Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
AGH
Unia Europejska Europejski Fundusz Społeczny

Zintegrowany Program Rozwoju
Akademii Górniczo-Hutniczej w Krakowie
Nr umowy: **POWR.03.05.00-00-Z307/17**

Instrukcja do ćwiczeń laboratoryjnych

Nazwa przedmiotu	Systemy rozproszone
Numer ćwiczenia	6
Temat ćwiczenia	Systemy reaktywne - Akka

Poziom studiów	I stopień
Kierunek	Informatyka
Forma i tryb studiów	Stacjonarne
Semestr	6

dr inż. Filip Malawski

Wydział Informatyki, Elektroniki i Telekomunikacji

Kraków, 2020

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z zagadnieniami związanymi z tworzeniem systemów rozproszonych w podejściu reaktywnym. Zagadnienia prezentowane są na przykładzie platformy Akka.

Wynikiem ćwiczenia mają być **kody źródłowe** do zadań (w języku Java) oraz **raport** w pliku **tekstowym lub PDF** zawierający: **imię, nazwisko, numer indeksu** oraz **odpowiedzi do zadań** (w opisie zadań podano co ma się znaleźć w raporcie).

Rozwiązanie ćwiczenia należy przesłać w formie pojedynczego archiwum ZIP o nazwie w formacie: ***Nazwisko_Imie_akka_lab***, np. ***Mickiewicz_Adam_akka_lab***.

Ważne:
- należy wysłać **jeden plik** – archiwum ZIP
- nazwa archiwum ma zaczynać się od **nazwiska**
- w archiwum mają się znaleźć **tylko** kody źródłowe i raport w formacie tekstowym lub PDF (proszę nie wysyłać bibliotek itp.)
- w archiwum nie powinno być **żadnych katalogów**, poza katalogiem nadrzędnym

UWAGA: Niezastosowanie się do powyższych wytycznych powoduje automatyczne uzyskanie 0 punktów za ćwiczenie.

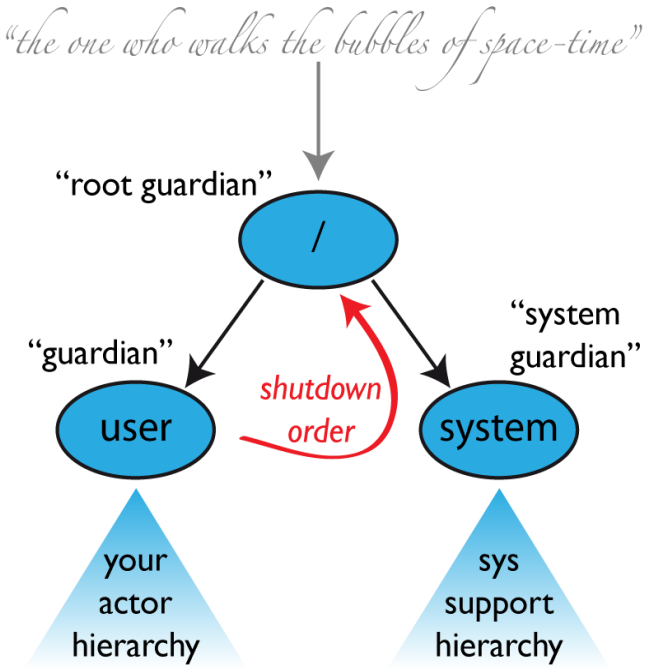
2.Wprowadzenie do ćwiczenia

W podejściu reaktywnym zamiast próbować stworzyć system, w którym błędy się nie pojawią, zakładamy, że jakieś błędy zawsze wystąpią i stosujemy odpowiednie mechanizmy obsługi błędów, aby zminimalizować ich negatywny wpływ na działanie systemu. Koncepcja ta została przedstawiona w dokumencie **Reactive Manifesto**: https://www.reactivemanifesto.org/

Najważniejsze cechy systemów reaktywnych to:
- Responsywność – zapewnione są niskie czasy odpowiedzi, nawet gdy pojawią się błędy
- Żywotność – system pozostaje responsywny, nawet w przypadku awarii
- Elastyczność – system jest odporny na zmianę obciążenia
- Komunikacja przez wiadomości – zapewnia luźne powiązania i izolację komponentów oraz transparentność ich lokalizacji

Akka jest platformą umożliwiającą łatwe tworzenie aplikacji w podejściu reaktywnym. Najważniejsze cechy platformy:
- Aktorzy – są to lekkie wątki, komunikujące się przez wiadomości
- Wysokopoziomowa obsługa błędów – różne dostępne strategie
- Transparentność lokalizacji – aktorzy mogą być lokalni lub zdalni, jednak komunikacja z nimi jest analogiczna
- Skalowalność – przez zrównoleglenie (*Scale up*) oraz rozproszenie (*Scale out*)

Aktorzy ułożeni są w hierarchię:
- Hierarchia ma strukturę drzewa
- Aktor może tworzyć nowych aktorów, którymi zarządza
- Każdy aktor ma dokładnie jednego rodzica (*Supervisor*)
- Supervisor decyduje o tym jak zostaną obsłużone błędy u jego podwładnych
- Przy starcie tworzeni są aktorzy: *Root, User, System*

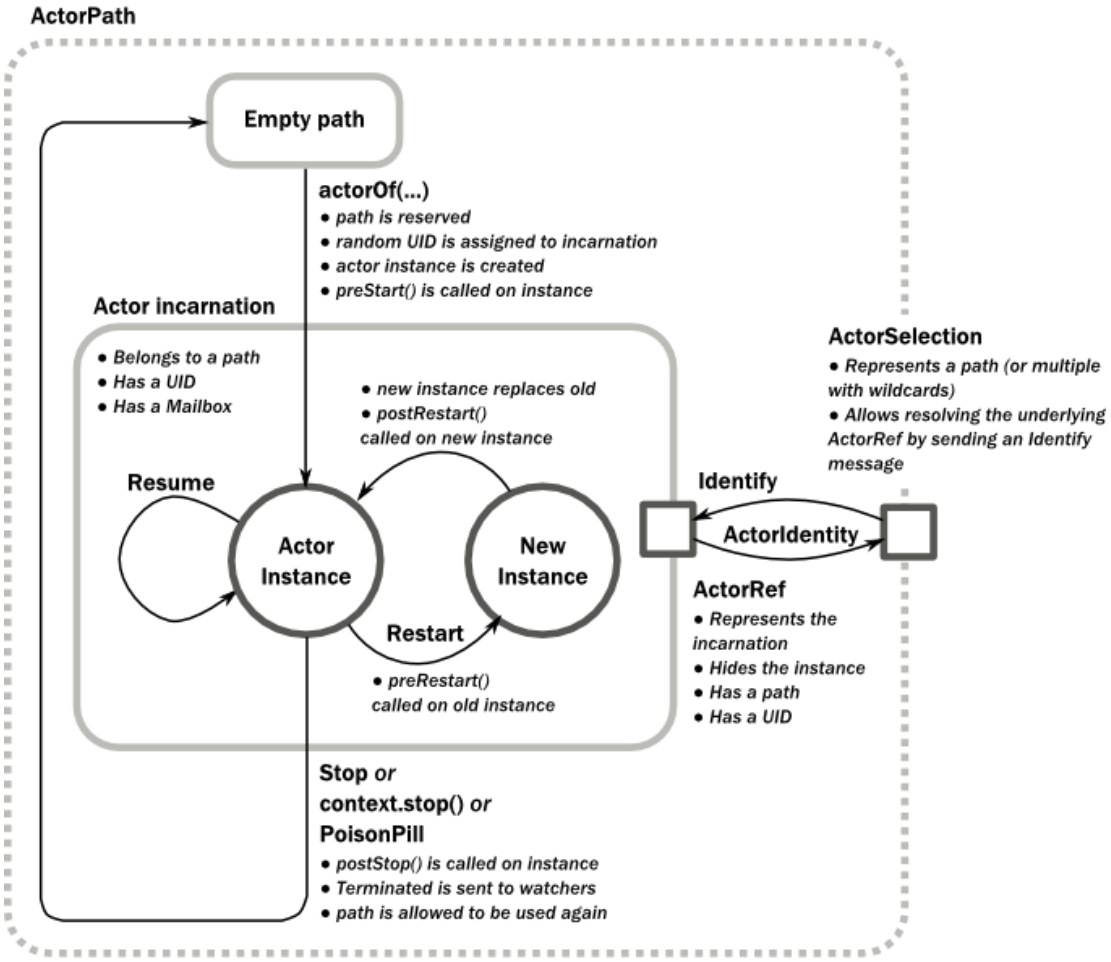


Obsługa błędów:

- Jeśli aktor zgłosi błąd (wyjątek), wtedy jego Supervisor postępuje zgodnie z jedną ze strategii:
  - *Resume* – wznawia działanie podwładnego, zachowując jego stan
  - *Restart* – restartuje podwładnego, kasując jego stan
  - *Stop* – zatrzymuje działanie podwładnego
  - *Escalate* – zgłasza wyjątek do swojego Supervisora
- Jeśli *Root* zgłosi wyjątek, system jest zatrzymywany
- Strategia obsługi wyjątku u aktora wpływa też na jego wszystkich podwładnych
- Strategia może być:
  - *OneForOne* – uruchamiana tylko dla podwładnego, który zgłosił błąd
  - *AllForOne* – uruchamiana dla wszystkich podwładnych (mimo, że błąd zgłosił tylko jeden)

Aktor dostępny jest przez referencję *ActorRef*, nie ma do niego dostępu bezpośrednio. Pozwala to uzyskać transparentność lokalizacji (może się on znajdować w różnych lokalizacjach - lokalnie lub zdalnie).

Cykl życia aktora rozróżnia pojęcia instancji oraz inkarnacji. Aktor znajduje się w drzewie, a ścieżka w drzewie wskazuje na jego inkarnację. Jeśli wznowimy aktora po błędzie, to jest to ta sama inkarnacja oraz ta sama instancja. Jeśli zrestartujemy aktora to jest to ta sama inkarnacja, ale nowa instancja. W ten sposób możemy zawsze komunikować się z tą samą inkarnacją aktora, która wewnętrznie może mieć podmieniane instancje, w przypadku błędów. Należy zwrócić uwagę, że ścieżki logiczne oraz fizyczne dla aktora mogą być różne, jeśli aktor jest zdalny.



3.Program ćwiczenia

Na platformie UPEL znajdują się materiały do ćwiczenia: kody źródłowe oraz wymagane biblioteki. Ćwiczenie należy wykonać w języku **Java**.

1. Stwórz projekt w Javie, umieść w nim załączone kody źródłowe oraz dołącz biblioteki.
   Zapoznaj się z kodami źródłowymi Z1.
   - Zwróć uwagę na sposób tworzenia aktorów i zwracany typ *ActorRef*
   - Zapoznaj się z metodami wysyłania oraz odbierania wiadomości
   - Zapoznaj się z hierarchią aktorów w przykładzie Z1 (MathActor oraz MultiplyWorker)
   - Uruchom przykład

2. **Zad 1 (2 punkty)**

   Zaimplementuj dodatkowego aktora, który będzie wykonywał zadanie dzielenia.
   - Stwórz klasę DivideWorker, analogiczną do MultiplyWorker
   - Zaimplementuj obsługę dzielenia w klasie DivideWorker
   - Dodaj obsługę polecenia dzielenia: d [liczba][liczba]
   - Dodaj tworzenie aktora DivideWorker w aktorze MathActor

Zapoznaj się z działaniem różnych strategii obsługi błędów.

- Zaobserwuj, że w przypadku wystąpienia błędu (np. dzielenie przez zero) aktor dalej kontynuuje działanie (wypisywanie wyjątku można wyłączyć w konfiguracji systemu aktorów)
- Dodaj zliczanie liczby wykonanych operacji u aktorów obsługujących mnożenie oraz dzielenie (u każdego z osobna)
- Dodaj wypisywanie informacji o liczbie wykonanych operacji przy zwracaniu wyniku działania, np.: `m 2 2` powinno wypisać: `4 (operation count: 1)`
- W aktorze MathActor zastosuj strategię, która:
  - wznawia (resume) podwładnego, jeśli rzucił wyjątek arytmetyczny
    np. po wpisaniu: `'d 2 0'`
  - restartuje podwładnego, jeśli rzucił inny wyjątek
    np. po wpisaniu: `'d aaa'`
- Porównaj działanie OneForOneStrategy oraz AllForOneStrategy
- **W raporcie należy umieścić**: output pokazujący różnice w działaniu resume/restart oraz OneForOneStrategy/AllForOneStrategy wraz z krótkim komentarzem na czym te różnice polegają

3. Zdalni aktorzy
   - Aktorzy dostępni są przez referencję ActorRef, co zapewnia transparentność lokalizacji
   - Komunikacja ze zdalnym systemem aktorów odbywa się analogicznie jak z lokalnym
   - Należy jednak pamiętać, że wiadomości wysyłane przez sieć mają większe ograniczenia
   - Wymagane uruchomienie modułu Remoting
   - Zdalny aktor znajdowany jest przez podanie ścieżki

4. Zapoznaj się kodami źródłowymi Z2
   - Aplikacja Z2_AppLocal to aplikacja lokalna
   - Aplikacja Z2_AppRemote symuluje aplikację zdalną (przez uruchomienie na innym porcie)
   - Każda aplikacja ma jednego aktora odpowiednio (Z2_LocalActor, Z2_RemoteActor)
   - Zwróć uwagę na wczytywanie konfiguracji Remoting z plików `local_app.conf` oraz `remote_app.conf`
   - Pliki konfiguracyjne należy umieścić w katalogu, w którym startuje aplikacja (w IntelliJ jest to główny katalog projektu)
   - Z2_AppLocal oraz Z2_AppRemote automatycznie uruchamiają moduł Remoting, co widać po wypisanym w konsoli komunikacie (brak komunikatu wskazuje na problem z uruchomieniem Remoting): `(…) Remoting started with transport [Artery tcp](…)`

5. **Zad 2 (1 punkt). Zaimplementuj komunikację pomiędzy zdalnymi aktorami**
   - Lokalny aktor ma wczytywać wiadomości przez konsolę, a następnie wysyłać je do zdalnego aktora, identyfikowanego po ścieżce
   - Zdalny aktor ma odebrać wiadomość, wypisać na swoją konsolę, zamienić na wielkie litery oraz odesłać do nadawcy
   - Lokalny aktor ma wypisać otrzymaną odpowiedź
   - Komunikacja odbywa się przez localhost
   - Wskazówki:
     - Wysłanie wiadomości po ścieżce:
       `getContext().actorSelection(path).tell(…)`
     - Sprawdzenie swojej ścieżki:
       `getSelf().path()`
     - Format ścieżki (należy wpisać poprawne wartości):
       `akka://systemName@127.0.0.1:2552/user/actorName`
   - **W raporcie należy umieścić:** output z Z2_AppLocal oraz Z2_AppRemote po przesłaniu przykładowej wiadomości (output ma zawierać również komunikaty z Remoting)

6. Koncepcja reaktywnych strumieni, przedstawiona w dokumencie http://www.reactive-streams.org/, zakłada realizację strumieniowego przesyłu danych tak, aby zapewnić:
    - Asynchroniczność
    - Sterowanie przepływem

7. Akka udostępnia dedykowane API dla strumieni reaktywnych (Akka Streams API). Budowanie strumieni polega na składaniu elementów różnego typu. Główne kategorie elementów to:
    - *Source* – źródło danych
    - *Flow* – transformacja danych
    - *Sink* – odbiór danych

    Z wykorzystaniem powyższych klas elementów można budować dowolne grafy przepływu. Każdy element może być wykorzystany wielokrotnie. Samo zdefiniowanie strumienia nie oznacza jednak jego stworzenia – elementy tworzone są dopiero w momencie uruchomienia strumieniowania.

8. Zapoznaj się z kodem źródłowym Z3:
    - Mamy 3 przykładowe strumienie
        - Pierwszy podwaja liczby z zakresu 1 do 10 i wypisuje wszystkie na konsolę
        - Drugi zamienia napisy z listy na wielkie litery
        - Trzeci podwaja liczby z zakresu -5 do -1 i wypisuje ostatnią na konsolę
    - Uruchom przykład
    - Zauważ, że strumienie uruchamiane są równolegle – wyniki w konsoli mogą być przemieszane

9. **Zad 3 (1 punkt). Zaobserwuj działanie asynchronicznego wykonania oraz różnych strategii back-pressure.**
    - Stwórz strumień który:
        - Jako źródło podaje liczby od 1 do 10
        - Następnie przeprowadza mapowanie `.map(…)`, które wykonuje `Thread.sleep(500)`, a potem zwraca wartość przemnożoną przez 10
        - Następnie przeprowadza kolejne mapowanie, które również wykonuje `Thread.sleep(500)`, a potem zwraca wartość przemnożoną przez 2
        - Wypisuje wszystkie wartości z użyciem `Sink.foreach`
        - Wypisuje czas wykonania całego strumienia z użyciem `.thenRun(…)`
    - Dodaj wywołanie `.async()` po każdym mapowaniu
    - Zmierz ponownie czas wykonania
    - Dodaj bufor o wielkości 3 między źródłem, a pierwszym mapowaniem, z overflow strategy typu dropHead: `.buffer(3, OverflowStrategy.dropHead())`
    - Sprawdź jak buforowanie z podaną strategią wpływa na zwracane elementy i czas wykonania
    - Zmień strategię na `dropTail`
    - Sprawdź jak buforowanie z tą strategią wpływa na zwracane elementy i czas wykonania
    - Zmień strategię na `backpressure`
    - Sprawdź jak buforowanie z tą strategią wpływa na zwracane elementy i czas wykonania
    - **W raporcie należy umieścić:** output z 5 etapów zadania (w tym czasy wykonania):
        - Mapowanie bez async
        - Mapowanie z async
        - Dodany bufor ze strategią dropHead
        - Bufor ze strategią dropTail
        - Bufor ze strategią backpressure

4. Sposób oceny/ uzyskania zaliczenia

Ocena z ćwiczeń lab. 0 – 5 punktów:

- 1 punkt za potwierdzenie obecności na zajęciach (20% oceny)
- 4 punkty za wykonane zadania (80% oceny)

5. Literatura

- https://doc.akka.io/docs/akka/current/index-classic.html

Załączniki

- Załączniki do ćwiczeń znajdują się na platformie UPEL w kursie Systemy Rozproszone