Model aktorów - prostsza skalowalność, odporność na awarie, współbieżność i komunikacja zdalna

Marek Lewandowski
Wydział Elektroniki i Technik Informacyjnych, Politechnika Warszawska
marek.m.lewandowski@gmail.com

Streszczenie. *Artykuł omawia model aktorów oraz bibliotekę Akka. Model aktorów pozwala na łatwiejsze pisanie współbieżnych i wielowątkowych programów, dzięki wyższemu poziomowi abstrakcji od wątków i innych prymitywów. Artykuł omawia zastosowanie modelu aktorów w wysoko dostępnych rozproszonych systemach.*

1. Motywacja. Zacznijmy od przedstawienia kilku powodów, dlaczego warto zdecydować się na model aktorów. Jednym z głównych powodów, dla których tworzenie współbieżnych programów jest trudne, jest *stan*, a dokładnie *zmienny współdzielony stan*. Weźmy za przykład konto bankowe, w którym stanem jest aktualne saldo na rachunku. Aby operacje takie jak wpłaty, wypłaty, przelewy i inne wykonywały się poprawnie, muszą mieć dostęp do poprawnego stanu salda, a zmiany salda z tych operacji nie mogą być nadpisane przez inne operacje. Sprowadza się to do ogólnie znanej synchronizacji.

Istnieje wiele różnych technik synchronizacji. Do podstawowych należą prymitywy synchronizacyjne, takie jak:

- semafory,
- muteksy,
- monitory.

Każdy z nich pozwala na synchronizację, która uchroni stan, ale poprawna synchronizacja nie należy do najprostszych zadań, zwłaszcza w bardzo współbieżnych programach. Trzeba wiedzieć, w jakiej kolejności zajmować blokady, w jakiej kolejności je zwalniać; można ich zająć za dużo lub za mało, oraz trzeba wiedzieć, co tak naprawdę w danej sytuacji powinno być chronione. Łatwo w takich przypadkach popełnić błąd, który może doprowadzić do zakleszczeń, zagłodzeń lub walki o zasoby.

Należy zauważyć, że jeśli stan nie ulega żadnych zmianom, to nie ma potrzeby synchronizacji, a zatem znikają wszystkie problemy dotyczące synchronizacji. Używanie tak niskopoziomowych mechanizmów jak wątki, blokady, semafory, jest problematyczne dla programisty, ze względu na synchronizację. Potrzeba zatem czegoś więcej, potrzeba lepszej abstrakcji, która pozwoli uniknąć niskopoziomowych problemów.

1.1. Dostępność. Większość systemów powinna mieć wysoką dostępność w ciągu dnia, tygodnia, roku. W przeciwnym wypadku tracimy klientów, a co za tym idzie -pieniądze.

Systemy mają różną rangę dostępności liczoną w tzw. dziewiątkach. 99% dostępności oznacza, że na rok system może być niedostępny przez 3.65 dni, czyli 1.68 godzin na tydzień. Taki system ma kategorię dostępności w wysokości dwóch dziewiątek.

Systemy telekomunikacyjne mają nawet siedem dziewiątek, co pozwala na przestój w tygodniu w wymiarze 0.0605 sekund. Są to systemy działające praktycznie ciągle, które nigdy się nie zatrzymują. W jaki sposób zrealizować system z wysokim poziomem dostępności?

Istnieje kilka metod zapewnienia dostępności systemów. Można spróbować redundancji sprzętu, ale takie rozwiązanie marnuje zasoby oraz nie jest skalowalne. Inną metodą jest stworzenie systemu, który jest odporny na błędy i samoleczący się.

Systemy samoleczące się nie zakładają, że sytuacje fatalne nie występują. One po prostu sobie z nimi radzą. W klasycznym przypadku, gdy używamy jedynie wątków, musimy chronić się przed błędami, które zostaną wyeskalowane na poziom wątku, ponieważ gdy tak się stanie, to nie mamy możliwości reakcji. W praktyce oznacza to, że w kodzie programu jest bardzo dużo instrukcji typu *try catch*, które „łapią" wszystkie możliwe wyjątki, aby tylko wątek nie wybuchł. Jednak takie rozwiązanie na nie wiele się zda w przypadku rozproszonego systemu, w którym np. awarii ulegnie jeden z węzłów. Nikt wtedy się o tym nie dowie. Potrzeba zatem mechanizmu, który pozwala sobie radzić z błędami, a więc pozwala na tworzenie samoleczących się systemów.

1.2. Komunikacja zdalna. Klasyczne rozwiązania, takie jak *RPC*, dają iluzję bezpieczeństwa, chowając złożone mechanizmy przed programistą. Programista dostaje mechanizm, który wygląda jak proste wołanie metody, ale z ceną braku możliwości obsługi problemów z siecią. Problemy z siecią nie należą do rzadkości. Przepustowość jest ograniczona, topologia sieci może się zmienić, występują spore opóźnienia. Są to problemy, którym po prostu należy stawić czoła, a zatem chcemy mieć możliwość radzenia sobie z nimi.

1.3. Skalowalność. Skalowalność to zdolność systemu do obsłużenia rosnącego obciążenia. Zazwyczaj chcemy skalować horyzontalnie, a więc dodawać kolejne maszyny i w ten sposób obsługiwać większe obciążenie.

Aby wykorzystać pełne możliwości pojedynczego komputera i wszystkich procesorów potrzeba także tworzyć współbieżne programy. Prawo Amdahla mówi o tym, gdzie jest limit przyspieszenia systemu w zależności od tego, w jakiej części jest on współbieżny. Jeśli program jest w 90% współbieżny, to limit 10-krotnego przyspieszenia zostanie osiągnięty dla 1024 procesorów. Dla 50% 2-krotne przyspieszenie jest dla 16 procesorów. Dalsze zwiększanie liczby procesorów nic nie da. Wniosek jest taki, że system powinien być jak najbardziej współbieżny, aby pozwalał w pełni korzystać ze wszystkich procesorów.

2. Model aktorów. Model aktorów ma swoje początki teoretyczne w publikacji Carla Hewitta [1]. Jest to dość prosty koncepcyjnie model. Występuje w nim byt *aktor*, który porozumiewa się z innymi aktorami tylko poprzez *wiadomości*. Każdy aktor posiada swoją skrzynkę na wiadomości, z której to sekwencyjnie pobiera wiadomość i ją przetwarza. Aktor działa tylko, gdy przetwarza wiadomość - oznacza to, że aktor nie zajmuje zasobów, jeśli nie ma nic do zrobienia. Wysłanie wiadomości jest nieblokujące. Przetwarzanie jest asynchroniczne. Komunikacja jest jednokierunkowa - na wiadomość nie trzeba odpowiadać.

Aktor enkapsuluje w sobie stan oraz zachowanie. Stan jest chroniony z definicji, ponieważ jest modyfikowany tylko i wyłącznie przez aktora, który operuje na wiadomościach sekwencyjnie, a więc nie ma potrzeby synchronizacji.

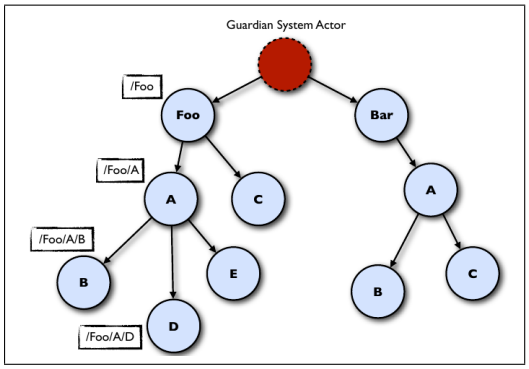
Zachowanie jest zdefiniowane na poziomie aplikacji, poprzez odpowiedni protokół, który programista nadaje wiadomościom. Aktorzy tworzą hierarchię rodzic-dziecko. Każdy aktor może stworzyć innego aktora, dla którego staje się rodzicem.

Rozwiązywanie problemów w modelu aktorów można rozpatrywać w podobny sposób, jak gdyby grupa ludzi miała rozwiązać ten problem. Prosty problem może być rozwiązany przez jedną, bądź kilka osób. Przy większej liczbie osób dość naturalnie pojawią się grupy, które będą pracować razem. Możliwe, że grupy będą miały kogoś, kto będzie nimi kierował, a zatem będzie istnieć hierarchia. Jak mówi Carl Hewitt, jeśli problem jest zbyt skomplikowany, to należy dodać więcej aktorów i rozbić go na mniejsze problemy.

3. Biblioteka Akka. Akka jest biblioteką implementującą model aktorów. Blisko jej do implementacji aktorów z języka Erlang. Można z niej korzystać z języka Java oraz Scala. Charakteryzuje się tym, że aktor ma bardzo małe wymagania pamięciowe. Na 1Gb sterty można stworzyć 2.5 miliona aktorów. Sama biblioteka została napisana z myślą o rozproszonych systemach, a zatem jest rozproszona od podstaw. Lokalne wysyłanie wiadomości między aktorami zostało jednak zoptymalizowane.

3.1. Przezroczystość lokalizacji. Ten mechanizm zapewnia łatwe rozproszenie systemu. Każdy aktor jest dostępny dla innego aktora przez specjalny obiekt *ActorRef* - referencja na aktora. Jest to też zabezpieczenie przed bezpośrednią modyfikacją stanu aktora. Nie wiadomo zatem, czy aktor pod daną referencją jest aktorem lokalnym (działającym na tej samej maszynie JVM), czy aktorem dostępnym na innej maszynie.

3.2. Hierarchie. Aktorzy tworzą hierarchię. Hierarchia jest drzewem, w którym korzeniem jest aktor specjalnego typu. Ten aktor nigdy nie może umrzeć. Chroni się go poprzez hierarchię aktorów pod nim. Na ilustracji widzimy, że aktor Foo jest rodzicem dla aktorów A i C. Do aktora można się również dostać poprzez jego adres. Adres jest ścieżką względem korzenia. Przykładowe ścieżki do aktorów zostały zaznaczone w prostokątach.



Rysunek 1. A picture of the same gull looking the other way!

3.3. Rozproszenie. Aby rozproszyć system używając Akka, należy jedynie zmodyfikować jego konfigurację. Nie ruszamy zatem żadnej linii kodu. Konfiguracji podlega struktura hierarchii. Należy skonfigurować, jakie poddrzewo jest gdzie uruchamiane. Dla ilustacji */Foo* może dotyczyć jednej maszyny, a */Bar* drugiej.

3.4. Strategie nadzoru. Strategie nadzoru pozwalają na samoleczenie się systemu i poprawną obsługę błędów. Każdy rodzic ustala, jaka jest strategia nadzoru dla jego dzieci. Może to być Jeden-Za-Jeden lub Wszyscy-Za-Jednego. W przypadku Jeden-Za-Jeden, gdy aktor umrze, np. wskutek nieobsłużonego wyjątku, tylko ten aktor zostanie uruchomiony ponowie w nowym wcieleniu - a zatem ze stanem początkowym. W drugim przypadku wszystkie dzieci danego aktora zostaną uruchomione ponownie. Sama akcja, jak ponowne uruchomienie, to tylko przykład. Można wykonać dowolny kod, ale ponowne uruchomienie jest wbudowane w bibliotekę i często używane.

4. Podsumowanie. Model aktorów jest prosty i zrozumiały. Pozwala na bezpiecznie programowanie współbieżne. Udostępnia nowy poziom abstrakcji - aktora. Pozwala tworzyć systemy, które w łatwy sposób mogą obsługiwać błędy aplikacji, sieciowe, sprzętowe. Akka ma wysoką wydajność, jest w pełni asynchroniczna i pozbawiona blokad. Pozwala na bardzo proste rozproszenie aplikacji. Istnieje do niej wiele rozszerzeń, np. do budowy klastra, do zapisu stanu aktora w bazie danych.

Moja praca magisterka dotyczy opracowania transakcji lub mechanizmu dającego podobny efekt dla rozproszonej bazy danych Apache Cassandra. Model aktorów pozwoli na stworzenie asynchronicznego, wysoko wydajnego rozproszonego sterownika transakcji.

Literatura

[1] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR

Formalism for Artificial Intelligence. *IJ-CAI*, pages 235–245, 1973.