goProbe: A Scalable Distributed Network Monitoring Solution

Lennart Elsen, Fabian Kohn
Open Systems AG
{*lel,fko*}*@open.ch*

Christian Decker, Roger Wattenhofer
Distributed Computing Group, ETH Zurich
{*cdecker,wattenhofer*}*@tik.ee.ethz.ch*

***Abstract*—The Internet has developed into the primary means of communication, while ensuring availability and stability is becoming an increasingly challenging task. Traffic monitoring enables network operators to comprehend the composition of traffic flowing through individual corporate and private networks, making it essential for planning, reporting and debugging purposes. Classical packet capture and aggregation concepts (e.g. NetFlow) typically rely on centralized collection of traffic metadata. With the proliferation of network enabled devices and the resulting increase in data volume, such approaches suffer from scalability issues, often prohibiting the transfer of raw metadata as such. This paper describes a decentralized approach, eliminating the need for a central collector and storing local views of network traffic patterns on the respective devices performing the capture. In order to allow for the analysis of captured data, queries formulated by analysts are distributed across all devices. Processing takes place in a parallelized fashion on the respective local data. Consequently, instead of continually transferring raw metadata, significantly smaller aggregate results are sent to a central location which are then combined into the requested final result. The proposed system describes a lightweight and scalable monitoring solution, enabling the efficient use of available system resources on the distributed devices, hence allowing for high performance, real-time traffic analysis on a global scale. The solution was implemented and deployed globally on hosts managed and maintained by a large managed network security services provider.**

***Keywords*-network monitoring, map-reduce, netflow, realtime**

I. Introduction

Network monitoring is an important tool that allows a network operator to have a better understanding of its network infrastructure. Monitoring enables problem diagnosis, capacity planning and attack analysis, by allowing transparent and efficient access to the abundance of information produced in the network. A network operator can query the server, or the server can actively raise an alarm if a potential anomaly is detected. As such the operator can detect early if, e.g., a switch is down or slow (failure detection), routing should be improved by means of traffic engineering (efficiency), or whether there is an ongoing distributed denial of service attack (security).

Even in a relatively small network it is not feasible to send all the traffic information to the server. Traditionally, network analysis tools make use of network probes exporting traffic information, using, e.g., the NetFlow standard. In a typical NetFlow infrastructure packets routed through a monitor located in the network are analyzed and combined into sets of packets that share the same attributes, called *flows*, and transferred to a central collector for later inspection.

Nevertheless, if a network is large enough, even aggregated flow information produced by NetFlow systems poses a scalability problem. In this paper we propose a new architecture called goProbe that gives real-time access to the collected information while scaling with the number of monitored networks. In goProbe, the data is stored locally at the probing nodes, and the central server is requesting the network data whenever needed. We show that goProbe scales well, even to global networks with thousands of nodes. Since aggregated flow information is only transmitted when needed, we can generally save considerable network bandwidth. This eliminates the passive background traffic that would be needed to ship flow information to a central collector. In addition, flow information is not lost should the connectivity to the operator be interrupted at any time, restoring forensic analysis capabilities once the connectivity has been re-established.

On the other hand, data analysis queries are distributed to the network routers on which the relevant data is retrieved from the respective local databases. Significantly smaller aggregated results are returned to the query coordinator which collects the aggregated results and performs post processing operations.

The system was fully implemented, including a network probe that captures individual packages from the monitored interfaces and collates them into flows, a columnar database for simple and fast storage of flow information, as well as a comprehensive querying system that makes use of a Map-Reduce like framework to distribute queries to hosts holding relevant information. The proposed system was deployed in the network of a large managed network security services provider. The evaluation shows that goProbe outperforms classical, centralized monitoring solutions with respect to scalability and generated traffic.

II. Collection of Network Traffic Information

In this section we give an overview on how network traffic information is captured and analyzed in classical monitoring

solutions and introduce our specific scenario, which lead to the design of goProbe.

The first step for any monitoring solution is to gain access to raw packets sent and received by a single host. This is achieved through a packet capture process attached to one or more network interfaces which records the packets flowing in and out of it. The hosts under discussion are usually installed in central locations of corporate networks as well as branch offices and thus typically experience traffic far beyond the amount of commodity end user devices. These hosts are firewalls, secured gateways or proxies, which process thousands of packets per seconds. To analyze traffic on a per-packet level would be infeasible due to the amount of routed data. A first aggregation is to only consider metadata of the communication between endpoints, ignoring the payload. This significantly reduces the amount of information that needs to be stored, yet allows for complex analysis. It also implies that packets need to be classified based on a set of attributes such that traffic patterns can be established. An existing standard which sets out to achieve this is the NetFlow standard [1]. With NetFlow the packets are grouped into *flows* based on attributes they share, which, in the case of a network flow, comprise the following: network interface, source and destination IP address, IP protocol, source and destination port (if available) and type of service. Each flow maintains counters which track the data volume and the amount of packets being recorded. In addition, flows are assigned a lifetime which will mark the flow as expired once it is exceeded.

In order to track traffic information over longer periods of time, NetFlow data needs to be archived and made available for later analysis. Companies implementing NetFlow usually follow an exporter-collector model. Once a flow expires, either by timing out or being closed explicitly, it is exported by sending it to a central location, the NetFlow collector, which stores all received flow records in a central database. The situation is outlined in Figure 1.

Operators of the networks query the collector for information to analyze the flow records received by it. This enables analysts to formulate high-level queries regarding the most active endpoints or most abundant applications on the network in terms of traffic volume or packet rates. These queries are processed by the collector, returning the desired result and providing a clear picture about the network traffic composition based on its key features.

The export of NetFlow records generates additional traffic on the network paths between exporter and collector. For networks in which sufficient bandwidth is available and in networks that are geographically proximal this may not present any issues. However, if NetFlow exporter and collector are situated at two entirely different locations across the customer WAN and are possibly connected over lossy links, NetFlow records may be lost. Moreover, additional network traffic can impose additional costs on links which

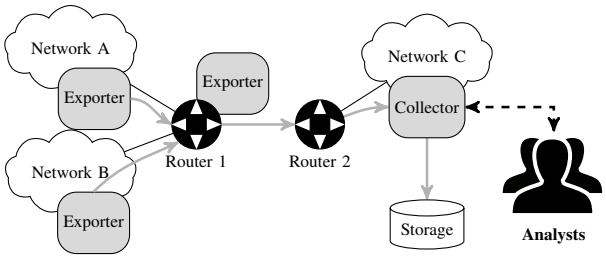

Figure 1. General scheme for NetFlow exporting and collection. The exporters are placed in their respective networks and capture traffic and aggregate it in NetFlow records. The records are exported to a central collector possibly placed in a different network. The grey arrows indicate the path of the NetFlow records. The collector receives these records and stores them in a storage back end.

are provisioned by traffic volume rather than by bandwidth, e.g., when using satellite links in remote locations.

Centralized flow and cross-customer collection becomes infeasible as thousands of hosts spread around the globe are monitored. Most of the networks in which the hosts are deployed are owned by the companies which make use of the provided hardware. Consequently a part of the customer's bandwidth would be used for NetFlow exports in a classic scenario. Furthermore, the handling of export traffic from a large number of hosts would require substantial bandwidth and accumulate large amounts of traffic on the collector side. This requires the availability of a redundant high-bandwidth link to allow for stable NetFlow collection, which does not scale well with the number of monitored networks.

Open Systems AG provides managed network security services to a wide variety of companies, ranging from small startups to globally operating companies and NGOs. In this context, the capability to monitor customer networks for debugging, provisioning and threat analysis & mitigation is central to operations.

Due to the large number of networks which are managed the monitoring solution needs to scale with the number of deployed service locations. In addition the monitored networks are owned by the customers making it prohibitive to send and receive large amounts of data, as this would effectively be consuming customer bandwidth. This is compounded by the fact that some customers are connected over expensive satellite links or over lossy links, resulting in additional costs to the customer or flow information being lost in transit to a central collector.

The solution we propose is to decentralize the flow collection and provide local views of the data. This entails flow capturing and storage be performed on the same device, alleviating the need for export and collection functionality. Since flow data only has to be accessed in order to satisfy an analytic query, only then should it leave the boundaries of its network – condensed, in the form of an aggregate end result.

goProbe enables high-performance retrieval and analysis

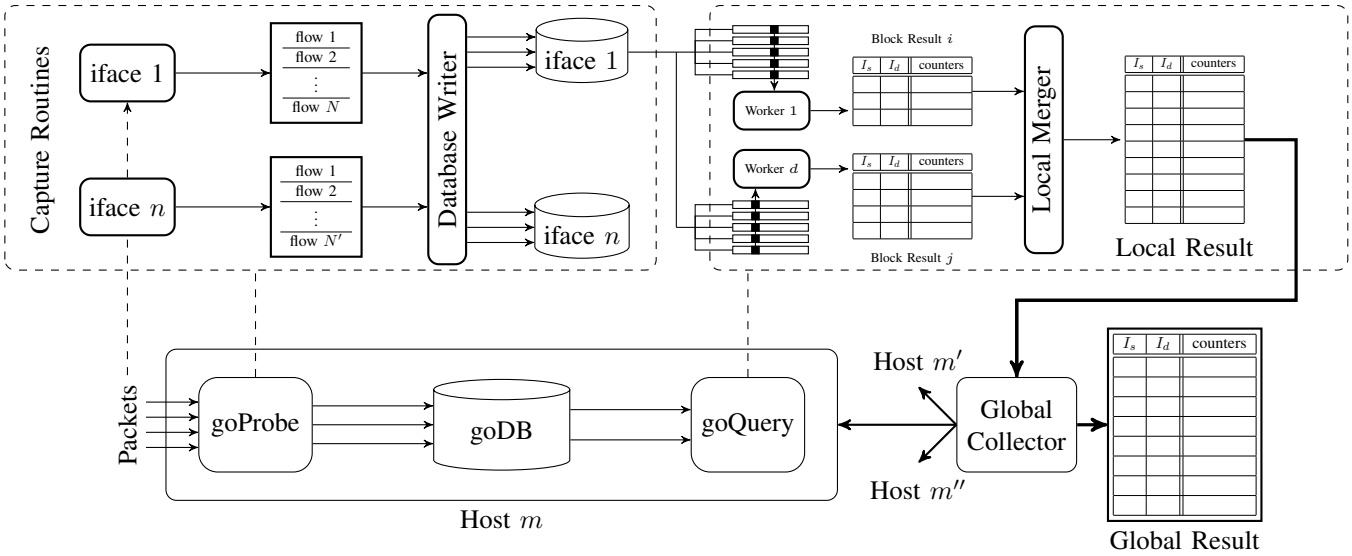

Figure 2. An overview of the architecture including goProbe for the capture, the local goDB database, the local goQuery instance and the global goQuery aggregator.

of the data while keeping the impact on system resources minimal. Reliable traffic capture is guaranteed without interference with other operationally critical services running on the device and an acceptable memory footprint is maintained by only reserving memory for processing and storage of the essential attributes used for the flow logic.

To enable a real time analysis of the recorded flow data, database queries are executed quickly and efficiently. The database stores data in a way that facilitates efficient loading of data. Furthermore, the database features concurrent and independent processing of data in order to accelerate query execution by fully utilizing the available computing resources.

The database may not grow indefinitely as disk space is limited on the hosts. As a remedy, information lifecycle management is necessary, which results in the retention of the last d days of data and the deletion of outdated entries. Data compression is employed to further reduce the database size on disk. Additionally compression can be used to alleviate disk I/O limitations when data is loaded into memory. If less data is loaded from disk it can be processed in memory sooner and faster query execution times can be achieved [2].

III. System Design

In order to address the described scenario, a comprehensive data capturing and processing system for flow data was implemented. The solution enables real time queries against captured data, yet is lightweight enough to run alongside operationally critical services. The system comprises a number of specialized components:

- ***goProbe***, performing capture of packets, collation into flows and storage management on local disk;
- ***goDB***, a columnar database library that manages flow data on disk;
- ***goQuery***, allowing for Map-Reduce-like query processing of queries against the data on the hosts;

These components run on the network device that is being monitored, keeping information local and responding to incoming queries from the central coordinator. Both goProbe and goQuery are not CPU bound, with goProbe polling for incoming packets and goQuery loading the necessary information from disk. In order to exploit the blocking nature of these tasks the system was implemented in `go`, which supports cooperative routines, called *goroutines*, that actively yield control of the processor to other goroutines upon blocking operations. Furthermore, goroutines are lighter than OS level threads and allow greater control over the impact the tool has on operationally critical applications by limiting memory consumption and restricting of capture operations to a single processor core.

A. goProbe

The goProbe tool is used for capturing packets, extraction of packet attributes and to perform maintenance of the processed information in a flow table. It relies mainly on the information obtained from the packet headers, effectively reducing the amount of data that has to be captured. Furthermore, packets are immediately discarded after the relevant information has been extracted. The primary design goal of goProbe was to store only the absolute minimum amount of information necessary to maintain flow records and consequently reduce the flow tracking complexity. This was achieved by implementing a simplified flow format.

goProbe runs as a single process being capable of han-

dling the capture of packets on multiple interfaces. The design features a lightweight deep packet inspection (DPI) engine called `libprotoident` [3], which only requires minimal payload information to correctly infer the application layer protocol in a large number of cases.

In the text, the following variables will be used to describe attributes of a packet: I_s for the source IP address, I_d for the destination IP address, P_s for the source port, P_d for the destination port, P_I for the IP protocol and L_7 for the application layer protocol.

Packet capture on an interface is performed using `libpcap` [4], a library which provides a `C` API interface to raw packets captured from the network hardware by the underlying operating system. The `libpcap` library supports packet filtering on kernel level. A widely used filter is the Berkeley Packet Filter (BPF) [5]. For the given use case, the following filter was used in order to exclude address resolution (ARP) and monitoring traffic (ICMP, ICMP6):

```
not arp and not icmp and not icmp6
```

Both of these protocols are frequently observed in a network and can create large amounts of small and hence inefficient flows conveying only small amounts of actual information with little analytical value. Hence, these protocols were chosen to be ignored.

The kernel has the ability to directly truncate captured packets, which means that only the first few bytes are made available via `libpcap`. This parameter is called *snaplength* and is useful when mostly the headers of a packet are examined, as is the case with goProbe. A smaller snaplength implies that more packets can be held in the kernel packet buffer and thus, a higher packet rate is possible before packets have to be dropped by the kernel. Consequently, more packets can be processed by `libpcap`. The snaplength is set to 90 bytes in order to ensure that the packet headers are completely captured as well as parts of the payload are available for deep packet inspection.

In addition to the packet data, `libpcap` stores meta information about each packet, such as a capture timestamp, the amount of bytes captured and the length of the original packet. The Linux kernel uses a flag specifying the direction in which a packet traverses the network interface. Although `libpcap` in its current implementation uses this flag to filter inbound or outbound traffic, it does not expose it via the API. Identification of the packet direction is crucial for the identification of global traffic flow patterns throughout the network. In order to make the flag available to goProbe, `libpcap` was patched and the data structure storing the capture meta information extended by a corresponding field, set if an observed packet was inbound w.r.t. the network interface. The benefit of identifying the packet direction at capture time is that no additional correlation with the network topology is needed in later stages of an analysis.

A capture routine in goProbe is attached to a single interface. As an extension to this concept, multiple capture routines are run concurrently. This is achieved by execution of the respective code in a `goroutine` for each interface taken into account. Each of the capture routines stores a local flow table which holds all flows for the respective interface.

A flow is defined through packet attributes, packet counters, data volume counters and the application layer protocol. The packet and data volume counters describe the total amount of packets that were observed in the flow and the cumulative traffic volume of these packets, respectively. Four counters exist in total, covering the number of packets received and sent as well as the traffic volume for both directions. The counters are incremented independently based on the direction flag set for the respective packet.

For a packet p_{t_i} captured at time t_i, the attributes uniquely identifying origin and destination of the packet can be summarized in a 5-tuple [6]:

$$h^p_{t_i} = (I_s, I_d, P_s, P_d, P_I)$$

If no corresponding flow for p_{t_i} is found in the flow table, a new entry f_{t_i} is allocated and initialized with the respective metadata, identified by its attribute tuple:

$$h^f_{t_i} \leftarrow h^p_{t_i}.$$

A packet p_{t_j} captured at time $t_j > t_i$ matches flow f_{t_i} if and only if $h^p_{t_j} = h^f_{t_i}$. In that case, the counters for flow f_{t_i} are updated depending on the value of the inbound flag.

To incorporate return packets into a flow an additional check is necessary. For a return packet p_{t_j}, the source and destination attributes from the initial packet p_{t_i} are reversed, implying that $h^p_{t_i} \neq h^p_{t_j}$. Consequently, a reverse tuple r was introduced, which was computed for every packet and for which source and destination attributes were exchanged. For a given packet p_{t_j} both $h^p_{t_j}$ and $r^p_{t_j}$ were validated against the flows in the flow table. The matching condition can thus be formulated as follows: a packet p_{t_j} captured at time $t_j > t_i$ matches flow f_{t_i} if and only if the following holds:

$$(h^p_{t_j} = h^f_{t_i}) \vee (r^p_{t_j} = h^f_{t_i}).$$

The goProbe flow table was implemented using a map data structure. The attribute tuple of the flow was used as map key, while the counters, the application layer protocol and the direction flag designated the map value. `go` makes use of Advanced Encryption Standard (AES) [7] native instructions to hash all keys in a map, making lookups in the flow map highly efficient due to the hardware-accelerated implementation of AES instructions in most modern CPUs.

When packets from an ongoing connection between two endpoints are captured, it is not clear by which endpoint it was initiated since source and destination attributes in a packet are always set relative to the sending endpoint. By setting source and destination attributes in a flow relative to the endpoint which initiated the conversation correct identification of the origin of the packets was made possible.

Many network applications implement a client-server model where one endpoint requests information, while the other one provides it in the form of a response. The endpoint which provides information usually listens for incoming connections on a designated port. Often the application uses ports from the range 1 to 1023 [8], which are system ports used for well-known types of network services such as `SSH` or `HTTPS`. If a packet with such a low destination port is captured, a well-known network service is assumed to be addressed and the source IP of the packet coincides with the initiator of the conversation. In that case, the source port will mostly be chosen from the range of dynamic ports, i.e., those above 49152 [6].

A heuristic procedure was implemented in goProbe in order to establish whether a packet is a return packet or was sent by the initiator of the conversation between the two respective endpoints. The latter is chosen as the packet direction if any of the following conditions are met:

- $(P_I \in \{\text{TCP}, \text{UDP}\}) \wedge (P_d < 1024) \wedge (P_s > 20000)$
- $(I_d = 224.0.\{0,1\}.x) \vee (I_d$ = $255.255.255.255)$,

where $0 <= x < 256$. If the IP protocol is neither UDP nor TCP, a port based packet heuristic is not possible. The bound for the dynamic source port range was lowered to 20000 as source ports below 49152 were observed frequently in the use case environment. The second condition addresses situations where a broadcast or multicast IP address was the recipient of a packet. As these addresses are used to describe a set of hosts, the source IPs in the reply packets will originate from individual hosts instead of a multicast address, and will be tracked in separate flows. The direction of these packets was stored by goProbe in a corresponding flag, which was set to zero in the above case. The packet is tagged as a return packet if the following applies:

- $(P_I \in \{\text{TCP}, \text{UDP}\}) \wedge (P_s < 1024) \wedge (P_d > 20000)$

In this case the described flag is set to one to indicate that the packet originated from the recipient of the initial packet.

The direction of a flow was set according to the packet flags and designated as unknown per default. The direction heuristic was run on every packet until it was possible to establish from which end the communication was initiated. This information was stored and if the flag was set to one, the flow attributes specifying source and destination were reversed upon writeout of the flow, reflecting the reversed flow direction.

In addition to the packet headers from which most attributes were extracted, the packet payload was used to identify the OSI model application layer protocol. In contrast to other engines which require the inspection of the entire packet payload, `libprotoident` only uses the first four bytes of the payload and uses heuristics which correlate the partial payload with information about the port, IP and payload length. Consequently, the attribute tuple which is established by the capture routine can be passed directly to the library along with four bytes of the payload.

For each supported protocol, `libprotoident` contains a module defining specific validation rules and byte pattern against which the payload is checked. If a module does not return a positive match the next module is checked until a match is found or no further modules remain. The modules are prioritized depending on the prevalence of the application described by it.

In `libprotoident` application layer protocol detection is always performed on bidirectional flows which explicitly distinguish between request packets that were sent by one endpoint and response packets sent by the other endpoint. In order to satisfy the bidirectional flow structure, a sliding window approach was implemented in goProbe. If a packet was added to an already existing flow, the attributes, payload lengths and payload bytes from the two most recent packets were passed to the engine. If a new flow was created by a packet, the response payload bytes and payload length were not set. Matching was performed until an application layer protocol was identified by `libprotoident`, otherwise the flow application protocol was classified as unknown.

Several conversations between two identical endpoints may involve different application layer protocols but occur on the same destination port (e.g. protocol multiplexing). The source port, however, is unique for each conversation and can be used to distinguish different protocols in different flows. If two flows describe different application content, the source port is necessary to quantify this difference since the application layer protocol is not part of a flow's attribute tuple.

Prior to writing flows to the database, the source port information is discarded, while for the aggregation of flows the application layer information is taken into account. Flows sharing all attributes and application layer information but the source port are merged into a single flow. The aggregation is performed in memory.

Aggregated flows are written out to disk in regular intervals. A timer structure sends a write signal to the interface capture routines every five minutes. The capture routines respond to the signal by aggregating flows in the respective flow tables. Each flow field is stored in an array, following a columnar structure. The attributes, flow counters and application layer information thus produce nine arrays in total and each store a block of flow data corresponding to a time frame of five minutes. The prepared block is tagged with a timestamp t_w and the interface from which the data originated is specified. The timestamp is defined as the time of data write out, implying that the data block holds flow data acquired during the interval $[t_w - 300\text{s}, t_w]$. The prepared block is passed on to a database writer, appending the individual arrays to separate binary column files. Once a flow table is written out to disk a new, empty table is created which is filled over the next 5 minutes.

B. goDB

The *goDB* database backend provides a lightweight database with a custom query format and data storage model tailored to the acquired flow data with respect to storage and data access.

It can be shown that columnar databases are suited for scenarios in which data is written often and queried rarely [9]. Additionally, a file based database exhibits a well-arranged storage concept for the kind of data under consideration and a generally simpler process model. goDB does not provide a complex, continuously running service and is only run upon query execution, within a single process. Furthermore, data is stored in binary format in columnar files, where one file is created per flow field per day. This way of data partitioning is useful in query scenarios that span time intervals, based on which folders can be excluded that only store data created outside of this interval. Furthermore, data lifecycle management is simplified, since the least recent day can be discarded on file system level by removing the corresponding folder, incurring close to zero overhead.

Within the columnar files maintained by goDB, data is partitioned into compressed blocks. A block-wise partitioning of data is advantageous with respect to both preselection and parallel processing. Since only a subset of all blocks has to be loaded into memory simultaneously, the memory footprint of the query process is greatly reduced. goDB allows for an arbitrary number of entries to be stored within a block to accommodate flow tables of different size acquired during the respective time intervals.

Concurrent processing of data is imperative to reducing query runtime and provides the fundamental building block of goDB for both data writing and processing during queries. The database solution introduces worker and merge modules for query workload distribution in a Map-Reduce like framework, where blocks are processed in parallel and partial results are read by a central merge routine, producing a consolidated result and performing sort and limit operations.

Flow data accumulated within a five minute time interval is passed to goDB in the form of column arrays. Each of these arrays is appended to a designated file corresponding to the respective flow field. Timestamp information is used to locate the folder holding the files into which the data has to be written. All column blocks were concurrently compressed and written to the respective files. Compression is applied on block level, i.e. every column array is passed to a compressor routine returning the compressed data in the form of a byte array. The compressed binary data is then appended to the columnar file for the respective flow field. The compression algorithm in use is `LZ4` [10] which employs dictionary based encoding. One of the advantages of `LZ4` is that very fast compression and in particular decompression speed can be achieved while maintaining a high compression ratio.

The goProbe file (gpf) format: To enable access to specific blocks inside a flow field file decompressor routines need information about which part of the file to decompress. Thus, each `gpf` file contains a file header describing the consecutive stored data. This header is divided into three parts:

- **Identification:** provide information whether the block written at time t_w is present in the file
- **Localization:** provide the locations within the file at which data from block t_w starts and ends
- **Description**: provide the length of the uncompressed block for efficient decompression

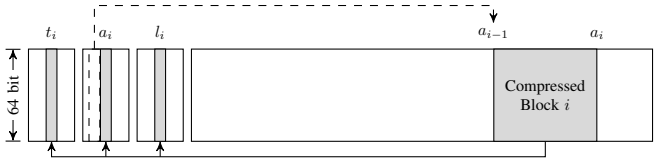

Figure 3. Structure of the `gpf` file. The i-th block is written at position $a_{i-1}+1$, which is retrieved prior to writing. Upon completion of the write operation, the block information is inserted at the corresponding positions in the header.

Within one day, 288 five minute blocks are written to the database files. The header sections reserve 512 slots for their description, where the remaining 224 slots are reserved in case additional data writes are triggered in goProbe. The block identifier t is a 64 bit unsigned integer in which the timestamp of the written block is stored. The block localizer a is a 64 bit address which stores the location of the last byte of the written block. The following block is then inserted one byte after its preceding block localizer. The block descriptor l stores the size of the uncompressed block in a 64 bit unsigned integer. This variable 3-tuple is computed after every block write and only stored in the header if writing is successful. Incompletely written data would thus be overwritten during the next write process.

Consider the i-th block of length l_i which is prepared for write out at time t_i. Figure 3 illustrates how information is written to the file and how the header is updated.

It is noteworthy that writing can be executed in parallel since the different flow fields, i.e., the columns, are written independently to their respective `gpf` files. Correct alignment of the blocks can be verified by examining the timestamp values for all flow fields and checking whether all header fields match at a given position i.

C. goQuery

The command line tool goQuery enables real-time queries against the data collected by goProbe on each individual host. Queries are processed in a two step Map-Reduce like framework. goQuery consists of two components: a global coordinator distributing queries on remote hosts and a local processor running on the capture hosts processing queries based on the local data.

Upon receiving a query from a user, it is analyzed by the global coordinator, identifying the relevant target hosts for

the query, i.e. all hosts which store information relevant for the query. The actual query parameters are then distributed to the individual hosts using the Balu [11] message routing system.

The goQuery processor on the capture host receives the query from the global coordinator and starts a first Map-Reduce step. Similarly to writing, data read operations are performed in a block wise fashion. The local query manager analyzes the incoming query and identifies the timespan and set of fields that are necessary. It then locates the blocks that are to be read by inspecting the headers of the `gpf` files. Should the timestamp of a given block coincide with the query time span then its location and its uncompressed length is added to a block loading plan, grouping blocks by `gpf` file. Each group results in a new worker goroutine being instantiated to load and process the data from disk, reducing the number of long seeks and maximizes disk throughput by transferring this task to the I/O scheduler of the operating system. Since for each block the uncompressed length is known the byte array in which the decompressed information will be stored can be pre-allocated without the need for dynamic resizing.

After loading the required fields from disk, the map step is performed by filtering flows based on their attributes, projecting flows according to the query and merging flows that are projected on the same result. The block result table is filled in analogy to the flow table in goProbe. An in-memory map is created in which the projection of the fields is used as a key and the aggregation results are used as values.

Once a block result table has been computed it is sent to a local merger routine, which then performs the first reduce step by collecting all block result tables from the workers and by computing the local result table. An in-memory local result table is allocated and for each block result a lookup in the local result table is performed. Should the local result entry not exist it is initialized, otherwise the result is added to the result for aggregation. Finally, the local result table is streamed to the global coordinator as JSON container making use of Balu.

A second Map-Reduce step is performed by the global coordinator routines. Incoming local results are received from the local processors and final filtering, aggregation, sorting and cropping of the results is performed before returning the result to the user issuing the original query.

A schematic outline of the entire system can be found in Figure 2.

IV. Evaluation

The proposed system was implemented and deployed on the infrastructure of a large managed network security service provider, which services customers ranging from small startups to global companies and NGOs. One customer network was chosen for the evaluation of goProbe. The customer's infrastructure comprises 290 routers in a variety of networks, each running the goProbe software. The collected data covers the internal network interface servicing the corporate network on each of the routers.

During the measurement period of a week the routers transferred a total of 99.5 TB. goProbe collated the individual packets into flows and stored the metadata of the flows, resulting in a goDB database with a cumulative size of 5.2 GB. A single database thus requires a mere 18.4 MB per router on average. If a centralized collector had been used this data would have to be transferred over the customer's network, adding to potential congestion. With goProbe no traffic is produced until the information needs to be accessed.

Notice that above comprises data of a single interface on each router. A typical router has between 4 and 300 interfaces which would also be monitored, resulting in proportionally larger datasets. Furthermore, the 5.2 GB constitute the compressed size of the flow metadata, while the network traffic resulting from a exporter-collector infrastructure is far higher. In total, goProbe runs on 3164 globally dispersed routers. Considering only the outgoing interface of these routers, it records and stores metadata of 4.4 PB of raw traffic over a period of 90 days, resulting in a cumulative database size across all routers of only 340.0 GB.

In order to gauge the gains in scalability the system was assessed by comparing a decentralized analysis of the data with a centralized analysis. To perform this test the flow data collected during the measurement period was transferred from the respective hosts to a central location in which it was merged into a single goDB database. To compare the performance gains achieved by distributing the query processing over a large number of devices with respect to the centralized collector, a single query was executed multiple times in both scenarios. The used query covered the aggregation of all unique (I_s, I_d, P_d, P_I) tuples, thereby representing a processing intensive query, since the majority of database columns are involved in it, maximizing the amount of data to be read from disk.

In the decentralized scenario the query was distributed to the individual hosts which then calculated the partial results which were merged upon receipt at the central location. In the centralized scenario, the query was executed directly against the central database without performing any network operations.

The hosts on which distributed queries were run featured commodity hardware, whereas the central database host featured more specialized hardware w.r.t. to processing capacity and available memory. The test host was equipped with eight cores and queries were executed in parallel making use of all cores. All of the used devices used a conventional hard disk drive (HDD) for data storage.

The primary metrics recorded were the overall execution time of the query and the amount of data that was necessary to transfer from the hosts in order to obtain the final result. In the centralized solution, the query was run a total of five

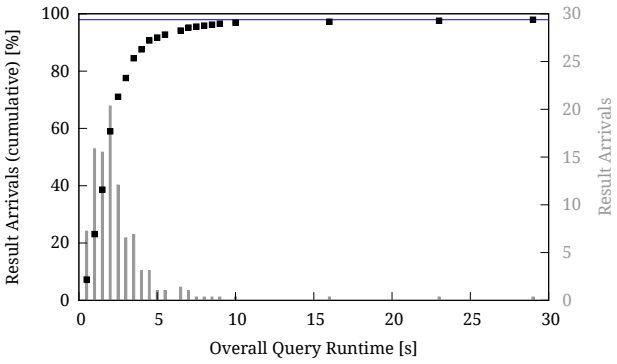

Figure 4. Cumulative density function of sub-result arrivals over time. The number of arriving results per unit of time is aggregated in bins of 0.5 seconds and shown on the right axis. The separator line indicates how many hosts were reachable.

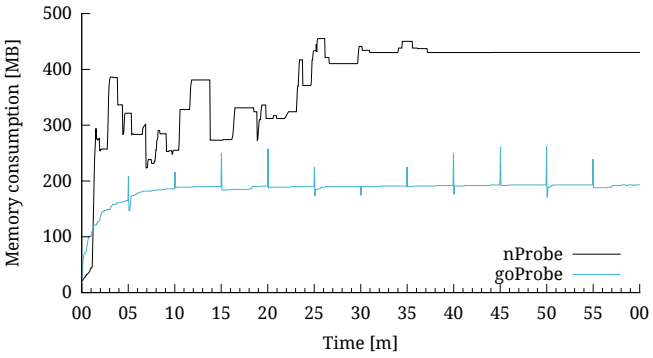

Figure 5. Comparison of the memory usage of goProbe and nProbe over time. The regular spikes in goProbe's usage show the database write operations every five minutes in which additional memory is required for source port aggregation.

times, yielding a mean recorded execution time of

$$(159.3 \pm 4.7)\ \mathrm{s}$$

The response time distribution of the query sub-results in the distributed case is shown in Figure 4. In this case, the overall query runtime incorporates the query execution time on the individual hosts, the round trip times for the messages sent with Balu and the final merge step. Out of 290 hosts, 284 were reachable at the time of querying and returned their results in under 30 seconds. It can be seen that over 95% of the sub-results were received and processed within 6.5 seconds.

In contrast to the centralized case, the cumulative amount of data transferred from the hosts involved in the distributed query case amounted to only 294.9 kB of compressed JSON data, which constitutes just over 1 kB transferred per host on average. Depending on query type and frequency of queries being issued, goProbe achieves several orders of magnitude lower bandwidth consumption to arrive at the same result.

To gauge the system footprint of goProbe we compared it with nProbe, another popular monitoring solution implementing NetFlow. The system footprint is important since, unlike the classical NetFlow probe which taps into routers from a dedicated system, goProbe runs on the networking infrastructure itself. nProbe and goProbe were run on a test system, while identical test traffic was generated using a traffic generator. Figure 5 visualizes the memory reserved by the probes over time. Due to the simplified flow model and expiration logic, goProbe needs to reserve far less memory. Furthermore, its memory consumption is stable over time, which simplifies provisioning. The regular spikes in memory consumption are due to aggregation and to disk write operations.

In order to evaluate the performance of the proposed database solution, a direct comparison with a different, widely used columnar database solution was performed. As benchmark FastBit [12] was chosen for its similarity and consequently its comparability to goDB. The FastBit database management system (DBMS) is a file based, columnar data store, primarily developed to support data warehousing scenarios in which bulks of scientific data have to be consolidated and evaluated. The database also consists of several files where one file is used per database attribute. Column data is stored in a binary format in an append-only fashion, without the support for compression techniques. The FastBit database was divided into daily partitions to enable a direct comparison between the database implementations.

The core criteria of the database evaluation were the runtime of a query, the memory footprint of the involved process(es), the utilization of the CPU and the hard disk I/O performance during a typical aggregation query performed on an identical set of 7.8 GB of raw data using both DMBS. The tests were performed on a machine with 4 CPU cores and a standard HDD.

The query runtime was recorded in wall-clock time, which measured the total elapsed seconds that the involved database processes needed to invoke the query, execute it and produce the output data.

The memory footprint was obtained by measuring the physical reserved memory by the database process(es). This observable served as an indicator to how extensively the tested DBMSs used the available memory.

The CPU usage was measured with regard to how much of the available processing power the DBMS effectively harnessed to execute a query. The user and system based CPU utilization of the involved database process(es) were accumulated and recorded as process CPU utilization. This observable gave an indication of the efficiency of parallel processing capabilities and the ability to process data without prolonged wait periods due to disk I/O limiting the availability of data. Notice that in order to limit interference with operationally critical applications goProbe is limited to using fewer CPUs than are available and with reduced

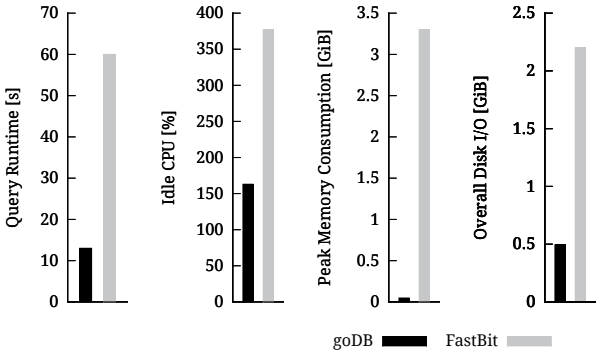

Figure 6. Comparison of key metrics between goDB and FastBit DBMS. Covered are query runtime, CPU and memory consumption, and disk I/O. The maximum possible CPU usage was 400%. As can be seen, goDB outperforms FastBit in each aspect.

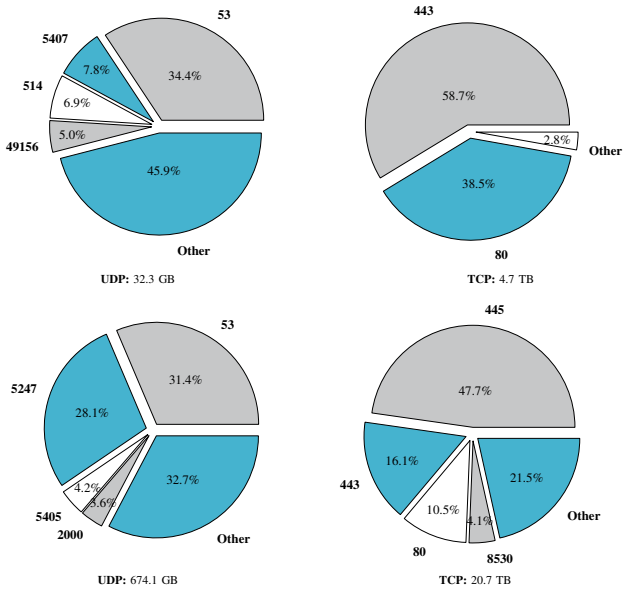

Figure 7. IP Protocol/destination port distribution for one day for the external network (top row) and internal network (bottom row). The percentages are computed with respect to the total traffic volume observed for the protocol.

scheduling priority.

Regarding I/O performance, the total number of bytes read from and written to disk within the time frame of query execution was measured in order to quantify disk read and write efficiency and additionally served as indicator of the data compression if supported by the respective database.

The results in each category for both goDB and FastBit can be found in Figure 6, corroborating the fact that goDB outperforms FastBit in each metric. Since the proposed database solution loads attributes block-wise into memory, processing can begin after the first block has been loaded, while FastBit requires the full dataset to be loaded prior to processing, which can be seen in the higher CPU utilization for goProbe. This processing model also reduces the overall memory consumption, since only blocks currently being processed have to be kept in memory. In addition, compressed data was accessed which accelerated the loading process and decreased the the number of necessary I/O operations. Lastly, goDB supports a parallel processing model making use of goroutines, hence increasing the CPU usage at the benefit of runtime significantly.

To demonstrate the capabilities of goProbe, we present the results of a real world query from the productive system. Figure 7 shows a breakdown of the ports and protocols used on the internal and external interface of one of the routers in production.

V. Related Work

The NetFlow standard is discussed in [13]. In [14], Deri proposes a software based NetFlow exporter, nProbe, which can be deployed on other devices. Inacio and Trammell [15] discuss the benefits of flow collection and exporting for analytical purposes.

Enhancements of NetFlow by means of incorporating application layer information are further covered by Danelutto et al. [16]. Bujlow provides a broad overview of the tools used for application layer detection and the challenges that come with it in [17]. Alcock [3] proposes an application layer detection based on limited payload inspection.

Abadi et al. discuss the use of columnar databases for data storage [9], showing large improvements over row-based solutions. A file based column store for bulks of scientific data is proposed by Wu et al. in [12]. Slezak and Eastwood show state-of-the art query processing and indexing strategies for a popular columnar database in [18].

Distributed sensor data collection has been extensively researched in the past. Some of these efforts concentrate on reducing the data size shipped to the central collector, e.g., [19], [20], [21], but the analysis is still performed on the collector, whereas goProbe enables a distributed query model, utilizing data locality and otherwise unused resources. Other systems, such as Star [22], Astrolabe [23] and DIPStorage [24], enable distributed query processing, but they focus on aggregating results on the return paths in order to minimize aggregate bandwidth needs. However, due to the nature of the networks managed by Open Systems AG it is not advisable to perform on path aggregations: forwarding an intermediate result to another device, over the Internet, increases the consumed bandwith on that device, and thus cost on some connections, e.g., satellite links. Additionally, lossy or slow links on the return path would cause a substantial part of the result being lost or delayed. Finally, systems like NG-MON [25] split the monitoring process into distinct parts that may be distributed, however unlike our scenario their main concern is to distribute the processing load, at the expense of high bandwidth.

In the context of data acquisition from the hosts main-

tained by Open System, Seebacher discusses a flexible messaging infrastructure in [11]. In spirit goProbe is similar to other Wide-Area Big-Data systems, e.g., [26] and [27], however its ability to be deployed on existing hardware running mission critical tasks and connected through high latency links is unique.

VI. Conclusion

In this paper we presented a distributed network monitoring architecture specifically designed to scale with the size and number of networks being monitored. The increased scalability with respect to classical monitoring solutions is a result of various improvements: Traffic is captured and stored directly on the respective devices throughout the network, making use of tailored compression techniques optimized for fast decompression. Analytic queries are distributed from a central coordinator to these devices for processing. In a first Map-Reduce step data is processed in a parallelized and selective fashion based on the parameters required for the actual query. The consolidated results are transmitted back to the central coordinator, where a second Map-Reduce step is performed to obtain the global result.

Compared to a classical NetFlow approach the amount of data archived and transferred during analysis is considerably reduced. Temporary interruption of connectivity to the network devices no longer results in the loss data due to the localized storage model. In addition to increased scalability of the data storage, a considerable query speedup was achieved by making use of otherwise unused and already present resources, resulting in true real-time queries.

The proposed system is fully operational and has been integrated into the productive environment of Open Systems AG.Both goProbe and goDB were designed to be standalone, lightweight, yet versatile components, and runs on commodity hardware. The project has been open sourced and released on GitHub[1] for public use and further development.

References

[1] B. Claise, "Cisco Systems NetFlow Services Export Version 9." RFC 3954 (Informational), Oct. 2004.

[2] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier, "Speeding Up Queries in Column Stores A Case for Compression," in *Data Warehousing and Knowledge Discovery*, 2010.

[3] S. Alcock and R. Nelson, "Libprotoident: Traffic Classification Using Lightweight Packet Inspection Categories and Subject Descriptors," tech. rep., WAND Network Research Group, University of Waikato, NZ, 2012.

[4] V. Jacobson, C. Leres, and S. McCanne, "libpcap – Initial public release," 1994.

[5] S. Mccanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," 1992.

[6] M. Larsen and F. Gont, "Recommendations for Transport-Protocol Port Randomization." RFC 6056, 2011.

[7] "Advanced encryption standard," *Federal Information Processing Standard, FIPS-197*, 2001.

[8] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry." RFC 6335, 2011.

[9] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?," in *ACM SIGMOD Management of data*, 2008.

[10] S. Almeida, V. Oliveira, A. Pina, and M. Melle-Franco, "Two High-Performance Alternatives to ZLIB Scientific-Data Compression," in *Computational Science and Its Applications ICCSA 2014*, 2014.

[11] R. Seebacher, "Message Queueing Challenges in a Globally Distributed Network," Master's thesis, ETH Zurich, 2013.

[12] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, *et al.*, "FastBit: interactively searching massive data," in *Journal of Physics*, 2009.

[13] C. Systems, "Introduction to Cisco IOS NetFlow," 2012.

[14] L. Deri, "nProbe: an Open Source NetFlow Probe for Gigabit Networks," in *TERENA Networking Conference*, 2003.

[15] C. M. Inacio and B. Trammell, "Yaf: Yet another flowmeter," in *Large Installation System Administration Conference*.

[16] M. Danelutto, L. Deri, D. De Sensi, and M. Torquati, "Deep Packet Inspection on Commodity Hardware using FastFlow," in *Parallel Computing: Accelerating Computational Science and Engineering*, 2014.

[17] T. Bujlow, *Classification and Analysis of Computer Network Traffic*. 2014. Phd Thesis.

[18] D. Ślezak and V. Eastwood, "Data warehouse technology by infobright," in *Knowledge Creation Diffusion Utilization*, pp. 841–845, 2009.

[19] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of thresholded counts," in *ACM SIGMOD Conference on Management of Data*, 2006.

[20] M. N. Garofalakis, D. Keren, and V. Samoladas, "Sketch-based geometric monitoring of distributed stream queries," *PVLDB*, vol. 6, 2013.

[21] G. Cormode and S. Muthukrishnan, "What's hot and what's not: Tracking most frequent items dynamically," *ACM Trans. Database Syst.*, 2005.

[22] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang, "Star: Self-tuning aggregation for scalable monitoring," in *Conference on Very large data bases*, 2007.

[23] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM transactions on computer systems (TOCS)*, 2003.

[24] C. Morariu, T. Kramis, and B. Stiller, "Dipstorage: Distributed storage of ip flow records," in *Local and Metropolitan Area Networks*, 2008.

[25] S.-H. Han, M.-S. Kim, H.-T. Ju, and J. W.-K. Hong, "The architecture of ng-mon: A passive network monitoring system for high-speed ip networks1," in *Management Technologies for E-Commerce and E-Business Applications*, 2002.

[26] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *VLDB Endowment*, 2010.

[27] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," 2014.

[1] https://github.com/open-ch