Using P4 and RDMA to collect telemetry data

Beltman, R.; Knossen, S.; Hill, J.; Grosso, P.

DOI
10.1109/INDIS51933.2020.00006

Publication date
2020

Document Version
Final published version

Published in
Proceedings of INDIS 2020

Citation for published version (APA):
Beltman, R., Knossen, S., Hill, J., & Grosso, P. (2020). Using P4 and RDMA to collect telemetry data. In *Proceedings of INDIS 2020: Innovating the Network for Data-Intensive Science : held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-9). IEEE Computer Society. https://doi.org/10.1109/INDIS51933.2020.00006

Using P4 and RDMA to collect telemetry data

1st Rutger Beltman
Security and Network Engineering master
University of Amsterdam
Amsterdam, The Netherlands
rutger.beltman@os3.nl

2nd Silke Knossen
Security and Network Engineering master
University of Amsterdam
Amsterdam, The Netherlands
silke.knossen@os3.nl

3rd Joseph Hill
Multiscale Networked Systems group
University of Amsterdam
Amsterdam, The Netherlands
j.d.hill@uva.nl

4thPaola Grosso
Multiscale Networked Systems group
University of Amsterdam
Amsterdam, The Netherlands
p.grosso@uva.nl

***Abstract*—Telemetry data can be carried within network packets as they transit through the network. This in-band telemetry requires in turn to have efficient means for the collection and processing of these data. We show how the P4 language can be used to extract telemetry data from incoming packets, as it allows for efficient controlling of the data plane of network devices. Furthermore, we exploit remote direct memory access (RDMA) as this allows for direct placement of data into the memory of an external machine. We implemented the RDMA over Converged Ethernet (RoCE) protocol in a switch using a P4 program. The program keeps the state of the variables required to perform a write-only operation from the switch to a collector machine. The telemetry data is stored on persistent storage in the machine using memory mapping. Experiments with our implementation show that the telemetry data is saved to the designated addresses. We were able to achieve a rate of around 20 million packets per second without any packet loss.**

***Index Terms*—P4, RDMA, telemetry**

I. Introduction

Monitoring the status of the network in real-time, collecting information on the flows traversing the network devices is a cornerstone of the operation of any network, either commercial or for research. The real-time and offline analysis that can be made on such network data is crucial ingredient in guaranteeing performance and security of the network operations.

The latest development in the areas of programmable network devices open up new possibilities for a new approach to telemetry and monitoring. In particular, programmable network devices allows for the inclusion of in-band network telemetry directly in packets. This in turns allows for gathering significantly more data, which provides more details about the current state of the network. Still, in light of this additional amount of data, a crucial problem emerges: how to collect and process the data. We need a machine capable of high-resolution data processing that can also collect this information in real-time. We are aware that there are many efficient packet collectors for Linux that process large amounts of network traffic in real-time [1]. Our research wants to provide an efficient alternative technique for such collectors.

Our work relies on two technologies: P4 and RDMA. The Programming Protocol-independent Packet Processors (P4) language can extract telemetry data from incoming packets [2], [3], as it allows for efficient controlling of the data plane of network devices [4]. We use the P4 features to develop an efficient method to extract telemetry data. Remote Direct Memory Access (RDMA), as the name implies, makes it possible to access memory remotely. It allows for direct placement of data into the memory of an external machine, removing the need for CPU involvement. RDMA has the potential to store the P4 extracted data with high throughput. Our work wants to determine if and how a RDMA combined with P4 is a viable approach for collecting network telemetry data. In the following we will address a number of core aspects.

- We will show how telemetry data can be encapsulated in an RDMA message
- We will show how an RDMA session can be maintained on a P4 switch and how telemetry data can be placed into persistent storage using RDMA
- Finally we will, using an experimental setup built for this purpose, investigate the packet rate that can be achieved using RDMA

II. Background

A. Network Telemetry in P4

When discussing network telemetry we mean monitoring information about network traffic. Many methods exist to gather information about network health: it is possible to maintain counters in a network device and actively probe a device to get information about the buffer occupancy [5]; or, we can rely on in-band network telemetry by including monitoring data directly in network traffic packets [3].

In-band telemetry has multiple advantages over more traditional methods of monitoring network traffic from out of band. First of all, it enables the collection of internal state data of any network device, such as identification (used in path tracking [6]), queue occupancy, and processing latency. Secondly, where often the collected data in traditional methods is obtained using sampling and results in much overhead, in-band network telemetry allows for collecting data from every

978-1-6654-0454-9/20/$31.00 ©2020 IEEE
DOI 10.1109/INDIS51933.2020.00006

packet with low overhead [5]. A protocol that implements this mechanism by including additional headers in each packet is called In-band Network Telemetry (INT) [7]. Telemetry data is processed through a pipeline in order to analyze it and perform actions accordingly.

Figure 1 shows an example of a telemetry pipeline.

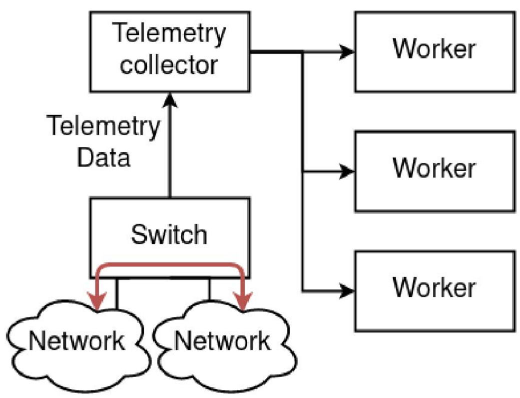

Fig. 1. Typical telemetry workflow: a network device (switch) provides telemetry data to a telemetry collector. The telemetry collector passes this information to worker nodes for further processing

As seen in the figure, the switch retrieves telemetry data from every packet that traverses through it and sends it to a collector. Depending on the pipeline implementation, a collector may store and forward the data to one or multiple workers. A worker will perform an analysis of the data. However, the vast amount of telemetry data coming from the switch may be too much for one worker to process in real-time. Batching the telemetry data into smaller jobs and delegating the task of analyzing the data to a worker, divides this workload.

Our research focuses on a specific part of the telemetry pipeline shown in Figure 1: we will in fact focus on the left part of the workflow, i.e. the part in which packets arrive at the switch where telemetry data is extracted and forwarded to the collector. The INT protocol is one of many use cases that could benefit from an efficient implementation for this pipeline. However, we did not make protocol specific optimizations, neither we implemented any specific telemetry protocol, as our goal was to provide a general solution for telemetry collection.

The emergence of P4 capable devices to be used in the telemetry pipeline opens up new possibilities. P4 offers more flexibility than what is currently available on network devices because there is no predetermined definition of a packet format. This makes it protocol independent and allows for the design of new protocols. This flexibility motivated us to test the potential of P4 for implementing RDMA from a P4 enabled device that typically does not create RDMA packets. A second advantage that comes with using P4 is the flexible allocation of device memory, which makes it possible to assign memory locations originally intended for forwarding tables as general-purpose registers.

However, P4 has a number of limitations, since it is a domain-specific language and not Turing-complete [8]. For instance, it can only keep limited state between packets, with the use of counters and registers. Counters keep an incremental state between packets, while registers can keep any state, and the CPU can also interact with them. Furthermore, it can not allocate memory arbitrarily, as a C program can. Moreover, there is no support for loops, typical data structures such as dictionaries, or packet trailers [4]. Programs written in general-purpose programming languages are not trivial to implement in P4. For all these reason, it is challenging to implement the RDMA protocol in a P4 switch but our work shows that it is possible to work around the limitations of P4 to successfully implement RDMA.

B. RDMA

In high-performance computing, there is a strong demand for responsive and high throughput data transfers. A Direct Memory Access (DMA) engine is a component that the CPU can use to facilitate a transfer between two buffers. When data from a buffer needs to be transferred, the CPU can set up the DMA engine with the source and destination address. After setup, the transfer can complete without further involvement of the CPU. With the transfer completing in the background, the CPU can continue processing other tasks [9].

To allow DMA from a remote agent, RDMA was created [10]. With RDMA, the DMA engine is placed in a NIC. RDMA allows a remote agent to initiate a memory transfer to and from a target machine without involvement of its CPU. For example, an RDMA write operation sends data to a buffer in a target machine; on arrival of the RDMA write operation, the NIC processes the packet to determine where the payload needs to be placed in memory; finally, after boundary checking, the NIC initiates the write operation and writes the data directly to memory.

Exposing memory to the network creates security concerns. To alleviate this, RDMA has strict boundary enforcement. During the setup of RDMA, a pointer to the buffer and its size are passed as arguments. This informs the NIC that RDMA will only be allowed in this memory region. The NIC will be responsible for filtering out a memory read or write operation that attempts to access memory outside the defined region. Even with the built-in security, a network architect should make sure her network is properly segmented to prevent an attacker from interfering with the RDMA data stream. [10] and [11] provide an extensive overview of the security considerations related to the use of RDMA.

In our research, we wanted to use this protocol for use in Ethernet networks, hence we focused specifically onto RDMA over Converged Ethernet (RoCE). RoCE is a protocol that allows transferring RDMA packets over Ethernet networks. There are two versions of this protocol. RoCEv1 provides network traffic to be switched over a layer 2 network, while RoCEv2 enables routing RDMA traffic over a layer 3 network, using IPv4 or IPv6, and it enables inter-subnet RDMA communications. In the work presented here we used

RoCEv1 to simplify our implementation and no routing was necessary in our case. The protocol was initially born out of Infiniband, a switched fabric interconnected architecture for server and storage connectivity [12]. RoCE uses Infiniband headers for their packet format. The headers required for a RoCEv1 RDMA write-only operation are found in Figure 2.

Fig. 2. Example of an RDMA write request.

The first header is for Ethernet, allowing the traffic to travel over regular Ethernet networks. The Global Route Header (GRH) is the Infiniband version of an IPv6 header. It contains the same header fields as an IPv6 header but is used on Infiniband networks instead. The Base Transport Header (BTH) contains information regarding the action the NIC needs to take on the payload. Because this example is an RDMA write-only request, an RDMA Extended Transport Header (RETH) needs to be included. This header contains information about where in memory the payload needs to be written. The invariant CRC behind the payload is a checksum, similar, but not equivalent, to the Ethernet checksum.

III. Related work

There is previous research that has focused on the use of RoCE in high-performance networks. [13] compared the performance of TCP, UDP, UDT, and RoCE. They used the RDMA write operation due to its low overhead and showed that RoCE provides consistently good performance, with a CPU usage that is is much less than for the other evaluated protocols. They tested two scenarios: using a dedicated path for RoCE traffic, and simultaneously sending RoCE and TCP flows. [13] concluded that the ability of RoCE to provide low latency and system overhead makes it a compelling technology for high-resolution data transfers.

The work in [14] provides us with a rich wealth of information we can use in our work. Here researchers examined the feasibility of implementing RoCE in a P4 capable switch. Three different use cases show how the switch performs RDMA read, write, and atomic fetch-and-add operations.

The first use case investigated if it is possible to use RDMA for extending the buffer of a switch. The switch uses buffers to deal with bursts of outgoing traffic on one of the ports. Unfortunately, if the buffer is full, packets are dropped. However, RDMA can temporarily store bursts of traffic in memory of a server until the buffers get cleared up. When the buffers have free space left, the switch can make an RDMA read request to retrieve the stored packets from memory.

The second use case used RDMA for lookup tables. By default, lookup tables in switches are in the order of tens of megabytes. The switch can use RDMA to access external memory to increase the size of lookup tables. First, a packet is stored in the memory address after the action for the packet. This makes the lookup operation stateless for the switch. The buffers of the switch would fill up if the packet had to be stored. Subsequently, an RDMA read request can pull the packet from memory together with the action.

The third use case investigates RDMA for telemetry data. The experiments show how the atomic fetch-and-add operation can increment counters on remote telemetry servers. While the research talks about the possibility of using RDMA write-only operation for storing header information of telemetry data, they only tested the fetch-and-add operation for counters. The implementation in this research effectively borrows memory from host systems. This means the host does not know what happens in the loaned out memory regions, nor does it perform any action on the stored data.

In our research, we will test the feasibility of using the RDMA write-only operation for storing telemetry data into external memory. We will also examine if it is possible to use RDMA to access persistent external memory. Eventually, the server will further process this data by forwarding it to one or multiple workers. For this reason, the server and switch need to have some communication about the data that is sent to the server.

IV. Methodology

Before we describe the methodology we adopted to implement the RoCEv1 protocol in the P4 language we need to clarify that there are a number of aspects we consider out-of-scope and are left to future adopters of our work: data analysis, telemetry protocols, and signaling. We focus exclusively on the collection of telemetry data rather than its analysis. We do not implement a specific protocol for telemetry data gathering, but we use telemetry data collected from the headers in the packet that traverses the switch. Finally we do not implement signaling to the the CPU the presence of data that can be analyzed. We recognize that an efficient approach would result in the CPU only reading the addresses that are ready to read from and prevent data from being overwritten before it is read. In Sec. VII we will propose a technique that could perform signaling.

A. Experimental setup

For our experimental setup, we used two servers and one P4 programmable switch. The switch we used is an Edgecore Wedge 100BF-32X. The switch contains 32x 100 GbE QSFP28 ports and has the Tofino 3.3T ASIC for P4 capability. One of the servers is the Dell PowerEdge R540, which has 128 GB of ram, 2x Intel Xeon Silver 4114, Mellanox ConnectX-3, 2x Netronome Agilio CX 2x25 GbE SmartNIC (P4 programmability turned off), and runs the Ubuntu 18.04 operating system. The other server is the Supermicro, which has 196 GB RAM, 2x Intel Xeon Gold 5122, Mellanox ConnectX-5, 8x 1TB NVMe, and runs the CentOS 7.7 operating system. The driver version used for the Mellanox NICs is the MLNX OFED version 5.0-2.1.8.0. Figure 3 shows the physical topology used for experimentation. In this topology, we use the Supermicro server as the collector of telemetry data. This is because the Mellanox ConnectX-5 in this server supports higher through-

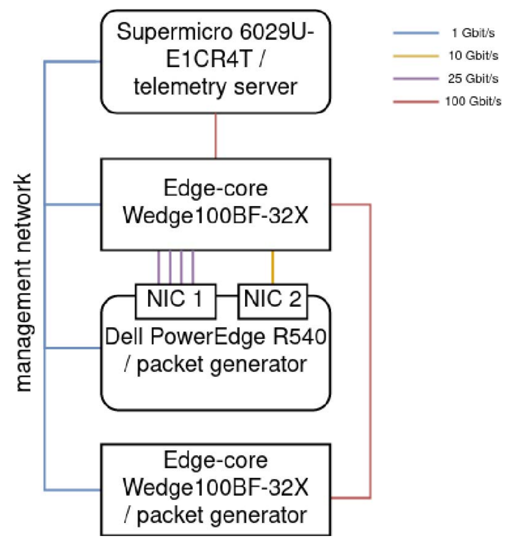

Fig. 3. Physical lab setup used for experimentation. NIC 1 are the Netronome Agilio CX 2x25 GbE SmartNICs, NIC 2 is the Mellanox ConnectX-3.

put than the Mellanox ConnectX-3. The Supermicro server also has 8TB of NVMe storage for storing telemetry data.

The Mellanox ConnectX-3 on the Dell server is for experimenting and examining RoCEv1 traffic. The four connections to the Netronome CX NICs will generate TCP traffic that will go through the P4 switch. The management plane is used to access the equipment and injecting variables for RDMA in the P4 switch. For the performance analysis, we run a packet generator on a second Edge-core Wedge100BF-32X, which is directly connected to the switch. We use this switch, because it can send packets at 100 Gbit/s.

B. Server implementation

For the proof of concept application, we use the RDMA write-only operation. There are multiple reasons for this. First, because it is an RDMA operation, no interaction is required with the server after establishing a session. Secondly, the only state information that needs to be kept is the variables required to construct the packet. Another choice we could have made is the send operation from the Infiniband verbs API. However, this operation requires CPU interaction to locate where the payload should be stored in memory. The RDMA write-only operation already has this information included, and therefore can directly interact with memory. This allows for less CPU involvement, which could result in a faster performing operation. In a regular RDMA application, two hosts that both have RoCEv1 support set up an RDMA session by sharing their queue pair numbers. A queue pair is a number that identifies the packet and puts it in the appropriate queue. If one of the hosts wants to access another host's buffer, it requires a pointer to the virtual memory address of the buffer together with a remote key. The purpose of the remote key is to authorize the RDMA to the virtual memory address. In our implementation, the switch requires a queue pair, virtual address, and remote key from the host to send RDMA write-only operations to remote memory. The Infiniband drivers are used to set up RDMA on the Supermicro server. These drives return the remote key and queue pair that is needed to configure the RDMA session. Between the switch and host, there is a TCP connection active, responsible for transporting the variables from the host to the switch. The TCP connection is made over the management network (Figure3), because the variables are configured from the switch's operating system. With these variables, the switch can now craft RoCE packets. In a typical RoCE application, two devices send information about how to reach their available memory regions to each other. However, in our application, the switch sends information to the other host to remotely store telemetry data. Since the server will not send write or read requests to the switch, the only information the host requires is the queue pair number. The server requires the queue pair number to transit to a state where it is ready to receive packets. We will use an unrelated queue pair number from a previous experiment to allow the host to go to the correct state. The queue pair for accessing the switch is statically configured on the host since it will not be used. In this research, the data will not be forwarded after arriving on the telemetry collector. For this reason, we will write the data directly into a file using the mmap function. We implemented this by first opening a file that is filled with zeros. With its file descriptor, we call the mmap function to map it to virtual memory. This results in the whole file being registered as a buffer. We use function calls from the Infiniband libraries to make this buffer accessible from the NIC. As a result, the NIC can perform the RDMA operations on the memory-mapped file. This process removes the overhead created by manually saving the data to a file. This server application is written in C, as the libraries are available for this programming language, and it allows for extensive control of memory.

C. Switch implementation

In our workflow, network traffic that arrives at the switch will be forwarded to both its desired destination and a switch port connected to the collector server. This is done by copying the packet in the ingress pipeline of P4, and setting its egress port to the collector port. According to the packet egress port, telemetry data will be encapsulated in RoCEv1 headers used for RDMA write-only operations. As the P4 switch has no native support for RDMA, we implement the ROCEv1 protocol ourselves. To do so, we defined the headers (as shown in Figure 2), including their fields in a P4 program. According to the definition of the header fields from the Infiniband specification [15] in combination with experimentation, we assigned them the correct values. We will explain the important and relevant header fields for our implementation in this section. The GRH contains the same header fields as used in IPv6 for compatibility. The next header field is 27, which identifies a BTH from Infiniband. The migrate request flag in the BTH is set as we migrate data from one device to another. The packet sequence number in the BTH is increased by one each time a RoCE packet is sent. To include a sequence

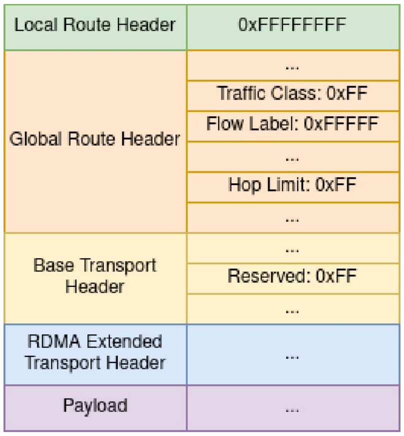

Fig. 4. Headers and masked fields used in the Invariant CRC calculation.

number in our implementation, we created a counter using a register. The counter is increased each time a packet is sent to the telemetry server. The destination queue pair, remote key, and virtual address are obtained from the server, as discussed in Section IV-B. These parameters are sent to the switch's control plane over a TCP connection. In our implementation, we created a forwarding table entry when the three parameters are received. The forwarding table has the egress port as key that is matched against the egress port of the packets in the P4 egress pipeline. In the case of a match, the parameters are assigned to the RoCE packet. As the virtual address must be changed after one packet is sent, we created a counter to calculate the packet's virtual address offset. The counter increments each time a packet is sent with the number of bytes that are written to the storage on the server.

As defined in the Infiniband specification, the invariant CRC is a 32-bit checksum using the same polynomial as used in Ethernet. The CRC is calculated over the fields shown in Figure4. There is one Infiniband header, the Local Route Header, that is not part of the RoCE protocol. However, this field is included when calculating the CRC, but all fields are masked to one. Some other fields are also masked to one, as shown in the figure. The switch supports the calculation of CRCs with a custom created polynomial in P4.

```
CRCPolynomial<bit<32>>(
    coeff = 0x04C11DB7,
        reversed = true,
        msb = false,
        extended = false,
        init = 0xFFFFFFFF,
        xor = 0xFFFFFFFF) poly;
```

Listing 1. Custom CRC polynomial in P4.

The parameters used to create the same polynomial as used in Ethernet, are shown in Listing 1. However, the CRC calculation using this polynomial results in a little-endian version of the CRC value, while the correct value must be in big-endian. To solve this, we used bit masking and operations to swap the 4 bytes from little to big-endian.

V. EXPERIMENTS

We performed three different experiments to analyze RoCEv1 and our implementation. In the first experiment, we analyzed how a RoCEv1 session is established and what values are used. Secondly, we tested our implementation to determine if it worked correctly. Finally, we analyzed the performance of the implementation to evaluate packet loss.

A. RoCEv1 between two hosts

While the Infiniband architecture documentation describes the purpose of the fields in each header, we decided to perform an experiment to determine the actual values used between two compliant RoCEv1 hosts. We established a connection between the Dell and Supermicro, both using their Mellanox NICs. From the Mellanox programming manual [16], we used an example program written in C, which included four different RDMA operations. We modified the program to only send one single RDMA write-only packet from the Dell to the Supermicro server. Subsequently, we examined the receiving host to see if its buffer was modified. With Wireshark, a packet capture is created to evaluate the RoCEv1 packet.

B. RoCEv1 from the switch

We tested our implementation where RoCEv1 traffic is supposed to flow from the switch to the Supermicro Mellanox NIC. To evaluate if the implementation is working correctly, we sent three TCP packets from the Dell, crafted using Scapy. Scapy is a Python program used to craft and dissect network packets. The Scapy script is shown in Listing 2.

```
1  sendp(Ether()/IPv6(
2      src="fc00::5555:6666:7777:8888",
3      dst="fc00::1111:2222:3333:4444")/
4          TCP(dport=111,sport=222,
5              seq=0x1212,ack=0x3434),
6      iface="rename5")
7  sendp(Ether()/IPv6(
8      src="fc00::1111:2222:3333:4444"
9      dst="fc00::5555:6666:7777:8888")/
10         TCP(dport=222,sport=111,
11             seq=0x3435,ack=0x1213),
12     iface="rename5")
13 sendp(Ether()/IPv6(
14     src="fc00::5555:6666:7777:8888",
15     dst="fc00::1111:2222:3333:4444")/
16         TCP(dport=111,sport=222,
17             seq=0x1214,ack=0x3436),
18     iface="rename5")
```

Listing 2. Scapy script for sending the three packets.

The packets we send include an Ethernet, IPv6, and TCP header. Line 1 up to 7 show this for the first packet. The switch forwards traffic between two interfaces on the Dell. The telemetry packet is forwarded to the Mellanox NIC in the Supermicro system. The IPv6 addresses (blue and red), ports (magenta), sequence number (orange), and acknowledgment number (green) are arbitrary values to be able to confirm that the data ends up in the Supermicro's persistent storage. We examine the file on the disk where the telemetry data is supposed to be stored to evaluate the implementation. This experiment will be successful if the payload of the incoming packets ends up in this file.

C. Performance

Finally, we created an experiment to analyze packet loss under different data rates. To examine the performance of memory mapping, we compared it to volatile memory. We wrote a script in Bash that automatically initializes the packet generator and telemetry server. For experiments that used memory mapping, a 10 GB file was initialized before starting the telemetry collector. The bandwidth of the links that these packets traverse is 100 Gbit/s.

The packet generator can send packets with a configurable time interval between them, i.e. *frequency*. We performed 40 unique experiments with a frequency ranging from 25 to 65 nanoseconds.

The packet rate is equal to the inverse of the traffic's frequency. For example, if the frequency is 20 nanoseconds we calculate the rate using the following formula:

$$50,000,000\, packets/second = \frac{1}{20 * 10^{-9}\, seconds}$$

This rate allows us to calculate the link's throughput. Every RoCEv1 packet is 146 bytes in total, including a 48 byte payload. The equivalent network throughput for this range is 17.5 to 46.7 Gbit/s. The NIC stores the payload of the received packet. With the receiving buffer defined as 10 GB, we know how many bytes the buffer should be able to receive. For each data point, we performed ten measurements. We calculate the ratio of correctly stored packets by dividing the stored packets by the total amount sent.

VI. Results

In the following subsections we present the results of the three experiments we previously described.

A. RoCEv1 between two hosts

Figure 5 shows the Infiniband headers of the RoCEv1 packet generated by the Mellanox NIC on the Dell. Although the packet also contains an Ethernet header, this is not of interest. Therefore, we did not include this in the figure. We examined the values in the headers that are specific to the RoCE protocol and RDMA write-only operation.

The static values are the next header (27), opcode (10), solicited event (0), migrate request (1), and header version (0). The dynamic values used for RDMA session maintenance are the destination queue pair (red), virtual address (blue), remote key (orange).

Listing 3 shows the output of the C program we used in this experiment. We compare the highlighted values of Listing 3 to Figure 5 to confirm the RDMA write worked. The virtual address (blue) and remote key (orange) correspond to each other: the hexadecimal numbers in Figure 5 have the same value as the decimals in Listing 3. The destination queue pair numbers (red) have the same hexadecimal values. The data that is written to the buffer of the host is also shown in the

```
▾ InfiniBand
  ▾ Global Route Header
      0110 .... = IP Version: 6
      .... 0000 0000 .... = Traffic Class: 0
      .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0
      Payload Length: 56
      Next Header: 27
      Hop Limit: 64
      Source GID: ::ffff:10.1.2.1
      Destination GID: ::ffff:10.1.2.2
  ▾ Base Transport Header
      Opcode: Reliable Connection (RC) - RDMA WRITE Only (10)
      0... .... = Solicited Event: False
      .1.. .... = MigReq: True
      ..11 .... = Pad Count: 3
      .... 0000 = Header Version: 0
      Partition Key: 65535
      Reserved: 00
      Destination Queue Pair: 0x000932
      1... .... = Acknowledge Request: True
      .000 0000 = Reserved (7 bits): 0
      Packet Sequence Number: 1
  ▾ RETH - RDMA Extended Transport Header
      Virtual Address: 26600384
      Remote Key: 178013
      DMA Length: 21
    Invariant CRC: 0xdd492f73
▾ Data (24 bytes)
    Data: 52444d41207772697465206f7065726174696f6e00000000
    [Length: 24]
```

Fig. 5. RoCEv1 packet for RDMA write-only operation generated as an example.

output (magenta). This data corresponds to the payload of the packet in Figure 5. The data is the ASCII representation of the string "RDMA Write operation".

```
[rutger@sne-dtn-04 rdma-writeonly]$ ./rdma-tutorial -d mlx5_1 -i 1 -g 2
...
TCP connection was established
...
MR was registered with addr=0x195e3c0,
    lkey=0x2b75d, rkey=0x2b75d, flags=0x7
QP was created, QP number=0x932
...
completion was found in CQ with status 0x0
Contents of server buffer: 'RDMA write operation'

test result is 0
```

Listing 3. Output of RDMA write operation on the server.

B. RoCEv1 from the switch

Listing 4 shows the written bytes in the Supermicro's file. This includes the header information from three packets that were sent with Scapy. The first 32 bytes contain the source (blue) and destination (red) addresses from the original packet (subnet fc00::/64). The next bytes contain the source and destination ports (magenta): 111 and 222. The sequence (orange) and acknowledgment (green) numbers are placed

```
Supermicro $ hexdump -C /mnt/nvme/output5
00 | fc00 0000 0000 0000 5555 6666 7777 8888
10 | fc00 0000 0000 0000 1111 2222 3333 4444
20 | 00de 006f 0000 1212 0000 3434 0000 0000
30 | fc00 0000 0000 0000 1111 2222 3333 4444
40 | fc00 0000 0000 0000 5555 6666 7777 8888
50 | 006f 00de 0000 3435 0000 1213 0000 0001
60 | fc00 0000 0000 0000 5555 6666 7777 8888
70 | fc00 0000 0000 0000 1111 2222 3333 4444
80 | 00de 006f 0000 1214 0000 3436 0000 0002
```

Listing 4. Hexdump of file on the Supermicro (note: this has been slightly modified for readability).

after the port information. Finally, the last 4 bytes (black) contain data from a counter in the switch. This pattern repeats for each packet stored in memory.

C. Performance

Figure 6 shows the performance experiments, indicating the ratio of stored packets versus the packet rate. We first averaged ten measurements per data point and we then connected these values with a line to better show the trend. The surrounding areas show the standard deviation. In this graph we observe packet loss earlier when data is stored to memory, compared to a memory-mapped file. Furthermore, from around 32 million packets per seconds, less than 0.4% of the packets get stored.

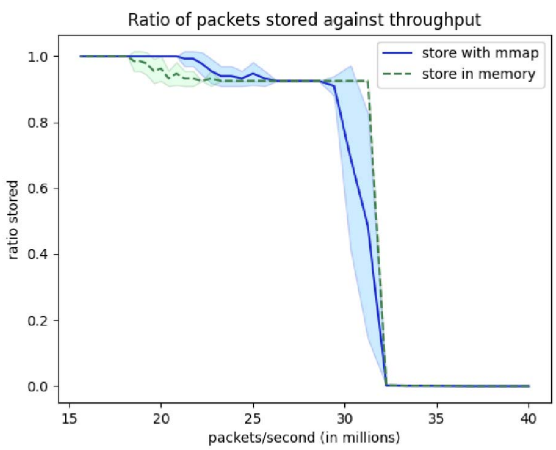

Fig. 6. Percentage of packets stored with a given amount of packets per second.

Table I shows an example of a measurement from the experiments. From the measurements from 17 to 29 million packets per second, we observed that either all of the packets got stored correctly or there was a gap of exactly 2^{24} packets, as shown in Table I. We provide an explanation for this behavior in the Discussion.

Million packets per second	Packets received	Packets total	Ratio
27.03	206918997	223696213	0.925

TABLE I
SINGLE MEASUREMENT POINT AT 27 MILLION PACKETS PER SECOND.

VII. Discussion

Thanks to the results in Section VI-A, we could observe the values for the header fields in the RoCEv1 packet. This experiment confirmed the exact values we needed to use in our packets for a RoCE RDMA write-only operation. During the analysis of the header, we found out a number of interesting properties of the Global Route Header. The session to the Supermicro used IPv4 addresses, while the header is an IPv6 address. In the Infiniband header, we can see that the original IPv4 header is embedded into IPv6, with IPv4 mapping (*::ffff:a.b.c.d*).

The results in Section VI-B allowed us to determine that our implementation works correctly. In the Supermicro server's file, we looked at the payload of all three packets. We could confirm that the original packets were correctly stored in the final file with RDMA commands from the switch.

Finally, based on the results in Section VI-C, we were able to analyze the performance of our implementation. We were expecting less performance from memory mapping compared to volatile memory, because the memory mapped area lies in slower storage. However, the difference between memory mapping and volatile memory was not significant. We concluded from this that memory mapping is not a performance constraining factor for this application. We must point out that this experiment was conducted over 100 Gbit/s per links, but the performance of this system did not reflect this. The first packet loss occurred at around 17 million packets per second when writing to memory, and 20 million packets per second when writing to a memory-mapped file. The throughput over the link at these points was 20 to 23 Gbit/s. This told us that packet loss is not due to the link's capacity. The limiting factor in the performance instead had to do with the sequence number. The RDMA implementation of Mellanox expects a sequence number that is incremented by one for every packet. The sequence number is 24 bits long. In Table I, we see the difference between the processed and total amount of packets is exactly 2^{24}. When a packet is dropped before it is processed by the NIC, there occurs a gap in the sequence numbers. The NIC silently drops packets that do not contain the expected sequence number. The switch has no knowledge about a drop and even if it did, it can not re-transmit the packet. As a consequence, the NIC halts until a packet with the correct sequence number is processed. This means the NIC keeps dropping packets until the sequence number wraps around, which is 2^{24} packets later. As a result, the NIC will ignore the payload of 16 million packets when there is 1 packet dropped in the network. We did not have a sufficient amount of data to determine why the first packet is dropped. In the results, we observed a steep decrease in stored packets at 32 million packets per second. From this rate, a significant amount of packets were dropped, which caused the NIC to have frequent issues with the sequence number. This resulted in the NIC only occasionally storing packets.

The use of RDMA for transferring data has some implications. The advantage of RDMA is that no CPU involvement is required to store data. With the CPU removed from the process of receiving packets, it can no longer be a bottleneck in obtaining data from the network and storing it into memory. We show that establishing an RDMA session between a switch and server is feasible. This session allows for even less involvement of the CPU than other methods. However we must point out that in our implementation the CPU is not informed about the change to memory, as we have not developed a signaling method for making the CPU aware that a part of memory can be read.

There are multiple possible solutions for implementing the workflow after a collector gathers telemetry data. For this research, we stored the data to a file. However, the approach for gathering and processing the data further depends on the complete telemetry workflow. For instance, a workflow that does not store the data in the collector would use a different technique to forward the data to workers efficiently. In this case, it might be efficient to create large buffers in volatile memory and send data directly to workers.

Our implementation has some limitations. First of all, we considered implementing CPU signaling out of the scope of our research. However, in order to use RDMA from a switch for data telemetry in a production environment, there is a need for signaling. The reason for this is that the collector server needs to know when data is ready to be analyzed. A possible solution for implementing signaling is using the *RDMA write-only with immediate* operation in a RoCEv1 packet. This operation allows for an RDMA write while signaling the CPU with a 32-bit immediate value [15]. This value could signal the CPU that a specific action needs to be taken on the previously received telemetry data. If each packet includes a signal for the CPU, this would be a CPU intensive approach. This operation could be sent once every x amount of packets to decrease CPU utilization.

Secondly, P4 does not support the processing of packet trailers. In order to add data to the end of a packet, all the bytes that come before it must be parsed as headers. Additionally, implementations of P4 can have a maximum header length. If this maximum is less than the maximum total packet size, implementations of this protocol would only be able to create packets smaller or equal to the maximum header length. Another implementation dependent feature is the checksum calculation. The P4 specification has no requirements on the implementation of a CRC function [4]. For this reason, a manufacturer has to implement it as an external function.

VIII. Conclusion

In our research we investigated how RDMA can be combined with P4 in order to collect telemetry data efficiently. We created the part of a telemetry workflow where data from network packets is extracted and transferred to a collector. We examined how telemetry data can be encapsulated in RDMA messages. By experimenting with and implementing the RoCEv1 protocol, we were able to craft RoCEv1 packets on a P4 switch. The payload of this packet carries the telemetry data. This data is written to the remote collector using an RDMA write-only operation. An RDMA session can be maintained on the switch by keeping the state of the variables that are required for this operation in the memory of the switch. On the collector, the NIC can transfer the received telemetry data to persistent storage. This operation is performed using the memory-mapped file on the NVMe device. We were able to achieve a rate of around 20 million packets per second with memory mapping without any packet loss. Using volatile memory for the storage, we achieved a rate of 17 million packets per second without packet loss. Since the NIC will ignore 16 million packets if a single packet is dropped in the network, this implementation would only be useful in a lossless network. From this research, we can conclude that the implementation of RoCEv1 in P4 can be used to extract telemetry data and save telemetry data directly to persistent storage.

A. Future Work

We believe that our work has a lot of potential for adoption in two related use cases: processing large amounts of telemetry data and transferring state between P4 devices and compute systems. While our work demonstrates the usability of the proposed techniques it does still require additional work. The software is available on GitHub[1,2].

First, it is interesting to compare our implementation with other methods for high-speed packet captures. For instance, emerging technologies to which we can compare to are the Data Plane Development Kit (DPDK) and the extended Berkeley Packet Filter (eBPF).

Secondly, there are a number of areas that could be optimized. The system performance of the collector could be improved to allow faster storage of the data. One of the possible methods is to use NVMe over fabric instead of memory mapping a disk to virtual memory. Another possible method is to use the Storage Performance Development Kit (SPDK). In our application, we used RDMA to send data to virtual memory. The operating system mapped this virtual memory to a file on the disk. SPDK provides a method to bypass this step and interact directly with the NVMe drive from the NIC.

Finally, if we refer back to Fig.1 we see that the next bottleneck that needs to be addressed is the step from the collector to the workers. Implementing this next part of the pipeline requires us to determine how a worker can retrieve the data from the collector while maintaining high performance. Part of the solution could involve signaling to the CPU the presence of new data, with the approach we proposed in Section VII.

Acknowledgment

Part of this work was funded by the RoN - Research on Networks - from SURFnet. We are particularly thankful to Ronald van der Pol and Marijke Kaat. This research is also part of the 2STiC program - Security, Stability and Transparency in inter-network Communication - (https://www.2stic.nl/). This work was inspired by the High Touch Services research being done at ESnet, Lawrence Berkeley National Laboratory.

References

[1] N. Van Tu, J. Hyun, G. Y. Kim, J.-H. Yoo, and J. W.-K. Hong, "Int-collector: A high-performance collector for in-band network telemetry," IEEE, pp. 10–18, 2018.

[2] T. P. A. W. Group, "In-Band Network Telemetry (INT) Dataplane Specification," 2020. [Online]. Available: https://github.com/p4lang/p4-applications/blob/master/docs/INT_latest.pdf

[3] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," 2015.

[1] https://github.com/rutgerbeltman/telemetry-rdma-p4-server/
[2] https://github.com/rutgerbeltman/telemetry-rdma-p4-switch/

[4] T. P. L. Consortium, "P4-16 language specification," 2019. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf
[5] A. Gulenko, M. Wallschlager, and O. Kao, "A practical implementation of in-band network telemetry in open vswitch," in *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. IEEE, 2018, pp. 1–4.
[6] S. Knossen, J. Hill, and P. Grosso, "Hop recording and forwarding state logging: Two implementations for path tracking in p4," in *2019 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*. IEEE, 2019, pp. 36–47.
[7] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie, "In-band network telemetry (int)," 2016. [Online]. Available: https://p4.org/assets/INT-current-spec.pdf
[8] M. Budiu. (2017) Programming networks with p4. [Online]. Available: https://blogs.vmware.com/research/2017/04/07/programming-networks-p4/
[9] D. Works, "DMA Fundamentals on Various PC Platforms."
[10] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," Internet Requests for Comments, RFC Editor, RFC 5040, October 2007.
[11] J. Pinkerton and E. Deleganes, "Direct data placement protocol (ddp) / remote direct memory access protocol (rdmap) security," Internet Requests for Comments, RFC Editor, RFC 5042, October 2007.
[12] I. trade association. (2002) Infiniband™ architecture specification frequently asked questions.
[13] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul, "Efficient data transfer protocols for big data," in *2012 IEEE 8th International Conference on E-Science*. IEEE, 2012, pp. 1–9.
[14] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018, pp. 1–7.
[15] I. T. Association, "InfiniBand Architecture Specification Volume 1, Release 1.2.1," 2007. [Online]. Available: https://www.afs.enea.it/asantoro/V1r1_2_1.Release_12062007.pdf
[16] *Mellanox Adapters Programmer's Reference Manual (PRM)*, Mellanox Technologies, 2016. [Online]. Available: https://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf