School of Electronic Engineering and Computer Science

Science without Borders (2013/14):
3-Months Project Report

Geo-Distributed Application Monitoring

Eder John Scheid

Queen Mary University of London

26/08/2014

Acknowledgements

We thank Dr. Felix Cuadrado for all the guidance during the entire project. Also, the School of Electrical Engineering, Electronics and Computer Science (EECS) part of Queen Mary - University of London for all the facilities and support. Additionality, we are glad that the "*Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq)", a Brazilian Federal agency, trusted and funded us in our one year program here in the United Kingdom.

We also thank Hamed Saljooghinejad and Kishan Narendra Patel, also part of Queen Mary University, who showed us the way to deploy all the framework in PlanetLab and how to configure it.

Lastly, the Storm topology was inspired and built upon the work of the developers of the project "storm-kafka-stater" and the project "kafka-storm-starter", from Nathan Marz (https://github.com/nathanmarz) and Michael G Noll (https://github.com/miguno) respectivelly.

Abstract

The main goal of the project was to develop a Geo-Distributed framework that could monitor and analyse in real-time the load of a Geo-Distributed Database. Before designing the framework we had to research and select the most suitable tools for the task. The selected tools to build this framework were Apache Cassandra, Apache Kafka and Apache Storm. The framework consists of configuring and deploying these three softwares in a distributed scenario.

In order to better distribute the load among the Geo-Distributed Cassandra Servers it was necessary to create a tool that could measure and rank in real-time how loaded each server was. To accomplish this we created a program that would poll status information from the local Cassandra server and send the information to the configured Kafka broker. This code works as a Kafka Producer and because of that, it was deployed in each one of the database servers that we planned to monitor.

The Kafka broker works as a very fast and reliable database to log all the status information that the Cassandra servers sends to it. The broker separates the information it receives by topics, considering this, five topics were used to classify the status data. These topics were: The Cassandra Server's CPU Load (percentage), RAM Usage (percentage), Pending Tasks, Read Latency and Write Latency.

The Storm topology consumes the information from each one these topics in a different Spout in order to process it and rank all the servers by their load in real-time, based on the information that is provided by the Kafka broker. These five topics were chosen because they contain enough information about the load of a Cassandra server and by comparing this information with the information from the others servers, the Storm topology is able to calculate which server is the most loaded, the second most loaded and so on.

The motivation behind this work is that sometimes it’s hard for system administrators to manage and decide in real-time which server has more difficulty to process all the requests. This tool provides them all the information they need to take action to balance the load.

One big system can be composed by more than 10 nodes and millions of clients requesting data. So the automation of a system similar to this is extremely helpful.

Creating a tool that can help these type of systems to be less human dependent, making their own decisions based on the data collected from themselves and acting to balance the load within, contribute to all the Information Technology community by providing a reliable tool to system administrators. This framework doesn't handle load balancing yet, but with further work, the Storm output could be used by another software to remotely balance the load of the servers.

All the details of how the framework works is explained in the next pages, it starts by demonstrating how the status information from Cassandra is acquired, how it is sent to Kafka, how the broker is set up and how the Storm topology consumes this information. The demonstration ends by explaining step-by-step how the ranking algorithm works inside the Storm topology.

Table of Contents

Introduction

Cloud computing and distributed systems are massive constituents of the Internet we know today. An enormous shift has taken place in the way that we consume data on the Internet. Therefore it is necessary to have highly available and fault tolerant systems.

A crucial aspect of building these systems is load monitoring, the process of monitoring and analysing in real time the workload on a given system. We set out to monitor a set of Geo-Distributed Cassandra servers by using Apache Kafka and Apache Storm to retrieve the metrics from these systems and analyse them in real-time, respectively.

The selected tools to build this framework were Apache Cassandra, Apache Kafka and Apache Storm because all of them were designed to be used in a distributed scenario. Also, we could use a lot of support from the Apache Community in order to set up these tools right.

Apache Kafka is a publish-subscribe messaging framework designed as a distributed commit log. It is fast. A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients. Also it very scalable because Kafka is designed to allow a single cluster to serve as the central data backbone for a large organization. It can be elastically and transparently expanded without downtime. Data streams are partitioned and spread over a cluster of machines to allow data streams larger than the capability of any single machine and to allow clusters of co-ordinated consumers.

Apache Storm is basically a distributed real time computation system. Storm makes it easy to reliably process unbounded streams of data, doing for real time processing what Hadoop can do for batch processing.

Here is a brief explanation of the responsibility of each software in the framework:

Apache JMeter

The Apache JMeter™ [1] is an application designed to load test functional behavior and measure performance in several types of applications. We used it to create a custom load of Read/Write requests for each Cassandra server in order to test our framework.

Apache JMeter may be used to test performance both on static and dynamic resources (Files, Web dynamic languages - PHP, Java, ASP.NET, etc. -, Java Objects, Databases and Queries, FTP Servers and more). It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze its overall performance under different load types. It is possible to use it to make a graphical analysis of performance or to test your server/script/object behavior under heavy concurrent load.

Apache Cassandra

Cassandra is a highly scalable and available database that does not compromise performance by doing so [2]. It also provides linear scalability, fault-tolerance and replication across multiple datacenters on commodity hardware or cloud infrastructure. These characteristics make it the perfect platform for mission-critical data.

We have used it in our framework because it is a highly used database in the current computing scenario and also because of its easy data-replication across multiple datacenters.

Apache Kafka

Apache Kafka is a messaging system that is based in the paradigm of publish-subscribe, where anyone can publish to a single Kafka broker and the subscribers will read the messages in real-time [3]. This single cluster, which is called broker, can manage thousands of bytes of reads and writes per second from many clients (publishers/subscribers). Another interesting feature of the Apache Kafka is the preservation of the messages in the disk which means that if a subscriber loses any data it can always read it back from the Kafka broker.

This setup fits perfectly in our framework because we are going to monitor in real-time three Cassandra databases. These databases will send status information to the Kafka broker every second and the Kafka process running in the broker is able to handle and log all the messages in order to allow Storm to read it.

Apache Storm

Apache Storm is a distributed real time computation system written predominantly in the Clojure programming language [4]. Storm reliably process unbounded streams of data. It uses custom created "spouts" (sources of information) and "bolts" to define information sources, manipulations and the data flow inside the framework in order to allow local and distributed processing of streaming data.

This software was used to process in real time all the Cassandra server's status information that arrived at the Kafka broker. The ranking of each server is calculated upon the status information that is read from the Kafka broker by the Storm process.

Background Research

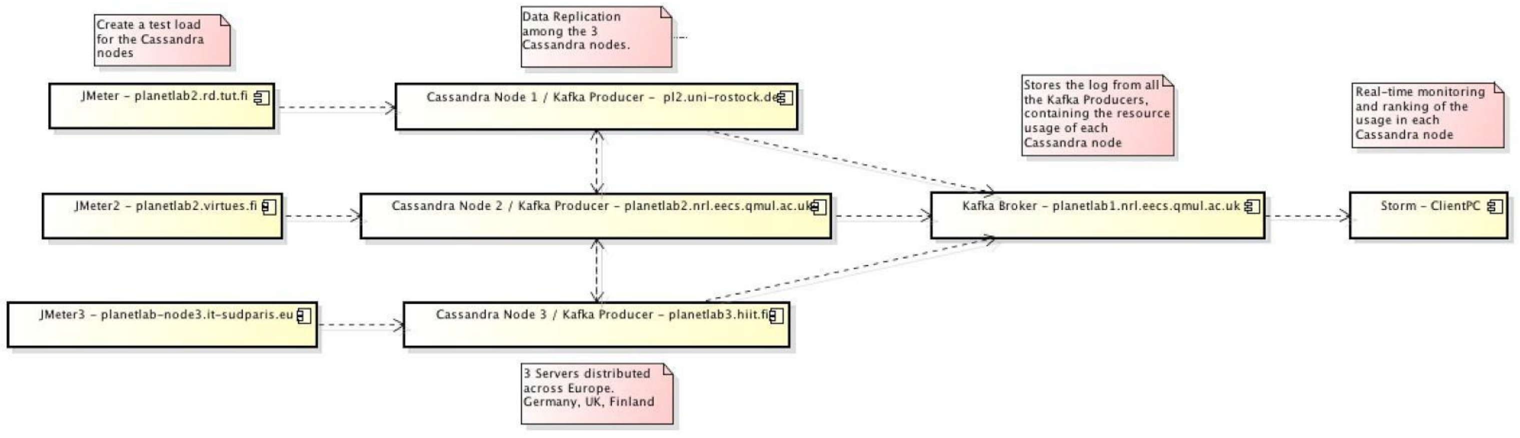
Storm was originally created by Nathan Marz and team at BackType, the project was open sourced after being acquired by Twitter. A Storm application is designed as a topology of interfaces which create a "stream" of transformations. It provides similar functionality as a MapReduce job (Hadoop) but instead of working for batch processing it is mostly used for real-time information. Also, the topology will run indefinitely unless it is manually terminated or programmed to time-out. In 2013 Storm became part of the Apache Software Foundation in its incubator program. [4]

Several pre-defined topologies are available in Storm's website. They cover the general solution of most of the problems a user can come across in terms of real-time computation.

Other developers have been working on Storm also and have made huge contributions to the Open Source Community. Michael G. Noll is one of them, he made great contributions to the integration between Kafka and Storm and also redesigned some of the topologies of Storm. His work on "Implementing Real-Time Trending Topics With a Distributed Rolling Count Algorithm in Storm" was used by us as a starting point for our topology. [10]

Explaining the Framework

It was necessary to use the knowledge of the Distributed Computing and Big Data Analysis (Hadoop) fields to plan how the framework should work and how to connect the pieces of the puzzle. We were able to create an ecosystem to monitor and analyse in real time the performance of a distributed database by using a set up in the following manner:


Create a test load for the Cassandra nodes
Data Replication among the 3 Cassandra nodes.
Stores the log from all the Kafka Producers, containing the resource usage of each Cassandra node
Real-time monitoring and ranking of the usage in each Cassandra node
JMeter – planetlab2.rd.tut.fi
Cassandra Node 1 / Kafka Producer – pl2.uni-rostock.de
JMeter2 – planetlab2.virtues.fi
Cassandra Node 2 / Kafka Producer – planetlab2.nrl.eecs.qmul.ac.uk
Kafka Broker – planetlab1.nrl.eecs.qmul.ac.uk
Storm – ClientPC
JMeter3 – planetlab-node3.it-sudparis.eu
Cassandra Node 3 / Kafka Producer – planetlab3.hiit.fi
3 Servers distributed across Europe. Germany, UK, Finland

(Figure 1 - Framework Setup)

Each Cassandra database was deployed in a server across Europe (Germany, UK, Finland). In each one of these nodes an application (Kafka Producer) was deployed to pull information from the local Cassandra database and send it in regular intervals of time to a fourth node (Kafka Broker). The Kafka Broker stores all these status information from each one of the Cassandra nodes in log files. The Storm client pull the data from the Kafka Broker as soon as a new message arrives, and calculates in real-time which servers are the most loaded on that particular time interval.

Requirements Analysis

In order to properly test our framework it was necessary to gain access to multiple servers distributed across the globe. We managed to get access to them through a PlanetLab account [5].

Design and Tests

At first, we set up all the tools we needed to perform the analysis (Apache Cassandra, Apache Kafka and Apache Storm) on a local node and got familiar with it. Meaning that no data were transmitted between the tools.

Secondly, we connected these frameworks to complete the dataflow in a local machine and tested it. This provided a simple environment that was stable and worked as expected, all the messages were transmitted successfully and all the parts of the system were functional.

Finally, we deployed and tested it in a distributed scenario, which was PlanetLab. We used Apache JMeter to simulate the load on the Cassandra servers. We also coded an application (Kafka Producer) to pull information from Cassandra locally in each server and send them to the Kafka Broker in regular time intervals. The Kafka broker instance would log all these informations so the Storm client could pull the data and process it.

Implementation

The implementation of the project was split in 3 parts, so that we could have more control over the system and know when some part of the system failed.

The division was made accordingly with the functionality of each one the software that composed the tool. First part is the Apache Cassandra Database alongside with the JMeter Setup to simulate the load in the servers. The second part is the Apache Kafka Setup to receive all the information from the database servers. The last part is the Storm Setup that consumes and process the information from the Apache Kafka broker.

Apache Cassandra / JMeter Setup

Cassandra is a distributed database which works over a peer-to-peer communication between its nodes to make them consistent. So, everytime you add more Cassandra nodes, this nodes will try to communicate between themselves to replicate the information of each node with at least one node in the system. This offers a system that has no single point of failure.

In our setup, the Cassandra database was deployed in 3 nodes spread across Europe and every time one table is modified in one of them, all the others 2 are modified as well. The table created was a simple table just for the purpose of testing, because no real data was going to be stored in the database. The data chosen to be analyzed was:

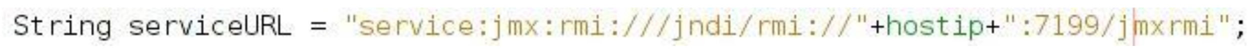
1. Recent Read Latency: this data shows how many milliseconds is taking for the Cassandra Server to read information from its tables, recently.

2. Recent Write Latency: this shows the same type of information as the Read Latency, how long its taking to write information in the tables.
3. Pending Tasks: how many tasks are waiting to be processed, if this number grows and does not decrease, we know that something is wrong with the server.
4. CPU Load: the current percentage of the CPU that is being used, this can tell how well the server is performing.
5. Memory Load: how much of the physical memory is used, this is linked with Java Machine as well, which Cassandra is running.

To retrieve all this data about the Cassandra servers we chose a technology called *Java Management Extensions* (JMX) [6]. It provides tools to applications to retrieve information about Java based programs, locally or remotely like Cassandra.

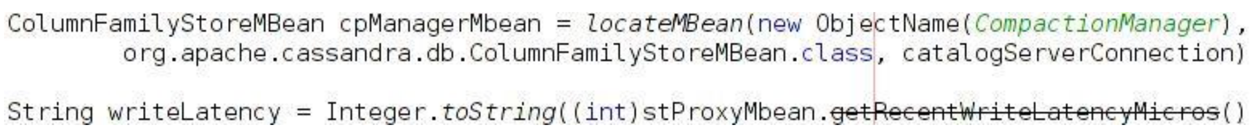
The data available is called *JavaBean*, in Cassandra there is a special type of *JavaBean*, called *ManagedBean (MBean)* [7]. Cassandra offers a huge amount of *ManagedBeans*, divided in groups, some groups concerns about the entire system, while others groups for specific tables.

The connection with the JMX of the Cassandra servers uses the Remote Method Invocation (RMI) interface (Figure 2), after the connection is completed the retrieval of the MBeans is based on the location of the MBean using the connection, once the MBean is located, methods are available for the programmer to use, such as the *getRecentWriteLatency* (Figure 3).



```
String serviceURL = "service:jmx:rmi:///jndi/rmi://"+hostip+":7199/jmxrmi";
```

(Figure 2. RMI Service String Connection for JMX)



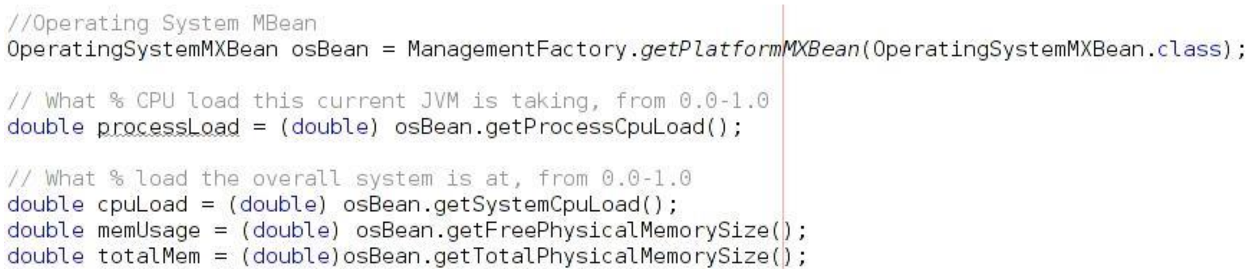
```
ColumnFamilyStoreMBean cpManagerMbean = locateMBean(new ObjectName(CompactionManager),
        org.apache.cassandra.db.ColumnFamilyStoreMBean.class, catalogServerConnection)

String writeLatency = Integer.toString((int)stProxyMbean.~~getRecentWriteLatencyMicros~~()
```

(Figure 3. Retrieving information (Recent Write Latency) from the MBean)

Using the JMX, all the communication details between the host and the client is hidden, so we don't have to be concerned about how both are going to send and receive data, we only have to decide which data to be retrieved.

To avoid an overload of the server and overload of requests to the JMX, the Thread that is sending all the information to the Kafka Server only runs one time per second, so the information is updated in the Kafka Server every second, which leaves network traffic and the processor free to execute other tasks of the Cassandra node.

The information about the node itself, such as the CPU and Memory Load, are retrieved using the same technology as the information from the Cassandra Server, using JMX.Since Java version 1.5, it provides a *JavaBean* called *OperatingSystemMXBean* [8]. The working principle is the same, only that the connection is with the JVM instead of a Java application (Figure 4).



```
//Operating System MBean
OperatingSystemMXBean osBean = ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);

// What % CPU load this current JVM is taking, from 0.0-1.0
double processLoad = (double) osBean.getProcessCpuLoad();

// What % load the overall system is at, from 0.0-1.0
double cpuLoad = (double) osBean.getSystemCpuLoad();
double memUsage = (double) osBean.getFreePhysicalMemorySize();
double totalMem = (double)osBean.getTotalPhysicalMemorySize();
```

(Figure 4. JavaBean to retrieve information about the current machine)

Apache Kafka Setup

The part of the system that connects the whole setup together is the Apache Kafka, it consists in nodes, called *brokers*, that manage and publish message from topics that were created. Also, within the Kafka Server a framework called *Apache Zookeeper* is responsible for all the synchronization of the brokers.

In our system, we choose to have just one *broker*, hosted in the Queen Mary - University of London, there is no need for more than one because our objective is to monitor more than one Cassandra Server and not test the limitations or performance of the Kafka Server.

As Kafka works based on topics to categorize the messages, we created for each information sent from the Cassandra Servers a topic. So, at the end we will have 5 topics, the consumers of the Kafka Server, in our case the Storm, chooses which topic to get the information from, resulting in a more organized environment. The topics were created with meaningful identifiers, such as readLatency and writeLatency.

Storm Setup

Storm works based on the submission of topologies. A topology is basically how the Spouts and Bolts are connected. Generally, a Spout is a class responsible to get information inside the topology. It works as a source of data, in this project we created a Kafka Spout for each topic that we wanted to read from Kafka. These topics were: the Cassandra server's CPU Load, Memory Load, Pending Tasks, Write Latency and Read Latency. Each one of these spouts were then connect to a message parsing Bolt (Kafka Parser Bolt).

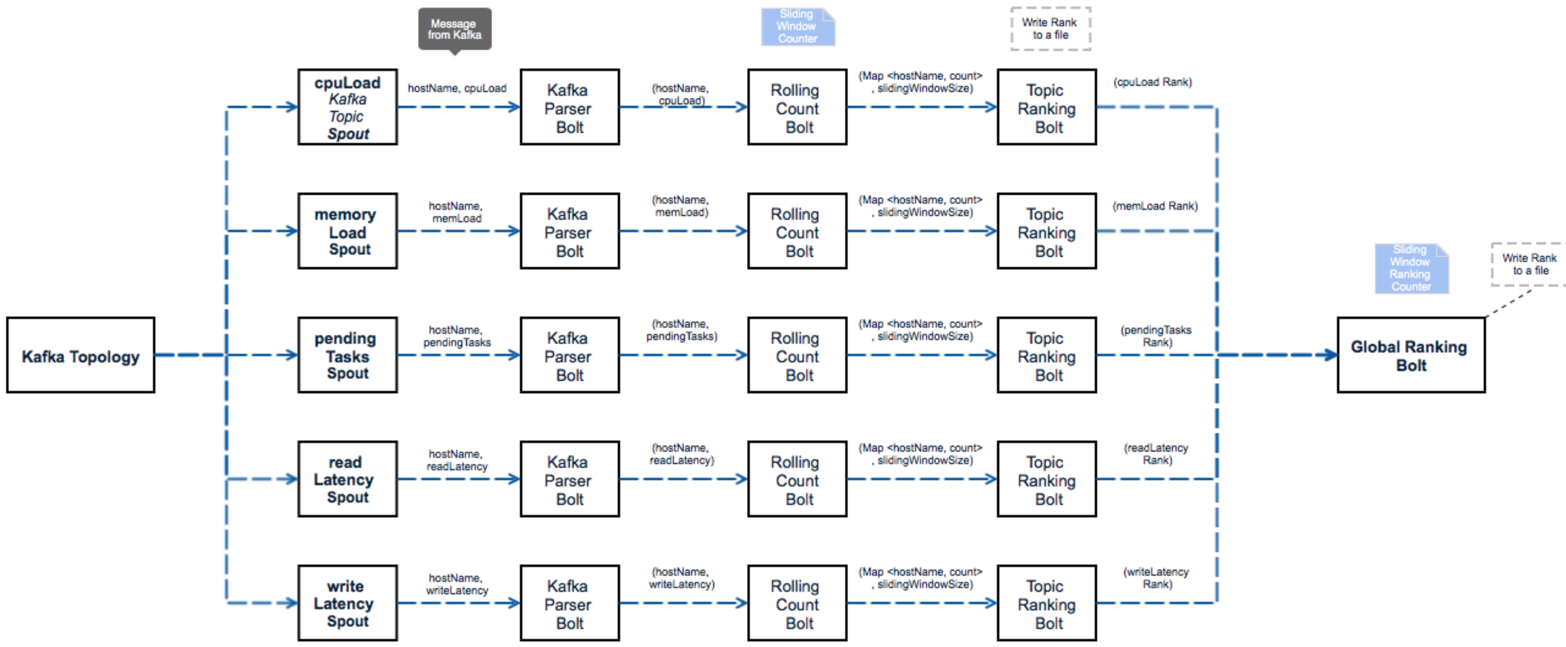
A Bolt in Storm is a class responsible to perform some transformation or calculation over the data provided. The Spouts send information to the Bolts inside the Storm framework in the form of Tuples. Also, it is possible to send information from one Bolt to another.

Then, after parsing the message, the Kafka Parser Bolt sends the information for the next Bolt in the topology: the Rolling Count Bolt. This Bolt is responsible to keep track of the value of each message that each Cassandra Server sent in the past 60 seconds (this is the default size of the sliding window counter inside the class). It also keeps track of how many messages each server server sent. Both information are useful to calculate the average Cpu Load, Memory Load or any other topic message for each server.

The Rolling Count Bolt sends a tuple to the next Bolt (Topic Ranking Bolt) containing a Map of a hostname and a value for each topic. The Topic Ranking Bolt then calculates the average value of a Topic for each Server that publishes its status messages to Kafka, and Rank the servers according to the values of the particular topic.

Finally, the Topic Ranking Bolt sends the rank that it has just calculated for the topic that it was responsible for, and sends it to the Global Ranking Bolt. This Bolt will receive ranks for all the subscribed Topics and calculate a final rank that takes into consideration all the ranks for all topics. The output of this bolt is a rank where a value is associated to a hostname, this value means how loaded that server is in comparison to the other servers.

It is created one object of each one these bolts and spouts for each topic. That means if the topology will watch five Kafka topics, it will be created five spout objects, five parsing bolts, five rolling count bolts and so on. On the other hand, only one Global Ranking Bolt will we instantiated for the whole topology. The following picture will illustrate how the topology is set up:



(Figure 5 - Topology Setup)

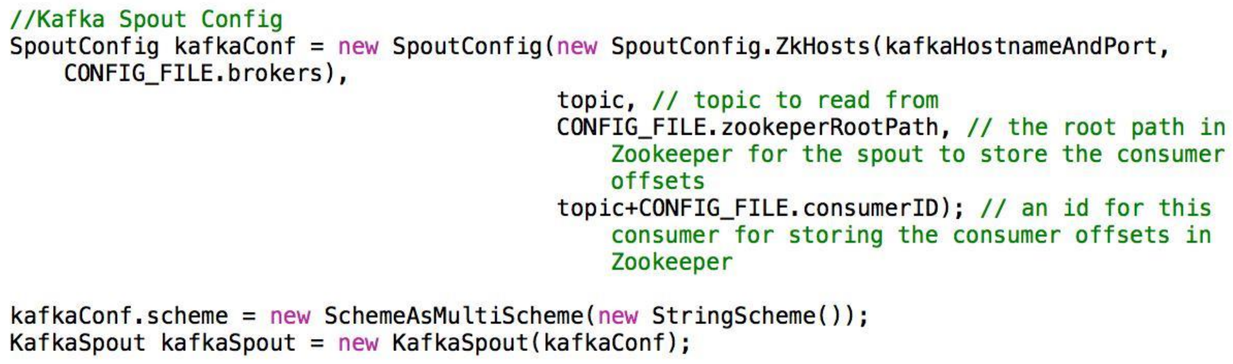
The next session will explain the inner workings of each part of the topology and its responsibilities.

Storm and Kafka connection (Kafka Spout)

The topology setup started by configuring the KafkaSpout class from the package "storm.kafka" to read the data from our Kafka broker. In order to connect Storm to Kafka, the KafkaSpout class needed the following parameters:

- The Kafka broker IP and zookeeper port;
- The topic from Kafka that the spout was responsible to read from;
- The root path in the Zookeeper for the spout to store the consumer offsets;
- A consumer ID to identify itself with the Kafka Broker;

The next picture demonstrate how the connection between Storm and Kafka is done inside the code:



```
//Kafka Spout Config
SpoutConfig kafkaConf = new SpoutConfig(new SpoutConfig.ZkHosts(kafkaHostnameAndPort,
    CONFIG_FILE.brokers),
                                        topic, // topic to read from
                                        CONFIG_FILE.zookeperRootPath, // the root path in
                                            Zookeeper for the spout to store the consumer
                                            offsets
                                        topic+CONFIG_FILE.consumerID); // an id for this
                                            consumer for storing the consumer offsets in
                                            Zookeeper

kafkaConf.scheme = new SchemeAsMultiScheme(new StringScheme());
KafkaSpout kafkaSpout = new KafkaSpout(kafkaConf);
```

(Figure 6 - Storm Setup - Connection to Kafka)

Kafka Message Parser Bolt

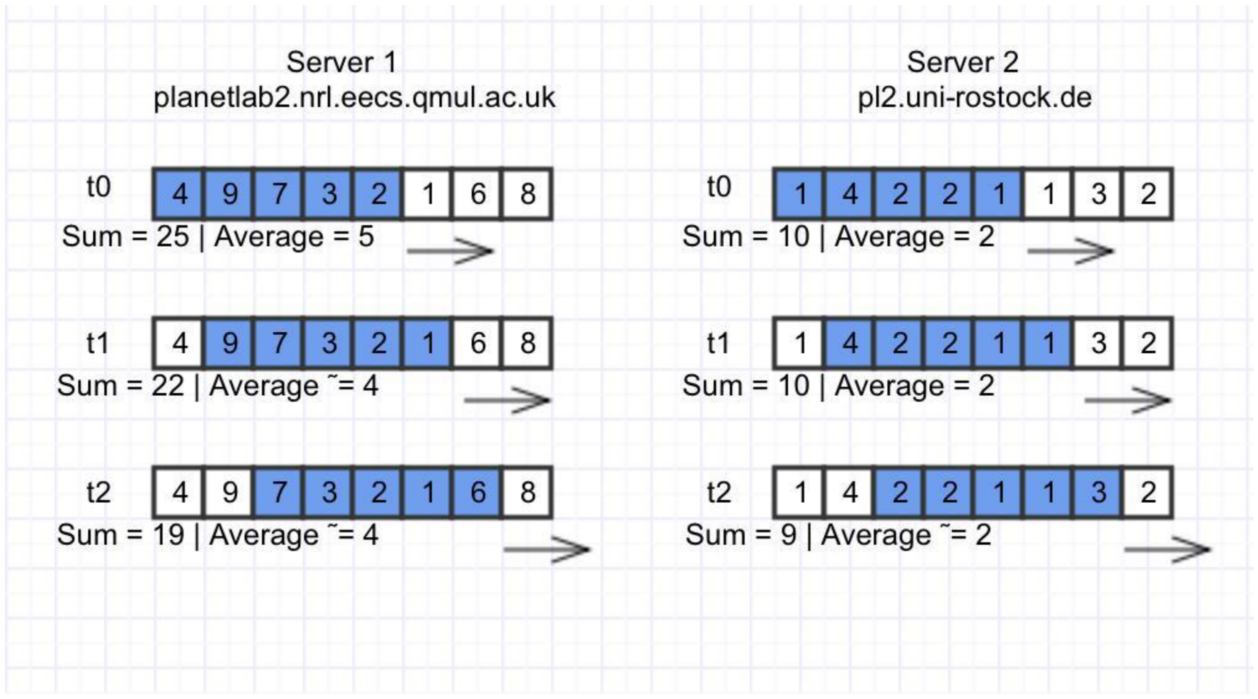
This bolt is responsible to parse the message from Kafka and send a corresponding tuple to the next bolt (Rolling Count Bolt).
The message from Kafka must be in following format : "(STRING),(LONG)"
 * e.g. : Message from server google.com -> google.com,7
would generate the following tuple: (google.com, 7).

 * e.g.2: Message from somewhere -> UniversidadeDeBrasilia,9
would generate the following tuple: (UniversidadeDeBrasilia, 9).

Rolling Count Bolt

This bolt performs rolling counts of incoming objects, by using a sliding window based counter. Two parameters are responsible to setup this bolt, the length of the sliding window in seconds (how many seconds in the past the system will store information) and the emit frequency in seconds (how many seconds the bolt will wait before emiting the count to next bolt). For example, if the window length is set to 60 seconds and the emit frequency to two seconds, then the bolt will output the count from the past minute every two seconds.

The following picture ilustrates how a sliding window counter works. In the example of the picture, it was used a sliding window of size 5.



(Figure 7 - Storm Setup - Sliding Window)

The bolt emits the rolling count tuple per object, consisting of the object itself, the average of the rolling count, and the actual duration of the sliding window. The latter is included in case the expected sliding window length (as configured by the user) is different from the actual length, e.g. due to high system load. Note that the actual window length is tracked and calculated for the window, and not individually for each object within a window.

"Note: During the startup phase you will usually observe that the bolt warns you about the actual sliding window length being smaller than the expected length. This behavior is expected and is caused by the way the sliding window counts are initially "loaded up". You can safely ignore this warning during startup (e.g. you will see this warning during the first ~ five minutes of startup time if the window length is set to five minutes)." This part of the code was kept practically the same as the code provided by Michael G. Noll in his blog.

Topic Ranking Bolt

This bolt receives a tuple from the previous bolt containing the hostname of the server, the average value (count) of this Topic for that hostname, and the actual duration of the sliding window. It uses this tuple to update the Rank of the topic that the bolt is responsible for. The rank is basically a list with all the server's hostnames ordered by their value. Periodically the system flushes this rank to a file. This file is named after the Topic name.

All the output files can be found in the STORM_OUTPUT folder.

e.g. The file cpuLoad.txt will contain the average CPU Load of each one of the Cassandra servers. It will also contain the sliding window size (in seconds) used to count this information. This is useful to know from how many seconds in the past to the present the average count is about. The cpuLoad.txt would look like this when there is three Cassandra Servers sending information to Kafka:

[[pl2.uni-rostock.de|55|60], [planetlab2.nrl.eecs.qmul.ac.uk|37|60], [planetlab3.hiit.fi|26|60]]

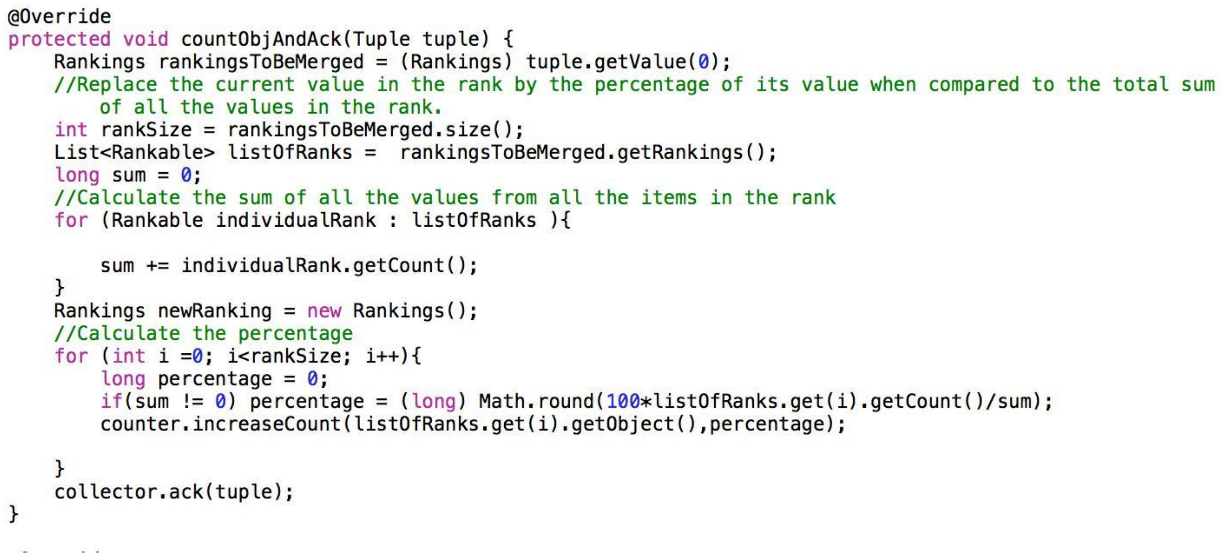
This means that the server "pl2.uni-rostock.de" has a CPU Load of 55% and this value is the average of all the values sent by this server in the past 60 seconds. This also means that the server "planetlab2.nrl.eecs.qmul.ac.uk" has a CPU Load of 37% based on the past minute. Moreover, the server "planetlab3.hiit.fi" has a load of 26% in the same window of time.

Global Ranking Bolt

This bolt is responsible to aggregate all the information and ranks from all the Topics and create a single rank that shows which server is the most loaded, second most loaded and so on.

This metric (Global Rank) is calculated upon the individual values of each server in each one of the topics that they are publishing information about. This metric is the sum of all the percentages that each server has in each one of the topics.

Inside the code the calculation of this percentage is performed in this way:



```
@Override
protected void countObjAndAck(Tuple tuple) {
    Rankings rankingsToBeMerged = (Rankings) tuple.getValue(0);
    //Replace the current value in the rank by the percentage of its value when compared to the total sum
        of all the values in the rank.
    int rankSize = rankingsToBeMerged.size();
    List<Rankable> listOfRanks =  rankingsToBeMerged.getRankings();
    long sum = 0;
    //Calculate the sum of all the values from all the items in the rank
    for (Rankable individualRank : listOfRanks ){

        sum += individualRank.getCount();
    }
    Rankings newRanking = new Rankings();
    //Calculate the percentage
    for (int i =0; i<rankSize; i++){
        long percentage = 0;
        if(sum != 0) percentage = (long) Math.round(100*listOfRanks.get(i).getCount()/sum);
        counter.increaseCount(listOfRanks.get(i).getObject(),percentage);

    }
    collector.ack(tuple);
}
```

(Figure 8 - Storm Setup - Global Rank)

After calculating the percentage that each server is responsible for each topic the Rank is updated by the sum of all 5 (number of Topics) percentages for each server. This Rank is written to a file named "GlobalRankings.txt" in the STORM_OUTPUT folder.

The server with the highest number is the one most loaded. Lets suppose that we have three servers (A, B and C) and three topics (cpuLoad, memLoad, latency). Their current status without the sliding window size is the following:

cpuLoad
[A | 20], [B | 10], [C | 20]

memLoad
[A | 30], [B | 20], [C | 50]

latency
[A | 20], [B | 20], [C | 20]

the Global rank would be calculated this way:

- for each topic it calculates the percentage that each server contribute to the total aggregate amount of that topic:

cpuLoad - the total in that case is 50 (20+10+20)
A - 40% ; B - 20% ; C - 40%
memLoad - the total is 100 (30+20+50)
A - 30% ; B - 20% ; C - 50%
latency - the total is 60 (20+20+20)
A - 33% ; B - 33% ; C - 33%

- for each server, it will add up their percentage for each topic and get the average.

A - (40+30+33)/3 = 34
B - (20+20+33)/3 = 24
C - (40+50+33)/3 = 41

Global Rank:
{A=41, B=34, C=24}

Testing and Evaluation

In order to test and deploy our monitoring system we needed a cluster that consisted of at least seven nodes. We used a Planet Lab account to configure this cluster on the cloud and to have a controlled testing environment. This cluster consisted in three nodes running the Cassandra databases, one node running the Kafka Server and Zookeeper and other three nodes that served as JMeter load generator.

We also used our personal laptops to configure and test the system before deploying the framework to the cluster, avoiding errors and helping the understanding of the working principle of the system.

To achieve our aims and objectives we have proposed some milestones, these milestones were:

1. Became familiarized with the tools that were chosen to be used in the project.
2. Set up a distributed system with all these tools in a testing environment (Planet Lab). Deployed Cassandra, JMeter and Kafka each in one node at planet lab. We deployed 3 nodes for Cassandra, 3 nodes for JMeter and one node for Kafka.
3. Conducted some preliminary tests in every instance of the system, such as a communication test between Apache Kafka and Storm in order to verify if they were working properly.
4. Performed an integration test between Cassandra and our Ecosystem.
5. Simulated the load on Cassandra by using JMeter.

Conclusion

Developing a tool that can monitor and analyze Cassandra Servers that are distributed geographically in real-time is very useful to system administrators. Deciding which server is the most loaded and perform actions to minimize that, are jobs that most times aren’t that simple, but with the tool developed in this project, the work is decreased for those who manage geo distributed servers.

Real-time computing is a paradigm that grows constantly, because of the fact that several problems must be dealt in real time. Not dealing with them in real time can result in loss of information and inaccurate results. This can mean to a business company loss of money or clients. So providing a real-time service that is reliable and fast at all times is extremely important.

Also, this framework can be easily modified to solve other problems. For instance, if we wanted to analyse which London's subway station is the most loaded one in real time, we could get the information of how many people are inside each station regularly and get the information of how many trains have departed from each station in the past 10 minutes (this would be the sliding window size). By using these two topics and deploying a kafka producer client in each station, we could monitor which station is the most loaded by performing only small changes in the Storm framework. This proves that the tool that we have created is very useful, because it can be easily applied to several other problems by performing only very small changes in the code.

The combination of the Kafka Broker, Apache Storm in the tool that we created proved to be satisfactory and passed all the tests that we proposed, so we believe that with a couple more tests and some work within the code the tool should be ready to be released to the public and scientific community.

Further Work

Besides everything that was done in the project, we could extend the project by deploying all the setup in more robust servers, such as the ones in the Amazon Cloud [9]. The use of servers with more capacity can reflect real-world situations and therefore providing more accurate results and data to be processed later on.

Also, the tests that were conducted in the system were simple requests in the databases, without any prior investigation. Simulating a real-world based load in the servers can produce a more significant analysis, consequently helping system administrators to automate the balance load in the servers.

This framework doesn't handle load balancing yet, but with further work, the Storm output could be used by another software to remotely balance the load of the servers.

Expanding this tool to auto balance the load of the servers based on the data collected from themselves will contribute even more to all the Information Technology community by providing a very useful, fast and reliable tool to system administrators.

We hope that this framework can be used and extended by the scientific community in other works related to the topic.

References and Bibliography

[1] Apache ™. Apache JMeter. Retrivied from http://jmeter.apache.org/ Acessed on July 2014.

[2] Apache ™. The Apache Cassandra. Retrieved from http://cassandra.apache.org/ Acessed on August 2014

[3] Apache ™. Apache Kafka - A high-throughput distributed messaging system. Retrivied from http://kafka.apache.org/ Acessed on August 2014.

[4] Apache ™. Storm - Distributed and fault-tolerant realtime computation. Retrivied from *https://storm.incubator.apache.org/* Acessed on August 2014.

[5] PlanetLab Europe. PlanetLab - An Open Platform for Developing, and accessing planetary-scale services. Retrivied from https://www.planet-lab.eu/ Acessed on August 2014.

[6] Oracle and/or its affiliates. Java Management Extensions (JMX) Technology. Retrieved from http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html Acessed on August 2014

[7] Lubow, E and Bradberry, R. The Basics of Monitoring Cassandra (Jan 2014). Retrieved from http://www.informit.com/articles/article.aspx?p=2169293&seqNum=2 Accessed on August 2014.

[8] Oracle and/or its affiliates. Interface OperatingSystemMXBean (2014). Retrieved from http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html Acessed on August 2014.

[9] Amazon Web Services. AWS Amazon Elastic Compute Cloud. Retrieved from http://aws.amazon.com/ec2/ Acessed on August 2014.

[10] Implementing Real-Time Trending Topics With a Distributed Rolling Count Algorithm in Storm. Retrieved from http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/ at 10th of August 2014.

Appendices

- All the code of the framework can be found at: https://github.com/luisfrt/Geo-Distributed-Application-Monitoring-Framework