High Speed DAQ with DPDK

June - August 2016

Author:
Saiyida Noor Fatima

Supervisors:
Niko Neufeld
Sebastian Valat

CERN Openlab Summer Student Report 2016

Acknowledgment

My sincere thanks to my supervisor Niko Neufeld, Sebastien Valat, Balazs Voneki, LHCb Secretariat and the entire LHCb team; without their guidance this project would not be possible. I would like to express my gratitude towards CERN Openlab team, my parents and fellow Openlab summer students for their constant support.

Project Specification

The experiments at CERN generate mountainous amounts of data. In the first run of the Large Hadron Collider (LHC) [1], 30 petabytes of data was produced annually, and up to 40 Terabits per second amount of data more is expected to be produced during the future run of LHC in 2020. To filter the data coming from the LHCb [2] detector, LHCb team is working on a benchmark application called DAQPIPE [3] which emulates the data coming from the detector in a local area network. One of the major challenges faced by DAQPIPE is to prevent any type of data loss by implementing high-speed networks which can bear the load of the data coming in. The aim of the project is to implement a DPDK [4] driver for DAQPIPE to create a high speed packet processing, transfer and communication mechanism for future LHCb upgrades.

Abstract

DPDK is a new frame-work for very fast software defined networking. It allows multi-stage Ethernet networks for DAQ to be implemented with very cost-effective hardware by offloading all intelligence and most of the buffering into commodity servers. The LHCb data acquisition for Run3 will need a 40 Terabit/s network, where 100 Gigabit Ethernet is one of the interesting candidates. In this project we aim to port the existing DAQ exerciser software of LHCb, DAQPIPE, to DPDK and perform tests on state of the art network hardware.

Table of Contents

Table of Figures

1 Introduction

The experiments at CERN generate huge amount of data which has to be filtered, processed and stored. The LHCb is working on DAQ Protocol-Independent Performance Evaluator (DAQPIPE), a benchmark application to test network fabrics for the future LHCb upgrade, for emulating the data input, filtration and storage.

1.1 DAQPIPE

DAQPIPE emulates reading data from input cards, which in a real DAQ is connected to the detector. Data coming from one event is correlated and needs to be brought together in one place. This event data is spread over multiple nodes (~ 500 to 1000 for the LHCb DAQ upgrade) so DAQPIPE has to aggregate all the parts. The LHCb data acquisition will need a 40 Terabit/s network in 2020. 40 Tb/s data coming from the LHCb detector in the DAQ system, which consists of 500 computing nodes, makes the required throughput to be 80 Gb/s.

1.1.1 Components

DAQPIPE has 4 components:

i. The **Readout Unit** reads data from the detector (or input cards in case of emulation) and sends it to the Builder Unit.
ii. The **Builder Unit** aggregates the data of one particular event and sends it to a Filter Unit.
iii. The **Filter Unit** has to decide whether or not the event is to be stored.
iv. The **Event Manager** manages the job of dispatching between the Readout Unit and Builder Unit.

1.1.2 Supported Networks

DAQPIPE supports various network interfaces [5] which can be paired with different technologies to provide user with multiple pair options while keeping the user transparent to the underlying working. Some of the networks supported by the DAQPIPE include:

i. MPI
ii. LibFabric
iii. Infiniband VERBS
iv. TCP/IP
v. RapidIO
vi. PSM2

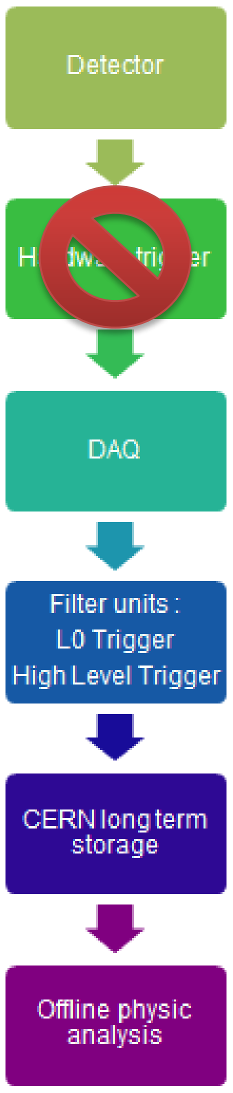


Figure 1. DAQ System Flowchart

Figure 1 Source: https://openfabrics.org/images/eventpresos/2016presentations/312LHCBEventBld.pdf

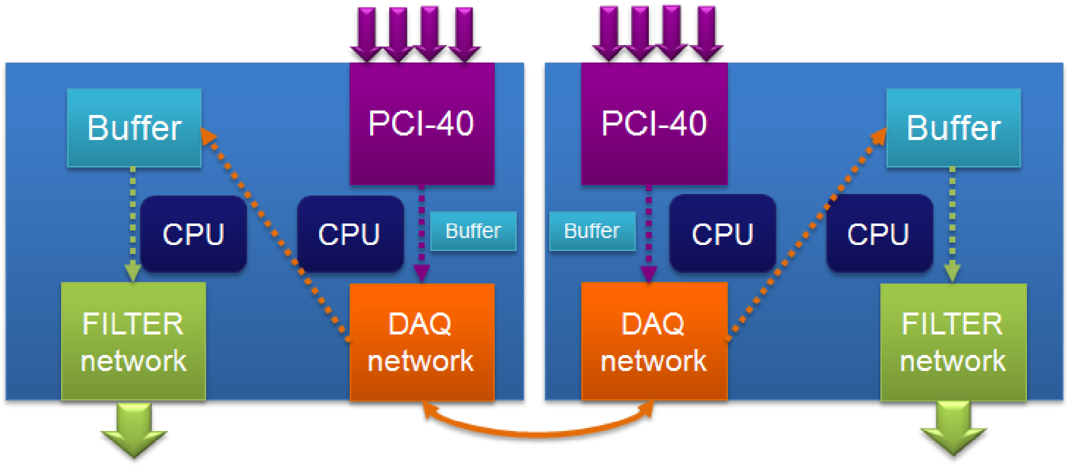


Figure 2. DAQ System Hardware View

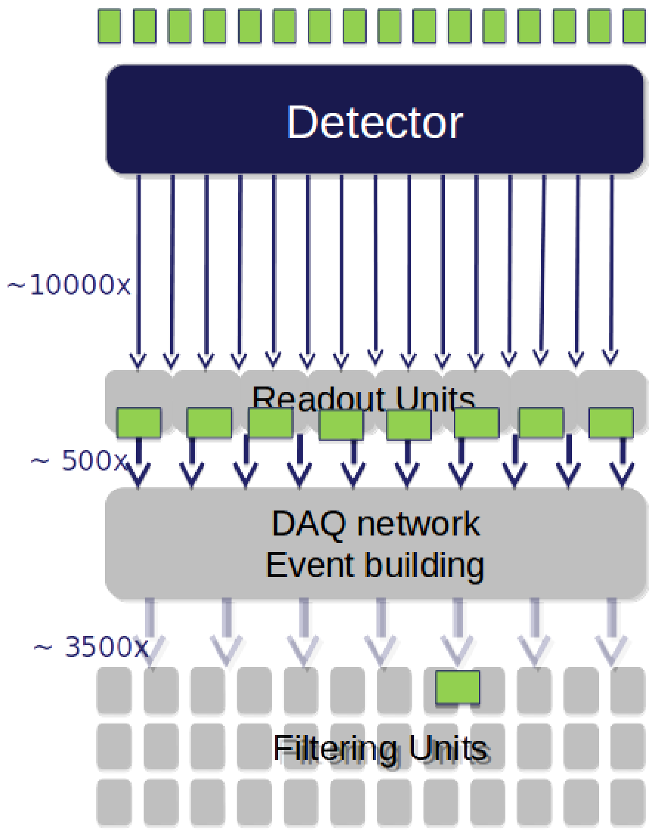


Figure 3. DAQ System View

Figure 2 and 3 Source: https://openfabrics.org/images/eventpresos/2016presentations/312LHCBEventBld.pdf

1.2 DPDK

DPDK is a set of libraries and drivers for fast packet processing. It is designed to run on any processor including Intel x86, IBM Power 8, EZchip TILE-Gx and ARM. It is a new frame-work for very fast software defined networking. It allows multi-stage Ethernet networks for DAQ to be implemented with very cost-effective hardware by offloading all intelligence and most of the buffering into commodity servers. It works with Ethernet. DPDK is aimed for switch implementation, however, in this project, it is being used for data transfer and communication.

1.2.1 Components

DPDK consists of the following major modules:

i. **Environment Abstraction Layer (EAL)**: It is responsible for gaining access to low level resources such as hardware and memory space.

ii. **Memory Manager**: Responsible for allocating pools of objects in memory space.

iii. **Buffer Manager**: Assigns outgoing and incoming (TX/RX) packet queues.

iv. **Queue Manager**: Manages the queues allocated by the buffer manager.

v. **Packet Flow Classifier:** Implements hash based flow classification to place packets into flows for processing.

vi. **Poll Mode Drivers**: These are designed to work without Interrupt-based signalling, which saves the CPU time.

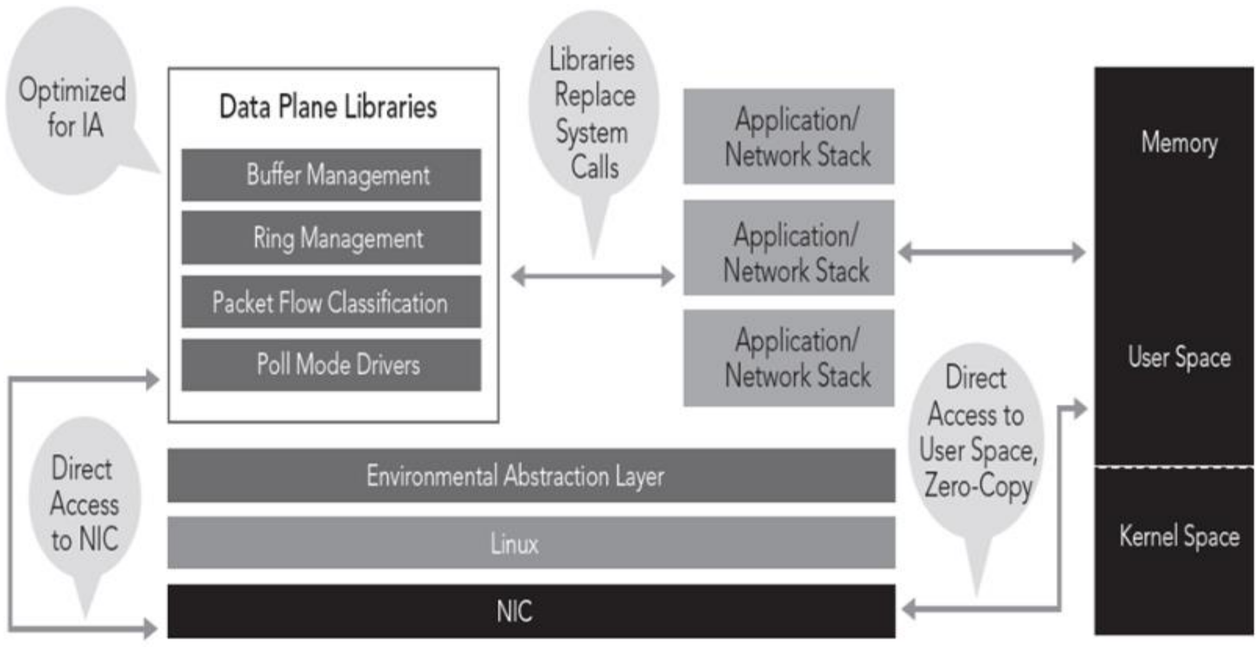

Optimized for IA
Data Plane Libraries
Buffer Management
Ring Management
Packet Flow Classification
Poll Mode Drivers
Libraries Replace System Calls
Application/ Network Stack
Application/ Network Stack
Application/ Network Stack
Memory
User Space
Kernel Space
Direct Access to User Space, Zero-Copy
Direct Access to NIC
Environmental Abstraction Layer
Linux
NIC

Figure 4. DPDK Components

Figure 4 Source: https://www.linkedin.com/pulse/dpdk-layman-aayush-shrut

2 Problem Statement

2.1 Problem

As explained earlier in section 1.1, it is estimated that in the 2020 upgrade, Run3, the LHCb data acquisition will need a 40 Terabit/s network, where 100 Gigabit Ethernet is one of the most interesting candidates.

2.1.1 Desired Throughput

80 Gb/s is the throughput required by the network in DAQPIPE to transfer data among Readout Units and Builder Units.

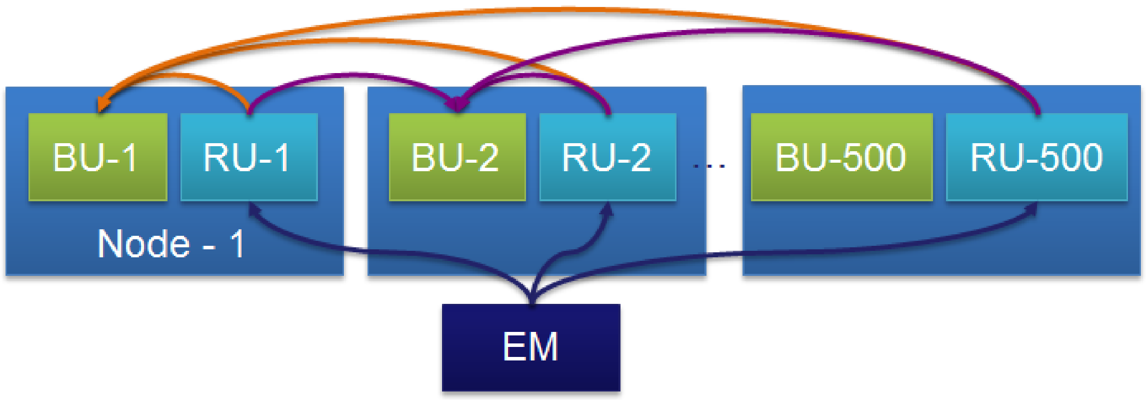


Figure 5. Readout Units and Builder Units communication chanels

2.1.2 Linux Kernel Network Stack - Performance Limitations

Linux is not designed as a high operation and high throughput for network traffic operating system, but as a general purpose OS. Forwarding process of TCP/IP packets in Linux goes through many layers of the network stack, in both, reception of the packets and transmittance of the packet. These layers of the operating systems don't give the user much control over the hardware features. Therefore, high throughput communication and specialized workloads require methods other than the general purpose methods working with Linux OS which cannot get us very high throughput as desired in the LHCb DAQPIPE.

2.2 Solution

Over the years, many attempts have been made to overcome the Linux kernel performance limitations. Specialized APIs have been designed to increase packet reception at high speed in Linux environment.

2.2.1 Kernel Bypass

One of the most popularly used techniques used to send and receive more packets from existing hardware is by working around the Linux kernel networking stack. This is known as Kernel Bypassing [6].

2.2.2 DPDK Driver for DAQPIPE

DPDK is a set of libraries and drivers for high speed packet processing. It allows the user to send and receive packets with the minimum number of CPU cycles. It also allows the development of fast packet capture algorithms.

In this project, we are looking to implement a driver of DPDK for DAQPIPE to produce a high speed network for the future LHCb upgrades.

Comparisons of the packet send/receive path in Linux and DPDK is shown in the following figures.

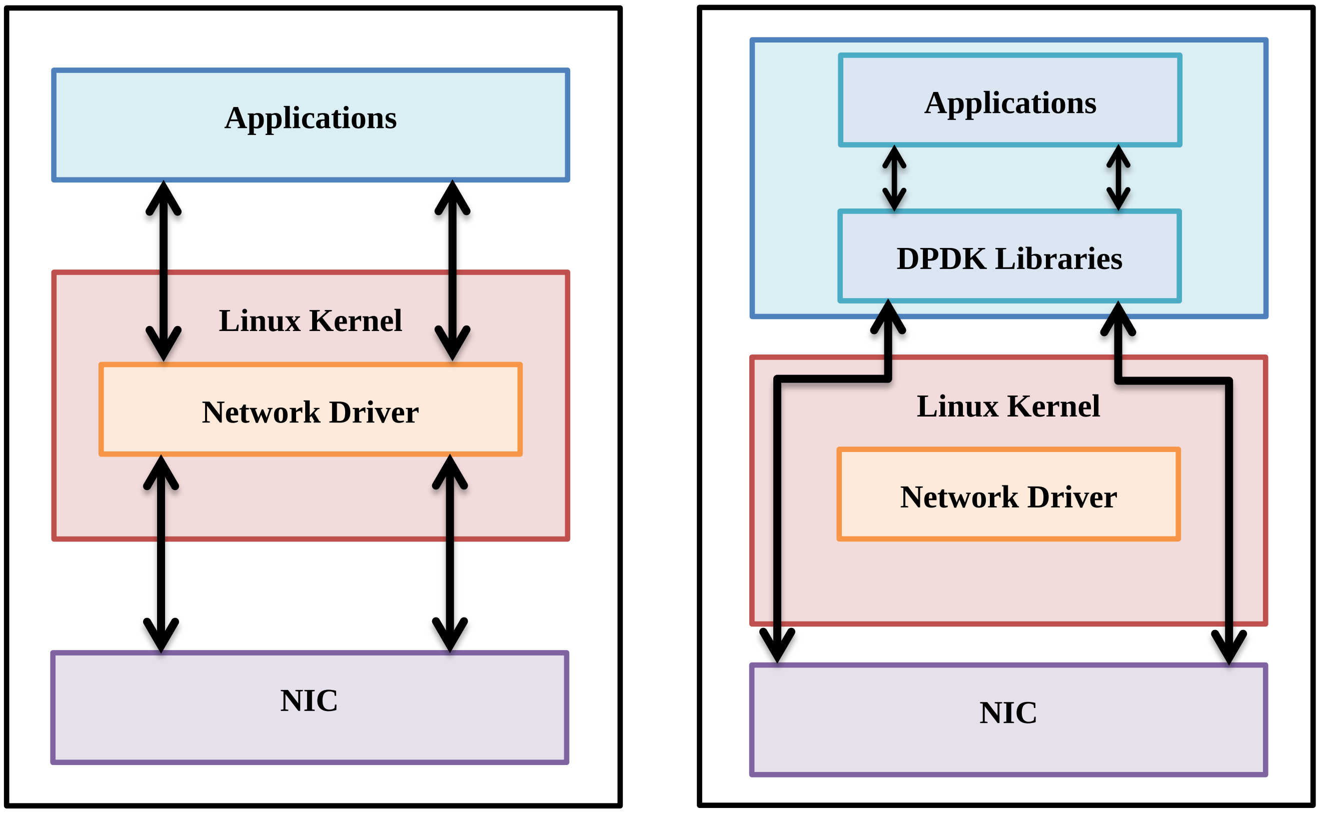


Figure 6. Packet Transfer with and without DPDK

3 Forwarding Packets with DPDK

3.1 API Examples

As DPDK is a set of libraries, it has the following major topics in which the headers can be categorized:

i. Device
ii. Memory
iii. Timers
iv. Locks
v. CPU arch
vi. CPU multicore
vii. Layers
viii. QoS
ix. Hashes
x. Containers
xi. Packet framework
xii. Basic
xiii. Debug

For a simple hello world program:

i. The EAL has to be initialized first.

```
ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

ii. A function to print “hello world” on console.

```
static int lcore_hello( attribute ((unused)) void *arg)
{
    unsigned lcore_id;

    lcore_id = rte_lcore_id();
    printf("hello from core %u\n", lcore_id);
    return 0;
}
```

iii. Call the function on ach available slave node.

```
RTE_LCORE_FOREACH_SLAVE(lcore_id) {
   rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
}
```

iv. Call the function on master (current) node.

```
lcore_hello(NULL);
```

3.2 Packet Construction

For a more customized communication and packet transfer, DPDK allows us to create a packet of our choice and send/receive it.

i. Initialize the EAL.

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

ii. Create a memory pool for the program.

```
packet_pool = rte_mempool_create("packet_pool",
                   nb_ports,
                   MBUF_SIZE,
                   MBUF_CACHE_SIZE,
                   MBUF_DATA_SIZE,
                   rte_pktmbuf_pool_init, NULL,
                   rte_pktmbuf_init,      NULL,
                   rte_socket_id(),
                   0);
```

iii. Allocate a packet in memory pool.

```
packet = rte_pktmbuf_alloc(packet_pool);
```

iv. Populate the packet with data, source address, destination address, and initialize packet fields.

```
pkt_size = sizeof(struct ether_hdr) +
        sizeof(struct data);
packet->data_len = pkt_size;
packet->pkt_len = pkt_size;
eth_hdr = rte_pktmbuf_mtod(packet, struct ether_hdr *);
rte_eth_macaddr_get(src_port, &eth_hdr->s_addr);
rte_eth_macaddr_get(dst_port, &eth_hdr->d_addr);
```

v. Transmit the packet to the destination port address.

```
rte_eth_tx_burst(dst_port, 0, &packet, 1);
```

vi. The receiver side can receive the incoming packets in a similar way.

```
rte_eth_rx_burst(port, 0, &packet, 1);
```

vii. Free the memory occupied by packet after processing.

```
rte_pktmbuf_free(packet);
```

3.3 Forwarding Example

Following is a basic forwarding example derived from the basicfwd.c example in the DPDK sample applications [7]. The lcore_main function is reads from an input port and writes to an output port. The forwarding loop continues for 10 seconds in this example.

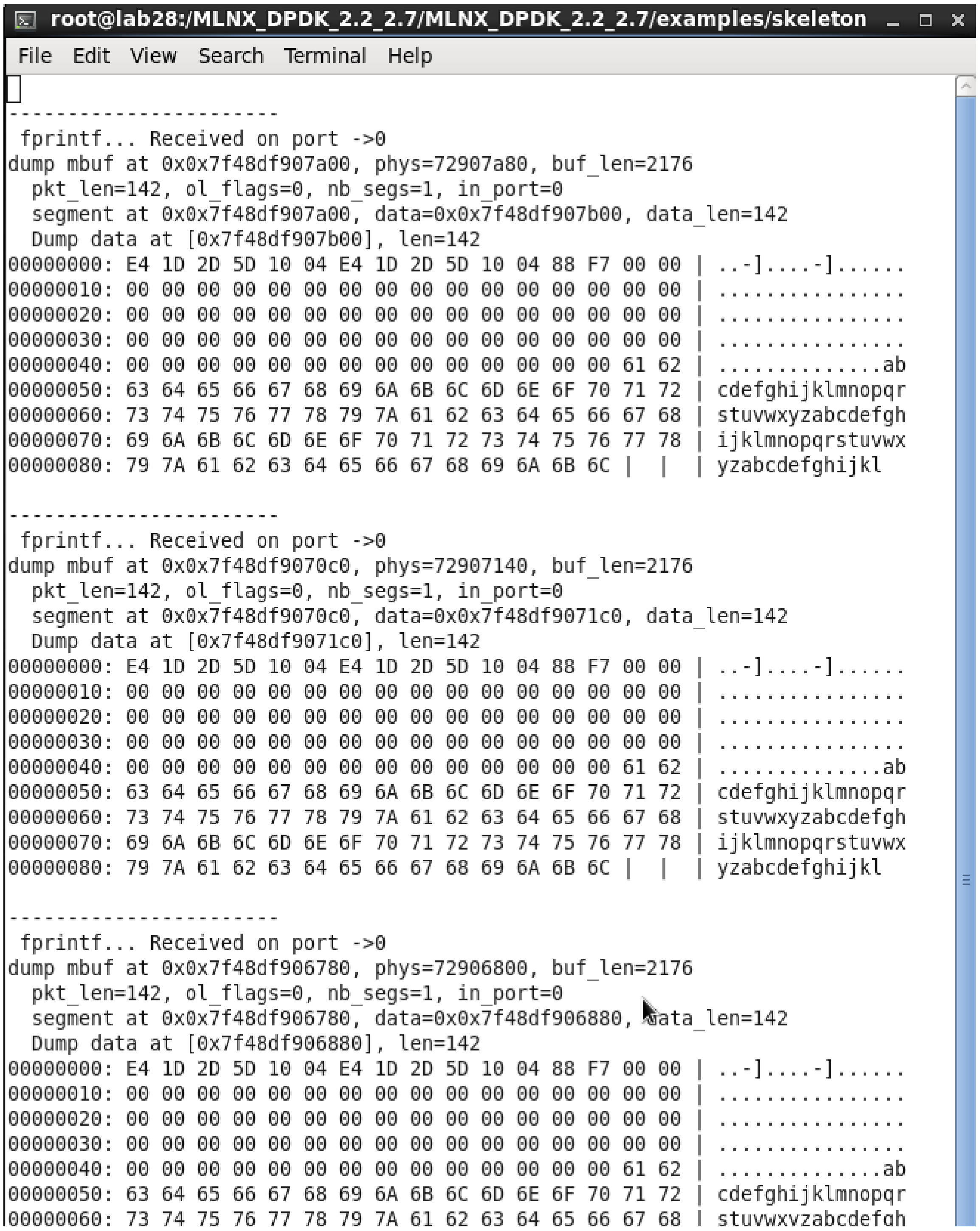
```
/*
 * The main thread that does the work, reading from
 * an input port and writing to an output port.
 */
struct message {
	char data[DATA_SIZE];
};

static __attribute__(()) void
lcore_main(void)
{
	const uint8_t nb_ports = rte_eth_dev_count();
	uint8_t port;

	for (port = 0; port < nb_ports; port++)
		if (rte_eth_dev_socket_id(port) > 0 &&
			rte_eth_dev_socket_id(port) !=
			(int)rte_socket_id())
			printf("WARNING, port %u is on remote NUMA node to "
			"polling thread.\n\tPerformance will "
			"not be optimal.\n", port);

	struct rte_mbuf *pkt;
	struct ether_hdr *eth_hdr;

	struct message obj;
	struct message *msg;
	int nb_rx = 0, nb_tx = 0, cnt = 0, pkt_size = 0;
	int count = 0;
	int k = 0;
	for (count = 0; count < DATA_SIZE; count++){
		obj.data[count] = (char)(97 + (k++));
		if (k == 26)
			k = 0;
	}
	time_t endtime = time(NULL) + 10;
	port = 0;
	while (time(NULL) < endtime) {
		cnt = rte_eth_rx_burst(port, 0, &pkt, 1);
		nb_rx += cnt;

		if (cnt > 0)
		{
			eth_hdr = rte_pktmbuf_mtod(pkt, struct ether_hdr *);

			rte_eth_macaddr_get(port, &eth_hdr->s_addr);

			//	printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02"
PRIx8
```

```
                    //        " %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",
                    //            (unsigned)port,
                    //            eth_hdr->s_addr.addr_bytes[0], eth_hdr-
>s_addr.addr_bytes[1],
                    //            eth_hdr->s_addr.addr_bytes[2], eth_hdr-
>s_addr.addr_bytes[3],
                    //            eth_hdr->s_addr.addr_bytes[4], eth_hdr-
>s_addr.addr_bytes[5]);

                    pkt_size = sizeof(struct message) + sizeof(struct
ether_hdr);
                    msg = (struct message *) (rte_pktmbuf_mtod(pkt, char*)) +
sizeof(struct ether_hdr);
                    rte_pktmbuf_free(pkt);
             }

             msg = &obj;
             pkt = rte_pktmbuf_alloc(mbuf_pool);
             pkt_size = sizeof(struct message) + sizeof(struct ether_hdr);
             pkt->data_len = pkt_size;
             pkt->pkt_len = pkt_size;
             eth_hdr = rte_pktmbuf_mtod(pkt, struct ether_hdr *);
             rte_eth_macaddr_get(port, &eth_hdr->d_addr);
             rte_eth_macaddr_get(port ^ 1, &eth_hdr->s_addr);
             eth_hdr->ether_type = htons(PTP_PROTOCOL);
             char* data;

             data = rte_pktmbuf_append(pkt, sizeof(struct message));
             if (data != NULL)
                    rte_memcpy(data, msg, sizeof(struct message));

             nb_tx += rte_eth_tx_burst(port ^ 1, 0, &pkt, 1);
      }
      printf("----\nData size: %d\nPacket size: %d\nRX : %d, TX : %d\n\n",
DATA_SIZE, pkt_size, nb_rx, nb_tx);
}
```

3.4 Packet Example

Following are the packet examples being sent and received on a port.

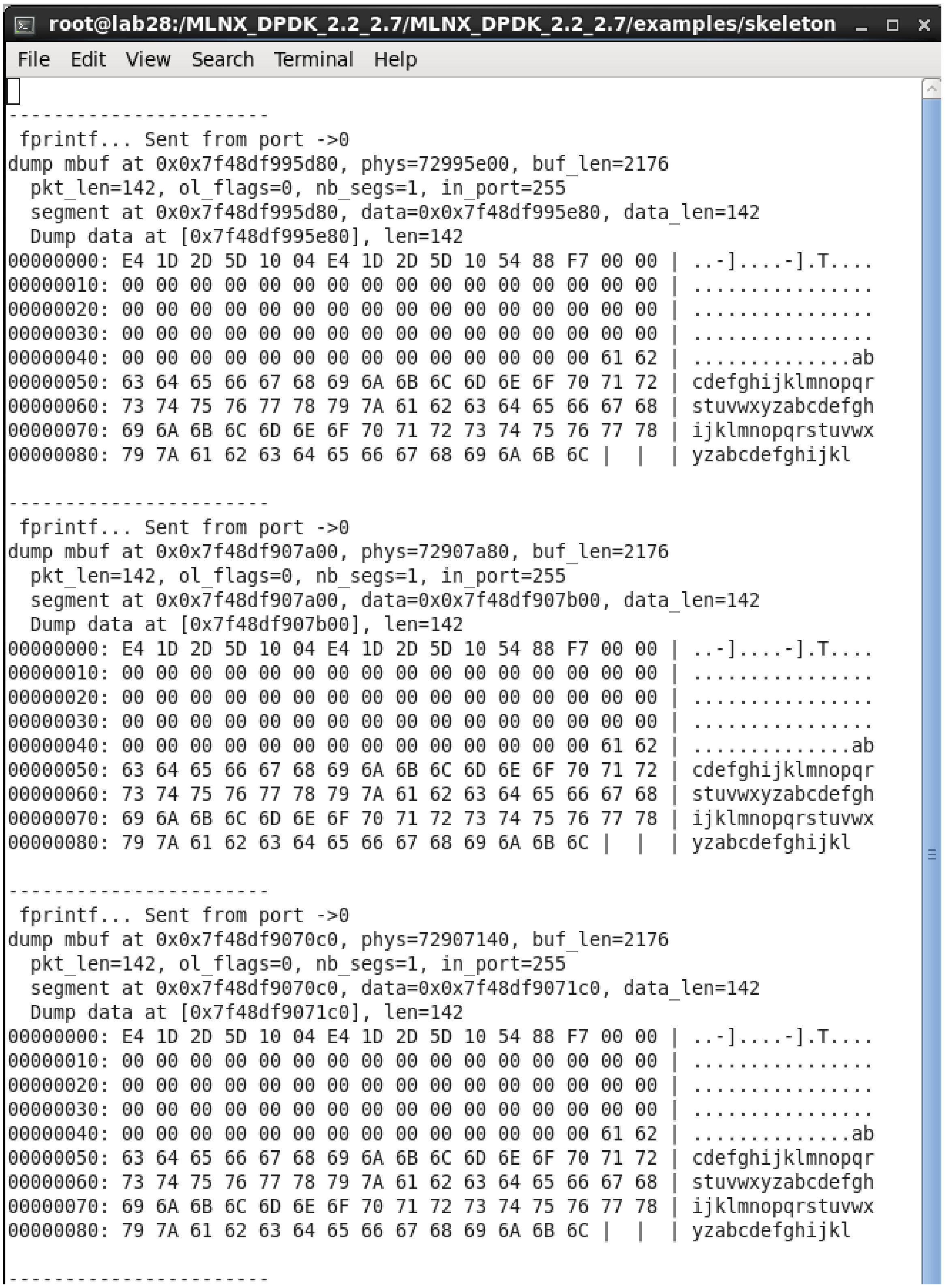
3.4.1.1 Received


root@lab28:/MLNX_DPDK_2.2_2.7/MLNX_DPDK_2.2_2.7/examples/skeleton
File Edit View Search Terminal Help

```
-----------------------
 fprintf... Received on port ->0
dump mbuf at 0x0x7f48df907a00, phys=72907a80, buf_len=2176
  pkt_len=142, ol_flags=0, nb_segs=1, in_port=0
  segment at 0x0x7f48df907a00, data=0x0x7f48df907b00, data_len=142
  Dump data at [0x7f48df907b00], len=142
00000000: E4 1D 2D 5D 10 04 E4 1D 2D 5D 10 04 88 F7 00 00 | ..-]....-]......
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 62 | ..............ab
00000050: 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 | cdefghijklmnopqr
00000060: 73 74 75 76 77 78 79 7A 61 62 63 64 65 66 67 68 | stuvwxyzabcdefgh
00000070: 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 | ijklmnopqrstuvwx
00000080: 79 7A 61 62 63 64 65 66 67 68 69 6A 6B 6C |  |  | yzabcdefghijkl

-----------------------
 fprintf... Received on port ->0
dump mbuf at 0x0x7f48df9070c0, phys=72907140, buf_len=2176
  pkt_len=142, ol_flags=0, nb_segs=1, in_port=0
  segment at 0x0x7f48df9070c0, data=0x0x7f48df9071c0, data_len=142
  Dump data at [0x7f48df9071c0], len=142
00000000: E4 1D 2D 5D 10 04 E4 1D 2D 5D 10 04 88 F7 00 00 | ..-]....-]......
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 62 | ..............ab
00000050: 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 | cdefghijklmnopqr
00000060: 73 74 75 76 77 78 79 7A 61 62 63 64 65 66 67 68 | stuvwxyzabcdefgh
00000070: 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 | ijklmnopqrstuvwx
00000080: 79 7A 61 62 63 64 65 66 67 68 69 6A 6B 6C |  |  | yzabcdefghijkl

-----------------------
 fprintf... Received on port ->0
dump mbuf at 0x0x7f48df906780, phys=72906800, buf_len=2176
  pkt_len=142, ol_flags=0, nb_segs=1, in_port=0
  segment at 0x0x7f48df906780, data=0x0x7f48df906880, data_len=142
  Dump data at [0x7f48df906880], len=142
00000000: E4 1D 2D 5D 10 04 E4 1D 2D 5D 10 04 88 F7 00 00 | ..-]....-]......
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 62 | ..............ab
00000050: 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 | cdefghijklmnopqr
00000060: 73 74 75 76 77 78 79 7A 61 62 63 64 65 66 67 68 | stuvwxyzabcdefgh
```

Figure 7. Received Packet

3.4.1.2 Sent



root@lab28:/MLNX_DPDK_2.2_2.7/MLNX_DPDK_2.2_2.7/examples/skeleton

```
----------------------
 fprintf... Sent from port ->0
dump mbuf at 0x0x7f48df995d80, phys=72995e00, buf_len=2176
  pkt_len=142, ol_flags=0, nb_segs=1, in_port=255
  segment at 0x0x7f48df995d80, data=0x0x7f48df995e80, data_len=142
  Dump data at [0x7f48df995e80], len=142
00000000: E4 1D 2D 5D 10 04 E4 1D 2D 5D 10 54 88 F7 00 00 | ..-].....-].T....
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 62 | ..............ab
00000050: 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 | cdefghijklmnopqr
00000060: 73 74 75 76 77 78 79 7A 61 62 63 64 65 66 67 68 | stuvwxyzabcdefgh
00000070: 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 | ijklmnopqrstuvwx
00000080: 79 7A 61 62 63 64 65 66 67 68 69 6A 6B 6C |  |  | yzabcdefghijkl

----------------------
 fprintf... Sent from port ->0
dump mbuf at 0x0x7f48df907a00, phys=72907a80, buf_len=2176
  pkt_len=142, ol_flags=0, nb_segs=1, in_port=255
  segment at 0x0x7f48df907a00, data=0x0x7f48df907b00, data_len=142
  Dump data at [0x7f48df907b00], len=142
00000000: E4 1D 2D 5D 10 04 E4 1D 2D 5D 10 54 88 F7 00 00 | ..-].....-].T....
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 62 | ..............ab
00000050: 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 | cdefghijklmnopqr
00000060: 73 74 75 76 77 78 79 7A 61 62 63 64 65 66 67 68 | stuvwxyzabcdefgh
00000070: 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 | ijklmnopqrstuvwx
00000080: 79 7A 61 62 63 64 65 66 67 68 69 6A 6B 6C |  |  | yzabcdefghijkl

----------------------
 fprintf... Sent from port ->0
dump mbuf at 0x0x7f48df9070c0, phys=72907140, buf_len=2176
  pkt_len=142, ol_flags=0, nb_segs=1, in_port=255
  segment at 0x0x7f48df9070c0, data=0x0x7f48df9071c0, data_len=142
  Dump data at [0x7f48df9071c0], len=142
00000000: E4 1D 2D 5D 10 04 E4 1D 2D 5D 10 54 88 F7 00 00 | ..-].....-].T....
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 62 | ..............ab
00000050: 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 | cdefghijklmnopqr
00000060: 73 74 75 76 77 78 79 7A 61 62 63 64 65 66 67 68 | stuvwxyzabcdefgh
00000070: 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 | ijklmnopqrstuvwx
00000080: 79 7A 61 62 63 64 65 66 67 68 69 6A 6B 6C |  |  | yzabcdefghijkl

----------------------
```

Figure 8. Sent Packets

4 Results

4.1 Bandwidth

The maximum bandwidth achieved is on packet size = 1200. Figure 9 shows how the bandwidth is affected by packet size. The graph displays 5 runs each for packet sizes: 64, 128, 256, 512, 1024, 1200 bytes.

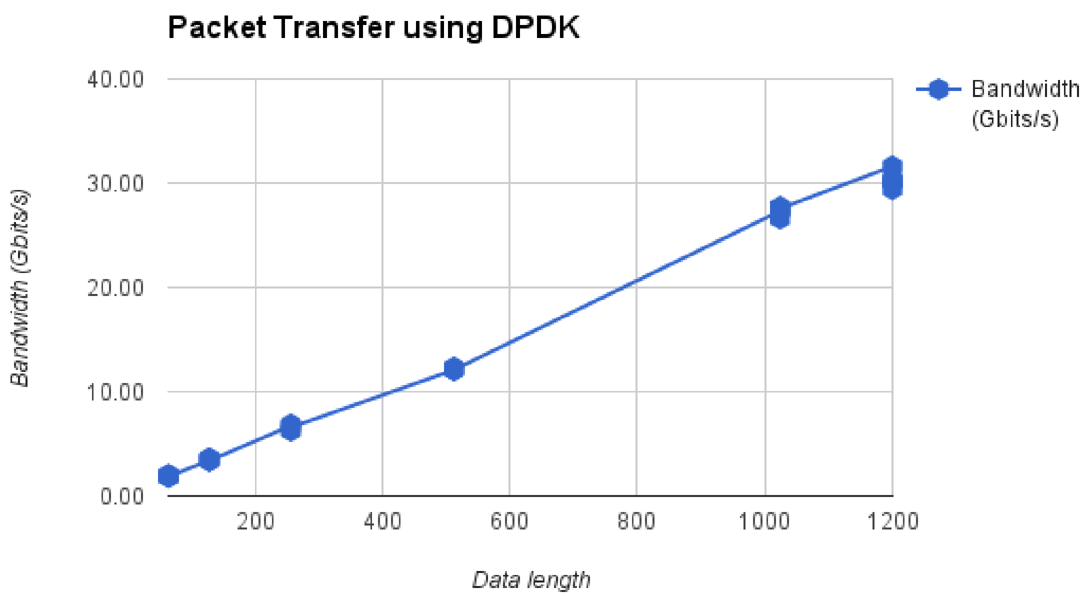


Figure 9. Bandwidth Results

5 Issues

A few issues faced during the project include:

I. DPDK requires specialized hardware [8] support to work. Initially, the NIC chosen to work with was Chelsio, but due to the compatibility constraints between the Chelsio driver and the OS used by the test servers, the cards had to be changed to Mellanox. This caused a lag in the timeline estimated for the project.

II. DPDK is aimed for switch implementation. However, we are using it as a communication protocol. Hence, it was quite difficult to find any example or previous work regarding our area of interest.

III. Currently, there are no DPDK experts in the computing team in LHCb. Therefore, obtaining expert guidance was not possible.

IV. The maximum packet sizes that worked initially were not impressive. The maximum packet size being communicated at the moment is 1200 bytes.

6 Future Work

6.1 Threading in DPDK

DPDK offers users to write parallel code using its thread-safe libraries. The typical runtime environment of DPDK is a single-threaded process per logical core. However, it can be multi-threaded or multi-process in some cases. According to DPDK documentation, there are three threading models:

i. One EAL thread per physical core.
ii. Multiple EAL threads per physical core.
iii. Multiple light-weight threads per EAL thread.

In future work for this project, the developers can utilize threading to increase parallelism and throughput of the current system.

6.2 Packet Size in DPDK

The current maximum packet size is 1200 bytes. Further optimizations can be made to increase the packet size for the data transfer.

6.3 Segmentation of Packets

DPDK allows the developers to create segmented packets. This feature can be tested and utilized to increase the packet size efficiently.

6.4 Implementation of DPDK Driver for DAQPIPE

Due to unforeseen complications in the project, the DPDK driver for DAQPIPE could not be completely implemented in time. Therefore, the next step is to implement a complete API for the driver.

6.5 Exploring DPDK further

This project, as mentioned before, was a prospective project as no one in the LHCb group had worked with DPDK before. It leaves the team with a chance to further explore and exploit the features by DPDK.

7 Conclusion

This project was a prospective step by the LHCb team. The bandwidth results obtained so far (i.e. 32Gb/s approximately) signify that DPDK can be an interesting candidate for the LHCb upgrade in 2020. However, there is still a need for further exploration and optimization for the driver implementation in DAQPIPE. Given the specified hardware support, and feature such as threading and segmentation utilized, DPDK might be able to produce close to the desired throughput (80 Gb/s).

8 Acronyms and Abbreviations

#	Keyword	Meaning
1	API	Application Programming Interface
2	DAQ	Data Acquisition
3	DAQPIPE	DAQ Protocol-Independent Performance Evaluator
4	DPDK	Data Plane Development Kit
5	EAL	Environment Abstraction Layer
6	Gb/s	Gigabits per second
7	LHC	Large Hadron Collider
8	LHCb	Large Hadron Collider beauty
9	NIC	Network Interface Card
10	OS	Operating System
11	RX	Receiver
12	Tb/s	Terabits per second
13	TX	Transmitter

9 References

[1] Large Hadron Collider (LHC). https://home.cern/topics/large-hadron-collider

[2] LHCb Detector. https://lhcb-public.web.cern.ch/lhcb-public/en/detector/Detector-en.html

[3] DAQPIPE. https://gitlab.cern.ch/svalat/lhcb-daqpipe-v2

[4] DPDK. http://dpdk.org/

[5] Networks Supported by DAQPIPE. https://gitlab.cern.ch/svalat/lhcb-daqpipe-v2#select-your-transport-driver

[6] Kernel Bypass. https://blog.cloudflare.com/kernel-bypass/

[7] DPDK Sample Applications. http://dpdk.org/doc/api/examples.html

[8] NICs supported by DPDK. http://dpdk.org/doc/nics