**HU Extension School E-185 Big Data Analytics**

**Handed out: 04/09/2013 Due by 10 PM on Thursday, 05/09/2013**

**Final Project, Assignment 11 - Report**

**Real time stream processing using Kafka and Storm**

This document covers the installation process and the preparation of the proposed demo project using Kafka and Storm. These steps must be followed in order to install the frameworks on a single machine and execute the demo code provided.

**Installation**

All the installation steps described here were performed on a CentOS 6.4 machine with CDH 4.2.1 (YARN) installed, in accordance with the process that was implemented in class on Assignment 05. In fact, it was precisely the same Virtual Machine I had used for all Assignments after Assignment 05.

This project used 3 main code bases in order to be implemented, each with their own requirements:

* Kafka version 0.7.2 (<http://kafka.apache.org/index.html>)
* Storm version 0.9.0 (<https://github.com/nathanmarz/storm> and <http://storm-project.net/>)
* Storm-Kafka (<https://github.com/nathanmarz/storm-contrib/tree/master/storm-kafka>)

The specific versions of Kafka and Storm were selected because those were the versions that supported the use of the Storm-Kafka contrib package. It is a pity that this report is not able to convene the level of frustration and trial-and-error necessary to get all of this to work together, but if this report can help someone not lose days of their lives cursing developers I will feel vindicated.

**Installing Kafka**

In order to install Kafka, we followed the steps in <http://kafka.apache.org/07/quickstart.html>, as detailed below.

First of all, we need to download and unpack the source files of the tool

**[cloudera@centos-e185 ~]$ wget http://ftp.unicamp.br/pub/apache/incubator/kafka/kafka-0.7.2-incubating/kafka-0.7.2-incubating-src.tgz**

(...)

2013-04-29 14:30:51 (1.17 MB/s) - “kafka-0.7.2-incubating-src.tgz” saved [1595334/1595334]

**[cloudera@centos-e185 ~]$ tar -xzf kafka-0.7.2-incubating-src.tgz**

**[cloudera@centos-e185 ~]$ mv kafka-0.7.2-incubating-src kafka**

**[cloudera@centos-e185 ~]$ cd kafka**

The next step is to update the local packages and compile the tool (which is partially written in Scala) using sbt, the Scala Build Tool. The output of the ./sbt package command is a JAR package.

**[cloudera@centos-e185 kafka]$ ./sbt update**

Getting Scala 2.7.7 ...

(...)

[info] == Kafka / update ==

[success] Successful.

[info]

[info] Total time: 151 s, completed Apr 29, 2013 2:35:39 PM

[info]

[info] Total session time: 192 s, completed Apr 29, 2013 2:35:39 PM

[success] Build completed successfully.

**[cloudera@centos-e185 kafka]$ ./sbt package**

[info] Building project Kafka 0.7.2 against Scala 2.8.0

[info] using KafkaProject with sbt 0.7.5 and Scala 2.7.7

(...)

[info] == Kafka / package ==

[success] Successful.

[info]

[info] Total time: 80 s, completed Apr 29, 2013 2:40:30 PM

[info]

[info] Total session time: 80 s, completed Apr 29, 2013 2:40:30 PM

[success] Build completed successfully.

Now that the tool has allegedly been build, I test the framework out according to the steps described in the QuickStart guide.

On one session window, we run our Zookeeper server:

**[cloudera@centos-e185 kafka]$ bin/zookeeper-server-start.sh config/zookeeper.properties**

[2013-04-29 14:54:23,225] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)

[2013-04-29 14:54:23,226] WARN Either no config or no quorum defined in config, running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)

[2013-04-29 14:54:23,239] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)

(..)

[2013-04-29 14:54:23,271] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxn)

[2013-04-29 14:54:23,282] INFO Snapshotting: 0 (org.apache.zookeeper.server.persistence.FileTxnSnapLog)

And on another, we start the Kafka server:

**[cloudera@centos-e185 kafka]$ bin/kafka-server-start.sh config/server.properties**

[2013-04-29 14:57:13,600] INFO Starting Kafka server... (kafka.server.KafkaServer)

(...)

[2013-04-29 14:57:13,810] INFO Registering broker /brokers/ids/0 succeeded with id:0,creatorId:127.0.0.1-1367272633783,host:127.0.0.1,port:9092 (kafka.server.KafkaZooKeeper)

[2013-04-29 14:57:13,822] INFO Starting log flusher every 1000 ms with the following overrides Map() (kafka.log.LogManager)

[2013-04-29 14:57:13,823] INFO Kafka server started. (kafka.server.KafkaServer)

Now, we try to send some messages through the structure by creating a sample Producer. Note that I have selected the messaging topic as ***test***, which will need to match the topic I use when I instance my Consumer application.

**[cloudera@centos-e185 kafka]$ bin/kafka-console-producer.sh --zookeeper localhost:2181 --topic test**

[2013-04-29 14:59:15,692] INFO Starting ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)

(...)

2013-04-29 14:59:15,717] INFO Session establishment complete on server localhost/127.0.0.1:2181, sessionid = 0x13e57ca1bb80001, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)

[2013-04-29 14:59:15,718] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)

[2013-04-29 14:59:15,800] INFO Creating async producer for broker id = 0 at 127.0.0.1:9092 (kafka.producer.ProducerPool)

**This is a test of the Kafka infrastructure**

[2013-04-29 14:59:36,859] INFO Connected to 127.0.0.1:9092 for producing (kafka.producer.SyncProducer)

**Really, this is a test. No joke here**

Finally, when I create a Consumer application on the same topic, it receives the messages I sent through the Producer.

**[cloudera@centos-e185 kafka]$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning**

[2013-04-29 15:02:40,089] INFO console-consumer-23974\_centos-e185.aperture-1367272960040-1ef7d8a8 Connecting to zookeeper instance at localhost:2181 (kafka.consumer.ZookeeperConsumerConnector)

(...)

[2013-04-29 15:02:40,337] INFO FetchRunnable-0 start fetching topic: test part: 0 offset: 0 from 127.0.0.1:9092 (kafka.consumer.FetcherRunnable)

This is a test of the Kafka infrastructure

Really, this is a test. No joke here

With this test completed, we are satisfied that the framework works, and we go to our next step, which is installing and testing Storm.

**Installing Storm**

Using a link available from the project website (<http://storm-project.net/downloads.html>), we download the Storm framework package witn version 0.9.0. Although this version is not listed as stable, and only as a development version, this is the one we will require for compatibility with the Storm-Kafka package.

**[cloudera@centos-e185 ~]$ wget https://dl.dropbox.com/u/133901206/storm-0.9.0-wip16.zip**

--2013-04-30 12:05:46-- https://dl.dropbox.com/u/133901206/storm-0.9.0-wip16.zip

(...)

2013-04-30 12:06:02 (1.18 MB/s) - “storm-0.9.0-wip16.zip” saved [16206479/16206479]

**[cloudera@centos-e185 ~]$ unzip storm-0.9.0-wip16.zip**

Archive: storm-0.9.0-wip16.zip

creating: storm-0.9.0-wip16/

creating: storm-0.9.0-wip16/bin/

inflating: storm-0.9.0-wip16/bin/build\_release.sh

(...)

extracting: storm-0.9.0-wip16/RELEASE

inflating: storm-0.9.0-wip16/storm-0.9.0-wip16.jar

So in order to check basic Storm functionality, we download the storm-starter project from github (at <https://github.com/nathanmarz/storm-starter>) and compile it to test out some sample usages of Storm.

All Storm development is Clojure based, so we need to install the Clojure build tool, called Leiningen (or lein) in order to compile and run the sample projects that are available. Initially, we create a bin directory and download the lein utility to it, from its project page on Github.

**[cloudera@centos-e185 ~]$ cd bin**

**[cloudera@centos-e185 bin]$ wget https://raw.github.com/technomancy/leiningen/stable/bin/lein**

(...)

2013-04-29 15:31:43 (2.25 MB/s) - “lein” saved [10706/10706]

**[cloudera@centos-e185 bin]$ chmod 755 lein**

**[cloudera@centos-e185 bin]$ ls**

lein

We put Leiningen and Storm binary scripts on the path so that we can access them from our project folders. The Maven and Mahout paths on the file below are not important, and refer to old projects executed on this machine.

**[cloudera@centos-e185 ~]$ sudo vi /etc/profile.d/java.sh**

[sudo] password for cloudera:

**[cloudera@centos-e185 ~]$ cat /etc/profile.d/java.sh**

## Exporting JAVA\_HOME and an updated PATH to all user on the machine

export JAVA\_HOME=/usr/java/jdk1.6.0\_31

export MAHOUT\_HOME=/usr/lib/mahout

export HADOOP\_HOME=/usr/lib/hadoop

export PATH=/home/cloudera/bin:/home/cloudera/storm-0.9.0-wip16/bin:$JAVA\_HOME/bin:$PATH:$MAVEN\_HOME/bin:$MAHOUT\_HOME

With the build tool downloaded and on the executable path, we proceed to download and compile the storm-starter package to try it out.

**[cloudera@centos-e185 ~]$ git clone https://github.com/nathanmarz/storm-starter.git**

Initialized empty Git repository in /home/cloudera/storm-starter/.git/

remote: Counting objects: 534, done.

remote: Compressing objects: 100% (279/279), done.

remote: Total 534 (delta 196), reused 479 (delta 155)

Receiving objects: 100% (534/534), 104.77 KiB | 114 KiB/s, done.

Resolving deltas: 100% (196/196), done.

**[cloudera@centos-e185 ~]$ cd storm-starter/**

**[cloudera@centos-e185 storm-starter]$ lein deps**

Downloading Leiningen to /home/cloudera/.lein/self-installs/leiningen-2.1.3-standalone.jar now...

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 13.1M 100 13.1M 0 0 433k 0 0:00:31 0:00:31 --:--:-- 439k

Retrieving commons-collections/commons-collections/3.2.1/commons-collections-3.2.1.pom from central

Retrieving org/apache/commons/commons-parent/9/commons-parent-9.pom from central

(...)

Retrieving storm/carbonite/1.5.0/carbonite-1.5.0.jar from clojars

Retrieving storm/tools.cli/0.2.2/tools.cli-0.2.2.jar from clojars

Retrieving storm/jgrapht/0.8.3/jgrapht-0.8.3.jar from clojars

**[cloudera@centos-e185 storm-starter]$ lein compile**

Compiling 28 source files to /home/cloudera/storm-starter/target/classes

Note: Some input files use or override a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

Note: Some input files use unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

Compiling storm.starter.clj.word-count

In order to test out our Storm deployment, we choose a simple example from the project called ExclamationTopology, which consists of the following simple topology:

* A single Spout node called **word**, which generates names (such as "nathan" or "mike") randomly, and emits them to the other nodes;
* A Bolt called **exclaim1**, which consumes emissions from **word**, appends "!!!" to the end of the word and emits the result;
* And a Bolt called **exclaim2**, which consumes emissions from **exclaim1** and does the same as the other one, appending "!!!" to the result.

This is expressed in the code for the class as:

46 TopologyBuilder builder = new TopologyBuilder();

47

48 builder.setSpout("word", new TestWordSpout(), 10);

49 builder.setBolt("exclaim1", new ExclamationBolt(), 3)

50 .shuffleGrouping("word");

51 builder.setBolt("exclaim2", new ExclamationBolt(), 2)

52 .shuffleGrouping("exclaim1");

53

Testing ExclamationTopology out, we get the following result:

**[cloudera@centos-e185 storm-starter]$ java -cp $(lein classpath) storm.starter.ExclamationTopology**

0 [main] INFO backtype.storm.zookeeper - Starting inprocess zookeeper at port 2000 and dir /tmp/dafec8b4-b148-4e0f-b22c-0d4fa4c6c5cf

(...)

2219 [Thread-32] INFO backtype.storm.daemon.task - Emitting: word default [mike]

2219 [Thread-16] INFO backtype.storm.daemon.executor - Processing received message source: word:10, stream: default, id: {}, [mike]

2219 [Thread-16] INFO backtype.storm.daemon.task - Emitting: exclaim1 default [mike!!!]

2220 [Thread-22] INFO backtype.storm.daemon.executor - Processing received message source: exclaim1:2, stream: default, id: {}, [mike!!!]

2220 [Thread-22] INFO backtype.storm.daemon.task - Emitting: exclaim2 default [mike!!!!!!]

(...)

2228 [Thread-34] INFO backtype.storm.daemon.task - Emitting: word default [bertels]

2228 [Thread-16] INFO backtype.storm.daemon.executor - Processing received message source: word:11, stream: default, id: {}, [bertels]

2229 [Thread-16] INFO backtype.storm.daemon.task - Emitting: exclaim1 default [bertels!!!]

2229 [Thread-24] INFO backtype.storm.daemon.executor - Processing received message source: exclaim1:2, stream: default, id: {}, [bertels!!!]

2229 [Thread-24] INFO backtype.storm.daemon.task - Emitting: exclaim2 default [bertels!!!!!!]

(...)

2243 [Thread-36] INFO backtype.storm.daemon.task - Emitting: word default [nathan]

2244 [Thread-20] INFO backtype.storm.daemon.executor - Processing received message source: word:12, stream: default, id: {}, [nathan]

2244 [Thread-20] INFO backtype.storm.daemon.task - Emitting: exclaim1 default [nathan!!!]

2244 [Thread-22] INFO backtype.storm.daemon.executor - Processing received message source: exclaim1:4, stream: default, id: {}, [nathan!!!]

2244 [Thread-22] INFO backtype.storm.daemon.task - Emitting: exclaim2 default [nathan!!!!!!]

(...)

10998 [main] INFO backtype.storm.testing - Shutting down in process zookeeper

10998 [main] INFO backtype.storm.testing - Done shutting down in process zookeeper

10999 [main] INFO backtype.storm.testing - Deleting temporary path /tmp/62a14212-4bb4-489e-83e5-b3b0c10a6876

10999 [main] INFO backtype.storm.testing - Deleting temporary path /tmp/dafec8b4-b148-4e0f-b22c-0d4fa4c6c5cf

10999 [main] INFO backtype.storm.testing - Deleting temporary path /tmp/3068b931-ff21-48a7-9925-6a3225ce5bc4

11010 [main] INFO backtype.storm.testing - Deleting temporary path /tmp/febbb918-a9ff-4576-be9c-542fa1da0218

Notice the groups of debug logs that have been singled out on the output above, where we can see the evolution of the data as it passes through the bolts. We can consider this a sucessful test.

Installing storm-kafka

Finally, we move on to install the storm-kafka binding, which is greatly simplified by the fact that we have already loaded and installed lein in the machine. We proceed to download the project from Github and compile it.

**[cloudera@centos-e185 ~]$ git clone https://github.com/nathanmarz/storm-contrib.git**

Initialized empty Git repository in /home/cloudera/storm-contrib/.git/

remote: Counting objects: 2295, done.

remote: Compressing objects: 100% (1043/1043), done.

remote: Total 2295 (delta 757), reused 2151 (delta 653)

Receiving objects: 100% (2295/2295), 430.23 KiB | 263 KiB/s, done.

Resolving deltas: 100% (757/757), done.

**[cloudera@centos-e185 ~]$ cd storm-contrib/storm-kafka/**

**[cloudera@centos-e185 storm-kafka]$ ls**

project.clj README.markdown src

**[cloudera@centos-e185 storm-kafka]$ lein deps**

Retrieving org/scala-lang/scala-library/2.9.2/scala-library-2.9.2.pom from central

Retrieving com/twitter/kafka\_2.9.2/0.7.0/kafka\_2.9.2-0.7.0.pom from conjars

(...)

Retrieving storm/storm/0.9.0-wip15/storm-0.9.0-wip15.jar from clojars

Retrieving com/twitter/kafka\_2.9.2/0.7.0/kafka\_2.9.2-0.7.0.jar from conjars

**[cloudera@centos-e185 storm-kafka]$ lein jar**

Compiling 26 source files to /home/cloudera/storm-contrib/storm-kafka/target/classes

Note: Some input files use unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

Created /home/cloudera/storm-contrib/storm-kafka/target/storm-kafka-0.9.0-wip16b-scala292.jar

Sadly, the storm-kafka package does not have built-in tests, so we have to take its word that we manage to compile it successfully. Regardless, with all the JARs created, we copy them to our host machine in order to start the development of sample applications with Eclipse.

In the end, these were the JAR files we had to add to our project in order to be able to compile it and make it work. They are included in the directory JARs-Include on the Demo folder.

**aperture-2:Demo alexcp$ cd JARs-Include/**

**aperture-2:JARs-Include alexcp$ ls -la**

total 24960

drwxr-xr-x 7 alexcp staff 238 May 9 14:41 .

drwxr-xr-x 5 alexcp staff 170 May 9 14:38 ..

-rw-r--r-- 1 alexcp staff 109043 Jan 11 20:45 commons-io-1.4.jar

-rw-r--r-- 1 alexcp staff 1312106 Apr 29 18:40 kafka-0.7.2.jar

-rw-r--r-- 1 alexcp staff 6160791 Jul 14 2010 scala-library.jar

-rw-r--r-- 1 alexcp staff 5088898 May 9 14:37 storm-0.9.0-wip16.jar

-rw-r--r-- 1 alexcp staff 97727 Apr 29 20:17 storm-kafka-0.9.0-wip16b-scala292.jar

Also, in order to run the Demo later, we need to make sure that the storm script has access to all the JAR files it needs, specially ones from Kafka. After a little probing, we come up with a list of JARs that should be added to the classpath and we add them to the Storm root folder.

**[cloudera@centos-e185 ~]$ cat kafka/bin/kafka-run-class.sh | grep file**

# contributor license agreements. See the NOTICE file distributed with

# The ASF licenses this file to You under the Apache License, Version 2.0

# (the "License"); you may not use this file except in compliance with

for file in $base\_dir/project/boot/scala-2.8.0/lib/\*.jar;

CLASSPATH=$CLASSPATH:$file

for file in $base\_dir/core/target/scala\_2.8.0/\*.jar;

CLASSPATH=$CLASSPATH:$file

for file in $base\_dir/core/lib/\*.jar;

CLASSPATH=$CLASSPATH:$file

for file in $base\_dir/perf/target/scala\_2.8.0/kafka\*.jar;

CLASSPATH=$CLASSPATH:$file

for file in $base\_dir/core/lib\_managed/scala\_2.8.0/compile/\*.jar;

if [ ${file##\*/} != "sbt-launch.jar" ]; then

CLASSPATH=$CLASSPATH:$file

KAFKA\_OPTS="-Xmx512M -server -Dlog4j.configuration=file:$base\_dir/config/log4j.properties"

**[cloudera@centos-e185 ~]$ ls -l kafka/core/target/scala\_2.8.0/\*.jar**

-rw-rw-r--. 1 cloudera cloudera 1312106 Apr 29 14:40 kafka/core/target/scala\_2.8.0/kafka-0.7.2.jar

**[cloudera@centos-e185 ~]$ ls -l kafka/core/lib\_managed/scala\_2.8.0/compile/\*.jar**

-rw-rw-r--. 1 cloudera cloudera 53244 Dec 7 2009 kafka/core/lib\_managed/scala\_2.8.0/compile/jopt-simple-3.2.jar

-rw-rw-r--. 1 cloudera cloudera 391834 Aug 30 2007 kafka/core/lib\_managed/scala\_2.8.0/compile/log4j-1.2.15.jar

-rw-rw-r--. 1 cloudera cloudera 995968 Oct 4 2011 kafka/core/lib\_managed/scala\_2.8.0/compile/snappy-java-1.0.4.1.jar

-rw-rw-r--. 1 cloudera cloudera 62913 Apr 13 2011 kafka/core/lib\_managed/scala\_2.8.0/compile/zkclient-0.1.jar

-rw-rw-r--. 1 cloudera cloudera 604182 Nov 29 2011 kafka/core/lib\_managed/scala\_2.8.0/compile/zookeeper-3.3.4.jar

We copy all of these Kafka and dependencies JAR files, our compiled storm-kafka JAR and the Scala default library JAR, which is also used by Kafka, to the Storm root directory, which is part of its classpath.

**[cloudera@centos-e185 ~]$ ls -l classpath/**

total 9468

-rw-rw-r--. 1 cloudera cloudera 53244 Apr 30 12:10 jopt-simple-3.2.jar

-rw-rw-r--. 1 cloudera cloudera 1312106 Apr 30 12:10 kafka-0.7.2.jar

-rw-rw-r--. 1 cloudera cloudera 391834 Apr 30 12:10 log4j-1.2.15.jar

-rw-rw-r--. 1 cloudera cloudera 6160791 Apr 30 12:10 scala-library.jar

-rw-rw-r--. 1 cloudera cloudera 995968 Apr 30 12:10 snappy-java-1.0.4.1.jar

-rw-rw-r--. 1 cloudera cloudera 97727 Apr 30 12:10 storm-kafka-0.9.0-wip16b-scala292.jar

-rw-rw-r--. 1 cloudera cloudera 62913 Apr 30 12:10 zkclient-0.1.jar

-rw-rw-r--. 1 cloudera cloudera 604182 Apr 30 12:10 zookeeper-3.3.4.jar

**[cloudera@centos-e185 ~]$ cp classpath/\* storm-0.9.0-wip16/**

**[cloudera@centos-e185 ~]$ ls -la storm-0.9.0-wip16/\*.jar**

-rw-rw-r--. 1 cloudera cloudera 53244 Apr 30 12:29 storm-0.9.0-wip16/jopt-simple-3.2.jar

-rw-rw-r--. 1 cloudera cloudera 1312106 Apr 30 12:29 storm-0.9.0-wip16/kafka-0.7.2.jar

-rw-rw-r--. 1 cloudera cloudera 391834 Apr 30 12:29 storm-0.9.0-wip16/log4j-1.2.15.jar

-rw-rw-r--. 1 cloudera cloudera 6160791 Apr 30 12:29 storm-0.9.0-wip16/scala-library.jar

-rw-rw-r--. 1 cloudera cloudera 995968 Apr 30 12:29 storm-0.9.0-wip16/snappy-java-1.0.4.1.jar

-rw-r--r--. 1 cloudera cloudera 5088898 Feb 18 18:56 storm-0.9.0-wip16/storm-0.9.0-wip16.jar

-rw-rw-r--. 1 cloudera cloudera 97727 Apr 30 12:29 storm-0.9.0-wip16/storm-kafka-0.9.0-wip16b-scala292.jar

-rw-rw-r--. 1 cloudera cloudera 62913 Apr 30 12:29 storm-0.9.0-wip16/zkclient-0.1.jar

-rw-rw-r--. 1 cloudera cloudera 604182 Apr 30 12:29 storm-0.9.0-wip16/zookeeper-3.3.4.jar

**Demo - Very Complicated WordCount**

In order to demonstrate the capabilities of the tools, we have decided to put a spin on one of the most classical examples of distributed data processing - the Word Count problem.

All the processing on this word count example will be processed by Storm Bolts, which will be responsible for splitting the sentences, combining the word counts, and outputting the results to the console and files in a local directory.

The data intake, however, will be performed by reading messages from Kafka. We designed a simple class that takes a text file name as parameter, instances a Kafka Producer and pushes it as messages as a specific Kafka topic a number of times that might be desired.

On the other end of the wire, a KafkaSpout (original name, no?) connects to the same topic and emits the message to the Bolts we described above.

**Kafka Producer - KafkaFileStreamer.java**

The overall structure of the **KafkaFileStreamer** is very simple, as it only instances a Producer and send the contents of the file down the message stream on the topic we selected.

String topic = args[0];

String fileName = args[1];

int times = Integer.parseInt(args[2]);

Producer<String, String> producer = initProducer();

List<String> lines = FileUtils.readLines(new File(fileName));

for (int i = 0; i<times; i++) {

System.out.println("Sending " + fileName + " to Kafka system. Iteration: " + (i+1));

sendMessage(producer, topic, lines);

}

System.out.println("Messgaes sent!");

producer.close();

The initProducer function instances the Producer as a member of the Kafka Server advertized by the local Zookeeper:

public static Producer<String, String> initProducer() {

Properties props = new Properties();

props.put("zk.connect", "127.0.0.1:2181");

props.put("serializer.class", "kafka.serializer.StringEncoder");

ProducerConfig config = new ProducerConfig(props);

Producer<String, String> producer = new Producer<String, String>(config);

return producer;

}

And the sendMessage function is simple given the preparation of the data done beforehand:

public static void sendMessage(Producer<String, String> producer, String topic, List<String> value) {

ProducerData<String, String> data = new ProducerData<String, String>(topic, value);

producer.send(data);

}

As made evident by the code above, the class requires 3 parameters to be passed to it, and it will fail with an error message otherwise:

**[cloudera@centos-e185 ~]$ storm jar e185-FinalProject.jar e185.FinalProject.KafkaFileStreamer**

(...)

e185 - Final Project

Alexandre de Melo Correia Pinto

Kafka File Streamer Class - sends the content of a text file through a Kafka Server that has been registered on this machine's Zookeeper

usage: e185.FinalProject.KafkaFileStreamer topic filename repetitions

topic - The topic to be used on the Kafka system to send the files

filename - file name of the text file to be sent through Kafka

repetitions - number of times to send the file through Kafka

**Storm Consumer – StormWordCountLocal.java**

The other class on this demo is a Kafka Consumer class that performs the word count computation on the messages it receives from the specific Kafka topic it connects to. The word count computation is done using a series of Bolts in the Storm infrastructure.

As discussed previously, this is the Storm Topology of the example:

// Building the topology for Storm

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("kafka", kafkaSpout, 5);

builder.setBolt("split", new SplitSentence(), 10)

.shuffleGrouping("kafka");

builder.setBolt("count", new WordCount(), 20)

.fieldsGrouping("split", new Fields("word"));

builder.setBolt("print", new ConsolePrinter(), 5)

.shuffleGrouping("count");

builder.setBolt("output", new LocalFilePerisitence(outputDir), 5)

.shuffleGrouping("count");

The topology starts with the **kafka** node, which is based on the KafkaSpout class that is provided by the storm-kafka project we worked so hard to install previously. This object makes the interoperation between Kafka and Storm very straightforward, as it allows us to connect to a Zookeeper server (localhost in this case), and subscribe to a topic. We set it to emit Strings by changing its scheme variable and we are good to go.

// Creating the Kafka Spout to point to the specific topic

List<String> hosts = new ArrayList<String>();

hosts.add("localhost");

SpoutConfig spoutConfig = new SpoutConfig(

KafkaConfig.StaticHosts.fromHostString(hosts, 1),

topic, // topic to read from

"/kafkastorm", // the root path in Zookeeper for the spout to store the consumer offsets

"discovery"); // an id for this consumer for storing the consumer offsets in Zookeeper

spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());

spoutConfig.forceStartOffsetTime(-2);

KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);

Then, we have the **split** Bolt, which received each message line from the **kafka** Spout, splits it on the spaces and emits the split words to the next Bolt. Notice how in addition to the execute function, we also have to override a declareOutputFields function which programmatically defines what will be emitted by the bolt. This can lead to some very interesting topologies where the emissions could change mid-computation, although the receiving Bolts would have to expect that change.

public static class SplitSentence extends BaseBasicBolt {

@Override

public void execute(Tuple tuple, BasicOutputCollector collector) {

String sentence = tuple.getString(0);

StringTokenizer itr = new StringTokenizer(sentence, " ");

while (itr.hasMoreTokens()) {

String word = itr.nextToken();

collector.emit(new Values(word, 1));

}

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word", "count"));

}

}

Next, we get to the **count** Bolt, which receives the words from the **split** Bolt that came previously. The interesting thing here is that although this is clearly a “Reduce”-like function, the Bolt itself is indifferent to it, and we declare and implement it as we do the previous “Map”-like function.

public static class WordCount extends BaseBasicBolt {

Map<String, Integer> counts = new HashMap<String, Integer>();

@Override

public void execute(Tuple tuple, BasicOutputCollector collector) {

String word = tuple.getString(0);

Integer count = counts.get(word);

if(count==null) count = 0;

count++;

counts.put(word, count);

collector.emit(new Values(word, count));

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word", "count"));

}

}

The difference comes from the topology, where we declare the **split** Bolt as receiving its input from the **kafka** Spout using shuffleGrouping (which means what it says, it is indifferent to the order or which node is assigned to it), while on the count Bolt, we specifically request a fieldGrouping on the field (variable) word, which guarantees that the same word will always be sent to the same **count** Bolt, in a behavior akin to the key assignment to a specific Reducer.

Next up, we fork our stream, sending the results from the **count** Bolt to both a **print** Bolt and an **output** Bolt. The **print** Bolt is very straightforward, printing to the console what it receives from the **count** Bolt.

public static class ConsolePrinter extends BaseBasicBolt {

@Override

public void execute(Tuple tuple, BasicOutputCollector collector) {

String word = tuple.getString(0);

Integer count = tuple.getInteger(1);

System.out.println(word + " -- " + count);

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

}

}

But we jazz it up a little bit on the **output** Bolt, where we write to a local file directory the progress we are receiving through the **count** Bolt through its emissions. Since we have to create and open the file, we override a prepare function that is called when the thread that runs the Bolt is created. We also make use of a constructor to pass the name of the output folder we want to create.

public static class LocalFilePerisitence extends BaseBasicBolt {

String \_outputDir;

File \_outputFile;

Writer \_w;

public LocalFilePerisitence(String outputDir) {

\_outputDir = outputDir;

File dir = new File(\_outputDir);

if (!dir.exists()) dir.mkdirs();

}

@Override

public void prepare(Map conf, TopologyContext context) {

int myId = context.getThisTaskId();

\_outputFile = new File(\_outputDir + "/output\_" + myId);

FileOutputStream is;

try {

is = new FileOutputStream(\_outputFile);

OutputStreamWriter osw = new OutputStreamWriter(is);

\_w = new BufferedWriter(osw);

} catch (FileNotFoundException e) {

System.out.println("Unable to open " + \_outputFile);

e.printStackTrace();

}

}

@Override

public void execute(Tuple tuple, BasicOutputCollector collector) {

String word = tuple.getString(0);

Integer count = tuple.getInteger(1);

try {

\_w.write(word + "\t" + count + "\n");

\_w.flush();

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

}

}

These print and output functions are imperfect due to the nature of the stream processing in the sense that they print not only the “final” value, but the intermediates in the computation as well as the data flows in. However, if you think about it, there is no “final” value of the computation, and the counts will always increment as the data flows in.

Like the other class, the StormWordCountLocal class expects a few parameters to work, and will complain if it does not get them all:

[cloudera@centos-e185 ~]$ storm jar e185-FinalProject.jar e185.FinalProject.StormWordCountLocal

(…)

e185 - Final Project

Alexandre de Melo Correia Pinto

Storm WordCount Local Class - connects to local Zookeeper, attaches to Kafka topic and performs wordcount on the messages, outputing incremental counts to a local directory

usage: e185.FinalProject.StormWordCountLocal topic outputDir

topic - The Kafka topic to be connected to

outputDir - The output directory for the incremental wordcounts

A more advanced example could have the output Bolt sending the data to an Hbase or Cassandra database, which then could be efficiently queried for the “latest” data available on a specific word at a given time. I leave a suggestion for the next offering of this course for a Kafka + Storm + Cassandra final project, I fear for the sanity of this student, though.

Anyway, this was fun. I hope this report can help any readers navigate all the issues I had with this setup without any hiccups. Thanks for reading!