УДК 004.424

А.Ю. Дорошенко, Д.С. Тітов

ПАРАЛЕЛЬНА РОЗПОДІЛЕНА СИСТЕМА ДЛЯ АНАЛІЗУ ПОТОКОВИХ ДАНИХ СОЦІАЛЬНИХ МЕРЕЖ

Проведено порівняння ефективності технології віддаленого виклику процедур (RMI) та фреймворку для розподілених обчислень (Hazelcast), як засобів для розробки кластерної системи. Запропонована паралельна розподілена динамічно масштабована відмовостійка система для обробки потокових даних великого обсягу. Проведена перевірка та первинне дослідження цієї системи на прикладі обробки даних соціальної мережі Twitter. Розглянутий механізм розгортання створеного кластеру в хмарній платформі.

Ключові слова: мережа, аналіз, відмовостійкість, масштабування, кластер.

Вступ

Зростання популярності соціальних медіа останнім часом змусило суттєво трансформуватись індустрію маркетологічніх та соціологічних досліджень. Замість того, щоб формувати контрольні групи, коли організація хоче отримати джерело думок про їхню продукцію, рекламу або брендинг, стало набагато простіше і дешевше знайти цільову аудиторію, – та їх думки – в соціальних мережах.

На початку розвитку соціальних ЗМІ, дослідники мали можливість оцінювати соціальні настрої в ручному режимі, адже обсяг інформації був відносно малий. Проте наразі потоки даних в Інтернеті в цілому та соціальних мережах зокрема сягнули колосальних масштабів, тому виникла потреба у створенні високопродуктивних систем для автоматизованої обробки цих даних [1]. Більш того, сучасні бізнес стратегії, що реалізують CRM-підхід, потребують не лише пасивного аналізу соціального контенту, а ще й передбачають проактивну реакцію на певні сигнали, що надходять з соціального Інтернет-простору. Таким чином, дослідження показують, що 64 % споживачів очікують отримати зворотній зв'язок від компанії в режимі реального часу та 94 % готові відмовитись від послуг певної компанії, якщо обслуговування клієнтів не передбачає цього [2].

У роботі [3] зазначається, що однією з найбільш цікавих з точки зору аналізу відгуків про бренди соціальних мереж є мікроблогінгова платформа Twitter. Перевагою Twitter є те, що повідомлення про актуальні події з'являються в цій соціальній мережі набагато швидше, ніж в будь-яких інших новинних джерелах. Тому Twitter стали розглядати як корисне джерело інформації, яку можна використовувати для прогнозування будь-яких показників, у тому числі і фінансових. Twitter може слугувати джерелом інформації про настрої користувачів, тобто з нього зручно виокремлювати психологічну складову соціальних обговорень.

Відзначено, що пікове навантаження цієї соціальної мережі оцінюється в 140000 повідомлень в секунду та час обробки такого обсягу даних за допомогою наївного баєсівського класифікатору з використанням апаратного забезпечення [4] може сягати 12.458 секунди. Це свідчить про те, що система, призначена для обробки потокових даних великого обсягу, має бути паралельною та розподіленою, для того щоб встигати обробляти вхідні дані вчасно.

Також обчислення у реальному часі великої кількості даних грає велику роль у високочастотному трейдингу – основній формі алгоритмічної торгівлі на фінансових ринках, в якій сучасне обладнання та алгоритми використовуються для швидкого здійснення торгівельних операцій над цінними паперами. Як зазначається в [5], затримка в обробці даних вже навіть в 10 мілісекунд може призвести до суттєвих фінансових збитків.

© А.Ю. Дорошенко, Д.С. Тітов, 2015

В роботі розглянуто дві технології, на базі яких можна створити паралельну розподілену систему для обробки даних з соціальних мереж в реальному часу: Java Remote Method Invocation [6] та Hazelcast Framework [7]. Також представлена їх ефективна порівняльна характеристика.

1. Отримання даних з соціальної мережі Twitter

Соціальна мережа Twitter надає доступ до своїх даних у вигляді Twitter REST API та Twitter Firehose. В даній статті не розглядається Twitter REST API, оскільки доступ до нього є пакетним: клієнт відправляє запит до серверу, а той відповідає певним пакетом даних. Така форма доступу не є отриманням інформації у реальному часі, на відміну від Twitter Firehose.

Firehose («пожежний шланг») – технологія, за якою Twitter надає текстові повідомлення одразу як вони з'являються в системі, тобто у реальному часі [8]. Доступ до повного потоку контенту є платним, проте існує безкоштовна лімітована версія – Twitter Sample Stream, що фактично являє собою Twitter Firehose, обмежений 1 % кількості повідомлень.

Існує декілька Java-провайдерів для взаємодії з Twitter Sample Stream, одним з найпопулярніших та найзручніших є Spring Social, а семе його підмодуль – Spring Social Twitter [9]. Для початку прослуховування вхідного потоку достатню створити Spring Bean, що імплементує інтерфейс *StreamListener*:

```
@Component
public class TwitterStreamListener implements StreamListener {
    @Override
    public void onTweet(Tweet tweet) {
    }

    @Override
    public void onDelete(StreamDeleteEvent streamDeletevent) {
    }

    @Override
    public void onLimit(int i) {
    }
    @Override
    public void onWarning(StreamWarningEvent streamWarningEvent) {
    }
}
```

Метод *onTweet()* цього обробника спрацьовуватиме кожного разу, коли до системи буде надходити чергове повідомлення.

2. Реалізація паралельної розподіленої системи за допомогою RMI

2.1. Загальні відомості про технологію RMI. Перша розглянута технологія, придатна для розподілення обчислень, це Java remote method invocation (RMI) – програмний інтерфейс виклику віддалених методів платформи Java. RMI представляє собою розподілену об'єктну модель, що описує механізм виклику віддалених методів, які працюють на іншій віртуальній машині Java. Під час виклику методу віддаленого об'єкта, в дійсності викликається звичайний Java метод, інкапсульований у спеціальному об'єкті-заглушці, який являє собою представника серверного об'єкта. Заглушка розташовується на клієнтському комп'ютері, а не на сервері. Вона упаковує параметри віддаленого методу в пакет байтів. Кожен параметр кодується за допомогою алгоритму, що забезпечує незалежність від апаратного забезпечення.

2.2. Архітектура паралельної розподіленої системи на основі RMI. В результаті була спроектована архітектура паралельної розподіленої системи, яка передбачає динамічне масштабування обчислювального кластеру за допомогою синхронізації через сервер бази даних. Під час ініціалізації першого обчислювального вузла, він реєструється на сервері бази даних та бере на себе керуючу роль (master). В подальшому цей вузол відповідає за отримання вхідного потоку даних та розподілення його, також master бере участь безпосередньо в обробці даних. Інші обчи-

слювальні вузли можна динамічно підключати до системи, запускаючи той самий Java-додаток на сторонніх машинах. Ці вузли беруть на себе роль «підлеглих» (slaves) та відповідають лише за обробку даних, що надходять від master-вузла.

Далі представлений приклад RMI-сервісу, що може викликатись віддалено для обробки повідомлень:

```
@Component
public class TwitterConsumer implements IConsumer {
    @Autowired
    private TwitterProcessor twitterProcessor;

    public void consume(TweetWrapper tweetWrapper) {
        twitterProcessor.process(tweetWrapper);
    }

    @Override
    public Class<IConsumer> getServiceInterface() {
        return IConsumer.class;
    }
}
```

Інтерфейс *IConsumer* в свою чергу наслідує інтерфейс Remote, для того щоб сервіс можна було внести до RMI-реєстру так експортувати назовні.

До переваг створеної системи можна віднести можливість запуску обчислювального вузла на будь-якому комп'ютері, що має реальна IP-адреса, тобто обчислювальні машини не обов'язково повинні знаходитись в межах однієї локальної мережі (але повинні бути доступними для з'єднання ззовні).

Слабким місцем подібної архітектури є необхідність використовувати окремий сервер бази даних для синхронізації вузлів між собою, адже технологія RMI не має вбудованого механізму для динамічного знаходження інших вузлів на рівні додатку. Отже, якщо сервер бази даних буде виведений з ладу, система перестане функціонувати.

2.3. Відмовостійкість системи на основі RMI. Для забезпечення відмовостійкості системи потрібно реалізувати зміну ролі обчислювального вузла з master на slave у випадку, коли поточний master-вузол припиняє функціонування.

Як вже було зазначено, RMI не має вбудованого механізму синхронізації вузлів, отже цю функціональність необхідно імплементувати окремо. Проблема полягає в тому, що тоді коли master-вузол може легко дізнатись про те, що певний slave-вузол відключився (перехвативши виключну ситуацію при зверненні до віддаленої машини), slave-вузли не мають такої нагоди. Тому постає необхідність реалізувати так званий планувальник, якій з певним інтервалом буде опитувати master-вузол, щоб дізнатись, чи той ще функціонує. Приклад подібного планувальнику наведений нижче:

```
@Component
public class StateChecker {

    @Autowired
    private NodeManager nodeManager;

    @Scheduled(initialDelay = 5000, fixedRate = 500)
    public void checkMasterNode() throws RemoteException, NotBoundException {
        if (nodeManager.isSlaveNode()) {
            Node masterNode = nodeManager.getMasterNode();

            if (masterNode == null || !nodeManager.isNodeRunning(masterNode)) {
                nodeManager.selectNewMasterNode(masterNode);
            }
        }
    }
}
```

Метод *checkMasterNode()* викликається кожні 500 мілісекунд та у разі необхідності ініціює реініціалізацію master-вузла.

3. Реалізація паралельної розподіленої системи за допомогою Hazelcast

3.1. Загальні відомості про фреймворк Hazelcast. Hazelcast – це платформа з відкритим програмним кодом для Java, яка використовується для побудови кластерів та масштабованого розподілу даних. Типовими функціями фреймворку є:

- організація обміну даними/станом серед багатьох серверів та кешування даних (розподілений кеш);
- кластерне розподілення програми та забезпечення захищеної комунікації між обчислювальними вузлами;
- синхронізоване використання даних у пам’яті;
- розподіл обчислень між багатьма серверами та паралельне виконання задач;
- забезпечення відмовостійкого управління даними.

3.2. Архітектура паралельної розподіленої системи на основі Hazelcast. Перевагою використання Hazelcast є автоматичне розгортання та управління обчислювальним кластером: розробник не повинен передбачати окремого механізму синхронізації обчислювальних вузлів між собою (як, наприклад, розглянутий вище підхід з використанням бази даних). Це гарантує ще більшу відмовостійкість, оскільки збій жодного окремого серверу не зможе призвести до того, що система перестане функціонувати в цілому.

Загалом підхід до розподілення обчислень нагадує підхід, розглянутий в попередньому розділі, за винятком того, що для комунікації між вузлами використовуються не об’єкти-заглушки, а розподілений *ExecutorService*, який отримує на вхід екземпляри класів, імплементуючих інтерфейси *Callable* або *Runnable*. Ці представляють собою команди (з інкапсульованими всередині даними), які будуть виконані на певному обчислювальному вузлі. Фрагмент такого класу наведений нижче:

```
public class TwitterCallable implements Callable<TweetWrapper>, Serializable {
…
    public TwitterCallable(TweetWrapper tweetWrapper) {
        this.tweetWrapper = tweetWrapper;
    }

    @Override
    public TweetWrapper call() throws Exception {
        TweetWrapper result = twitterProcessor.process(tweetWrapper);
        return result;
    }
…
}
```

3.3. Відмовостійкість системи на основі Hazelcast. Hazelcast надає дуже зручний механізм моніторингу подій, які трапляються з вузлами кластеру – інтерфейс *MembershipListener*. Цей інтерфейс має три методи, що дозволяють відслідковувати додавання нового вузла, відключення вузла та зміну атрибутів вузла. Саме можливість задавати значення певних атрибутів на елементах кластеру дозволяє відділяти master-вузол від інших та динамічно змінювати конфігурацію системи. Приклад реалізації цього інтерфейсу наведений нижче:

```
public class ClusterMembershipListener implements MembershipListener {
    private static final Logger LOGGER = LoggerFactory.getLogger(ClusterMembershipListener.class);

    private ApplicationContext applicationContext;
```

```
        public void memberAdded(MembershipEvent membershipEvent) {
            LOGGER.info("Member added: {}", membershipEvent);
        }

        public void memberRemoved(MembershipEvent membershipEvent) {
            Boolean isMasterNode = membershipEvent.getMember().getBooleanAttribute(SocialBootApplication.IS_MASTER_NODE);
            if (isMasterNode) {

applicationContext.getBean(NodeManager.class).selectNewMasterNode();
            }
            LOGGER.info("Member removed: {}", membershipEvent);
        }

        public void memberAttributeChanged(MemberAttributeEvent memberAttributeEvent) {
            LOGGER.info("Member attribute changed: {}", memberAttributeEvent);
        }

        public void setApplicationContext(ApplicationContext applicationContext) {
            this.applicationContext = applicationContext;
        }
    }
```

3.4. Розгортання кластеру в хмарині Amazon Elastic Compute Cloud. Для розгортання системи на основі Hazelcast була обрана хмарна платформа Amazon Elastic Compute Cloud (EC2). EC2 представляє собою веб-сервіс, який дозволяє отримати доступ до обчислювальних потужностей і налаштувати ресурси з мінімальними затратами. Він надає користувачам повний контроль над обчислювальними ресурсами, а також доступне середовище для роботи. Служба скорочує час, необхідний для отримання і завантаження нового сервера [10].

Для тестування кластеру в EC2 було замовлено 5 серверів, які мають фіксовані технічні характеристики [11]. На серверах встановлена операційна система Ubuntu Linux, доступ здійснюється за допомогою протоколу SSH.

Загалом Hazelcast передбачає три опції для знаходження вузлів один-одним:

- статична IP-адресація;
- multicast в локальній мережі;
- AWS EC2 Auto Discovery.

Перший підхід не задовольняє потребу динамічного підключення обчислювальних вузлів, а між другим та третім був обраний AWS EC2 Auto Discovery, оскільки він представляє собою просту та зручну інтеграцію з середовищем EC2.

Нижче наведений метод, що відповідає за створення *HazelcastInstance* – основного класу фреймворку Hazelcast, який інкапсулює в собі всі налаштування кластеру:

```
    @Bean
    public HazelcastInstance hazelcastInstance(@Value("${aws.access.key}") String accessKey,

@Value("${aws.secret.key}") String secretKey) {
        Config config = new Config();

config.getNetworkConfig().setPort(5900);

config.getNetworkConfig().setPortAutoIncrement(true);
        config.addListenerConfig(new ListenerConfig(clusterMembershipListener()));
        NetworkConfig network = config.getNetworkConfig();
        JoinConfig join = network.getJoin();

join.getMulticastConfig().setEnabled(false);

join.getTcpIpConfig().setEnabled(false);
```

```
join.getAwsConfig().setAccessKey(accessKey).setSecretKey(secretKey).setEnabled(true);
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);
        Set<Member> members = hazelcastInstance.getCluster().getMembers();
        Member localMember = hazelcastInstance.getCluster().getLocalMember();

localMember.setBooleanAttribute(IS_MASTER_NODE, members.size() == 1);
        return hazelcastInstance;
    }
```

Як бачимо, інтеграція з EC2 полягає у вказанні ключу доступу та секретного ключу доступу до платформи, які можна отримати в панелі керування хмарним сервісом.

Після запуску додатку на першому сервері, Hazelcast створює кластер, що наразі складається з одного обчислювального вузла. В терміналі це виглядає наступним чином:

```
Members [1] {
    Member [172.31.37.199]:5900 this
}
```

Коли будуть додаватись інші вузли, їх синхронізація між собою відбуватиметься повністю прозоро для розробника, так відображатиметься в терміналі. Крім того, існує можливість запуску декількох екземплярів додатків на одному сервері – в такому випадку кожний вузол стартує на своєму власному порту, завдяки опції *setPortAutoIncrement(true)*:

```
Members [5] {
    Member [172.31.37.199]:5900 this
    Member [172.31.37.198]:5900
    Member [172.31.37.197]:5900
    Member [172.31.37.197]:5901
}
```

Відповідно, динамічне відключення вузлів також миттєво відображається в консолі серверу, що дозволяє своєчасно реагувати на раптові зміни топології розгорнутого кластеру.

4. Інтерфейс управління кластером

Для зручного управління кластером в обох системах реалізований WEB-інтерфейс, який дозволяє керувати станом кожного окремого вузла за допомогою запитів до REST-сервісів.

WEB-інтерфейс побудований на основі фреймворку Jersey: при запуску кожного екземпляру додатку на сервері стартує WEB-сервер, доступний, за замовченням, за адресою http://localhost:8080 (або за значенням зовнішньої реальної IP-адреси, якщо доступ виконується ззовні). Наприклад, для запуску отримання повідомлень від Twitter на master-вузлі, необхідно звернутися за допомогою GET-запиту до точки доступу «/twitter/start», та, відповідно, для припинення обробки до «/twitter/stop». Клас, що реалізує даний сервіс, наведений далі:

```
@Service
@Path("/twitter")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class TwitterService {

    @Autowired
    private TwitterTemplate twitterTemplate;

    @Autowired
    private List<StreamListener> streamListeners;

    private Stream userStream;

    private void startListening() {
        if (userStream == null) {
            userStream = twitterTemplate.streamingOperations().sample(streamListeners);
        }
    }
    private void stopListening() {
```

```
            if (userStream != null) {
                userStream.close();
                userStream = null;
            }
        }
        @GET
        @Path("/start")
        public String start() {
            startListening();
            return "Started";
        }
        @GET
        @Path("/stop")
        public String stop() {
            stopListening();
            return "Stopped";
        }
    }
```

5. Ефективна порівняльна характеристика систем на основі RMI та Hazelcast

5.1. Проведення тестування системи. Для порівняння ефективності розглянутих технологій, була проведена серія експериментів: кожна система запускалась на одному, двох, трьох, чотирьох та п'яти серверах. В кожному випадку десять разів з інтервалом в сто секунд обчислювалась кількість оброблених за цей інтервал повідомлень та фіксувалось середнє арифметичне значення. Результати експерименту та побудований графік представлені далі в таблиці та на рисунку, де S – це кількість серверів, а P – кількість опрацьованих повідомлень.

Таблиця

S	RMI, P	Hazelcast, P
1	404.7	407.8
2	409.3	414.5
3	425.4	431.1
4	410.4	438.5
5	414.2	448.5

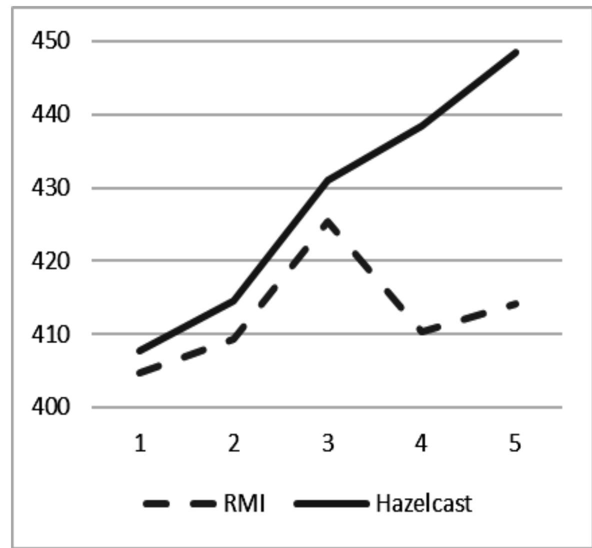

Рисунок. Графік ефективності розглянутих систем

5.2. Аналіз отриманих результатів. Як бачимо, при малій кількості серверів RMI та Hazelcast показують схожу динаміку приросту ефективності (втім, Hazelcast має відносно кращі результати), проте після підключення четвертого вузла RMI демонструю різкий спад ефективності, а Hazelcast продовжує збільшувати ефективність майже лінійно.

Потенційним поясненням такої динаміки є те, що RMI оперує зі з'єднаннями, використовуючи певний кеш, внутрішня реалізація якого може знижувати ефективність системи при збільшенні кількості обчислювальних вузлів. Hazelcast, між тим, використовує пул безпосередніх з'єднань між вузлами через сокети. Крім того, RMI виконує постійні DNS-запити для встановлення відповідності між IP- та DNS-адресами, а це також може призводити до затримок зі збільшенням кількості вузлів.

Інше потенційне пояснення даного ефекту полягає у тому, що реалізований RMI-планувальник, який кожні пів секунди перевіряє статус master-вузла, при збільшенні вузлів може створювати надмірне навантаження на мережу і через це швидкість обміну даними значно падає. В такому випадку може допомогти збільшення інтервалу опитування планувальника, але при збільшенні цього інтервалу також збільшується час, який система буде простоювати у разі раптового відключення

master-вузла, що є неприйнятним, коли існують вимоги до сталого функціонування системи.

Висновки

В роботі розглянуте актуальне питання створення високонавантажених систем, які можуть горизонтально масштабуватись та забезпечувати безперебійну обробку потокових даних великих обсягів. Як приклад джерела даних для обробки обрана соціальна мережа Twitter та її потоковий API - Twitter Firehose (а саме його тестова версія Twitter Sample Stream).

Розроблено дві системи: з використанням Java Remote Method Invocation та Hazelcast Framework. Системи представляють собою динамічно масштабовані та відмовостійкі кластерні рішення, придатні до розгортання в локальній мережі або в хмарній платформі Amazon EC2. Одним з головних надбань розроблених рішень є те, що дослідник може нарощувати ресурси для обчислень за своїм бажанням, різної конфігурації та потужності, що дає змогу досягати бажаної швидкодії системи.

Ці рішення порівняні як з точки зору їх архітектурних та концептуальних переваг і недоліків, так и з точки зору їхньої ефективності. В результаті серії експериментів встановлено, що Hazelcast демонструє кращі результати, та чим більше вузлів додається до кластеру, тим ефективніше він працює, на відміну від RMI. Розглянуто декілька гіпотез відносно цього ефекту.

В подальшому планується оптимізація рішення на основі Hazelcast, реалізація криптографічного захисту даних, що передаються та низка інших покращень.

1. *Social Listening in Practice. Market Research* [Електронний ресурс]. – Режим доступу: https://www.brandwatch.com/guide-market-research/ – 2015 р.
2. *Social Listening in Practice. Social customer service* [Електронний ресурс]. – Режим доступу: https://www.brandwatch.com/customer-service-guide/ – 2015 р.
3. *Тітов Д.С., Дорошенко А.Ю.* Моніторинг соціальних мереж в системі реального часу // Наукова дискусія: теорія, практика, іновації. – 2015. – С. 93–96.
4. *Intel Core i7-3770k Processor* [Электроний ресурс]. – Режим доступу: http://ark.intel.com/products/65523. – 01.11.2012 г.
5. *Charlie Munger*: HFT is Legalized Front-Running [Електронний ресурс]. – Режим доступу: http://blogs.barrons.com/stockstowatchtoday/2013/05/03/charlie-munger-hft-is-legalized-front-running/ – 03.05.2015 р.
6. *Remote Method Invocation* [Електронний ресурс]. – Режим доступу: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html – 2015 р.
7. *Hazelcast* [Електронний ресурс]. – Режим доступу: https://hazelcast.org/ – 2015 р.
8. *Public streams* [Електронний ресурс]. – Режим доступу: https://dev.twitter.com/streaming/public – 2015 р.
9. *Spring Social* [Електронний ресурс]. – Режим доступу: http://projects.spring.io/spring-social/ – 2015 р.
10. *Amazon EC2* [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/ec2/ – 2015 р.
11. *Amazon EC2 Instances* [Електронний ресурс]. – Режим доступу: http://aws.amazon.com/ec2/instance-types/ – 2015 р.

1. *Brandwatch* Social Listening in Practice. Market Research [Online] Available from: https://www.brandwatch.com/guide-market-research/ [Accessed: 26 September 2015].
2. *Brandwatch* Social Listening in Practice. Social customer service [Online] Available from: https://www.brandwatch.com/customer-service-guide/ [Accessed: 26 September 2015].
3. *Titov D.S. & Doroshenko A.Yu.* Social networks monitoring in real-time systems. In Scientific discussion: theory, practice, innovation. Kyiv, Wednesday 27th to Thursday 28th March 2015. – Kyiv: IOMP. – P. 93–96 (in Ukrainian).
4. *Intel* Intel Core i7-3770k Processor [Online] Available from: http://ark.intel.com/products/65523 [Accessed: 26 September 2015].

5. *Charlie Munger* HFT is Legalized Front-Running [Online] Available from: http://blogs.barrons.com/stockstowatchtoday/2013/05/03/charlie-munger-hft-is-legalized-front-running/ [Accessed: 26 September 2015].
6. *Oracle* Remote Method Invocation Home [Online] Available from: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html [Accessed: 26 September 2015]
7. *Hazelcast* Hazelcast [Online] Available from: https://hazelcast.org/ [Accesed: 26 September 2015].
8. *Twitter* Public streams [Online] Available from: https://dev.twitter.com/streaming/public [Accessed: 26 September 2015]
9. *Spring* Spring Social [Online] Available from: http://projects.spring.io/spring-social/ [Accesed: 26 September 2015].
10. *Amazon* Amazon EC2 [Online] Available from: https://aws.amazon.com/ec2/ [Accessed: 26 September 2015].
11. *Amazon* Amazon EC2 Instances [Online] Available from: http://aws.amazon.com/ec2/instance-types/ [Accessed: 26 September 2015].

Одержано 01.10.2015

Про авторів:

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор, завідувач відділу теорії
комп'ютерних обчислень Інституту
програмних систем НАН України,
професор кафедри автоматики і управління в технічних системах НТУУ "КПІ".
Кількість наукових публікацій в
українських виданнях – понад 150.
Кількість наукових публікацій в
іноземних виданнях – понад 30,
Індекс Гірша – 3.
ORCID orcid.org/0000-0002-8435-1451,

Тітов Дмитро Сергійович,
студент факультету інформатики та
обчислювальної техніки,
кафедри автоматики і управління
в технічних системах НТУУ "КПІ".
Кількість наукових публікацій в
українських виданнях – 2.
ORCID orcid.org/0000-0003-1607-5405.

Місце роботи авторів:

Інститут програмних систем
НАН України,
03680, Київ-187,
проспект Академіка Глушкова, 40.
Тел.: (044) 526 1538.
E-mail: dor@isofts.kiev.ua,

Національний технічний університет
України "КПІ"
03056, Київ-56, проспект Перемоги, 37.
Тел.: (044) 236 7989.
E-mail: dmytro.titov@gmail.com