Low Overhead Distributed IP Flow Records Collection and Analysis

Jan Wrona
Brno University of Technology
Božetěchova 1/2, Brno, Czech Republic
Email: iwrona@fit.vutbr.cz

Martin Žádník
CESNET, a. l. e.
Zikova 1903/4, Prague, Czech Republic
Email: zadnik@cesnet.cz

***Abstract*—Collection and analysis of IP flow records belong to a class of data-intensive tasks, the class for which big data analytics systems should be effective. Several Hadoop-based solutions for network traffic processing exist but are generally suitable only for truly big data, otherwise the disadvantages of Hadoop dominate. In this work, we present a distributed platform for IP flow records collection and analysis together with a reference implementation. It focuses on smaller clusters, has low overhead, allows interactive work, and exploits the prospects of distributed systems like high throughput and scalability. Experiments show low query latency and linear scalability with respect to the growth of both amount of work and computer cluster. Extensions for data mining and machine learning are easy to include and are already work in progress. Moreover, the whole software stack is open-source.**

***Index Terms*—NetFlow, IPFIX, IP flow collector, distributed system, parallel computing, Hadoop, big data**

I. Introduction

Both global and local internet protocol (IP) traffic grow significantly every year and studies show that the growth is expected to continue [1]. To monitor traffic on that scale, data-reducing technologies such as IP flow records or sFlow are necessary. Current solutions in the area of flow records collection and analysis can be divided into two categories: simple centralized and complex distributed. The former is still used by the majority of network administrators, even though it might form a bottleneck in the monitoring pipeline. Flow records exported from large networks may generate considerable volumes of data, but centralized computing is limited by the processing power and I/O throughput of a single node. Moreover, redundancy, high availability (HA), and scalability are hard to achieve. On the other hand, centralized systems usually provide lower overhead and are easier and less costly to deploy and maintain. The latter, distributed and parallel computing scheme, has also found a way to the analysis of computer network traffic due to its notable fitness for data-intensive tasks and much better scalability.

Our research in the area of existent flow-based collectors results in the following conclusion: there are traditional widespread centralized solutions (e.g., NfDump) and there are less mature Hadoop-based distributed solutions (e.g., [2], [3], [4], [5]). Centralized collection software may be a viable option now, but the processing power deficiency of a single server is inevitable. On the other hand, Hadoop will likely be a good choice only for large organizations, otherwise it would probably be overkill. For these reasons both available solutions for flow-based systems are far from ideal.

978-3-903176-15-7 © 2019 IFIP

The gap between the two described options is too wide. Therefore, we propose a low overhead distributed IP flow records collection and analysis system to address this gap by combining the features of both mentioned paradigms. It preserves interactivity, efficiency, and bare metal performance of the centralized system while exploiting properties of the parallel and distributed system (high I/O throughput, redundancy, and scalability). Compared to the experimental implementation of a Hadoop-based collector on a small cluster, our solution utilizes the resources more effectively, has low latency and linear scalability. Additionally, advanced flow records analysis and intrusion detection methods unfeasible for centralized collectors become achievable with the extra processing resources, memory, and I/O throughput.

The rest of the paper is organized as follows. We discuss related work and motivation for our work in Section II, in Section III we propose the distributed collector. We evaluate the reference implementation of the collector, compare it with the Hadoop-based implementation, and discuss the results in Section IV. We conclude in Section V.

II. Related Work and Motivation

The research related to this article usually has the same goal: to make it possible to handle the increasing amount of exported flow records. Earlier there was an effort to obtain the best performance out of the centralized collector [6], [7], [8], but the research has eventually shifted on the flow collectors implemented on top of the distributed system (DS). With the advent of a MapReduce programming model also came the idea of distributed NetFlow processing using the same paradigm. Since it was introduced in 2010 [2], most of the consecutive publications also focus their work on the MapReduce programming model and applications belonging to the Hadoop ecosystem. Such tools for network monitoring ([3], [4]), however, often require the whole dataset to be stored in advance, aim at offline processing, and do not offer capabilities to continuously receive and store the incoming stream of flow records. There are also systems ([3], [5]) which focus on the efficient joining of a large monitoring dataset with external smaller datasets (e.g., IP address to autonomous system number, IP address to domain name). Our work is rather orthogonal as it operates with a dynamic monitoring dataset, concentrates on efficient storage and analysis without joins.

This work is motivated by a rather constrained problem, that is also why the design has to be strongly targeted and none of the existing systems we are aware of is suitable. Nevertheless, this is a problem that many smaller organizations face: the flow records data scale to terabytes at most, which is far away from the big data problem addressed by the complex big data analytics systems (this order of magnitude is sometimes denoted by the term "medium data", but there is neither any well-established definition nor clear boundaries). Widespread centralized collectors are sufficient only for small data, anything bigger hits the scalability issue. Even with a powerful machine and storage area network, ad hoc queries can take several days to finish, which is very limiting for both human users and algorithmic data mining. On the contrary, Hadoop and MapReduce are not efficient in smaller collector deployments. Its well known high latency is unacceptable here because queries which cover only modest database subsets are common and users expect the results promptly. For those reasons, we propose a solution aimed at smaller clusters, which fills the gap between the two described extremes by combining their features. Moreover, our solution is tailored for IP flow collection and analysis, thus avoids the performance penalty which universal data-processing systems usually experience in this domain.

III. The Distributed Collector

In this section, we introduce the distributed collector, a tailored flow-records-processing DS.

A. Concepts and Ideas

This system is designed to run on commodity hardware, which is susceptible to failures. Since nodes can fail at any time, the collector has to deal with hardware or software failures at the system level. Our attitude towards hardware failure is relaxed compared to Hadoop distributed file system (HDFS); to maintain full availability, only one node can fail (temporarily or permanently) at the same time. Multiple failures at the same time will not break the system but will render certain parts of the data unavailable.

Data collection exhibits specific data access patterns (immutability and write once read many), because it essentially means appending incoming records into the database. However, reading operations can be both random and sequential. A data coherency model is simplified by these characteristics, which makes data replication much simpler. Another principle that will be further discussed is "moving computation is cheaper than moving data". The requested information retrieval operation is always executed on the node where the data are stored, which reduces network utilization, decreases response times, and improves throughput.

The collector globally honors the principles of a shared nothing architecture. One of the advantages of this architecture is its possibility to eliminate any single point of failure (SPOF) at the system level. Their complete elimination is the fundamental goal of every system targeted to HA, and so is ours.

B. Architecture Overview

The physical architecture is composed of several independent nodes interconnected by a computer network. The logical architecture is formed by two types of nodes, a *proxy* and a *subcollector* (for an example see Fig. 1). The decision about the layout of logical nodes on top of physical nodes is made by an administrator during cluster initialization.

The proxy node lies on the boundary of internal and external networks. It is a front-end, an intermediary for clients seeking resources on a private network, and a load balancer (LB). The proxy makes the cluster fully transparent, clients making requests may not be aware of the internal structure at all. The subcollector node is a worker/slave and a data carrier. Both the proxy and the subcollectors are fully replicated to maintain HA, which is described later in Subsection III-F.

The minimal number of physical nodes in the collector is two since that is the minimum required to provide redundancy. The architecture also requires at least one proxy node and two subcollector nodes. The minimal number of physical nodes is two because it is possible to use a configuration where an arbitrary node works as proxy and subcollector at the same time. This is the cheapest option, but the load is not balanced uniformly; for that reason, this configuration will not be discussed further. In the rest of this paper, we will consider the configuration shown in Fig. 1, where each node is either a dedicated proxy or a subcollector, nothing is shared.

C. Data Collection

From the data collection point of view, clients are flow exporters. Exporters are continuously transmitting a stream of records towards the proxy, which operates as an LB: it distributes the traffic uniformly across a pool of subcollectors using a round-robin algorithm. The subcollector pool is managed automatically; offline or overloaded nodes are temporarily excluded, which eliminates the loss of incoming data.

Each subcollector works in the same way as a centralized collector does, except the reduced load: it only has to handle $\frac{1}{N}$ of the total volume, where N is the subcollector pool size. Aside from writing the records on local drives, two types of indexes are built. The first type is timestamp-based indexes which are accomplished by simply creating a new data file every five minutes (so-called file rotation). Each file is labeled by a corresponding timestamp to allow skipping as many data files as possible during time-constrained queries. The other type uses a Bloom filter, where the elements of the set are source and destination IP addresses. It is used in filtering queries to quickly test whether the data file contains a certain IP address or not.

The program responsible for the whole data collection process in our reference collector implementation is called

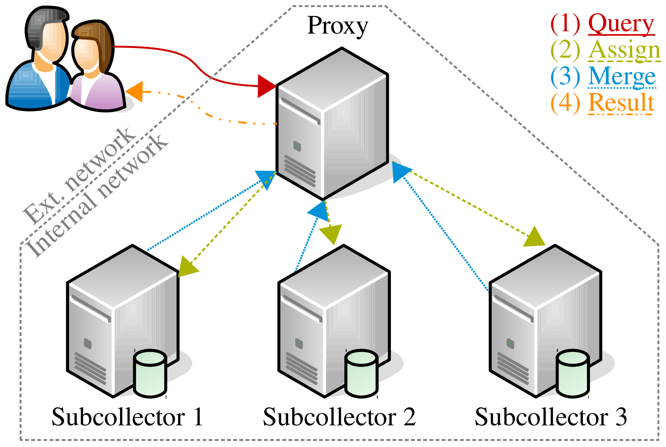

Fig. 1. General steps of information retrieval

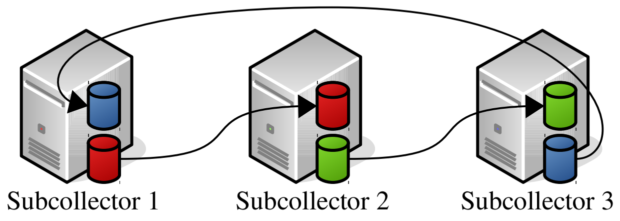

Fig. 2. A ring topology data replication strategy

IPFIXcol [9]. IPFIXcol is a framework for receiving, processing and storing/forwarding flow information records. We implement subcollectors using this framework and we also extend IPFIXcol to work as an efficient application-protocol aware LB on the proxy node.

D. Information Retrieval

The information retrieval process on the distributed collector consists of the following steps (shown in Fig. 1): (1) The query request is sent from a client to the proxy. (2) The proxy forwards the request to all subcollectors. Each query reads data from a certain time range, and since records are uniformly distributed in a time domain, a uniform workload distribution is achieved. (3) After the request is processed on all subcollectors, partial results are sent back to the proxy where a union of all partial results is made. (4) The outcome is presented to the client.

We have identified three elemental operations for information retrieval from flow records: *listing*, *sorting*, and *aggregation*. Another set of derived operations emerges by combining these operations together, applying a record filter and/or a record limit. For example, the combination of aggregation, sorting, and record limit is known as Top-N query. These operations are well known from sequential flow processing tools so it is necessary to provide them also to the users of the distributed collector. However, not all of them are easy to implement on DS, especially when performance metrics like bandwidth consumption and the number of round trips are taken into account.

We have created a tool specifically for the purpose of information retrieval in the distributed collector. It is called `fdistdump` and implements all three mentioned elemental operations:

Listing: Slave processes concurrently read records from local databases, filter them, and immediately send them towards the master. The master process receives and prints records as they come.

Sorting: Similar to the listing, except for immediately sending, slave processes store records into their memory at first. Records are then sorted and sent to the master in a preserved order, where a merge algorithm produces a single sorted list.

Aggregation: Similar to the listing, but except for immediately sending, slave processes store records in hash tables first, where records are aggregated according to an aggregation key. Afterward, records are sent to the master, where the aggregation of pre-aggregated records from all subcollectors takes place.

Combining these operations together is also possible. Emphasis has been laid on the algorithm for the Top-N queries, which is similar to the three-phase uniform threshold (TPUT) algorithm [10]. Apart from node-level parallelism, `fdistdump` also offers thread-level parallelism to fully utilize multi-core and multi-processor computers.

E. Storage

As well as the whole collector, the storage subsystem also follows the shared nothing architecture, thus no physical shared storage is used and each node uses its own local drives instead. This is a cheap and scalable option because with the addition of another node grows not only the processor count and memory size but also the overall storage capacity and throughput. However, without any additional measures, each local drive would introduce SPOF; failure would mean temporarily unavailable data at best, permanent data loss at worst. The strategy used to overcome this problem is called a ring topology replication. We logically arrange nodes into a directed cycle graph, where vertices represent nodes and edges represent where the nodes place their replicas. As can be seen in Fig. 2, each subcollector stores all of "its own" records and a read-only copy of all records belonging to its direct predecessor – a fully replicated scheme. This means that each subcollector stores $\frac{2}{N}$ of records in total, $\frac{2}{N} * N = 2$ means that every record is independently stored on two different nodes.

F. High Availability

The HA functionality is supported by the ClusterLabs cluster stack [11], which provides interaction between nodes, liveness detection, recovery of machine and application-level failure, and more. They are fully configurable and the configuration is part of the proposed system. A substantial part for the collector's HA is a virtual IP address (VIP). The HA stack ensures that it is always assigned to only one node at a time and the VIP thus works as a unique cluster access point.

The proxy and all subcollectors have to be replicated in order to eliminate the SPOFs. The proxy uses an active/passive redundancy model: the service is actively operating only on one node at a time, but some other node is ready to take over the role of the active proxy. The active proxy is in possession of the VIP, running the IPFIXcol LB service and thus is receiving and load balancing flow data. In case of the active proxy failure, the VIP is moved to the passive proxy; this node becomes a new active proxy and at the same time, a new passive proxy is spawned.

Each subcollector runs an IPFIXcol storage service which writes data to the local storage. Subcollectors use an active/active redundancy model: the service is actively operating on two or more nodes, which are sharing the load. In case of a subcollector failure, no action has to be taken: the failed node is excluded from the pool and the other nodes will take over the failed node's load. The collector's database-management system (DBMS) also uses the subcollector pool to construct parameters for `fdistdump`. When a certain subcollector is offline, the DBMS has to instruct `fdistdump` to omit that node and use its replica instead. If the node carrying the replica is also offline, the client is notified, but the query is executed despite the fact that a certain part of the data is unavailable. The DBMS is constantly monitoring the cluster's health, and if a failure is detected, the query is automatically restarted with updated parameters.

IV. Evaluation

Evaluations of a collector performance have been done before, and in our experiments we use the same methodology as in [7], [8]. The objective is to answer questions such as: How fast can the collector retrieve flow records from the storage? What is the latency of a no-op job? How large is the overhead of the DS? What is the efficiency like compared to a conventional centralized solution?

As mentioned above, we are particularly interested in medium data, that is why we have chosen a relatively small commodity cluster for our experiments: six x86-64 computers equipped with four cores, 8 GB of RAM and two 600 GB 10000 RPM SATA hard drives in RAID 0. The nodes were interconnected by a Gigabit Ethernet network.

We have experimented with three different datasets of IP flow records from links between the CESNET2 network and another network with a different type of traffic to create diversity in the datasets. We have also used data from different time periods, but all the combinations produced similar results. Therefore, only one dataset is displayed in our evaluation: it consists of records exported during a single day, that is 878 M of flow records, using approximately 85 GB of disk space in the comma-separated values (CSV) format. We make this dataset publicly available[1].

We have selected three queries that represent typical information retrieval and operations utilized during flow data analysis:

1) A number of records, a sum of packets and bytes.
2) List of source IP addresses with an associated number of records, packets, and bytes, sorted by number of records (aggregation and sorting).
3) List of top 10 source IP addresses with an associated number of records, packets, and bytes, sorted by a number of records.

In fact, we have tested several more types of queries (as described in Subsection III-D) but the results were correlated with one of the selected queries. Query 3 is the Top-N version of query 2 and should demonstrate an advantage of the implemented Top-N algorithm over the general aggregation and sorting.

A. Hadoop as a Flow Analyzer

Before the development of the system presented in this paper, we have performed measurements of several methods and tools from the Hadoop ecosystem in order to evaluate their suitability to serve as a flow analyzer. The goal was to compare this big data analytics solution with a widespread centralized solution.

The centralized NfDump [12] was chosen as a reference collector. To measure Hadoop, we have implemented two versions of the collector: *MapReduce CSV* (expects flow records as comma-separated text strings) and *MapReduce binary* (expects flow records in a custom binary format with constant structure). We have also measured two high-level tools from the Hadoop ecosystem: *Apache Hive* and *Apache Pig*.

The measurement matrix, therefore, consists of three queries, five collectors, and 25 distinct parameters determining the amount of data. Each measurement was run three times and the resulting job completion wall-clock times were averaged. We were launching the jobs sequentially with an increasing time span (i.e., increasing the amount of data the collector has to process) and recording the completion times.

The results of queries 1 and 2 are depicted in Figures 3 and 4, respectively. Query 3 was strongly correlated with query 2 and the graph is omitted. On the left vertical axes are the job completion times in seconds, on the horizontal axes appears the data amount measured in hours of flow records (one hour step corresponds to approximately 36 M of records). In most cases, the completion times rose linearly with the growing time span. Surprisingly, the single-node NfDump performed very well compared to the six-node Hadoop. The best performing for short time spans was the NfDump, but when the time span crossed the break-even point (the point at which NfDump and MapReduce binary are equal), the parallel processing power of Hadoop took over the overhead and the MapReduce Binary was the fastest further along. It is important to note the latency of Hadoop's MapReduce: in such a situation, when NfDump finished in a matter of milliseconds, the Hadoop's latency did not descend below 19 seconds. Moreover, the aggregation and sorting involved in query 2 more than doubled the latency of those no-op jobs.

However, efficiency is more important than the absolute numbers. Efficiency is a metric of the utilization of the

[1]Anonymized dataset in the CSV format is available at https://www.liberouter.org/pub/im_2019_dataset.csv.xz

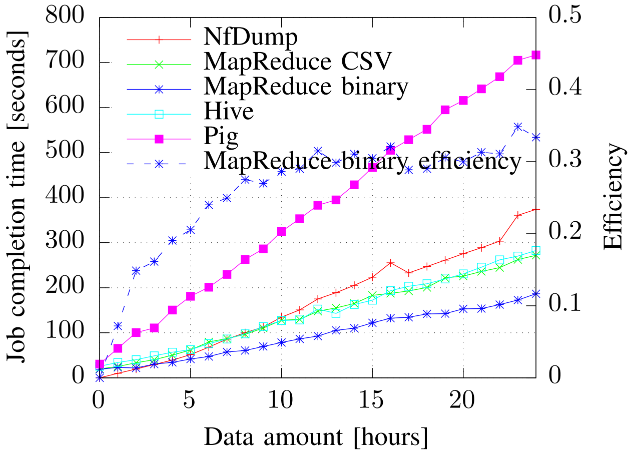

Fig. 3. Completion times and efficiency of query 1 on Hadoop

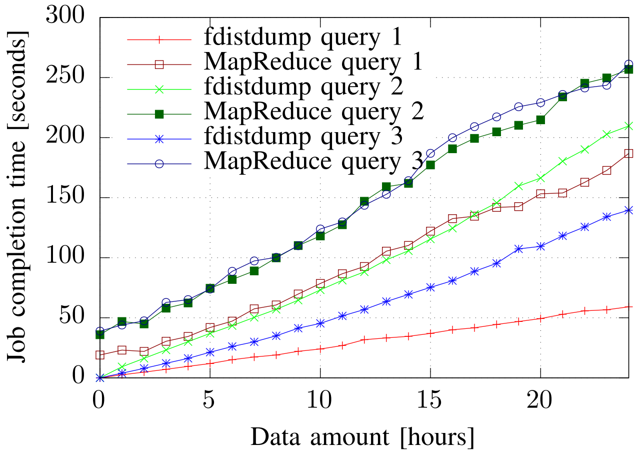

Fig. 5. `fdistdump`'s completion times on constant number of nodes

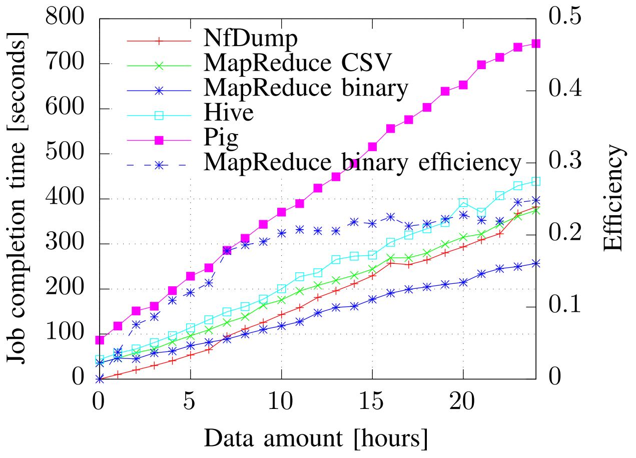

Fig. 4. Completion times and efficiency of query 2 on Hadoop

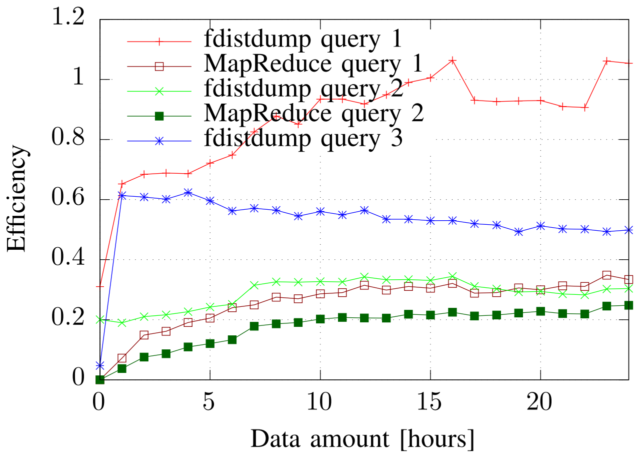

Fig. 6. `fdistdump`'s efficiency on constant number of nodes

resources of the improved system defined as $\eta = \frac{S}{s}$, where S is either a speedup in latency $S_L = \frac{L_{old}}{L_{new}}$ or a speedup in throughput $S_T = \frac{T_{new}}{T_{old}}$. In our measurements, we have used the speedup in latency, where L_{old} is the reference sequential NfDump collector's latency and L_{new} is the improved MapReduce binary collector's latency. The $s = 6$ because we have used six nodes. The values are depicted on the right vertical axes of Figures 3 and 4. While the first query reaches the efficiency of up to 0.35, the second query fluctuates around 0.2. This is probably due to a bigger overhead of aggregation and sorting on distributed systems.

The latency of no-op jobs we experienced is unacceptable because queries which span only modest database subsets are common and users are used to getting results in the matter of milliseconds. Even when it comes to the throughput, our measurements have shown that Hadoop performs worse than a native application (on a per-node basis). Although Hadoop unsurprisingly scales better with respect to the increasing amount of work, the performance was still poor for a six-node cluster. This can be seen on efficiency, where the average utilization of the cluster was at most 35 % for query 1. For the sake of completeness, we have also performed the same measurements on a four times bigger cluster (24 DataNodes) and we observed a single difference in results: execution times rose at a much slower pace. The omnipresent delay lasting at least 20 seconds was still there and the efficiency was even worse.

B. Proposed flow collector

In this subsection, we present an evaluation of the proposed flow collector and a comparison with the MapReduce binary analyzer. The evaluation methodology remains the same to produce comparable results.

Fig. 5 is analogous to Figures 3 and 4. It is obvious from the plot that `fdistdump` performs better than MapReduce. For instance, the latencies of the no-op jobs are ranging from 19 to 35 seconds with MapReduce but with `fdistdump` they are negligible. `fdistdump` also outperforms MapReduce when the amount of data grows.

Fig. 6 compares the efficiency of `fdistdump` and MapReduce queries. In general, `fdistdump` is about two times more efficient compared to MapReduce. Moreover, significant improvement can be observed when the distributed Top-N algorithm is in effect (query 3).

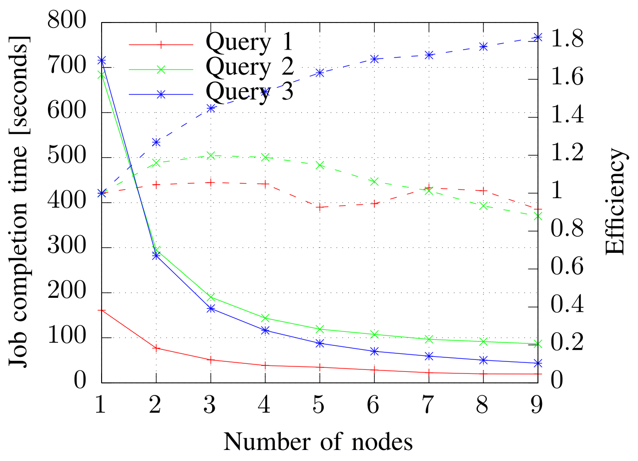

Fig. 7. `fdistdump`'s completion times (solid lines) and efficiency (dashed lines) on variable number of nodes

Fig. 7 demonstrates how `fdistdump` scales with respect to the changing number of used nodes while processing a constant amount of data (the complete dataset). MapReduce results are not included because we could not modify the tested Hadoop cluster to use a variable number of nodes, but previous experiments with 6 and 24 nodes revealed a slight efficiency drop. The figure plots both job completion times and efficiency for all three queries. However, this time L_{old} is the `fdistdump`'s latency on one node and L_{new} is the `fdistdump`'s latency on multiple nodes. Note the difference in the efficiency of queries 2 and 3: on a single node the latency is similar, but with an increasing number of nodes, the lines diverge in behalf of query 3. This confirms that the implemented Top-N algorithm is efficient for flow records.

Another experiment reveals how the throughput of the data collection subsystem scales with respect to the changing number of used nodes. Each node is individually able to handle about 700 K records per second and the overall throughput of the LB rises in this pace up to 2.3 M records, where it hits a limit introduced by a saturated proxy's network connection.

All of the measurements so far are performed on a healthy cluster without any node failures. But since our system provides HA, we also measured how this failure affects the performance. Theoretically, the query on a cluster with a failed node should take about two times longer, since the direct predecessor of the failed node has to process twice as many records. If a failure is detected *during* the query, it is automatically restarted with updated parameters. This can take up to three times longer: the latency of the query on the healthy cluster plus the latency of the same query after the failure. Experiments confirmed these hypotheses with slight variations, but graphs are omitted due to space constraints.

V. Conclusion and Future Work

In this paper, we considered the problem of IP flow records collection and analysis from the perspective of a smaller organization, where the flow records scale to terabytes at most (i.e., medium data). This order of magnitude is too big to be adequately managed by a single node, but our experiments prove that the Hadoop-based distributed solutions are not very efficient in smaller collector deployments.

We created a distributed platform for IP flow records collection and analysis which focuses on smaller clusters, has low overhead, and exploits the prospects of distributed systems. We also developed a reference implementation as a set of open-source software which we publicly provide[2].

To support our statements, we evaluated the collector and compared the results to the Hadoop-based implementation. The information retrieval subsystems of our solution exhibited linear scalability. In contrast to the MapReduce queries, the efficiency was at least doubled.

Distributed computing is effective in data-intensive tasks, and thus the DS collector opens new research directions, which were not possible before. We are currently working on several data mining and machine learning techniques, heading towards an automated distributed flow-based intrusion detection.

Acknowledgment

This work was supported by Brno University of Technology grant under number FIT-S-17-3994.

References

[1] Cisco Systems, Inc. (2017, Jun.) The zettabyte era: Trends and analysis. Retrieved February 17, 2018. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html
[2] Y. Lee, W. Kang, and H. Son, "An internet traffic analysis method with MapReduce," in *IEEE/IFIP NOMS Workshops*. Osaka, Japan: IEEE, Jan. 2010, pp. 357–361.
[3] D. Sarlis, N. Papailiou, I. Konstantinou, G. Smaragdakis, and N. Koziris, "Datix: A system for scalable network analytics," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 21–28, Sep. 2015.
[4] A. M. Hendawi, F. Alali, X. Wang, Y. Guan, T. Zhou, X. Liu, N. Basit, and J. A. Stankovic, "Hobbits: Hadoop and Hive based internet traffic analysis," in *2016 IEEE International Conference on Big Data*. Bethesda, MD, USA: IEEE, Dec. 2016, pp. 2590–2599.
[5] G. Touloupas, I. Konstantinou, and N. Koziris, "RASP: Real-time network analytics with distributed NoSQL stream processing," in *2017 IEEE International Conference on Big Data*. Boston, MA, USA: IEEE, Dec. 2017, pp. 2414–2419.
[6] L. Deri, V. Lorenzetti, and S. Mortimer, "Collection and exploration of large data monitoring sets using bitmap databases," in *International Workshop on Traffic Monitoring and Analysis*. Zurich, Switzerland: Springer, Apr. 2010, pp. 73–86.
[7] R. Hofstede, A. Sperotto, T. Fioreze, and A. Pras, "The network data handling war: MySQL vs. NfDump," in *Networked Services and Applications–Engineering, Control and Management*. Trondheim, Norway: Springer, Jun. 2010, pp. 167–176.
[8] P. Velan, "Practical experience with IPFIX flow collectors," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. Ghent, Belgium: IEEE, May 2013, pp. 1021–1026.
[9] P. Velan and R. Krejčí, "Flow information storage assessment using IPFIXcol," in *AIMS 2012: Dependable Networks and Services*. Orlando, Florida: Springer, Jun. 2012, pp. 155–158.
[10] P. Cao and Z. Wang, "Efficient top-k query calculation in distributed networks," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. St. John's, Newfoundland, Canada: ACM, Jul. 2004, pp. 206–215.
[11] ClusterLabs. Open source high availability cluster stack. Retrieved February 6, 2018. [Online]. Available: http://clusterlabs.org
[12] P. Haag. (2014, Dec.) NfDump. Retrieved February 5, 2018. [Online]. Available: http://nfdump.sourceforge.net/

[2]https://github.com/CESNET/SecurityCloud