УДК 004.42

А.Н. Глибовец

Решение задачи распределенного индексирования в оперативной памяти на базе модели актеров с использованием фреймворка *Akka*

Описана программная система построения распределенного координатного индекса в оперативной памяти на базе модели актеров с использованием языка программирования *Java* и фреймворка *Akka*. Система позволяет подключать в сеть любое количество машин, легко расширятся и обрабатывать фразовые запросы. Реализованная в системе поддержка репликаций обеспечивает стабильную работу.

The distributed solving system for construction of coordinate index in the RAM based on model of actors using Java programming language and framework Akka is described. The system allows to connect any number of machines, it expands easily, allowing to process phrase queries. The system provides ability to replicate that ensures its stable operations.

Описано програмну систему побудови розподіленого координатного індексу в оперативній пам'яті на базі моделі акторів з використанням мови програмування *Java* і фреймворку *Akka*. Система дозволяє під'єднувати будь-яку кількість машин, легко розширюватися та опрацьовувати фразові запити. Реалізована в системі підтримка реплікації забезпечує стабільну роботу.

Введение. При построении поисковых машин существенное место занимает задача индексирования [1]. В последнее десятилетие начали говорить о возможности ее решения с использованием оперативной памяти. Предлагаем распределенный вариант построения индекса и его хранения в оперативной памяти с использованием *актеров* и фреймворка *Akka* [2].

Примем, что на всех компьютерах распределенной системы (сети) установлена и запущена некоторая среда работы с актерами. Входящие документы распределяются по сети. После инициализации запуска системы создаются актеры индексирования файлов. На каждом компьютере запущен актер обработки запроса пользователей через графический интерфейс. Получив запрос, он направляет его всем актерам, ответственным за индексирование в сети. Запросы обрабатываются, и результаты возвращаются к актеру. Окончательный результат пересылается пользователю. В случае, если какой-то актер не ответил на запрос, последний переадресовывается на репликацию. Агенты строят координатный индекс, позволяющий добывать различную информацию из документов и качественнее отвечать на информационные запросы пользователя.

Практическим применением этой системы может быть построение поисковой машины для небольших фирм, нуждающихся в быстром доступе к данным. Понятно, что для значительных объемов информации этот подход проблематичен из-за потребности в большом количестве оперативной памяти.

Задача индексирования заключается в сокращении количества обрабатываемого текста при запросе поиска. Для этого массив текстовых данных, по которому будет продолжаться поиск, сокращается до некоторого набора ключевых слов. Различают два основных типа индексов: прямой и обратный [1]. В первом для каждого документа строится массив ключевых терминов, которые в нем встречаются. Массив добавляется в индекс. Во втором – для каждого ключевого термина сопоставляется массив документов, в которых он встречается. Это избавляет от хранения повторяющихся терминов и позволяет удобнее обрабатывать поисковые запросы. Координатный индекс – это обратный индекс, который для каждого термина, кроме документов, в которых этот термин встречается, также сохраняет позиции этих терминов. Для удобства обработки будем хранить количество вхождений термина в каждый документ и общую частоту его появления.

Модель актеров

В настоящее время в разработке многопоточных приложений доминирует подход использования общего переменного состояния – большо-

го количества объектов, каждый из которых работает в своем потоке. Объекты характеризуются состоянием и могут быть изменены в любой момент в разных частях программы. Обычно в таком коде для синхронизации взаимодействия потоков накапливается много блоков, управляемых *замками* (примитивами синхронизации). Последние используются для обеспечения контролируемого изменения состояния и предотвращения ситуации, когда несколько потоков решают изменить состояние одновременно. Но чрезмерное количество таких блоков может значительно снизить быстродействие программы, поскольку потоки долго будут ждать высвобождения нужных ресурсов.

Главная проблема использования низкоуровневых конструкций синхронизации (замки, потоки) – сложность определения поведения и работы программы. В таких системах часто появляются дефекты («борьба за ресурсы, дедлок»), которые могут быть не найдены на этапе тестирования, но могут привести к катастрофическим ошибкам при загрузке и использовании приложения на реальных серверах. Понятно, что при увеличении проекта также становится все труднее достичь эффективного выполнения программы с использованием низкоуровневых конструкций синхронизации [3].

Для решения упомянутых проблем рекомендуется использовать модель актеров, позволяющую писать эффективные многопоточные программы и легко анализировать поведение программы во время работы.

Толчком к разработке модели стала «необходимость поддержки параллельных вычислительных систем, которые содержат большое количество независимых процессоров, каждый с собственной локальной памятью и процессором обмена сообщениями, и общаются через скоростные сети передачи данных» [4]. Модель актеров (*Actor model*), как один из подходов к построению таких систем, возникла на почве физических идей, в частности квантовой физики и общей теории относительности. На нее также повлияли языки программирования *Lisp*, *Simula* 67 и *Smalltalk*-72 вместе с концепциями систем на базе мандата доступа (*capability-based systems*) и коммутации пакетов (*packet switching*). Впервые модель актеров появилась в трудах группы исследователей под руководством Карла Хьюитта (*Carl Hewitt*). Логическим итогом различных исследований стало появление фреймворка *Akka*, предложившего инструментарий для создания отказоустойчивых масштабируемых распределенных приложений. Теперь часть фреймворка, ответственная за проведение модели актеров, стала частью программной системы языка *Scala*.

Основной идеей модели есть среда, состоящая из большого количества «легких» сущностей, называемых актерами. Традиционно актер ответствен за выполнение отдельного небольшого задания. В общем, система решает более сложные задачи, взаимодействуя с актерами, делегируя задачи новым актерам и обмениваясь сообщениями. Примером такого обмена сообщениями в реальной жизни могут быть сигналы, передаваемые системной шиной между процессором и памятью, передача параметров между функциями программы, сообщения между географически распределенными компьютерами в сети.

Для обеспечения детерминированного поведения актеров и системы в целом, для поддержки прозрачного анализа работы программы на реализацию модели актеров накладываются следующие дополнительные ограничения: каждый отдельно взятый актер может обрабатывать в определенный момент только одно сообщение; обработка одного сообщения – это атомарная операция (т.е. во время обработки сообщения актер не может начать обработку другого сообщения) [5, 6].

Карл Хьюитт рассматривал актеров в качестве универсальных примитивов одновременных цифровых вычислений (*concurrent digital computation*) [7]. Ученый считает, что модель может быть полезной как для теоретического понимания параллелизма (*concurrency*), так и служить теоретическим базисом для различных практических реализаций параллельных систем. Можно утверждать, что модель также содержит все свойства, необходимые для создания и анализа распределенных систем. Язык

актеров (*Actor language*) – гибкий и мощный, что позволяет создавать программы, отдельные модули которых могут быть как сильно, так и слабо связаны. Например, модель использовалась для анализа обменов сообщениями без привязки к местонахождению (*location independent messaging*) и перемещения активных программ между несколькими компьютерами [6].

Актер можно рассматривать как вычислительную сущность (*computational entity*), которая, получая сообщение, может осуществлять следующие действия: отправлять сообщения другим актерам для создания новых актеров; принимать решение о том, как обрабатывать следующее сообщение, которое она получит.

Нет никакого обусловленного порядка, согласно которому могут происходить указанные действия. Также следует отметить, что два одновременно отправленных сообщения могут прибыть к конкретному актеру в произвольном порядке. В классической формулировке модели особенности поступления некоторого сообщения от одного актера к другому никак не оговариваются. Как отмечено в [5, 7], отделение актера отправителя от сообщений, которые он направляет, стало главным преимуществом модели актеров, обусловившей такие характерные особенности передачи сообщений (*patterns of message passing*), как возможность асинхронного обмена сообщениями и эмуляцию структур управления путём передачи сообщений с особым содержанием.

Актер может обмениваться сообщениями только с теми актерами, чьи адреса ему известны. Адреса могут быть реализованы несколькими способами [5]: посредством прямой физической привязки (*direct physical attachment*); как адреса в памяти или в дисковом пространстве; как сетевые адреса; как адреса электронной почты.

Детали имплементации адресов, а также такие вопросы, как совместное владение несколькими актерами одного адреса и сопутствующие проблемы в модели не рассматриваются. Потенциальные разработчики конкретной имплементации модели имеют право принимать собственное решение, обеспечивая (или нет) при этом соблюдение основных положений модели. Заметим, что соблюдение основных аксиом модели или пренебрежение отдельными из них с целью создания более эффективной реализации модели на базе конкретного языка программирования весьма дискуссионно. Хьюитт отмечает, что актер может хранить только те адреса, которые были ему предоставлены, когда он был создан; полученные из сообщений; предназначенные для актеров, создаваемых текущим актером.

Как отмечено в [5, 7], модель актеров отличается от ее предшественников и большинства современных моделей вычислений следующими положениями:

- параллельным исполнением (*concurrent execution*) во время обработки сообщения;
- актер не требует отдельного потока (*thread*), хранилища сообщений, очереди сообщений, отдельного процесса операционной системы;
- реализация передачи сообщений имеет расходы, подобные реализации циклов и вызовов процедур;
- поведение актера определено только во время обработки сообщений.

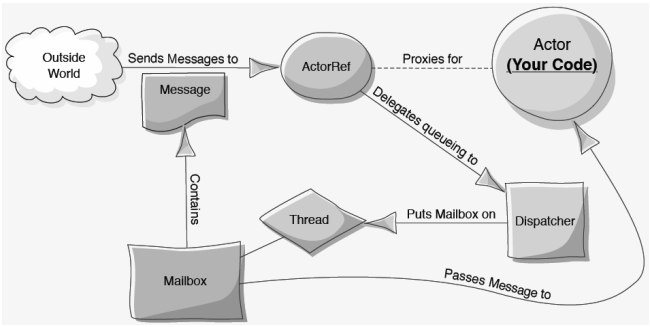
После трудов Гуля Аги и его коллег [8–11], а также с развитием объектно-ориентированного подхода, взгляд на модель актеров несколько изменился, а актера начали воспринимать как расширение объекта с определенными свойствами.

До последнего времени существовало две наиболее используемые реализации модели актеров – библиотека *Scala.lang.actors*, поступавшая вместе со стандартной средой разработки для *Scala*, и часть библиотеки *Akka*, посвященная именно реализации модели актеров. Обе библиотеки рассматривали реализацию модели и соответствующие вспомогательные компоненты как часть *Scala*, а именно как конкретные классы, реализующие соответствующие интерфейсы и поддерживающие поведение, отражающее свойства модели с учетом внутренних особенностей реализации. Разработчиками языка *Scala* было принято решение о прекращении дальнейшего развития библиотеки *Scala.lang.actors* и включения именно реализации на базе библиотеки *Akka* в стандартную систему программирования для

Scala. Поэтому в дальнейшем сосредоточимся на инструментарии, предоставляемом разработчику фреймворком *Akka*. Описание подхода к внутренней реализации модели актеров и обоснование выбранных архитектурных решений представлено в [12–14]. Современные решения для создания параллельных приложений и их реализации с помощью *Scala* описаны в [15, 16]. Последняя информация по компонентам, поддерживаемым библиотекой *Akka*, описана в [17].

Фреймворк *Akka*

Когда создаем актеров средствами библиотеки *Akka*, вместе с ними во время выполнения программы и на этапе компиляции создаются сопутствующие компоненты, обеспечивающие корректную работу. Основные элементы, которые создает *Akka*, – *Actor* (конкретный экземпляр актера), *Mailbox* (ящик сообщений), *Dispatcher* (диспетчер), *ActorRef* (прокси над актером). Структура их взаимодействия представлена на рисунке.



Актер в *Akka* и его компоненты [18]

При передаче сообщения актёру, с ним непосредственно никогда взаимодействия не происходит. Вместо этого отсылается сообщение к объекту-прокси – *ActorRef*. Прокси обращается к диспетчеру по поводу размещения сообщения в очереди сообщений почтового ящика актера. Диспетчер разместит сообщения в поток (*thread*), и когда тот отработает, то достанет из очереди одно из сообщений и отправит его специально определенному методу *receive* для обработки. Следует помнить, что как только сообщение положено (доставлено) в ящик, тот, кто осуществляет вызов, может поступать как пожелает. Единственная блокировка – это размещение сообщения в очередь. После этого, вся остальная работа и обработка осуществляются в другом потоке. Такое уменьшение связности системы позволяет предоставить значительную функциональность реализации актеров в библиотеке *Akka*.

Актеры – «живые» объекты, реагирующие на сообщения, поступающие извне. Поэтому нецелесообразно моделировать с помощью актеров что-то статическое. Есть только один способ обратиться к актеру – отправить ему сообщение. Аналогично, единственный способ что-то сделать с этим сообщением – обработать его в режиме *receive*.

Сообщения на языке *Scala* реализуются посредством особой структуры – частичных классов (*case classes*). Объекты таких классов – неизменны, значение их атрибутов задаются в конструкторе при создании. Также они реализуют совместный с *Java* интерфейс *Serializable*, что позволяет свободно передавать объекты-сообщения по сети.

Akka не допускает явного создания экземпляров актеров с помощью оператора создания экземпляров объектов (в *Java* и *Scala* это оператор *new*). Для создания актера следует обратиться к экземпляру класса *ActorSystem*. Объекты данного класса служат «фабриками» для создания объектов актеров. Вызов соответствующего метода класса *ActorSystem* возвращает как результат своей работы не прямую ссылку на объект класса актера, а экземпляр класса *ActorRef*, с помощью которого должны происходить все взаимодействия с актером. Данный подход «косвенного обращения» к объекту обеспечивает, прежде всего прозрачность дислокации (*location transparency*), т.е. настоящее расположение экземпляра не имеет значения. Также возникает возможность естественным образом оптимизировать топологию приложения во время выполнения, изменяя местонахождение актера. «Косвенное обращение» позволяет использовать модель «пусть упадет» («*let it crash*») для управления отказами, которая позволяет системе «вылечить» себя, уничтожив и создав заново те актеры, которые вызывают ошибки.

Следует отметить, что строчные идентификаторы, передаваемые в конструктор, и вызов

метода *actorOf* достаточно важны в *Akka* и позволяют в дальнейшем находить конкретные экземпляры актеров.

Решение задачи распределенного индексирования в оперативной памяти

Предлагаемый подход к решению задачи может быть описан следующим сценарием. На каждом компьютере среды (сети) запускается *Java*-машина с реализованными *Akka*-актерами. Актеры, ответственные за индексацию, индексируют свой массив файлов и строят координатный индекс, т.е. на каждом компьютере в сети находится часть общего индекса. С любого компьютера можно сделать запрос по общему индексу. Отметим, что пользователь не привязан к какому-то одному узлу, который может работать нестабильно. Запрос рассылается по всем компьютерам в сети и обрабатывается соответствующими актерами. После обработки результаты возвращаются на компьютер, с которого пришел запрос, и выводятся пользователю. В случае недоступности какого-либо из узлов, ищется его репликация и переадресовывается запрос к ней.

Рассмотрим подход более конкретно.

Запуск и создание актеров. При запуске системы первое, что создается – это среда, в которой будут работать актеры. На каждой машине создается объект *ActorSystem*, поддерживающий работу с актерами. Все последующие актеры будут созданы в этой среде. При запуске среды указывается *IP*-адрес хоста и порт, по которому можно будет обратиться к ней, т.е. имеем префикс физического адреса актеров в среде. Например: *IndexingSystem*@192.168.0.100: 2552.

После создания среды на машине строятся два актера: актер-создатель индекса, и актер-обработчик запроса по общему индексу.

Типы сообщений между актерами. Прежде чем перейти к описанию актеров следует определить, какими сообщениями они могут обмениваться. В системе существует два типа сообщений: *SearchQueryResult* и *SearchQueryAsk*.

Сообщение *SearchQueryAsk* направляется как запрос от пользователя к актерам, ответственным за индексирование. Оно содержит только одно текстовое поле *query*, конструктор класса и метод, по которому можно получить запрос *public String getQuery* ().

Сообщение *SearchQueryResult* содержит массив с результирующими файлами, которые соответствуют критериям поиска *ArrayList <String> resultFileNames*, конструктор и гетер результирующих имен файлов.

Оба сообщения реализуют интерфейс *Serializable*, так как для передачи сообщений между актерами эти сообщения необходимо сериализовывать.

Теперь рассмотрим возможные актеры в системе.

Актер индексирования называется *IndexingActor*. Он создает экземпляр класса *PositionalIndex*, который и занимается построением индекса. Файлы, из которых будет строиться индекс, берут из каталога *Books*, который находится в проекте. Актер индексирования реагирует на сообщение *SearchQueryAsk* и обрабатывает их. Все остальные сообщения игнорируются. После получения сообщения запрос, содержащийся в нем, извлекается и обрабатывается методом *public ArrayList <String> search* (*String word*) класса *PositionalIndex*. Последний принимает на вход запрос, а возвращает список имен файлов, удовлетворивших этот запрос. Далее результат записывается в сообщения типа *SearchQueryResult* и посылается отправителю. В результате имеем такой метод обработки:

```
public void onReceive (Object message) throws Exception {
 if (message instanceof SearchQuery.SearchQueryAsk) {
  String query = ((SearchQuery.SearchQueryAsk) message). GetQuery ();
  SearchQuery.SearchQueryResult resul = new SearchQuery.SearchQueryResult
      (this.index.search (query));
  getSender (). tell (result, getSelf ());
 } Else {
  unhandled (message);
 }
}.
```

Индексирование и поиск. Рассмотрим подход к индексированию, а именно класс *PositionalIndex*. Класс содержит два поля: массив файлов *ArrayList <File> files*, которые предсто-

ит проиндексировать, и результирующий индекс *Map <String, TreeMap <Integer, TreeSet <Integer> >> dictionary*. Структура индекса такова. Для каждого термина, встречающегося в тексте, сохраняется список файлов, содержавших этот термин, и для каждого файла хранится список – слово, позиции этого термина.

Рассмотрим работу метода индексирования. Он в цикле считывает поочередно все файлы из перечня, полученного на вход от актера, удаляет знаки препинания и сводит слова к нижнему регистру. Конечно, нужно использовать дополнительные методы обработки, такие как стеминг или лемматизации. Но в данной статье не ставилась цель построить полнофункциональную поисковую систему. Затем каждое слово обрабатывается отдельно. Если входного слова еще нет в регистре, то для него создаются соответствующие массивы и сети, добавляемые в индекс:

```
if (! dictionary.containsKey (word)) {
TreeMap <Integer, TreeSet <Integer>> map =
new TreeMap <Integer, TreeSet <Integer>> ();
TreeSet <Integer> set = new TreeSet <Integer> ();
set.add (wordCount);
map.put (files.indexOf (file), set);
dictionary.put (word, map);
}.
```

Если слово уже есть в словаре, то, возможно, оно встречалось уже в этом файле или только в других файлах:

```
else {
TreeMap <Integer, TreeSet <Integer>> map = dictionary.get (word);
if (! map.containsKey (files.indexOf (file))) {
TreeSet <Integer> set = new TreeSet <Integer> ();
set.add (wordCount);
map.put (files.indexOf (file), set);
} Else {
map.get (files.indexOf (file)). add (wordCount);
}
}
```

При обработке файлов ведется счетчик индекса слов в тексте, который обнуляется перед рассмотрением следующего файла.

Рассмотрим реализацию алгоритма поиска по индексу. Сначала запрос сводится к нижнему регистру, далее убираются все знаки пунктуации и запрос разбивается по словам:

```
public ArrayList <String> search (String query) {
query = query.toLowerCase ();
String [] words = query.split ("[\\ p {Punct} \\ s] +");.
```

Для первого слова из запроса извлекается его список файлов. Если этот список пуст, то в цикле перебираем все файлы, берем координаты вхождения первого слова запроса и смотрим в этом же файле, соответствуют ли следующие координаты тем словам, которые поступили в запрос. Если соответствуют, то добавляем в результирующий массив, иначе переходим к следующему файлу:

```
for (int i = 1; i <words.length; i + +) {
TreeMap <Integer, TreeSet <Integer>> map = dictionary.get (words [i]);
if (map! = null && map.containsKey (fileIndex)) {
if (! map.get (fileIndex). contains (wordCoord + i)) {
  containsAllWords = false;
  break;
}
} Else {
containsAllWords = false;
   break;
}
}
if (containsAllWords) {
  resultFiles.add (fileIndex);
  break;
}
```

После обработки всех файлов возвращаем имена, соответствующие входному запросу.

Актер запроса. Актер, ответственный за работу с запросами, называется *QueryActor*. Его интерфейс состоит из поля для ввода запроса, кнопки поиска и поля для вывода результата.

При нажатии на кнопку *Search* срабатывает слушатель (лисенер), передающий запрос на обработку актеру в виде сообщения *SearchQueryAsk*.

Актер запроса имеет список всех актеров индексирования в виде их физического адреса. В случае расширения системы или переноса ее в другую сеть, все, что необходимо измененить в программе, – это список адресов актеров:

```
private static final Map <String, List <String>> ACTORS = new HashMap <String, List <String>> ();
static {
ACTORS.put ("A", new ArrayList <String> ());
ACTORS.get ("A"). Add ("akka.tcp :/ / IndexingSystem@192.168.0.100: 2552/user/indexActor");
```

```
ACTORS.put ("B", new ArrayList <String> ());
ACTORS.get ("B"). Add ("akka.tcp / / IndexingSystem@192.168.0.142: 2552/user/indexActor");
ACTORS.get ("C"). AddAll (Arrays.asList ("akka.tcp :/ / IndexingSystem@192.168.0.142: 2553/user/indexActor",
"akka.tcp :/ / IndexingSystem@192.168.0.142: 2554 / user / indexActor "," akka.tcp :/ / IndexingSystem@192.168.0.142: 2555/user/indexActor "));
}
```

Отметим, что все актеры разбиты на группы. Таким решением система поддерживает репликации. Когда актер получает сообщение от графического интерфейса пользователя, он рассылает каждому актеру из списка сообщение *SearchQueryAsk* и ожидает ответа:

```
ArrayList <String> totalResult = new ArrayList <String> ();
boolean success = false;
for (String group: ACTORS.keySet ()) {
for (String actor: ACTORS.get (group)) {
if (! success) {
Timeout timeout = new Timeout (Duration. Create (5, "seconds"));
Future <Object> future = Patterns.ask (getContext (). ActorSelection (actor), message, timeout);
try {
SearchQuery.SearchQueryResult result = (SearchQuery.SearchQueryResult) Await.result (future, timeout.duration ()); totalResult.addAll (result.getResultFileNames ());
success = true;
} Catch (Exception e) {
 System.out.println (actor + "is unreachable");
}
success = false;
}.
```

Если ответ не приходит в течение пяти секунд, то такой актер признается недействительным для данного запроса, и запрос перенаправляется следующему актеру из группы. Если ни один актер из группы не соответствует, то запрос возвращает значение из других, и, несмотря на это, выводит на экран:

```
gui.showResults (totalResult).
```

Заключение. Разработанный подход позволяет включать в сеть любое количество машин, легко расширяется. Запросы индексу можно отправлять из любой машины в сети. Построение координатного индекса позволяет обрабатывать фразовые запросы. Реализованная в системе поддержка работ с репликациями обеспечивает ей стабильность.

На базе предложенной системы в дальнейшем возможна разработка полнофункциональной поисковой системы для локальной сети по заданным файлам.

1. *Christopher D. Manning*, *Prabhakar Raghavan*, *Hinrich Schütze* Introduction to information retrieval / Cambridge University Press, 2008. – *http*://nlp.stanford.edu/IR-book/html/htmledition/irbook. html
2. *Gul A*., Abdulnabi Agha. Actors: A Model Of Concurrent Computation In Distributed Systems. – June, 1985
3. *Daniel Westheide* The Neophyte's Guide to Scala Part 14: The Actor Approach to Concurrency. – http://daniel-westheide.com/ blog/2013/02/27/the-neophytes-guide-to-scala-part-14-the-actor-approach-to-concurrency. html
4. *William D. Clinger*. Foundations of Actor Semantics. – http://dspace.mit.edu/bitstream/handle/1721.1/6935/AITR-633.pdf
5. *Carl Hewitt*, *Peter Bishop*, *Richard Steiger*. A Universal Modular ACTOR Formalism for Artificial Intelligence. – http://citeseerx.ist.psu.edu/viewdoc/summry?doi=10.1.1.77.7898
6. *Botev N*. Actor-based Concurrency in Newspeak 4. – Master's Projects. Paper 231, 2012. – http://scholarworks.sjsu.edu/etd_projects/231
7. *Hewitt C*. What is computation?Actor Model versus Turing's Model / A Computable Universe: Understanding Computation & Exploring Nature as Computation. Dedicated to the memory of Alan M. Turing on the 100th anniversary of his birth. Edited by Hector Zenil. – World Scientific Publ., 2012. – 856 p. – http://what-is-computation.carlhewitt.info
8. *Agha G.A*. ACTORS: A Model of Concurrent Computation in Distributed Systems. – MIT Press, Cambridge, Mass., 1986. – 190 p.
9. *Agha G*. Concurrent Object-oriented Programming // Commun. ACM. – 1990. – **33**, N 9. – P. 125–141.
10. *Foundation* for Actor Computation / Gul Agha, Ian A. Mason, Scott Smith et al. // J. of Functional Programming. – 1997. – **7**, N 1. – P. 1–72.
11. *Clinger W*. Foundations of Actor Semantics. – Cambridge, Mass., MIT Press, 1981. – 178 p.
12. *Haller Ph*., *Odersky M.* Event-based Programming without Inversion of Control// Modular Programming Languages: 7th Joint Modular Languages Conf., JMLC 2006 Oxford, UK, Sept. 13–15, 2006: Proc. – Lecture Notes in Computer Sci., 2006. – **4228**. – P. 4–22.
13. *Haller Ph*., *Odersky M.* Actors That Unify Threads and Events // Proc. of the 9th Inter. Conf. on Coordination Models and Languages, COORDINATION'07. – Berlin, Heidelberg: Springer-Verlag, 2007. – P. 171–190.
14. *Haller Ph*., *Odersky M.* Scala Actors: Unifying Thread-based and Event-based Programming // J. of Theoretical Comp. Science, 2009. – **410**, N 2–3. – P. 202–220.

Окончание на стр. 72

15. *Haller Ph*., *Sommers F*. Actors in Scala. – Walnut Creek, Calif.: Artima Press, 2011. – 184 p.
16. *Subramaniam V*. Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors. – Pragmatic Bookshelf, 2011. – 280 p.
17. *Akka* Documentation. Release 2.2.3. – Typesafe Inc., Oct. 23, 2013. – http://doc.akka.io/docs/akka/2.2.3/Akka Scala.pdf
18. *Wyatt D*. Akka Concurrency. – Walnut Creek, Calif.: Artima Inc., 2013. – 515 p.

Поступила 04.07.2014
Тел. для справок: +38 067 409-4355 (Киев)
E-mail: *andriy@glybovets.com.ua*

●