Московский Государственный Университет имени М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Системного Программирования

Курсовая работа

Разработка и реализация прототипа системы потоковой обработки данных

Автор:
гр. 427
Овчинников Дмитрий

Научный руководитель:
д.т.н профессор Кузнецов Сергей Дмитриевич

Москва, 2015

Содержание

1 Постановка задачи

1.1 Актуальность

Поскольку рост информации коллосален в последнее время, ее нужно как-то быстро обрабатывать, и чем быстрее, тем лучше. Существуют такие технологии, как Hadoop, MapReduce, которые безусловно полезные, но довольно-таки медленные и непригодные для "Big data in realtime". Это сфера, где нужно обрабатывать много (потенциально бесконечно) данных за очень короткий промежуток времени, к таким относятся, например:

1. Системы анализа и мониторинга
2. Задачи оптимизации работы сетей и отдельно каждого устройства в сети
3. Обнаружение подозрительных действий на охраняемых территориях
4. Телефония, обслуживание АТС
5. Финансовый трейдинг

Это сфера ИТ еще нова и только набирает популярность, и до некоторого момента проблема быстрых вычислений решалась путем закупки дорогостоящего оборудования. Однако сейчас решение проблемы все больше и больше ложится на плечи программистов, которые меняют подходы к обработке информации.
Существует тенденция, когда все вычисления (которых слишком много) ставят "на поток т.е. выделяют отдельно систему, которая обрабатывает непрерывные данные, приходящие к ней и передает дальше, не задерживая. О разработке такой системы и пойдет речь в данной курсовой работе.

1.2 Общие требования

Планируется реализовать систему, ввиде отдельной библиотеки, которая представляла бы собой один кластер распределенной системы потоковой обработки данных. Для начала разработки необходимо обозначить набор требований, которым должна удовлетворять система:

1. Система не должна использовать работу с жестким диском, чтобы уменьшить время выполнения.

2. Система должна предоставлять инструменты для создания графов(топологий) вычислений.

3. Система должна иметь возможность пользоваться всеми ресурсами процессора, т.е. должна иметь возможно распараллеливания.

4. Система желательно быть кросс-платформенной

5. Система должна гарантировать полный проход графа вычислений.

6. Система не должна заботиться о том, как принимать данные и куда их потом передавать.
   Об этом должен позаботиться пользователь-программист, испольузющий эту библиотеку.

7. Система должна быть достаточно гибкой.
   Это означает, что пользователь, имел бы возможность настраивать множество параметров, таких как коэфициент распараллеливания, время жизни топологии, максимальную память, которые занимают данные в системе.

Так выглядят требования в общем виде. В процессе разработке они будут уточняться и поясняться. Сразу следует отметить то, что данная реализация - это прототип системы потоковой обработки данных, а значит в этой системе не будет реализовано то, как кластеры будут взаимодействовать между собой. Будут лишь некоторые замечания и предположения о том, как это будет проходить. А во всем остальном - должен получиться полноценный распараллеливаемый кластер потоковой обработки данных.

1.3 Выбор языка программирования

Есть множество замечательных языков программирования, у каждого есть свои преимущества и недостатки. Автор же остановился на языке Java, поскольку он кросс-платформенный и довольно-таки распространенный в тех сферах, где планируется использовать данную систему (Постановка задачи). Однако, у Java есть один большой

недостаток - программы на Java чуть ли не самые медленные среди всех языков программирования. Для такой системы закономерно выбирать какой-нибудь функциональный язык из семейства Lisp, поскольку обладают лаконичностью, компактностью и выразительностью. Хорошо подходят для этого Clojure, Scala или Groovy, которые реализованы на java-машине, т.е. обладают полной совместимостью с Java и обладают той же кросс-платформенностью. Но в силу ограниченности по времени написания данной работы, не было возможности выучить один из этих языков. Поэтому, жертвуя производительностью в пользу удобства написания, данная система будет полностью написана на Java.

2 Разработка прототипа

2.1 Общие сведения о схеме

Здесь, как говорится не будем изобретать велосипед заново (хотя, возможно, стоило бы и попытаться сделать все по-другому) и возьмем уже привычную схему для потоковой системы.

Основное место занимает топология(Topology), которая определяет, как именно данные перемещаются внутри программы. Элементами топологии являются воронки(spouts) и сита (bolts). Роль воронок заключается в том, чтобы извлекать данные извне, например, из БД, http, xml или из файлов. В воронках нет никаких вычислительных элементов, они всего лишь занимаются извлечением данных. Говоря терминами функционального программирования воронки - это “грязные”функции, т.е. они взаимодействуют с внешним миром, и их результат не всегда зависит от входных данных.

Смысл же сит(bolts) заключается в том, чтобы обрабатывать кортежи данных, которые им предназначены и передавать их дальше. Сита также могут взимодействовать с внешним миром - передавать результат работы топологии дальше.

На рис. Схема топологии показана примерная схема топологии (в зависимости от реализации она может различаться).

Итак все в этом приложении должно строиться вокруг топологии и для топологии, чтобы иметь возможность вычислить кортежи, которые будут подаваться на вход. С точки зрения пользователя этой информации будет достаточно для понимания, как работать с данной системой в целом.

2.2 Взаимодествие с пользователем

Один из первых вопросов, которые возникают при разработке любого приложения - как данные от пользователя буду попадать в программу? Каким образом пользователь будет взаимодействовать с программой? Здесь мы воспользуемся опытом уже известной и достаточно популярной системой потоковой обработки данных - Apache Storm. Кластер системы будет предоставляться как библиотека на языке java, который программист-пользователь, импортируя в свою программу, получит набор интерфейсов

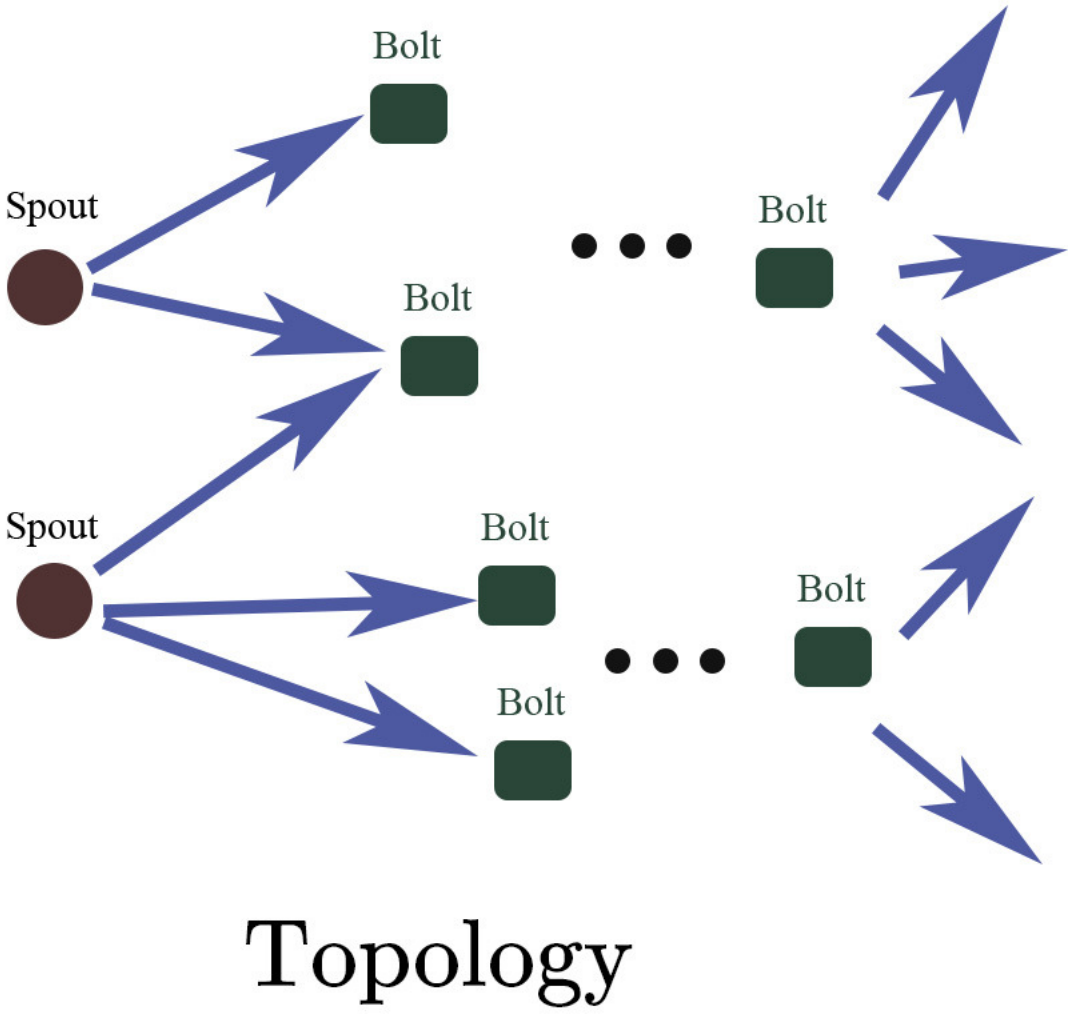

Рис. 1: Схема топологии

для коммуникации с кластером.
Основные возможности, которые доступны пользователю - это создание воронок и сит, наследующих определенный интерфейс, описанный автором. Предполагается, что пользователь будет добавлять сита и воронки в объект класса Topology - основной объект, которым будет манипулировать система при вычислении.
Пользователю также должна предоставляться возможно определять кортежи. Единственное требование на данном этапе, которое должно удовлетворяться кортежами - оно должно наследовать стандартный интерфейс кортежа, описанный в системе. На данном этапе этот интерфейс не имеет определенных методов, только лишь расширяет java-интерфейс Serializable. Это необходимо в случае распределения системы на несколько кластеров.

2.3 Логика приложения в общих чертах

За неимением опыта разработки больших и сложных систем, автор испытывает небольшую сложность в формальном описании системы и приносит свои извинения.
Конечно, выглядит не все так просто, как было написано в схемах общего вида, давайте же разберемся в этом. Первое, что нужно сказать, что топология(читать "граф") представлена стандартным образом, в виде списка воронок, сит и связей между ними. Пока здесь все ясно и не требует пояснений. Воронки и сита не должны храниться в одной "куче потому что воронки не могут принимать данные, а значит, стоило бы их отделить, для того, чтобы программа знала, кому передавать управление. Естественно, у каждой воронки и сита должно быть уникальное имя в виде строки. Это нужно для того, чтобы не было коллизий при хешировании (предполагается, что имена будут захешированы для более быстрого доступа).
Также необходимо обговорить еще один важный вопрос: где будут располагаться кортежи данных? Как они будут перемещаться по графу вычислений? Для кортежей будет выделена конкретная память, настраиваемая пользователем, у каждого кортежа будет имя - имя сита, для которого он предназначен. Поскольку одновременно могут несколько кортежей предназначаться к одному ситу, то хранить придется кортежи не по одному, а списками, в котором содержатся все кортежи предназначенные для конкретного сита с уникальным именем. Хранение будет в специальном классе, который

будет обеспечивать поиск по имени кортежа и добавление кортежа с именем, который только что его обработал. Такой подход был реализован по причине того, что эти методы буду вызываться внутри сит, где недоступен список со связями графа, т.е. сито не знает, кому передавать дальше. Хранение будет реализовано в виде отдельного класса Collector. Ссылка на этот класс будет передаваться при инициализации сита. Класс Collector берет на себя ответственность за хранение кортежей и дальнейшего продвижения по графу.

Теперь поговорим о том, как будет происходить выполнение операций определенных в сите или воронке. Во-первых, сито и воронка будут унаследовать стандартные интерфейсы, в которых будут находиться методы, предназначенные для описания действий над кортежами. Как уже понятно из написанного выше, сита сами будут извлекать кортежи из коллектора, и это ложится на плечи программиста-пользователя. Т.е. чтобы пропустить через сито один кортеж, достаточно выполнить известный метод из сита и он сам достанет и/или вернет в коллектор кортежи. После этого необходимо выполнить метод коллектора, который будет совершать подмену имени (т.к. сито не знает, кому передавать дальше, он возвращает кортежи со своим именем) на то, которое будет в паре с данным в списке связей графа.

С воронками все гораздо проще, они будут выдавать по одному кортежу в коллектор со своим именем, а коллектор сам позаботится, какое имя ему подставить вместо этого. Очевидно, коллектору не обязательно делать подмену после каждой обработки кортежа, можно накапливать некоторое количество, а потом обработать пачкой.

Теперь нужно упомянуть о том, как будет происходить распараллеливание. Оно будет проводиться стандартными классами Java. Предполагается, что будут отдельные классы-работники, которые будут принимать на себя топологию и делиться еще на 2 потока, одни будут выполнять методы из воронок, а другие из сит. Значит, при минимальной конфигурации, будет 4 потока для работы с топологией + 1 основной, который выполняет логику кластера. При необходимости, основной поток должен уметь завершить потомков (хотя в Java это не приветствуется - предполагается, что потоки-потомки сами завершают работу, но в нашем случае потомки работают бесконечно).

На данном этапе, автор считает, что с логикой все довольно-таки понятно и пора приступать к реализации.

3 Реализация прототипа

3.1 Более формальное описание требований

Итак, принимая все выше сказанное к сведению, мы получаем следующую картину. Нам необходимо реализовать набор интерфейсов и классов, которые будут в совокупности выполнять топологию, заданную пользователем.
Имеем на данный момент:

1. Предоставлять кластер (Cluster) в виде java-библиотеки для использования в других системах
   Кластер должен иметь методы:
   - запустить кластер
   - остановить кластер
   - добавить/удалить топологию из кластера
   - установить настройки кластера

2. Должен предоставляться интерфейс Топологии (StreamletTopology), в котором должны содержаться методы:
   - добавить/удалить воронку (Spout)
   - добавить/удалить сито (Bolt)
   (+ должна иметься одна стандартная реализация топологии (TopologyDefault))

3. Должен предоставляться интерфейс Воронки (StreamletSpout), которая играет роль источника потока в системе.
   Смысл воронки в том, чтобы получать данные извне (БД, http, xml, файлы) Должны содержаться методы
   - остановить поток.
   - инициализировать поток
   - получить следующий кортеж данных.

4. Должен предоставляться интерфейс "Сита"(StreamletBolt). Сито играет роль обработчика кортежей. Он может накапливать какую-либо информацию, фильтровать, отправлять в другие сита - в зависимости от того, что именно реализует потоковая система.
   Сито должно содержать следующие методы:
   - инициализировать сито.
   - обработать определенный кортеж данных.

5. Должен предоставляться интерфейс "Кортежа"(StreamletTuple): Кортеж должен унаследовать интерфейс Serializable И произвольное количество полей, определенное пользователем.

Сразу бросается в глаза, что перед каждым названием интерфейса фигурирует слово Streamlet. Это нечто иное, как автор решил назвать свою программу именно так. (Streamlet (англ.) - ручеек).

3.2 Интерфейсы

Итак, приступим к реализации интерфейсов. Пользуясь требованиями выше, получаем примерно вот интерфейс для сита:

```
 1 package main.intefaces;
 2
 3 /**
 4  * Created by dmitry on 25.10.15.
 5  */
 6 public interface StreamletBolt {
 7     String name = null;
 8     public void init();
 9     public void beforeTerminate();
10     public void exec();
11 }
```

Метод beforeTerminate() будет запускаться перед тем, как кластер остановит топологию, возможно нужно будет сохранить какие-либо данные или отправить результат куда-либо. Метод init() - инициализация сита, exec() - место нахождения самих операций сита, собственно, для чего все это и задумывалось.
Интерфейс воронки тоже выглядит очевидным образом:

```
 1 package main.intefaces;
 2
 3 /**
 4  * Created by dmitry on 25.10.15.
 5  */
 6 public interface StreamletSpout {
 7     String name = null;
 8     public void init();
 9     public void nextTuple();
10     public void beforeTerminate();
11 }
```

nextTuple() - загружает новый кортеж в коллектор. Остальные методы аналогичны, что и для сита.
Интерфейс кортежа вообще не несет особого смысла:

```
 1 package main.intefaces;
 2
 3 import java.io.Serializable;
 4
 5 /**
 6  * Created by dmitry on 22.10.15.
 7  */
 8 public interface StreamletTuple extends Serializable
{
```

```
 9
10 }
```

3.3 Классы

Отсылаясь на часть 2(Разработка прототипа), где описана основная логика программы, приступим к классам, которые ее реализуют. Начнем, пожалуй, с класса Collector, которые обслуживает все манипуляции с кортежами, а также хранит их:

```
 1 package main.implementations;
 2
 3 import main.intefaces.StreamletTuple;
 4
 5 import java.util.ArrayList;
 6 import java.util.HashMap;
 7 import java.util.List;
 8 import java.util.Map;
 9
10 /**
11  * Created by dmitry on 15.11.15.
12  */
13 public class Collector {
14     Map<String, List<StreamletTuple>> tuples;
15     Map<String, List<StreamletTuple>> tuplesResult;
16
17     public Collector(){
18         tuples = new HashMap<String, List<StreamletT
uple>>();
19         tuplesResult = new HashMap<String, List<Stre
amletTuple>>();
20     }
21
22     public StreamletTuple get(String name){
```

```
    23          List<StreamletTuple> list = this.tuples.get(
name);
    24          if (list != null) {
    25              if (list.isEmpty()) return null;
    26              StreamletTuple tuple = list.get(0);
    27              list.remove(tuple);
    28              return tuple;
    29          }
    30          return null;
    31      }
    32
    33      public void emit(String name, StreamletTuple tup
le){
    34          List<StreamletTuple> list = this.tuplesResul
t.get(name);
    35          if (list == null){
    36              list = new ArrayList<StreamletTuple>();
    37              list.add(tuple);
    38              this.tuplesResult.put(name, list);
    39          }
    40          else {
    41              list.add(tuple);
    42          }
    43      }
    44
    45      public void handleResult(String src, String dst)
{
    46          List<StreamletTuple> listSrc = this.tuplesRe
sult.get(src);
    47          if (listSrc != null) {
    48              List<StreamletTuple> listDst = this.tupl
es.get(dst);
```

```
49              if (listDst == null) listDst = new Array
List<StreamletTuple>();
50              listDst.addAll(listSrc);
51              listSrc.clear();
52              this.tuples.put(dst, listDst);
53          }
54      }
55 }
```

Здесь метод get() принимает на вход имя сита, чтобы достать для него кортеж, предназначенный ему. После чего кортеж удаляется из коллектора, поскольку уже обрабатывается соответствующим ситом.

Метод emit() служит для добавления кортежа в коллектор. Как уже не раз было сказано выше, сито, которое обрабатывает кортеж, не знает о дальнейшем предназначении кортежа, и поэтому передает коллектору его со своим именем, а задача коллектора подменить имя на соответствующее. handleResult() как раз и занимается тем, что меняет имя у кортежа и переносит их в основную кучу кортежей, готовых для обработки ситами.

Далее рассмотрим стандартную реализацию топологии и ее классу TopologyDefault:

```
 1 package main.implementations;
 2
 3 import main.intefaces.StreamletBolt;
 4 import main.intefaces.StreamletSpout;
 5 import main.intefaces.StreamletTopology;
 6
 7 import java.util.HashMap;
 8 import java.util.Map;
 9
10 /**
11  * Created by dmitry on 22.10.15.
12  */
```

```
    13 public class TopologyDefault implements StreamletTop
ology {
    14     private String topologyName;
    15     private Collector collector;
    16     private Map<String, StreamletSpout> spouts;
    17     private Map<String, StreamletBolt> bolts;
    18     private Map<String, String> associations;
    19
    20     public TopologyDefault(String topologyName) {
    21         this.topologyName = topologyName;
    22         this.spouts = new HashMap<String, StreamletS
pout>();
    23         this.bolts = new HashMap<String, StreamletBo
lt>();
    24         this.associations = new HashMap<String, Stri
ng>();
    25         this.collector = new Collector();
    26     }
    27
    28     public void setBolt(String name, StreamletBolt b
olt){
    29         this.bolts.put(name, bolt);
    30     }
    31
    32     public void setSpout(String name, StreamletSpout
 spout){
    33         this.spouts.put(name, spout);
    34     }
    35
    36     public void setAssociation(String node1, String
node2){
    37         this.associations.put(node1, node2);
```

```
38     }
39
40     public Collector getCollector() {
41         return collector;
42     }
43 }
```

Здесь нет ничего необычного, все строго так, как описано в логике программы(Разработка прототипа). Класс TopologyDefault по сути ничего не выполняет, а только хранит набор воронок, сит и связей между ними. Класс также хранит ссылку на собственный коллектор. Подразумевается, что для каждой топологии будет свой коллектор. Такое решение было принято в силу безопасности и уменьшения коллизий между разными топологиями.

Следующим к показу представлен класс-работник Worker:

```
 1 package main.implementations;
 2
 3 import main.intefaces.StreamletBolt;
 4 import main.intefaces.StreamletSpout;
 5
 6 import java.util.Map;
 7
 8 /**
 9  * Created by dmitry on 27.10.15.
10  */
11 public class Worker extends Thread {
12
13     private TopologyDefault topologyDefault;
14
15     public Worker(TopologyDefault topologyDefault){
16         this.topologyDefault = topologyDefault;
17     }
18
```

```
    19     public Worker(){}
    20
    21     public void setTopologyDefault(TopologyDefault t
opologyDefault) {
    22         this.topologyDefault = topologyDefault;
    23     }
    24
    25     @Override
    26     public void run(){
    27         /**
    28          * Раздвоить на 2 потока
    29          * 1 поток обслуживает во
онки
    30          * 2 поток обслуживает си
а
    31          */
    32         while (true) {
    33             for (Map.Entry<String, StreamletSpout> e
ntry : topologyDefault.getSpouts().entrySet())
    34                 entry.getValue().nextTuple();
    35             for (Map.Entry<String, StreamletBolt> en
try : topologyDefault.getBolts().entrySet())
    36                 entry.getValue().exec();
    37             for (Map.Entry<String, String> entry : t
opologyDefault.getAssociations().entrySet())
    38                 topologyDefault.getCollector().handl
eResult(entry.getKey(), entry.getKey());
    39         }
    40     }
    41 }
```

Класс наследует java-класс Thread из стандартной бибилотеки. Здесь необходимо было

переопределить метод run(), который работает при вызове метода start(). Т.е. если мы хотим запустить какой-либо метод в отдельном потоке, нам необходимо определить этот метод в методе run(), а после чего вызвать start() у этого класса. Как и было описано выше классы-работники будут запускаться в отдельном потоке и работать бесконечно, пока кластер не остановит их.
Ну и непосредственно сам кластер:

```
 1 package main.implementations;
 2
 3 import java.util.ArrayList;
 4 import java.util.List;
 5
 6 /**
 7  * Created by dmitry on 1.12.15.
 8  */
 9 public class Cluster {
10
11     private List<TopologyDefault> topologies;
12     private List<Worker> workers;
13     private Settings settings;
14
15     public  Cluster(Settings settings){
16         this.collector = new Collector();
17         this.topologies = new ArrayList<TopologyDefa
ult>();
18         this.settings = settings;
19         this.workers = new ArrayList<Worker>();
20         for (int i = 0; i < settings.getWorkerNumber
(); i++)
21             this.workers.add(new Worker());
22     }
23
24     public void deployTopology(TopologyDefault topol
```

```
ogyDefault){
    25          this.topologies.add(topologyDefault);
    26      }
    27
    28      public void setWorkerNumber(int workerNumber) {
    29          this.settings.setWorkerNumber(workerNumber);
    30      }
    31
    32      public void startCluster(){
    33          for (int i = 0; i < settings.getWorkerNumber
(); i++) {
    34              for (int j = 0; j < topologies.size(); j
++) {
    35                  workers.get(i).setTopologyDefault(to
pologies.get(j));
    36                  workers.get(i).start();
    37              }
    38          }
    39      }
    40
    41      public void stopCluster(){
    42          for (Worker worker : workers)
    43              worker.stop();
    44      }
    45
    46 }
```

У кластера есть несколько методов, которые также были оговорены выше, кластер хранит несколько топологий, настройки и список работников, готовых взяться за дело. При вызове метода startCluster() начинается работа, при stopCluater() - заканчивается. Класс настроек не представляет собой ценного материала, поскольку в данной версии программы там хранится только количество работников.

На этом реализация прототипа потоковой системы обработки данных можно считать

законченной. Осталось лишь все упаковать в jar архив для возможно удобного подключения к другим проектам. Для этого достаточно воспользоваться одним из множества замечательных инструментов таких, как Apache Ant, Gradle, Maven.

Заключение

И в заключении хотелось сказать, что получилась довольно-таки несложная система - инструмент, упрощающий работу с потоками данных. Все требования, которые были описаны в частях Постановка задачи, Разработка прототипа выполнены успешно. Конечно, необходимо протестировать разработанную систему на различные тест-кейсы, например, на отказоустойчивость и/или покрытия всех состояний системы, но это тема для отдельной работы.

Также хотелось заметить, что приложение гарантирует полное прохождение кортежем всех сит. Это следует из реализации - кластер будет работать до тех пор, пока в коллекторе есть кортежи.

Еще одно замечание по поводу работы: приложение не гарантирует последовательную обработку кортежей - кортежи попадают к работникам в произвольном порядке. Для того, чтобы реализовать последовательность кортежей, необходимо еще больше расширить функционал коллектора, кластера и класса-работника.

Для распределенной системы из кластеров можно применить неплохой опыт распределенной системы Apache Hadoop, которую с таким же успехом применяют в популярной системе потоковой обработки данных Apache Storm. Разумеется, данное приложение нельзя выпускать в продакшн, поскольку она писалась одним человеком.