УДК 004.4

М.М. Глибовець, С.С. Гороховський, С.О. Зінчук, М.В. Кравченко

РЕНЕСАНС ВИКОРИСТАННЯ МОДЕЛІ АКТОРІВ ДО ПОБУДОВИ ПАРАЛЕЛЬНИХ ТА РОЗПОДІЛЕНИХ ЗАСТОСУНКІВ

У статті проаналізована модель акторів (Actor model) як засіб високорівневого підходу до побудови паралельних та розподілених систем. Досліджено розвиток моделі під впливом парадигми об'єктно-орієнтованого програмування та формування головних властивостей акторів. Розглянуто основні особливості реалізації моделі за допомогою об'єктно-функціональної мови Scala на прикладі бібліотеки Akka.

Вступ

Сучасні тенденції розвитку мікропроцесорної архітектури свідчать про рух у напрямку до широкого використання багатоядерних процесорів у спеціалізованих серверах та комп'ютерах. Розробники новітніх програмних систем мають враховувати дані особливості розвитку апаратного забезпечення, створюючи паралельні та розподілені застосунки, які вільно масштабуються для різної кількості наявних процесорів та ядер, а також обсягів задач [1, 2].

Проблемам створення масштабованих застосувань, здатних одночасно обробляти великі обсяги даних, приділяється значна увага. За значного розширення програмного проекту стає все важче і важче досягати ефективного виконання програми з використанням низькорівневих конструкцій синхронізації [3]. Тому виникла гостра потреба ефективної підтримки паралельних і розподілених обчислювальних систем, що складаються з десятків, сотень або навіть тисяч незалежних мікропроцесорів, що мають власну локальну пам'ять та процесор обміну повідомленнями, які б спілкувалися через швидкісні мережі передачі даних [4].

Модель акторів (Actor model), як один із підходів до побудови та аналізу таких систем, постала на ґрунті фізичних ідей, зокрема квантової фізики та загальної теорії відносності. Вперше модель акторів (МА) з'явилась у працях групи дослідників під керівництвом Карла Х'юітта (Carl Hewitt) [5]. Згодом свій внесок у розвиток моделі зробили дослідження Ірен Грейф та Вільяма Клінгера [6, 7]. Нарешті, після публікації у 1986 праці Гуля Аги відбулося остаточне становлення моделі [8].

Серед сучасних доробків виділяють дослідження, що відбувалися в Федеральній політехнічній школі Лозанни (EPFL), а саме праці Філіпа Халлера та Мартіна Одерськи. У них досліджувались практичні аспекти впровадження моделі акторів усередину об'єктно-функціональної мови Scala та пов'язані з цим дослідницькі рішення [8–13]. Логічним підсумком цих досліджень стала поява фреймворку Akka, який пропонує інструментарій для створення відмовостійких масштабованих розподілених застосунків [11, 14, 15, 16]. Наразі частина фреймворку, що відповідає за імплементацію моделі акторів, стала частиною програмної системи мови Scala.

Отже, метою цієї статті будуть дослідження особливостей МА як засобу проектування та аналізу розподілених програмних систем з високою завантаженістю (high load systems), що здатні обробляти великі обсяги даних та ефективно використовувати наявні обчислювальні потужності. Певною мірою у статті використані результати роботи [17], де проведено порівняння основних можливостей мов Scala, Erlang і Haskell та їх ефективності при розв'язанні практичних задач на багатоядерних архітектурах. На прикладі задачі знаходження досконалих чисел на певному проміжку було показано, як за допомогою акторів отримати значне при-

скорення програми на багатоядерних архітектурах.

Модель акторів

Карл Х'юітт розглядав МА як спеціалізовану математичну теорію, яка розглядає «акторів» як універсальні примітиви одночасних цифрових обчислень (concurrent digital computation) [16]. Потенційна «мова акторів» (Actor language) є дуже гнучкою і потужною, що дозволяє створювати програми, окремі модулі, які можуть бути як сильно, так і слабко зв'язані за необхідністю. Свого часу модель також було використано для аналізу сучасних концепцій, таких як обмін повідомленнями без прив'язки до місцезнаходження (location independent messaging) та переміщення активних програм між кількома комп'ютерами [18].

Основною ідеєю моделі є спеціалізоване середовище, яке складається з великої кількості «легких» сутностей, які називаються акторами, кожен з яких відповідає за виконання певного невеликого завдання. Загалом, система розв'язує більш складні задачі за допомогою взаємодії між акторами, делегування завдань новим акторам та обміном повідомленнями між акторами. Аналогом такого обміну повідомлень можуть бути: сигнали, що передаються системною шиною між процесором та пам'яттю; передача параметрів між функціями програми; повідомлення, що передаються між географічно розподіленими комп'ютерами у мережі.

Основним елементом моделі є актор. Його можна охарактеризувати як об'єкт, який унікально ідентифікується в системі і має поведінку, що описує його. Взаємодія з іншими акторами відбувається за допомогою асинхронної передачі повідомлень.

Для забезпечення детермінованої поведінки акторів та системи в цілому, для підтримки прозорого аналізу роботи програми на реалізацію МА накладаються наступні додаткові обмеження: кожен окремо взятий актор може обробляти в певний момент часу тільки одне повідомлення; обробка одного повідомлення є атомарною операцією (тобто під час обробки повідомлення актор не може розпочати обробку іншого повідомлення) [3].

Відокремлення актора-відправника від повідомлень, які він надсилає, стало фундаментальною перевагою моделі акторів, яка зумовила такі характерні особливості передачі повідомлень (patterns of message passing) як можливість асинхронного обміну повідомленнями та емуляцію структур керування завдяки передачі повідомлень з особливим вмістом.

Актор може обмінюватись повідомленням лише з актором, чия адреса йому відома. У роботі [16] виділяють такі способи реалізації адрес: за допомогою прямої фізичної прив'язки (direct physical attachment); як адреси в пам'яті або у дисковому просторі; як мережеві адреси; як адреси електронної пошти. Розробники конкретної реалізації моделі мають право приймати власне рішення щодо деталей імплементації адрес та інших питань, наприклад, подібних до можливості кількох акторів володіти однією адресою.

Більшість дослідників виділяють наступні відмінності МА від її попередників та більшості сучасних моделей обчислень. Найперше – це можливість паралельного виконання (concurrent execution) під час опрацювання повідомлення. До того ж актор не потребує спеціальної реалізації типових для інших моделей сутностей, таких як окремий потік (thread), скриня повідомлень, черга повідомлень, окремий процес в операційній системі. Організація передачі повідомлень має подібні накладні витрати до організації циклів та виклику процедур. Важливим є і те, що поведінка актора визначена лише під час обробки повідомлень. Повідомлення у моделі відокремлені від відправника і в системі акторів доставляються за принципом «максимуму зусиль» (best efforts basis) [16]. Це значно відрізняється від попередніх підходів до моделі паралельних обчислень, в яких передача повідомлень тісно пов'язана з відправником. Відсутність синхронності викликала досить багато непорозумінь за часи розробки моделі акторів і до цього часу залишається суперечливим аспектом. Завжди потрібно пам'ятати, що ця теоретична модель ви-

користовує обмін повідомленнями як абстракцію, і цю абстракцію потрібно відокремлювати від будь-якої конкретної реалізації моделі. Конкретна імплементація повинна враховувати особливості апаратного забезпечення. Окремі реалізації МА мають право вільно використовувати потоки, черги, глобальні присвоєння, когерентну та транзакційну пам'ять, ядра процесора, до тих пір поки це не суперечить положенням моделі.

Розвиток МА та її зв'язок з об'єктно-орієнтованою парадигмою

Після праць Гуля Аги та його колег, а також з розвитком об'єктно-орієнтованого підходу погляд на модель акторів дещо змінився, а актор почав трактуватись як розширення об'єкта з певними властивостями.

В об'єктно-орієнтованій парадигмі програмування об'єкт інкапсулює дані і поведінку. Це відокремлює інтерфейс об'єкта (те, що робить об'єкт) від його реалізації (як він це робить). Такий поділ дозволяє модульний підхід до аналізу об'єктно-орієнтованих програм і полегшує їхню еволюцію. Актори розширюють переваги об'єктів для конкурентних обчислень, відокремлюючи контроль (де і коли відбуваються певні дії) від логіки обчислень. Модель дозволяє розбиття програми на автономні, незалежні, інтерактивні компоненти, що працюють асинхронно. Надалі ми розглянемо основні властивості акторів з точки зору об'єктно-орієнтованого підходу.

Для початку розглянемо поняття системи (середовища, мережі) акторів. Система акторів складається з деякої кількості автономних сутностей – акторів. Кожен актор має власний потік контролю (thread of control) та стан. Він також є окремою одиницею паралелізму (unit of concurrency) в МА. Стан актора інкапсулюється і не поширюється між іншими акторами. Кожен актор оновлює свій локальний стан за допомогою обробки повідомлень, які він отримує у свою скриню повідомлень. Можна стверджувати, що скринька повідомлень тепер стає складовою частиною актора, або принаймні тісно з ним пов'язана.

Актори спілкуються між собою за допомогою обміну повідомленнями. Актор-відправник надсилає повідомлення до іншого актора-отримувача і продовжує роботу без блокування. Отримувач має скриньку повідомлень для отримання всіх повідомлень. Він обробляє кожне повідомлення за наступним принципом: у кожен момент часу оброблюється одне повідомлення за один атомарний крок (single atomic step). Крок складається з дій – реакцій на отримане повідомлення, які були описані в попередньому пункті. Таким чином забезпечується макрокрокова семантика (macro-step semantics), яка є критично важливою для міркувань про системи акторів.

На рис. 1 описується ідеологія спілкування між акторами, останні не мають спільного стану. Натомість, кожен актор забирає повідомлення зі скриньки повідомлень, оброблює його та оновлює свій стан. Кожне повідомлення уособлює у собі завдання. Терміни «завдання» та «повідомлення» можуть використовуватись для заміни одне одного. Згідно з [19], завдання визначається як кортеж, який складається з дескриптора (tag, який відрізняє завдання від усіх інших завдань), місця призначення (яке є ім'ям актора, що має отримати завдання), повідомлення (яке є інформацією, потрібною для виконання завдання).

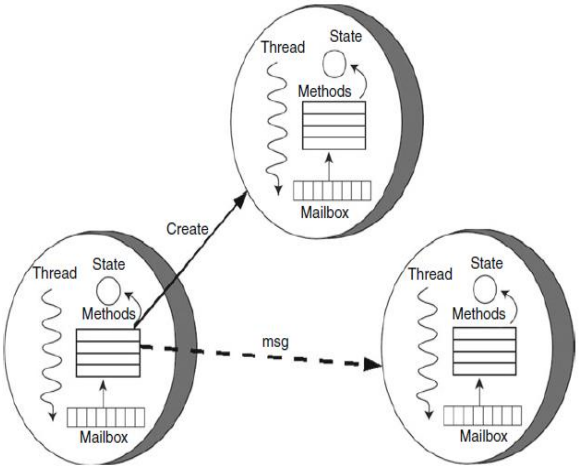

Рис. 1. Спілкування між акторами [19]

Місцем призначення завдання є адреса актора-отримувача. Ця адреса може бути адресою скриньки повідомлень. В деяких системах, які побудовані на моделі акторів, вона є унікальним ім'ям актора, яке має таку саму функціональність як і адреса скриньки повідомлень. Ім'я актора не має бути чимось таким, що можна відгадати. Актор може відправляти повідомлення лише тим акторам, про яких йому відомо. Він може дізнатися адреси інших акторів з отриманих повідомлень або під час створення за допомогою конструктора (як класичний об'єкт отримує певні параметри при створенні).

Структура актора, а конкретніше його реалізації за допомогою найбільш поширених мов програмування, містять дві основні компоненти: представлення стану та представлення поведінки.

Представлення стану значно залежить від конкретної мови, за допомогою якої реалізована модель акторів. В об'єктно-орієнтованих мовах, таких як Java або Scala, стан представлений за допомогою полів об'єктів. В функціональних мовах, таких як Erlang, стан може бути переданий як кортеж значень у циклі обробки та виконання.

Представлення поведінки тісно пов'язано з процесом обробки та надходження повідомлень. У кожного актора є скринька чи черга повідомлень. В окремих мовах та бібліотеках програміст може звертатись до них напряму, інші системи приховують їх від користувача. Процес доставки повідомлення M полягає у пошуку відповідного обробника для нього та надання обробнику потрібної інформації з повідомлення. Під час обробки повідомлення жодна інша дія всередині актора не відбувається.

Зазвичай для пошуку відповідного обробника для повідомлення використовується так званий процес зіставлення зі зразком (pattern matching process). Кожне повідомлення зіставляється з набором зразків. Якщо зіставлення з певним зразком пройшло успішно, перегляд інших зразків припиняється і відбувається обчислення виразів або виконання операторів, які асоціюються з даним зразком. Після цього, актор готовий отримати та обробити нове повідомлення. Якщо повідомлень для обробки немає, актор чекає доти, доки отримає нове повідомлення для обробки. Абстрактна структура обробки та отримання повідомлень всередині актора може бути описана наступним чином:

```
loop {
  receive M
  match M with
    P1 : S1
    P2 : S2
    .
    .
    .
    Pn : Sn
}
```

Конкретні реалізації акторів можуть використовувати техніку зіставлення зі зразком, якщо дозволяють можливості мови програмування (Scala, Erlang) . У випадку мови Java та бібліотек, побудованих з її допомогою, використовуються інші методи обробки, наприклад, підхід з використанням Java Reflection.

Завершуючи огляд основних положень МА, варто підкреслити і описати чотири фундаментальні властивості систем акторів: інкапсуляція, справедливість (fairness) або справедливе планування (fair scheduling), прозорість дислокації (location transparency) та мобільність.

Інкапсуляція є одним з основних принципів об'єктно-орієнтованого програмування. Захист меж інкапсуляції між об'єктами забезпечує такі властивості безпеки, як надійність пам'яті, свободу від перегонів за даними (data race freedom), безпечну модифікацію стану об'єкта. Наприклад, Java розглядається як мова з надійною пам'яттю, оскільки вона ховає покажчики пам'яті за посиланнями на об'єкт, що забезпечує безпечний доступ до об'єкта (наприклад, заборонено використовувати арифметичний покажчик). Надійність пам'яті важлива для збереження об'єктної семантики: вона дозволяє доступ до стану об'єкта тільки з використанням добре визначених інтерфейсів. У контексті МА існують дві важливі вимоги до інкапсуляції: інкапсуляція стану і безпечна передача повідомлень.

Інкапсуляція стану полягає у тому, що актор не може мати доступу до змінних стану іншого актора. Єдиний спосіб для доступу і зміни стану актора полягає у відправці йому повідомлень. Таким чином, мови програмування або бібліотека, на базі якої мають реалізовуватись актори, мають надавати певний механізм для приховування даних і забезпечення того, що доступ до них та їх зміна можливі лише за допомогою передачі повідомлень.

Безпечна передача повідомлень пов'язана з тим, що передається всередині повідомлень. Для того щоб гарантувати, що у акторів немає спільного стану, обмін повідомленнями має мати семантику виклику за значенням (call-by-value semantics). Така семантика вимагає створення копій конкретних даних та вкладення цих копій всередину повідомлення навіть у випадку систем зі спільною пам'яттю. Іншими словами, потрібно забезпечити те, що жоден вказівник або посилання на дані не передаються у повідомленнях. Окремі мови спонукають програмістів використовувати незмінювані об'єкти всередині повідомлень або явно здійснювати копіювання даних перед відправкою повідомлень. Інші дозволяють вирішувати, копії яких даних мають створитись, а для яких можна дозволити надсилати посилання. Такі мови надають необхідні засоби для анотації параметрів.

Концепція справедливого планування полягає в тому, що у підсумку кожне повідомлення в системі акторів надійде до свого одержувача. Винятком може слугувати лише випадок, коли актор, якому призначене повідомлення, «вимкнений» (перебуває у нескінченному циклі або намагається здійснити заборонену операцію). Під доставкою повідомлення ми розуміємо успішну обробку повідомлення та видалення його зі скрині повідомлень. В рамках семантики акторів не визначено жодного точного способу гарантування справедливого планування. Концепцію можна реалізувати за допомогою черги повідомлень типу FIFO або іншого механізму. Хай би яким був підхід, головна мета – переконатись у тому, що жодне повідомлення не залишиться у поштовій скриньці нескінченно.

В межах семантики моделі акторів, місцезнаходження актора жодним чином не впливає на процес обчислень і діяльність інших акторів. Ім'я актора ніяк не відображає його місцезнаходження. Кожен актор володіє унікальним списком адрес інших акторів, з якими він взаємодіє, і ці актори можуть працювати на графічному процесорі, на іншому вузлі в комп'ютерній мережі або на тому ж центральному процесорі. Місцезнаходження актора впливає лише на час очікування повідомлення, а не на семантику системи. Прозорість дислокації (location transparency) надає розробникам можливість створювати програми, не турбуючись про справжнє фізичне розташування актора. Оскільки один актор не має відомостей про простір адрес інших акторів, бажаним наслідком даного підходу є інкапсуляція стану. Також відсутність зв'язку між іменем актора та фізичним розташування забезпечує можливість міграції акторів між різними вузлами або мобільність.

Досить важливо мати здатність переміщувати дані та обрахунки з одного вузла-обробника на інший. Це стає все більш актуальним у зв'язку з поширеною на даний час гетерогенною архітектурою, за якої ми хочемо мати можливість створювати збалансоване робоче навантаження та ефективно використовувати такі різноманітні обчислювальні потужності як графічні процесори та цифрові сигнальні процесори. Для забезпечення мобільності ми повинні мати змогу переміщувати дані, програму та стан обчислень з одного місця на інше. Існують два типи мобільності – слабка та сильна. Сильна мобільність полягає у здатності системи підтримувати переміщення коду та стану виконання (execution state). Слабка мобільність, з іншого боку, дозволяє лише переміщення коду (за винятком, можливо, ще і початкового стану, який також може бути переміщений). В системах акторів слабка мобільність зручна для міграції «застиглого» актора, у котрого порожня черга обробки повідомлень. Сильна мобільність означає можливість для актора мігрувати водночас, коли він все ще займається обробкою повідомлення. Її складніше реалізувати, оскільки пот-

рібно забезпечити можливість переривання обчислень в певній точці коду. Як зазначено у [19], завдяки підтримці модульності контролю та інкапсуляції, мобільність є досить природною для МА.

Реалізації МА

До останнього часу в системі програмування мови Scala та бібліотек, що базуються на ній, знаходилось дві найбільш популярні реалізації моделі акторів: бібліотека Scala.lang.actors, яка надходила разом зі стандартним середовищем розробки для Scala, та частина бібліотеки Akka, присвячена саме реалізації моделі акторів. Обидві бібліотеки розглядали реалізацію моделі і відповідні допоміжні компоненти як частину Scala, а саме як конкретні класи, які реалізовують відповідні інтерфейси та підтримують поведінку, яка відображає властивості моделі з урахуванням внутрішніх особливостей реалізації. Нещодавно всередині команди розробників мови Scala було прийнято рішення про припинення подальшого розвитку бібліотеки Scala.lang.actors і включення саме реалізації на базі бібліотеки Akka всередину стандартної системи програмування для Scala. Саме тому в подальшому ми зосередимось на спільних підходах для обох бібліотек, роблячи наголос на інструментарії, який надає розробникові Akka. Підходи до внутрішньої реалізації моделі акторів та обґрунтування обраних архітектурних рішень можна знайти у [9–11]. Сучасні рішення для створення паралельних застосунків та їхня реалізація за допомогою Scala добре описана у [2, 21]. Найновішу інформацію щодо компонентів, які підтримуються бібліотекою Akka, можна знайти у [22].

У мові програмування Scala є власний вбудований пакет для реалізації МА. Але наразі він все ще не знайшов активного застосування в комерційних проектах, в основному через наявність певних дефектів та відсутність функціоналу, який необхідний для запуску системи акторів у розподіленому середовищі. Тому розвивалися інші проекти, які прагнули усунути обмеження стандартної реалізації. До таких проектів можна віднести такі бібліотеки, як Scalaz, Lift actors та Akka. Найбільшої популярності набула бібліотека Akka, яка доступна не тільки для мови Scala, але й також для мови програмування Java.

В описі змін до останньої версії 2.11 мови Scala, яка була запущена 6 березня 2014 року, описано перспективи розвитку моделі акторів у мові програмування Scala [22, 23]. Автори зазначають, що тепер до складу стандартних бібліотек входитиме остання на момент виходу scala-2.11 версія бібліотеки akka-actor (2.3.0-RC4).

Розширення можливостей стандартної реалізації бібліотекою Akka в основному містить наступні нововведення. Модель «наглядачів» (Supervisors) надає підтримку обробки системних помилок при роботі моделі, які можуть бути пов’язані з вимкненням вузлів, на яких запущена програма, втраті повідомлень під час обміну тощо. За допомогою механізмів тайм-аутів, стратегій відкату (roll-back) та розподілення відправки повідомлень досягається ефективне та якісне розподілення навантаження в системі. Як зазначають розробники бібліотеки Akka [24] їхня система була успішно впроваджена у великих організаціях, для яких критичною є висока пропускна здатність та низька затримка обробки величезної кількості інформації та запитів.

Коли ми створюємо актори засобами бібліотеки Akka, разом з ними під час виконання програми і на етапі компіляції створюються не лише актори, а й супутні компоненти, які забезпечують коректну роботу з акторами. Основними елементами, котрі створює Akka, є Actor (конкретний екземпляр актора), скринька повідомлень Mailbox, диспетчер Dispatcher і проксі над актором ActorRef. Структуру їх взаємодії показано на рис. 2, [25].

Актори – це «живі» об’єкти, які реагують на повідомлення, що надходять ззовні. Тому немає сенсу моделювати за допомогою акторів щось, що є статичним. Є лише один спосіб звернутися до актора – надіслати йому повідомлення. І так само єдиний спосіб щось зробити з цим повідомлення – обробити його у методі receive.

Повідомлення на мові Scala реалізуються за допомогою особливої структури – часткових класів (case classes).

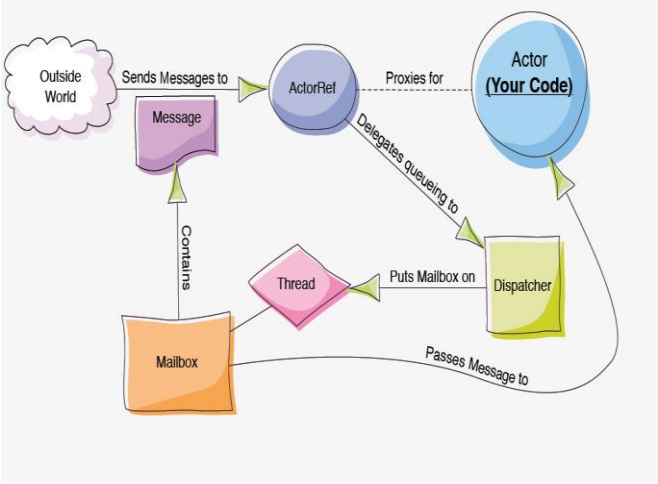

Рис. 2. Погляд на актора в Akka та його компоненти

Об'єкти таких класів є незмінними, значення їхніх атрибутів задаються в конструкторі під час створення. Також вони реалізують спільний з Java інтерфейс Serializable, що дозволяє вільно передавати об'єкти-повідомлення по мережі. Найпростіші класи повідомлень можуть виглядати наступним чином:

```
case object Greet
case class WhoToGreet(who: String)
case class Greeting(message: String)
```

Менш лаконічне та більш знайоме представлення даних класів на мові Java матиме вигляд:

```
public static class Greet implements Serializable {}

public static class WhoToGreet implements
Serializable {
public final String who;
public WhoToGreet(String who) {
 this.who = who;
 }
}

public static class Greeting implements
Serializable {
public final String message;
public Greeting(String message) {
 this.message = message; }}.
```

Варто наголосити, що протягом усього часу свого існування бібліотека Akka надає два набори API (Application Programming Interface) – прикладного програмного інтерфейсу, котрий дозволяє використовувати можливості бібліотеки як зі Scala-застосунків, так і зі застосунків, створених за допомогою технології платформи Java. Така можливість присутня завдяки тому, що обидві мови компілюються у байт-код (виконуваний код), який виконується в середовищі JVM (Java Virtual Machine).

Кожен клас актора, який створює користувач бібліотеки, має імплементувати трейт Actor (аналог абстракції інтерфейсу у Java) у Scala і перевизначити метод receive, в якому розробник має описати, яким чином актор має реагувати на відповідні повідомлення, що надходять. Scala, як уже згадувалось раніше, використовує підхід «зіставлення зі зразком» (pattern matching), який походить з парадигми функціонального програмування і дозволяє зручно та лаконічно описувати обробку повідомлень. Ключовими у реалізації даного підходу в Scala є оператори case та => . З точки зору об'єктно-орієнтованої парадигми, клас актора є звичайним класом, стан якого міститься у його внутрішніх змінних. Однак на відміну від класичних об'єктів, «змінити» цей стан можна лише надіславши відповідне повідомлення. Реалізація класу актора, який буде обробляти повідомлення описаних вище типів, може виглядати таким чином:

```
class Greeter extends Actor {
var greeting = ""

def receive = {
case WhoToGreet(who)=>greeting=s"hello,$who"
case Greet => sender tell Greeting(greeting)}}.
```

Akka забороняє явно створювати екземпляри акторів за допомогою оператора створення екземплярів об'єктів (в Java та Scala це оператор new). Натомість для створення актора потрібно звернутись до екземпляру класу ActorSystem. Об'єкти даного класу слугують «фабриками» для створення об'єктів акторів. Виклик відповідного методу класу ActorSystem повертає як результат своєї роботи не пряме посилання на об'єкт класу актора, а екземпляр класу ActorRef, за допомогою якого мають відбуватись усі взаємодії з актором. Даний підхід «непрямого звертання» до об'єкта забезпечує, насамперед, згадану в

попередньому розділі концепцію прозорості дислокації (location transparency). Це означає, що ActorRef**,** зберігаючи одну й ту саму семантику, може репрезентувати екземпляр актора, який може знаходитися в одному процесі або на віддаленому персональному комп'ютері, тобто справжнє розташування екземпляру не має жодного значення. Також виникає можливість природнім чином оптимізувати топологію застосунку під час виконання, змінюючи місцезнаходження актора. Підхід «непрямого звертання» дозволяє також використовувати модель «нехай впаде» («let it crash») для керування відмовами, яка дозволяє системі «вилікувати» себе, знищивши та створивши заново ті актори, які викликають помилки. Scala-код створення системи та екземпляру актора досить короткий.

```
val system = ActorSystem("helloakka")
val greeter = system.actorOf(Props[Greeter], "greeter")
```

Варто підкреслити, що рядкові ідентифікатори, які передаються в конструктор та виклик метода actorOf є досить важливими в Akka і дозволяють у подальшому віднаходити конкретні екземпляри акторів.

В бібліотеці Akka [24] актор описується як трейт.

```
type Receive = PartialFunction[Any, Unit]
trait Actor {
def receive: Receive
    …
 }
```

Трейт (trait) – одна з основних конструкції мови програмування Scala, яка подібна до абстрактних класів мови програмування Java. Як і в Java, наявна можливість надання реалізації методів та множиний мікс-ін трейтів у конкретому класі, який реалізує відповідні методи. Цю концепцію було впроваджено в Java 8, яка була запущена 18 березня 2014, у вигляді «методів за замовчуванням» (default methods) в інтерфейсах.

Трейт-актор має функцію *receive*, що повертає часткову функцію, яка отримує на вхід параметр з типом Any і повертає результат з типом Unit. Ця функція описує відповідь актора на вхідне повідомлення.

Оскільки в Scala будь-який об'єкт є класом, то є необхідність у визначенні базового класу для всіх класів. Таким класом і є *Any*. Будь-який інший клас прямо або непрямо є нащадком цього класу.

Клас Unit є аналогом ключового слова *void* в мові програмування Java, і використовується для позначення того, що функція не повертає результат.

Функція приймає на вхід будь-яке повідомлення (тип Any), що надає можливість легкої абстракції та додавання нових типів повідомлень в майбутньому, не змінюючи основного інтерфейсу класу актора. Ця семантика схожа на реалізацію моделі акторів в мові програмування Erlang, в якій використовується динамічна система типів. Використання базового надкласу всіх класів дозволяє функції обробки повідомлення абстрагуватися від конкретних типів, подібно до того, як в мовах програмування з динамічною типізацією параметр може бути будь-якого типу.

Функція виконує певний набір операцій, але нічого не повертає тому, хто її викликав, адже через асинхронну обробку повідомлень можлива ситуація коли той, хто її викликав, уже завершив своє виконання.

Розглянемо приклад реалізації простого актора за допомогою бібліотеки Akka:

```
class Counter extends Actor {
        var count = 0
        def receive = {
                case "increment" => count += 1
                case ("get", customer: ActorRef) => customer ! count
        }
}
```

В цьому коді описується клас актора під назвою Counter, який має свій внутрішній стан – змінну count, що є лічильником. Також у класі визначається метод *receive*, який повертає часткову функцію Receive, що була описана раніше.

З прикладу можна побачити, що актор обробляє два типи повідомлень: «increment» (збільшує лічильник на одиницю) та повідомлення “get”, що має додатковий параметр з типом ActorRef, який відображає унікальну «адресу» актора в системі. У ході обробки цього повідомлення актор-обробник відправляє нове повідомлення, що містить значення лічильника count, актору, від якого було отримане початкове повідомлення.

Оператор **!** є скороченням для оператора **tell** в класі ActorRef, що дозволяє відправляти повідомлення акторам.

Для полегшення написання коду взаємодії між акторами в трейті Actor визначені два додаткових поля з типом ActorRef: implicit val self: ActorRef (зберігає власну адресу), def sender : ActorRef (зберігає адресу актора, що надіслав повідомлення, яке зараз обробляється; правильність цієї адреси гарантується обмеженням, що актор може обробляти тільки одне повідомлення в конкретний момент часу).

Таким чином обробку повідомлення «get» можна переписати у більш зрозумілий варіант, прибравши параметр *customer*. Модифікований варіант матиме наступний вигляд:

```
case “get” => sender ! count.
```

Незважаючи на те, що поведінка актора описується за допомогою реалізації функції *receive,* дозволяється динамічно змінювати поведінку акторів за допомогою поля трейту під назвою *context* з типом ActorContext, що відповідає за виконання коду, який описує поведінку актора. В цьому класі визначаються дві функції def become і def unbecome.

Метод def become(behavior: Receive, discardOld: Boolean = true): Unit додає на вершину стеку нову поведінку. Оскільки ActorContext виконує поведінку, яка знаходиться на вершині стеку, то ця щойно додана починає виконуватися для нових повідомлень.

Метод def unbecome() : Unit виштовхує поведінку, що знаходиться на вершині стеку. ActorContext бере поведінку, що знаходиться у вершині модифікованого стека і виконує її для всіх нових повідомлень.

Врахувавши останнє, початковий приклад можна переписати, забравши змінну лічильника:

```
class Counter extends Actor {
  def counter(n: Int) : Receive = {
        case “increment” =>
context.become(counter(n+1))
        case “get” => sender ! n
  }
  def receive = counter(0)
}.
```

Для цього в акторі Counter визначається метод *counter,* який повертає поведінку. Його параметром є лічильник. Початкова поведінка описується в функції *receive* і відповідає поведінці з лічильником 0 (початкове значення). У випадку якщо надійшло повідомлення «get», функція *counter* повертає поточне значення лічильника в поведінці. Якщо ж надійшло повідомлення «increment», то поведінка змінюється – лічильник збільшується на одиницю. У разі, якщо наступне повідомлення буде «get», то як результат повернеться актуальне значення лічильника.

Такий підхід до створення обробників повідомлень належить до стилю функціонального програмування. Окремі автори називають його «асинхронною хвостовою рекурсією». Перевагою цього підходу вважають те, що зміна стану відбувається тільки в одному місці (у виклику функції become) і стан локалізований у конкретній поведінці та залишається незмінним у межах поведінки, що унеможливлює його випадкову або неконтрольовану зміну. Також це гарантує виконання поведінки з визначеним для неї станом.

Ще двома корисними методами в трейті ActorContext є def actorOf і def stop:

- def actorOf(p: Props, name: String): ActorRef отримує на вхід властивості актора та унікальне ім’я (яке корисне під час логування роботи акторів) та повертає “адресу” цього актора в системі;
- def stop(a: ActorRef) : Unit дозволяє припинити роботу актора, в тому числі того, відносно якого розглядається даний ActorContext.

Підсумовуючи, можна визначити наступні особливості реалізації моделі акторів у межах фреймворку Akka. Актори розглядаються як повністю інкапсульовані, незалежні блоки виконання. Актори можуть бути створені тільки іншими акторами, завдяки чому створюється природна ієрархія, що дозволяє застосовувати багато технік з наглядання (supervising), розподілення тощо. Єдиним механізмом взаємодії акторів є система обміну повідомленнями. Кожен окремо взятий актор може обробляти в певний момент часу тільки одне повідомлення. Обробка одного повідомлення є атомарною операцією (тобто під час обробки повідомлення актор не може розпочати обробку іншого повідомлення). Повідомлення, які відправляються, будуть гарантовано оброблені у тому порядку, у якому вони були відправлені. Опрацьовуючи повідомлення, актори можуть відправляти або створювати нові повідомлення, змінювати свою поведінку для обробки наступних повідомлень тощо.

Висновки

У цій роботі здійснено огляд основних положень моделі акторів як підходу, що знову став актуальним відповідно до сучасних тенденцій розвитку апаратного забезпечення та вимог масштабування. Основну увагу приділено тим властивостям акторів, які мають безпосередній вплив на реалізації за допомогою найпоширеніших мов програмування: інкапсуляції, справедливому плануванню (fair scheduling), прозорості розташування (location transparency), сильній та слабкій мобільності. На прикладі фреймворку Akka розглянуто основні елементи реалізації моделі акторів на платформі Java Virtual Machine, засоби комунікації акторів та особливості використання часткових класів для створення повідомлень, які відображають предметну специфіку задач, що розв'язуються.

1. *Годич О.В., Давидов М.В., Нікольський Ю.В. та інші.* Обчислювальні аспекти аналізу даних на основі карт Кохонена // Вісник Національного університету "Львівська політехніка". Серія: Інформаційні системи і мережі. – 2011. – № 699. – С. 63–72 .
2. *Haller Ph., Sommers F.* Actors in Scala. Artima Press, Walnut Creek, California, 2011. – 184 p.
3. *The* Neophyte's Guide to Scala Part 14: The Actor Approach to Concurrency [Електронний ресурс] / Daniel Westheide – Режим доступу: http://danielwestheide.com/blog/2013/02/27/the-neophytes-guide-to-scala-part-14-the-actor-approach-to-concurrency.html
4. *Foundations* of Actor Semantics [Електронний ресурс] / William D. Clinger – Режим доступу: http://dspace.mit.edu/bitstream/handle/1721.1/6935/AITR-633.pdf
5. *Hewitt C., Bishop P. and Steiger R.* A Universal Modular Actor Formalism for Artificial Intelligence // IJCAI'73 Proceedings of the 3rd International Joint Conference on Artificial Intelligence.– Morgan Kaufmann Publishers Inc., San Francisco, 1973. – P. 235–245.
6. *Clinger W.* Foundations of Actor Semantics // MIT Press, Cambridge, Massachusetts, 1981. – 178 p.
7. *Greif I.* Semantics of Communicating Parallel Processes // MIT Press, Cambridge, Massachusetts, 1975. – 189 p.
8. *Agha G.A.* ACTORS: A Model of Concurrent Computation in Distributed Systems // MIT Press, Cambridge, Massachusetts, 1986. – 190 p.
9. *Haller Ph., Odersky M.* Event-based Programming without Inversion of Control // Modular Programming Languages: 7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13–15, 2006: Proceedings. – Lecture Notes in Computer Science, 2006. – Vol. 4228. – P. 4–22.
10. *Haller Ph., Odersky M.* Actors That Unify Threads and Events // Proc. of the 9th Inter. Conf. on Coordination Models and Languages, COORDINATION'07. – Springer-Verlag, Berlin, Heidelberg, 2007. – P. 171–190.
11. *Haller Ph., Odersky M.* Scala Actors: Unifying Thread-based and Event-based Programming // Journal of Theoretical Computer Science, 2009. – Vol. 410, N 2–3. – P. 202–220.
12. Odersky M., Spoon L., Venners B. Programming in Scala: A Comprehensive Step-by-Step Guide – Artima Inc, 2011. – 852 p.

13. *Odersky M.* Scala by Example [Електронний ресурс] // Programming methods laboratory EPFL, Switzerland. – 2014. – Режим доступу: http://www.scala-lang.org/docu/files/ScalaByExample.pdf .
14. *Lockney T., Tay R.* Developing an Akka Edge // Bleeding Edge Press, 2014. – 173 p.
15. *Wyatt D.* Akka Concurrency // Artima Inc., Walnut Creek, California, 2013. – 515 p.
16. *Hewitt C.* Actor Model of Computation: Scalable Robust Information Systems [Електронний ресурс] //Presented at Inconsistency Robustness2011. Stanford University. August 16–18, 2011. – Режим доступу: http://arxiv.org/ftp/arxiv/papers/1008/1008.1459.pdf.
17. *Гороховський С.С., Кравченко М.В.* Порівняння ефективності застосування мов Scala, Erlang і Haskell в умовах багатоядерних архітектур // Наукові записки НаУКМА. – 2013. – Т. 151. – С. 68–74.
18. *Greif I.* Semantics of Communicating Parallel Processes // MIT Press, Cambridge, Massachusetts, 1975. – 189 p.
19. *Karmani R.K., Shali A., Agha G.* Actor frameworks for the JVM platform: a comparative analysis // PPPJ ’09: proceedings of the 7th international conference on principles and practice of programming in java, Calgary, Alberta.– ACM, New York, 2009. – P. 11–20.
20. *Subramaniam V.* Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors // Pragmatic Bookshelf, 2011. – 280 p.
21. *Akka* Documentation. Release 2.2.3 [Електронний ресурс]. – Typesafe Inc., October 23, 2013. – Режим доступу: http://doc.akka.io/docs/akka/2.2.3/AkkaScala.pdf.
22. *SCALA* 2.11.0-RC1 IS NOW AVAILABLE! [Електронний ресурс] / Scala Docs – Режим доступу: http://www.scala-lang.org/news/2014/03/06/release-notes-2.11.0-RC1.html.
23. *What* features can the Akka platform offer, over the competition? [Електронний ресурс] / Akka docs – Режим доступу: http://doc.akka.io/docs/akka/snapshot/intro/why-akka.html.
24. *Actors* [Електронний ресурс] / Akka docs – Режим доступу: http://doc.akka.io/docs/akka/snapshot/scala/actors.html.
25. *Wyatt D.* Akka Concurrency // Artima Inc., Walnut Creek, California, 2013. – 515 p.

Одержано 04.07.2014

Про авторів:

Глибовець Микола Миколайович,
доктор фізико-математичних наук,
професор,
декан факультету інформатики НаУКМА,

Горохівський Семен Самуїлович,
кандидат фізико-математичних наук,
доцент кафедри інформатики
факультету інформатики НаУКМА,

Зінчук Сергій Олександрович,
магістр факультету інформатики,

Кравченко Михайло Васильович,
магістр факультету інформатики.

Місце роботи авторів:

Національний університет
«Києво-Могилянська академія»,
04655, Київ,
вул. Г. Сковороди, 2.
Тел.: (044) 463 6985.
E-mail: glib@ukma.kiev.ua,
gor@ukma.kiev.ua

Тел.: +38 (093) 148 6220.
E-mail: serge.zinchuk@gmail.com

Тел.: +38 (093) 447 6960.
E-mail: mike.kravchenko.ua@gmail.com