguracji klastrów i zapewnienia niezawodnej komunikacji pomiędzy węzłami było rozsądnym wyborem przy tworzeniu systemu jNode [61].

[Alan Hawrot]

Non-Blocking I/O

Non-Blocking I/O to kolekcja interfejsów programistycznych aplikacji Javy (pakiet java.nio) wprowadzona do wersji 1.4 Java SE i rozszerzona w wersji 1.7 o nową obsługę systemu plików. Główną cechą pakietu jest zapewnienie nieblokujących operacji wejścia-wyjścia w oparciu o kanały, selektory i zastosowanie buforów. Dzięki temu pojedynczy wątek może monitorować wiele kanałów, które zarejestrowane są u jednego selektora. Selektor wybiera kanał, który jest w danej chwili dostępny w celu zapisu lub odczytu danych. Podejście nieblokujące powoduje, że przykładowo podczas odczytu danych z danego kanału wątek nie

zostanie zablokowany, gdy w określonej chwili nie ma żadnych danych do odczytu. Zamiast tego wątek może obsługiwać inny kanał w tym czasie [62].

[Alan Hawrot]

WatchService

WatchService jest to serwis wchodzący w skład pakietu `java.nio.file`, który służy do obserwowania zarejestrowanych obiektów w celu detekcji zmian i zdarzeń. Przykładowo może służyć do obserwowania katalogu, by wykryć zachodzące w nim zmiany takie jak utworzenie pliku lub jego usunięcie. Wykorzystanie WatchService polega na zarejestrowaniu obiektów typu `Watchable`. Po wystąpieniu zdarzenia zostaje zasygnalizowany i umieszczony w wewnętrznej kolejce WatchService obiekt typu `WatchKey` reprezentujący dany, zarejestrowany wcześniej obiekt, dla którego wystąpiło wspomniane zdarzenie. Gdy WatchService jest gotowy do przetwarzania zdarzeń, wywołuje metodę *take* lub *poll*. Jeżeli w trakcie wywołania tej metody w kolejce nie znajdują się żadne obiekty `WatchKey` do przetworzenia, WatchService zawiesza się, oczekując na wystąpienie zdarzeń. Po ewentualnej obsłudze zdarzenia wywoływana jest metoda *reset* obiektu `WatchKey`, by ten mógł być ponownie zasygnalizowany i zakolejkowany. Ze względu na to, iż w plikach systemowych zdarzenia mogą występować szybciej niż ich obsługa przez udostępnioną implementację, pozwala ona na akumulację zdarzeń i ich zbiorowe przetwarzanie. Każde zdarzenie jest reprezentowane przez obiekt typu `WatchEvent` [63].

[Alan Hawrot]

java.util.concurrent

Pakiet `java.util.concurrent` zawiera narzędzia wspierające programowanie współbieżne. Zestaw narzędzi jest bardzo duży, wymienimy przede wszystkim te, które zostały przez nas wykorzystane. Biblioteka ta udostępnia szereg kolekcji umożliwiających wykonywanie operacji wielowątkowych. Jedną z wykorzystanych kolekcji jest `PriorityBlockingQueue`, która jest nieograniczoną kolejką priorytetową, implementującą semantykę blokowania w przypadku, gdy kolejka jest pusta. Kolekcja ta przydała się do szeregowania operacji wykonywanych przez pulę wątków. Wykorzystaliśmy dwie implementacje puli wątków - `ScheduledThreadPoolExecutor` i `ThreadPoolExecutor`. `ThreadPoolExecutor` umożliwia jedynie

wykonywanie zadań. `ScheduledThreadPool` wyróżnia się tym, że pozwala na wykonywanie zadań okresowo, lub jednorazowo po upływie pewnego czasu. Wykorzystaliśmy ją niejawnie, poprzez API dostarczane przez Spring. Pule pozwalają na dynamicznie tworzenie wątków. Ilość tych wątków jest ograniczona poprzez przedział podawany przy tworzeniu puli. Kolejnym elementem pakietu jest podpakiet `atomic`, zawierający typy umożliwiające atomowe i nieblokujące operacje. Dla nas szczególnie przydatny był `AtomicReference`, który pozwala na atomową podmianę obiektu. Wszystkie klasy w tym pakiecie bazują na atomowej operacji `compareAndSwap`, która pozwala na warunkową podmianę wartości. Przykładowo, można zaimplementować atomową, nieblokującą operację transform jak w przykładzie poniżej.

```
8  long getAndTransform(AtomicLong var) {
9      long prev, next;
10     do {
11         prev = var.get();
12         next = transform(prev);
13     } while (!var.compareAndSet(prev, next));
14     return prev; // return next; for transformAndGet
15 }
```

Kod źródłowy 3. Metoda `getAndTransform` [64].

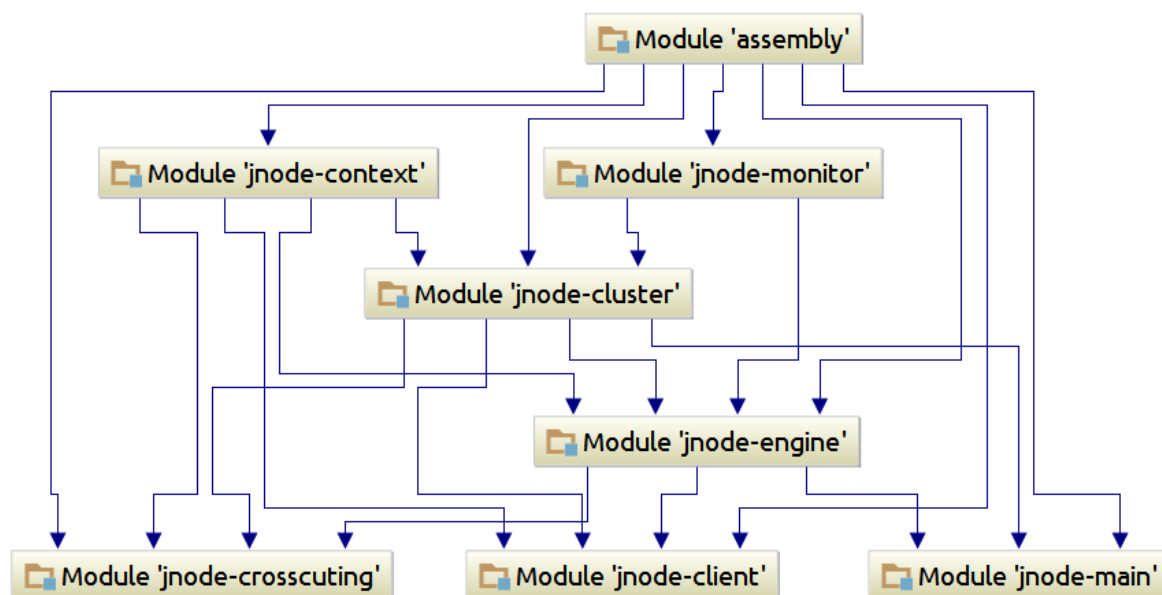
Schemat ten można wykorzystać do implementacji wielu podobnych operacji i okazał się kluczowy przy implementacji maszyny stanowej [65,64].

[Michał Semik]

Architektura systemu i algorytmy

Jak już wcześniej wspomniano w rozdziale *Maven*, czyli *zautomatyzowane zarządzanie projektem*, korzystając z tego narzędzia stworzony został system w pełni modułarny. Niesie to ze sobą szereg zalet.

Modularne aplikacje są łatwiejsze w utrzymaniu, zrozumieniu i analizie. Dzięki podziałowi aplikacji na moduły ograniczamy możliwości stworzenia zależności pomiędzy klasami. Klasy jednego modułu mogą korzystać z klas drugiego modułu, ale zostało to tylko jawnie uwzględnione w pliku `pom.xml`.



Rysunek 9. Diagram komponentów platformy jNode.

Powyższy diagram przedstawia komplet modułów należących do systemu. Strzałki reprezentują zależności pomiędzy modułami.

Dzięki architekturze modularnej zyskujemy również poprawę tzw. Reusability [66], czyli możliwości ponownego wykorzystania kodu aplikacji. Przykładowo moduł engine wraz z jego zależnościami pozwala na skompilowanie samowystarczalnej platformy do realizowania obliczeń na jednym węźle. Skompilowanie aplikacji dodatkowo z modulem cluster dodaje możliwość przetwarzania na klastrze węzłów. Dzięki takiemu podziałowi możemy dopisać dodatkowy moduł cluster2 oparty o inny protokół, przykładowo http, wykorzystujący moduł engine wraz z jego zależnościami. Sposób komunikacji pomiędzy tymi modułami będzie dokładniej omawiany w rozdziale *Moduł engine* oraz *Moduł cluster*. Komunikacja jest na tyle elastyczna, że pozwala na dodanie dodatkowego modułu cluster2 współdziałającego wraz z cluster1.

W wyższych warstwach systemu znajdują się moduły context oraz monitor. Moduł context dostarcza do systemu nieco uproszczoną funkcjonalność kontenera wstrzykiwania zależności. Natomiast monitor służy do okresowego badania stanu w całym systemie. Oba te moduły są opcjonalne w systemie i ich obecność nie jest konieczna z perspektywy działania systemu. Są one jednak przydatne do tworzenia czytelnych, obiektowych aplikacji klienckich oraz analizy stanu technicznego węzłów.

Moduł assembly zawiera zależności na wszystkie moduły systemu. Wynika jedynie z kwestii technicznych, ze sposobu w jaki Maven umożliwia zbudowanie archiwum aplikacji wielomodułowej. Wygodnie jest dziedziczyć przez podprojekty całą konfigurację projektu. W tym wszystkie zależności z projektu głównego. Natomiast żeby zbudować z projektu archiwum zawierające wszystkie klasy, trzeba dodać zależności na wszystkie podprojekty. Gdyby dodać wszystkie zależności do projektu głównego spowodowałoby to, że każdy podprojekt zna każdy inny podprojekt, co całkowicie psuje podział komponentów.

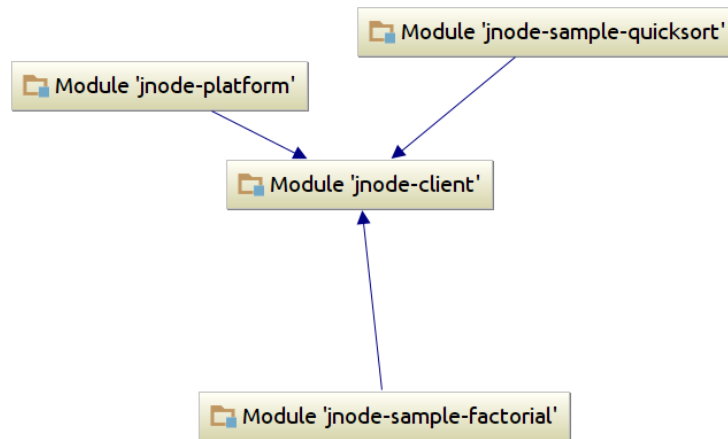
Pozostałe moduły będą również omówione w odpowiednich rozdziałach. Moduł crosscutting zawiera jedynie narzędzia pomocnicze, wykorzystywane w innych modułach. Nie realizują one logiki związanej z którymkolwiek z modułów, a jednak przydają się w różnych. Realizują one tzw. zagadnienia przekrojowe (ang. cross-cutting concern) [67]. Moduł main służy do inicjalizacji platformy, natomiast moduł client jest częścią wspólną pomiędzy aplikacją kliencką a platformą, przede wszystkim umożliwia tworzenie nowych zadań.

[Michał Semik]

Moduł client

Moduł client jest jednym z najprostszych w implementacji elementów systemu, mimo to jest on bardzo istotny, ponieważ jest to pośrednik pomiędzy aplikacją kliencką a platformą.

Program użytkownika korzystający z platformy jNode, np. program sortujący tablicę lub liczący silnię musi mieć dostęp do API umożliwiającego zgłaszanie do wykonania asynchronicznych zadań. Jednocześnie do działania modułu context wymagane jest oznaczenie klas adnotacjami informującymi gdzie wstrzyknąć zależności. Zatem jest to komponent pozwalający na wzajemną komunikację aplikacji klienckich i platformy. Zależności klas tych komponentów obrazowane są przez poniższy diagram.

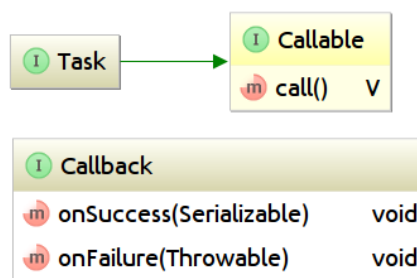


Rysunek 10. Przykład wejściowych zależności modułu client.

[Michał Semik]

Tworzenie zadań i wywołania zwrotne

Podstawową jednostką pracy w jNode jest task (zadanie). Task jest reprezentowany poprzez interfejs i każde zlecane zadanie powinno go implementować. Każdemu rejestrowanemu w systemie zadaniu towarzyszy implementacja interfejsu Callback, której metoda onSuccess zostanie wykonana po prawidłowym wykonaniu zadania, a w przypadku błędu wykonana - onFailure.



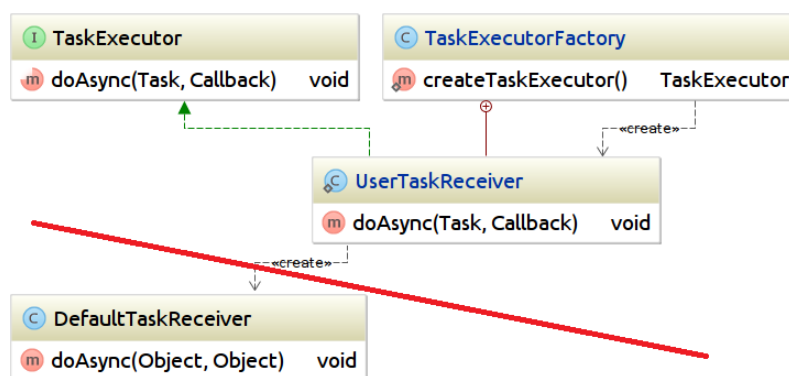
Rysunek 11. Interfejsy Task oraz Callback.

[Michał Semik]

Rejestrowanie zadań

Aplikacja użytkownika po utworzeniu zadania powinna go zarejestrować w systemie. Aby tego dokonać wywoływana jest specjalna metoda doAsync, która realizuje paradygmat programowania asynchronicznego.

Dokładniej omawiając, najpierw użytkownik za pomocą fabryki pobiera obiekt implementujący `TaskExecutor`. Obiekt ten (obiekt klasy `UserTaskReceiver`) podczas konstrukcji szuka klasy `DefaultTaskReceiver` po nazwie, korzystając z mechanizmu refleksji. Klasa ta, na poniższym diagramie oddzielona czerwoną linią, nie znajduje się w module client, a w engine. Mając już referencje do obiektu klasy `DefaultTaskReceiver`, obiekt klasy `UserTaskReceiver` deleguje do niej każde wywołanie `doAsync`, również poprzez mechanizm refleksji. Każdy obiekt klasy `DefaultTaskReceiver` dzięki AspectJ i mechanizmowi *Compile Time Weaving* jest beanem springowym, nawet jeśli został stworzony poprzez bezpośrednie wywołanie konstruktora. Skoro obiekt klasy `DefaultTaskReceiver` jest springowym beanem, to potrafi się komunikować z resztą platformy jNode. Więcej na ten temat jest w rozdziale *AOP*, *Compile Time Weaving* oraz w rozdziale *Moduł engine*.



Rysunek 12. Delegacja tasku od użytkownika do platformy.

[Michał Semik]

Adnotacje

Jak na początku rozdziału było powiedziane, nie tylko aplikacja użytkownika przekazuje informacje do platformy. Platforma również sama pozyskuje informacje z aplikacji użytkownika poprzez przeanalizowanie jej klas pod kątem występowania adnotacji dostarczanych przez bibliotekę client. Temat przedstawionych tutaj adnotacji będzie również poruszony w rozdziale *Moduł context*, który odpowiada za analizę tych adnotacji i wstrzykiwanie zależności do tasków i callbacków. Tutaj jedynie zostaną krótko wymienione udostępnione adnotacje:

- `@ContextScan` – jest to adnotacja, która powinna zostać umieszczona nad klasą zawierającą funkcję `main`. Adnotacja ta jako parametr przyjmuje listę prefixów pakietów, których klasy będą skanowane przez moduł `context`.
- `@Context` – do wykorzystania przy definicji typu. Określa, że dany typ jest beanem `jNode`.
- `@InjectContext` – do wykorzystania nad polem. Jest to deklaracja oznajmiająca, że bean o typie odpowiadającym typowi tego pola powinien być wstrzyknięty do taska lub callbacka.

[Michał Semik]

Moduł engine

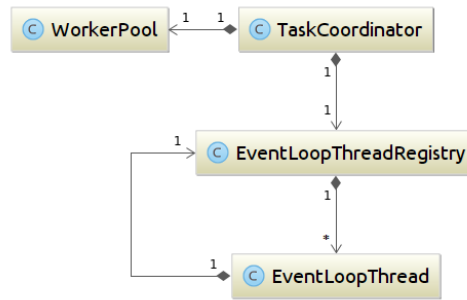
Rozpoczynanie obliczeń

Moduł `engine` jest odpowiedzialny za wykonywanie tasków oraz powiązanych callbacków zleconych od użytkownika. Początkowym, pierwszym zadaniem jest wykonanie metody `main`, której klasa została zdefiniowana w pliku manifest zawartego w archiwum `jar` przekazanego do `jNode`. Każde kolejne zadanie pochodzi już od użytkownika, na zasadzie omawianej przy okazji modułu `client`, zostaje wywołana metoda `doAsync`. Wykonywanie kodu użytkownika może się odbywać w dwóch typach wątków. Pierwszym typem jest pula wątków, gdzie trafiają wszystkie taski. Oprócz tego, na każdy program użytkownika przypada jeden, dedykowany wątek, który wykonuje callbacki na rezultatach otrzymywanych po wykonaniu tasków.

[Michał Semik]

Najważniejsze klasy

Moduł `engine` jest dość rozbudowanym podsystemem, natomiast jest kilka podstawowych klas, które zarządzają wykonywaniem obliczeń. Są one przedstawione na poniższym diagramie i zostaną opisane w kolejnych rozdziałach.



Rysunek 13. Diagram reprezentujący wybrane klasy modułu.

[Michał Semik]

TaskCoordinator

TaskCoordinator jest mediatorem tasków. Odbiera nowe taski, zarówno po otrzymaniu nowego archiwum jar, jak i taski zlecone z aplikacji użytkownika. Przekazuje taski do wykonania i oczekuje na ich wykonanie. Po ich wykonaniu przekazuje rezultat do EventLoopThread. Poza zależnościami objętymi diagramem, komunikacja z innymi elementami systemu oparta jest o system zdarzeń udostępniony przez Spring Framework. Zatem komunikacja z zewnętrznymi elementami następuje poprzez nadejście zdarzeń zgłoszenia taska i jego zakończenia, zdarzenie otrzymania nowego archiwum jar lub jego usunięcia. Dzięki zastosowaniu zdarzeń TaskCoordinator utrzymuje niezależność modułu engine od innych modułów, w szczególności od client i cluster.

[Michał Semik]

EventLoopThread i EventLoopThreadRegistry

EventLoopThread jest to klasa, która utrzymuje wątek per archiwum jar. W dedykowanym wątku sekwencyjnie wykonują się wszystkie callbacki dla wykonanych już tasków. Koncepcja ta została wykorzystana pierwszy raz w Node.js. Podejście to jest bardzo dobre ze względu na to, że wszystkie callbacki mogą korzystać z współdzielonej pamięci bez synchronizacji. Wątek EventLoopThread zamyka się w momencie, gdy wszystkie callbacki dla wszystkich tasków zostały wykonane. EventLoopThreadRegistry zawiera wszystkie EventLoopThready w systemie i umożliwia dostęp do nich. Obie klasy współpracują ze sobą w celu poprawnego zwolnienia zasobów podczas awarii, a także przy poprawnym zakończeniu wątku. O obsłudze awarii będzie mówione w rozdziale *Obsługa błędów*.

WorkerPool

Każdy utworzony w systemie task trafia ostatecznie do WorkerPool. Jest to główny silnik jNode, współdzielony przez wszystkie aplikacje użytkownika. Jak sama nazwa wskazuje jest to pula wątków, w której każdy z wątków wykonuje taski. Wielkość puli tzn. liczba dostępnych wątków w systemie może zostać zdefiniowana przez użytkownika podczas uruchamiania węzła. W sytuacji, gdy użytkownik nie poda tego argumentu, system automatycznie utworzy pulę o wielkości równej liczbie udostępnianych przez procesor wątków. Wszystkie dostępne opcje uruchomieniowe zostaną omówione w rozdziale *Moduł main*.

Zostało wspomniane, iż WorkerPool jest współdzielona przez wszystkie aplikacje, a więc różne wątki mogą wykonywać w jednej chwili taski pochodzące z różnych aplikacji. Po wykonaniu każdego taska, WorkerPool przy pomocy systemu zdarzeń udostępnionego przez Spring Framework ogłasza zdarzenie zakończenia taska. W zdarzeniu tym umieszcza rezultat obliczeń, wykonania kodu taska. Rezultatem może być dowolny obiekt klasy zdefiniowanej przez użytkownika lub jednej z klas standardowych, dostępnych w Javie. Zastrzeżenie jest takie, iż musi to być klasa serializowalna. Jest to wymagane i zostanie szerzej omówione w rozdziale *Moduł cluster*. W przypadku, gdy podczas wykonywania kodu taska wystąpił błąd, wyjątek, to rezultatem będzie ten wyjątek. Pozwala to na późniejsze, odpowiednie zareagowanie i obsługę tej sytuacji podczas wykonywania powiązanego z taskiem callbacka w EventLoopThread.

Jak już zostało powiedziane, pula wątków ma stałą, określoną wielkość i w sytuacji, gdy liczba zdefiniowanych tasków jest większa od liczby wątków, WorkerPool będzie kolejować następne, przychodzące taski i każdy z tasków będzie oczekiwać w kolejce na zwolnienie któregoś z wątków. Ponadto WorkerPool ogłasza zdarzenie przepełnienia puli wątków. Obsługa tego zdarzenia będzie omówiona w rozdziale *Moduł cluster*. WorkerPool reaguje również na zdarzenia anulowania archiwum jar, czy też tasków. Mechanizm ten jest opisany w rozdziale *Obsługa błędów*.

Modyfikacja implementacji puli wątków dostarczonej przez Spring Framework

Do zrealizowania puli wątków została wykorzystana głównie implementacja udostępniona przez Spring Framework. Pula składa się z wątków wykonujących taski i kolejki tasków oczekujących na wykonanie. W systemie jNode pula ta została skonfigurowana w następujący sposób:

- Pula jest typu *fixed-size* tzn. liczba wątków w puli jest zawsze stała. Wielkość może zostać określona przez użytkownika podczas uruchamiania instancji jNode. W przypadku, gdy nie zostanie przez niego określona, liczba wątków w puli będzie równa liczbie wątków udostępnianych przez procesor.
- Kolejka jest typu *unbounded* tzn. kolejka nie ma zdefiniowanej wielkości, jej wielkość jest ograniczona wielkością dostępnej pamięci operacyjnej.

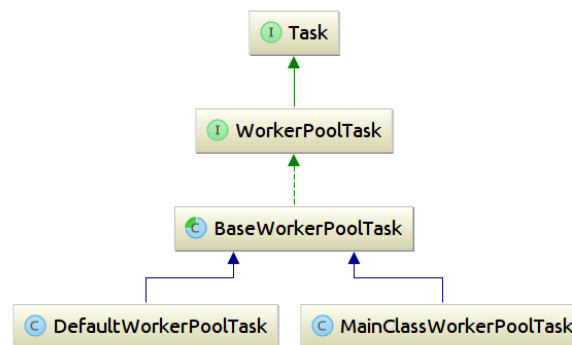
Domyślną kolejką typu *unbounded* w dostarczonej przez Spring Framework implementacji jest kolejka `LinkedBlockingQueue`. Jest to kolejka blokująca, typu FIFO (first-in-first-out). Oznacza to, że oczekujące taski są kolejgowane względem tego w jakiej kolejności zostały przekazane do `WorkerPool`, a więc kolejność wykonania tasków oczekujących jest zgodna z kolejnością umieszczenia tasków w kolejce.

W systemie jNode zastosowanie kolejki FIFO jest rozwiązaniem niewystarczającym i konieczne było wprowadzenie modyfikacji. System jNode w przypadku, gdy jest uruchomionych wiele instancji i w poszczególnych instancjach zgłaszane są zdarzenia przepełnienia `WorkerPool`, oczekujące taski mogą być przesyłane w celu wykonania do instancji, w których w danej chwili znajdują się wątki niewykonujące pracy. Mechanizm ten jest omówiony w rozdziale *Moduł cluster*. Istnieje jednak prawdopodobieństwo, że podczas przesyłania tasków pule wątków mogą zostać wcześniej zajęte przez inne taski, z innych źródeł. Powoduje to, że oczekujące na wykonanie taski będą ponownie rozsyłane, przekazywane do innych instancji. W pesymistycznym przypadku mogłoby to doprowadzić do tego, że niektóre taski mogą nigdy nie zostać wykonane. Jest to problem podobny do problemu *zagłodzenia* (ang. *starvation*). Aby uniknąć tej sytuacji i zapobiec temu problemowi w systemie jNode wprowadziliśmy priorytety dla tasków. Podczas ponownego przesyłania tasków pomiędzy instancjami ich priorytety są zwiększane. W `WorkerPool` zamiast kolejki *unbounded* FIFO zastosowaliśmy kolejkę priorytetową również typu *unbounded*. Dzięki temu oczekujące taski

są kolejgowane względem priorytetów i pewnym jest, że każdy z nich w końcu zostanie wykonany, a problem zagłódnienia jest rozwiązany.

[Alan Hawrot]

Struktura tasków na poziomie modułu engine



Rysunek 14. Diagram przedstawiający strukturę tasków na poziomie modułu engine.

Moduł engine obsługuje dowolną liczbę archiwów jar wykonywanych jednocześnie, bazując na jednej wspólnej puli wątków. Jak wiadomo w każdym z archiwów jar poszczególne taski mogą mieć różne implementacje. System jNode musi jednak obsługiwać te taski w jednakowy sposób, niezależnie od ich konkretnych implementacji i pochodzenia. W tym celu stworzyliśmy odpowiednią strukturę tasków.

Na samym szczycie znajduje się interfejs Task udostępniany użytkownikowi. Użytkownik wykorzystując ten interfejs i dostarczając konkretnych implementacji definiuje co ma zostać wykonane asynchronicznie. Jest to podstawowa jednostka pracy. Zostało to omówione w rozdziale *Moduł client*, jak również w rozdziale *Biblioteka użytkownika i przykładowe programy*, gdzie jest to opisane na konkretnych przykładach. Jednak to nie wszystko. Każdy rejestrowany w systemie jNode task musi mieć nadany unikalny identyfikator, jak również sam framework musi wiedzieć z jakiego archiwum jar pochodzi dany task, aby mógł po jego wykonaniu umieścić w odpowiednim EventLoopThread rezultat. Nabiera to również dodatkowego znaczenia w części rozproszonej, gdzie współpracuje ze sobą wiele instancji. Jest to omówione w rozdziale *Moduł cluster*. Jak zostało także wspomniane, każdy z tasków posiada priorytet. Aby spełnić te wszystkie wymagania, interfejs Task jest rozszerzany o odpowiednie metody w WorkerPoolTask. BaseWorkerPoolTask to klasa abstrakcyjna, która dostarcza wspólnej implementacji dla większości metod.

DefaultWorkerPoolTask to podstawowa implementacja tasku w systemie jNode. Pełni on rolę *Dekoratora* dla taska zdefiniowanego przez użytkownika. MainClassWorkerPoolTask różni się tym, że jego zadaniem jest uruchomienie metody main z podanego archiwum jar. To pierwszy task, który jest wykonywany dla danego archiwum jar.

[Alan Hawrot]

JarPath, czyli katalog na archiwa

JarPath jest fundamentalnym elementem systemu. Wszystkie jary, których kod chcemy wykonać muszą trafić do katalogu JarPath. Użytkownik może samodzielnie skopiować do katalogu swoje archiwa. Istnieje też opcja programu, której parametrem jest ścieżka do archiwum jar, natomiast jest ona obsługiwana poprzez przekopiowanie archiwum do katalogu JarPath.

System automatycznie wykrywa wszystkie archiwa, które zostały dodane do JarPath. Odpowiadającym elementem jest klasa JarPathWatcher. Klasa ta implementuje singleton oraz enkapsuluje wątek, który oczekuje na nadchodzące zdarzenia w katalogu. Wykrywanie zmian w katalogu jest zaimplementowane przy pomocy omawianego już *WatchService*. Dzięki zastosowaniu takiego rozwiązania, wątek JarPathWatcher nie marnuje czasu procesora aktywnie oczekując zmian w katalogu, a zasypia oczekując na nadejście zdarzenia. Wątek reaguje jedynie na pliki o rozszerzeniu jar lub properties i informację o ich utworzeniu, modyfikacji oraz usunięciu przekazuje do JarPathManager. Warto jest podkreślić, że klasa WatchService jak wszystkie publiczne API Javy jest międzyplatformowa, mimo to są różnice w semantyce pomiędzy platformami. Jak się okazało, w systemie Ubuntu kopiowanie pliku jest reprezentowane przez dwa osobne zdarzenia. Najpierw tworzony jest plik pusty i po jego utworzeniu generowane jest zdarzenie utworzenia pliku. Kolejne zdarzenie modyfikacji pliku jest ogłaszane po jego wypełnieniu. Podczas gdy na OS X generowane jest jedno zdarzenie utworzenia pliku, który już posiada swoją zawartość.

JarPathManager jest również singletonem, którego zadaniem jest podejmowanie decyzji związanych z zdarzeniami zgłoszonymi przez JarPathWatcher o zmianach w JarPath. W systemie jNode dla każdego archiwum tworzony jest odpowiadający mu plik properties, który zawiera informację o identyfikatorze węzła, z którego pochodzi aplikacja oraz tego, jaki jest jej

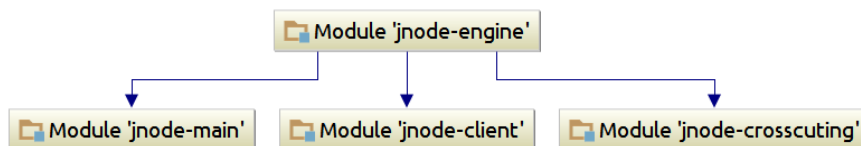
stan wykonywania (DURING_EXECUTING, CANCELLED, FINISHED). JarPathManager udostępnia stan tych plików i aktualizuje go obsługując odpowiednie zdarzenia zgłaszane przez JarPathWatcher, jak i zdarzenia zgłaszane przez mechanizm Spring Framework z innych części systemu. Gdy nastąpi utworzenie lub usunięcie archiwum, JarPathManager ogłasza odpowiednie zdarzenia. Na podstawie tych zdarzeń inne części systemu włączają lub wyłączają program użytkownika zwalniając wszystkie zasoby.

JarPath domyślnie jest to katalog o nazwie jarpath, który znajduje się w głównym katalogu programu. Jest to jednak konfigurowalne poprzez podanie ścieżki w systemie przy starcie programu za pomocą parametru, co będzie opisane w rozdziale *Moduł main*.

[Michał Semik]

Zależności modułu engine

Moduł engine wraz ze swoimi zależnościami tworzy okrojona, ale działającą wersję systemu. Jest to minimalny zestaw klas, który umożliwia wykonywanie zadań w systemie.



Rysunek 15. Diagram przedstawiający komponenty zależne od komponentu engine.

Moduł engine bazuje na najbardziej fundamentalnych elementach systemu, odpowiedzialnych za uruchamianie aplikacji, bibliotekę współdzieloną oraz narzędzia przekrojowe.

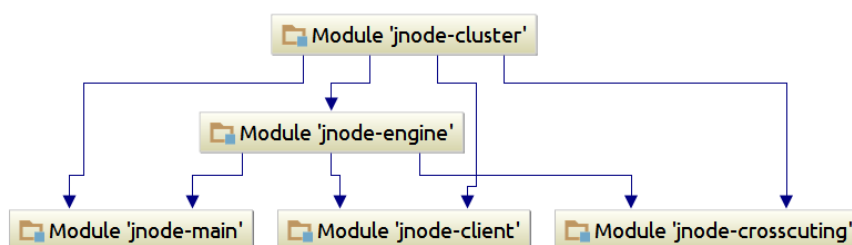
[Michał Semik]

Moduł cluster

Wstęp

Moduł cluster rozszerza możliwości systemu o możliwość delegowania tasków do innych węzłów (instancji programu). Węzły komunikują się ze sobą poprzez moduł cluster. Dzięki takiemu podejściu, wykonywany program może wykorzystywać większą liczbę komputerów w celu szybszego wykonania zadań. Inaczej mówiąc jNode zapewnia dobre *skalowanie horyzontalne*, omówione przy okazji omawiania podstawowych pojęć.

Moduł cluster korzysta z modułu engine w celu pozyskiwania tasków do wysłania na inne węzły, a także do wykonywania tasków otrzymywanych od innych węzłów. Dzięki zastosowaniu zdarzeń udostępnianych przez Spring Framework, moduł engine może informować moduł cluster o nadmiarze tasków do wykonania, a moduł engine nie wie o istnieniu modułu cluster i jest od niego w pełni niezależny.



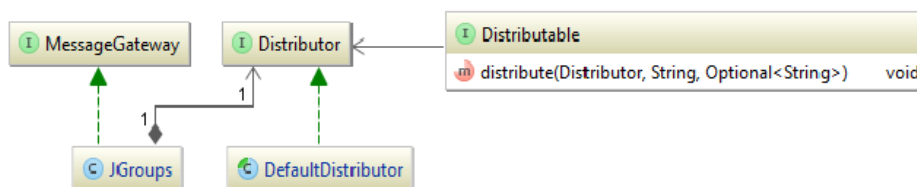
Rysunek 16. Zależności modułu cluster.

Gdy dwa węzły znajdują się w jednej sieci lokalnej, połączenie między nimi zostaje automatycznie ustanowione i mogą się komunikować. Żadna dodatkowa konfiguracja nie jest wymagana. Każdy węzeł, który zostanie włączony automatycznie dołącza do klastra oraz każdy z węzłów utrzymuje listę wszystkich węzłów w klastrze. W liście utrzymywane są identyfikatory węzłów. Identyfikator węzła może być ustanowiony przy jego uruchamianiu poprzez podanie odpowiedniego parametru. Jeśli nie został on podany, węzeł otrzyma automatycznie wygenerowany identyfikator poprzez połączenie nazwy zalogowanego użytkownika i wygenerowanych liczb. Węzły w celu wzajemnej komunikacji korzystają z techniki *message passing*. Każdy typ wiadomości w systemie jest reprezentowany poprzez klasę. Obiekty tych klas są serializowane przy pomocy mechanizmu serializacji udostępnianego przez API Javy.

[Michał Semik]

Główna abstrakcja, czyli Distributor oraz integracja z biblioteką JGroups

Wiadomość, która jest otrzymana przez system trafia do klasy JGroups. Klasa ta znajduje się w module cluster i jest odpowiedzialna za współpracę z biblioteką JGroups. Klasa JGroups enkapsuluje całą integrację z tą biblioteką. Odpowiada za stworzenie obiektów reprezentujących klaster, inicjalizację połączenia z klastrem, wysyłanie oraz odbieranie wiadomości. Natomiast nie obsługuje ani nie interpretuje wiadomości, które przez niego przechodzą. Tym zajmuje się implementacja interfejsu Distributor, który jest mediatorem komunikatów przychodzących. Każdy komunikat, który przychodzi do węzła trafia do Distributora, który zleca ich dalszą obsługę do klas pomocniczych. Distributor posiada szereg metod, m.in. po jednej do obsługi każdego typu komunikatu w systemie. W tym miejscu stanęliśmy przed następującym problemem: jak wybrać odpowiednią metodę w Distributorze, zakładając, że nie chcemy skorzystać z instrukcji switch wybierającej metodę na podstawie typu otrzymanego komunikatu. Wybrany przez nas rozwiązaniem było zastosowanie wzorca projektowego *Wizytator* (lub też zwany odwiedzający) [68]. Każda klasa reprezentująca komunikat implementuje metodę `distribute` interfejsu `Distributable`, która przyjmuje jako argument m.in. implementację interfejsu `Distributor`. Implementacja metody `distribute` wybiera odpowiednią metodę `Distributora` obsługującą ten typ komunikatu. Rozwiązanie to jest dobre, ponieważ klasa JGroups nie ma i nie powinna mieć informacji na temat konkretnych typów komunikatów oraz metod, które go obsługują. Inną zaletą jest fakt, że dodanie nowego komunikatu wymaga jedynie implementacji metody `distribute` oraz dodanie metody obsługującej ten komunikat do `Distributora`. Relacje pomiędzy tymi klasami są reprezentowane na poniższym diagramie.



Rysunek 17. Diagram reprezentujący relacje klas rozpoznających otrzymany komunikat.

Pozostałe parametry metody `distribute` to identyfikatory węzła źródłowego oraz docelowego, z tym że identyfikator węzła docelowego jest opcjonalny, ponieważ w systemie istnieje możliwość wydajnego wysyłania wiadomości do wszystkich węzłów, dzięki technice IP multicast.

Automatyczne ustanawianie połączenia między węzłami jest możliwe również dzięki zastosowaniu techniki IP multicast. Technologia JGroups domyślnie została skonfigurowana ze stosem UDP, który umożliwia skorzystanie z IP multicast. Dlatego też automatyczne wykrywanie węzłów jest ograniczone do sieci lokalnej.

jNode można również uruchomić w oparciu o stos TCP. JGroups umożliwia taką konfigurację. Aby to zrobić należy podczas uruchamiania programu ustalić nasłuchiwany adres oraz grupę adresów będącymi zaufanymi węzłami w klastrze. Nie trzeba podawać wszystkich węzłów należących do klastra.

Ostatnim elementem, który pojawił się na diagramie, a nie był omawiany jest MessageService. To interfejs, który jest implementowany przez JGroups i umożliwia wysyłanie wiadomości oraz sprawdzenie identyfikatora węzła. Pozostałe części systemu posługują się tym interfejsem i są całkowicie niezależne od klasy JGroups. Jedyny obiekt tej klasy jest inicjalizowany przez Spring Framework. Zdecydowaliśmy się zabezpieczyć w ten sposób, gdyby trzeba było zrezygnować z JGroups, zastępując go inną technologią lub napisać własną. Dzięki temu rozwiązaniu można zrobić to bez wprowadzania jakichkolwiek zmian w istniejącym systemie.

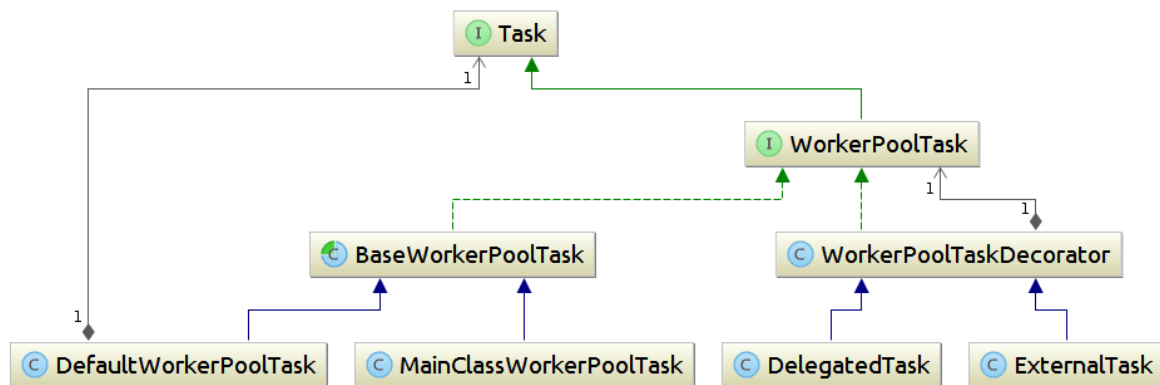
Po otrzymaniu wiadomości, biblioteka JGroups wywołuje naszą implementację metody receive z interfejsu Receiver, który jest dostarczany przez bibliotekę. Zalecane przez twórców jest, żeby obsługa komunikatu była możliwie krótka i najlepiej, żeby każdy z komunikatów był obsługiwany w osobnym wątku. Dodatkowo też nasz system wymaga, aby komunikaty były obsługiwane sekwencyjnie. Chcemy uniknąć sytuacji, w których kolejne komunikaty przychodzą do systemu, a poprzednie nie zostały jeszcze obsłużone. Przykładowo nie chcemy, aby węzeł otrzymywał komunikaty od węzłów, które jeszcze nie zostały zarejestrowane w tym węźle. Inaczej mówiąc chcemy uniknąć omawianego we wstępie *race condition*. W celu zaspokojenia powyższych wymagań utrzymywany jest jeden wątek, a dokładniej pula wątków o wielkości 1, która obsługuje wszystkie nadchodzące komunikaty.

Jak już było powiedziane we wstępie, każdy węzeł utrzymuje listę wszystkich węzłów w klastrze. Jest to zrealizowane analogicznie jak w przypadku otrzymywania wiadomości. Klasa JGroups implementuje interfejs MembershipListener dostarczany przez bibliotekę.

Zawiera ona metodę `viewAccepted`. Argument tej metody zawiera informację na temat obecnego zestawu węzłów klastra. Na podstawie tej informacji możemy sprawdzić, które węzły opuściły klastro lub do niego dołączyły. Informacja ta jak się później okaże jest kluczowa w celu poprawnego działania systemu. W przypadku, gdy któryś z węzłów ulegnie awarii, każdy węzeł zostanie o tym poinformowany poprzez komunikat `viewAccepted`. Informacja na temat węzłów, które dołączyły do klastra lub go opuściły również jest przekazywana do Distributora.

[Michał Semik]

Struktura tasków na poziomie modułu cluster



Rysunek 18. Diagram przedstawiający strukturę tasków na poziomie modułu cluster.

Część struktury tasków została omówiona w rozdziale *Struktura tasków na poziomie modułu engine*. W części rozproszonej systemu `jNode`, w której współpracuje ze sobą wiele instancji (węzłów) koniecznym było rozszerzenie tej struktury tasków. Zostało to zrealizowane przy pomocy wzorca projektowego o nazwie *Dekorator* [69].

Do istniejącej struktury zostały dodane trzy elementy realizujące ten wzorzec. Są nimi: `WorkerPoolTaskDecorator`, `DelegatedTask` oraz `ExternalTask`. `WorkerPoolTaskDecorator` to główna klasa, która jak sama nazwa wskazuje, pełni główną rolę *Dekoratora*. Zawiera ona implementację, która jest wspólna dla `DelegatedTask` oraz `ExternalTask`. `DelegatedTask` rozszerza `WorkerPoolTaskDecorator` o możliwość przechowywania informacji na temat tego, do jakiego węzła został oddelegowany dany task. Z kolei `ExternalTask` przechowuje informację na temat tego, z jakiego węzła pochodzi dany task. Obiekty klasy `ExternalTask` to taski, które są przesyłane pomiędzy węzłami. Dodatkowo klasa ta zawiera specjalną implementację

interfejsu `Serializable`. Wprowadzenie tej specjalnej obsługi serializacji było konieczne ze względu na fakt, iż po przesłaniu taska do innego węzła, w węźle tym może nie być jeszcze znana, zdefiniowana przez użytkownika klasa tasku. Do maszyny wirtualnej zostanie załadowana ta klasa dopiero po przesłaniu archiwum jar do tego węzła. Jest to szerzej omówione w rozdziale *Propagowanie archiwum jar pomiędzy węzłami*.

[Alan Hawrot]

Rejestry tasków, czyli nadzorowanie stanu systemu

Każda instancja `jNode` posiada dwa rejestry tasków. Są nimi: `ExternalTaskRegistry` oraz `DelegatedTaskRegistry`. Rejestry te zostały wprowadzone w celu nadzorowania stanu systemu i głównie są wykorzystywane przy obsłudze błędów. W rozdziale *Obsługa błędów* ich wykorzystanie jest dokładnie omówione. `DelegatedTaskRegistry` to rejestr wszystkich tasków oddelegowanych do innych węzłów, natomiast `ExternalTaskRegistry` jest rejestrem wszystkich tasków przyjętych do wykonania od innych węzłów. Każdy z tych rejestrów jest na bieżąco aktualizowany przy wysyłaniu i odbieraniu tasków, jak również w przypadku anulowania tasków lub wystąpienia awarii. Podsumowując każda instancja `jNode` cały czas prowadzi rejestr tasków oddelegowanych do innych instancji oraz rejestr tasków przyjętych do wykonania od innych instancji. Pozwala to każdemu z węzłów lepiej reagować na różnego rodzaju zdarzenia zachodzące w systemie.

[Alan Hawrot]

Mechanizm wyboru węzła

Gdy w `jNode` jest uruchomionych wiele współpracujących ze sobą węzłów i zakładając, że w którymś z nich dochodzi do przepełnienia `WorkerPool` taskami, to podejmowana jest próba wysłania tasków oczekujących na wykonanie do węzła, który w danej chwili posiada wątki niewykonujące żadnej pracy. Taki węzeł jest gotowy na przyjęcie tasków od innych węzłów. Istnieje jednak prawdopodobieństwo wystąpienia sytuacji, w której kilka węzłów będzie próbowało przesłać swoje taski oczekujące na wykonanie do tego samego węzła, co spowoduje szybkie przepełnienie `WorkerPool` w tym węźle i ponowne rozsyłanie tychże tasków. Celem było opracowanie wydajnej metody wyboru węzła, która zminimalizuje ryzyko

wystąpienia opisanej wyżej sytuacji. Dzięki temu ilość przesyłanych komunikatów pomiędzy węzłami jest znacznie niższa.

W jNode węzły w pewnych odstępach czasu wysyłają do siebie informację o liczbie wątków w puli (każdy węzeł może mieć inną wielkość puli) i liczbie zajętych wątków. Algorytm rozsyłania tych informacji jest oparty o algorytm HeartBeat i opisany w rozdziale *HeartBeat*. Dodatkowo każdy węzeł posiada swój unikatowy numer. Na podstawie tych danych opracowaliśmy mechanizm wyboru węzła.

Każdy węzeł w klastrze posiada swoją kolejkę priorytetową, która przechowuje informacje na temat działających węzłów w klastrze. Kolejka ta jest aktualizowana po każdym otrzymaniu informacji o stanie innego węzła. Gdy węzły dołączają do klastra lub się od niego odłączają albo, gdy zachodzi awaria, kolejka ta również jest aktualizowana. Kolejka priorytetowa sortuje węzły w taki sposób, aby na początku znajdował się identyfikator węzła, do którego najkorzystniej będzie wysłać task do wykonania. Podsumowując każdy z węzłów przechowywanych w kolejce posiada swój priorytet i im wyższy jest ten priorytet, tym bliżej początku kolejki zostanie umieszczony i korzystniej jest do niego wysłać task do wykonania.

Po wprowadzeniu kolejki priorytetowej sortującej węzły, kolejnym krokiem było opracowanie funkcji liczącej priorytety dla tych węzłów. Do systemu jNode wprowadziliśmy następującą funkcję:

wejście:

n – ilość węzłów w klastrze

m – wielkość puli wątków

z – liczba zajętych wątków

i – unikatowy numer węzła

id – unikatowy numer węzła, w którym aktualizowana jest kolejka

max_m – wielkość największej puli wątków

$dist = \min(|id - i|, n - |id - i|)$

– odległość węzłów w pierścieniu, dla przykładu $dist(n, 1) = 1$

wyjście:

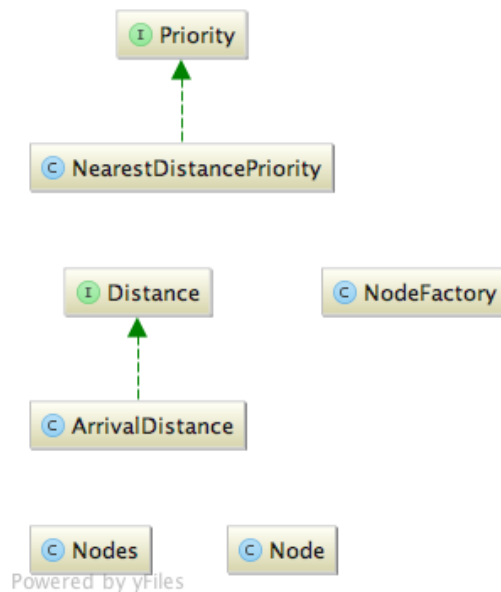
y = priority

$$y = \frac{m - z}{\max_m + 1} + \text{sgn}(m - z)(n - \text{dist} + 1)$$

- Funkcja ta charakteryzuje się tym, że priorytetyzuje bliskość węzła ponad wielkość puli. Wielkość puli ma znaczenie tylko wtedy, gdy odległość pomiędzy węzłami jest taka sama.
- Pierwszy czynnik sumy jest dzielony przez $\max_m + 1$, aby był zawsze mniejszy niż 1.
- Drugi czynnik jest liczbą mniejszą od 0, gdy pula ma wolne wątki. W pozostałych przypadkach wynosi co najmniej 1. Im odległość węzłów jest mniejsza, tym ten czynnik jest większy. W ten sposób uzyskuje on wyższy priorytet, pod warunkiem, że pula nie jest pusta.

Jest to domyślna funkcja licząca priorytety w jNode. Funkcja ta w połączeniu z omówioną kolejką priorytetową rozwiązuje przedstawiony na początku problem. Ilość sytuacji, w których trzeba ponownie rozsyłać taski w wyniku wyboru tego samego węzła przez inne węzły z przepełnionymi WorkerPool i tym samym doprowadzenie do szybkiego przepełnienia puli wątków wybranego węzła jest niższa. W wyniku czego liczba przesyłanych komunikatów w systemie jest zredukowana.

Architektura systemu jest na tyle elastyczna, że gdy zaistnieje potrzeba wprowadzenia nowej funkcji i zamiany domyślnej, to będzie można w łatwy i szybki sposób tego dokonać. Architektura podsystemu związanego z opisanym mechanizmem jest przedstawiona na poniższym diagramie.



Rysunek 19. Diagram przedstawiający wybrane klasy modułu.

Wspomnianą elastyczność zapewniają dwa główne interfejsy: Priority i Distance. Obecne w systemie jNode implementacje tych interfejsów, czyli NearestDistancePriority oraz ArrivalDistance realizują wyżej opisaną funkcję liczącą priorytety. Interfejs Priority posiada jedną zadeklarowaną metodę calculate. Jej celem jest obliczenie priorytetów dla wszystkich węzłów w kolejce. Interfejs Distance również ma jedną zadeklarowaną metodę. Jej zadaniem z kolei jest obliczenie odległości pomiędzy dwoma dowolnymi węzłami w klastrze. W przypadku klasy ArrivalDistance i implementacji tej metody odległość pomiędzy węzłami jest rozumiana jako odległość węzłów w pierścieniu. Każdy węzeł, który dołącza do klastra otrzymuje kolejną liczbę naturalną. Węzły przy pomocy tych liczb ustawione są w pierścień liczbowy i odległość liczona jest w sposób jaki został opisany przy okazji omawiania funkcji. Jak widać, gdyby zaistniała potrzeba wprowadzenia nowej funkcji, wystarczy dodać kolejne implementacje interfejsów Priority i Distance.

Omówiona kolejka priorytetowa jest reprezentowana przez klasę Nodes. Posiada ona wszystkie podstawowe metody związane z aktualizacją zawartości tej kolejki, metody utrzymywania właściwego porządku przy obsłudze różnych zdarzeń zachodzących w systemie (przykładowo otrzymanie komunikatu HeartBeat lub odłączenie się węzła od klastra) oraz metodę pobierania węzła z najwyższym priorytetem.

Elementy przechowywane w kolejce to obiekty klasy Node. Klasa ta przechowuje wszystkie wspomniane wcześniej informacje konieczne dla mechanizmu wyboru węzła, czyli: identyfikator węzła, wielkość puli wątków, liczbę dostępnych wątków (niewykonujących pracy), priorytet węzła i referencję do obiektu klasy implementującej interfejs Distance.

NodeFactory to *fabryka* produkująca obiekty wyżej omówionych klas.

[Alan Hawrot]

Komunikacja między węzłami, czyli nurkując w Distributora

HeartBeat

HeartBeat jest komunikatem oznajmiającym o stanie węzła. Każdy węzeł informuje inne o swoim stanie. W komunikacie tym wysyłana jest ilość wątków w WorkerPool oraz ilość znajdujących się tam zadań, zarówno wykonujących się, jak i oczekujących. Celem tego komunikatu jest poinformowanie innych węzłów o ilości wolnych zasobów obliczeniowych. Na podstawie tych informacji wybierane są węzły, do których najlepiej wysyłać nadmierne taski. Algorytm, który korzysta z tych informacji był omawiany w rozdziale *Mechanizm wyboru węzła*. Komunikat ten jest wysyłany do wszystkich węzłów jednocześnie, wykorzystując IP multicast. Jest on propagowany okresowo. Okres jest ustawiony na stałe, użyliśmy 50 milisekund.

[Michał Semik]

Dostarczanie zadań pomiędzy węzłami i rozsyłanie wyników

Zanim zostanie omówiona część systemu jNode (w tym architektura i algorytmy) odpowiedzialna za dostarczanie zadań asynchronicznych tj. przesyłanie tasków pomiędzy węzłami i rozsyłanie rezultatów wykonania tychże zadań, należy opisać wymagania jakie musi spełniać podsystem za to odpowiedzialny.

Przy okazji omawiania struktury tasków w poprzednich rozdziałach zostało wspomniane, że każdy z rejestrowanych w systemie zadań asynchronicznych musi posiadać unikalny identyfikator oraz platforma jNode po wykonaniu danego zadania musi umieścić rezultat jego wykonania w odpowiednim EventLoopThread, gdzie zostanie uruchomione

powiązane z danym zadaniem wywołanie zwrotne. Wywołanie zwrotne przyjmuje jako argument rezultat wykonania zadania asynchronicznego. O ile w przypadku działania pojedynczej instancji jNode nie było to trudne do osiągnięcia tak w sytuacji, gdy uruchomionych jest wiele instancji platformy jNode jako współpracujące ze sobą węzły w klastrze, wymagania te nabierają dodatkowego znaczenia i wagi.

W rozdziale *Struktura tasków na poziomie cluster* została opisana klasa ExternalTask. Obiekty tej klasy to zadania asynchroniczne, które są przesyłane pomiędzy węzłami w celu wykonania. Jest to podstawowa klasa, która reprezentuje taski na poziomie cluster. Dla przypomnienia, pełni ona rolę *Dekoratora* dla klasy WorkerPoolTask i dodaje informację z jakiego węzła pochodzi dany task oraz zapewnia specjalną obsługę serializacji. Ze względu na to, że zadania asynchroniczne są przesyłane pomiędzy węzłami w klastrze, muszą one posiadać identyfikator, który jest unikalny w obrębie nie pojedynczej instancji, ale całego klastra. Jest to wymagane, aby uniknąć kolizji w rejestrach tasków, która mogłaby doprowadzić do szeregu niepożądanych sytuacji. W przypadku kolizji identyfikatorów, przykładowo zadanie mogłoby nigdy nie zostać wykonane lub rezultat wykonania zadania mógłby zostać przekazany nie do tego wywołania zwrotnego, do którego powinno. Takich sytuacji jest więcej. Aby zapobiec temu problemowi i stworzyć mechanizm, dzięki któremu identyfikator każdego rejestrowanego zadania asynchronicznego będzie unikalny w obrębie całego klastra, skorzystaliśmy z następującego faktu: identyfikator węzła, który jest unikalny w obrębie klastra w połączeniu z generowanym na pojedynczej instancji i unikalnym w jej obrębie identyfikatorem zadania tworzą razem unikalny identyfikator taska w obrębie całego klastra. By zapewnić unikalność identyfikatora w obrębie pojedynczej instancji skorzystaliśmy z implementacji UUID, dostępnej w Javie [70]. Już sam ten mechanizm i dostępna implementacja mogłaby w zupełności wystarczyć do wygenerowania unikalnego identyfikatora w obrębie całego klastra, gdyż zapewnia niezwykle niskie prawdopodobieństwo kolizji [71], wystąpienia duplikatów. Dokładając jednak identyfikator węzła i łącząc te dwa, prawdopodobieństwo jest jeszcze niższe.

Dane zadanie asynchroniczne może zostać wykonane w dowolnym z działających w klastrze węzłów, natomiast powiązane wywołanie zwrotne działające na argumencie będącym rezultatem wykonania zadania musi zostać wykonane w węźle, w którym pierwotnie zostało umieszczone archiwum jar z programem klienckim. Węzeł ten, zwany źródłowym lub macierzystym posiada uruchomiony EventLoopThread, który jest powiązany z archiwum jar i

proceedzi rejestr zadań asynchronicznych związanych z wykonywanym programem klienckim. Zatem jeżeli dane zadanie zostanie wykonane na węźle, który nie jest węzłem źródłowym, węzeł ten musi przesłać rezultat wykonania zadania asynchronicznego do węzła źródłowego. Jak widać pojawia się tutaj dodatkowe wymaganie. Rezultat wykonania zadania musi być obiektem klasy serializowalnej, by mógł być przesyłany pomiędzy węzłami. W module client zostało to wyszczególnione w udostępnionym użytkownikowi interfejsie.

W systemie jNode dane, zdefiniowane przez użytkownika zadanie asynchroniczne może tworzyć kolejne podzadania, dzieląc w ten sposób pracę jaką musi wykonać na wiele części, które również mają wykonać się asynchronicznie. Przy tworzeniu jNode musieliśmy uwzględnić takie sytuacje. Gdy zadanie zostało przesłane w celu wykonania do innego węzła i tam jest wykonywane, węzeł ten musi wykryć wszystkie ewentualne podzadania jakie utworzyło pierwotne zadanie i powiadomić o tym węzeł źródłowy. Dla przypomnienia rejestracja zadania składa się z samego zadania i powiązanego wywołania zwrotnego, tak więc obiekty klasy implementującej interfejs Callback muszą być serializowalne, by mogły być przesyłane pomiędzy węzłami i to również zostało wyszczególnione w interfejsie udostępnionym użytkownikowi, który znajduje się w module client. Ewentualna konieczność poinformowania węzła źródłowego o wymaganej rejestracji podzadań jest wykrywana poprzez sprawdzenie lokalizacji powiązanego archiwum jar. O lokalizacji i przesyłaniu archiwów jar będzie mówione w rozdziale *Propagowanie archiwum jar pomiędzy węzłami*.

Główną jednostką odpowiedzialną za przesyłanie zadań asynchronicznych i rozsyłanie rezultatów jest Distributor. To obiekt, który jest singletonem w obrębie jednej instancji i obsługuje komunikację z innymi instancjami platformy jNode. Distributor posiada szereg metod do przetwarzania komunikatów przychodzących od innych węzłów jak również zdarzeń zachodzących w danej instancji. Obiekt ten nie obsługuje samodzielnie wszystkich zdarzeń i komunikatów. Pełni on rolę *Mediatora* delegując obsługę do odpowiednich podjednostek jNode. Distributor stanowi jednak punkt centralny części rozproszonej systemu, który decyduje o podjęciu odpowiednich działań.

| Distributor | |
|---|------|
| on(WorkerPoolOverflowEvent) | void |
| onTaskDelegation(ExternalTask) | void |
| onRedirect(String, String, String) | void |
| onSry(String, String) | void |
| onTaskExecutionCompleted(String, SerializableTaskResultWrapper) | void |
| on(TaskFinishedEvent) | void |
| on(CancelJarJobsEvent) | void |
| on(TaskCancelledEvent) | void |
| on(ExternalSubTaskReceivedEvent) | void |
| onRegisterDelegatedSubTask(String, ExternalTask, SerializableCallbackWrapper) | void |
| onNodeGone(String) | void |
| onNewNode(String) | void |
| onCancelJarJobs(String, String) | void |
| onPrimaryHeartBeat(String, PrimaryHeartBeat) | void |
| onJarRequest(String, String) | void |
| onJarDelivery(String, String, byte[]) | void |

Powered by yFiles

Rysunek 20. Metody interfejsu Distributor.

[Alan Hawrot]

Dostarczanie zadań pomiędzy węzłami

Procedura dostarczania zadań asynchronicznych do innych węzłów w celu wykonania następuje wtedy, gdy w danym węźle w jednej chwili liczba zarejestrowanych zadań będzie większa niż liczba dostępnych wątków w WorkerPool. Zadania te zostają umieszczone w kolejce oczekujących na wykonanie, a pula wątków ogłasza przy użyciu systemu zdarzeń udostępnionego przez Spring Framework zdarzenie o nazwie WorkerPoolOverflowEvent. Ze względu na jego skomplikowaną obsługę w systemie jNode została utworzona specjalna jednostka o nazwie DelegationHandler. Zostanie ona szczegółowo omówiona w rozdziale *DelegationHandler*. Distributor reaguje na wystąpienie tego zdarzenia i deleguje zadanie jego obsługi do obiektu klasy implementującej interfejs DelegationHandler.

Po zakończeniu obsługi zdarzenia zostaje wysłany do wybranego węzła komunikat o nazwie TaskDelegation. Jak zostało wspomniane w rozdziale *Główna abstrakcja, czyli Distributor oraz integracja z biblioteką JGroups*, komunikat ten zostaje odebrany w danym węźle w klasie JGroups i przekazany do Distributora. Distributor po otrzymaniu tego komunikatu dodaje odpowiedni wpis w rejestrze ExternalTaskRegistry i deleguje dalszą

obsługę do jednostki zwanej JarHandler. Zostanie ona omówiona w rozdziale *Propagowanie archiwum jar pomiędzy węzłami*.

W rozdziałach *Modyfikacja implementacji puli wątków dostarczonej przez Spring Framework* oraz *Mechanizm wyboru węzła* została opisana niekorzystna sytuacja, w której kilka węzłów może oddelegować wykonanie zadań asynchronicznych do tego samego węzła. Pomimo tego, iż węzeł ten jest gotowy na przyjęcie tasków, jego pula wątków może zostać szybko wypełniona przesłanymi zadaniami i nastąpi konieczność dalszego rozsyłania tych tasków, które nie mogą być w danej chwili wykonane. Modyfikacja implementacji puli wątków dostarczonej przez Spring Framework i wprowadzenie priorytetów dla zadań zapobiega *zagłodzeniu* i zapewnia, że każdy przesłany task zostanie w końcu wykonany. Opracowanie specjalnego mechanizmu wyboru węzła obniża prawdopodobieństwo wystąpienia opisywanej sytuacji. Ryzyko jednak nadal istnieje i sytuacja ta może mieć miejsce. Przykładowo uruchomione są trzy instancje platformy jNode. Jedna z nich posiada wątek niewykonujący pracy, zatem jest gotowa na przyjęcie zadania. Pozostałe dwie mają po jednym oczekującym na wykonanie zadaniu, więc będą próbowały go przesłać do innej instancji w klastrze. Ze względu na to, iż tylko jeden węzeł jest gotowy na przyjęcie zadania, zostaje on wybrany przez obie instancje i jednocześnie dwa zadania są do niego przesłane w celu wykonania. Niestety węzeł ten może wykonać w danej chwili tylko jedno z nich, zatem drugie musi być ponownie rozesłane.

Prawdopodobieństwo wystąpienia opisywanej, niekorzystnej sytuacji i ryzyko awarii węzła w dowolnym momencie wprowadziło konieczność opracowania procedury ponownego przenoszenia zadania na inny węzeł. Zostało stworzonych kilka koncepcji, które zostaną poniżej szczegółowo omówione.

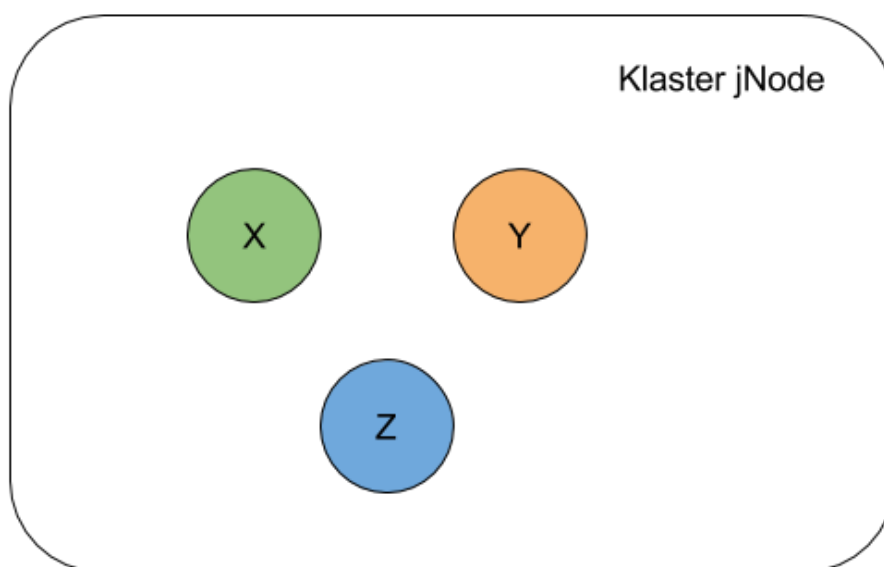
[Alan Hawrot]

Procedura ponownego przenoszenia zadania na inny węzeł

Procedura ma zastosowanie, gdy WorkerPool ogłasza zdarzenie WorkerPoolOverflowEvent, a z kolejki zadań oczekujących na wykonanie zostało wybrane zadanie, które pierwotnie pochodzi z innego węzła. Jest to obiekt klasy ExternalTask, który w tym przypadku będzie ponownie przesłany na inny węzeł.

Opis scenariusza:

1. W klastrze znajdują się trzy węzły: X, Y i Z.
2. Węzeł X przesyła zadanie do węzła Y.
3. Zanim zadanie zostało dostarczone w celu wykonania, pula wątków w węźle Y została zajęta przez zadania pochodzące z innego źródła.
4. Węzeł Z jest gotowy na przyjęcie zadania.
5. Węzeł Y przesyła zadanie otrzymane od węzła X.



Rysunek 21. Przykładowa konfiguracja.

[Alan Hawrot]

Podjęcie pierwsze

Węzeł Y zanim prześle zadanie do innego węzła wysyła komunikat *redirected* do węzła źródłowego (X) z informacją o przekazaniu zadania do wybranego węzła (Z). Następnie dostarcza zadanie w celu wykonania do wybranego węzła (Z).

Problem:

1. Węzeł Y wysyła komunikat *redirected* do węzła X z informacją o przekazaniu zadania do węzła Z.

2. Następuje awaria węzła Y.

Węzeł X po otrzymaniu komunikatu *redirected* zaktualizował rejestr DelegatedTaskRegistry i czeka, aż węzeł Z wykona zadanie. Węzeł Z natomiast nie otrzymał zadania do wykonania, gdyż węzeł Y uległ awarii zanim zdążył przesłać zadanie do węzła Z. W takim przypadku zadanie może nigdy nie zostać wykonane.

[Alan Hawrot]

Podejście drugie

Węzeł Y w pierwszej kolejności przesyła zadanie do wybranego węzła (Z), a następnie wysyła komunikat *redirected* do węzła źródłowego (X).

Problem:

1. Węzeł Y przesyła zadanie do węzła Z.
2. Następuje awaria węzła Y.

W tej sytuacji węzeł Y po dostarczeniu zadania do węzła Z nie zdążył wysłać do węzła X komunikatu *redirected* z informacją o przekazaniu zadania. Zatem węzeł X nie zaktualizuje rejestru DelegatedTaskRegistry i po otrzymaniu informacji o awarii węzła Y zacznie ponowną obsługę zadania. Prowadzi to do tego, że to samo zadanie może zostać wykonane w klastrze dwukrotnie: w węźle Z i przykładowo w węźle X, jeżeli w danej chwili posiadał wątek niewykonujący pracy. W pesymistycznym przypadku, gdyby omówiony scenariusz się powtarzał, a w klastrze było uruchomionych wiele instancji, to samo zadanie mogłoby wykonać się wielokrotnie. Pomimo tego, że nie doprowadza to do błędów w systemie, jest to sytuacja niepożądana. Pierwszy rezultat wykonania zostanie wykorzystany, a wszystkie kolejne zignorowane. Wielokrotne wykonanie tego samego zadania w klastrze jest jednak niepotrzebnym zużyciem zasobów.

[Alan Hawrot]

Podejście trzecie

Zadanie zostaje przesłane z węzła Y do wybranego węzła (Z). Wybrany węzeł (Z) po otrzymaniu zadania ma obowiązek poinformować węzeł źródłowy o przekazaniu zadania w celu wykonania i wysyła do niego komunikat *redirected*.

Problem:

1. Węzeł Y przesyła zadanie do węzła Z.
2. Następuje awaria węzła Y.
3. Węzeł Z wysyła komunikat *redirected* do węzła X.

Jest to sytuacja bardzo podobna do opisanej w podejściu drugim. Zanim węzeł X otrzyma komunikat *redirected* od węzła Z i zaktualizuje rejestr, może zacząć ponowną obsługę i rozsyłanie zadania ze względu na komunikat o awarii węzła Y.

[Alan Hawrot]

Podejście czwarte

Jednym z rozwiązań dla wyżej omówionych problemów może być zastosowanie podejścia drugiego razem z trzecim poprzez wprowadzenie dodatkowych komunikatów. Poza komunikatem *redirected* byłyby to: *redirected-ack* i *redirection-finished*. Koncepcja opiera się na tym, iż węzeł Y jest odpowiedzialny za zadanie dopóki procedura nie przebiegnie pomyślnie. Jeśli nastąpi awaria węzła Y, to węzeł źródłowy (X) przejmuje odpowiedzialność za zadanie. Gdyby węzeł źródłowy (X) uległ awarii, to zadanie jest anulowane w całym klastrze, więc nie jest to w algorytmie rozważane.

Przy udanej wymianie scenariusz jest następujący:

1. Węzeł Y przesyła zadanie do węzła Z.
2. Węzeł Y wysyła komunikat *redirected* do węzła X.
3. Węzeł Z wysyła komunikat *redirected-ack* do węzła X.
4. Węzeł X wysyła komunikat *redirection-finished* do węzła Y oraz do węzła Z.
5. Węzeł Z rozpoczyna wykonywanie zadania.

Może się zdarzyć, że węzeł X otrzyma komunikat *redirected-ack* przed *redirected*, ale zakładamy, że również jest to udana wymiana.

Opis algorytmu:

Węzeł Y rozpoczyna procedurę:

1. *Węzeł Y ustawia zadanie w tryb DURING_REDIRECTION.*
2. *Węzeł Y wysyła zadanie do węzła Z.*
3. *Węzeł Y wysyła komunikat redirected do węzła X.*

Węzeł Y po otrzymaniu informacji o awarii węzła Z:

1. *Węzeł Y wszystkim zadaniom znajdującym się w stanie DURING_REDIRECTION i wysłanym do węzła Z zmienia stan na NORMAL i ponownie wstawia je do puli w celu wykonania.*

Węzeł Y po otrzymaniu komunikatu redirection-finished:

1. *Węzeł Y aktualizuje rejestr ExternalTaskRegistry poprzez usunięcie odpowiedniego wpisu, nie jest już odpowiedzialny za dane zadanie.*

Węzeł X po otrzymaniu komunikatu redirected od węzła Y lub redirection-ack od węzła Z:

1. *Węzeł X zapisuje identyfikator węzła Z.*
2. *Jeśli zadanie znajduje się w stanie DURING_REDIRECTION to:*
 1. *Węzeł X wysyła komunikat redirection-finished do węzła Y.*
 2. *Węzeł X wysyła komunikat redirection-finished do węzła Z.*
 3. *Węzeł X ustawia stan zadania na NORMAL.*
 4. *Węzeł X aktualizuje rejestr DelegatedTaskRegistry zmieniając identyfikator węzła Y na identyfikator węzła Z dla danego zadania.*
3. *W przeciwnym wypadku węzeł X zmienia stan zadania na DURING_REDIRECTION.*

Węzeł X po otrzymaniu informacji o awarii węzła Y:

1. *Węzeł X wstawia zadanie do puli w celu wykonania.*

Węzeł X po otrzymaniu informacji o awarii węzła Z:

1. *Węzeł X zmienia stan zadania na NORMAL (węzeł Y jest nadal odpowiedzialny za dane zadanie).*

Węzeł Z po otrzymaniu zadania od węzła Y:

1. *Węzeł Z zmienia stan zadania na DURING_REDIRECTION.*
2. *Węzeł Z wysyła komunikat redirection-ack do węzła X.*

Węzeł Z po otrzymaniu informacji o awarii węzła Y:

1. *Węzeł Z anuluje wykonanie zadania.*

Węzeł Z po otrzymaniu komunikatu redirection-finished:

1. *Węzeł Z zmienia stan zadania na NORMAL.*
2. *Węzeł Z wstawia zadanie do puli w celu wykonania.*

Wadą przedstawionego algorytmu jest duża liczba komunikatów potrzebnych do prawidłowego przesłania jednego zadania. Przy większej ilości przesyłanych zadań i wielu działających instancjach platformy jNode w klastrze, pięć komunikatów potrzebnych do przesłania jednego zadania może powodować zbyt duży narzut.

[Alan Hawrot]

Podejście piąte

Całkowicie odmienną koncepcją rozwiązania problemu jest nieprzekazywanie odpowiedzialności za zadanie na inny węzeł. Oznacza to, że węzeł źródłowy (X) może oddelegować zadanie do innego węzła (Y), ale w przypadku, gdy w danym węźle (Y) pula zostanie zajęta przez pochodzące z innych źródeł zadania, to węzeł źródłowy (X) zostaje o tym poinformowany i sam podejmuje decyzję odnośnie dalszej obsługi tego zadania. Węzeł źródłowy w takiej sytuacji może samodzielnie wykonać zadanie, jeśli w tym czasie zwolniło się miejsce w puli i posiada wątek niewykonujący pracy lub ponowić procedurę delegacji wykonania i spróbować przesłać zadanie do innego węzła (Z) w przypadku przepełnienia puli wątków. Takie podejście znacznie upraszcza sytuację i redukuje liczbę przesyłanych komunikatów.

1. *Węzeł Y wysyła komunikat sry do węzła X.*
2. *Węzeł X po otrzymaniu komunikatu sry wstawia ponownie zadanie do puli w celu wykonania.*

Podejście szóste

Jest to rozwiązanie, które zostało wprowadzone w jNode. Polega ono na wykorzystaniu idei przekazywania zadania do kolejnych węzłów, ale jednocześnie nie nakładając na nie dodatkowej odpowiedzialności za przekazywane zadanie. Węzeł Y po otrzymaniu zadania od węzła źródłowego (X), kiedy nie jest w stanie go wykonać ze względu na to, iż jego pula została wcześniej zajęta przez inne zadania, może wybrać (korzystając ze swojej kolejki węzłów - rozdział *Mechanizm wyboru węzła*) kolejny węzeł (Z), który jest gotowy na przyjęcie zadań. Węzeł Y nie przesyła jednak zadania do tego wybranego węzła (Z), lecz informuje o zaistniałej sytuacji bezpośrednio węzeł źródłowy (X) i przedstawia swojego kandydata (Z) gotowego do wykonywania zadań. Jeżeli węzeł Y nie znajdzie takiego kandydata lub okaże się, że najkorzystniej jest odesłać zadanie do węzła źródłowego (X), ponieważ ten w międzyczasie wykonał zadania i posiada wątek niewykonujący pracy, to węzeł Y zachowuje się tak samo, jak w przypadku omówionego, poprzedniego rozwiązania.

Węzeł Y:

1. *Węzeł Y wybiera węzeł, do którego najkorzystniej jest przesłać zadanie (Mechanizm wyboru węzła).*
2. *Jeżeli wybrany węzeł jest węzłem źródłowym (X), to węzeł Y wysyła komunikat sry do węzła źródłowego (X).*
3. *W przeciwnym wypadku węzeł Y wysyła do węzła źródłowego (X) komunikat redirected z informacją o wybranym węźle-kandydacie (przykładowo Z).*

Węzeł X po otrzymaniu komunikatu redirected:

1. *Węzeł X sprawdza, czy może wykonać zadanie.*
 1. *Jeżeli tak, to węzeł X usuwa wpis w DelegatedTaskRegistry i wstawia zadanie do puli.*
 2. *W przeciwnym wypadku węzeł X aktualizuje rejestr DelegatedTaskRegistry zmieniając identyfikator węzła Y na identyfikator węzła-kandydata (Z) dla danego zadania i wysyła zadanie do węzła-kandydata (Z).*

Węzeł X po otrzymaniu komunikatu sry:

1. Węzeł X wstawia zadanie do puli.

Wprowadzenie omówionego rozwiązania, które w połączeniu z odpowiednią obsługą błędów opisaną w rozdziale *Obsługa błędów*, zapewnia niezawodność systemu jNode. Wykorzystuje ono zalety poprzednich rozwiązań, eliminując jednocześnie ich wady. Dane zadanie po przesłaniu do innego węzła może być ponownie rozsyłane w klastrze w przypadku zajęcia puli wątków przez inne zadania w tym węźle, lecz odbywa się to przy wykorzystaniu niewielkiej liczby komunikatów. Nie doprowadza ono również do niekorzystnych i trudnych w obsłudze sytuacji, gdy któryś z węzłów biorących udział w procedurze ulega awarii. Dodatkowo wprowadzenie specjalnego komunikatu *redirected* (z informacją o węźle-kandydacie) przyspiesza wykonanie zadania.

[Alan Hawrot]

Rozsyłanie wyników

Na początku rozdziału *Dostarczanie zadań pomiędzy węzłami i rozsyłanie wyników* zostało powiedziane, że węzeł wykonujący zadanie pochodzące z innej instancji platformy musi przesłać rezultat wykonania do węzła źródłowego. Dla przypomnienia, dane zadanie asynchroniczne może zostać wykonane w dowolnym z działających w klastrze węzłów, natomiast powiązane wywołanie zwrotne musi zostać wykonane w węźle źródłowym w odpowiednim EventLoopThread. Zatem węzeł, który wykonał zadanie pochodzące z innej instancji musi przesłać rezultat wykonania do węzła źródłowego. Cała procedura zostanie szczegółowo omówiona.

WorkerPool po wykonaniu zadania przy pomocy systemu zdarzeń udostępnionego przez Spring Framework ogłasza zdarzenie o nazwie TaskFinishedEvent, w którym umieszcza dane zadanie i rezultat jego wykonania. Na zdarzenie to reagują dwie jednostki w jNode: Distributor i TaskCoordinator. Obiekty te sprawdzają, czy zadanie pochodzi z innego węzła. Jeżeli jest to zadanie pochodzące z innego węzła, to dalszą obsługą zajmie się Distributor. W takim przypadku aktualizuje rejestr ExternalTaskRegistry usuwając odpowiedni wpis i następnie wysyła do węzła źródłowego komunikat o nazwie TaskExecutionCompleted. Komunikat ten zawiera rezultat wykonania.

W węźle źródłowym po otrzymaniu komunikatu `TaskExecutionCompleted` jego obsługa zostaje przekazana do Distributora. Usuwa on wpis w rejestrze `DelegatedTaskRegistry`, po czym również ogłasza zdarzenie `TaskFinishedEvent`. Tym razem obsługą tego zdarzenia zajmie się `TaskCoordinator`, gdyż zadanie i jego rezultat znajdują się w węźle źródłowym. `TaskCoordinator` umieszcza rezultat wykonania w odpowiednim `EventLoopThread`, gdzie zostanie wykonane powiązane z zadaniem wywołanie zwrotne.

[Alan Hawrot]

Wykrywanie i rejestracja podzadań powstałych na innym węźle

We wstępie do rozdziału *Dostarczanie zadań pomiędzy węzłami i rozsyłanie wyników* zostało wspomniane, iż dane zadanie asynchroniczne może tworzyć kolejne podzadania, dzieląc w ten sposób pracę na kolejne części, które również mają wykonać się asynchronicznie. Należało więc wprowadzić odpowiednią obsługę takich sytuacji. O ile w przypadku pojedynczego węzła nie stanowi to żadnego problemu, tak po przesłaniu zadania na inny węzeł, jeżeli tworzy on kolejne podzadania, należy powiadomić o tym węzeł macierzysty. Węzeł macierzysty musi zarejestrować te zadania, aby później mógł prawidłowo wykonać odpowiednie wywołania zwrotne na otrzymanych rezultatach wykonanych podzadań. W innym wypadku wyniki wykonania podzadań nie byłyby w ogóle obsługane, gdyż węzeł macierzysty nie wiedziałby o istnieniu podzadań powstałych na innym węźle.

W celu wykrycia podzadań powstałych podczas wykonywania zadania asynchronicznego pochodzącego z innego węzła wykorzystywana jest ścieżka do archiwum jar. Każde archiwum jar pochodzące z innego węzła jest umieszczone w katalogu o nazwie będącej identyfikatorem tego węzła. Jeżeli więc na podstawie ścieżki do archiwum jar zostanie wykryte, że dane zadanie zostało utworzone przez inne, pochodzące z innego węzła - zostaje ogłoszone zdarzenie o nazwie `ExternalSubTaskReceivedEvent`. Reaguje na nie Distributor, który wysyła do węzła macierzystego komunikat o nazwie `RegisterDelegatedSubTask`, a następnie poprzez zdarzenie `TaskReceivedEvent` przekazuje dalszą obsługę zadania do `TaskCoordinator`, która od tego momentu nie różni się niczym od obsługi zadań pochodzących z innego węzła.

W węźle macierzystym po otrzymaniu komunikatu `RegisterDelegatedSubTask` i przekazaniu obsługi do Distributora zadanie zostaje zarejestrowane w `DelegatedTaskRegistry`, jak również w odpowiednim `EventLoopThread`. Tak więc każde powstałe na innym węźle podzadanie i rezultat jego wykonania zostaną prawidłowo obsłużone.

[Alan Hawrot]

DelegationHandler

Jak już było wspomniane w rozdziale *Dostarczanie zadań pomiędzy węzłami*, tytułowy mechanizm służy przede wszystkim do realizowania delegowania zadań do innych węzłów. Reaguje on zatem na zdarzenie `WorkerPoolOverflow`, generowane z modułu `engine`.

Celem jest obsłużenie zdarzenia. Natomiast, aby to zrobić poprawnie, należało rozważyć szereg problemów.

- Ilość zasobów obliczeniowych w klastrze, podlega nieustannej aktualizacji dzięki mechanizmowi `HeartBeat`, gdy istnieją oczekujące zadania do wykonania i pojawią się nowe wolne zasoby w innych węzłach, należy je wykorzystać i zadania delegować.
- Delegowanie na inny węzeł zadania wymaga synchronicznej, wraz z obsługą komunikatu `HeartBeat`, aktualizacji stanu klastra. Jest tak, ponieważ po przesłaniu zadania na inny węzeł chcemy od razu aktualizować stan jego puli, aby zapewnić poprawny wybór węzłów poprzez mechanizm wyboru węzła przy delegowaniu kolejnych zadań. Unikamy w ten sposób sytuacji, że kolejne zadania trafią do tego samego, pełnego już zadań węzła.
- Podczas obsługi zdarzenia, zadanie mogło rozpocząć się już wykonywać, a nawet zakończyć. Nie chcemy dodatkowo synchronizować wątków puli i mechanizmu `DelegationHandler` przy dostępie do nowych zadań, aby zapobiec takiej sytuacji. Lepiej jest reagować na zdarzenie `WorkerPoolOverflow` niezwiązane z konkretnym zadaniem i obsłużyć je bez dodatkowej synchronizacji.

- Podczas obsługi jednego zdarzenia, mogą występować kolejne, co jest bardzo prawdopodobne. Zadania w puli są pobierane przez wątki puli bez synchronizacji. Powoduje to, że nie wiemy ile tak naprawdę zadań będzie gotowych do wysłania w momencie działania mechanizmu delegacji. Zdecydowaliśmy, że podczas obsługi zdarzenia mechanizm będzie wysyłać wszystkie znajdujące się w puli oczekujące zadania, natomiast nadmierne zdarzenia `WorkerPoolOverflow` będą ignorowane.

W stworzonym rozwiązaniu uwzględniającym powyższe rozważania wykorzystany został wzorzec projektowy *Stan* [72]. Stan maszyny `DelegationHandler` reprezentowany jest poprzez enum `DefaultState`. W języku Java każda wartość enum jest obiektem klasy pochodnej po klasie `Enum`. Dodatkowo każdy z tych obiektów może implementować interfejsy na inny sposób. Fakt ten został wykorzystany w implementacji wzorca. `DefaultState` implementuje interfejs `State`, który pozwala na wykonanie akcji w zależności od stanu w jakim znajduje się maszyna. Wykorzystanie klasy `Enum` do implementacji wzorca stan jest rozwiązaniem zaproponowanym przez Roberta C. Martina w jego materiałach video [73]. W implementacji tej występują pojęcia zdarzenia, stanu oraz akcji. Akcja jest zdefiniowana dla danej pary stan-zdarzenie. W naszej implementacji, dodatkowo każda akcja może generować nowe zdarzenie. `DelegationHandler` reaguje na dwa zdarzenia z zewnątrz maszyny - `OVERFLOW` (podczas `WorkerPoolOverflow`) oraz `PRIMARY` (po aktualizacji stanu klastra, który następuje przy otrzymaniu komunikatu `HeartBeat`). Wykorzystana tutaj nazwa `PRIMARY` może dziwić. Jest ona związana z dalszymi planami rozwoju systemu o system monitorujący, który wprowadza podział na komunikaty `PrimaryHeartBeat` i `ExtendedHeartBeat`. System ten jest omówiony w rozdziale *Możliwości rozwoju jNode*.

Podczas zdarzenia `OVERFLOW`, gdy maszyna jest w stanie `NO_DELEGATION` przechodzi do stanu `DURING_DELEGATION`. W stanie tym wykonywana jest m.in. akcja delegowania zadań. W przypadku, gdy nastąpi kolejne zdarzenie `OVERFLOW`, a maszyna będzie w stanie `DURING_OVERFLOW`, to przejdzie w stan `SCHEDULED_RE_EXECUTE`. Oznacza to, że po skończeniu obecnej delegacji zadań mechanizm ma się uruchomić jeszcze raz w celu upewnienia się, że wszystkie zadania zostały oddelegowane. Dodatkowo, gdy podczas wykonywania akcji delegowania zabraknie wątków również w innych węzłach lub w klastrze jest jedynie jeden węzeł, maszyna przejdzie w stan `AWAITING_THREADS`. Na stan ten maszyna reaguje podczas zdarzenia `PRIMARY`. Wtedy jest sens, aby ponowić akcję delegowania zadań. Dodatkowo akcja `delegateTasks` po wykonaniu, zakładając że nie

zakończyła się z powodu braku wątków generuje zdarzenie DELEGATION_FINISHED, które służy do ponownego wykonania akcji delegateTasks, gdy w międzyczasie pojawiło się zdarzenie OVERFLOW. Generowanie tego zdarzenia jest kwestią techniczną i będzie wyjaśnione przy okazji omawiania mechanizmu zmiany stanu i wywoływania akcji. Zdarzenie NO_THREADS ma na celu przejście w stan AWAITING_THREADS i jest ono wykorzystywane jedynie do wywołania mechanizmu zmiany stanu i wywołania akcji (w tym przypadku jedynie służy do zmiany stanu). Zestawienie stanu, zdarzenia i wykonywanej akcji reprezentuje poniższa tabela.

| Obecny stan | Zdarzenie | Stan docelowy | Wykonana akcja |
|----------------------|---------------------|----------------------|-----------------|
| NO_DELEGATION | OVERFLOW | DURING_DELEGATION | delegateTasks() |
| NO_DELEGATION | PRIMARY | NO_DELEGATION | empty() |
| NO_DELEGATION | DELEGATION_FINISHED | NO_DELEGATION | error() |
| NO_DELEGATION | NO_THREADS | NO_DELEGATION | error() |
| DURING_DELEGATION | OVERFLOW | SCHEDULED_RE_EXECUTE | empty() |
| DURING_DELEGATION | PRIMARY | SCHEDULED_RE_EXECUTE | empty() |
| DURING_DELEGATION | DELEGATION_FINISHED | NO_DELEGATION | empty() |
| DURING_DELEGATION | NO_THREADS | AWAITING_THREADS | empty() |
| SCHEDULED_RE_EXECUTE | OVERFLOW | SCHEDULED_RE_EXECUTE | empty() |
| SCHEDULED_RE_EXECUTE | PRIMARY | SCHEDULED_RE_EXECUTE | empty() |
| SCHEDULED_RE_EXECUTE | DELEGATION_FINISHED | DURING_DELEGATION | delegateTasks() |
| SCHEDULED_RE_EXECUTE | NO_THREADS | AWAITING_THREADS | empty() |
| AWAITING_THREADS | OVERFLOW | AWAITING_THREADS | empty() |
| AWAITING_THREADS | PRIMARY | DURING_DELEGATION | delegateTasks() |
| AWAITING_THREADS | DELEGATION_FINISHED | AWAITING_THREADS | empty() |
| AWAITING_THREADS | NO_THREADS | AWAITING_THREADS | error() |

Tabela 1. Tabela przejść maszyny stanowej DelegationHandler.

Oprócz podstawowej akcji delegowania zadań wykorzystano akcje empty, która nic nie robi, oraz error, która zgłasza błąd w programie. Akcja delegateTasks, zgodnie z rozważaniami deleguje wszystkie dostępne zadania. Jest ona wykonywana zaraz po przejściu do stanu TASK_DELEGATION, będąc w stanie SCHEDULED_RE_EXECUTION przy zdarzeniu DELEGATION_FINISHED oraz przy zdarzeniu PRIMARY, będąc w stanie

AWAITING_THREADS. Z zdefiniowanych w ten sposób przejść, wynika, że `delegateTasks` może wykonywać się tylko jedna na raz. Nie da się wykonać tej akcji współbieżnie.

Wspomniany wcześniej mechanizm zmiany stanu i wywoływania akcji, dla każdego zdarzenia wykonuje następujący algorytm:

1. Dla danej pary: stan maszyny, zdarzenie zmień stan maszyny na powiązany stan docelowy.
2. Wykonaj powiązaną z przejściem akcję.
3. Jeżeli akcja zwróciła zdarzenie, wykonaj ponownie punkt nr 1.

Jak wcześniej wspomniano, zastosowane generowanie zdarzenia `DELEGATION_FINISHED` jest sprawą techniczną. Zostało zastosowane, ponieważ zmiana stanu wewnątrz akcji `delegateTasks` w stan `SCHEDULED_RE_EXECUTION` przy wykorzystaniu powyższego mechanizmu mogłaby spowodować przepełnienie stosu wywołań akcji `delegateTasks`, gdyby kolejne wywołania `delegateTasks` były kolejkowane. Dzięki temu podejściu jedynie jedna akcja `delegateTasks` może być na stosie.

Elementem, który wydaje się wymagać synchronizacji jest moment przejścia maszyny z jednego stanu do drugiego. Okazuje się, że i to udało nam się zrealizować bez jakiejkolwiek synchronizacji. Aby jej uniknąć zastosowaliśmy podejście oparte o atomową operację `compareAndSet` udostępnianą z poziomu klasy `AtomicReference`, omawianej przy okazji pakietu `java.util.concurrent`. Poniżej fragment programu, odpowiadający za realizację punktu pierwszego mechanizmu zmiany stanu i wywoływania akcji:

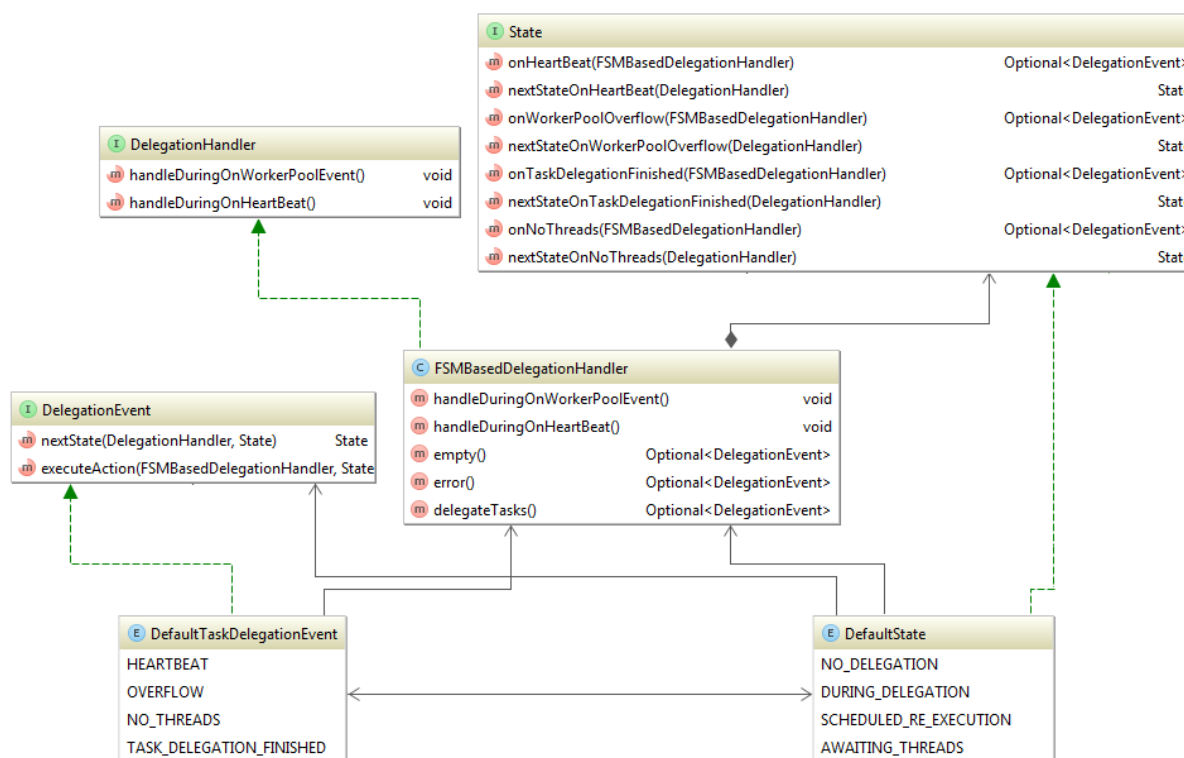
```
72 do {  
73     prevState = delegationState.get();  
74     nextState = event.nextState(this, prevState);  
75     while (!delegationState.compareAndSet(prevState, nextState));  
}
```

Kod źródłowy 4. Mechanizm atomowej zmiany stanu maszyny stanowej `DelegationHandler`.

Zmienna `delegationState` jest instancją `AtomicReference<State>`. Po wywołaniu metody `get()` otrzymujemy stan maszyny. Każde zdarzenie jest instancją `DefaultTaskDelegationEvent` i ma własną implementację metody `nextState`, dzięki implementacji interfejsu `TaskDelegationEvent`. Każda implementacja metody `nextState` wywołuje na obiekcie `prevState` metodę `nextStateOn[nazwa zdarzenia]`, natomiast metody te

zwracają stan docelowy dla każdej pary stan obecny, zdarzenie. Inaczej mówiąc, zostało zastosowane podwójnie polimorficzne wywołanie w celu zdefiniowania przejść maszyny stanowej. W warunku pętli wykonywana jest operacja compareAndSet, która zmieni stan na następny tylko, gdy stan maszyny jest taki sam jak w czasie jego pobrania metodą get(). Symulowana jest w ten sposób operacja atomowa. Z perspektywy dalszego wykonywania programu nie ma znaczenia, czy pętla ta wykonała się raz czy więcej razy. Również nie ma znaczenia czy maszyna w tym czasie przeszła przez inny stan i zdążyła już wrócić do tego samego. Efekt jest taki jakby operacja przejścia do innego stanu wykonała się atomowo.

Na poniższym diagramie znajduje się zestaw klas reprezentujący maszynę stanową DelegationHandler.



Rysunek 22. Zestaw klas mechanizmu DelegationHandler.

Podsumowując DelegationHandler jest to mechanizm umożliwiający zarządzanie delegowaniem zadań do innych węzłów, który reaguje na zmiany stanów każdej puli w klastrze. Został napisany w stylu asynchronicznym, aby wydajnie realizować swoje zadanie.

[Michał Semik]

Mając ustanowiony klaster jNode, gdy chcemy wykonać kod archiwum jar, wystarczy, że do jednego folderu JarPath skopiujemy nasze archiwum. Natomiast każdy węzeł w klastrze może otrzymać zadania do wykonania z tego archiwum. Do wykonania otrzymanego zadania węzeł również potrzebuje mieć dostęp do archiwum. W celu zapewnienia dostępu do niego rozważane były dwa podejścia.

Podejście 1

Węzeł zlecający zadanie wysyła archiwa do węzłów przed przesłaniem pierwszego zadania.

Podejście 2

Węzeł odbierający zadanie odpytuje w razie potrzeby węzeł macierzysty o jara do danego zadania.

Podejście nr 1 wymaga dodatkowego utrzymywania informacji o przechowywanych archiwach w innych węzłach. Jest ono o tyle dobre, że nie wymaga żadnej obsługi przy odbieraniu zadania. Natomiast może się zdarzyć, że przesyłane archiwum jest wysyłane niepotrzebnie lub w ogóle go tam nie ma. Przykładowo, gdy węzeł A wysłał zadanie do węzła B, natomiast węzeł B stworzył podzadanie, które próbował wysłać do węzła C. W tym momencie węzeł B nie wie, czy węzeł C posiada to archiwum, czy nie, a posiadać już może. Innym przypadkiem może być sytuacja, w której węzeł A wysłał zadanie do węzła B. Następnie węzeł B usunął archiwum. W tym samym momencie węzeł A może wysyłać do B kolejne zadanie, natomiast brak archiwum doprowadziłby do błędu.

W wybranym przez nas podejściu nr 2, węzeł sam pilnuje, aby mieć wymagane archiwum. Po otrzymaniu zadania, sprawdza czy archiwum istnieje. Jeśli archiwum nie istnieje wysyła komunikat JarRequest do węzła macierzystego. Węzeł odpytany o archiwum odpowiada komunikatem JarDelivery z jego zawartością. Dopiero po dostarczeniu archiwum zadanie może być wykonane, dlatego też wszystkie zadania, które są otrzymane od węzła macierzystego, muszą być przetrzymywane aż do otrzymania archiwum. Dopiero po

otrzymaniu archiwum zadania są przekazywane do wykonania. Mechanizm ten realizuje klasa nazywająca się JarHandler. Mechanizm ten zapewnia, że archiwum istnieje w momencie wykonywania zadania, nawet gdyby użytkownik niechcący go usunął - zostanie w razie potrzeby pobrany ponownie. Zapobiega także nadmiarowemu przesyłaniu archiwów i doskonale radzi sobie przy delegowaniu zadań z węzła pomocniczego. Możliwość wprowadzenia udoskonalenia mechanizmu została omówiona w rozdziale *Możliwości rozwoju jNode*.

Po otrzymaniu zadania od innego węzła, Distributor przekazuje zadanie do JarHandlera. Otrzymane zadanie jest instancją klasy ExternalTask. Jak było omawiane w rozdziale *Struktura tasków na poziomie modułu cluster*, ExternalTask posiada specjalną obsługę serializacji. ExternalTask zawiera instancję klasy implementującą interfejs Task. Zawarta instancja nie może zostać zdeserializowana przy użyciu class loadera jNode, ponieważ klasa tej instancji została zaimplementowana w archiwum użytkownika. W tym celu musi być wykorzystany class loader, który ma dostęp do klas użytkownika. JarHandler dokonuje deserializacji wszystkich zadań przed przekazaniem ich do wykonania. Do załadowania klas użytkownika JarHandler wykorzystuje class loader typu child first, omawiany w rozdziale *Moduł crosscutting*.

[Michał Semik]

Obsługa błędów

W jNode jak w każdym środowisku rozproszonym ilość sytuacji wyjątkowych jest duża, ponieważ każdy z elementów środowiska może ulec awarii w dowolnym momencie.

Jak już było omawiane przy okazji JGroups, system odbiera komunikaty o zmianie stanu węzłów za pomocą metody viewAccepted. Na podstawie obecnego stanu i poprzedniego, pozyskujemy informacje, które węzły opuściły klaster (lub uległy awarii) i które do niego dołączyły. Informacja ta w postaci wywołań metod onNewNode i onNodeGone jest przekazywana do Distributora.

W systemie istnieją dwa rodzaje zasobów, których stan należy skorygować przy awarii węzła. Jednym z zasobów jest informacja o samym węźle, w tym przypadku korekta sprowadza

się do usunięcia węzła z listy podczas komunikatu `onNodeGone`. Drugim zasobem są zadania i na nich skupia się ten rozdział.

Zakładając, że węzeł opuścił klaster, pozostałe węzły w klastrze muszą rozważyć dwa typy zadań: zadania, które zostały oddelegowane do tego węzła oraz zadania otrzymane od tego węzła. Zadania te przetrzymywane są w rejestrach, odpowiednio `DelegatedTaskRegistry`, jak i `ExternalTaskRegistry`, co było omawiane w rozdziale *Rejestry tasków, czyli nadzorowanie stanu systemu*. Zadania te są jednak odrębnie traktowane. Zadania z `DelegatedTaskRegistry` zostaną ponownie zlecone do wykonania w puli, dzięki temu praca archiwów uruchomionych z pozostałych węzłów może zostać kontynuowana. Natomiast dla zadań z `ExternalTaskRegistry` nie ma węzła, dla którego powinny zostać zwrócone wyniki, zatem nie można ukończyć prac. Zdecydowaliśmy, że w tym przypadku zadania powinny zostać anulowane. Istnieje szansa, że awaria została spowodowana krótkotrwałym brakiem dostępu do sieci, w takim przypadku węzeł odłączony ponowi wykonywanie swoich tasków, a stan w klastrze pozostanie spójny.

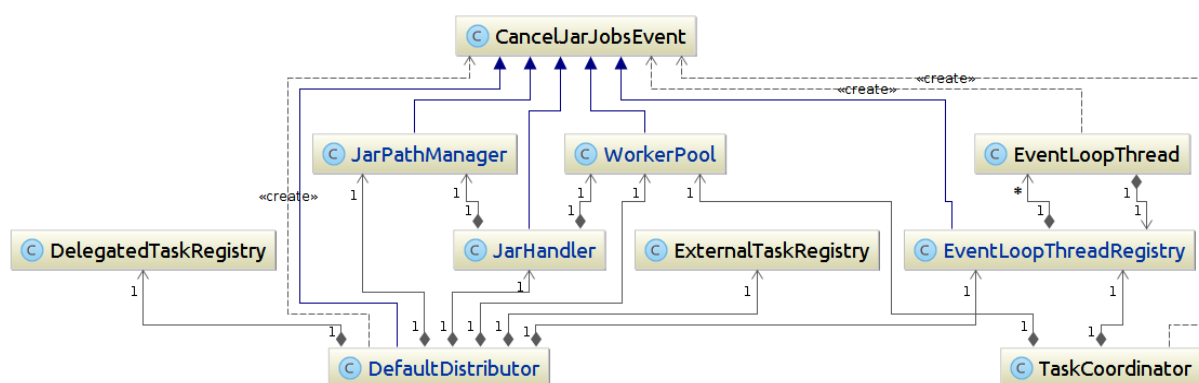
Problem anulowania wszystkich zadań w systemie jak się okazało nie należy do trywialnych. Jest tak, ponieważ po otrzymaniu komunikatu `onNodeGone` nie wiadomo dokładnie, gdzie znajduje się zadanie. Może się zdarzyć, że zadanie:

1. Jest przetrzymywane w `JarHandlerze`, ponieważ zadanie oczekuje na archiwum.
2. Jest już w puli i oczekuje na wykonanie.
3. Jest w puli i jest wykonywane przez wątek puli.
4. Zostało już wykonane i wyrejestrowane z puli. Jest w zmiennej lokalnej wątku puli wysyłającego wynik zadania do węzła macierzystego.
5. Komunikat `Task Execution Completed` został już wysłany do węzła macierzystego.

Jak już było mówione przy okazji omawiania `JGroups`, zdecydowaliśmy się na rozwiązanie, które realizuje komunikaty otrzymane od innych węzłów sekwencyjnie, w tym także `onNodeGone`. Dzięki takiemu rozwiązaniu nie może się zdarzyć, że podczas obsługi `onNodeGone` zadanie znajduje się w zmiennej lokalnej wątku i dopiero trafi do puli po jego obsłudze. Może się jednak zdarzyć, że podzadanie na węzle pomocniczym zostało dodane do puli i nie zostało jeszcze zarejestrowane. W takim skrajnym przypadku zadanie to zostanie wykonane, natomiast dalsze działania dla niego nie zostaną podjęte.

Przypadek (4) zdecydowaliśmy zignorować. Wiadomość zostanie wysłana do nieistniejącego już węzła. Do wszystkich pozostałych przypadków wykorzystujemy ujednolicony mechanizm anulowania zadań oparty o zdarzenie `CancelJarJobsEvent`, które jest generowane podczas wykonywania metody `onNodeGone`. Na wymienione zdarzenie reaguje zarówno `JarHandler`, jak i `WorkerPool` w celu anulowania przechowywanych, oczekujących i wykonywanych zadań. Oprócz kluczowych elementów na zdarzenie oczekuje także `JarPathManager`, zapisujący w pliku properties stan wykonywania archiwum na anulowany.

Oprócz odłączenia się węzła od klastra, awaryjne zakończenie zadania może wystąpić z innych powodów. Użytkownik mógł usunąć archiwum. Zdecydowaliśmy się, że w takim przypadku `TaskCoordinator` publikuje zdarzenie `CancelJarJobsEvent`. Również podczas wykonywania metody `onSuccess` lub `onFailure` w `Callback`, w wątku `EventLoopThread` może zostać rzucony wyjątek. W takim przypadku `EventLoopThread` ogłasza `CancelJarJobsEvent`. Jeżeli wewnątrz węzła zostanie zgłoszony `CancelJarJobsEvent`, do wszystkich węzłów, dla których istnieją oddelegowane zadania, wysyłany jest komunikat `CancelJarJobs`, na który reaguje `Distributor` generując zdarzenie `CancelJarJobsEvent` w węzłach pomocniczych. Dzięki takiemu podejściu, jeden komunikat oraz jedno zdarzenie odpowiada za anulowanie zadań w całym systemie. Zestaw klas uczestniczących w anulowaniu zadań oraz samo zdarzenie `CancelJarJobsEvent` przedstawia poniższy diagram.

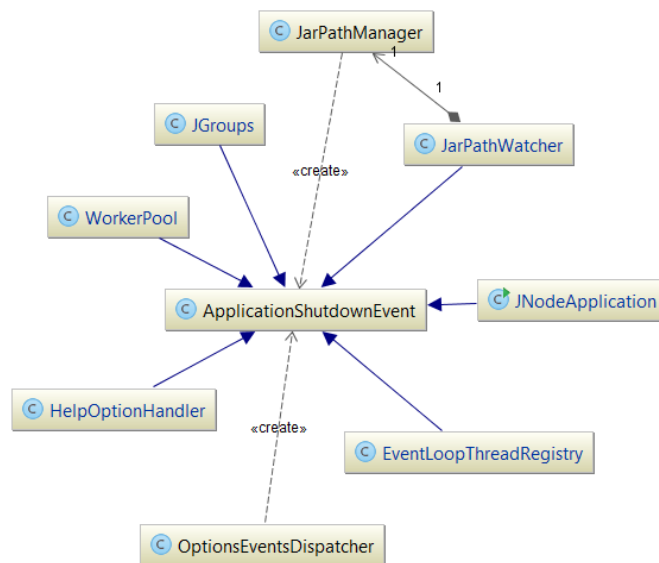


Rysunek 23. Diagram przedstawiający zdarzenie `CancelJarJobsEvent` i powiązane klasy.

[Michał Semik]

Zwalnianie zasobów przy zamykaniu aplikacji

Ciekawym aspektem systemu jest jego wyłączanie. W jNode istnieje kilka typów wątków: wątki klasy EventLoopThread, wątki w puli WorkerPool, JarPathWatcher posiada własny wątek. Klasa JGroups posiada jednowątkową pulę, a także biblioteka JGroups posiada własne wątki i również wymaga rozłączenia z klastrem. Także Spring Framework korzysta z puli wątków, w mechanizmie umożliwiającym okresowe wywołania metod. W celu poprawnego zakończenia programu, np. w sytuacji, gdy podano nieprawidłowe opcje programu stworzyliśmy ujednolicony mechanizm zamykania platformy oparty o zdarzenie ApplicationShutdownEvent. Poniższy diagram reprezentuje klasy powiązane z tym zdarzeniem.



Rysunek 24. Przedstawia ApplicationShutdownEvent i powiązane klasy tworzące zdarzenie i reagujące na nie.

[Michał Semik]

Spis komunikatów w systemie

1. Delegacja zadań (Task Delegation). To komunikat zawierający ExternalTask. Jest on wysyłany, gdy zlecamy zadania zdalnemu węzłowi.
2. Dostarczenie archiwum jar (Jar Delivery). Komunikat zawiera nazwę archiwum oraz jego zawartość jako tablicę bajtów. Wysyłany zostaje od razu po otrzymaniu komunikatu Jar Request.
3. Prośba o dostarczenie archiwum jar (Jar Request). Komunikat jest wysyłany przez mechanizm JarHandler po otrzymaniu zadania, dla którego archiwum nie istnieje.
4. Primary HeartBeat. Komunikat informujący o stanie węzła, jego liczbie wątków w puli i obecnie zajętych wątków. Jest wysyłany co stały okres czasu.

5. Przeniesienie zadania (Redirect). Wysyłany do węzła macierzystego, gdy węzeł pomocniczy próbuje przekazać ExternalTask innemu węzłowi (nie będącemu węzłem źródłowym dla tego zadania). W polu destinationNodeId zamieszczony zostaje identyfikator proponowanego węzła.
6. Rejestracja podzadania (Register Delegated SubTask). Komunikat wysyłany do węzła źródłowego w celu poinformowania o utworzeniu podzadania w węźle pomocniczym.
7. Niepowodzenie wykonania ExternalTaska (Sry). Wysyłany, gdy nie ma już wątków dostępnych do zrealizowania zewnętrznego zadania podczas próby dalszego delegowania tego zadania i wybranym przez mechanizm wyboru węzła węzłem docelowym został źródłowy.
8. Wykonano zadanie (Task Execution Completed). Wysyłane, gdy węzeł zrealizuje zadanie zewnętrzne. W komunikacie załączony jest wynik wykonania zadania lub rzucony wyjątek w trakcie wykonywania zadania.
9. Anulowanie wykonywania zadań związanych z danym archiwum jar (Cancel Jar Jobs). Wysyłany przez węzeł źródłowy do wszystkich węzłów pomocniczych w celu zaprzestania wykonywania zadań powiązanych z archiwum, reprezentowanym przez jego nazwę.

[Michał Semik]

Moduł crosscutting

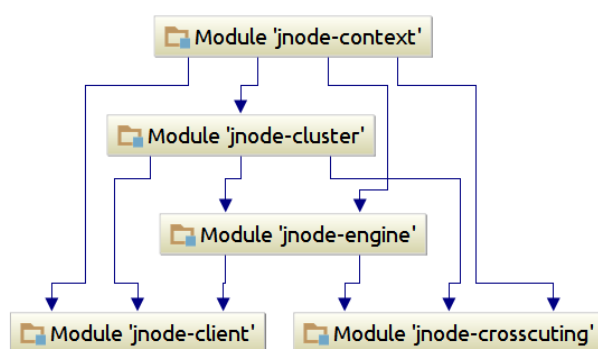
Moduł crosscutting zawiera narzędzia, które są odpowiedzialne za zagadnienia przekrojowe [67]. Najważniejszym elementem tego modułu jest implementacja własnego class loadera.

Jak było omawiane we wstępie, ClassLoader jest to klasa, której obiekty służą do ładowania klas. Podstawowym założeniem class loaderów jest, że każdy może mieć ojca. Natomiast klasy ładowane są najpierw z ojca, a potem z dziecka. Jest to dobry przykład zastosowania wzorca projektowego *Chain of Responsibility* [74]. Założenie jest sensowne, ponieważ jeśli w ojcu już wcześniej została załadowana klasa, to nie zostanie ona załadowana po raz drugi.

W celu załadowania archiwum jar mogliśmy skorzystać z klasy `URLClassLoader`, dostarczanej przez wbudowaną w język Java bibliotekę. Naszym problemem okazało się założenie, że klasy muszą być ładowane najpierw z ojca. Gdy ustawimy jako ojca class loader systemowy, doprowadzi to do tego, że najpierw będą ładowane klasy z `jNode`, a dopiero potem z aplikacji użytkownika. Przykładowo, nasza aplikacja korzysta z Spring Framework. Jeżeli aplikacja użytkownika też korzysta z tego narzędzia, do tego w innej wersji, to aplikacja korzystałaby z klas frameworka w nieodpowiedniej wersji, co może doprowadzić do trudnych do wykrycia błędów. W `jNode` potrzebowaliśmy class loader, który ładuje najpierw klasy użytkownika, a dopiero potem class loader systemowy. Realizuje to tzw. `child first class loader`. Problem jest dość popularny, aczkolwiek nie znaleźliśmy odpowiedniej dla nas implementacji. Powodem tego było dodatkowe wymaganie, że oprócz `child first class loadera`, potrzebny był class loader, który ładuje klasy jedynie z archiwum jar, nie korzystając z klas systemowych. Mechanizm ten przydał się do usprawnienia tzw. `classpath scanning`, omawianego w rozdziale *Moduł context*. Zaimplementowaliśmy class loader, który posiada zarówno semantykę `child-first`, jak i `child-only`, zależnie od ustawienia. Implementacja ta nie była prosta w realizacji ani nie stanowi sztandarowego przykładu rzemiosła, aczkolwiek działa i znajduje się właśnie w tym module.

[Michał Semik]

Moduł context



Rysunek 25. Zależności modułu context.

Moduł `context` zawiera implementację uproszczonego kontenera CDI (*Context and Dependency Injection*). Dla przypomnienia kontener CDI to obiekt (nazywany często fabryką) odpowiedzialny za tworzenie obiektów i ich odpowiednie łączenie (wstrzykiwanie zależności).

Kontener w celu powiązania obiektów posługuje się konfiguracją, która określa jak obiekty powinny być powiązane. Wstrzykiwanie zależności jest jedną z realizacji zasady odwrócenia sterowania (*Inversion of Control*) w sensie tworzenia i wiązania obiektów. Pozwala ona na tworzenie kodu o luźniejszych powiązaniach (*loose coupling*). Pojęcia te zostały omówione we wstępie.

Obecnie na rynku istnieje wiele rozbudowanych bibliotek obsługujących wstrzykiwanie zależności i dostarczających implementacji kontenera CDI. Wiele z nich poza samym tworzeniem obiektów i wstrzykiwaniem zależności posiada dodatkowo inne funkcjonalności jak np.:

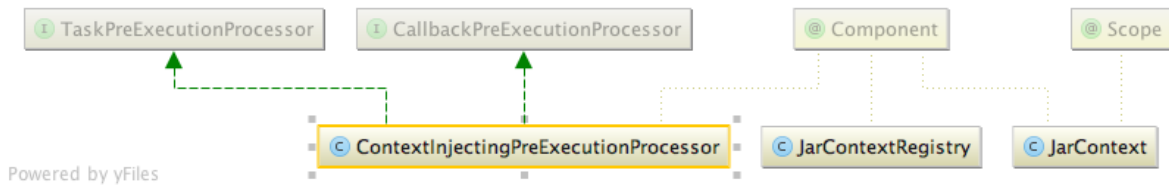
- integracja z językiem EL (Expression Language)
- możliwość dekorowania wstrzykniętych komponentów
- model zdarzeń-powiadomień
- zastosowanie interceptorów, czy też aspektów
- oraz wiele innych w zależności od biblioteki i implementacji.

Dla potrzeb jNode stworzyliśmy własną implementację uproszczonego kontenera CDI. Jego zadaniem jest umożliwienie interakcji pomiędzy obiektami nieposiadającymi stanu jakimi są zadania (taski) oraz wywołania zwrotne (callbacks), poprzez wprowadzenie specjalnego obiektu *context*. Obiekt ten jest tworzony jako singleton w kontenerze i wstrzykiwany przez niego do odpowiednich zadań i wywołań zwrotnych przed ich wykonaniem. W kontenerze jednak może być wiele różnych obiektów *context* w oparciu o ich różne typy. Wstrzykiwanie zależności odbywa się w oparciu o typy Javy, a nie w oparciu o tekst, nazwy obiektów, więc dodatkowo spełniona jest zasada *Type Safety* [75] umożliwiająca lepsze wykrywanie ewentualnych błędów. Wszystko to ułatwia użytkownikowi pisanie własnych programów z wykorzystaniem jNode.

W rozdziale *Moduł client* zostały opisane adnotacje, które użytkownik może użyć, jeżeli chce skorzystać z wyżej wspomnianych funkcjonalności. Dla przypomnienia są to trzy adnotacje:

- `@ContextScan`
- `@Context`
- `@InjectContext`

Przy ich pomocy użytkownik definiuje, które klasy mają zostać skanowane przez moduł context, jakie obiekty *context* powinien utworzyć kontener i gdzie obiekty te powinny zostać wstrzyknięte przed wykonaniem zadań, czy wywołań zwrotnych.



Rysunek 26. Diagram reprezentujący wybrane klasy modułu.

Moduł context składa się z trzech głównych klas realizujących opisywaną funkcjonalność. Podstawową klasą jest JarContext, której obiekty pełnią rolę kontenerów CDI. Dla każdej aplikacji klienta, czyli archiwum jar będzie utworzony jeden kontener. Obiekty te są tworzone i rejestrowane przez jednostkę o nazwie JarContextRegistry, która nimi zarządza. Interfejsy CallbackPreExecutionProcessor, TaskPreExecutionProcessor i ich domyślna implementacja o nazwie ContextInjectingPreExecutionProcessor są wykorzystywane przez TaskCoordinatora i EventLoopThread w celu przygotowania odpowiednio zadań asynchronicznych i wywołań zwrotnych do wykonania. Zostały one wprowadzone do odwrócenia zależności. Obsługa jest przekazana do kontenera, a przygotowanie obiektów do wykonania polega na wstrzyknięciu przez kontener w postaci JarContext odpowiednich zależności.

Po umieszczeniu archiwum jar zawierającego aplikację w systemie zostaje ogłoszone zdarzenie o nazwie NewJarCreatedEvent. Zdarzenie to jest między innymi obsługiwane przez JarContextRegistry, który tworzy i rejestruje dla umieszczonego archiwum jar odpowiadający mu JarContext. JarContext podczas procesu tworzenia wykrywa, czy w danym archiwum jar znajdują się przedstawione wcześniej adnotacje. Poza wspomnianymi adnotacjami sprawdza również, czy używana jest adnotacja @Autowired udostępniana przez Spring Framework. Jeżeli adnotacje te znajdują się w archiwum jar i są używane, to pobierana jest lista prefiksów pakietów z adnotacji @ContextScan umieszczonej nad klasą zawierającą funkcję main. Lista ta jest wykorzystywana w procesie skanowania klas. Do procesu skanowania wykorzystywana jest klasa udostępniona przez Spring Framework o nazwie ClassPathScanningCandidateComponentProvider, która wykonuje *classpath scanning*, polegający na przeszukiwaniu drzewa zasobów poprzez wywołanie metody

ClassLoader.getResourceAsStream. Mechanizm ten działa dość wolno, jeśli jako argumentu nie poda się początkowego pakietu, dlatego też wspomniana lista prefiksów jest konieczna w celu przyspieszenia skanowania. Przy wykorzystaniu obiektu tej klasy, listy prefiksów pakietów, class loadera odpowiedzialnego za załadowane klas z archiwum jar (class loader został omówiony w rozdziale *Moduł crosscutting*) oraz adnotacji @Context, która pełni rolę filtra zostają wykryte wszystkie klasy opatrzone adnotacją @Context. Po skanowaniu zostają utworzone w kontenerze pojedyncze instancje (singletony) znalezionych klas.

Przed wykonaniem zadań asynchronicznych i wywołań zwrotnych zostają one przekazane do pre-procesorów, których zadaniem jest przygotowanie tych obiektów do wykonania. Pre-procesory przekazują obsługę do kontenera, którego odpowiedzialnością jest wstrzyknięcie odpowiednich zależności. Polega to na tym, że JarContext wykorzystując mechanizm refleksji pobiera dla danego obiektu wszystkie pola, które zostały opatrzone adnotacjami @InjectContext lub @Autowired. Następnie spośród wcześniej utworzonych obiektów poszukiwany jest w oparciu o typ ten obiekt, będący beanem, który powinien zostać wstrzyknięty. Po wstrzyknięciu zależności zadania lub wywołania zwrotne są gotowe do wykonania.

[Alan Hawrot]

Moduł main

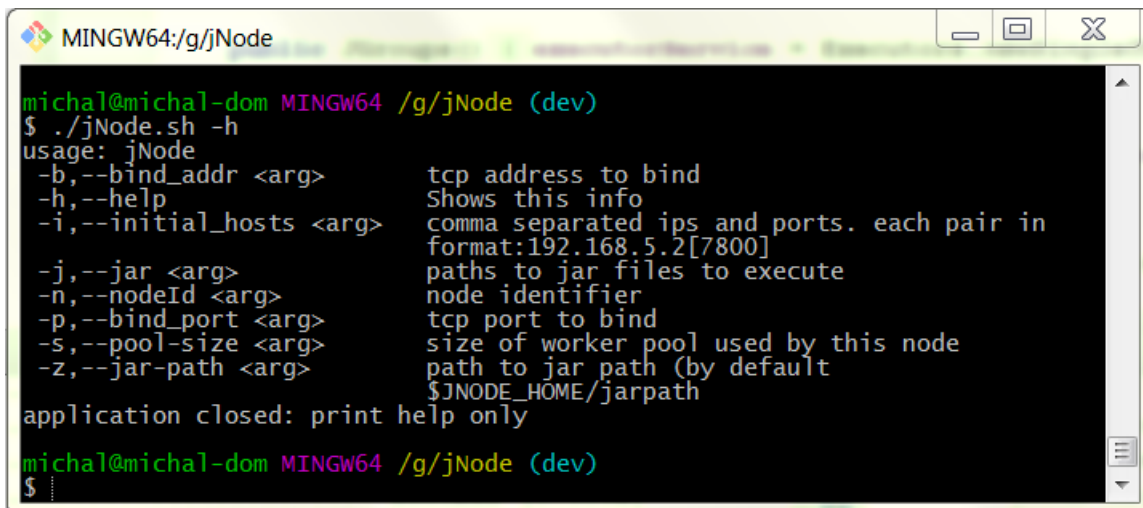
Moduł main ma za zadanie zainicjalizowanie aplikacji. Włączenie aplikacji jest realizowane poprzez uruchomienie skryptu jNode.sh, który włącza maszynę wirtualną Javy oraz uruchamia funkcję main klasy JNodeApplication. Aplikacja nie wymaga uruchomienia skryptu, natomiast jest to wygodne. Można bezpośrednio uruchomić aplikację korzystając z następującego polecenia:

```
java -jar -Djava.net.preferIPv4Stack=true jnode-platform-0.0.1-SNAPSHOT.jar
```

Polecenie to uwzględnia argument java.net.preferIPv4Stack=true. Argument ten jest wymagany na systemach m.in. OSX i Windows 7 przez bibliotekę JGroups. Dodając tę flagę wymuszamy używanie przez JVM innej implementacji datagramów UDP, takiej, z którą JGroups potrafi współpracować. Informację tę pozyskaliśmy z WIKI JGroups [76].

Cała platforma jest oparta o Spring Framework. Wszystkie obiekty głównych klas w systemie są beanami (singletonami) inicjalizowanymi wraz z inicjalizacją podstawowej klasy Spring Framework, `ApplicationContext`. Dlatego też pierwszym krokiem, który realizuje platforma jest utworzenie obiektu `ApplicationContext` i inicjalizacja beanów.

Kolejnym etapem inicjalizacji jest rozpropagowanie zdarzeń, po jednym dla każdej podanej opcji programu. Każdy element w systemie, który jest zainteresowany opcją, przechwytywa zdarzenie i pobiera z niego argumenty. Podstawową opcją jest opcja „--help”, która wyświetla informacje i kończy działanie programu.



```
mingw64:/g/jNode
$ ./jNode.sh -h
usage: jNode
  -b,--bind_addr <arg>      tcp address to bind
  -h,--help                  Shows this info
  -i,--initial_hosts <arg>  comma separated ips and ports. each pair in
                             format:192.168.5.2[7800]
  -j,--jar <arg>            paths to jar files to execute
  -n,--nodeId <arg>         node identifier
  -p,--bind_port <arg>      tcp port to bind
  -s,--pool-size <arg>      size of worker pool used by this node
  -z,--jar-path <arg>       path to jar path (by default
                             $JNODE_HOME/jarpath
application closed: print help only

mingw64:/g/jNode
$
```

Rysunek 27. Opcje programu.

Po rozpropagowaniu zdarzeń generowane jest zdarzenie `OptionsDispatchedEvent`, które oznacza, że wszystkie opcje zostały rozpropagowane. Jest ono generowane po to, aby elementy systemu, które wymagają pewnych opcji do działania mogły przyjąć wartości domyślne z powodu ich braku.

Ostatnim zdarzeniem generowanym w funkcji `main` jest `ApplicationInitializedEvent`, na które reagują wszystkie obiekty zarządzające wątkami i inicjalizują wszystkie podstawowe wątki i pule w systemie.

Oprócz inicjalizacji obiektu klasy `ApplicationContext` klasa `JNodeApplication` jest odpowiedzialna także za jego zamykanie. Realizuje to poprzez nasłuchiwanie na zdarzenie `ApplicationShutdownEvent` i wywołanie odpowiedniej metody `close`.

Niektóre opcje są przydatne jedynie w procesie testowania. Opcja `-j`, pozwala na wykonanie archiwum `jar` dla zadanej ścieżki do archiwum. Implementacja tej opcji polega na skopiowaniu archiwum do katalogu `JarPath`. Opcja `-n` pozwala na ustalenie identyfikatora węzła. Jeżeli nie zostanie ona podana, to identyfikator węzła zostanie wygenerowany przez system na podstawie nazwy zalogowanego użytkownika w połączeniu z losowym ciągiem liczb. Opcja `-s` pozwala na ustawienie wielkości `WorkerPool`. Opcja `-z` pozwala na zmianę katalogu `JarPath` na dowolny, inny. Jest ona przydatna, gdy chcemy uruchomić więcej węzłów na jednym komputerze. Opcje `--bind_addr`, `--bind_port` i `--initial_hosts` pozwalają na ustanowienie klastra węzłów na stosie `TCP` (domyślnie wykorzystywany jest `UDP`).

[Michał Semik]

Moduł monitor

Moduł `monitor` jest jedynie załącznikiem kolejnej funkcjonalności, która ma powstać w systemie. Planowany jest rozwój modułu w celu umożliwienia graficznego monitorowania stanu w klastrze, przede wszystkim wyświetlania węzłów w klastrze, monitorowania posiadanych archiwów, istniejących zadań, wolnych zasobów obliczeniowych. Na dany moment, niektóre z tych informacji wyświetlane są jedynie w konsoli.

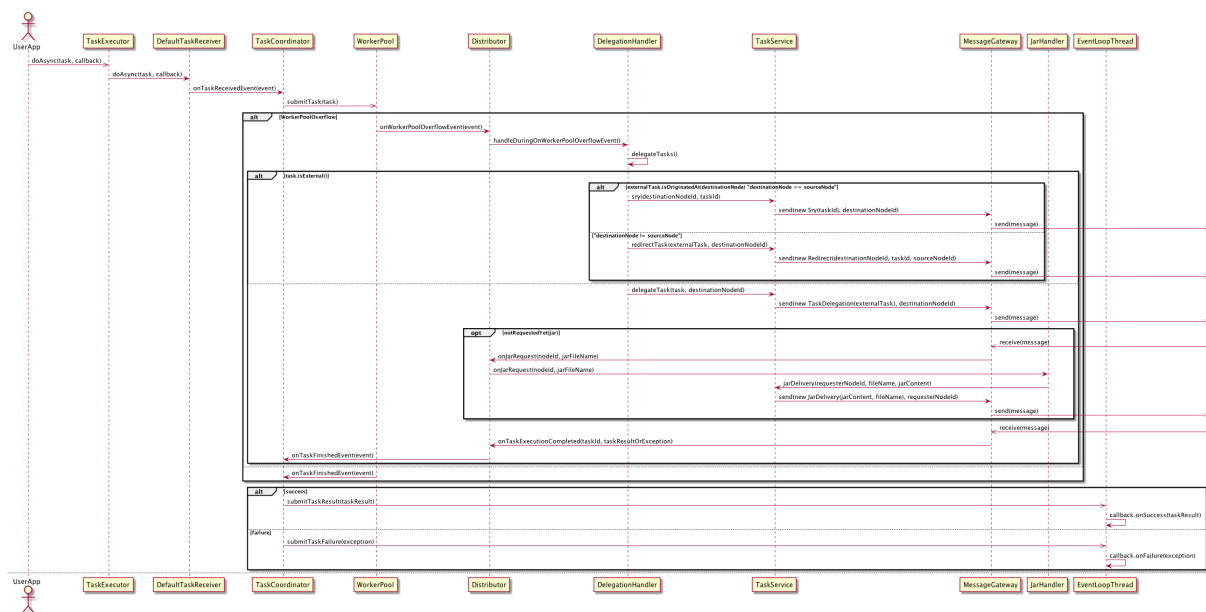
[Michał Semik]

Podsumowanie działania systemu jNode

zadania zostają umieszczone w kolejce zadań i oczekują na wykonanie. Po wykonaniu zadań asynchronicznych ich rezultaty są przekazywane ostatecznie do EventLoopThread, gdzie następuje wstrzyknięcie zależności do wywołań zwrotnych przez JarContext i ich wykonanie. Gdy wszystkie zadania zostaną wykonane następuje zamknięcie EventLoopThread oraz JarContext i ich usunięcie z rejestrów, a stan archiwum jar zostaje zmieniony na FINISHED w powiązanym mu pliku properties.

Każdy z działających w klastrze węzłów zna stan pozostałych. Jest to możliwe dzięki komunikatom HeartBeat, które węzły rozsyłają pomiędzy sobą informując o swoim stanie. Komunikaty te są obsługiwane przez jednostkę HeartBeatHandler. Węzły mogą w dowolnej chwili dołączać do klastra i się od niego odłączać. Zadania mogą być przesyłane pomiędzy węzłami w celu wykonania. Wprowadzenie do systemu mechanizmu wyboru węzła oraz priorytetowej kolejki zadań oczekujących zredukowało liczbę niekorzystnych sytuacji ponownego rozsyłania zadania i gwarancję, że każde z zadań zostanie wykonane. Gdy w danym węźle w kolejce zadań znajdują się zadania oczekujące na wykonanie, gdyż wszystkie wątki w WorkerPool są zajęte, węzeł ten może przesłać zadania do innego węzła. W takim przypadku zostaje ogłoszone zdarzenie WorkerPoolOverflowEvent, na które reaguje Distributor – centralna jednostka modułu cluster. Distributor przekazuje obsługę zdarzenia do jednostki o nazwie DelegationHandler, która zajmuje się oddelegowywaniem wykonywania zadań do innych węzłów. Zadania przesyłane w klastrze są na każdym z węzłów przechowywane w dwóch rejestrach: DelegatedTaskRegistry i ExternalTaskRegistry. Zostały one wprowadzone w celu nadzorowania stanu systemu i są wykorzystywane przy ewentualnej obsłudze błędów. Wszystkie komunikaty są wysyłane poprzez warstwę TaskService i ostatecznie JGroups. Komunikaty przychodzące do danego węzła również odbierane są przez JGroups, jednak ich przetwarzanie jest przekazywane do Distributora. Zatem po przesłaniu zadania w celu wykonania do wybranego węzła zostaje ono odebrane i przekazane do Distributora, który deleguje dalszą obsługę do jednostki JarHandler. Ze względu na to, że dany węzeł może nie posiadać załadowanej klasy otrzymanego zadania, JarHandler musi przechować to zadanie w specjalnym schowku na czas dostarczenia archiwum jar z definicjami wszystkich klas. Po otrzymaniu archiwum jar od węzła źródłowego następuje deserializacja zadania i umieszczenie go w WorkerPool. Po jego wykonaniu rezultat zostaje przekazany do Distributora, który poprzez warstwę TaskService wysyła komunikat do węzła źródłowego. Węzeł źródłowy po otrzymaniu komunikatu zawierającego rezultat wykonania zadania może wykonać powiązane

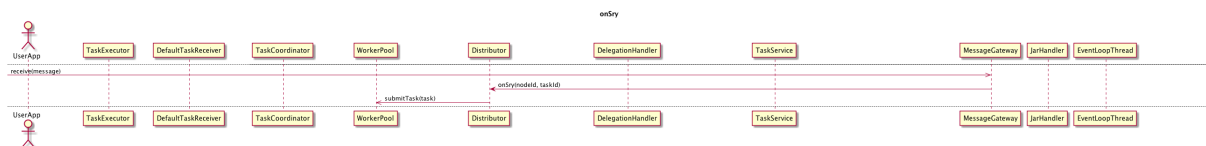
wywołanie zwrotne w odpowiednim EventLoopThread. W ten sposób procedura dochodzi do końca.



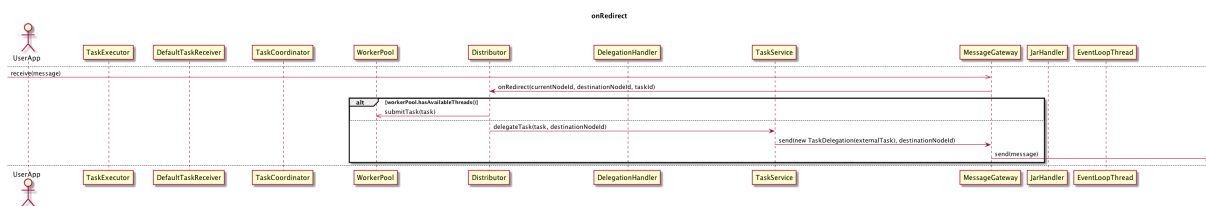
Rysunek 29. Diagram sekwencji przedstawiający podstawową, uproszczoną obsługę zadania asynchronicznego.



Rysunek 30. Diagram sekwencji obsługujący odelegowane zadanie asynchroniczne.



Rysunek 31. Diagram sekwencji przedstawiający obsługę komunikatu Sry.



Rysunek 32. Diagram sekwencji przedstawiający obsługę komunikatu Redirect.

Powyżej została omówiona i powtórzona podstawowa procedura wykonania zadania asynchronicznego w klastrze. Na diagramach przedstawiono najważniejsze jednostki odpowiedzialne za pracę instancji i komunikację z innymi instancjami platformy jNode. Natomiast wszystkie najważniejsze algorytmy, procedury, jednostki, zdarzenia i komunikaty zostały szczegółowo opisane i uzupełnione o dodatkowe informacje w poprzednich rozdziałach.

[Alan Hawrot]

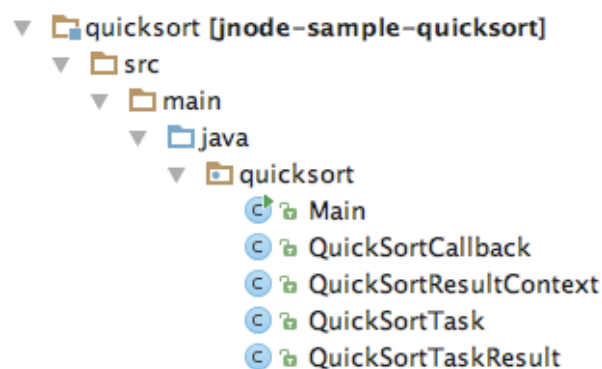
Biblioteka użytkownika i przykładowe programy

Rozdział ten ma na celu przybliżenie czytelnikowi w jaki sposób należy korzystać z biblioteki użytkownika. Zostały tutaj przedstawione przykładowe programy klienckie napisane przy użyciu tejże biblioteki. Aplikacje w postaci archiwum jar są gotowe do uruchomienia w klastrze jNode. Sama zawartość biblioteki użytkownika została już szczegółowo opisana w rozdziale *Moduł client*.

[Alan Hawrot]

Program: Quicksort

W programie tym został zaimplementowany algorytm **Sortowania szybkiego** (ang. *Quicksort*) działający na zasadzie „dziel i zwyciężaj”. Program składa się z jednego pakietu, który zawiera pięć klas.



Rysunek 33. Hierarchia plików programu Quicksort.

```

1  package quicksort;
2
3  import pl.edu.uj.jnode.context.ContextScan;
4  import pl.edu.uj.jnode.userlib.TaskExecutor;
5  import pl.edu.uj.jnode.userlib.TaskExecutorFactory;
6
7  import java.util.Arrays;
8  import java.util.Random;
9
10 @ContextScan("quicksort")
11 public class Main {
12     public static void main(String[] args) {
13         int[] array = new Random().ints(100000, 0, 500000).toArray();
14
15         System.out.println("Array to sort: " + Arrays.toString(array));
16
17         TaskExecutor taskExecutor = TaskExecutorFactory.createTaskExecutor();
18         taskExecutor.doAsync(new QuickSortTask(array, 0, array.length - 1), new QuickSortCallback());
19     }
20 }
21

```

Kod źródłowy 5. Klasa Main.

Klasa **Main** to klasa zawierająca metodę główną *main*, która zostanie wywołana jako pierwsza podczas uruchamiania programu (pierwsze zadanie). Została tutaj użyta adnotacja `@ContextScan` (linia 10.), za pomocą której wskazana została nazwa pakietu używana w celu wyszukania klas opatrzonych adnotacją `@Context`. Linia 13. zawiera deklarację i inicjalizację tablicy liczb typu `Integer` z przedziału `[0, 500 000]`. Tablica ma wielkość 100 000 elementów, które zostaną wygenerowane w sposób losowy. Następnie tworzony jest obiekt typu `TaskExecutor` (linia 16.), który ma na celu przekazywanie platformie `jNode` zadań asynchronicznych i powiązanych z nimi wywołań zwrotnych (linia 18.). W tym przypadku są to obiekty typu `QuickSortTask` i `QuickSortCallback`.

```

1  package quicksort;
2
3  import pl.edu.uj.jnode.userlib.Task;
4  import pl.edu.uj.jnode.userlib.TaskExecutor;
5  import pl.edu.uj.jnode.userlib.TaskExecutorFactory;
6
7  import java.io.Serializable;
8
9  public class QuickSortTask implements Task {
10     private int[] array;
11     private int begin;
12     private int end;
13
14     public QuickSortTask(int[] array, int begin, int end) {
15         this.array = array;
16         this.begin = begin;
17         this.end = end;
18     }
19
20     @Override
21     public Serializable call() throws Exception {
22         if (end - begin <= 100) {
23             insertionSort(array, begin, end);
24             return new QuickSortTaskResult(array, begin, end);
25         }
26
27         TaskExecutor taskExecutor = TaskExecutorFactory.createTaskExecutor();
28         int j = partition(array, begin, end);
29         taskExecutor.doAsync(new QuickSortTask(array, begin, j - 1), new QuickSortCallback());
30         taskExecutor.doAsync(new QuickSortTask(array, j + 1, end), new QuickSortCallback());
31         return new QuickSortTaskResult(array, j, j);
32     }
33
34     private int partition(int[] array, int lo, int hi) {
35         int i = lo;
36         int j = hi + 1;
37         int v = array[lo];
38         while (true) {
39             while (array[++i] < v) {
40                 if (i == hi) {

```

```

41         break;
42     }
43 }
44 while (v < array[--j]) {
45     if (j == lo) {
46         break;
47     }
48 }
49 if (i >= j) {
50     break;
51 }
52 swap(array, i, j);
53 }
54 swap(array, lo, j);
55 return j;
56 }
57
58 private void insertionSort(int[] array, int lo, int hi) {
59     for (int i = lo + 1; i <= hi; i++) {
60         int x = array[i];
61         int j = i - 1;
62         while (j >= 0 && array[j] > x) {
63             array[j + 1] = array[j];
64             j--;
65         }
66         array[j + 1] = x;
67     }
68 }
69
70 private void swap(int[] array, int i, int j) {
71     int tmp = array[i];
72     array[i] = array[j];
73     array[j] = tmp;
74 }
75 }
76

```

Kod źródłowy 6. Klasa QuickSortTask.

Klasa **QuickSortTask** jest to klasa, której obiekty reprezentują zadania asynchroniczne. Implementuje ona interfejs o nazwie Task pochodzący z biblioteki użytkownika (metoda *call*). W klasie tej został zaimplementowany algorytm Quicksort. Dla tablicy zostaje wybrany element rozdzielający (linia 28.), a następnie tablica dzielona jest na dwa fragmenty. W początkowym zostaną umieszczone wszystkie elementy nie większe od rozdzielającego, a w końcowym wszystkie większe od niego. Następnie tworzone są kolejne zadania asynchroniczne, których celem jest sortowanie osobno wspomnianych fragmentów tablicy. Dla tablic (fragmentów), których wielkość jest nie większa od 100, zostaje uruchomiony algorytm **Sortowania przez wstawianie** (w celu optymalizacji). W linii 24. i 31. zwracany jest obiekt typu QuickSortTaskResult zawierający fragment tablicy, który został posortowany. Metoda

partition realizuje wybór elementu rozdzielającego, w *insertionSort* realizowane jest sortowanie przez wstawianie, a metoda *swap* to metoda pomocnicza odpowiedzialna za zamianę elementów.

```
1 package quicksort;
2
3 import java.io.Serializable;
4
5 public class QuickSortTaskResult implements Serializable {
6     private int[] array;
7     private int begin;
8     private int end;
9
10    public QuickSortTaskResult(int[] array, int begin, int end) {
11        this.array = array;
12        this.begin = begin;
13        this.end = end;
14    }
15
16    public int[] getArray() {
17        return array;
18    }
19
20    public int getBegin() {
21        return begin;
22    }
23
24    public int getEnd() {
25        return end;
26    }
27 }
28
```

Kod źródłowy 7. Klasa QuickSortTaskResult.

Obiekt klasy **QuickSortTaskResult** - jak zostało wspomniane - reprezentuje wynik realizacji pojedynczego zadania asynchronicznego. Zawiera ono fragment tablicy, która została posortowana.

```

1  package quicksort;
2
3  import pl.edu.uj.jnode.context.InjectContext;
4  import pl.edu.uj.jnode.userlib.Callback;
5
6  import java.io.Serializable;
7
8  public class QuickSortCallback implements Callback {
9      @InjectContext
10     private QuickSortResultContext resultContext;
11
12     @Override
13     public void onSuccess(Serializable taskResult) {
14         QuickSortTaskResult result = (QuickSortTaskResult) taskResult;
15         resultContext.copyResult(result.getResult(), result.getBegin(), result.getEnd());
16         resultContext.printResultIfSorted();
17     }
18
19     @Override
20     public void onFailure(Throwable ex) {
21         System.out.println(ex.getMessage());
22     }
23 }
24

```

Kod źródłowy 8. Klasa QuickSortCallback.

Obiekty klasy **QuickSortCallback** to - jak sama nazwa klasy wskazuje – wywołania zwrotne powiązane z zadaniami asynchronicznymi. Klasa implementuje interfejs `Callback` z udostępnionej biblioteki użytkownika (metody `onSuccess` oraz `onFailure`). W przypadku niepowodzenia zostanie wyświetlony komunikat o błędzie, natomiast w przypadku sukcesu - posortowany fragment tablicy z obiektu *result* (*QuickSortTaskResult*) zostaje przekopiowany do obiektu *resultContext*. Jest to pole opatrzone adnotacją `@InjectContext` z biblioteki użytkownika, dzięki której pole to zostało automatycznie wstrzyknięte przez kontener `jNode`. Jeżeli tablica jest już posortowana, to jej zawartość zostanie wypisana.

```

1 package quicksort;
2
3 import pl.edu.uj.jnode.context.Context;
4
5 import java.util.Arrays;
6
7 @Context
8 public class QuickSortResultContext {
9     private int[] array;
10    private boolean isSorted = false;
11    private boolean isPrinted = false;
12
13    public void copyResult(int[] array, int begin, int end) {
14        if (this.array == null) {
15            this.array = new int[array.length];
16            System.arraycopy(array, 0, this.array, 0, array.length);
17        } else {
18            System.arraycopy(array, begin, this.array, begin, end - begin + 1);
19        }
20    }
21
22    private boolean isSorted() {
23        if (isSorted) {
24            return true;
25        }
26        for (int i = 1; i < array.length; i++) {
27            if (array[i - 1] > array[i]) {
28                return false;
29            }
30        }
31        return isSorted = true;
32    }
33
34    public void printResultIfSorted() {
35        if (isSorted() && !isPrinted) {
36            System.out.println("Sorted array: " + Arrays.toString(array));
37            isPrinted = true;
38        }
39    }
40 }

```

Kod źródłowy 9. Klasa QuickSortResultContext.

Pozostała klasa o nazwie **QuickSortResultContext**. Klasa ta została opatrzona adnotacją `@Context`, która znajduje się w bibliotece użytkownika. Zatem obiekt tej klasy to singleton, który jest automatycznie wstrzykiwany jako zależność przez kontener `jNode` do obiektów typu `QuickSortCallback`. Jest on współdzielony przez wszystkie obiekty, które posiadają tę zależność. Obiekt ten wykorzystywany jest do łączenia wyników poszczególnych zadań asynchronicznych (posortowanych fragmentów tablicy).

[Alan Hawrot]

Program: Password Cracker

Program służy do łamania haseł zadanych przez użytkownika metodą brute force [77]. Zadane hasło jest przetwarzane przez funkcję mieszającą algorytmem md5 [78]. Następnie wszystkie możliwe hasła są mieszane tym samym algorytmem, aż uzyskamy hasło mające taki sam hasz. Hasła są tworzone na podstawie zadanego alfabetu, począwszy od haseł długości jeden. Gdy wszystkie kombinacje znaków dla haseł danej długości zostały sprawdzone, zostaje zwiększona długość hasła i procedura jest powtórzona.

Aby zrealizować pomysł w sposób rozproszony należało podzielić zbiór możliwych haseł na zadania. Najprościej byłoby odwzorowując jedno hasło w jedno zadanie, natomiast operacja sprawdzenia jednego hasła jest na tyle krótka, że rozwiązanie to byłoby skrajnie nieefektywne. Zdecydowałem się na umożliwienie sterowania wielkością podzbioru haseł do sprawdzenia przez jedno zdanie. Odpowiedzialność za wydzielanie podzbiorów haseł oraz dostarczanie kolejnych haseł zawarłem w klasie PasswordGenerator. Każdy podzbiór haseł jest również reprezentowany przez instancję klasy PasswordGenerator. Realizacja tej klasy jest oparta o reprezentację kolejnego hasła do wygenerowania poprzez tablicę iterationPointers, przechowującą indeksy znaków (zbiór możliwych znaków trzymamy w poindeksowanej kolekcji). Po pobraniu hasła, wykonywana jest procedura „inkrementacji” tablicy, ustawiająca wskaźniki tak, aby wskazywały kolejne słowo. W tym przypadku parametrem procedury jest indeks ostatniego elementu tablicy.

```
86 private void incrementIterationPointers(int startingPosition) {
87
88     for (int i = startingPosition; i >= 0; i--) {
89         if (iterationPointers[i] + 1 < charSet.length()) {
90             ++iterationPointers[i];
91             return;
92         } else {
93             iterationPointers[i] = 0;
94         }
95     }
96     iterationPointers = addAll(new byte[1], iterationPointers);
97 }
```

Kod źródłowy 10. Operacja inkrementacji tablicy reprezentującej kolejne słowa.

Operacja ta jest również wykorzystywana w celu odseparowania podzbioru haseł. Kontrola nad wielkością podzbioru jest zrealizowana za pomocą parametru jobsSeparationFactor podawanego na początku wykonywania programu. Operacja jest

wywoływana z parametrem `startingPosition` równym `"iterationPointers.length - 1 - jobsSeparationFactor"`, pomijając kolejne słowa w ilości równej ilości znaków w zadanym alfabecie podniesioną do potęgi `jobsSeparationFactor`. Z wyodrębnionych w ten sposób słów tworzony jest `PasswordGenerator`.

Funkcja `main` klasy `Main` jest pierwszym wykonywanym zadaniem w systemie. Jest odpowiedzialna za pobranie od użytkownika hasła oraz parametru `jobsSeparationFactor`. Następnie inicjalizuje `PasswordCrackerContext`, klasę przechowującą dane programu, czyli hasz hasła i główną instancję `PasswordCrackerGenerator`. Tłumaczenie wykorzystanych adnotacji zostało pominięte, ponieważ wszystkie adnotacje zostały omówione w poprzednim przykładzie.

```
1 package pl.edu.uj.passwordcracker;
2 import pl.edu.uj.jnode.context.ContextScan;
3 import pl.edu.uj.jnode.userlib.*;
4 import java.util.Scanner;
5 import static org.apache.commons.codec.digest.DigestUtils.md5;
6
7 @ContextScan("pl.edu.uj.passwordcracker")
8 public class Main {
9     private static final String CHARSET1 = "abcdefghijklmnopqrstuvwxyz";
10    private static int jobsSeparationFactor;
11
12    public static void main(String[] args) {
13        String line;
14        try (Scanner scanner = new Scanner(System.in)) {
15            System.out.println("Provide password you wish to hack");
16            line = scanner.nextLine();
17            System.out.println(line.length());
18            if (jobsSeparationFactor <= 0) {
19                System.out.println("Provide jobs separation factor");
20                jobsSeparationFactor = scanner.nextInt();
21            }
22        }
23        TaskExecutor taskExecutor = TaskExecutorFactory.createTaskExecutor();
24        Class<?> cls = PasswordCrackerContext.class;
25        PasswordCrackerContext passwordCrackerContext = (PasswordCrackerContext) taskExecutor.getBean(cls);
26        byte[] encryptedPassword = md5(line.getBytes());
27        passwordCrackerContext.setEncryptedPassword(encryptedPassword);
28        PasswordGenerator passwordGenerator = new PasswordGenerator(CHARSET1, jobsSeparationFactor);
29        passwordCrackerContext.setPasswordGenerator(passwordGenerator);
30        PasswordCrackerCallback callback = new PasswordCrackerCallback(taskExecutor);
31        taskExecutor.doAsync(() -> null, callback);
32    }
33 }
34
```

Kod źródłowy 11. Klasa Main.

Finalnym krokiem funkcji `main` jest wykonanie pustego zadania wraz z instancją `PasswordCrackerCallback`. Jest tak, ponieważ logikę odpowiedzialną za tworzenie nowych zadań umieściłem w klasie `PasswordCrackerCallback` i nie chciałem jej duplikować w funkcji `main`. Klasa ta tworzy zadania sprawdzające hasło na podstawie ilości wolnych zasobów

obliczeniowych w całym klastrze. Przyjąłem, że zadanie zwróci poprawne hasło lub pusty ciąg znaków / wartość null, gdy nie udało się go znaleźć.

```
1 package pl.edu.uj.passwordcracker;
2 import pl.edu.uj.jnode.context.InjectContext;
3 import pl.edu.uj.jnode.userlib.*;
4 import java.io.Serializable;
5
6 public class PasswordCrackerCallback implements Callback {
7     private final TaskExecutor taskExecutor;
8     @InjectContext
9     private PasswordCrackerContext passwordCrackerContext;
10    private boolean foundPassword = false;
11
12    public PasswordCrackerCallback(TaskExecutor taskExecutor) { this.taskExecutor = taskExecutor; }
13
14    @Override
15    public void onSuccess(Serializable taskResult) {
16        if (foundPassword) {
17            return;
18        } else if (taskResult != null && !taskResult.equals("")) {
19            foundPassword = true;
20            System.out.println("PasswordCrackerCallback: Found password: " + taskResult);
21        } else {
22            long availableWorkers = taskExecutor.getAvailableWorkers();
23            PasswordGenerator passwordGenerator = passwordCrackerContext.getPasswordGenerator();
24            System.out.println("PasswordCrackerCallback: Scheduling " + (availableWorkers + 1) + " tasks");
25
26            byte[] encryptedPassword = passwordCrackerContext.getEncryptedPassword();
27
28            for (int i = 0; i < availableWorkers + 1; i++) {
29                PasswordGenerator separatedJobsGenerator = passwordGenerator.separateJobSet();
30                PasswordCrackerTask task = new PasswordCrackerTask(separatedJobsGenerator, encryptedPassword);
31                taskExecutor.doAsync(task, this);
32            }
33        }
34    }
35
36    @Override
37    public void onFailure(Throwable ex) { ex.printStackTrace(); }
38
39 }
```

Kod źródłowy 12. Klasa PasswordCrackerCallback.

Ostatnim elementem programu jest klasa PasswordCrackerTask. Tworzone instancje przez PasswordCrackerCallback zajmują się sprawdzeniem udostępnionego podzbioru haseł.

```

1  package pl.edu.uj.passwordcracker;
2  import org.apache.commons.codec.digest.DigestUtils;
3  import pl.edu.uj.jnode.userlib.Task;
4  import java.io.Serializable;
5  import java.util.Arrays;
6
7  public class PasswordCrackerTask implements Task {
8      private PasswordGenerator passwordGenerator;
9      private byte[] encryptedPassword;
10
11     public PasswordCrackerTask(PasswordGenerator passwordGenerator, byte[] encryptedPassword) {
12         this.passwordGenerator = passwordGenerator;
13         this.encryptedPassword = encryptedPassword;
14     }
15
16     @Override
17     public Serializable call() throws Exception {
18         int i = 0;
19         int taskId = System.identityHashCode(this) % 101;
20         while (passwordGenerator.hasNext()) {
21             String candidateForPassword = passwordGenerator.next();
22             if (++i == 1) {
23                 System.out.println(taskId + " starting cracking since " + candidateForPassword);
24             }
25             byte[] digest = DigestUtils.md5(candidateForPassword.getBytes());
26             if (Arrays.equals(digest, encryptedPassword)) {
27                 System.out.println("Found password: " + candidateForPassword);
28                 return candidateForPassword;
29             }
30         }
31         System.out.println(taskId + " finishing after " + i + " tries.");
32         return "";
33     }
34 }
35

```

Kod źródłowy 13. Klasa PasswordCrackerTask.

[Michał Semik]

Przetestowane scenariusze

Framework jNode i uruchomione na nim, poprzednio opisane programy zostały przetestowane w różnych konfiguracjach w celu sprawdzenia poprawności działania systemu i jego odporności na błędy.

Lista przetestowanych konfiguracji (liczba komputerów w sieci; liczba uruchomionych instancji na jednym komputerze; wielkość puli jednej instancji):

1. 1 komputer; 1 instancja; wielkość puli równa 1.
2. 1 komputer; 1 instancja; wielkość puli równa 4.
3. 1 komputer; 3 instancje; wielkość puli równa 1.
4. 1 komputer; 3 instancje; wielkość puli równa 2.
5. 2 komputery; 1 instancja; wielkość puli równa 1.
6. 2 komputery; 1 instancja; wielkość puli równa 4.
7. 2 komputery; 2 instancje; wielkość puli równa 1.

8. 2 komputery; 2 instancje; wielkość puli równa 2.

Wykorzystane komputery posiadały niejednorodne konfiguracje zarówno sprzętowe, jak i pod względem zainstalowanego oprogramowania (systemy operacyjne: Windows 10, Ubuntu 14.04 LTS, OS X 10.11 El Capitan).

Do systemu uruchomionego w wyżej przedstawionych konfiguracjach wprowadzono najpierw jeden program do wykonania, a następnie przetestowano wykonanie obu programów jednocześnie. We wszystkich przypadkach programy zakończyły się powodzeniem.

Ponadto w systemie zostały zasymulowane różnego rodzaju błędy i awarie:

- Awaryjne węzłów na różnych etapach wykonywania programu.
- Przerwanie połączenia pomiędzy węzłami.
- Usunięcie/anulowanie aplikacji na różnych węzłach i na różnych etapach wykonywania programu.

jNode prawidłowo reagował na zasymulowane błędy i awarie. Tak jak zostało to opisane w poprzednich rozdziałach system - w zależności od scenariusza ponawiał wykonywanie, rozsyłanie, czy też anulowanie zadań i zamykanie aplikacji.

[Alan Hawrot]

Możliwości rozwoju jNode

Podstawową własnością frameworków jest ułatwianie użytkownikowi programowania w jak największym stopniu. Prawdopodobnie jest to również jeden z lepszych kierunków rozwoju jNode. Dostarczenie funkcjonalności takich jak rozproszony cache lub inne formy udostępniania współdzielonej pamięci pomiędzy węzłami, możliwość kopiowania beanów pomiędzy węzłami zamiast inicjalizacji nowych na węzłach pomocniczych, można wykorzystać w naszych przykładowych aplikacjach i z pewnością w wielu innych. Rozwój kontenera wstrzykiwania zależności również należy do priorytetów. Integracja ze Spring Framework poprzez wspieranie wstrzykiwania zależności z kontenera frameworku Spring w taskach i callbackach przyniosłaby duży zysk dla użytkownika. Można wprowadzić wsparcie dla metody dziel i zwyciężaj, przykładowo udostępnić możliwość oczekiwania na zakończenie

zadania wykorzystując mechanizm Future [79] lub udostępnić metodę fork znaną z systemów UNIX [80]. Jednak podejście to wprowadzałoby elementy programowania synchronicznego.

Drugim aspektem, którym chcemy się zająć jest administracja systemu. Zaplanowany jest już napomknięty system monitorujący. Możliwość graficznej reprezentacji modelu platformy, czyli węzłów, wykonywanych na nich aplikacji oraz wykonywanych w aplikacjach zadań. Przedstawienie obciążenia procesora poszczególnych węzłów, ilości wykonywanych zadań na węzłach, wolnych zasobów w silniku i wielkość kolejki wywołań zwrotnych byłaby dobrym źródłem informacji analitycznych na temat stanu klastra. Umożliwiłaby wygodną weryfikację wydajności i funkcjonowania wszystkich elementów systemu. Możliwość wysyłania i usuwania aplikacji z interfejsu graficznego również byłaby przydatna.

Kolejnym przydatnym krokiem mogłyby okazać się testy wydajnościowe i związane z nimi optymalizacje systemu. Przykładowo natychmiastowe propagowanie archiwów do najbliższych (w sensie mechanizmu wyboru węzła) węzłów w celu wcześniejszego rozpoczęcia wykonywania zadań na węzłach pomocniczych. Obliczanie odległości pomiędzy węzłami na podstawie czasu transmisji wiadomości zamiast kolejności w liście mogłoby mieć pozytywne skutki, gdy węzły byłyby bardziej odległe. Wykorzystanie czasu poprzednio wykonywanych zadań w ramach jednej aplikacji mogłoby pozwolić na ocenianie czy warto delegować kolejne zadania. Duża liczba krótkich zadań może okazać się wydajniejsza do wykonania na jednym węźle.

Innym, ciekawym kierunkiem jest stworzenie warstwy składającej się z komponentów wykorzystujących platformę do realizacji asynchronicznych operacji, jednocześnie enkapsulując dostęp do silnika platformy. W takim przypadku użytkownik korzystałby z wyższej warstwy i przekazywał do niej jedynie wywołania zwrotne do późniejszego wykonania. Komponenty odpowiadałyby za implementację zadań. Dzięki takiemu podejściu użytkownik warstwy wyższej może korzystać z zalet szybkich asynchronicznych operacji, bez konieczności ich rozumienia. Przykładowo można udostępnić komponenty: serwera http, klienta http i dostępu do danych. Komponent serwera http byłby odpowiedzialny za wykonywanie callbacków użytkownika przy nadejściu żądania http. Komponent klienta http pozwalałby na wysyłanie żądań, otrzymując callback po uzyskaniu odpowiedzi. Podobnie komponent dostępu do danych odpowiadałby za wysyłanie zapytań bazodanowych i po

otrzymaniu wyniku wykonywał wywołanie zwrotne. Dzięki takim trzem bazowym komponentom można pisać aplikacje w stylu podobnym do Node.js.

[Michał Semik]

Podsumowanie

Podsumowując można powiedzieć, że praca zakończyła się sukcesem. Udało się opracować framework przeznaczony dla języka programowania Java, który umożliwia użytkownikom łatwiejsze tworzenie i wykonywanie aplikacji w oparciu o model asynchroniczny. Dzięki temu powstałe przy użyciu programy są wydajniejsze od synchronicznych odpowiedników. Dodatkowo ze względu na bardzo dobrą skalowalność wertykalną i horyzontalną, jNode i powstałe przy jego użyciu aplikacje lepiej wykorzystują dostępną moc obliczeniową komputerów. Stworzony system jest w pełni zautomatyzowany i wygodny w obsłudze. Zaawansowana architektura, modularyzacja i osiągnięta wysoka elastyczność umożliwiają dostosowanie jNode do własnych potrzeb. Potencjał systemu jest wysoki, a jego możliwości rozwoju szerokie. Framework jNode nadal może być rozwijany o dodanie nowych funkcjonalności i doskonalenie istniejących mechanizmów.

[Alan Hawrot]

Bibliografia

1. Fowler, M.: Inversion Of Control. 14.05.2016.
<http://martinfowler.com/bliki/InversionOfControl.html>
2. Inversion of control - Wikipedia. 14.05.2016.
https://en.wikipedia.org/wiki/Inversion_of_control
3. Callback - Wikipedia. 07.04.2016.
https://en.wikipedia.org/wiki/Callback_%28computer_programming%29
4. Fowler, M. 14.05.2016.
<http://www.martinfowler.com/articles/injection.html>
5. Dependency Injection - Wikipedia. 14.05.2016.
https://pl.wikipedia.org/wiki/Wstrzykiwanie_zale%C5%BCno%C5%9Bci

6. Class path - Java documentation. 12.05.2016.
<http://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>
7. ClassLoader - Java documentation. 14.05.2016.
<https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html>
8. Class path scanning. 14.05.2016.
<https://github.com/lukehutch/fast-classpath-scanner>
9. Wilczak, D.: Programowanie współbieżne 2012/2013, Wprowadzenie.
10. Race condition - Wikipedia. 19.04.2016.
https://en.wikipedia.org/wiki/Race_condition#Software
11. Brown University Department of Computer Science: Introduction to Asynchronous Programming. 12.05.2016.
<http://cs.brown.edu/courses/cs168/s12/handouts/async.pdf>
12. System Rozproszony - Wikipedia. 18.05.2016.
https://pl.wikipedia.org/wiki/System_rozproszony
13. Klaster komputerowy - Wikipedia. 13.05.2016.
https://pl.wikipedia.org/wiki/Klaster_komputerowy
14. Pojęcie "modelu obliczeniowego". 19.03.2016.
<https://edux.pjwstk.edu.pl/mat/198/lec/main10.html>
15. Kalita, P.: Przedmiot Programowanie Rozproszone 2013/2014, wykład pierwszy.
16. Software framework - Wikipedia. 16.05.2016.
https://en.wikipedia.org/wiki/Software_framework
17. Open Source - Wikipedia. 19.05.2016.
https://en.wikipedia.org/wiki/Open-source_software
18. Mechanizm refleksji - Wikipedia. 21.05.2016.
https://pl.wikipedia.org/wiki/Mechanizm_refleksji
19. Mechanizm refleksji - Wikipedia. 21.05.2016.
https://en.wikipedia.org/wiki/Reflection_%28computer_programming%29
20. Refleksja, tutorial - Oracle. 21.05.2016.
<https://docs.oracle.com/javase/tutorial/reflect/>
21. Serialization - Wikipedia. 20.05.2016.
<https://en.wikipedia.org/wiki/Serialization>

22. Scalability - Wikipedia. 20.05.2016.
<https://en.wikipedia.org/wiki/Scalability>
23. Node.js - about thread pool. 15.03.2016.
<http://stackoverflow.com/questions/22644328/when-is-the-thread-pool-used>
24. Node.js - lack of shared memory. 15.03.2016.
<http://stackoverflow.com/questions/10965201/in-node-js-how-to-declare-a-shared-variable-that-can-be-initialized-by-master-p>
25. Node.js - Wikipedia. 15.03.2016.
<https://en.wikipedia.org/wiki/Node.js>
26. About Node.js. 15.03.2016.
<https://nodejs.org/en/about/>
27. Node.js - vertical scalling. 15.03.2016.
<http://blog.doselect.com/post/132069589293/vertically-scaling-a-nodejs-component>
28. Node.js - API. 15.03.2016.
<https://nodejs.org/api/>
29. Apache Hadoop - Wikipedia. 15.06.2016.
https://pl.wikipedia.org/wiki/Apache_Hadoop
30. Master/slave - Wikipedia. 20.05.2016.
https://en.wikipedia.org/wiki/Master/slave_%28technology%29
31. Single point of failure - Wikipedia. 15.06.2016.
https://en.wikipedia.org/wiki/Single_point_of_failure
32. Heartbeat - Wikipedia. 19.03.2016.
https://en.wikipedia.org/wiki/Heartbeat_%28computing%29
33. Remote Procedure Call - Wikipedia. 20.05.2016.
https://en.wikipedia.org/wiki/Remote_procedure_call
34. Apache Hadoop - HDFS Architecture. 15.06.2016.
https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
35. System plików - Wikipedia. 15.06.2016.
https://pl.wikipedia.org/wiki/System_plik%C3%B3w
36. MapReduce - Wikipedia. 15.06.2016.
<https://en.wikipedia.org/wiki/MapReduce>

37. Apache Hadoop MapReduce - code moves near data for computation - StackOverflow. 15.06.2016.
<http://stackoverflow.com/questions/11602699/hadoop-code-moves-near-data-for-computation>
38. Apache Hadoop MapReduce can be run over other filesystems - StackOverflow. 15.06.2016.
<http://stackoverflow.com/questions/8921376/can-hadoop-mapreduce-can-run-over-other-fileystems>
39. Job Tracker - Apache Hadoop. 15.06.2016.
<https://wiki.apache.org/hadoop/JobTracker>
40. Task Tracker - Apache Hadoop. 15.06.2016.
<https://wiki.apache.org/hadoop/TaskTracker>
41. Apache Hadoop introduction - Tutorialspoint. 15.06.2016.
http://www.tutorialspoint.com/hadoop/hadoop_introduction.htm
42. Apache Hadoop MapReduce tutorial. 15.06.2016.
https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
43. How Hadoop MapReduce works. 15.06.2016.
<https://dzone.com/articles/how-hadoop-mapreduce-works>
44. Java Archive - Wikipedia. 20.05.2016.
https://pl.wikipedia.org/wiki/JAR_%28format_pliku%29
45. Reactor - design pattern - Wikipedia.
https://en.wikipedia.org/wiki/Reactor_pattern
46. Brian Göetz, T.: Java Concurrency in Practice. 2004.
47. Load Balancing - Wikipedia. 14.05.2016.
https://pl.wikipedia.org/wiki/R%C3%B3wnowa%C5%BCenie_obci%C4%85%C5%BCenia
48. Big Data - Wikipedia. 16.05.2016.
https://pl.wikipedia.org/wiki/Big_data
49. Architektura Klient-serwer - Wikipedia. 15.03.2016.
<https://pl.wikipedia.org/wiki/Klient-serwer>

50. Big Data Overview - Tutorialspoint. 13.05.2016.
http://www.tutorialspoint.com/hadoop/hadoop_big_data_overview.htm
51. The IoC container - Spring Framework Reference. 14.05.2016.
<http://docs.spring.io/autorepo/docs/spring/current/spring-framework-reference/html/beans.html>
52. Application Events - Spring Framework. 14.05.2016.
<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#context-functionality-events>
53. Dependency Inversion - Wikipedia. 14.05.2016.
https://en.wikipedia.org/wiki/Dependency_inversion_principle
54. Spring Framework - Wikipedia. 14.05.2016.
https://pl.wikipedia.org/wiki/Spring_Framework
55. Optional and excluded dependencies. 10.05.2016.
<https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html>
56. Component based programming - Wikipedia. 10.05.2016.
https://en.wikipedia.org/wiki/Component-based_software_engineering
57. Java Module System - Wikipedia. 10.05.2016.
https://en.wikipedia.org/wiki/Java_Module_System
58. Loose coupling - Wikipedia. 12.05.2016.
https://en.wikipedia.org/wiki/Loose_coupling
59. Walter, B.: Wykład: Programowanie aspektowe. 12.05.2016.
<http://wazniak.mimuw.edu.pl/images/e/ea/Zpo-12-wyk.pdf>
60. Aspect Weaver - Wikipedia. 12.05.2016.
https://en.wikipedia.org/wiki/Aspect_weaver
61. JGroups. 13.05.2016.
<http://jgroups.org/>
62. Non-blocking I/O (Java) - Wikipedia. 13.05.2016.
[https://en.wikipedia.org/wiki/Non-blocking_I/O_\(Java\)](https://en.wikipedia.org/wiki/Non-blocking_I/O_(Java))
63. WatchService - Java documentation. 20.05.2016.
<https://docs.oracle.com/javase/8/docs/api/java/nio/file/WatchService.html>

64. java.util.concurrent.atomic - Java documentation. 22.05.2016.
<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html>
65. java.util.concurrent - Java documentation. 22.05.2016.
<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html>
66. Wikipedia - Reusability. 18.04.2016.
<https://en.wikipedia.org/wiki/Reusability>
67. Cross-cutting concern - Wikipedia. 20.04.2016.
https://en.wikipedia.org/wiki/Cross-cutting_concern
68. Odwiedzający - Wzorzec projektowy - Wikipedia. 18.04.2016.
<https://pl.wikipedia.org/wiki/Odwiedzaj%C4%85cy>
69. Decorator Pattern - Wikipedia. 19.04.2016.
https://en.wikipedia.org/wiki/Decorator_pattern
70. java.util.UUID - Java documentation. 18.05.2016.
<https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html>
71. Random UUID - Probability of duplicates - Wikipedia. 19.04.2016.
https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates
72. State design pattern - Wikipedia. 24.05.2016.
https://en.wikipedia.org/wiki/State_pattern
73. Finite State Machines and The State Pattern. 24.05.2016.
<https://cleancoders.com/episode/clean-code-episode-28/show>
74. Chain of responsibility design pattern. 14.04.2016.
https://sourcemaking.com/design_patterns/chain_of_responsibility
75. Type Safety - Wikipedia. 23.05.2016.
https://en.wikipedia.org/wiki/Type_safety
76. JGroups - FAQ. 13.05.2016.
<https://github.com/belaban/JGroups/wiki/FAQ>
77. Brute force attack - Wikipedia. 16.06.2016.
https://en.wikipedia.org/wiki/Brute-force_attack

78. MD5 - Wikipedia. 24.06.2016.

<https://pl.wikipedia.org/wiki/MD5>

79. Future - Java documentation. 10.05.2016.

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html>

80. Fork - Wikipedia. 24.04.2016.

https://pl.wikipedia.org/wiki/Fork#Fork_w_programowaniu