**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**Белорусский государственный Университет**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра многопроцессорных систем и сетей**

ВАТОЛИН

Ростислав Павлович

**КОМПЬЮТЕРНЫЙ МОДУЛЬ КОНТРОЛЯ СИСТЕМЫ ВЕБ-СЛУЖБ НА БАЗЕ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ**

Отчёт о преддипломной практике

студента 5 курса 1 группы

|  |  |  |
| --- | --- | --- |
| “Допустить к защите”  с предварительной оценкой \_\_\_  Руководитель практики  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  “\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2017 г. |  | Научный руководитель:  ассистент кафедры МСС  А.С. Гусейнова |
|  |  |  |

Минск, 2017

**РЕФЕРАТ**

Преддипломная практика, 18 с., 7 рис., 7 источников.

*Ключевые слова*: МЕТЕОРОЛОГИЯ, ВЫСОКОНАГРУЖЕННОЕ ПРИЛОЖЕНИЕ, APACHE STORM, ЗАДАЧА ПРОГНОЗИРОВАНИЯ, МЕТЕОСЛУЖБА, РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ, BIG DATA.

*Объект исследования* – метеорологические веб-сервисы, концепции и методы разработки высоконагруженных приложений, применение фреймворка Apache Storm в создании вычислительного модуля.

*Методы исследования* – обзор и изучение фреймворка Apache Storm, исследование метеорологических сервисов, предоставляющих данные о погоде, получение теоретический и практических знаний в разработке приложений, основанных на распределенных вычислениях.

*Цель работы* – разработка компьютерного модуля на базе фреймворка Apache Storm для контроля веб-метеослужб.

*Результатом* является реализованный модуль контроля веб-служб, предоставляющий достоверного источника и метеорологические данные с высокой точностью, изучение и освоение методов обработки огромного количества данных в реальном времени с помощью фреймворка Apache Storm.

*Областью применения* является метеороло́гия.

[**ВВЕДЕНИЕ 4**](#_Toc476008160)

[1. STORM FRAMEWORK 5](#_Toc476008161)

[1.1 Ключевые особенности фреймворка 5](#_Toc476008162)

[1.2 Элементы Storm 5](#_Toc476008163)

[1.3 Сохранность данных 6](#_Toc476008164)

[1.3.1 На уровне Spout 6](#_Toc476008165)

[1.3.2 На уровне Bolt 6](#_Toc476008166)

[1.3.3 На уровне Tuple 7](#_Toc476008167)

[1.4 Транзакции в Storm 7](#_Toc476008168)

[1.4.1 Atomicity 8](#_Toc476008169)

[1.4.2 Isolation 8](#_Toc476008170)

[1.4.3 Реализация во фреймворке 8](#_Toc476008171)

[1.4.4 Вывод 11](#_Toc476008172)

[2. ПОСТАНОВКА ЗАДАЧИ И РЕАЛИЗАЦИЯ 11](#_Toc476008173)

[2.1 Получение погодных данных 12](#_Toc476008174)

[2.1.1 Accuweather.com 12](#_Toc476008175)

[2.1.1 Darksky.net 13](#_Toc476008176)

[2.2 Хранение данных 14](#_Toc476008177)

[2.3 Отчет 15](#_Toc476008178)

[ЗАКЛЮЧЕНИЕ 17](#_Toc476008179)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 18](#_Toc476008180)

## ВВЕДЕНИЕ

Если следить за прогнозом погоды, то в итоге можно прийти к выводу, что каждый источник будет иметь свою погрешность. В температуре воздуха можно увидеть разницу в пару градусов, во влажности - до 10 процентов, а в атмосферном давлении - до 20 мм рт. ст. Если говорить об атмосферном давлении и влажности, то можно заметить, что расхождения от разных источников значительно отличаются. Неправильные данные об атмосферном давлении, могут причинить дискомфорт людям со слабыми сердечно-сосудистой, дыхательной, центральной нервной системами. Правильные данные о влажности воздуха нужны мамам с младенцами, ведь именно новорожденные очень серьезно воспринимают малейшее изменение окружающей среды.

Постоянный анализ достоверности источников позволит постоянно контролировать качество и точность предоставляемых данных. Имея постоянно работающий компьютерный модуль, который анализирует прогноз и действительные данные в разных точках как городов, так и областей, позволит сказать, какой на данный момент сервис выходит на передний план, среди остальных. Ведь одноразовый анализ и выявление лидера по качеству не будет лучшим решением поставленной задачи.

На сегодняшний день практически невозможно найти подобные механизмы анализа погодных данных. «ОПогоде» - украинский агрегатор погодных данных. Этот сервис решает похожую задачу, которая была описана выше, но имеет ряд недостатков, которые я собираюсь исправить в своем решении. Одним из основных является предоставление данных о погоде в результате усреднения нескольких сервисов, тем самым создавая данные, основываясь как на надежных источниках, так и нет.

В силу высокой загруженности данного модуля отличным решением будет использование фреймворка Apache Storm, написанного на Java и Clojure. Именно с его помощью можно будет осуществить обработку этого колоссально огромного количества данных.

1. **STORM FRAMEWORK**

В 2011 году Twitter открыл, под лицензией Eclipse Public License, проект

распределенных вычислений Storm. Storm был создан в компании BackType и перешел к Twitter после покупки.

Storm это система, ориентированная на распределенную обработку больших потоков данных, аналогичная Apache Hadoop, но в реальном времени.

### **Ключевые особенности фреймворка**

К ключевым особенностям фреймворка Storm относят:

1. Масштабируемость. Задачи обработки распределяются по узлам кластера и потокам на каждом узле;
2. Гарантированная защита от потери данных;
3. Простота развертывания и спровождения;
4. Восстановление после сбоев. Если какой-либо из обработчиков отказывает, задачи переадресуются на другие обработчики;
5. Возможность написания компонентов не только на Java. Простой Multilang protocol с использованием JSON объектов. Есть готовые адаптеры для языков Python, Ruby и Fancy.

### **Элементы Storm**

Основными элементами фреймворка являются:

1. Tuple. Элемент представления данных. По умолчанию может содержать Long, Integer, Short, Byte, String, Double, Float, Boolean и byte[] поля. Пользовательские типы, используемые в Tuple должны быть сериализуемыми;
2. Stream. Последовательность из Tuple. Содержит схему именования полей в Tuple;
3. Spout. Поставщик данных для Stream. Получает данные из внешних источников, формирует из них Tuple и отправляет в Stream. Может отправлять Tuple в несколько разных Stream. Есть готовые для популярных систем обмена сообщениями: RabbitMQ / AMQP, Kestrel, JMS, Kafka;
4. Bolt. Обработчик данных. На вход поступают Tuple. На выход отправляет 0 или более Tuple;
5. Topology. Совокупность элементов с описанием их взаимосвязи. Аналог MapReduce job в Hadoop. В отличии от MapReduce job — не останавливается после исчерпания входного потока данных. Осуществляет транспорт Tuple между элементами Spout и Bolt. Может запускаться локально или загружаться в Storm кластер.
6. **Сохранность данных**

Разные классы задач предъявляют различные требования к надежности. Одно дело пропустить пару записей при подсчете статистики посещений, где счет идет на сотни тысяч и особая точность не нужна. И совсем другое — потерять, например, информацию о платеже клиента.

Рассмотрим о механизмы защиты от потери данных, которые реализованы в Storm.

1. **На уровне Spout**

Если нам не важно были ли ошибки при обработке Tuple, то Spout отправляет Tuple в SpoutOutputCollector посредством вызова метода:

|  |  |
| --- | --- |
|  | (1) |

Eсли необходимо узнать успешно ли обработался Tuple, то вызов будет выглядеть как:

|  |  |
| --- | --- |
|  | (2) |

где - это объект произвольного класса. В этом случае интерфейс ISpout предоставляет методы:

1. - будет вызван, если Tuple обработан успешно;
2. - будет вызван, если Tuple не обработан из-за какой-либо ошибки.

Стоит обратить внимание, что msgId — это id сообщения, с которым был вызван SpoutOutputCollector.emit. Методы nextTuple, ack и fail, вызываются в одном потоке и не требуют дополнительной синхронизации при обращении к полям Spout.

1. **На уровне Bolt**

Для того что бы Bolt мог информировать Storm о результатах обработки, он должен реализовывать интерфейс IRichBolt. Проще всего это сделать, унаследовав класс BaseRichBolt. Bolt информирует Storm o результатах своей работы посредством вызова методов класса OutputCollector в методе :

1. – обработка прошла успешно;
2. – обработка завершилась из-за какой-либо ошибки.

В Bolt'ах унаследованных от класса BaseBasicBolt, ack( Tuple tuple ) вызывается после выхода из метода execute автоматически.

1. **На уровне Tuple**

При обработке входного Tuple, Bolt может генерировать более одного выходного Tuple. Если Bolt вызвал emit( Tuple sourceTuple,Tuple resultTuple), то образуется направленный ациклический граф (орграф, в котором отсутствуют направленные циклы, то есть пути, начинающиеся и кончающиеся в одной и той же вершине) с вершиной в виде исходного Tuple и потомками в виде порожденных Tuple. Storm отслеживает ошибки процессинга всех узлов графа. В случае возникновения ошибки на любом уровне иерархии, Spout, породивший исходный Tuple, будет уведомлен вызовом fail.

Поскольку Storm является распределенной системой, Tuple могут передаваться с одного узла кластера на другой. В связи с этим Storm обеспечивает отслеживание таймаутов обработки. По умолчанию, весь граф должен быть обработан за 30 секунд, или Storm вызовет метод fail у породившего граф Spout'а. Таймаут можно изменить.

### **Транзакции в Storm**

Представим ситуацию, когда обработка завершилась с ошибкой. Вполне возможно, что потеряна связь с одним из узлов кластера или временно недоступна база данных. В этом случае, нельзя с уверенностью сказать, какие операции выполнились успешно, а какие — нет. Если все операции в цепочке повторно применимы (идемпотентны), например, установка флага, то можно просто перезапустить обработку. Если нет, то на помощь приходят механизмы транзакций Storm.

Когда говорят о характеристиках транзакций, тут же всплывает термин ACID:

1. Atomicity (атомарность). Все изменения, произведенные в системе на протяжении транзакции, либо применяются полностью, либо не применяются совсем;
2. Consistency (cогласованность). Транзакция переводит систему из одного непртиворечивого состояния в другое;
3. Isolation (изолированность). Параллельно выполняемые транзакции не оказывают влияние на результат работы друг друга;
4. Durability (надежность). Зафиксированные транзакцией изменения гарантированно остаются в системе.
   * 1. **Atomicity**

В Topology создается объект реализующий интерфейс State, инкапсулирующий работу с БД. Входные данные, поступающие в Spout, разбиваются на Tuple и собираются в пакеты (batch). Batch ассоцируется с уникальным transaction id. Tuple образующие batch могут обрабатываться параллельно.

В конце цепочки обработки, набор Tuple, относящихся к одной транзакции, передается в метод updateState класса, реализующего интерфейс StateUpdater, который и производит модификацию State. В случае успешного завершения, Spout получает уведомление об успехе обработки batch'a. В случае ошибки, Spout должен передать на обработку весь batch повторно.

Таким образом Storm гарантирует, что Batch будет зафиксирован в БД полностью и только один раз.

* + 1. **Isolation**

Storm гарантирует, что Batch'и передаются в StateUpdater строго последовательно, в порядке возрастания transaction id. То есть Batch номер 2 будет зафиксирован только после успешной фиксации Batch'а номер 1.

* + 1. **Реализация во фреймворке**

Spout с поддержкой транзакций должен реализовывать интерфейс ICommitterTridentSpout<TransactionMetadata>. TransactionMetadata — любой класс, содержит данные для генерации Batch'ей и генерации следующей транзакции: TxMeta.

|  |  |
| --- | --- |
| **public class** TxMeta {  **private int start**;  **private int count**;   **public** TxMeta(**int** start, **int** count) {  **this**.**start** = start;  **this**.**count** = count;  } *// Skipped getters* } | (3) |

Класс реализующий интерфейс ITridentSpout.BatchCoordinator<TransactionMetadata> инициализирует TransactionMetadata при создании транзакции и отвечает на запрос готовы ли данные для следующей транзакции. Создается в единственном экземпляре для каждой Topology.

|  |  |
| --- | --- |
| **static class** BCoordinator **implements** BatchCoordinator<TxMeta> {  **private static final int *TRANSACTION\_COUNT*** = 5;  **private static final int *TRANSACTION\_ELEMENT\_COUNT*** = 5;   *//TxMeta - метаданные предыдущей транзакции* @Override  **public** TxMeta initializeTransaction(**long** l, TxMeta txMeta) {  **if**(txMeta != **null**) {   System.***out***.println(String.format(**"Initializing transaction id: %08d, "** + **"start: %04d, count: %04d"**, l, txMeta.getStart() +  txMeta.getCount(), txMeta.getCount()));   **return new** TxMeta(txMeta.getStart() + txMeta.getCount(),  ***TRANSACTION\_ELEMENT\_COUNT***);  } **else** {  **return new** TxMeta(0, ***TRANSACTION\_ELEMENT\_COUNT***);  }  }   *// Готовы ли данные для следующей транзакции* @Override  **public boolean** isReady(**long** l) {  **if**(l <= ***TRANSACTION\_COUNT***) {  System.***out***.println(**"ISREADY "** + l);  **return true**;  }  **return false**;  } } | (4) |

Класс реализующий интерфейс ICommitterTridentSpout.Emitter формирует Batch. В случае ошибки в обработке Batch'a, формирует Batch повторно.

Важно — повторно сформированный Batch должен содержать точно такой же набор Tuple, что и оригинальный.

|  |  |
| --- | --- |
| **static class** BEmitter **implements** Emitter {  *// Формирует Batch по информации из TransactionMetadata* @Override  **public void** emitBatch(TransactionAttempt transactionAttempt,  Object coordinatorMeta,  TridentCollector tridentCollector) {   TxMeta txMeta = (TxMeta) coordinatorMeta;   System.***out***.println(**"Emitting transaction id: "** +  transactionAttempt.getTransactionId() + **" attempt:"** +  transactionAttempt.getAttemptId()  );  **for**(**int** i = 0; i < txMeta.getCount(); ++i) {  tridentCollector.emit(**new** Values(**"TRANS ["** +  transactionAttempt.getAttemptId() +  **"] ["** + (txMeta.getStart() + i) + **"]"**)  );  }  }   *// Транзакция успешно закоммичена в State* @Override  **public void** success(TransactionAttempt transactionAttempt) {  System.***out***.println(**"BEmitter:Transaction success id:"** +  transactionAttempt.getTransactionId());  }   *// Попытка коммита транзакции в State* @Override  **public void** commit(TransactionAttempt transactionAttempt) {  System.***out***.println(**"BEmitter:Transaction commit id:"** +  transactionAttempt.getTransactionId());  } } | (5) |

Класс реализующий интерфейс State в нашем случае драйвер БД: TxDatabase.

|  |  |
| --- | --- |
| **public class** TxDatabase **implements** State {  *// Вызывается при начале транзакции в БД* @Override  **public void** beginCommit(Long txId) {  System.***out***.println(**"beginCommit ["** + Thread.*currentThread*().getId() + **"] "** + txId);  }   *// Вызывается для коммита транзакции в БД* @Override  **public void** commit(Long txId) {  System.***out***.println(**"commit ["** + Thread.*currentThread*().getId() + **"] "** + txId);  } } | (6) |

Класс наследующий BaseStateUpdater<S extends State>, вносит изменения в State (БД): TxDatabaseUpdater

|  |  |
| --- | --- |
| **public class** TxDatabaseUpdater **extends** BaseStateUpdater<TxDatabase> {  **int count**;   *// Вносит изменения в БД* @Override  **public void** updateState(TxDatabase txDatabase,  List<TridentTuple> tridentTuples,  TridentCollector tridentCollector) {   *// Эмуляция сбоя транзакции* **if**(++**count** == 2) **throw new** FailedException(**"YYYY"**);   **for**(TridentTuple t: tridentTuples) {  System.***out***.println(**"Updating: "** + t.getString(0));  }  } } | (7) |

Класс реализующий интерфейс StateFactory, создает экземпляры State: TxDatabaseFactory.

Собираем все вместе TridentTransactionApp:

|  |  |
| --- | --- |
| **public class** TridentTransactionApp {  **public static void** main( String[] args ) **throws** Throwable  {  Logger.getRootLogger().setLevel(Level.ERROR);  *// Создаем топологию* TridentTopology tridentTopology = **new** TridentTopology(); *// Добавляем наш Spout* tridentTopology.newStream(**"TridentTxSpout"**, **new** TridentTxSpout()). *// Обработка Tuple пойдет параллельно - OpPrintout просто печатает записи* shuffle().each(**new** Fields(**"msg"**), **new** OpPrintout()).  parallelismHint(2). *// Сливаем результаты параллельной обработки в один поток* global(). *// Записываем изменения в State (БД)* partitionPersist(**new** TxDatabaseFactory(),  **new** Fields(**"msg"**), **new** TxDatabaseUpdater()); *// Skipped* LocalCluster cluster = **new** LocalCluster();  cluster.submitTopology(**"T2"**, config, tridentTopology.build());  Thread.*sleep*(1000\*100);  cluster.shutdown();  } } | (8) |

* + 1. **Вывод**

Транзакционные возможности Storm очень удобно использовать для передачи данных из одной системы в другую, когда требуется нетривиальная обработка. Например, одна система генерирует файлы, Storm их разделяет на записи, обрабатывает в параллельном режиме и складывает в БД. В случае ошибки обработки есть гарантия, что файл не будет удален и не будет обработан дважды.

1. **ПОСТАНОВКА ЗАДАЧИ И РЕАЛИЗАЦИЯ**

В качестве проекта было решено реализовать модуль контроля погодный веб-служб, основанных на протоколе HTTP, использующие REST архитектуру передачи данных. Были разработаны компоненты фреймворка Apache Storm согласно рисунку 2.1. Вместе они работают для сравнения текущих погодных данных с данными на час вперед.

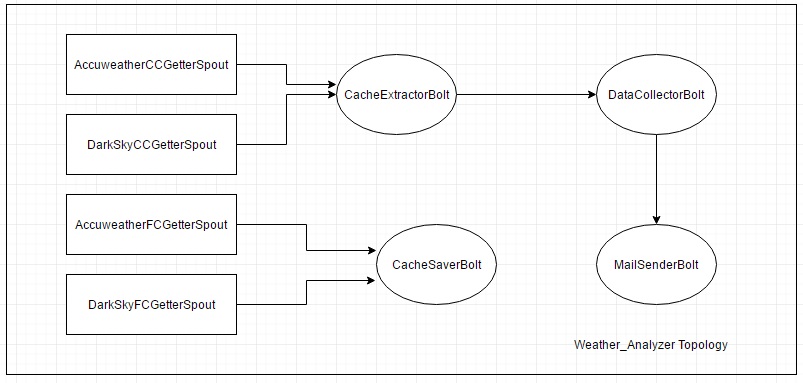


Рисунок 2.1 – Топология реализованного модуля

### **2.1 Получение погодных данных**

Для получения погодных данных были использованы следующие погодные веб-сервисы:

1. dataservice.accuweather.com;
2. api.darksky.net.

**2.1.1 Accuweather.com**

Accuweather.net предоставляет REST API с возможностью делать 1000 запросов в день бесплатно. На рисунке 2.2 и на рисунке 2.3 представлены результаты запросов на текущее состояние погоды и на час вперед в городе Минске.



Рисунок 2.2 – текущее состояние от Accuweather.com

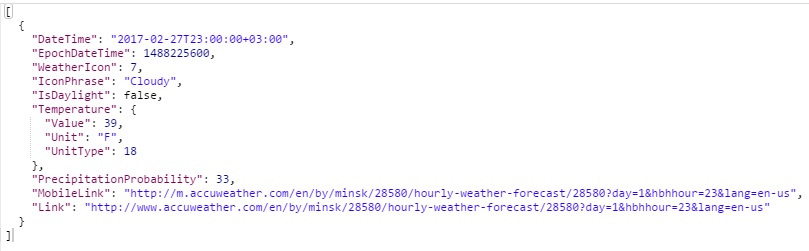


Рисунок 2.3 – прогноз на час от Accuweather.com

**2.1.1 Darksky.net**

Darksky.net также как и accuweather.com предоставляет REST API для получения результатов по 1000 запросам в день бесплатно. На рисунке 2.4 и рисунке 2.5 представлены ответы на запросы о текущих погодных данных и данных на час вперед.

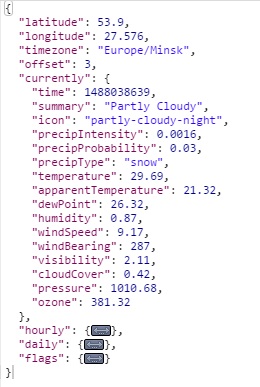


Рисунок 2.4 – текущее состояние от Darksky.net

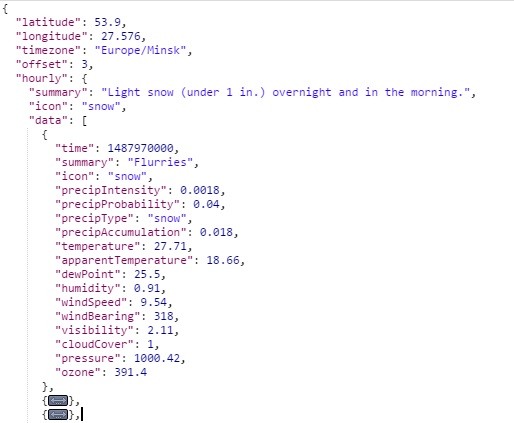


Рисунок 2.5 – прогноз от Darksky.net

**2.2 Хранение данных**

Для временного хранения данных было выбрано использование Ehcache – универсальной системы распределенного кэширования для Java приложений и сервлетов. Ehcache - небольшое приложение, которое реализует кэш как и динамический, так и дисковый. Данный кэш очень популярен среди проектов, написанных на Java, хоть изначально он был разработан для распределенных высоконагруженных систем, которые должны оперировать гигабайтами данных. Конфигурация, в свою очередь, очень проста, она показана на рисунке 2.6.



Рисунок 2.6 – конфигурация Ehcache

**2.3 Отчет**

После прохождения всех Bolt-ов представленного приложения, данные обрабатываются и отправляются в виде отчета на почту определенного человека. Пример части такого сообщения можно увидеть на рисуноке 2.7.

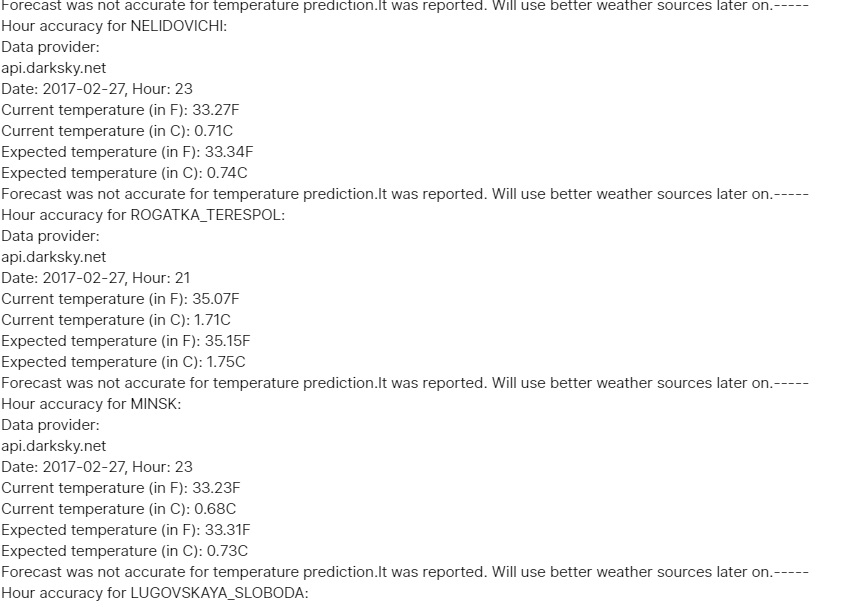


Рисунок 2.7 – часть отчета разработанного модуля

**ЗАКЛЮЧЕНИЕ**

В данной работе:

* изучен фреймворк Apache Storm и взаимодействие его компонентов;
* изучена работа кэша EhCache в распределенных вычислительных системах;
* рассмотрен спектр погодных веб-служб, предоставляющих бесплатный доступ к метеоданным;
* разработан высоконагруженный модуль получения, обработки, и предоставления отчета о данных, полученных от метеослужб.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Эккель Б. Философия Java. Библиотека программиста. – СПб.: Питер, 2001. – 880 с.: ил;
2. Буч Г. Объектно-ориентированный анализ и проектирование. 2-е изд. Пер. с англ. – М.: «Издательство Бином», 1998 – 560с.: ил;
3. Хабибуллин И. Ш. Создание распределённых приложений на Java 2. – СПб.: БХВ-Петербург, 2001 г. – 688с.: ил;
4. Java. Промышленное программирование : практ. пособие / И.Н. Блинов, В.С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с;
5. Шаблоны проектирования в Java / М. Гранд; Пер. с англ. С. Беликовой. – М.: Новое знание, 2004. – 559 с.: ил;
6. Материалы ресурса Habrahabr [Электрон. ресурс] Habrahabr Project – Режим доступа: https://habrahabr.ru/ – Дата доступа: 16.02.2017;
7. Материалы свободной энциклопедии Wikipedia [Электрон. ресурс] / Wikipedia Project – Режим доступа: http://wikipedia.org – Дата доступа: 02.02.2017.