19-Aug-14

University of Amsterdam
System and Network Engineering

Detecting DDOS attacks using distributed processing frameworks

#59: RP2 REPORT

SUDESH JETHOE
SUDESH.JETHOE@OS3.NL

Supervisor:
Anthony Potappel, Vancis

Summary

In this work three distributed processing frameworks were evaluated, namely Hive, Pig and Spark. From these frameworks Spark offered the most flexibility for implementing a DDOS detection algorithm. By initially slicing the data by interval, port and protocol we were able to identify anomalies in the network traffic data. We discussed several algorithms which can be applied for detecting these network anomalies. From the detection algorithms two were implemented in a distributed processing framework. To make the algorithms suitable for distributed processing we initially created flow counts by interval. After, these flow counts were grouped by service. Finally we applied the selected detection algorithms on the resulting set of flow counts per service per interval for the whole dataset. We found that algorithms configured for one dataset do not necessarily perform well for another dataset. However, the application of distributed processing for network traffic anomaly analysis itself did allow us to process up to 100 GiB of network traffic data in under 10 minutes.

Index

1 Introduction

1.1 Generic

Attacks on online services have been occurring on a regular basis for a long time. However, with the increased digitization of financial and public services, attacks can have greater impact. Even though services offered online can be secured quite well from attacks targeted at vulnerabilities in (web) frameworks it is still a challenge to protect against distributed attacks. The nature of distributed attacks prevents easy mitigation. It is therefore useful to gain more insight in these attacks by network traffic analysis. Analysing the large amounts of network traffic data captured from routers however, can be a challenge using classical analysis methods.

Several frameworks have been developed which are capable of effectively analysing large datasets. Examples are Storm [3], Hadoop [1] and DryadLINQ [4]. Frameworks such as these provide features to store large sets of data across multiple machines and process this data in parallel. This enables the analysis of large datasets in relatively short periods of time.

Vancis is a Dutch company which provides advanced IT-infrastructure services for its customers. Amongst these services they provide connectivity and also computing clusters for universities and research institutions. As a provider of connectivity services Vancis has its own routers and the network traffic data from them. Vancis also has a Hadoop cluster which is used by its customers for analysing large datasets.

1.2 Research Questions

In this research we will make use of the Hadoop distributed processing framework, provided by Vancis, to analyse the historical network traffic (netflow) data from Vancis.

Main research question:

- How can a distributed processing framework be utilized to identify network anomalies in historical netflow data?

Sub questions:

- Which processing framework is best suited for identifying DDOS attacks?
- How can we distinguish anomalies in netflow data?
- Which algorithms for detecting network anomalies exist and how can they be applied in a distributed processing environment?

1.3 Ethical considerations

The network traffic data which is received from Vancis reveals traffic patterns and addresses of customers of the company. Therefore, for this research an agreement was signed that the network traffic data must not leave the companies premises. Besides, all IP-addresses in the resulting data have been anonymised by using a hash algorithm, to ensure privacy of Vancis' customers.

2 Background

Originally Hadoop only supported the MapReduce algorithm. However, new frameworks have been developed which are capable of utilizing the distributed processing capabilities of the Hadoop framework in a wide variety of computer science disciplines. For example; machine learning, database systems, statistics and artificial intelligence. In this research we have assessed these frameworks in order to build a tool for analysis of historical netflow data in order to find network anomalies. Hereby we have focused on DDOS attacks in specific.

2.1 What is Hadoop?

Hadoop is a framework for distributed, parallel execution of computational tasks operating on large, distributed datasets. In order to achieve this, Hadoop consists of three base components, namely: HDFS, Yarn and MapReduce.

2.1.1 MapReduce

MapReduce [2] is the computation paradigm which underlies many distributed processing frameworks.

MapReduce consists of the following components (see figure 1):

- User program: Main component from which computational tasks are sent and received.
- Master: The master assigns computational tasks to the worker nodes.
- Worker: Nodes in the cluster which do the actual processing.

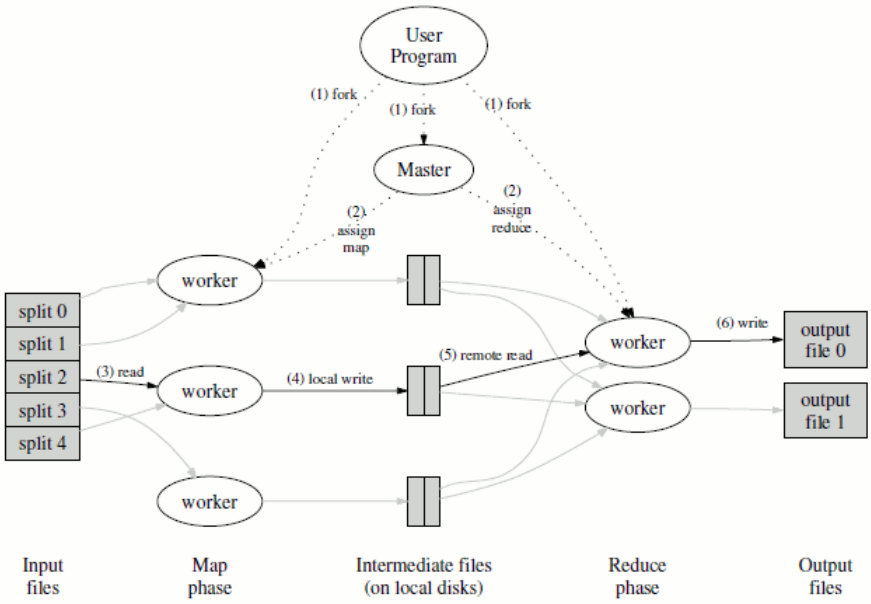

Figure 1 Architectural overview of MapReduce [2]

MapReduce identifies the following functions:

- Map: A function is applied to each individual element in a dataset. Map functions are used to prepare data for further processing by other distributed functions.

- Reduce: Apply a function which joins together all (distributed) elements of a dataset. An example is a function which adds ups all the values of each element in the set, returning a single value, the sum.
- Combine: Combination of map and reduce function which applies a transformation to a distributed dataset and applies a reduce function on local data (data residing on the node where the map function is applied). Combine functions are used to decrease the amount of data which has to be transmitted to, and processed by, the Reducers.
- Filter: Apply a filter function on the data in order to reduce the size of the dataset and remove obsolete data.
- Shuffle: The process which distributes the workload (data and computations) across the cluster. A shuffle is called to distribute workload to the mappers, when the map functions have been applied another shuffle is called to collect the intermediate results and send them to the reducers.

The canonical example of MapReduce is the "wordcount" algorithm. In this algorithm a text is processed to count the occurrences of each word in a given text. The "map" step creates key value combination where each key is a word and the value is 1. After giving each word a value of one, all the (key, value) combinations are "reduced". For this example, the function which is applied is a summation of the values by key. Finally this will result in the counts (occurrences) of each word in the text.

2.1.2 HDFS

The Hadoop distributed filesystem (HDFS) [5] is a framework used for distributing large datasets across multiple hosts. HDFS identifies three main components, the namenode, datanodes and the client. The namenode stores metadata such as the location of blocks and names of the files. The datanodes store the actual data. The client component is used to access the filesystem.

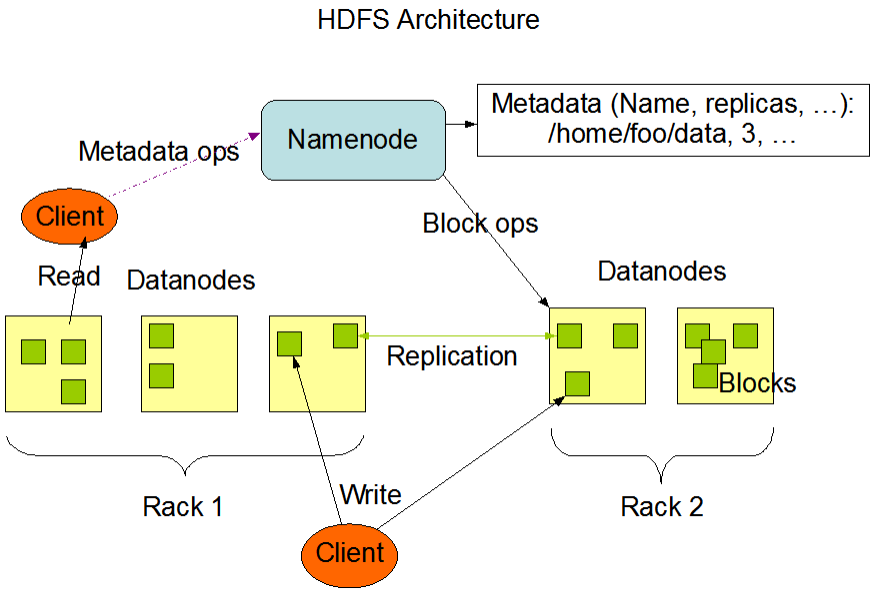

Figure 2 HDFS architectural overview [6]

2.1.3 YARN: cluster manager and resource divisor

YARN [7] is the component of the Hadoop framework which manages resources and takes care of distribution of applications across the cluster. To be able to do this YARN exists of the following (figure 3):

- Client: interface to resource manager, used to submit MapReduce jobs to the cluster
- Resource manager: manages resources
- Node manager: manages the resources on a single node
- Application manager: manages tasks of a single application
- Container: A collection of resources (e.g. processing-, memory-, disk- and network capacity) on a single node

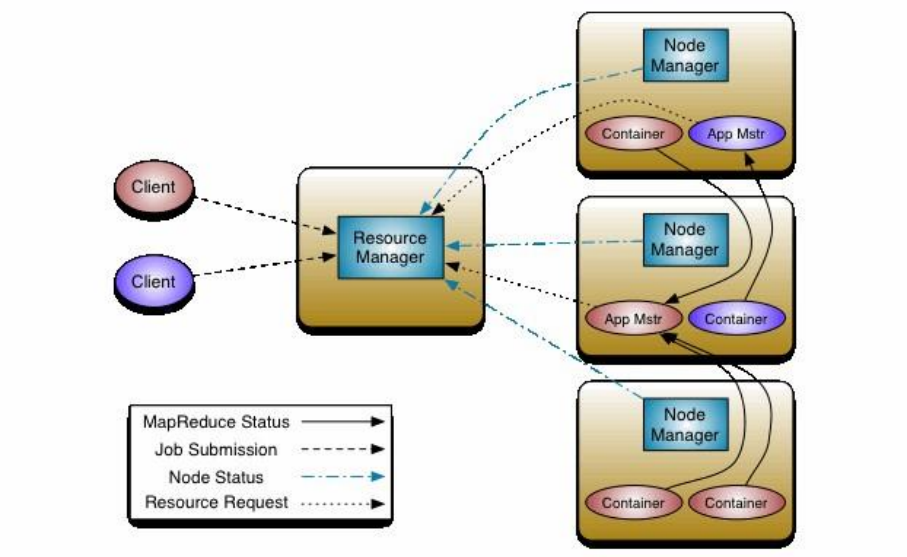

Figure 3 YARN architectural overview [7]

2.2 Extended components of Hadoop

Besides the three base components which together make a "traditional" hadoop cluster, several new tools have been developed which can make use of (parts of) the base components, but add extra functionality. Although there are many, we will only discuss those which can be used for data analysis, namely Hive, Pig and Spark.

Hive

Hive is a tool which can be used for querying structured data. Hive runs on top of HDFS and the data on the distributed filesystem can be queried using an SQL-like syntax.

Pig

Pig can be used to convert data residing on HDFS into a "queryable" format by executing a set of predefined statements on the data. Pig can also be used to query this data, however this require some extra coding, while transformed data can readily be queried using hive.

Spark

Spark [12] is a framework which can be used to run custom algorithms on distributed datasets. Spark was not originally developed to run on top of the Hadoop framework. Spark can be run in three modes, namely standalone or on top of the Apache Mesos [10] or YARN cluster managers. Spark is also capable of utilizing the HDFS filesystem for location aware data processing.

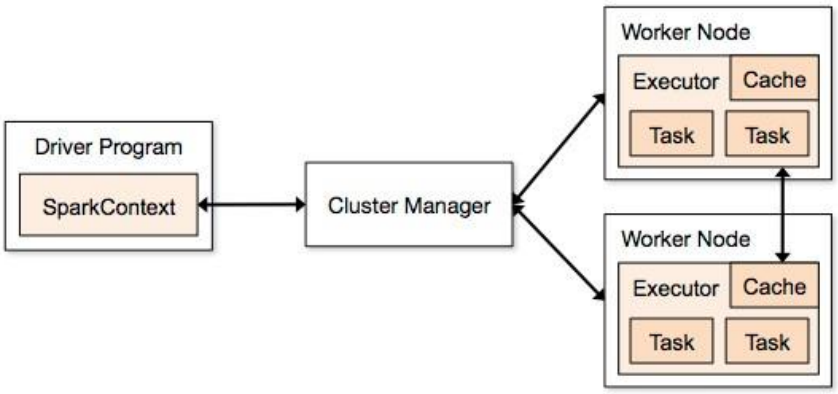

Figure 4 Spark job distribution [9]

In figure 4 an architectural overview of Spark is given. A Spark application works by first loading the “SparkContext”. The SparkContext is a class which references the cluster and exposes distributed objects and functions. The SparkContext requests resources from the cluster manager. The application, or “Driver Program” in Spark, communicates with the workers directly and does not require the cluster manager after the resources have been assigned (figure 5).

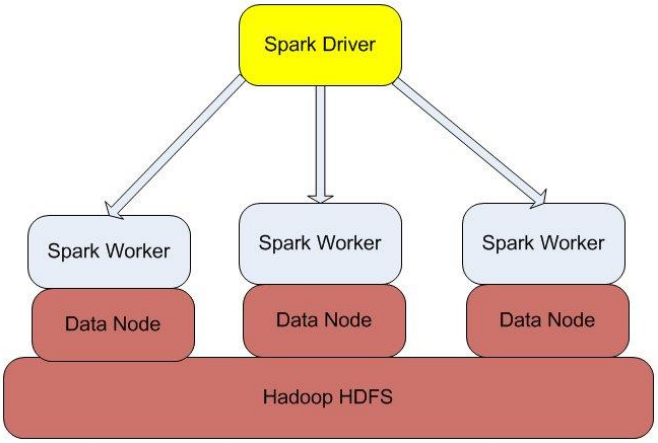

Figure 5 Resource utilization in Spark on top of HDFS [11]

The main concept of distributed processing in Spark is the Resilient Distributed Dataset (RDD). A Resilient Distributed Dataset (RDD) is a parallelized container holding objects distributed across multiple nodes. Spark can apply two types of operations on RDD objects, namely:

- transformations: create new dataset from existing one (e.g. map or filter functions)
- actions: convert objects to discrete data (e.g. reduce or collect functions)

Spark uses RDD's to store the current transformation of the data and is capable of applying new transformations on top of existing ones. Spark applies new transformations to an existing dataset by making use of "lazy execution". This means that the transformations are only executed on the actual dataset at the moment a final output is requested. By making use of "lazy executions" spark is capable of optimizing the execution of multiple map/reduce operations before they are applied to the dataset.

2.3 Detecting DDOS attacks

Several previous works have studied the analysis of DDOS attacks.

2.3.1 LADS: Large-scale Automated DDoS Detection System

In *LADS: Large-scale Automated DDoS Detection System* [13] a multi stage detection system is proposed, composing of SNMP logs and netflow data. In this work the SNMP data is used for a "lightweight" analysis for identifying anomalies (packets per second counters). When anomalies are found, netflow collectors are triggered and start collecting more detailed data.

For the initial, lightweight detection, step SNMP packets per second counters are used for a temporal analysis. The following stages were implemented for the initial step:

- Volume anomaly detection: Traffic anomalies on volume and link utilization measured in bytes/second or packets/second, router cpu utilization, packet drop counts.
- Traffic distribution anomalies: Many attacks can be identified by substantial changes in traffic distributions. Therefore this analysis was used as an augmentation to volume anomaly detection.

For the more detailed analysis, the following stages were implemented:

- Rule based detection: Identify attacks on distinct characteristics such as many traffic to single IP address, traffic originating from botnet blacklists.
- Unidimensional aggregation: Joining data from multiple source/destination addresses on subnet to reduce the size of the dataset.
- Multidimensional clustering: Report high counts of any of the values stored in a flow (src / dest ip, port, protocol ...).

2.3.2 Massively Parallel Anomaly Detection in Online Network Measurement

In *Massively Parallel Anomaly Detection in Online Network Measurement* (MPAD) [14] a combination of several statistical analysis methods is proposed. By combining several methods a higher sensitivity and specificity were achieved.
The following algorithms have been tested in MPAD:

- Average over Window
- Exponential Weighted Moving Average
- Holt Winter Forecasting Model
- Adaptive Threshold Algorithm
- Cumulative Sum Algorithm

Average over window

In [15] the average over window model is discussed. This model applies rule based classification and divides traffic into classes according to a table lookup mechanism. The algorithm is optimized to be applied in devices which have little memory but can do low latency processing, for example routers.

Exponential Weighted Moving Average

The exponential weighted moving average (EWMA) [16] is a moving average in which weights are assigned to the current estimated average and the previous averages. By applying weights, the current or previous averages gain higher importance in calculating the new average. In the implementation mentioned in [16], anomalies are excluded from the EWMA calculation until the exclusion reaches a certain limit, after which the EWMA is reset to the current value.

Holt Winter forecasting
Holt Winter forecasting is an extension of the exponential smoothing algorithm and takes into account [17]:

- *A trend over time (i.e., a gradual increase in application daemon requests over a two month period due to increased subscriber load).*
- *A seasonal trend or cycle (i.e., every day bytes per second increases in the morning hours, peaks in the afternoon and declines late at night).*
- *Seasonal variability (i.e., application requests fluctuate wildly minute by minute during the peak hours of 4-8 pm, but at 1 am application requests hardly vary at all).*
- *Gradual evolution of regularities (1) through (3) over time (i.e., the daily cycle gradual shifts as the number of evening daylight hours increases from December to June).*

Adaptive threshold algorithm
The adaptive threshold algorithm [18] works by calculating an expected value using, for example a *weighted moving average*, following the threshold is defined as a function depending on this average. When the actual value exceeds the calculated threshold for n consecutive intervals, an alarm is raised.

Cumulative Sum Algorithm
The cumulative sum algorithm [18], [19] is a change point detection algorithm which tries to identify cumulative deviations (increments or decrements) over longer periods of time. The algorithm does this by calculating for each following interval an average mean (like the EWMA) and the distance from this average compared to the actual measured value at this point in time. When the actual value is above the average for longer periods of time, the accumulated sum will cross a configured threshold and raise an alert.

3 Methodology

3.1 Dataset and Cluster

In this chapter we will discuss the methodology which we will apply in our final work. We will present the dataset and explain the specific tools and algorithms which we have selected and implemented. The following section is split up in three parts. First we present our dataset and cluster setup. In section 3.2 we will discuss our iterations on setting up an efficient algorithm for analyzing data using the distributed processing framework. After, in section 3.3 we go in further detail on the specific algorithms we have implemented for detecting DDOS-attacks.

Dataset

The network traffic data which we acquired from Vancis was in netflow dump format, originated from ten different routers and was in the period between 27th of February up to the 1st of April, 2014.

router	dataset size
1	84,3 MiB
2	126,7 MiB
3	1,1 GiB
4	3,1 GiB
5	10,0 GiB
6	41,5 GiB
7	88,2 GiB
8	99,3 GiB
9	296,4 GiB
10	444,4 GiB

Table 1 Size of network traffic datasets from different routers as plain text on HDFS

Cluster

The Hadoop cluster utilized for this research consisted of 26 nodes. Each node is equipped with:

- 20 GB's of RAM
- 2 independently configured 2 TB disks
- 1 AMD Opteron processor with 3 vCPUs running at 2,2 GHz
- Cloudera CDH 5.0 on CentOS as operating system and cloud framework distribution
- 1 GB/s ethernet

One node was configured as HDFS namenode. Another node was configured as Spark cluster manager. The remaining 24 nodes were configured as worker nodes.

3.2 Distributed processing algorithm

In section 2.2 several new distributed processing frameworks were discussed, namely Pig, Hive and Spark. To be able to process and analyze the network traffic data the selected framework has to have a few functionalities:

- Support querying of data
- Flexible parsing and processing
- Capable of reading data from netflow dumps

Hive supports querying of the data, however we would have to convert all data to a structured set. Besides, we do not exactly know which relationship should be analyzed. Therefore a more flexible tool is required. Pig supports more advanced queries and limited preprocessing of data, however to read data from netflow dumps external tools would be necessary. Spark however, seems to offer the flexibility which is required to build a tool which supports our requirements. Therefore, we decided to use Spark for implementing our analysis tool.

Implementing Spark

Although the Spark distributed processing framework is written in Scala, Spark also has a set of bindings for Python. Therefore, it is also possible to write applications in Python. Since this will allow us to develop our tool faster, we decided to use Python.

Before we will apply a detection algorithm to analyze the netflow data, the following components need to be created in Spark we need to get the data on the platform.

Read existing netflow data into Spark

There are a few options to approach problem, either we read the data directly into Spark memory. We can also send it to the HDFS cluster first. In this way the data is already distributed across the worker nodes when the analysis needs to be done. At the time of conducting the research Spark was not capable of reading binary data from a HDFS cluster. Therefore the data was converted to plain text first and after saved onto the HDFS cluster (see appendix I for the code).

Create optimized algorithm for distributing workload across cluster

Initially we took the following approach:

1. retrieve unique intervals
2. partition the data by interval
3. for each interval create counts of packets for each found socket

To determine the impact of first getting a list of intervals, the execution time for this step was measured for several datasets (see table 2). The code can be found in appendix II (get_sorted_keys).

router	dataset size in GiB	execution time in seconds	rate in MiB/second
2	0,128	10	12,8
3	1,1	45	24,5
8	99,3	316	314,2
10	444,4	1300	341,8

Table 2: Execution times of collecting all distinct intervals for given dataset and size

Looking at table 2 we can see that the performance hit of collecting the intervals is negligible considering the size of the dataset. The algorithm is more efficient when operating on larger volumes of data.

In the second step the data is not only partitioned by interval, also the number of packets are counted. These steps are combined such that Spark is capable of optimizing the query. In Spark combining queries is known as “pipelining”.

The code can be found in appendix II (*get_high_hits_by_interval*). Although the approach seemed to generate results, it was not very effective in utilizing the platform. It also did not perform well when analysing datasets larger than 4 GiB. Analysis of our method showed that there were over 700 one hour intervals and all intervals were processed sequentially. With an overhead of roughly 10 seconds for launching jobs on the cluster, the performance was severely impacted.

When dividing the complete dataset for a given router (composed of one month of netflow data) in intervals of one hour, roughly 700 intervals have to be analysed. With an average of 10 seconds of analysis time per interval, this adds up to 7000 seconds or 116 minutes of analysis time for even the smallest dataset.

To increase the utilization of the platform, the calculations for each separate interval can be executed in parallel. To achieve this, the program was modified to launch the computation tasks for each interval in separate threads (see appendix III). The Spark jobmanager subsequently processes the tasks in parallel, allocating resources according to a configured job scheduling mechanism (FIFO or fair share).

By applying parallel execution, the analysis time for the 128 MiB dataset was reduced to 8 minutes. Although the execution time was decreased by a factor of 10, it still does not overcome the overhead incurred with launching multiple map/reduce tasks. It seems that an algorithm which avoids running multiple different reduce tasks for each of the 727 intervals will be more efficient. Therefore, we applied a map on the entire dataset dividing each record in the following (key, value) pairs: ("interval:ip address:port:protocol", 1).

Now a reduce action will get the full counts by time and socket in one run. This method also removes the need to parse all the existing intervals before being able to process the data. The code for this method can be found in appendix IV and the results when running this method against different datasets is shown in table 3.

router	dataset size in GiB	execution time in seconds	rate in MiB/second
2	0,128	28	4,57
3	1,1	45,6	4,07
8	99,3	430,4	231
10	444,4	/	/

Table 3: Execution times of collecting all hits for all sockets by hour for all traffic of given router

It can be seen that applying map/reduce on the full dataset is very efficient and the efficiency increases when applied to larger datasets. However, we were unable to run the algorithm successfully against the biggest dataset.

We found the following limitations when trying to make the algorithm run for the biggest dataset:

1. Limited transport capabilities of underlying messaging framework.
2. Limited memory available to driver program.

The framework which Spark uses for transporting data between nodes has a maximum framesize of 10 MiB. The effect of this limitation is that maximum size of a single reduce job cannot exceed 10 MiB. Although this is sufficient for smaller datasets, datasets which have more data also have more

resulting data. For the bigger datasets this size was too limited, causing the application to crash when receiving the results of the reduce jobs.

The problem could be solved in two ways. Either the amount of data which is transmitted by the workers is reduced or the maximum framesize of the messaging framework is increased [20]. To reduce the amount of data transmitted in a reduce job, we applied a filter which increases the threshold of minimum number of hits a certain socket has to receive before it is measured. The maximum framesize of the messaging framework was also set to a higher limit of 64MiB.

When applying this fix, it was also necessary to increase the maximum amount of memory reserved for the driver application. By default the Spark application is configured to use up to 256 MiB. However, due to large number of reduce tasks and residual data this did not fit in the memory of the driver program. Unfortunately, even after increasing the maximum memory usage to 4GiB the application was still not able to process the largest dataset.

In section 4.1 our results are shown when only implementing the deduced algorithm for applying a Spark application against the raw dataset. No specific detection algorithms were implemented in this stage. However, still useful information can be gained from only sorting the data and generating the counts by interval and service on a global level. These results give us a general idea on what kind of specific DDOS algorithms would be applicable to our dataset.

Final Implementation Spark

In our final approach the application was implemented as follows (see appendix V):

1. Initialize cluster with proper parameters for given dataset
2. Read network traffic data from HDFS
3. Apply a map/reduce to get the flow counts for a combination of “destination IP address, port and protocol, hour”.
4. Filter out all combinations with a small number of hits, to reduce the size of the dataset
5. Group all results by combination of “port and protocol”.
6. Filter out all combinations of “port and protocol” which have a hits on only a small number of intervals.
7. Normalize the results by dividing the hits/ip/hour by the total number of hits on this “port and protocol” combination.
8. Plot the normalized hits/interval for all remaining “port and protocol” combinations without the IP address.

Hereby we only look to destination IP addresses, since we assume that DDOS attacks are initiated by remote hosts. Besides, we only focus on attacks targeted at specific services (port and protocol), not at hosts.

3.3 DDOS algorithms methodology

In this research two of the algorithms discussed in section 2.3. In this section we discuss which algorithms we selected and why.

Average over window

The average over window algorithm was designed for devices with low latency and small amounts of memory (routers). This proposition does not align with Spark, which is relatively high latency and is capable of addressing large amounts memory. We will therefore not consider this algorithm.

Cumulative Sum Algorithm

Due to the large amount of data we have to analyse, we group the data by intervals of one hour. However, the events we want to measure might fall exactly within one interval. Therefore, we suspect that algorithms which measure trends over longer periods of time, will not trigger on these short lived events. That is why we will not implement this algorithm.

Holt-Winters

Holt-Winters forecasting is an extension to the weighted moving average, but also takes into account recurring trends. Although trends can be expected when monitoring access to specific sockets (IP + port), in our research we only focus on the general data of a service and do not go into detail for specific IP addresses. Because of this and because we are also filtering out values which fall below a given threshold, we suspect that information of cycles and trends is lost. For our research therefore, we do not apply the Holt-Winters forecasting model.

EWMA

The formula used to calculate the exponential weighted moving average (EWMA) in [16] is given in formula (1). Hereby: $x_0 = x_1$.

$$\widehat{x_{i+1}} = \gamma x_i + (1 - \gamma)\widehat{x_i} \quad (1)$$

In which γ is a smoothing factor which applies weight to the estimation $\widehat{x_i}$ and the actual current value x_i.

The threshold on which an alert is raised is given as a multiple of the current EWMA. Further, the outliers will be excluded from calculating the new estimation. To ensure that a shift in the graph does not classify all subsequent values as outliers, a maximum gap is introduced after which the EWMA is reset. See appendix VI for our implementation. The values for the gap, smoothing- and threshold multiplication factor are determined by using the dataset from router 2 as training data.

Adaptive Threshold Algorithm

In section 2.3 we also discussed the adaptive threshold algorithm. Although the algorithm itself is applicable in our case, some adjustments are required. Because we use full hours for intervals, interesting anomalies occurring only in a short period of time might be missed using the default method. Therefore we will apply this model without "wait time". To implement it we do need to take care of setting a proper threshold to ensure correct discrimination of anomalous and normal traffic. The adaptive threshold algorithm in [18] also uses the EWMA. The main difference with the EWMA algorithm itself is that the adaptive threshold algorithm does not exclude detected anomalies from updates to the average. The algorithm is implemented in python as shown in appendix VI. Also the values for this algorithm are determined by testing it first on the router 2 dataset.

The complete program used to generate the final results is listed in appendix VII.

4 Results

In this section we discuss our results. First the results of running only the Spark algorithm are shown. Then we will show the results when applying the DDOS algorithms to our “training” dataset from router 2. Finally we will show how the DDOS detection algorithms perform against data of other routers.

4.1 Spark

In the results for Spark we highlighted a few results. First we show the anomalies which were found, in the second part trends are shown. Note that all results show a gap between the 8th and 11th of March, we had to discard the netflow data of this part because it was corrupt.

4.1.1 Anomalies

In the first image (5), we see the normalized flow counts (or hits) for 5 ports from data from router 2. The ports are shown in no specific order since the graphs were generated by the program. We do not know any specific details about the anomalies we are looking for. Therefore, we had the program plot all flow counts where the number of hits is above a specified threshold and there were at least a certain number of datapoints to plot.

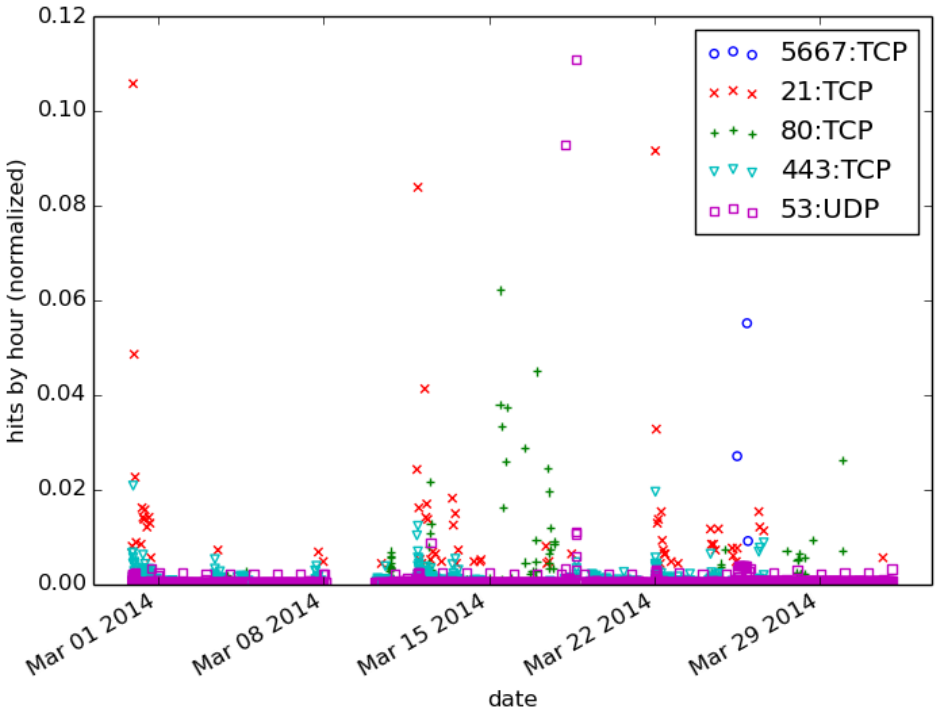

Figure 6 normalized hits for 5 sockets on router 2 (126,7 MiB) with threshold of 10 hits

In figure 6 we can see that the number of hits for on UDP port 53 (DNS) is generally quite stable. However, clearly there are two extreme outliers around the 20th of March. Further, there are also some outliers for traffic on TCP port 80 (HTTP) and TCP port 5667 (Nagios), however these are less extreme. Also, although no clear can be identified for most of the services it is interesting to see that there is a recurring weekly increase in traffic for traffic on TCP port 21 (FTP).

In figure 7 we show the second 5 normalized hits for router 2, here 2 anomalies can be distinguished. Both are for traffic on tcp port 22 (SSH).

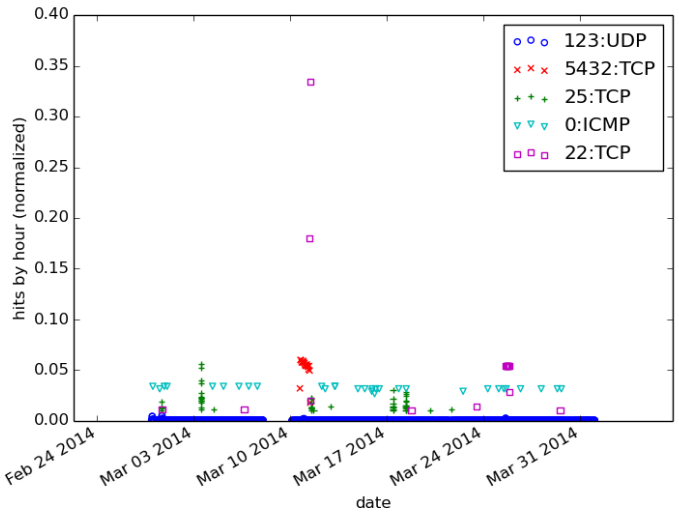

Figure 7 normalized hits for 5 sockets on router 2 with threshold of 10 hits

In figure 8 we show the normalized hits for router 3, here 3 anomalies can be distinguished. One for HTTPS (TCP port 443) and two for TCP port 113, which seems related to different kinds of malware [21].

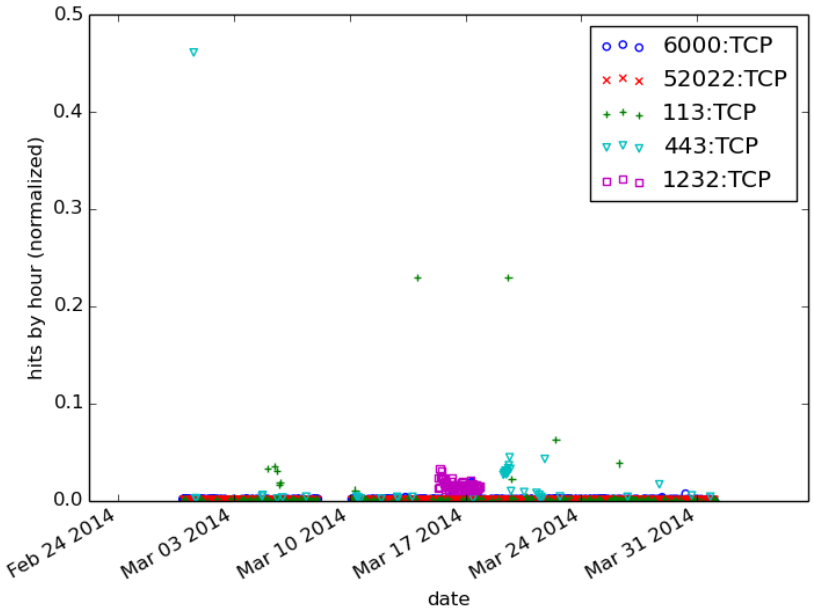

Figure 8 normalized hits for 5 sockets on router 3 (1,1 GiB) with threshold of 10 hits

In figure 9 we can see more data from router 3. Two clear anomalies can be distinguished, one on DNS traffic and one on TCP port 8000. Port 8000 seems mainly to be used by malware[22].

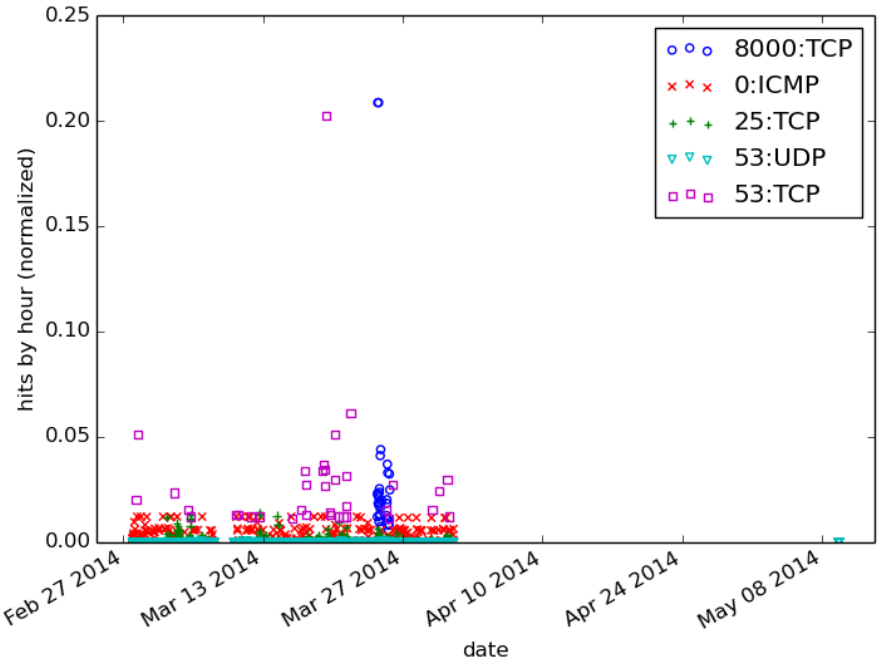

Figure 9 normalized hits for 5 sockets on router 3 (1,1 GiB) with threshold of 10 hits

4.1.2 Trends

In figure 10 and 11 a clear periodicity can be seen for certain types of traffic. In figure 10 columns can be identified corresponding with daytime periods of weekdays.

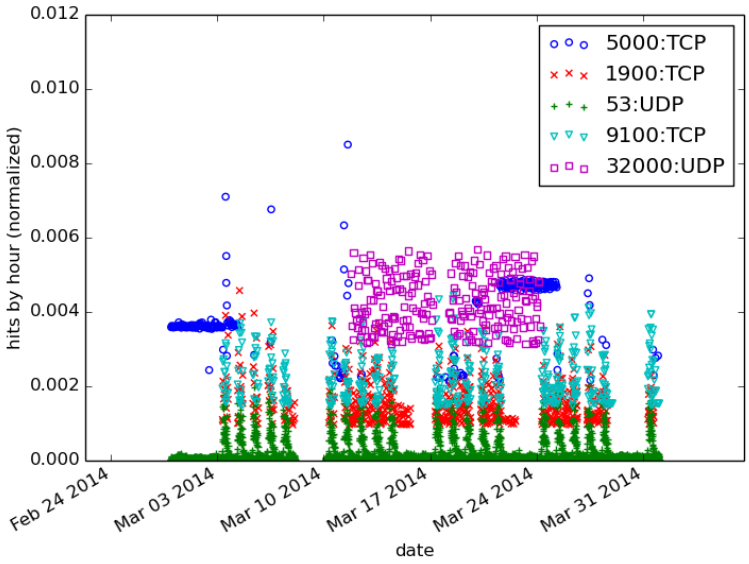

Figure 10 Normalized hits for first 5 port protocols on router 7 with threshold on 500 hits/socket:ip/hour

In figure 11 also a recurring weekly activity is shown for traffic on port 25.

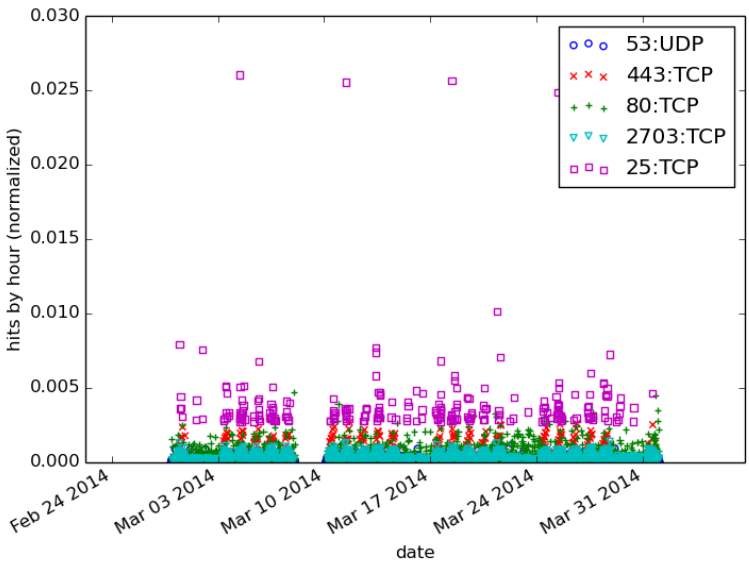

Figure 11 Normalized hits for first 5 ports on router 6 with threshold on 500 hits/socket/hour

In figure 12 bands can be identified which show the hit rates for normal traffic for these services. Lower bands (like for port 80) represent larger volumes of traffic, while higher bands represent lower volumes of traffic (like port 1033 and 1035). Also note the surge in traffic on port 23 in the final week. The vertical density indicates that this traffic is not directed to only one IP address, but targets many different IP addresses. Although it does not seem to indicate a DDOS attack, it can definitely be classified as an anomaly and further investigation might be useful.

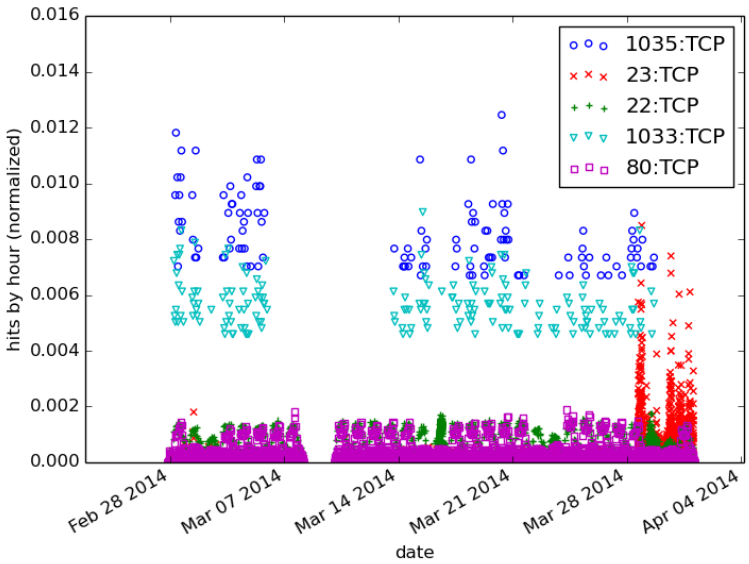

Figure 12 Normalized hits for 5 ports on router 5 with threshold on 20 hits/socket/hour

4.2 DDOS algorithms

The two DDOS detection algorithms we have implemented were first tested on the data from router 2 to configure the parameters. In section 4.2.1 we show how it performs with these configured parameters against this dataset. In section 4.2.2 we show how it performs against other datasets. To have a clear view on the anomalies we only plot specific events, namely:

- The top 1000 hits found on the complete interval
- The hits where an alert is raised for the EWMA algorithm
- The hits where an alert is raised for the ADAPT algorithm

We assume that all the events where an alert is raised should be well within the top 1000 range. Since we do not have a precise list of DDOS attacks we assume that using the top 1000 range will give us at least an indication of the performance of the algorithms.

4.2.1 Performance on training set

The parameters which were tuned had been configured with the following settings:

```
ewma = {
    'avg'       : array[0][4],   # Set initial ewma X0 = X1
    'gamma'     : 0.3,           # gamma
    'thresh'    : 4,             # threshold multiplication factor
    'max_gap'   : 6,             # max interval gap
    'gap'       : 0,             # gap counter
    'alerts'    : []             # Store alerts
}

adapt = {
    'avg'          : array[0][4],     # Set initial avg X0 = X1
    'gamma'        : 0.3,             # gamma
    'thresh'       : 1.2,             # threshold multiplication factor
    'alerts' : []
}
```

In figure 13 we can see the performance of the detection algorithms for detecting anomalies on TCP port 21 (FTP). We can see that both algorithms are capable of detecting the outliers. The EWMA algorithm triggers also on a later interval. This can be explained by the fact that EWMA excludes outliers from updates to the average. The running average will therefore still have a lower value compared to ADAPT where the average is also increased by the anomaly.

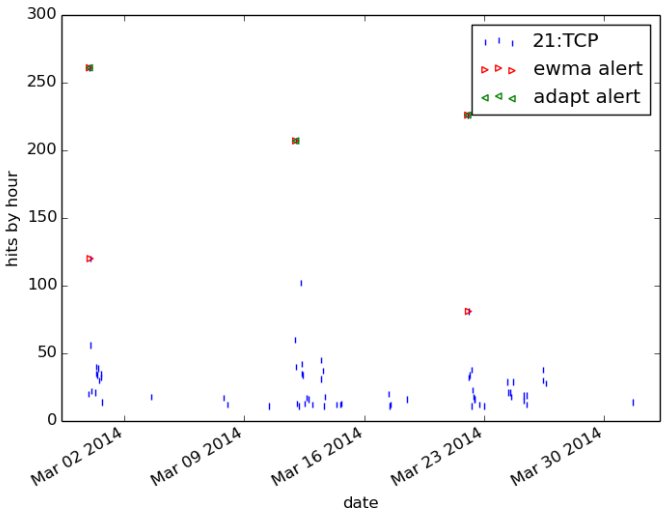

Figure 13 Performance of detection algorithms on FTP traffic for router 2

In figure 14 we see the results for TCP port 22 (SSH). Again both algorithms are capable of detecting the most extreme outliers.

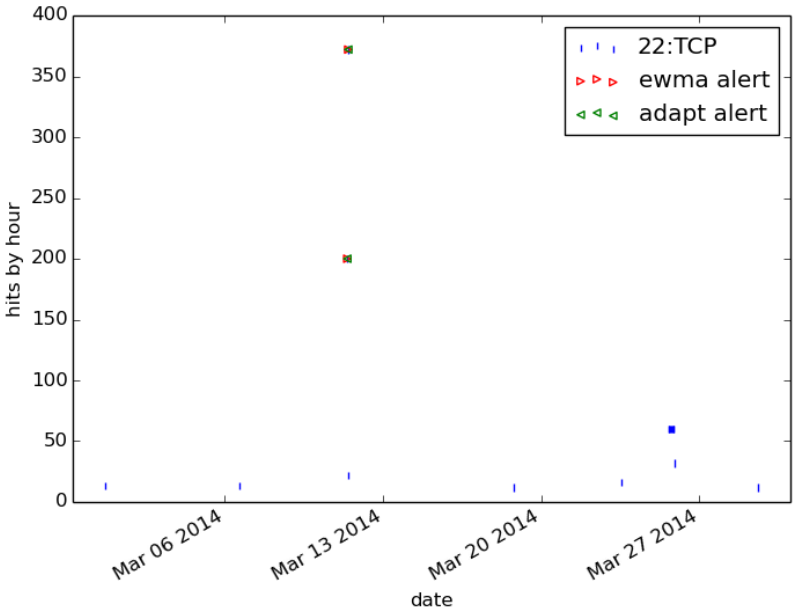

Figure 14 Performance of detection algorithms on SSH traffic for router 2

The same goes for traffic on UDP port 53 (DNS). Here we also start to see some potentially false positives (figure 15). This is probably caused by the relatively large amount of traffic. In ADAPT, a large number of low counts generates a low average, therefore even small deviations can trigger an alert. For EWMA we see the effect of excluding outliers from updating the average, causing the trigger to remain for several consecutive intervals.

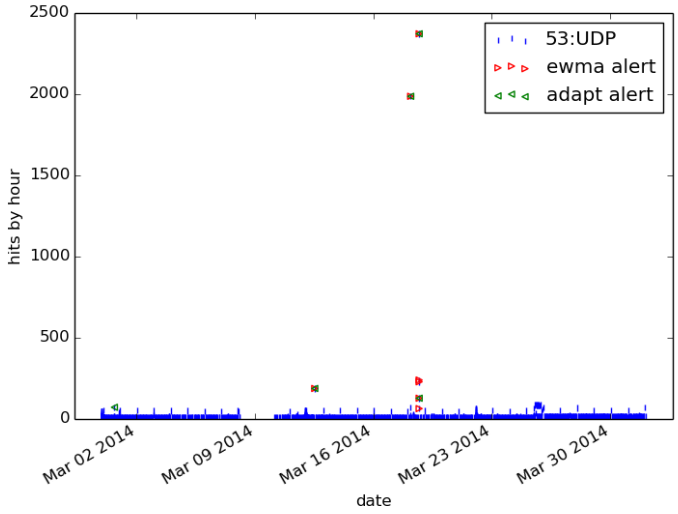

Figure 15 Performance of detection algorithms on DNS traffic for router 2

In contrast on port 80 (figure 16) where there is a large amount of traffic with an irregular pattern, more alerts are raised. For these types of traffic it is generically harder to determine which activities are malicious.

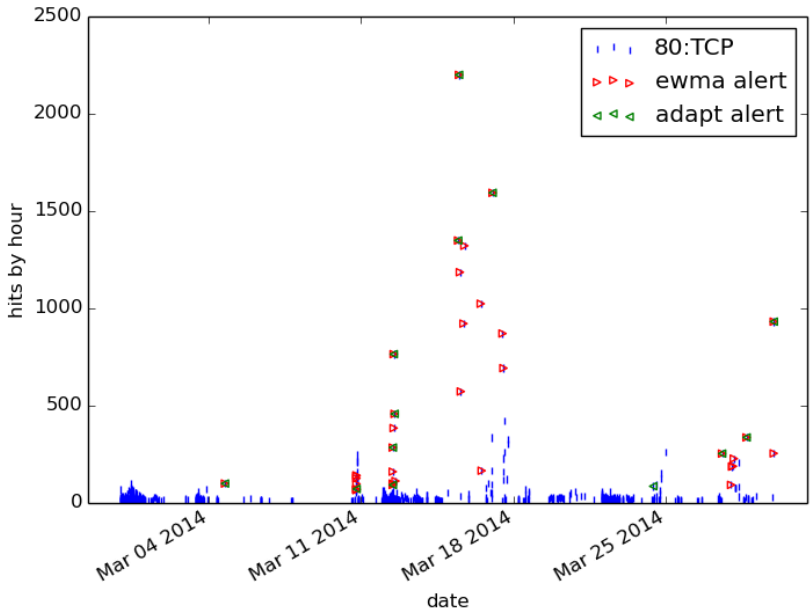

Figure 16 Performance of detection algorithms on HTTP traffic for router 2

4.2.2 Performance on non-training datasets

After we tested the performance of the algorithm and tuned the parameters we applied it on data from the other routers. To be able to run the algorithms successfully on datasets of different sizes, we had to adjust some of the Spark processing parameters. First we show these parameters and the processing speeds. After, we show a subset of the results which were gained.

router	dataset size	flowcount threshold	minimum # total flows/socket	processing time	processing rate
router 1	84,3 MiB	10	10	20s	4 MiB/s
router 2	126,7 MiB	10	10	33s	3,8 MiB/s
router 3	1,1 GiB	10	10	66s	17 MiB/s
router 4	3,1 GiB	20	20	43s	74 MiB/s
router 5	10,0 GiB	100	100	83s	123 MiB/s
router 6	41,5 GiB	100	100	247s	171 MiB/s
router 7	88,2 GiB	100	100	524s	172 MiB/s
router 8	99,3 GiB	100	100	577s	172 MiB/s
router 9	296,4 GiB	500	500	/	/
router 10	444,4 GiB	500	500	/	/

Table 2 Processing speeds for final program

In figure 17 we show the performance for traffic on UDP port 161 (SNMP) for the smallest dataset (router 1, 84 MiB). The regularity in the patterns seem to indicate all this traffic is directed to a single IP-address. Since it is SNMP traffic, this is most likely a monitoring server. Performance of both algorithms seems to be as expected. EWMA alerts on the number of intervals corresponding to its gap setting. ADAPT only raises an alert on extreme differences and not after because of the updated average.

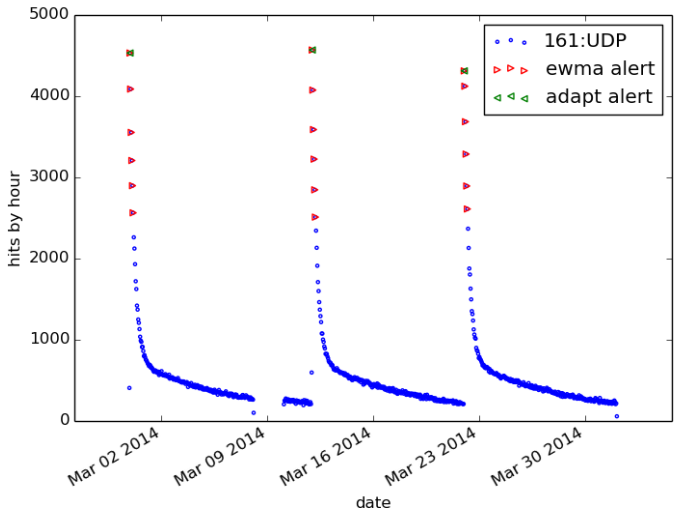

Figure 17 Performance of detection algorithms on SNMP traffic for router 1

In figure 18 we can see the alerts for router 7 (88,2 GiB). Note that since only the top 1000 flow counts are plotted by default the graph has a slight offset and counts do not start at 0. Again the algorithms behave as expected. The port is used for Windows file sharing and directory services.

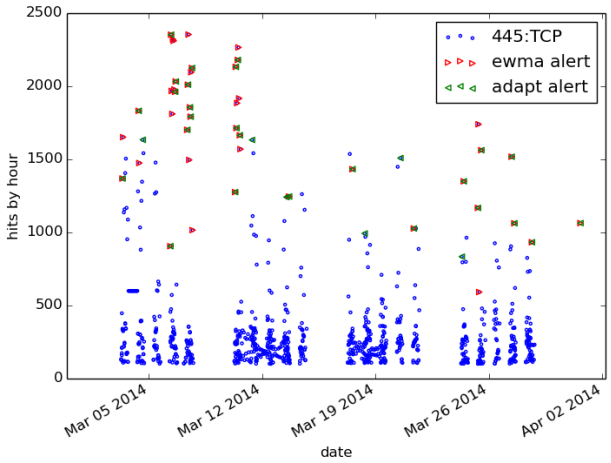

Figure 18 Performance of detection algorithms on Microsoft-DS traffic for router 7

In the final plot (figure 19) HTTP traffic is shown for router 7. The large volume of traffic makes it impossible for the algorithms to make any distinction in traffic. This figure clearly shows the dependency of the algorithms on having a regular traffic pattern. For common services on high traffic networks, a very specific setting with high thresholds would be required to gain any sensible results.

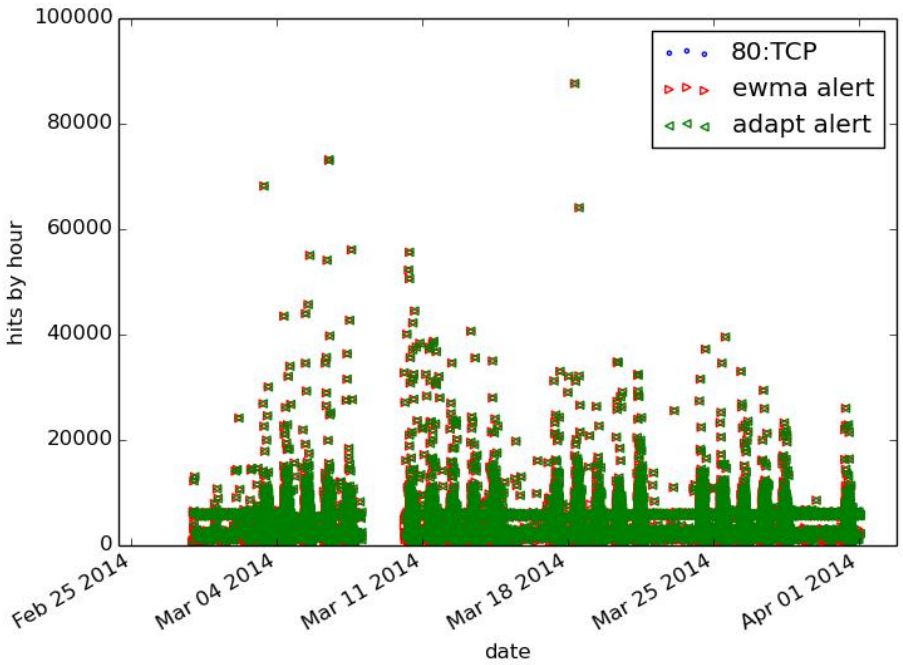

Figure 19 Performance of detection algorithms on HTTP traffic for router 7

5 Conclusion

In this section we discuss our results, conclude our work and make suggestions for future work. First we discuss our implemented Spark algorithm. After we discuss the performance of the DDOS algorithms. Finally we suggest how these methods can be improved.

5.1 Spark

Three different approaches were taken for implementing a distributed processing algorithm in Spark.

First we implemented a traditional method which merely sliced the data by interval and following processed this data by interval. This method turned out to be the most inefficient and not suitable for processing large amounts of data. We found that the inefficiency was mainly caused by two factors. The two factors were the overhead of launching jobs on the cluster and the fact that these jobs were processed sequentially.

In our second approach the jobs were launched in parallel. By using this method we were able to increase the processing speed for our testing dataset of 128 MiB by a tenfold.

However, by creating a method which processed the data in a single map/reduce job we managed to acquire an optimal processing speed which was yet another 17 times faster. This final solution was capable of processing up to 100 GiB of network traffic data in less than 10 minutes.

To be able to successfully scale up to this volume we had to apply the following measures:

- Limit the amount of data which is sent to the driver program to prevent the program from running out memory. In our case we decided to filter out low flow counts and services with a low number of datapoints.
- Increase the memory size assigned to the worker nodes
- Increase the memory size assigned to the driver program
- Increase the maximum size of the messages which can be passed by the underlying framework (Akka).
- Increase the number of data partitions (RDD's) across the cluster

5.2 DDOS Algorithms

When looking at the initial results for our anomaly analysis, a clear distinction could be made between regular and malicious traffic for some services.

Based on the analyses from data of one router we assumed that the application of DDOS attack algorithms would generate promising results when applied to the network traffic data of other routers.

However, the different routers not only had differing sizes but also differing traffic patterns. This had as effect that the performance of the DDOS detection algorithms varied greatly, when applied against datasets of other routers.

Also, it can be debated whether the implemented algorithms would actually be capable of performing well on the larger datasets. This can be seen clearly in figure 19, where there is a high volume of traffic with an irregular pattern. To report any extremities would require the implemented EWMA algorithm to use a very large threshold multiplication factor. However, this would bias the algorithms too much to a specific service and dataset size. In effect, such a modification would degrade the performance on services with a more regular pattern. A clear example is the traffic pattern in figure 19 for HTTP traffic. The range for this type of traffic is between 1 for barely used webservers and goes up to over 80.000 in the extreme case. When comparing this pattern with that shown in figure 14 for SSH traffic on a router with less traffic in general, it can clearly be seen that both types of traffic require a different approach for optimal anomaly detection performance.

6 Future work

In this research we have only implemented two algorithms, however it would be interesting to see how other algorithms perform using a distributed processing framework for analysis.

We also found irregular occurrences of traffic on unusual ports, this can be a clear indicator of malicious activities on a network. Further investigations on the usage pattern of non-common ports could aid in detection of malicious activity on a network.

Another recommendation would be to see how algorithms tuned for historical data, perform on live network traffic data.

Finally, the created method still needs further tuning to be capable of handling larger datasets.

7 References

[1] http://hadoop.apache.org/, 2014-07-01
[2] Dean, J., Ghemawat S. (2004) *MapReduce: Simplified Data Processing on Large Clusters*, OSDI '04: 6th Symposium on Operating Systems Design and Implementation
[3] https://storm.incubator.apache.org/, 2014-07-01
[4] Yu, Y., Isard, M., Fetterly, D., Mihai, B., Erlingsson, U., Gunda, P.K., Currey, J. (2008) *DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language*, OSDI '08: 8th Symposium on Operating Systems Design and Implementation
[5] Shvachko, K., Kuang, H., Radia, S., Chansler, R. (2010) *The Hadoop Distributed File System*, IEEE 26th Symposium on Mass Storage Systems and Technologies
[6] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2014-07-01
[7] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2014-07-01
[8] https://hive.apache.org/, 2014-07-01
[9] https://spark.apache.org/docs/latest/cluster-overview.html, 2014-07-01
[10] http://mesos.apache.org/, 2014-07-01
[11] http://bighadoop.wordpress.com/2014/04/03/apache-spark-a-fast-big-data-analytics-engine/, 2014-07-10
[12] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I. (2010) *Spark: Cluster Computing with Working Sets*, Proceedings of the 2nd USENIX conference on Hot topics in cloud computing
[13] Vyas, S., Duffield, N.K., Spatscheck, O., van der Merwe, J.E., Zhang, H. (2006) *LADS: Large-scale Automated DDoS Detection System*, USENIX Annual Technical Conference
[14] Shanbhag, S., Wolf, T. (2008), *Massively parallel anomaly detection in online network measurement*, Proceedings of 17th International Conference on Computer Communications and Networks
[15] Schwarzer, C. (2006), *Prediction and adaptation in a traffic-aware packet filtering method*, Master's thesis, Ecole Polytechnique Federale de Lausanne
[16] Roughan, M., Griffin, T., Mao, M., Greenberg, A., Freeman, B. (2004), *Combining Routing and Traffic Data for Detection of IP Forwarding Anomalies*, ACM SIGMETRICS Performance Evaluation Review
[17] Brutlag, J.D. (2000), *Aberrant Behavior Detection in Time Series for Network Monitoring*, LISA 14
[18] Siris, V.A., Papagalou, F. (2006), *Application of anomaly detection algorithms for detecting SYN flooding attacks*, Computer Communications 29
[19] NIST/SEMATECH (2013), *e-Handbook of Statistical Methods*, http://www.itl.nist.gov/div898/handbook/pmc/section3/pmc323.htm
[20] http://spark.apache.org/docs/0.9.0/configuration.html#spark-properties, 2014-07-10
[21] http://www.speedguide.net/port.php?port=113, 2014-07-10
[22] http://www.speedguide.net/port.php?port=8000, 2014-07-10

Appendix

I Write netflow data to HDFS

```
#!/usr/bin/env python2.7
# read netflow files and dump them as plain text
# to hdfs distributed filesystem

# pathos multiprocessing library
# pip2.7 install six
# wget http://danse.cacr.caltech.edu/packages/dev_danse_us/pyre-0.8.2.0-pathos.zip
# unzip pyre---
# cd pythia; python2.7 setup.py build; python2.7 setup.py install
# wget http://danse.cacr.caltech.edu/packages/dev_danse_us/processing-0.52-
pathos.zip
# unzip processing---
# pip2.7 install git+https://github.com/uqfoundation/pathos
import os, sys, subprocess, itertools
import pathos
import IPython

########### FUNCTIONS ########
def get_size(start_path):
    '''for given path return filesize in MB'''
    total_size = 0.0
    for dirpath, dirnames, filenames in os.walk(start_path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            total_size += os.path.getsize(fp)
    return total_size/1024**2
def read_nfdump(files):
    '''read nfdumps from binary files on local storage
    files: array of files to be read with nfdump
     %ts       Start Time - first seen
      %td       Duration
      %sa       Source Address
      %sp       Source Port
      %da       Destination Address
      %dp       Destination Port
      %pr       Protocol
      %flg      TCP Flags
      %ipkt     Input Packets
      %opkt     Output Packets
      %ibyt     Input Bytes
      %bpp      bps - Bytes per package
      %fl       Flows
    write output to hdfs'''
    # check for collisions
    router = files[0].split('/')[6]
    start  = files[0].split('.')[-1]
    end    = files[-1].split('.')[-1]
    try:
        existing_files = subprocess.check_output(
            ['hadoop', 'fs', '-ls', '/tmp/live/%s' % router])
    except:
        print "failed reading from hadoop cluster, please restart hdfs"
        sys.exit(1)
```

```
    print """
        start\tend\tfiles\tstarting new nfdump thread
        %s\t%s\t%s\tfiles """ % ( files[0], files[-1], len(files) )
    # nfdump only needs to know first and last file in interval
    common_prefix = os.path.commonprefix([files[0],files[-1]])
    prefix_dir    = os.path.dirname( common_prefix )
    # slice last_file (+ remove trailing '/') to get relative path
    last_file_rel = files[-1][len( prefix_dir )+1:]

    # read file with nfdump and store process in handle
    nfdump        = '/usr/bin/nfdump'
    format        = 'fmt:%ts,%td,%sa,%sp,%da,%dp,%pr,%flg,%ipkt,%bpp,%fl'
    process = subprocess.Popen( \
        [ nfdump, '-O', 'tstart', '-b', '-o', format, '-q', '-R',  "%s:%s" %
(files[0],last_file_rel) ],
        stdout=subprocess.PIPE )

    hdfs_file = "/tmp/live/%s/%s_%s.txt" % ( router, start, end )
    print "trying to write to hdfs:\n\t%s" % hdfs_file
    try:
        subprocess.check_call(
        [ 'hadoop', 'fs', '-put', '-', hdfs_file ], stdin=process.stdout )
    except:
        print "ERROR: failed writing to hdfs, file:\n\t%s" % hdfs_file

def read_files( files, part_size, number_of_files=None):
    '''read array of files and number of files to process
    per round, process all files if None'''
    files = files[:number_of_files] if number_of_files else files
    pool = pathos.multiprocessing.ProcessingPool()

    # slice files array to distribute files evenly over nfdump processes
    files_partitioned = map ( lambda i: files[i:i+part_size], range(0, len(files),
part_size ))
    # parallel execute read_nfdump, autoscales to #cores on local machine
    pool.map( lambda subset: read_nfdump(subset), files_partitioned)



############ MAIN #############
path_to_dumps = '/data/2/tsmrestore/profiles/live'
routers = os.listdir( path_to_dumps )
for router in routers:
    dirs = os.path.join(path_to_dumps, router)
    # get all dumpfiles for this router
    walk_arrays = filter( lambda walk: walk[2], os.walk(dirs))
    files = []
    # generate list of files with path
    for walk_array in walk_arrays:
        files.extend( map( lambda walk: os.path.join(walk_array[0], walk),
walk_array[2] ))
    files.sort()

    # calculate maximum number of files to process per iteration
    part_size = len(files)/pathos.multiprocessing.cpu_count()
    total_size = get_size(dirs)
    avg_size = float(total_size)/len(files)
    avg_part_size = part_size*avg_size
    # the average amount of data processed per iteration
    # cannot be more than $threshold MiB
    # otherwise failures will occur
    threshold_mb = 300
    limit_size = int(threshold_mb/avg_size)
    part_size = part_size if avg_part_size < threshold_mb else limit_size
```

```
print "INFO: start analysis for new router"
print "%s\trouter"  % router
print "%s\tfiles"   % len(files)
print "%.3f GB\tdirectory size"  % (total_size/1024)
print "%.3f MB\taverage file size"  % avg_size
print "%s\tfiles per nfdump process" % part_size
# make directory
try:
    subprocess.check_call(
        ['hadoop', 'fs', '-mkdir', '/tmp/live/%s' % router])
except:
    # we don't care if dir already exists
    pass
# read files and store text in hdfs
read_files( files, part_size )
print "INFO: finished writing data to hdfs for router:\t%s\n" % router
```

II Sequential analysis of intervals

```
def get_sorted_keys(rdd_object):
    '''return sorted array of unique keys for given rdd object
    (must be in (k, v) format))'''
    # 1. sort by key
    # 2. remap rdd to store keys only instead of (k, v)
    # 3. get unique items for rdd
    # 4. collect output
    keys = rdd_object.map( lambda (k, v): k) \
                    .distinct() \
                    .collect()
    return sorted( keys )

def get_high_hits_by_interval_old(rdd, intervals, threshold=0):
    '''# hits per socket is pipelined as:
    # 1. filter dist_records by current interval
    # 2. remove old key from record using map
    # 3. add new key from socket using: "ip_address:port:protocol"
    # 4. reduce by counting #values for this key using countByKey()
    # 5. collect ( create dictonary with key == interval
    #          and value == [array of records within interval]
    #
    # map flow_record to  'timestamp:ip:port:protocol' -> '1'

    new_key = lambda rec: "%s:%s:%s:%s" % (rec[0],rec[4],rec[5],rec[6])
    total_hits_per_socket_interval = []
    for interval in intervals:
        hits_per_socket = rdd.filter( lambda (k, v): k == interval) \
                             .map( lambda (k, v): (new_key(v),1)) \
                             .reduceByKey(operator.add) \
                             .filter( lambda (k, v): v > threshold ) \
                             .collectAsMap()

        total_hits_per_socket_interval.append( hits_per_socket )

### main ###
intervals = get_sorted_keys(netflow_records)
hits = get_high_hits_by_interval_old(netflow_records, intervals)
```

III Parallel analysis of intervals

```
### multi threading functs ###
def collect_rdd(worker_q, result_q):
    '''implement queue for processing tasks
    applies .collectAsMap() function on each rdd_obj'''
    while True:
        rdd_obj = worker_q.get()
        result = rdd_obj.collectAsMap()
        result_q.put(result)
        worker_q.task_done()

def spawn_collect_threads( rdd_array, num_threads = 22):
    '''call collect() function on multiple rdd_objects
    by using queues and threading
    returns nested array of result values'''
    worker_q = Queue.Queue()
    result_q = Queue.Queue()
    threads = []
    # start # threads pointing to collect_rdd function
    for i in range( num_threads ):
        t = threading.Thread(target=collect_rdd,args=(worker_q, result_q))
        t.daemon = True
        t.start()
    # fill queue with objects
    for rdd_obj in rdd_array:
        worker_q.put(rdd_obj)
    # wait for workers to finish
    worker_q.join()
    # collect resulting values from result_q
    array_of_discrete_values = [ result_q.get() for _ in xrange(result_q.qsize()) ]
    return array_of_discrete_values

### ddos analysis specific functions ###
def get_high_hits_by_interval_old(rdd, intervals, threshold=0):
    '''hits per socket is pipelined as:
    # 1. filter dist_records by current interval
    # 2. remove old key from record using map
    # 3. add new key from socket using: "ip_address:port:protocol"
    # 4. reduce by counting #values for this key using countByKey()
    # returns dict '''
    # map flow_record to  'timestamp:ip:port:protocol' -> '1'
    new_key = lambda rec: "%s:%s:%s:%s" % (rec[0],rec[4],rec[5],rec[6])
    total_hits_per_socket_interval = []
    for interval in intervals:
        hits_per_socket = rdd.filter( lambda (k, v): k == interval) \
                             .map( lambda (k, v): (new_key(v),1))   \
                             .reduceByKey(operator.add)             \
                             .filter( lambda (k, v): v > threshold )
        total_hits_per_socket_interval.append( hits_per_socket )
    all_hits = spawn_collect_threads(total_hits_per_socket_interval, num_threads =
30)
    return all_hits

### main ###
intervals = get_sorted_keys(netflow_records)
hits = get_high_hits_by_interval_old(netflow_records, intervals)
print "high hits by interval:"
for k, v in hits.iteritems():
    print "%s\t%s" % ( k, len(v))
```

IV Analysis of data in single map/reduce job

```
### ddos analysis specific functions ###
def get_high_hits_by_interval(rdd, threshold=0):
    # input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
    def map_by_time(entry):
        '''input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
        output : (u'2014-03-25 17', ('123.123.123.123', '42321', 'TCP', 1)  )'''
        record, value = entry
        interval, ip, port, proto = record.split(':')
        port = int(port)
        # anonymize ip
        ip = str(hash(ip))[1:10]
        return (interval, ip, port, proto, value)

   result = rdd.filter(lambda (k, v): v > threshold) \
             .map( map_by_time ) \
             .collect()

    result.insert(0, ('interval', 'ip', 'port', 'proto', 'hits') )
    return result

### main ###
# map flow_record to  'timestamp:ip:port:protocol' -> '1'
new_key = lambda rec: "%s:%s:%s:%s" % (rec[0],rec[4],rec[5],rec[6])
# count hits by interval
temp_rdd = netflow_records.map( lambda (k, v): (new_key(v), 1) ) \
                          .reduceByKey(operator.add)

hits = get_high_hits_by_interval(netflow_records, threshold)

print "high hits by interval:"
for k, v in hits.iteritems():
    print "%s\t%s" % ( k, len(v))
```

V Analysis of network data using only Spark

```
import pyspark
import cPickle as pickle
import re, operator, collections, csv, time
import os, sys, subprocess, itertools
import IPython
import xlwt
import datetime
import socket
from matplotlib import pyplot

########### FUNCTIONS ########
def create_spark_context( partitions=72):
    '''create spark context with given default partition size'''
    conf = pyspark.SparkConf()
    conf.setMaster('spark://node11.echo.hadoop.vancis.nl:7077')
    conf.setAppName('spark-test')
    # reserve #G RAM for spark thread per node
    conf.set('spark.executor.memory', '16g')
    # increase Akka framesize
    conf.set('spark.akka.frameSize', '64')
    conf.set('spark.local.dir', '/data/2/spark_tmp,/data/1/spark_tmp')
    conf.set('spark.default.parallelism', partitions)
    # enable compression on in memory objects (requires extra processing)
    # conf.set('spark.rdd.compress', 'true')
    # allow concurrent execution of multiple calculations
    # (default == FIFO)
    SCHEDULER = 'FIFO'
    conf.set('spark.scheduler.mode', SCHEDULER)
    conf.set('spark.scheduler.allocation.file','/root/bin/fairscheduler.xml')
    return pyspark.SparkContext(conf = conf)

def read_hdfs_files(file, sc):
    '''read file from hdfs
    file can be one file, wildcard, or directory!
    return RDD object'''
    dataset = sc.textFile("%s" % file )
    # store as array first, map after as in read_nfdump
    def parse_flow_record(line):
        '''internal function, for each split and
        distribute records as k: 'Y-M-d H', v: flow_record[]'''
        flow_array  = map( lambda word: word.strip(), line.split(','))
        # store date part without MM:SS
        flow_array[0] = flow_array[0].split(':')[0]
        # store array as (k, v): k = date, v = flow_record )
        flow_record = (flow_array[0], flow_array)
        return flow_record

    RDD = dataset.map(lambda line: parse_flow_record(line))
    return RDD


def get_high_hits_by_pport(rdd, threshold=0, min_pport=0):
    '''input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
    # 1. filter all values below threshold
    # 2. create (k, v) mapping by pport
    # 3. group all records by pport
    # 4. filter all (pport, [] ) where len([]) < min_pport
    # 5. sort values by hits and date (affects ewma!)'''
    def map_by_pport(entry):
        '''input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
        output : (u'42321:TCP', ('123.123.123.123', '42321', 'TCP', 1) )'''
        record, value = entry
        interval, ip, port, proto = record.split(':')
        interval = datetime.datetime.strptime( interval, '%Y-%m-%d %H')
```

```
        port = int(float(port))
        pport = "%s:%s" % (port, proto)
        # anonymize ip
        ip = str(hash(ip))[-9:]
        return (pport,  (interval, ip, port, proto, value) )
    def sort_values(records):
        '''sort by hits(4), interval(0)'''
        for i in (4,0):
            result = sorted( records, key=operator.itemgetter(i))
        return result

    result = rdd.filter(lambda (k, v): v > threshold) \
        .map( map_by_pport ) \
        .groupByKey() \
        .filter( lambda (pport, records): len(records) > min_pport) \
        .reduceByKey( operator.add ) \
        .mapValues( sort_values ) \
        .collectAsMap()
    return result

def save_plot(file_name):
    pyplot.ylim(ymin=0)
    pyplot.gcf().autofmt_xdate()
    pyplot.legend()
    pyplot.savefig(file_name)
    pyplot.clf()

def draw_plots(records_by_pport, threshold, router):
    '''make scatterplots of data by pport'''
    colors  = itertools.cycle(['b','r','g','c','m'])
    markers = itertools.cycle(['o','x','+','v','s'])
    lines = 0
    for pport, records in records_by_pport.iteritems():
        pyplot.xlabel('date')
        pyplot.ylabel('hits by hour (normalized)')
        dates = map( lambda x: x[0], records)
        # normalize hits
        hits  = map( lambda y: y[4], records)
        sum = reduce (operator.add, hits)
        hits_norm = map ( lambda x: float(x)/sum, hits)
        port, proto = pport.split(':')
        label = "%s:%s" % (port, proto)
        service = get_service(port, proto)
        pyplot.scatter(dates, hits_norm, label=label, color=next(colors),
marker=next(markers), facecolors='none')
        lines+=1
        if lines%5 == 0 :
            # draw 5 lines on each graph
            save_plot( "/root/bin/data/plots/%s_%s_%s_norm.png" %
(router,threshold,lines) )
    # capture remaining lines
    if lines%5 != 0:
        save_plot( "/root/bin/data/plots/%s_%s_%s_norm.png" %
(router,threshold,lines) )


########### GLOBAL ###########
partitions = 1500
threshold = 100
min_pport = 1500
sc = create_spark_context(partitions)
############ MAIN    #########
routers = [ 'router 1', 'router 2', 'router 3', 'router 4', 'router 5', 'router 6',
'router 7', 'router 8', 'router 9', 'router 10' ]
```

```
for router in routers:
    t_start = time.time()
    hdfs_dir = "hdfs:///tmp/live/%s" % router
    netflow_records = read_hdfs_files(hdfs_dir, sc)
    # create temp rdd which will be used to
    # - calculate hits/hour ratios
    # - retrieve IP's with highest hitrates (overall)
    # map flow_record to  'timestamp:ip:port:protocol' -> '1'
    new_key = lambda rec: "%s:%s:%s:%s" % (rec[0],rec[4],rec[5],rec[6])
    temp_rdd = netflow_records.map( lambda (k, v): (new_key(v), 1) ) \
           .reduceByKey(operator.add)

    temp_rdd.persist(pyspark.StorageLevel.MEMORY_AND_DISK_SER)

    # GET HITS BY PPORT
    hits_by_pport = get_high_hits_by_pport(temp_rdd, threshold, min_pport)
    draw_plots(hits_by_pport, threshold, router)
```

VI Code for detection algorithms

EWMA Algorithm

```
    ewma = {
        'avg'      : array[0][4],  # Set initial ewma X0 = X1
        'gamma'    : 0.3,          # gamma
        'thresh'   : 4,            # threshold multiplication factor
        'max_gap'  : 6,            # max interval gap
        'gap'      : 0,            # gap counter
        'alerts'   : []            # Store alerts
    }


    if value < ewma['avg']*(1+ewma['thresh']) and value > ewma['avg']*(1-
ewma['thresh']):
    # value is between upper and lower ewma threshold, no alert
        ewma['gap'] = 0
        ewma['avg'] = ewma['gamma']*value + (1-ewma['gamma'])*ewma['avg']
    elif value > ewma['avg']*(1+ewma['thresh']):
        # value exceeds upper bound threshold, alert
        # no alerts for values below lower bound threshold
        ewma['alerts'].append(record)
        ewma['gap'] += 1
        if ewma['gap'] >= ewma['max_gap']:
            # if gap exceeds max gap size:
            #   1. reset gap
            #   2. force update of ewma
            ewma['gap'] = 0
            ewma['avg'] = ewma['gamma']*value + (1-ewma['gamma'])*ewma['avg']
```

Adaptive Threshold Algorithm

```
        adapt = {
            'avg'         : array[0][4],    # X0 = X1
            'gamma'       : 0.3,            # gamma
            'thresh'      : 1.2,            # threshold multiplication factor
            'alerts' : []
        }


        adapt['avg'] = adapt['gamma']*value + (1-adapt['gamma'])*adapt['avg']
        if value > adapt['avg']*(1+adapt['thresh']):
            # value exceeds upper bound threshold, alert
            adapt['alerts'].append(record)
```

VII Final implementation

```
#!/usr/bin/env pyspark
# spark test

import pyspark
import cPickle as pickle
import re, operator, collections, csv, time
import os, sys, subprocess, itertools
import Queue, threading
import IPython
import xlwt
import datetime
import socket
from matplotlib import pyplot


########### FUNCTIONS ########
def create_spark_context( partitions=72):
    '''create spark context with given default partition size'''
    conf = pyspark.SparkConf()
    conf.setMaster('spark://node11.echo.hadoop.vancis.nl:7077')
    conf.setAppName('spark-test')
    # reserve #G RAM for spark thread per node
    conf.set('spark.executor.memory', '16g')
    # increase Akka framesize
    conf.set('spark.akka.frameSize', '64')
    conf.set('spark.local.dir', '/data/2/spark_tmp,/data/1/spark_tmp')
    conf.set('spark.default.parallelism', partitions)
    # enable compression on in memory objects (requires extra processing)
    # conf.set('spark.rdd.compress', 'true')
    # allow concurrent execution of multiple calculations
    # (default == FIFO)
    SCHEDULER = 'FIFO'
    conf.set('spark.scheduler.mode', SCHEDULER)
    conf.set('spark.scheduler.allocation.file','/root/bin/fairscheduler.xml')
    return pyspark.SparkContext(conf = conf)

def read_hdfs_files(file, sc):
    '''read file from hdfs
    file can be one file, wildcard, or directory!
    return RDD object'''
    dataset = sc.textFile("%s" % file )
    # store as array first, map after as in read_nfdump
    def parse_flow_record(line):
        '''internal function, for each split and
        distribute records as k: 'Y-M-d H', v: flow_record[]'''
        flow_array  = map( lambda word: word.strip(), line.split(','))
        # store date part without MM:SS
        flow_array[0] = flow_array[0].split(':')[0]
        # store array as (k, v): k = date, v = flow_record )
        flow_record = (flow_array[0], flow_array)
        return flow_record

    RDD = dataset.map(lambda line: parse_flow_record(line))
    return RDD

def read_object_from_disk(file):
    '''read pickled object from file and return it'''
    try:
        fh = open(file, 'rb')
        object = pickle.load(fh)
        fh.close()
        return object
    except:
        print "ERROR: can't read from file"
```

```
        return False
def write_object_to_disk(file, object):
    '''write object to file'''
    try:
        fh = open( file, 'wb')
        pickle.dump(object, fh)
        fh.close()
    except:
        print "ERROR: can't access file"
        return False
    print "INFO: write succeeded"
    return True

def get_data_size(router):
    '''retrieve size of data on hdfs in MB'''
    du_output = subprocess.Popen(
        ['/usr/bin/hadoop', 'fs', '-du', '-h', '/tmp/live' ],
        stdout=subprocess.PIPE).communicate()[0]
    for line in du_output.splitlines():
        # '126.7 M  /tmp/live/sarar9'
        if re.search( router, line ):
            size, unit, path = line.split()
            if unit == 'G':
                size = float(size)
                size *=1024
            return size
    print "ERROR: router not found"
    return False

def get_high_hits_by_pport(rdd, threshold=0, min_pport=0):
    '''input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
    # 1. filter all values below threshold
    # 2. create (k, v) mapping by pport
    # 3. group all records by pport
    # 4. filter all (pport, [] ) where len([]) < min_pport
    # 5. sort values by hits and date (affects ewma!)'''
    def map_by_pport(entry):
        '''input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
        output : (u'42321:TCP', ('123.123.123.123', '42321', 'TCP', 1) )'''
        record, value = entry
        interval, ip, port, proto = record.split(':')
        interval = datetime.datetime.strptime( interval, '%Y-%m-%d %H')
        port = int(float(port))
        pport = "%s:%s" % (port, proto)
        # anonymize ip
        ip = str(hash(ip))[-9:]
        return (pport,  (interval, ip, port, proto, value) )
    def sort_values(records):
        '''sort by hits(4), interval(0)'''
        for i in (4,0):
            result = sorted( records, key=operator.itemgetter(i))
        return result

    result = rdd.filter(lambda (k, v): v > threshold) \
        .map( map_by_pport ) \
        .groupByKey() \
        .filter( lambda (pport, records): len(records) > min_pport) \
        .reduceByKey( operator.add ) \
        .mapValues( sort_values ) \
        .collectAsMap()
    return result

def get_high_stats_by_interval(rdd, threshold, min_pport):
    '''input: rdd with:
    [(u'2014-03-25 17:123.123.123.123:42321:TCP', 1)]
```

```
    output:
        stats {}
    '''
    def map_by_pport(entry):
        '''input : (u'2014-03-25 17:123.123.123.123:42321:TCP', 1)
        output : (u'42321:TCP', ('123.123.123.123', '42321', 'TCP', 1) )'''
        record, value = entry
        interval, ip, port, proto = record.split(':')
        interval = datetime.datetime.strptime( interval, '%Y-%m-%d %H')
        port = int(float(port))
        pport = "%s:%s" % (port, proto)
        # anonymize ip
        ip = str(hash(ip))[-9:]
        return (pport,  (interval, ip, port, proto, value) )

    def apply_algorithms(array):
        '''algorithms to be applied on "values" which are grouped by
            "port:protocol" combination" e.g. array of:
            [
            (datetime.datetime(2014, 3, 24, 23, 0), '555170168', 123, u'UDP', 13),
            (datetime.datetime(2014, 3, 29, 19, 0), '829666315', 123, u'UDP', 13),
            (datetime.datetime(2014, 3, 3, 20, 0), '829666315', 123, u'UDP', 15),
        ]'''
        # loop through array by date
        array = sorted ( array, key=operator.itemgetter(0) )
        # parameters for ewma calculations
        # avg initialized as first value in array
        # orig gamma: 0.05 thresh 0.2
        ewma = {
            'avg'         : array[0][4],
            'gamma'       : 0.3,
            'thresh'      : 4,
            'max_gap'     : 6,
            'gap'         : 0,
            'alerts' : []
        }
        adapt = {
            'avg'         : array[0][4],
            'gamma'       : 0.3,
            'thresh'      : 1.2,
            'alerts' : []
        }
        n = len(array)
        total = 0
        top_1000        = []
        #adaptive_alerts = []
        for record in array:
            value = record[4]
            total +=value
            # calculate top 1000
            if len(top_1000) < 1000:
                top_1000.append(record)
            else:
                pass
                # update top1000 if current hits_count is higher than that of
                # the entry with the current lowest hitcount in top_1000 list
                index, min_record = min(enumerate(top_1000), key=lambda x: x[1][4])
                lowest = min_record[4]
                if value > min:
                    top_1000[index] = record
            # apply ewma algorithm
            if value < ewma['avg']*(1+ewma['thresh']) and value > ewma['avg']*(1-
ewma['thresh']):
                # value is between upper and lower ewma threshold, no alert
                ewma['gap'] = 0
```

```
                ewma['avg'] = ewma['gamma']*value + (1-ewma['gamma'])*ewma['avg']
            elif value > ewma['avg']*(1+ewma['thresh']):
                # value exceeds upper bound threshold, alert
                # no alerts for values below lower bound threshold
                ewma['alerts'].append(record)
                ewma['gap'] += 1
                if ewma['gap'] >= ewma['max_gap']:
                    # if gap exceeds max gap size:
                    #   1. reset gap
                    #   2. force update of ewma
                    ewma['gap'] = 0
                    ewma['avg'] = ewma['gamma']*value + (1-ewma['gam-
ma'])*ewma['avg']
            # apply adaptive threshold algorithm
            adapt['avg'] = adapt['gamma']*value + (1-adapt['gamma'])*adapt['avg']
            if value > adapt['avg']*(1+adapt['thresh']):
                # value exceeds upper bound threshold, alert
                adapt['alerts'].append(record)
        result = {
            'n': n,
            'top_1000': top_1000,
            'ewma_alerts': ewma['alerts'],
            'ewma_gamma' : ewma['gamma'],
            'ewma_thresh': ewma['thresh'],
            'ewma_max_gap': ewma['max_gap'],
            'adapt_alerts': adapt['alerts'],
            'adapt_gamma' : adapt['gamma'],
            'adapt_thresh': adapt['thresh'],
            'total':total }
        return result

    # filter out values below threshold
    # group records by "port:proto" combination
    # filter out arrays with data on less then min_pport intervals
    # apply algorithms to each array of results for certain "pport" combination
    # [ ( key, [values] ), ( key, [values]) ]
    stats_by_pport = rdd.filter(lambda (k, v): v > threshold) \
        .map( map_by_pport ) \
        .groupByKey() \
        .filter( lambda (pport, records): len(records) > min_pport) \
        .mapValues( apply_algorithms ) \
        .collectAsMap()
    return stats_by_pport
def get_hhi_stats(rdd):
    '''calculate the mean,median,max for given interval
    in rdd and return array with given stats by hour'''
    hits_hour = rdd.collectAsMap()
    hhi_arr = []
    header = ['interval','mean','max' ]
    hhi_arr.append(header)
    for (hour, value) in sorted(hits_hour.items()):
        interval = hour+':00'
        mean = value[0]
        max  = value[1]
        hhi_arr.append([interval,mean,max])
    return hhi_arr

### output functions ###
def write_array_to(array, file, mode='wb'):
    '''write array to file as csv'''
    with open(file, mode) as fh:
        writer = csv.writer(fh)
        writer.writerows(array)
def get_service(port, proto):
    try:
```

```
        return socket.getservbyport(int(port), proto.lower() )
    except:
        return ""
def save_plot(file_name):
    pyplot.ylim(ymin=0)
    pyplot.gcf().autofmt_xdate()
    pyplot.legend()
    pyplot.savefig(file_name)
    pyplot.clf()

def draw_plots(records_by_pport, threshold, router):
    '''make scatterplots of data by pport'''
    colors  = itertools.cycle(['b','r','g','c','m'])
    markers = itertools.cycle(['o','x','+','v','s'])
    lines = 0
    for pport, records in records_by_pport.iteritems():
        pyplot.xlabel('date')
        pyplot.ylabel('hits by hour (normalized)')
        dates = map( lambda x: x[0], records)
        # normalize hits
        hits  = map( lambda y: y[4], records)
        sum = reduce (operator.add, hits)
        hits_norm = map ( lambda x: float(x)/sum, hits)
        port, proto = pport.split(':')
        label = "%s:%s" % (port, proto)
        service = get_service(port, proto)
        pyplot.scatter(dates, hits_norm, label=label, color=next(colors),
marker=next(markers), facecolors='none')
        lines+=1
        if lines%5 == 0 :
            # draw 5 lines on each graph
            save_plot( "/root/bin/data/plots/%s_%s_%s_norm.png" % (router,thresh-
old,lines) )
    # capture remaining lines
    if lines%5 != 0:
        save_plot( "/root/bin/data/plots/%s_%s_%s_norm.png" % (router,thresh-
old,lines) )

def draw_stat_plots(results, threshold, router):
    '''make scatterplots of data by pport'''
    colors  = itertools.cycle(['b','r','g'])
    markers = itertools.cycle(['.','>','<'])
    for pport, records in results.iteritems():
        pyplot.xlabel('date')
        pyplot.ylabel('hits by hour')
        port, proto = pport.split(':')
        i = 0
        for arr in [ records['top_1000'], records['ewma_alerts'], rec-
ords['adapt_alerts'] ]:
            dates = map( lambda x: x[0], arr)
            hits  = map( lambda y: y[4], arr)
            port, proto = pport.split(':')
            i+=1
            if i%3 == 1:
                label = "%s:%s" % (port, proto)
            if i%3 == 2:
                label = 'ewma alert'
            if i%3 == 0:
                label = 'adapt alert'
            pyplot.scatter(dates, hits, label=label, color=next(colors),
marker=next(markers), facecolors='none')
        save_plot( "/root/bin/data/plots/%s_stats_%s_%s_%s.png" % (router,thresh-
old, port, proto) )
def write_array_to_excel(array, file):
    wb = xlwt.Workbook()
```

```
    xf =xlwt.easyxf(num_format_str='DD-MM-YY HH')
    for pport, records in array.iteritems():
        port, proto = pport.split(':')
        label = "%s_%s" % (port, proto)
        sheet = wb.add_sheet(label)
        # insert table header
        records.insert(0, ('interval', 'ip', 'port', 'proto', pport) )
        for row, array in enumerate(records):
            for col, value in enumerate(array):
                if col == 0 and row != 0:
                    sheet.write(row,col,value, xf)
                else:
                    sheet.write(row,col,value)
    wb.save(file)

def write_stats_to_excel(results, file):
    wb = xlwt.Workbook()
    xf =xlwt.easyxf(num_format_str='DD-MM-YY HH')
    for pport, records in results.iteritems():
        port, proto = pport.split(':')
        label = "%s_%s" % (port, proto)
        sheet = wb.add_sheet(label)
        # insert table header in first printed dataset
        records['top_1000'].insert(0, ('interval', 'ip', 'port', 'proto', pport ) )
        for row, array in enumerate(records['top_1000']):
            for col, value in enumerate(array):
                if col == 0 and row != 0:
                    sheet.write(row,col,value, xf)
                else:
                    sheet.write(row,col,value)
        records['ewma_alerts'].insert(0, ('interval', 'ip', 'port', 'proto', 'ewma'
) )
        for row, array in enumerate(records['ewma_alerts']):
            for col, value in enumerate(array):
                col += 5
                if col == 5 and row != 0:
                    sheet.write(row,col,value, xf)
                else:
                    sheet.write(row,col,value)
        records['adapt_alerts'].insert(0, ('interval', 'ip', 'port', 'proto',
'adapt' ) )
        for row, array in enumerate(records['adapt_alerts']):
            for col, value in enumerate(array):
                col += 10
                if col == 10 and row != 0:
                    sheet.write(row,col,value, xf)
                else:
                    sheet.write(row,col,value)

        arr = [( 'n',      records['n']),
               ('ewma_gamma' , records['ewma_gamma']),
               ('ewma_thresh', records['ewma_thresh']),
               ('ewma_max_gap', records['ewma_max_gap']),
               ('adapt_gamma' , records['adapt_gamma']),
               ('adapt_thresh', records['adapt_thresh']),
               ('total', records['total'])
            ]
        for row, array in enumerate(arr):
            for col, value in enumerate(array):
                col += 15
                sheet.write(row,col,value)
    wb.save(file)

########### GLOBAL ###########
# set cores to use per process 24 servers * 3 cores = 72
```

```
# higher for larger datasets
partitions = 1500
threshold = 100
min_pport = 1500
sc = create_spark_context(partitions)

############ MAIN    #########
routers = [ 'sarar4', 'sarar9', 'klantr2-alm', 'alm01-r05', 'saraxs1', 'klantr1-
alm', 'klantr2-asd', 'klantr1-asd', 'sarar1', 'sarar6' ]
# sarar4 = p:any, thr:10, min_pport:10
# alm01-r05    =p:600, thr:20, min_pport:20
# saraxs1      =p:1000, thr:100, min_pport:1000
# klantr1-alm  =p:1000, thr:500, min_pport:1000
# klantr2-asd  =p:1500, thr:100, min_pport:1500
# sarar1       =p:3000, thr:2000, min_pport:1000
execution_times = []
#for router in ['sarar9']:
#for router in [ 'sarar4', 'klantr2-alm', 'alm01-r05'  ]:
#for router in [ 'saraxs1', 'klantr1-alm' ]:
#for router in ['klantr2-asd']:
for router in [ 'klantr1-asd', 'sarar1', 'sarar6' ]:
    t_start = time.time()
    hdfs_dir = "hdfs:///tmp/live/%s" % router
    netflow_records = read_hdfs_files(hdfs_dir, sc)
    # ! parallel
    #intervals = get_sorted_keys(netflow_records)
    #hits = get_high_hits_by_interval_old(netflow_records, intervals)
    # create temp rdd which will be used to
    # - calculate hits/hour ratios
    # - retrieve IP's with highest hitrates (overall)
    # map flow_record to  'timestamp:ip:port:protocol' -> '1'
    new_key = lambda rec: "%s:%s:%s:%s" % (rec[0],rec[4],rec[5],rec[6])
    temp_rdd = netflow_records.map( lambda (k, v): (new_key(v), 1) ) \
           .reduceByKey(operator.add)
    temp_rdd.persist(pyspark.StorageLevel.MEMORY_AND_DISK_SER)
    # GET HITS BY PPORT
    hits_by_pport = get_high_hits_by_pport(temp_rdd, threshold, min_pport)
    draw_plots(hits_by_pport, threshold, router)
    if router in [ 'sarar4', 'sarar9', 'klantr2-alm', 'alm01-r05', 'saraxs1',
'klantr1-alm']:
    #    # only write if it fits into excel sheet
        hhp_xls = "/root/bin/data/%s_hhp.xls" % ( router )
        write_array_to_excel(hits_by_pport, hhp_xls)

    # GET HIGH STATS
    hits_by_stats = get_high_stats_by_interval(temp_rdd, threshold, min_pport)
    draw_stat_plots(hits_by_stats, threshold, router)
    stats_file = "/root/bin/data/%s_stats_%s_%s.xls" % (router, threshold,
min_pport)
    write_stats_to_excel( hits_by_stats, stats_file )

    d_size = float(get_data_size(router))
    t_finish = time.time()
    t_total = t_finish - t_start
    rate = d_size/t_total
    execution_times.append([router, d_size, t_total, rate,threshold, partitions,
threshold ])

write_array_to(execution_times, '/root/bin/data/execution_times.csv', 'a')
```