30-01-2023

White Paper: High-Performance Flow Monitoring Using Programmable Network Interface Cards

Grant Agreement No.:	856726
Work Package	WP6
Task Item:	Task 3
Document Type:	White Paper
Dissemination Level:	PU (Public)
Lead Partner:	UoB
Document ID:	GN4-3-22-N3R07F
Authors:	Marinos DImolianis (NTUA), David Schmitz (LRZ), Henrik Wessing (DTU), Pavle Vuletić (UoB)

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 856726 (GN4-3).

Abstract

This white paper presents the results of a development of a low-cost open-source flow monitoring system which leverages Netronome Agilio CX programmable network interface cards. The developed system is capable of non-sampled line-rate flow monitoring at 10 Gbps. In addition to giving a detailed system description, the white paper describes a set of challenges encountered while working on the system implementation, providing a realistic view of the capabilities of programming such devices.

Table of Contents

Table of Figures

Table of Tables

Executive Summary

Programmable network devices (for example, switches or network interface cards) became very popular in the last decade among network professionals, with the Programming Protocol-independent Packet Processors (P4) language as one of the most popular recent innovations. This white paper presents the results of a development of a low-cost open-source flow monitoring system which leverages Netronome Agilio CX programmable network interface cards. The system, which was designed and developed by the incubator team in the Monitoring and Management Task (Task 3) of the GN4-3 project Network Technologies and Services Development Work Package (WP6), is capable of non-sampled line-rate flow monitoring at 10 Gbps. In addition to giving a detailed system description, the white paper also describes a set of challenges encountered while working on the system implementation, relating to flow collisions, unavailable memory slots, packet digests, scalability, and race conditions, providing a realistic view of the capabilities of programming such devices and a useful reference for other professionals who are considering similar development efforts.

1 Introduction

Programmable network devices (for example, switches or network interface cards) became very popular in the last decade among network professionals. Technologies such as OpenFlow-based software-defined networks (SDNs), and then a few years later the new data plane programming paradigm (e.g. Programming Protocol-independent Packet Processors (P4) language), became the most popular topics at almost all networking events and conferences. The key reasons for an increased interest in these technologies are: the hope that network element programmability will enable the creation of new applications; that network element hardware and software will be unbundled, which could open the networking market to new vendors and lower the prices; and that a higher degree of network operations automation will be achieved.

In parallel to these changes in the network management protocols and styles, servers started to move away from the standard multicore-based server designs based on general purpose CPUs, towards the use of dedicated coprocessors based on domain-specific architectures (DSAs). DSA coprocessors are tailored for specific purposes, such as for traffic processing in network interface cards (NICs), as in the case that was explored in the Monitoring and Management Task (Task 3) of the GN4-3 project Network Technologies and Services Development Work Package (WP6) [NETDEV] and presented in this paper. NICs can now be equipped with massively parallel processing and multi-threading capabilities. Such cards can be programmed using various programming languages and styles (e.g. P4, C, assembler or a combination of those), and in this way offload a part of traffic processing from the central server processors and cores. The use of programmable NICs enables computing tasks and traffic processing to be executed closer to the data path, shortening processing times, limiting data transfers across the server, and enabling high-speed traffic processing and new applications or packet modifications without sacrificing network traffic performance.

Flow monitoring is one of the most common processes network professionals in network and security operations centres (NOCs and SOCs) use in daily operations. Because it provides per network conversation statistics, it is an ideal tool for the analysis and detection of unusual events and some types of network attacks (e.g. (D)DoS). The classical approach to flow monitoring is to use network elements (e.g. routers) as devices which gather the flow information. Such devices are either not optimised for flow capturing and can only capture flows by heavily sampling the analysed traffic (in some cases, one in 2,048 to 65,535 packets is being processed for flow monitoring [Cisco_Flow]) or need additional products in order to achieve unsampled flow monitoring [Cisco_SNA]. Packet sampling in flow monitoring means that some packets will pass through the network unnoticed, overestimating observed flows and thus limiting the usefulness of such flow monitoring. On the other side, additional products come with a non-negligible price tag.

In this paper, we describe a flow monitoring system prototype that uses a relatively low-cost programmable NIC (Netronome Agilio CX with 2 x 10 Gbps Ethernet interfaces and NFP-4000 architecture) to offload a part of the flow processing to the NIC and achieve unsampled flow monitoring on 10 Gbps interfaces. The system was designed and developed by the incubator team of WP6 Task 3. It leverages also Elasticsearch, a well-known open-source tool for flow collection, analytics and data representation [Elasticsearch]. The developed system is an example of the use of the P4 programming language to achieve on-card flow processing and statistics gathering for creating a new type of network appliance. The work was inspired by similar attempts that appeared in recent years, typically in the research literature. There were two developments in the GÉANT and National Research and Education Network (NREN) community: one from SWITCH [Gall], and the other from the University of Twente [Hendriks]. The former uses a two-stage (and two-device) flow gathering process using an in-house-developed P4-based packet broker and an in-house-developed IPFIX-compliant exporter based on the Snabb framework [IPFIX], [Snabb]. Such a system can achieve 100 Gbps unsampled flow monitoring or 25 Mpps on a 16-core server. The latter is a pure P4-based solution on a single server using the same Netronome Agilio cards as mentioned above. This system was tested using various packet and flow rates, up to 150,000 packets per second, which corresponds to several hundreds of megabits per second, depending on the packet size. This is significantly lower than the card capacity. The TurboFlow system developed in P4 [TurboFlow] showed that it is possible to build a system based on NFP-4000 that is able to process line-rate packets and capture all the flows. Our aim was to build a single-device system capable of processing line-rate packets using partially P4-based packet and flow processing. The developed system is able to process 2 million packets per second with 99.97% of flows accurately processed.

The design and development of the flow monitoring system prototype was not without challenges. P4-enabled devices have different hardware architectures and capabilities which have an impact on the final solution, and among the issues that had to be solved were race conditions in the memory accesses, memory transfers between threads in the card [Wu_Luo] as well as between the card and the server, hash collisions in flow lookup tables, selection out of available hash algorithms and others. These issues are more pronounced in stateful applications in which connection state is stored on the card and there is a need to transfer data to the server. The use of P4 in switches or in NICs ultimately requires a different skill set from the one which is typically found in the NRENs. We hope that WP6 T3's experience in the development of this flow monitoring system, which is summarised in this paper, will be a useful resource for NRENs who are either willing to try to use such a system or are considering starting the development of their own systems using programmable capabilities in network elements.

The document is organised as follows: Section 2 describes the system architecture, all the components, and design and development challenges. Section 3 outlines the testbed setup, provides the results of the experiments, and offers an evaluation of the system, while Section 4 recaps the key points and conclusions. Appendix A is an extract from user documentation for the *rtsym_read* tool for flow information transfer from the card towards the server memory, which was developed as a part of this project.

2 System Architecture

The developed system for flow monitoring follows a distributed processing architecture with the network interface card as one of the packet processing elements and the server CPU as the other. The flow monitoring system was developed on a server with a Netronome Agilio CX card, a part of the Netronome NFP-4000 device family. The server with the card was connected to two bare-metal servers (BMS7 and BMS8) via 10 Gbps links in different VLANs allowing line-rate traffic between the devices (Figure 2.1). One of the bare-metal servers was used as a source of the traffic, while the other one could be used as a sink. The flow monitoring server can be connected to a mirror port on a device whose traffic is being observed.

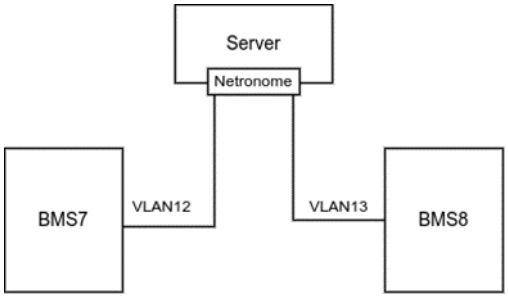

Figure 2.1: Testbed setup

2.1 Netronome NFP-4000

The NFP-4000 device family uses massively parallel processing and multi-threading techniques to achieve high packet processing performance. It incorporates a large pool of 32-bit processing elements referred to as flow processing cores (FPCs). NFP-4000 supports up to 60 FPCs and each of the FPCs supports 8 threads. Hence, the device will be able to process up to 480 packets simultaneously. There are also hardware accelerators that offload some tasks from the FPCs: ingress and egress packet modifiers, statistics engines, load balancers, lookup engines, transaction engines, bulk memory engines, and traffic managers. Finally, the whole architecture is connected via an on-card high-performance distributed switching fabric that provides high-bandwidth mesh connectivity between all the entities in the device. Further architectural details will not be presented here. For a more detailed introduction to the NFP-4000 theory of operation, readers should refer to the publicly available Netronome documentation [Netronome].

2.1.1 Programming P-4000 Cards

The NFP processing elements are distributed across the chip in the form of islands, with a high-speed bus connecting the islands for transfer of data between them. The NFP has the following main components:

1. Medium Access Control (MAC) island.
2. Ingress and Egress processing islands.
3. FPC islands with 12 FPCs.
4. PCI, Arm, Crypto and Interlaken islands with 4 FPCs.
5. Memory Units.

All the islands with the FPCs are programmable using one or more of the following programming languages: P4, C, or Microcode.

Agilio SmartNICs support the following programming models:

1. Host API-based Programming Model: using Agilio software supported APIs.
2. User Datapath Programming Model: C-based programming with configuration APIs.
3. User Datapath Programming Model: P4 and C-based programming with configuration APIs.
4. User Datapath Programming Model: Programming a C (or P4) sandbox or plug-in application into the Agilio software datapath.

More detail on NFP programming and each of the programming models is available at [Netronome_prg].

2.2 Flow Monitoring System Architecture

Network flows are separate conversations between computer systems defined with a 5-tuple (source IP address, destination IP address, source port, destination port, protocol). In order to monitor network flows, a flow monitoring system has to intercept all packets that enter a network device and process them to capture and store flow information (mainly flow start and end times and per-flow packet and byte counters). The main system components which are described in the following sections (and shown in Figure 2.2) are:

1. Capturing module: a P4-based capturing module which processes all packets received by the NIC.
2. Offloading module: a software component on the server that reads finished flows from the card.
3. Processing module: a software component which processes packets whose hashes collide in the capturing module and are punted to the virtual interface.
4. Visualisation module: ELK-based user interface.

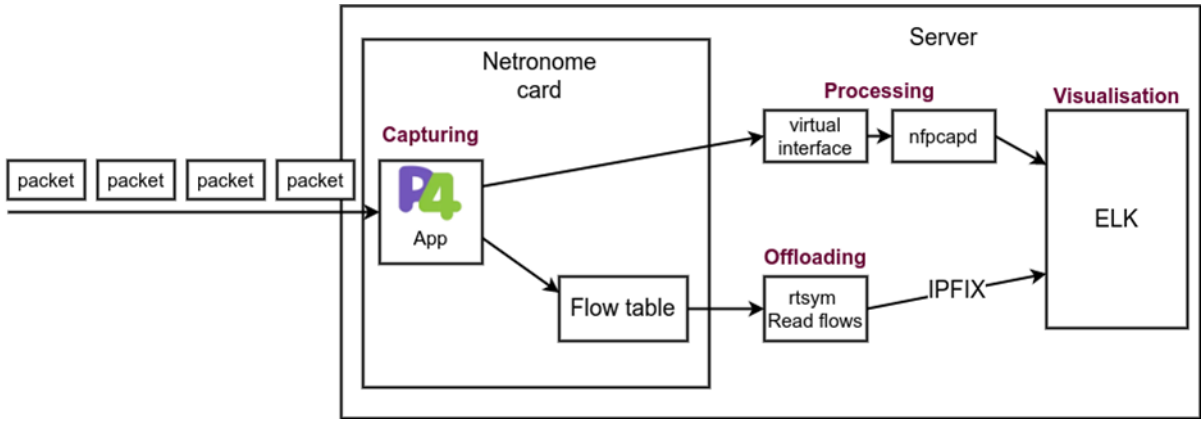

Figure 2.2: Flow monitoring system architecture

2.2.1 P4-Based Flow Capturing Model

The proposed P4-based flow monitoring mechanism comprises two different software components:

1. *P4 flow extractor* [P416_Spec] and
2. *nfdump* suite [nfdump]

that cooperatively extract flow data from the packets that are entering the NIC.

The *P4 flow extractor* program extracts and stores flow data in the available memory of the NIC. Flows that are not able to be stored in the SmartNIC's memory (due to flow collisions, described in Section 2.2.3) are redirected to a virtual interface in which *nfpcapd* is processing packets.

P4 flow extractor

The *P4 flow extractor* is a custom-developed P4 program that parses network packets (e.g. source IP, VLAN ID), and extracts and stores flow data from them. Initially, the P4 program parses and verifies the correct structure of the arrived packet headers considering only TCP and UDP packets. Subsequently, it extracts the source IP address, destination IP address, source port, destination port, protocol, henceforth referred to as the 5-tuple.

In the P4 program the 5-tuple is hashed using a CRC32 hash function, and from the calculated hash a key is generated that points to the index of the NIC memory that corresponds to that flow. P4 supports single array tables, i.e. simple key-value stores [P416_Spec] and the memory size is limited (see subsection 2.2.3). Therefore there is a high probability of a flow hash collision due to the well-known birthday problem [Birthday_Problem], even at relatively low flow rates. Flow collision is a situation in which two different flows have the same calculated hash and thus key (index).

Therefore, to utilise larger amounts of the available memory, we applied a simple linear probing approach [Linear_Probing] using a single iteration; this allows us to store two flows that have the same hash value. In case a third flow arrives that collides with the two flows already stored in memory, the packet is redirected to a virtual interface and processed by the *nfpcapd* process (Figure 2.3). This is operating in parallel with the P4 program at the CPU level of the server that hosts the P4

card. Finally, data processed either by the P4 program or the *nfpcapd* process are collected and aggregated. This is explained in the following subsection.

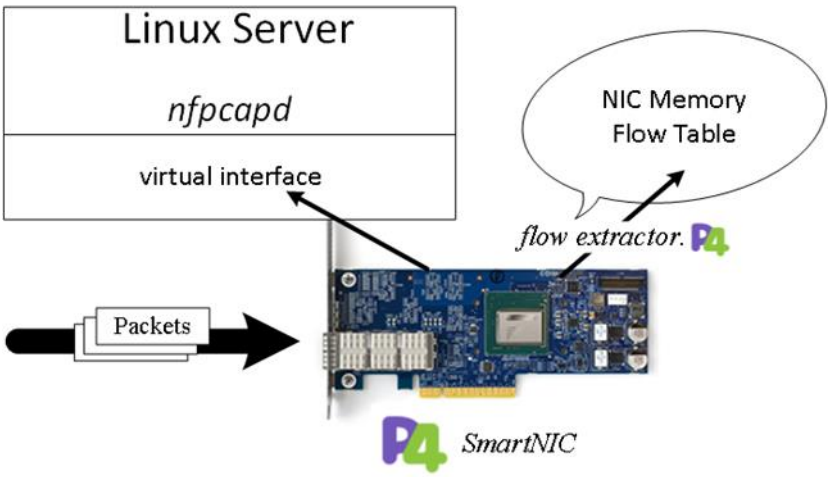

Figure 2.3: Two flow paths: through the P4 code and through the virtual interface

As described above, packets that belong to flows which experience double collision (collision after linear probing) are sent to the virtual interface. In order to be able to establish and use virtual interfaces between the P4 program and host system, the kernel module for the Netronome card `nfp.ko` is loaded with module parameters `"nfp_net_vnic=1 nfp_dev_cpp=1 nfp_pf_netdev=0"` and the P4 run-time daemon `/opt/nfp_pif/bin/pif_rte` is started by `nfp-sdk6-rte.service` with environment variables `LOAD_NETDEV=1` and `NUM_VFS=4`. Only one of the virtual interfaces is used by the P4 program for sending collided packets to the processing module realised by the *nfpcapd* software.

2.2.2 Flow Information Collection NIC to CPU

Several available approaches were tested to transfer the flow information from the card to the server. Netronome's provided application, *rtecli* (full path `/opt/netronome/p4/bin/rtecli`) turned out to be too slow (it was observed that it took 24 seconds to copy 22 MB of registered content to the server at the transfer rate of only 909 KBytes/s). Every P4 register is realised by the P4 Runtime environment as a variable allocated in the card's memory and assigned a symbol (like in C programming language), and so are all the P4 registers used for realising hash tables in the monitoring framework. Access to the list of symbols in the card and their memory content is possible using the *nfp-rtsym* application provided by Netronome (full path to the application is: `/opt/netronome/bin/nfp-rtsym`). However, the access by *nfp-rtsym* was also too slow for the needs of our application (we measured 6–13 MBytes/s).

That is why *mmap-ed* [mmap] access based on available functions was chosen instead. These functions for *mmap-ed* access are also a part of Netronome's software distribution. We have built a tool *rtsym_read* on top of *mmap-ed* which mainly allows dumping the currently finished or timeouted flows from the card's hash memory to the standard output. Unlike the *nfp-rtsym*, which

employs a device-driver-based read method, *rtsym_read* applies the mmap-related methods: `nfp_cpp_area_alloc`, `nfp_cpp_area_acquire` and `nfp_cpp_area_mapped`.

rtsym_read includes a set of different flags and modes intended to be used in various different cases, especially for distinguishing TCP flows from others:

- `dump*`, `time` and `cmp*` are all operating on an explicitly specified symbol and are meant for debugging.
- `mode3`: depending on its run-time flags (see Appendix A) `mode3` scans through all the hash tables and finds the flows terminated by the TCP FIN handshake or expired by a timeout and processes them automatically. Processing a flow in the hash tables means outputting them to the standard output in CSV (or list format) and wiping its slot memory in all hash tables on the card. The latter is the source of the race conditions described in the text below.
- `mode4` reads a hash table slot to operate on (in all hash tables simultaneously) specified by line-based standard input. It was envisioned to be used together with `"/opt/netronome/p4/bin/rtecli digest"` for terminated TCP flows, but unfortunately, this method was skipped, as `"rtecli digest ..."` was not able to handle the rate of the digests, and sometimes duplicate and missing IDs were received.

The combination of flags finally used in the controlling monitoring loop script is `mode3`:

```
rtsym_read --nfptime_hosttime_delta OFFSET_OF_TIME_DELTA --mode3.rw 30 --
tcpoldclean 90 1 MEM_DIVIDER
```

This option processes: non-TCP flows with timeout, terminated TCP flows, and non-terminated but expired TCP flows. The threshold for expired non-TCP flows is 30 seconds. The threshold for the non-terminated but expired TCP flows is set to 90 seconds. `MEM_DIVIDER` is an option needed as the whole card's memory area of registers with a large size cannot be mapped into the main memory at once. A detailed description of *rtsym_read* usage is given in Appendix A.

As described in this and the previous section, the P4 code writes the flow data into the hash table, while *rtsym_read* reads the expired and finished flows and clears their memory locations. However, there are the following race conditions in this process: between parallel invocations of the P4 code in the card but as well between the P4 code writing and *rtsym_read* reading and flushing the data. Fortunately, this is the case in a quite small percentage of the captured packets, as will be shown in Section 3. The main reason is the lack of locking facilities in the P4 code. This can only be solved by the Netronome-specific C extension of P4.

We have tried to use the `LOCK` instruction prefix on *rtsym_read* from the Intel CPU instruction level, as well as any related instructions, as e.g. `CMPXCHG` [CMPXCHG], and any higher-level programming language concepts based on the latter: e.g., `atomic_exchange` and `__atomic_compare_exchange_n` available in the *gcc* compiler [GCC]. However, all these failed, for unknown reasons, in the setup that was used. The locked memory access by the CPU instruction `CMPXCHG` (tested with `atomic_exchange()` which, when compiled by the C compiler, results basically in a `CMPXCHG` instruction) always produced a (previous) value with all bits set in the result register and no actual change at the referred memory location. Further discussion of the problem relating to `CMPXCHG` used with memory shared with a PCI device (Netronome card) can be found on Stack Overflow [Stack_Overflow].

Finally, each invocation of *rtsym_read* will pipe the *rtsym_read* to a Python script which creates IPFIX packets out of the read data and sends them to a number of hardcoded, locally running IPFIX collector ports (*nfdump*'s *nfpcapd*, and Elasticsearch Kibana at the moment).

2.2.3 Issues and Solutions

Several issues were encountered and successfully resolved while developing this flow monitoring system, relating to flow collisions, unavailable memory slots, packet digests, scalability, and race conditions. Each of them is explained in more detail below.

Flow collisions

As mentioned in subsection 2.2.1, the memory size of the available memory in the P4 card is limited and thus it is highly possible that different flows may be hashed to the same value. Due to the birthday problem [Birthday_Problem], for a table with 65,536 entries, a collision would appear with a probability higher than 50% when there are around 256 simultaneous flows (approximately the square root of the size of the table). Previously recorded traffic trace statistics [CAIDA] show that some of the recorded traces with around 4.5 Gbps throughput (something that should be expected on a 10 Gbps link) have on average around 40,000 active network flows. This means that in order to work in a collision-free regime or with a very low collision probability with such a high number of flows, a hash table of at least 2,000,000,000 entries is needed. Although cards can have this amount of RAM, having such a large table brings other problems: table searching and data transfers can be slow, while the occupancy of the table with data will remain very low. Therefore there is a tradeoff between the table size, collision probability and time to transfer the recorded data.

To resolve hash collisions and utilise the available memory as much as possible, we applied a simple linear probing approach [Linear_Probing] using a single iteration. If the hash value for a new flow is the same as an existing one, then we increase the corresponding hash value by one and use that memory slot to store flow information. This allows us to store two flows that resolve to the same hash value; in case a third collision occurs, we redirect traffic to the CPU of the server to be further processed by the *nfpcapd* tool.

Other solutions exist for dealing with the collision problem, yet they come with pitfalls. One possibility is to rehash conflicting entries with an alternative hash into a smaller hashtable. Thus complicated linked lists can be avoided, as in the applied linear probing approach. However, the processing time will vary for conflicting and non-conflicting entries. Variations in the processing delay are challenging in a P4 architecture. Additionally, complex data structures require an increased number of commands, which are typically constrained in P4 architectures.

Memory slots not available from P4

Although we were able to define various sizes for the flow tables, e.g. 262,144 cells (in P4 registers), the CRC32 hash function that was used (in that specific P4 implementation of the Netronome SmartNIC) did not provide a key greater than 65,535 (this may be considered an issue). This leads to unused memory slots of tables with size greater than 65,535. To overcome this problem, we applied the following algorithm:

1. The 5-tuple is first hashed to generate a value between [0,65535], henceforth referred to as *hash_value1*.

2. Subsequently, the *hash_value1* combined with the 5-tuple is hashed again to another value, *hash_value2* (also ranged between [0,65535]).
3. Depending on the available memory size, we apply a bitwise `AND` operation between a mask value and the *hash_value2*.
4. The result is shifted to the 16 most significant bits (MSB) and added to *hash_value1*. In a nutshell, the mask value allows us to split the employed memory table to *n* x 65,535 subtables and leverage the whole available memory.

Below we present an example for a table size of 262,144

```
mask = 0x00030000
hash_value1 = hash(5-tuple)
hash_value2 = hash(hash_value1,5-tuple)
index = (hash_value2 << 16bits) & mask) + hash_value1
```

Packet digests

Packet digests are a P4-based mechanism that allows the transfer of data available in the data plane to the control plane. Within the context of the Netronome framework, packet digests are handled by the *rtecli* tool. In our experience, the provided mechanism was not fully reliable. We faced two main issues that forced us to avoid the use of packet digests in high-speed packet rates:

- We identified missing packet digests, due either to the fact that the Netronome card could not generate them at high packet rates, or to the limited digest-capturing capabilities of the provided tool, i.e. *rtecli*.
- We received multiple times consecutive digests with mingled information (e.g. reports of non existing IP addresses).

Although packet digests were considered a promising way of asynchronously communicating between the data plane (P4) and control plane, in our case, it was not usable for timeouted flows or terminated TCP flows notification.

rtecli scalability

The *rtetool* (available via the *rtecli* tool), part of Netronome's software package, enables the transfer of P4 register contents (filled by operations performed in the data plane pipeline). As mentioned, the proposed application requires flow data and their metadata (counters) to be stored in a hash table format. These data needed to be extracted by the control plane both for storing and deletion purposes. *rtetool* could not cope with high flow rates, providing quite poor performance. This was aggravated by the fact that a single run of the *rtecli* command could only transfer the data contents of a single P4 register (e.g. packet counters). To overcome this problem, we developed a custom tool (see Section 2.2.2), able to retrieve multiple per-flow data (stored in different registers).

Race conditions

Race conditions have been experienced on two different levels (see Section 2.2.2):

A. Between concurrent memory access to P4 registers of two parallel threads, since parallel threads are handling packet data on a per-packet basis.
B. Between concurrent memory access of P4 threads (data plane) and the developed *rtsym_read* tool (control plane).

Race conditions between threads of P4 code (case A) in general are possible [Wu_Luo] and were also verified by analysing extracted packet data by consecutive runs of *rtsym_read*. Packets of the same flow, arriving at the NIC with very short inter-packet gaps, were mapped to different memory slots in P4 registers.

Race conditions between P4 threads and the *rtsym_read* tool (case B) are also potentially possible, because the use of Intel `LOCK` / `CMPXCHG` instructions, or high-level functions for memory access based on them, provided wrong values in the case of reading, and in the case of writing actually did not change values stored at the respective memory locations.

To avoid inconsistencies both between concurrent threads (in the data plane) but also with the control plane, we employed an empty bit technique. A single bit was used for every flow that indicated both to the control plane and the data plane that a specific memory slot is occupied. This design had a two-fold advantage. It enabled our mechanism to use a single bit to indicate for various P4 registers that a slot is occupied, reducing the number of race conditions (proportional to the number of P4 registers that were employed for storing flow data). Additionally, it facilitated the addition/deletion of flows in the flow table; the addition of a flow is handled by the data plane, while the deletion is handled from the control plane.

2.2.4 Flow Aggregation and Representation

Flow aggregation and representation is done in the well-known ELK stack. Its use and the setup are described in the remainder of this section.

ELK stack

The ELK stack comprises several software tools for record ingest management, storage and visualisation: Elasticsearch, Elastic Agents / Filebeat and Kibana. For the ingest management we also considered the use of ElastiFlow [ElastiFlow], which is an extension to Logstash. However, we returned to a pure Elastic Stack environment due to various factors including licensing.

The main component is the Elasticsearch document database [Elasticsearch], which utilises dynamic index templates and index patterns to make the saved documents searchable. All communication with the Elasticsearch engine is implemented via an API. Each document in the database has a timestamp field, which is used for time-series queries. The second component is the visualisation tool Kibana [Kibana], which provides extensive visualisations of the documents and document statistics from the Elasticsearch. In addition, Kibana uses a graphical control and management tool, which then communicates with Elasticsearch via the API. The communication between Kibana and Elasticsearch can be either secure, using TLS, or insecure for closed environments, although secure communication is always recommended. The third component is the Elastic Agents, which are basically probes receiving the datastream of documents from the data source. In this setup, the Elastic Agents are established using NetFlow integrations, a built-in collection of index templates and ingest pipelines for processing IPFIX and NetFlow documents. Each Elastic Agent is enrolled with

Elasticsearch via an enrolment token. Instead of enrolling each Elastic Agent with Elasticsearch individually they can be controlled and managed via a Fleet Server, which makes it quite simple to add new Elastic Agents. The setup is illustrated in Figure 2.4.

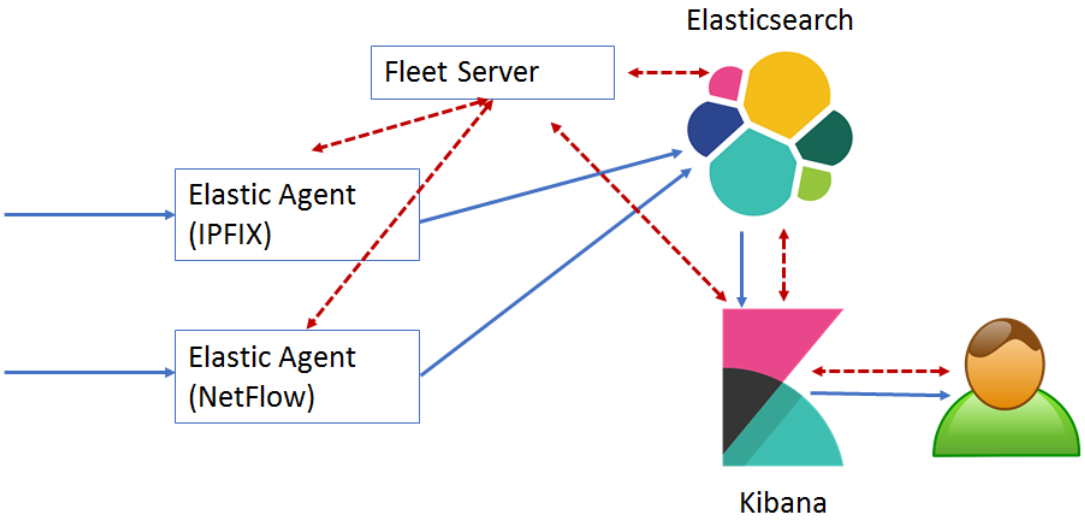

Figure 2.4: Visualisation subsystem components. The blue solid arrows indicate data plane connectivity; the red dashed arrows indicate control and management communication.

The ELK stack can be further integrated with NetFlow integrations [NetFlow_int], which are predefined policies to be installed on the Elastic Agents. The NetFlow integration modules read IPFIX and NetFlow records and export a common set of JSON formatted fields including flow information, timestamping, packet type and a myriad of other metadata about the flow. Most of the keys will not be used unless necessary, e.g., information on an MPLS record will only be available if the flow is indeed an MPLS flow.

NetFlow integrations can be added to both Elastic Agents, which can be separated by distinct namespaces, which makes it possible to separate the records from each namespace.

The NetFlow Elastic Agent integration also comes with a number of predefined visualisations, which can be altered depending on the use case.

For the flow monitoring system, basic metrics as given below can be useful:

- Number of flows and flows per time.
- Contributing source and destination endpoints.
- Port information to indicate application usage.
- Sharing between UDP and TCP flows.

In addition, it can be useful to collect information on the autonomous systems that flows are originating from / going to, and possible geolocation of sources and destinations.

3 Test Results

Our experiments illustrate the capabilities of the proposed flow-based monitoring platform that combines our custom P4 program with the *nfdump* suite. Specifically, we evaluate the following attributes of the proposed mechanism:

- Packet processing performance.
- Flow extraction accuracy (percentage of accurately observed flows).
- Number of captured and evaluated packets through the system.

Before presenting the results of that evaluation, this section outlines the testbed setup: hardware setup, traffic sources, and performance evaluation tools, process and metrics.

3.1 Testbed Setup

3.1.1 Hardware Setup

We have deployed three different servers in a local laboratory in GÉANT's point of presence (PoP) in Slough, UK, to evaluate the performance of the proposed flow monitoring mechanism. These are connected as illustrated in Figure 3.1 below:

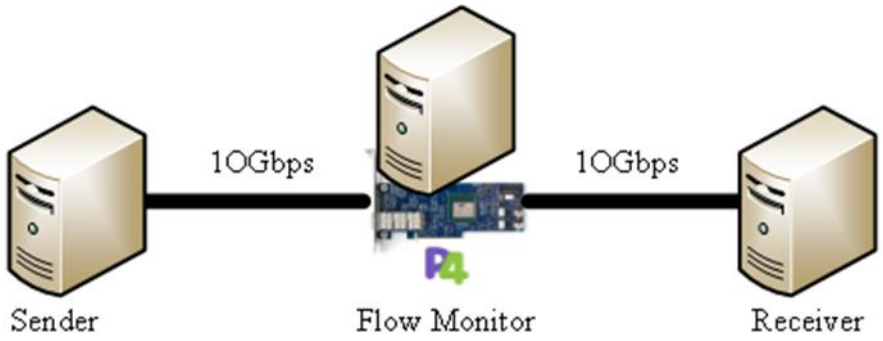

Figure 3.1: Server connections

The corresponding hardware specifications of the employed servers are presented in the following table:

Server	CPU	RAM
Sender	2 x Intel(R) Xeon(R) CPU E5-2660	40 GB
Receiver	2 x Intel(R) Xeon(R) CPU E5-2660	40 GB
Flow Monitor (P4-enabled)	2 x Intel(R) Xeon(R) CPU E5-2680 (56 threads)	64 GB

Table 3.1: Hardware specifications

3.1.2 Traffic Sources

We employed two different real traffic traces originating from CAIDA [CAIDA] containing traffic captured within one minute:

- A low rate trace with an average packet rate of ~560 kpps. Low rate: pcap duration: 2019-01-17 13:00:00 – 2019-01-17 13:00:59.99; average packet rate: 586 kpps.
- A high rate trace with an average packet rate of ~2 Mpps. High rate: pcap duration: 2018-08-16 13:45:00 – 2018-08-16 13:46:00; average packet rate: ~2 Mpps.

The traffic sources were preprocessed to include only IPv4 TCP and UDP flows for our experimental purposes. Minor changes in the code are needed to include IPv6 flows.

3.1.3 Performance Evaluation Tools, Process and Metrics

To evaluate the performance of the proposed mechanism, the following tools were used:

- *nfdump* suite v1.6.23 [nfdump].
- PF_RING [PF_RING] (for high-speed packet generation).

The aforementioned traffic sources (using PF_RING) were replayed from the Sender towards the Flow Monitor server. After the packet traces were replayed, we retrieved the observed flows from (i) the P4 program and (ii) the *nfpcapd* tool and organised them in a common format using the *nfdump* tool. Finally, we compared the observed flows with the original flows (being exported from the original packet trace using again the *nfdump* tool). We conducted each experiment multiple times using varying table sizes and traffic sources with different packet rates as described in Section 3.1.2. A packet rate of 2 Mpps corresponds to 8 Gbps of throughput or 80% of the card interface capacity if the average packet size is 500 bytes.

The metrics of interest that we wanted to evaluate are the following:

- *Jaccard index* [Jaccard_index] of flows: the cardinality of the intersection of the observed flows and the original flows divided by the cardinality of the union of the observed flows and the original flows. This metric allows us to identify how many of the original flows were observed by the Flow Monitor. The Jaccard index measurements are shown in Figure 3.2.
- *Packet ratio*: the number of packets observed by the Flow Monitor divided by the number of packets included in the original traffic source. The packet ratio measurements are given in Figure 3.3.

3.2 Results, Discussion and Conclusions from the Experiments

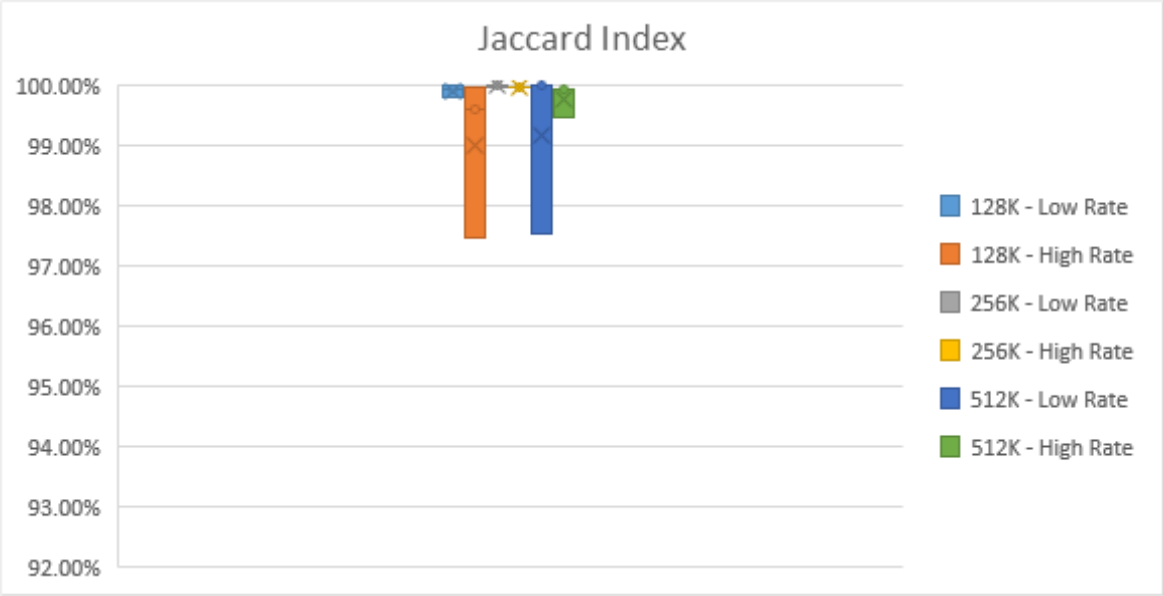

Figure 3.2: The percentage of flows captured by the flow monitoring mechanism from the original flows for various packet rates and flow table sizes

Figure 3.3: The percentage of packets captured by the flow monitoring mechanism from the original number of packets for various packet rates and flow table sizes

The proposed mechanism is able to parse the incoming packets and aggregate them in flows for various table sizes and packet rates. Specifically, it is able to correctly identify more than 99% of the received flows (on average). Moreover, it is able to do the flow monitoring with a low error rate

(approximately 2% either underestimation or overestimation). This may be related to parallel threads accessing the same memory cell, as explained in subsection 2.2.3.

We conducted the experiments using packet rates greater than 2 Mpps (by forcing traces to be replayed in higher rates, e.g. 5 Mpps). Our mechanism was not able to keep up with the number of packets/flows existing in the trace. This means either (i) that our P4-based program cannot handle that packet rate or (ii) that we reached a limit (in terms of packet rate) at traffic redirection. Note that according to our findings, the *nfdump* toolset does not miss packets during the experiments. This is verified by the *tcpdump* tool that does not report dropped packets by the kernel. This behaviour was also verified by packet counters available via the *ifconfig* tool. The PF_RING traffic generator is able to send traffic traces at high packet rates, but even when a specific rate is set, the framework sends traffic close to that rate. This means that our traffic generator introduces some randomness to the conducted experiments.

3.3 Flow Visualisation Implementation

The flow visualisation subsystem is implemented using a Docker environment in order to achieve better isolation and modularisation. This also makes it possible to inspect the dataflows between the components. An exception is the first Elastic Agent (IPFIX), which is enrolled as part of the Fleet Server deployment. The IP configuration for the components is given in Table 3.2, where it is also indicated how the Docker internal address/ports are mapped to the localhost (Netronome server).

Component	IP address	Ports, etc.
Elasticsearch	172.20.1.2/24	TCP 9200 for API. Exposed to localhost as 9200.
Kibana	172.20.1.3/24	TCP 5061 for user interface. Exposed to localhost as 5061.
Fleet Server and Elastic Agent IPFIX	172.20.1.4/24	UDP port 2055 for ingesting. Exposed to localhost as port 9995.
Elastic Agent Netflow	172.20.1.5/24	UDP port 2055 for ingesting. Exposed to localhost as port 9996.

Table 3.2: Components in Docker-based flow visualisation system

The two Elastic Agents are similar except that they tag the documents with the two namespaces “IPFIX” and “Netflow” for the two agents, respectively. By default the timestamp for the documents is the time when the document is received by Elasticsearch. Hence, when the documents containing flow data are received at the ingest pipeline between the Elastic Agents and the Elasticsearch engine, a processor is installed, which replaces the default timestamp with the `netflow.flow_start_milliseconds` timestamp. Otherwise the visualisations would have a wrong timestamp value.

3.3.1 Verification and Visualisations

In this subsection we present selected visualisations obtained after passing a pcap with 176,873 flows into the flow monitoring system. The visualisations provided are all altered versions of the visualisations provided through the NetFlow integration.

In Figure 3.4 the collected flows for the experiment are shown. The Kibana Query Language is used to filter those records with `data_stream namespace` "IPFIX" and "Netflow". The time selection tool is used to select only those records with flows starting within the selected time slot. This timestamping is based on the flow start time and not the timestamp for the records entering the visualisation subsystem. The number of flows, packet and byte counts are all in accordance with the expected values.

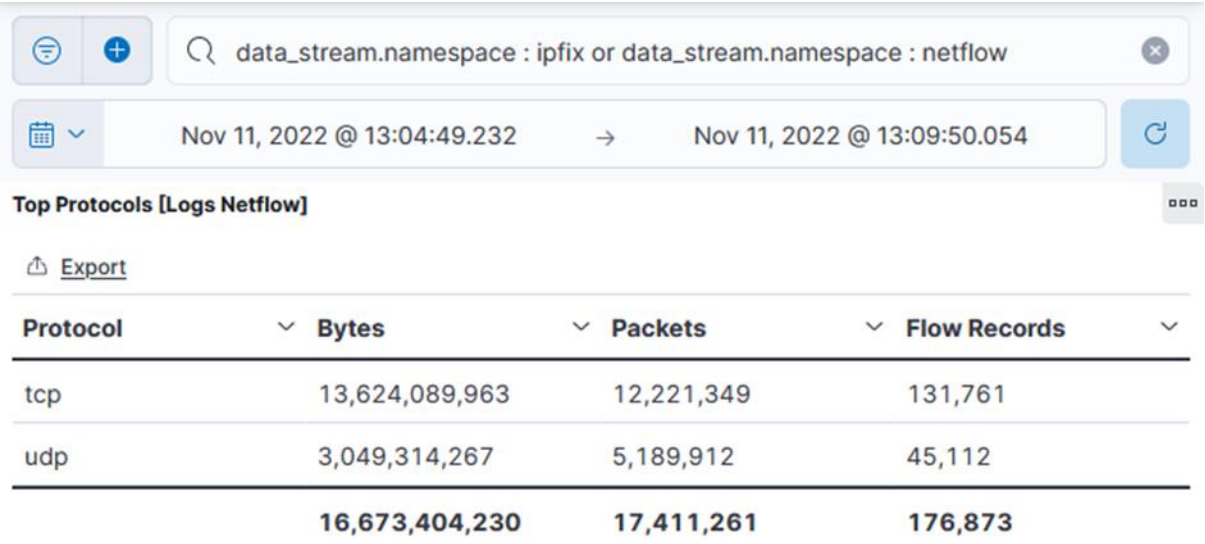

Figure 3.4: Basic description of number of flows and sharing between UDP and TCP

In Figure 3.5, an overview of the sources and destinations contributing to the flows is provided. The visualisation details the number of source and destination hosts that are part of the 176,873 flows, noting that the flows are all kinds of combinations of the sources and destinations hosts. In addition, the various ports are provided. While it can be interesting to observe the ports in order to understand the applications used, the raw number in Figure 3.5 also includes responses to clients with random ports in the areas above port 1024. If a `destination.port < 1024` filter is applied, the value of 53,703 different destination ports would become 897. Finally, Figure 3.5 also includes a graph, which indicates when the records were obtained. It is possible to zoom into this graph for easy time selection adjustment.

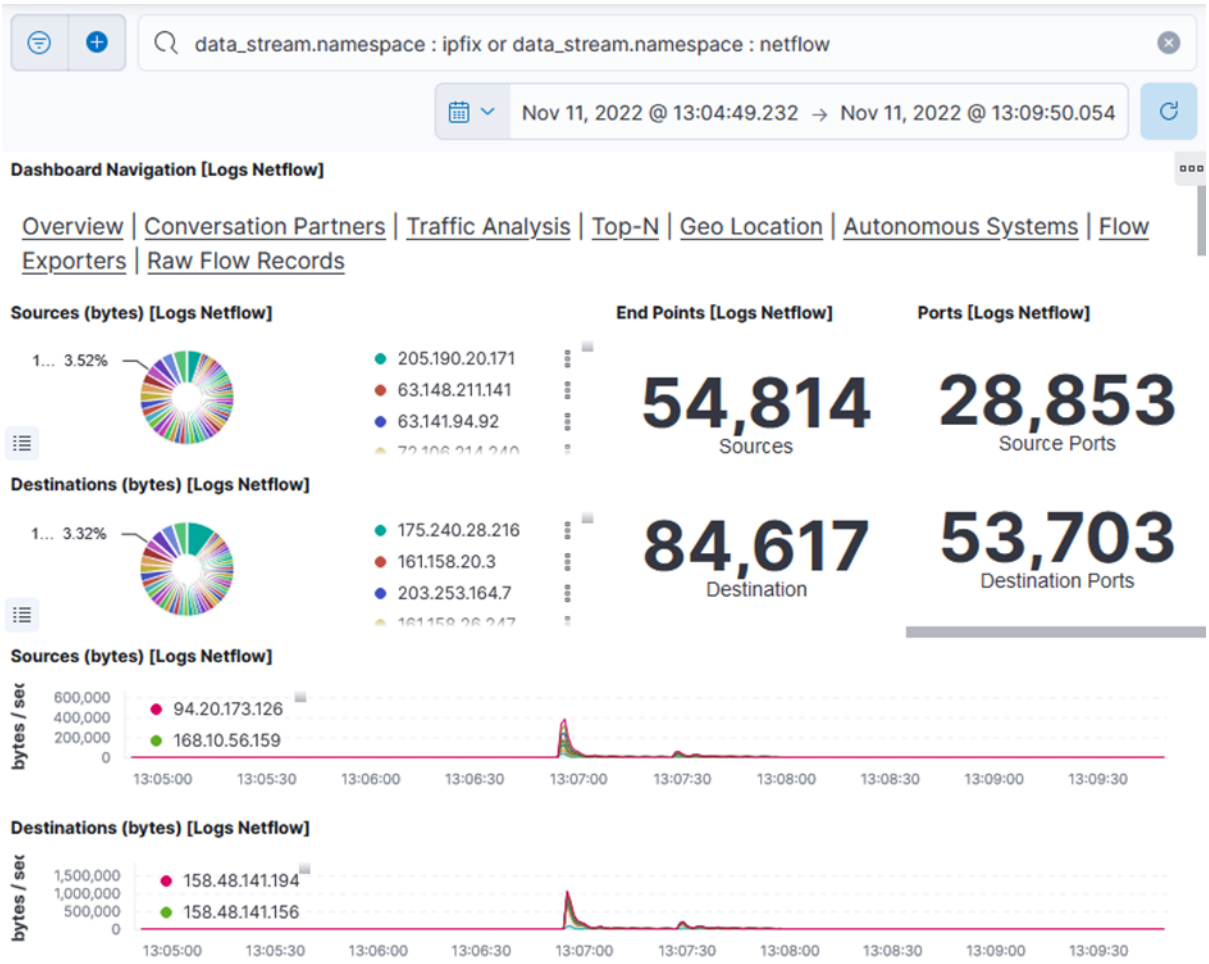

Figure 3.5: Source and destination variation overview

In addition to the independent sources and destinations described above, Figure 3.6 then shows which conversation partners (or source/destination pairs) top the list in terms of bytes, packets and flow records. This, or adapted versions, could be used to indicate activities in the network and in particular unexpected activities or even security breaches and attacks.

Top Sources [Logs Netflow]

Export

Source	Bytes	Packets	Flow Records
189.55.242.120	31,525,991	125,507	394
205.190.20.171	255,641,881	176,306	293
213.5.4.231	185,905,861	130,068	224
52.223.37.89	84,221,724	62,678	160
115.188.205.157	33,157,402	29,517	147
154.130.175.60	9,504,866	8,990	137
	12,317,738,879	**9,103,208**	**4,047**

1 of 50

Top Destinations [Logs Netflow]

Export

Destination	Bytes	Packets	Flow Records
146.55.105.106	59,506,283	90,111	695
146.55.105.92	70,911,054	86,778	682
146.55.105.97	83,590,688	96,409	661
146.55.105.105	92,887,179	100,790	660
146.55.105.107	52,388,877	72,850	653
146.55.105.98	54,828,179	82,648	649
	11,858,249,099	**9,294,663**	**16,663**

1 of 50

Figure 3.6: Top conversation partners in terms of flow records, packets and bytes

The security aspect of monitoring can be further addressed by adding a location overview to the sources and destinations contributing to the flows. Hence, in Figure 3.7, a heatmap of the contributing sources' and destinations' estimated locations is given. Elatiscsearch looks up the geo location and stores this information with the flow record document.

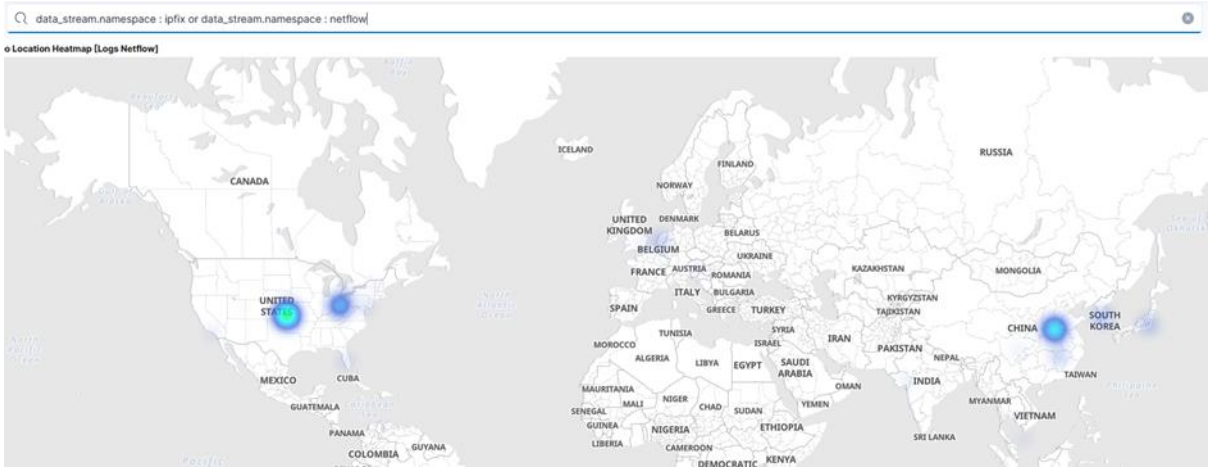

Figure 3.7: Geo location heatmap of sources and/or destinations

The screenshots and visualisations presented in this section serve to show the potential of the visualisation subsystem of the flow monitoring platform. While most of the visualisations are provided from the Elastic Agent NetFlow integration, it is easy to adjust and tailor the visualisations for the specific need of the organisation obtaining the flow monitoring.

In this implementation the Elasticsearch, Kibana, Fleet and Elastic Agents are all deployed on the same physical server isolated in Docker containers in addition to the P4 subsystems. Thus the performance would probably be greatly improved using dedicated servers for the different subcomponents. Such performance evaluation has not been fully validated.

4 Conclusions

This document has described a system for monitoring network flows using P4-enabled Netronome Agilio CX SmartNICs. The developed system is capable of monitoring all network flows on a 10 Gbps link with a minimal error rate. This proves that the SmartNICs used are capable of offloading a significant amount of work from the central server processor and storing flow information in an efficient way.

In addition to presenting a system that can be used in real networks, this document has given an insight into a set of issues and challenges that the team encountered during the development process. This can serve as a useful reference for those who aim to develop similar systems, or use similar hardware, especially if the aim is to build stateful network applications which store significant amounts of traffic information on the NICs. In these cases there are still issues which cannot be solved easily using pure P4 and the solutions are highly hardware dependent.

Appendix A rtsym_read: A Tool for Reading Netronome Card Memory Locations

This section presents an extract from the user manual for the *rtsym_read* tool created to read the memory locations and flow information from Netronome memory locations.

```
usage: /home/lab/rtsym_read/rtsym_read.bin  ...

     ....  [--debug] [--debug_args] [--debug_time] [--debug_map] ...

     ....  [--nfptime_hosttime_delta TIME_DELTA] ...

     ...   ( (--dump | --dump_bin | --dump_bin2 | --time | --cmp | --
cmp_detailed) [SYMBOL_NAME] ) | ...

     ... | ( (--mode3 | --mode3.rw | --mode3.rw.wipeall) [ TIME_THRES [
OUTFMT [ MEM_DIVIDER [ MEM_DIVIDER_IDX ] ] ] ] ...

           ... [ --dump_all | --dump_all_and_empty] ...

           ... [--notcp | --tcp | --tcpclean | --tcpcleanonly |
(tcpoldclean TCP_TIME_THRES) ] ) | ...

     ... | ( (--mode4 | --mode4.rw) [ TIME_THRES [ OUTFMT [ MEM_DIVIDER [
MEM_DIVIDER_IDX ] ] ] ] ...

           ... [ --dump_all | --dump_all_and_empty] )
```

default for `SYMBOL_NAME` is "_pif_register_time_start")
(`mode[34]*` work on multiple symbols instead)
`MEM_DIVIDER` (preset from env variable `RTSYMREAD_MEM_DIVIDER`): only map i-th slice of memory area at once into main memory

`MEM_DIVIDER_IDX`: only work on the i-ith slice of main memory (-1=all) – for testing only

rtsym modes:

- `dump` mode: dump memory area of symbol in hex
- `dump_bin` mode: dump memory area of symbol in binary
- `dump_bin2` mode: as `dump_bin` mode, but use single write call (very inefficient, for testing only)
- `time`: time measurement for mmap-based card mem test access
- `cmp` mode: performance compare readdev and mmap-based card mem test access
- `cmp_detailed` mode: as `cmp mode`, print details of each cycle
- `mode3`: output all expired (and terminated TCP) slots from

`mode3.rw`: as `mode3`, but wipe dumped slot entries
`mode3.rw.wipeall`: as `mode3`, but wipe all entries
`TIME_THRES`: threshold for expired flows (default: 10 seconds)
`OUTFMT`: 0=list format | 1=csv

separate handling of TCP flows:
`notcp`: do not handle TCP flows at all
`tcp`: handle TCP as other flows (ignore TCP flow termination and use `TIME_THRES` for them)
`tcpoldclean TCP_TIME_THRES`: honor TCP flow termination and expire TCP flows after `TCP_TIME_THRES` (default: 30)
`tcpclean`: only handle terminated TCP flows (no expiration)
`tcpcleanonly`: as `tcpclean`, but do NOT handle non TCP flows
`dump_all`: output all entries, not only terminated/expired
`dump_all_and_empty`: `dump_all`, but also wipe all entries

`mode4`: similar to `mode3`, but output slots of all slot indices read from STDIN
`mode4.rw`: as `mode4`, but wipe dumped slot entries

References

[Birthday_Problem] https://en.wikipedia.org/wiki/Birthday_problem

[CAIDA] https://www.caida.org/catalog/datasets/trace_stats/

[Cisco_Flow] https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r6-1/netflow/configuration/guide/b-netflow-cg-asr9k-61x/b-netflow-cg-asr9k-61x_chapter_010.html

[Cisco_SNA] https://www.cisco.com/c/en/us/products/collateral/security/stealthwatch/secure-network-analytics-aag.html

[CMPXCHG] https://www.felixcloutier.com/x86/cmpxchg

[Elasticsearch] https://www.elastic.co/elasticsearch/

[ElastiFlow] https://www.elastiflow.com/

[Gall] https://wiki.geant.org/download/attachments/148094173/A-Gall_netflow.pdf?version=2&modificationDate=1604927806000&api=v2

[GCC] https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

[Hendriks] https://wiki.surfnet.nl/download/attachments/23799958/ron-meeting-2019-ut-p4-flows.pdf?version=1&modificationDate=1578671925352&api=v2

[IPFIX] https://datatracker.ietf.org/group/ipfix/documents/

[Jaccard_index] https://en.wikipedia.org/wiki/Jaccard_index

[Kibana] https://www.elastic.co/kibana/

[Linear_Probing] https://en.wikipedia.org/wiki/Linear_probing

[mmap] https://en.wikipedia.org/wiki/Mmap

[NETDEV] https://wiki.geant.org/display/NETDEV

[NetFlow_int] https://docs.elastic.co/en/integrations/netflow

[Netronome] https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf

[Netronome_prg] https://www.netronome.com/media/redactor_files/WP_NFP_Programming_Model.pdf

[nfdump] https://github.com/phaag/nfdump

[P416_Spec] https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[PF_RING] https://www.ntop.org/solutions/wire-speed-traffic-generation/

[Snabb] https://github.com/snabbco/snabb

[Stack_Overflow] https://stackoverflow.com/questions/70674160/use-atomic-operations-on-the-pcie-host-device-shared-memory

[TurboFlow] https://dl.acm.org/doi/pdf/10.1145/3190508.3190558

[Wu_Luo] https://cdn.open-nfp.org/media/documents/Network_Measurement_with_P4_and_C_on_Netronome_NFP_UML_M9VbMwB.pdf

Glossary

API	Application Programming Interface
CAIDA	Centre for Applied Internet Data Analysis
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checksum
CSV	Comma-Separated Values
(D)DoS	(Distributed) Denial of Service
DSA	Domain-Specific Architecture
ELK	Elasticsearch, Logstash, and Kibana
FPC	Flow Processing Core
ID	Identifier
IP	Internet Protocol
IPFIX	Internet Protocol Flow Information Export
JSON	JavaScript Object Notation
Kpps	kilo packets per second
LAN	Local Area Network
MAC	Medium Access Control
MPLS	Multiprotocol Label Switching
Mpps	Mega packets per second
MSB	Most Significant Bits
NFP	Network Flow Processor
NIC	Network Interface Card
NREN	National Research and Education Network
NOC	Network Operations Centre
P4	Programming Protocol-independent Packet Processors – a domain-specific language for network devices, specifying how data plane devices (switches, NICs, routers, filters, etc.) process packets
pcap	packet capture
PoP	Point of Presence
RAM	Random Access Memory
SDN	Software-Defined Networks
SOC	Security Operations Centre
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VLAN	Virtual LAN
WP	Work Package
WP6	GN4-3 Work Package 6 Network Technologies and Services Development
WP6 T3	WP6 Task 3 Monitoring and Management