Accelerating 5G Data Plane using DPDK: Design and Development of RAN & DNN Emulator

A MTP report

Submitted in partial fulfillment of requirements for the degree of
Master of Technology
Computer Science and Engineering

By

Nilesh Unhale
183050050

Guided by
Prof. Mythili Vutukuru

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
June 28, 2020

Contents

1 Introduction **5**
- 1.1 Problem statement . . . 5
  - 1.1.1 Goals . . . 5
- 1.2 Contribution . . . 6

2 Background **7**
- 2.1 5G Architecture . . . 7
- 2.2 DPDK . . . 8

3 Existing RAN Emulator **10**
- 3.1 Limitations of existing Kernel based RAN Emulator . . . 11

4 DPDK based RAN emulator **12**
- 4.1 Architecture . . . 12
  - 4.1.1 Handling Incoming Packets . . . 12
  - 4.1.2 Outgoing packets from RAN emulator . . . 13
- 4.2 Bridging DPDK Kernel gap in RAN emulator . . . 14
- 4.3 Bridging DPDK Kernel gap in UPF . . . 15
- 4.4 Storing packets for reducing overhead . . . 16
- 4.5 Emulating Multiple UEs . . . 17
- 4.6 Latency Computation . . . 18
- 4.7 Controlling Outgoing data traffic . . . 19
  - 4.7.1 Variable data traffic rate . . . 20
- 4.8 Maintaining packet count . . . 20
- 4.9 Challenges . . . 20
  - 4.9.1 Delivering existing control plane messages over DPDK . . . 20
  - 4.9.2 Scaling performance with multiple cores . . . 21
  - 4.9.3 Failing to register a large number of UEs . . . 21

5 DPDK based DNN Emulator **22**
- 5.1 Architecture . . . 22
- 5.2 Mirroring incoming packets . . . 24
- 5.3 Throughput visualization and retention . . . 24
- 5.4 Maintaining packet count . . . 24
- 5.5 Emulating multiple UEs . . . 24
- 5.6 Controlling outgoing data traffic . . . 25
- 5.7 Challenges . . . 25
  - 5.7.1 Packets dropped at high incoming rate . . . 25
  - 5.7.2 Echoing all packets at greater than 100% echo rate . . . 25

6 Evaluations **26**
6.1 Experimental Setup 26
6.2 Results 26
6.2.1 Comparision between different methods of creating packets . . 27
6.2.2 Throughput vs packet sizes 28
6.2.3 Single UE vs Multiple UEs throughput on single core 29
6.2.4 Scalability 30
6.2.5 Batch size vs Throughput 30
6.2.6 CPU cores vs packet size 31
6.3 Proof of Correctness 31
6.3.1 Variable Data traffic 31
6.4 Extra features 32
6.4.1 Real time throughput display 32
6.4.2 Throughput logging periodically 33
6.4.3 Maintaining packet counts 33

7 Future Work **34**

8 Conclusions **35**

List of Figures

2.1 5G Architecture 7
2.2 DPDK architecture 8

3.1 5G Data Plane packet flow 10
3.2 Existing RAN Emulator Implementation 10
3.3 Code profiling 11

4.1 DPDK RAN emulator rx architecture 12
4.2 DPDK RAN emulator tx architecture 13
4.3 Bridging DPDK Kernel gap in RAN emulator 14
4.4 Packet delivery to socket program via TUN with DPDK 15
4.5 Control plane communication over DPDK in UPF 16
4.6 Multiplexing multiple UEs on single core 17
4.7 Latency packet flow 18
4.8 Controlling Outgoing data traffic 19

5.1 DPDK DNN Emulator rx architecture 22
5.2 DPDK DNN Emulator tx architecture 23

6.1 Server setup 26
6.2 Throughput of various packet size on 10G NIC 28
6.3 Throughput of various packet size on 40G NIC 28
6.4 Single UE vs Multiple UEs throughput on 10G NIC 29
6.5 Scalability across CPU cores on 40G NIC 30
6.6 Batch size vs Throughput on 10G NIC 30
6.7 Variable data transmit rate with time 31
6.8 Time series graph showing rate limiting 32
6.9 Throughput is displayed per second 32
6.10 Throughput is logged per second 33
6.11 Maintaining per UE packet count 33

List of Abbreviations

3GPP	3rd Generation Partnership Project
UPF	User Plane Function
AMF	Access and Mobility Management Function
SMF	Session Management Function
DNN	Data Network Name
UE	User Equipment
RAN	(Radio) Access Network
NRF	Network Repository Function
DPDK	Data Plane Development Kit
AUSF	Authentication Server Function
SUCI	Subscription Concealed Identifier
GTP	GPRS (General Packet Radio Service) Tunneling Protocol
PFCP	Packet Forwarding Control Protocol
NF	Network Function
ARP	Address Resolution Protocol

Chapter 1

Introduction

The number of smartphones and IoT devices have been increasing a lot. There is a need for the network technology which provides reliable, super high speed, responsive network. 5G is the fifth generation cellular network technology, which has many optimizations as compared to its predecessor 4G. 5G supports network slicing, separation of control plane and user plane, and many more. At IIT Bombay, there is a team developing 5G testbed and exploring various design options while implementing 3GPP specs [1].

1.1 Problem statement

UPF deals with data plane packets from UE. All traffic coming from UE is encapsulated by RAN and forwarded to the UPF. UPF decapsulates packets and forwards it to the DNN. RAN is a physical tower which is medium though which UEs communicate with the 5G core NFs. For testing purposes RAN emulator was developed, which also emulated control and data plane traffic form UE(s). The implementation of the existing RAN emulator used TUN for encapsulating data packets from UE in GTP. The kernel network stack is not able to keep up with high-speed network traffic, especially for small packets. Moreover, the TUN device performance deteriorates as the number of per-packet copies is increased, and context switching between userspace and kernel space also increased.

The primary focus of the project was to develop a RAN Emulator, which was capable of producing data traffic of variable payload sizes at line rate (10Gbps, 40Gbps).

1.1.1 Goals

- Generate data traffic at line rate (10Gbps) to test UPFs based on modern fast packet processing techniques.
- Making it easier to control data traffic generated and it's duration.
- Taking experiments readings should be easier.
- As traffic outgoing from UPF goes to DNN, it should be capable of handling such large traffic and also to generate the same.

- Existing control plane (CP) Network Functions (NFs), which were built to work on kernel interface, need not be reprogrammed to work on DPDK. How to handle incoming and outgoing CP traffic via DPDK application?
- Session establishment of a large set of User Equipments (UEs) should be faster to test UPF's performance when a large number of UEs' sessions need to be setup.

1.2 Contribution

In this project, the 5G project team has developed a RAN Emulator for testing the other NF's working. The data plane in the existing RAN Emulator was implemented on TUN device and was bottlenecked at 1.8 Gbps. Now, the data plane in RAN Emulator is optimized and is built on a kernel network bypass mechanism, data plane development kit (DPDK). The optimizations in RAN Emulator lead to an increase in the performance, and now it can saturate 10G and 40G NIC for any packet size. The DPDK based RAN Emulator is now capable of handling control plane packets without modifying the control plane NFs. The application now supports ARP protocol and emulating data traffic from multiple UEs. The rate of generated traffic can be controlled and varied with respect to time for testing the Quality of Service (QoS) capabilities of the UPF. Moreover, DPDK based data network name (DNN) is also developed to handle the high bandwidth traffic from UPF and can also generate the same if needed. Data collection from the RAN and DNN Emulator is effortless as it is logged per second and also displayed as per requirement.

This report is organized as follows. Chapter 2 gives a brief overview of 5G architecture and the working of DPDK. Chapter 3 discusses the implementation and bottlenecks of the existing RAN Emulator. Chapter 4 deals with the challenges and implementation of RAN over DPDK. Chapter 5 includes the architecture and features of DNN over DPDK. Chapter 6 talks about experimental evaluations. Chapter 7 shows the future work on optimizing DPDK based applications. Chapter 8 gives the conclusion of this report.

Chapter 2

Background

2.1 5G Architecture

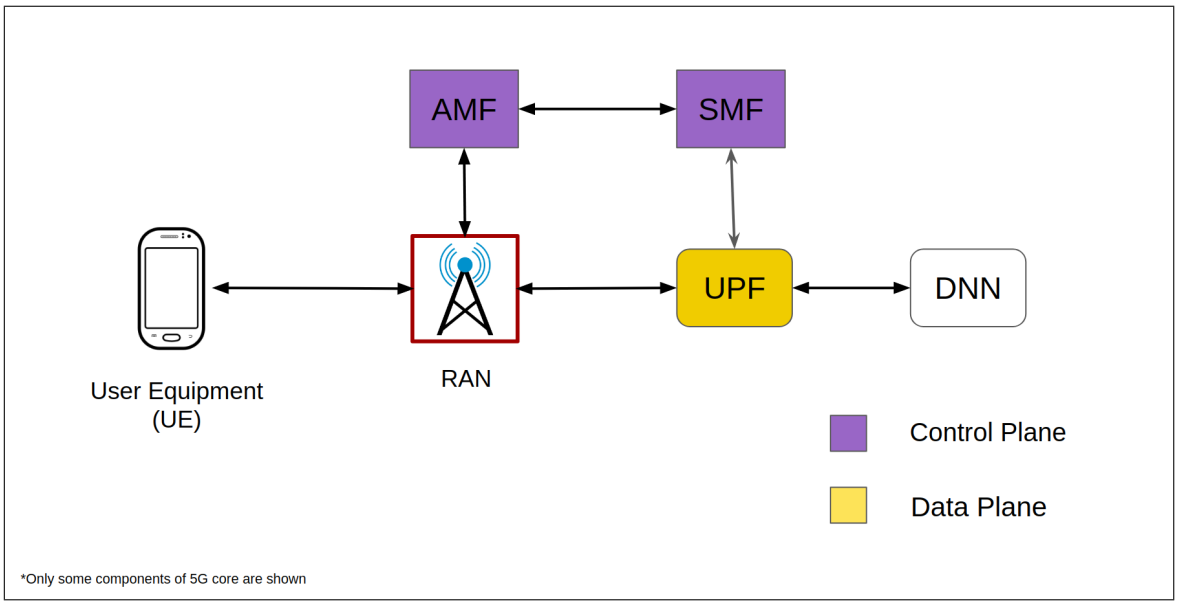

Figure 2.1: 5G Architecture

Figure 2.1 shows the crucial components in the 5G architecture [1, 2]. User Equipment (UE) can be any mobile device or IoT device accessing 5G. Data Network name (DNN) is the destination that UE wants to send data packets. Access and Mobility Management Function (AMF) deals with the registration of the UE. Session Management Function (SMF) is responsible for managing a session between UPF and UE. SMF install rules in the User Plane Function (UPF). UPF using these rules decides whether to forward or drop packets received from UE.

Whenever a UE wants to send or receive data from the internet, it must register itself into the 5G core. AMF deals with the starting registration and authentication requests. Before connecting to the internet, the UE should establish a PDU session. During the PDU session establishment, the SMF allocates an IP to the UE and gives the IP of compatible UPF. SMF also installs rules on the UPF side according to the UE's context. All data packets from UE are encapsulated in GTP by RAN and are forwarded to UPF. UPF checks the packet, and according to the rules, forward it to

DNN or UPF or drop it. When UPF receives a downlink data packet, it also sends to RAN after GTP encapsulation.

2.2 DPDK

Typical kernel I/O stack working involves copying of packet, context switching, etc. DPDK was proposed by intel and was previously compatible with intel NICs only. DPDK (Data Plane Development Kit) is a collection of libraries for fast packet I/O. DPDK [3] avoids context switching and copying of data by binding NIC to the user-space driver. This bypasses the kernel altogether, removing any overheads and meta-data maintained by the kernel.

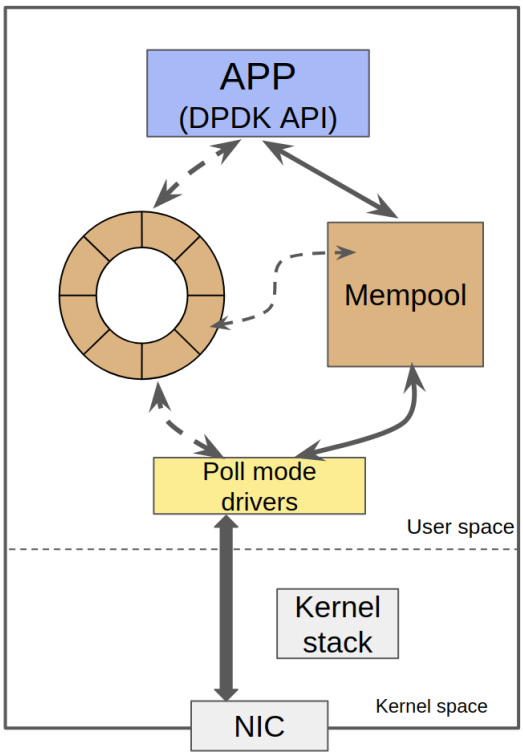

Figure 2.2: DPDK architecture

When DPDK [3] initializes it gains access to NIC and memory by Environment Abstraction Layer (EAL), this enables DPDK [3] to access all the hardware entities. DPDK [3] has a memory manager that manages free region and offsets in the memory pool created with the help of huge pages. The traditional kernel does per-packet allocation and deallocation of buffers. DPDK [3] has a buffer manager that preallocates buffers in bulk, reducing the cost of dynamic allocation of memory. For avoiding lock contention, the buffer manager maintains per-core cache buffers. All the control information related to the packet is stored in a hash table for faster access. DPDK [3] has a queue manager which takes care of all lock-less data structures used. The user-space drivers are called as Poll Mode Drivers (PMD), which are responsible for continuously polling the NIC instead of interrupt-driven mode. User Application talks to the user libraries of DPDK [3] instead of making a system call. Hence, DPDK [3] can be referred as true kernel bypass technique see figure 2.2.

For more efficient working, DPDK [3] uses huge pages for less TLB misses. Also, all the shared data structures are used in a lock-less manner to avoid lock contention. DPDK [3] uses all hardware knowledge to boost the packet I/O, it takes into account NUMA nodes, the number of CPU cores available, size of huge pages. DPDK [3] also handles packets in batches and uses RSS (Receive Side Scaling) to deliver packets to correct core.

As shown in figure 2.2, poll mode drivers keep polling the NIC, and whenever a packet comes, it is copied into the DPDK Mempool via DMA and a corresponding packet descriptor is put in a ring of pointers. The application knows that packet has come via DPDK libraries and application can read packet directly after getting its location with the help of packet descriptor.

Some important DPDK data structures and libraries –

1. **Ring Manager (rte_ring):** Rings are the FIFO queues with a limited number of capacity. They can store a pointer of the objects. Rings can be configured to work with multi-consumer and multiproducer, which makes them thread-safe. Rings have a lockless implementation, which makes them perfect for inter-core transfer of packets.

2. **Memory Management (rte_mempool):** Hugepages which are allocated to DPDk application are merged into segments and then divided into what is called as zones [4]. These zones are used to store the objects created by the application, one of which is memory pools (mempools). Mempools are created by rte_mempool library, which uses rte_ring to store the objects. These are lockless data structures and can be accessed using CAS (Compare and set operation). The performance can be increased by using memory alignment techniques.

3. **Buffer Management (rte_mbuf):** Linux kernel uses sk_buff [5] data structure for storing the incoming packets, but instead of one large sk_buff DPDK uses smaller rte_mbuf. These buffers (rte_mbuf) are smaller in size and are allocated at the beginning of the program and stored in the rte_mempool [4]. When packets are received, they are stored in the rte_mbuf, which also contain metadata about the packet. Metadata is typically packet length, data length, address of next buffer, headroom offset, tailroom offset, etc. Some metadata is left blank intentionally for user's custom usage, for example, to store the timestamp.

4. **Hash library (rte_hash):** Hash library is provided by DPDK for very fast lookups in hash tables. The primary key is unique for each entry, and the value stored is the address of the object. The key in the hash table needs to be of the same size (bytes) for faster performance and which can be created by using the hash functions provided by DPDK.

5. **Flow library (rte_flow):** This library may not be compatible with all the NICs. This library can be used to configure NIC to classify the incoming packets into different flows according to the rule provided. This rule can be from ARP packet identification to decapsulating GTP header according to requirement. This library can be used to classify packets into different rx queus, dropping invalid packets, sending ARP replies, etc. This library has a lot of useful functions but no known implementation, all the features also compatible devices and drivers needed for it to work properly.

Chapter 3

Existing RAN Emulator

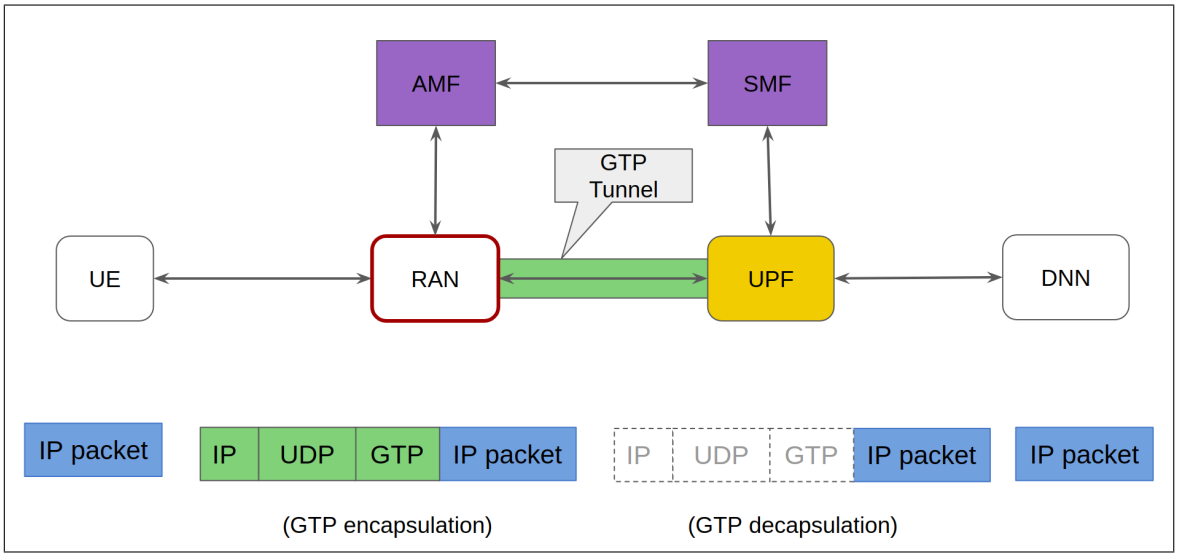

Figure 3.1: 5G Data Plane packet flow

After registration of UE with the AMF, UE needs to establish a PDU session with SMF. After the PDU session establishment, RAN has UE IP, UPF IP, TEID. Also, UPF gets rules from SMF regarding how to handle packets from a particular UE. During the PDU session establishment, TEID between UPF and RAN exchanged, which used for GTP tunneling through which data plane packets travel.

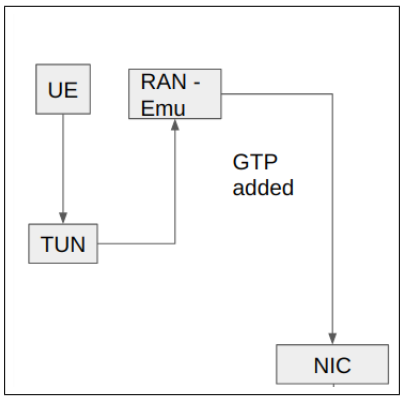

Figure 3.2: Existing RAN Emulator Implementation

Implementation of existing RAN Emulator is as follows –

For UE mobility across the radio towers, GTP tunneling is used to send and receive data to and from the UPF. For a packet to be encapsulated in GTP, it should be an IP packet. To achieve this, existing RAN installed a route to divert packet from interface to a TUN device. A TUN device is a virtual interface from an L3 packet that can be read or write. RAN Emulator then reads the L3 packet from TUN device then adds GTP header and sends it to the UPF.

3.1 Limitations of existing Kernel based RAN Emulator

When packet size is small, the number of packets that can enter a system with the same NIC becomes larger. In Linux, an interrupt is generated per packet. As the number of incoming packets increases, the interrupts also increase, and as a result, most of the time is wasted in kernel space handling interrupts.

As shown in figure 3.2, the packet generated in userspace goes via kernel network stack space up to IP header. This packet is captured via TUN device to the userspace to add GTP header and then is again sent via kernel stack. This design leads to multiple copies of a packet in userspace and kernel space and which increases the number of context switches per packet.

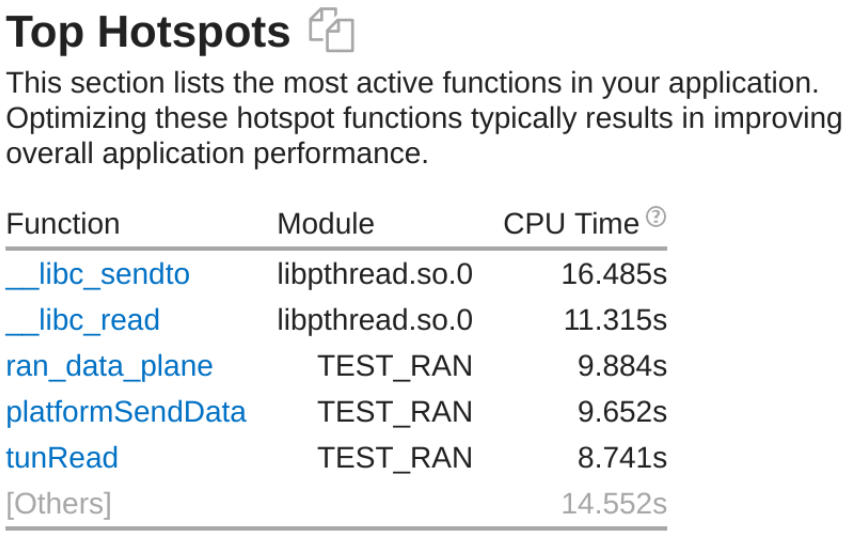

Figure 3.3: Code profiling

Code profiling if RAN Emulator was done using Intel Vtune as shown in figure 3.3. Code profiling shows that most of the time (greater than 39%) is spend on reading from TUN (__libc_read) and sending encapsulated packets from sockets (__libc_read). Due to above all the reasons the RAN Emulator was capable of generating 1.8Gbps for a payload size of 1422B.

Chapter 4

DPDK based RAN emulator

4.1 Architecture

The following subsections explain RAN emulator architecture when packets are being received or transmitted.

4.1.1 Handling Incoming Packets

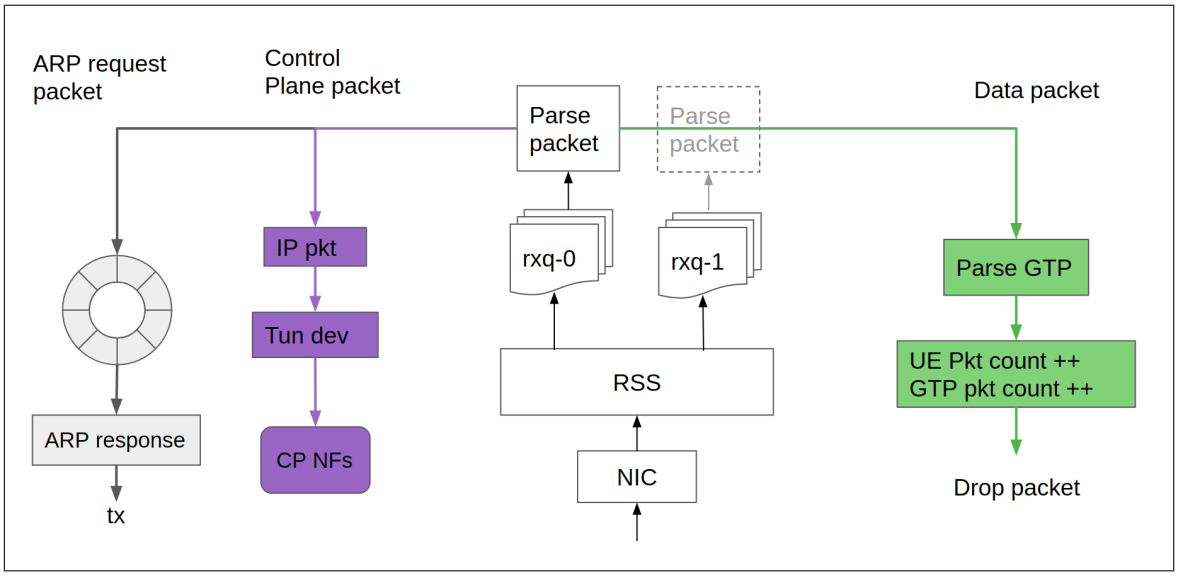

Figure 4.1: DPDK RAN emulator rx architecture

Figure 4.1 shows that all incoming packets go through hardware Receive side scaling (RSS). RSS hashes the incoming packets to the different rx queues. RSS takes into incoming packet's source IP, destination IP, source port, and destination port. As most of these fields are the same between UPF and RAN emulator, RSS hashes on the source port. Earlier designs of the RAN emulator did not have RSS but was using only one rx queue. As shown in figure 4.1 once a packet enters in rx queue it is parsed by CPU core and transferred to DPDK ring for further processing by other CPU cores. As the same core is receiving and parsing packets from only one queue, some packets are dropped when the incoming traffic is very high. In order to make receiving scalable with respect to the CPU cores, RSS is introduced. The incoming packets are now separated into different queues at the hardware level, and different CPU cores can now parse the packets parallelly.

During the parsing of each packet first, it is checked if it is an Address Resolution Protocol (ARP) packet. All the ARP request packets are enqueued to ARP DPDK ring. A DPDK ring is an efficient FIFO queue which can store pointers of objects. DPDK ring is also thread safe and has a lockless implementation. ARP request and reply threads are running which dequeue the ARP request packets. An appropriate response is generated if the ARP request is for the RAN IP and sent through the NIC, as shown in the figure 4.1. Any unsolicited ARP reply or invalid ARP reply/request is dropped.

If the received packet is not an ARP packet, then it is checked whether it is a control plane (CP) packet or not. Control Plane packets are typically HTTPs packets, TCP packets or UDP packets for session establishment. Such control plane packets are meant either for other control plane network functions (such as NRF, SMF, AMF) or for the control plane sockets, which are listening in the RAN emulator. As the NIC bound to the DPDK driver module is not accessible to the kernel, the packet is written on a tun device so that the respective CP NFs can receive them. This part is explained in detail in section 4.2.

The received packet is checked, whether it is a data plane packet. All data plane (DP) packet have GTP header, which can be easily identified by UDP destination port as 2152. All data packets are counted to cross-check with UPF's sent data. RAN emulator also provides a way to display per UE data plane packet count to check the correctness QoS enforced UPF. If the flag is enabled per UE packet count, each packet is further parsed to look at the UE IP and increment the respective packet count, and then the packet is dropped. If per UE packet count flag is not on, the packets are directly dropped, reducing packet parsing overhead.

4.1.2 Outgoing packets from RAN emulator

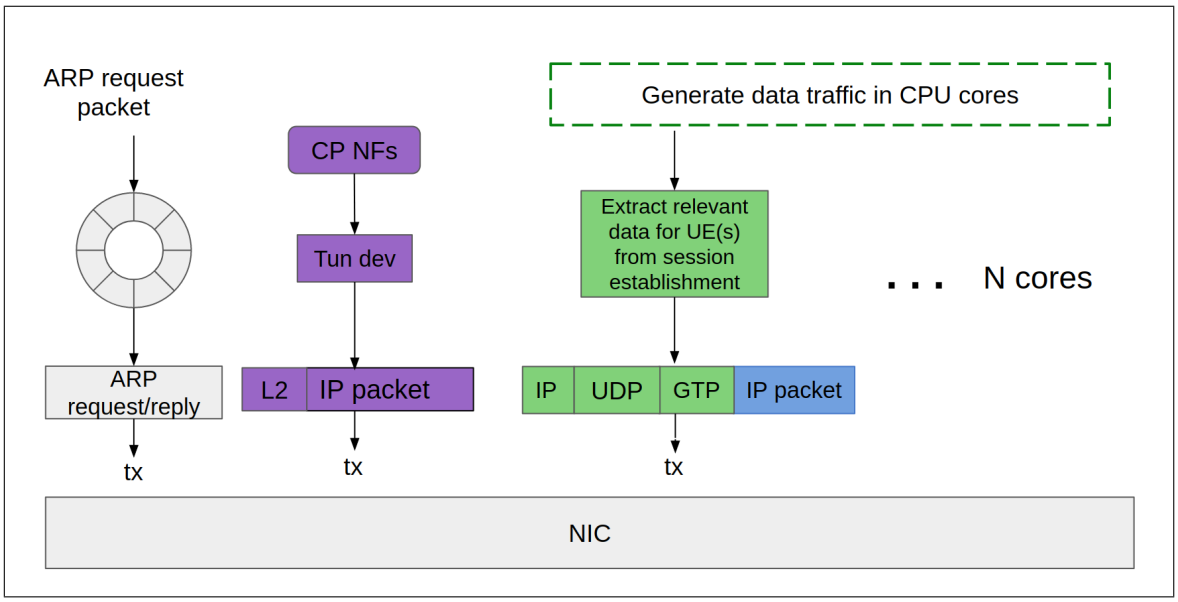

Figure 4.2: DPDK RAN emulator tx architecture

As DPDK works ethernet frames, the RAN emulator needs media access control (MAC) to form L2 packets. When the application starts, first it tries to look for UPF's mac address. This is achieved by sending out ARP requests for UPF's IP

periodically until a valid response is not received. Thus, just after the RAN emulator is started, only packets that DPDK is sending are ARP request packets.

As mentioned earlier, the kernel has no access to the DPDK driver bound NIC; tun device is used for bridging the gap between kernel and DPDK. When any application writes on the socket created on the tun device, an IP packet is captured from the tun device in the RAN emulator, and an appropriate L2 header is attached to it and sent via DPDK tx. The working of the tun device and packet flow is explained in detail in the section 4.2 .

Other than the above-mentioned scenarios, the RAN emulator transmits packets when traffic is to be generated at a given configuration. Traffic generated contains UDP packets structured as ethernet header + outer IP header + outer UDP header + GTP header + inner IP header + inner UDP header + payload as shown in figure 4.2. Configuration parameter for generating the traffic contains the number of cores to be used by the application, data traffic from a particular number of UEs, packets per UE, Number of seconds to send the traffic. For generating a correct data plane packet already session, establishment information is looked for information like UE IP, TEID for GTP header, UPF IP.

4.2 Bridging DPDK Kernel gap in RAN emulator

Following figure 4.3 shows the control plane path in purple color and accelerated data path in green color.

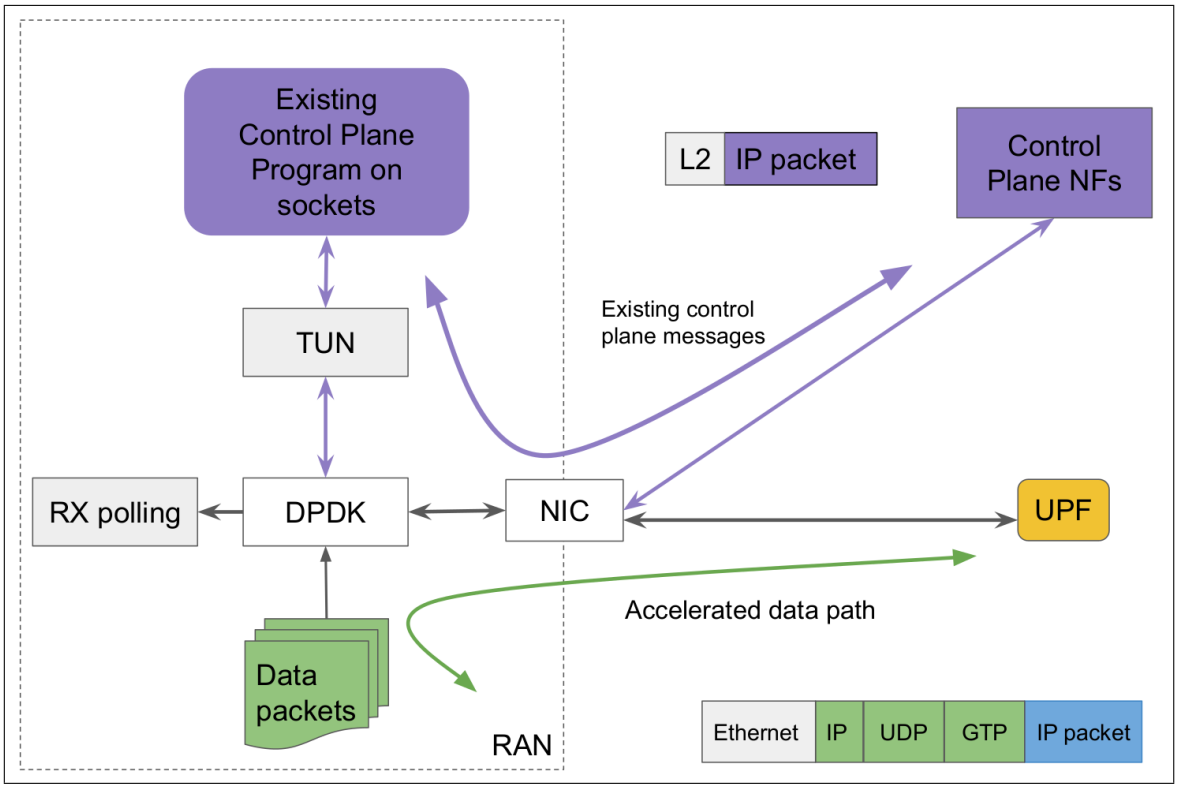

Figure 4.3: Bridging DPDK Kernel gap in RAN emulator

In this architecture, for optimum performance, no two DPDK send and receive functions are on the same core. One core is dedicated to polling for incoming packets. Another core handles all control plane messages via tun device. The data packets that

need to be sent are made once per UE and stored per UE per session. The packets are sent in the batches for higher performance. To further increase the throughput, the packets are sent via multicore via different tx queues.

As DPDK needs custom NIC drivers for working, the Linux kernel can not access the NIC. As a result, IP can not be assigned to NIC, and all the header parsing needs to be done in userspace. Handling UDP is straightforward as it is not a connection-oriented protocol, but it is not the case with TCP. Some control plane messages use TCP as the transport protocol. In order to handle such packets, they need to be directed towards kernel to parse them and handover payload to existing socket-based programs of RAN emulator.

Diverting packets towards kernel is achieved by creating a TUN device. The packet flow is shown in figure 4.4.

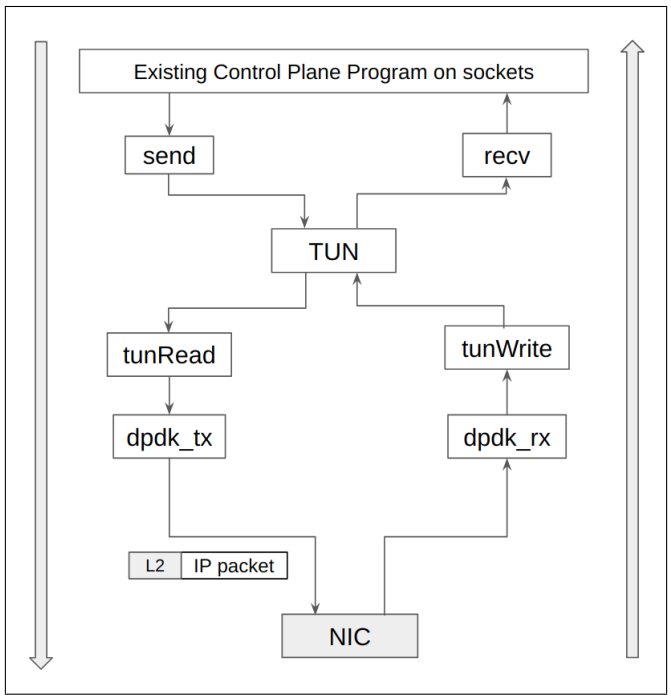

Figure 4.4: Packet delivery to socket program via TUN with DPDK

- **Incoming messages:** DPDK RX is polling on NIC to receive incoming packets. Whenever a control plane packet comes, it is checked based on IP protocol and port, whether it is a control plane packet or not. Once confirmed, it is diverted to TUN by discarding L2 headers and writing the remaining packet on TUN fd. Existing control plane programs bind to sockets, which are listening on a port get the packet via TUN interface.

- **Outgoing messages:** When the existing program on the socket sends a packet, it turns up on the TUN device. A corresponding event is triggered on the epoll of TUN fd, after that packet is captured from the TUN device using TUN read. The packet acquired has up to L3 headers. Then L2 header is added to the packet and is sent out of the NIC via DPDK TX.

4.3 Bridging DPDK Kernel gap in UPF

Following figure 4.5 shows the control plane in purple and data plane in yellow.

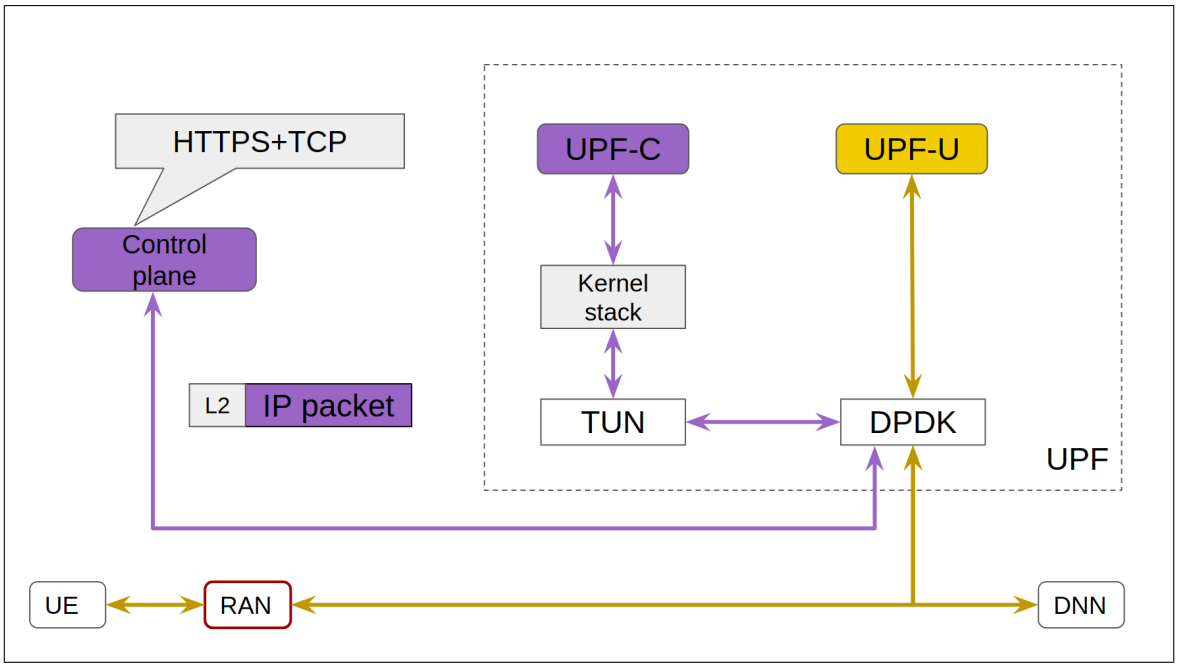

Figure 4.5: Control plane communication over DPDK in UPF

In this architecture, when UPF wants to send control plane messages, especially for NRF registration, those messages are over TCP. As DPDK drivers are bound to NIC so standard kernel sockets cant send data via NIC. Also, implementation whole TCP stack over DPDK is not worth the effort as the control plane messages are very infrequent.

The existing control plane socket programs are bound to a virtual TUN deice. Outgoing messages are processed by kernel till layer 3, then the packet is read and added ethernet and is sent via DPDK TX. Similarly, for incoming messages, the packet is received by DPDK, then the ethernet header is discarded, and the entire remaining packet is written to TUN. The socket programs then can get packet via socket receive function.

4.4 Storing packets for reducing overhead

Creating a data plane packet with inner IP, inner UDP, GTP, outer IP, outer UDP, Ethernet Address requires multiple *memcpys*, which hinders the performance of RAN emulator. Moreover, while creating the UDP header, UDP checksum needs to be calculated. As UDP checksum is calculated over the payload, having a larger payload size requires more checksum computation time. UDP checksum calculation is CPU intensive and needs to be calculated twice per packet.

For increasing performance, a packet is only created once per UE and stored as a byte array. This way avoids the UDP checksum calculation, as well as extra *memcpys* are reduced. Along with this, packets are created and sent in batches to increase performance further. Now for creating a packet, only one *memcpy* is sufficient. This is also shown in table 6.2

4.5 Emulating Multiple UEs

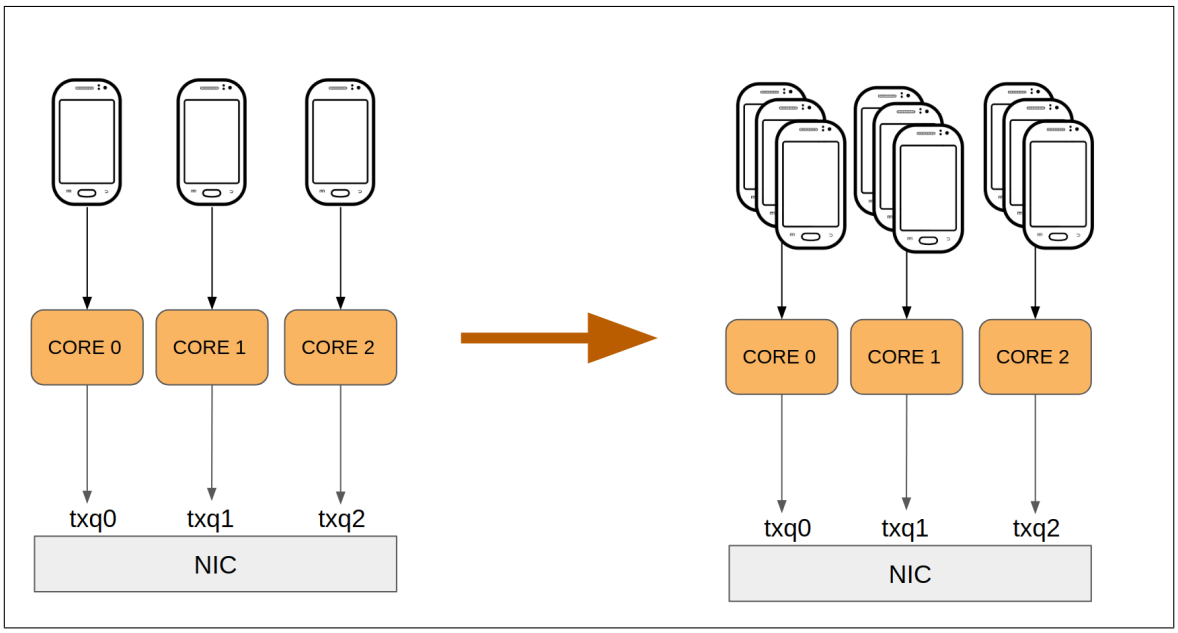

Figure 4.6: Multiplexing multiple UEs on single core

Existing RAN emulator was able to send a packet from a single UE. To test the UPF with multiple UEs and how does it perform with having multiple rules RAN emulator needs to send data plane traffic with multiple UEs.

Data plane packets are dependent on the established session details of the UE and identified by the session ID. Different UEs may have the same session IDs. Also, one UE can have multiple sessions. As packets are stored to improve performance, now a packet needs to be stored per UE and per session. For this, an unordered_map is maintained having key and structure as pktInfo as follows –

```
string key = UEsuci + '#' + sessionID;

typedef struct pktInfo {
    int pktlen=0;
    char pktData[MAX_MESSAGE_SIZE]={};
}pktInfo_t;
```

As shown in figure 4.6, the left side design was earlier implemented, and a map was maintained across all cores. The requirement was to send the data traffic from multiple UEs simultaneously, so each core was emulating a UE. This method perfectly but for the number of UEs was limited by the number of cores.

Sending packets from a large number of UEs greater than number cores at a particular instant is theoretically impossible. However, as the packets generated from RAN emulator range from 0.8 Mpps to 7.5 Mpps according to the packet size, data traffic from multiple UEs can be multiplexed on a single CPU core. This will result in the data traffic from multiple UEs over a particular duration of time.

This was implemented in the following manner. For each UE, a data packet is created beforehand and stored in a universal unordered map. When there is a need to send data from multiple UEs, various configuration parameters are taken from user like how many UEs to emulate per core and how frequently the UEs should

be multiplexed. Once the number of UEs per core is confirmed, an array is created per core to store pre-made packets for all UEs allotted to that core. Using an array reduces the lookup time and can be used to switch to other UEs frequently. The multiplexing is achieved by sending data one batch per UE in a round-robin manner, as shown in figure 4.6.

4.6 Latency Computation

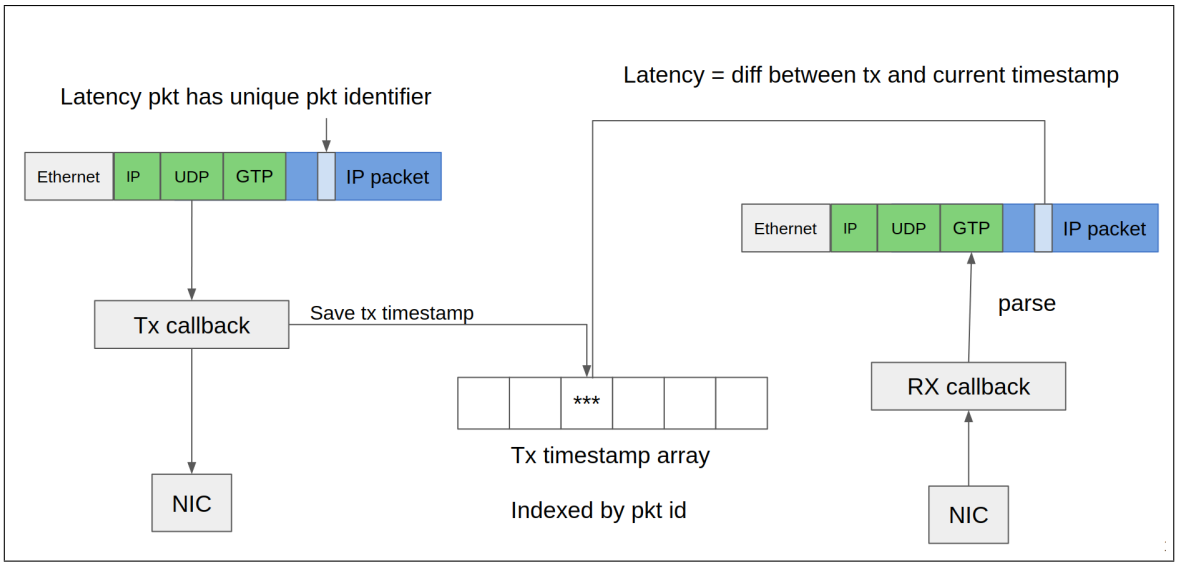

Figure 4.7: Latency packet flow

The end to end latency computation can be slightly tricky as the packet header 4.7 up to GTP header are parsed by UPF and discarded, and IP packet is forwarded to the DNN. During the downlink data packet is prepended with new (GTP+UDP+IP) headers. Due to this, any information that needs to be stored in the packet for latency computation should be inside the inner IP packet.

An earlier design of latency computation was very simple and did not account for any packet dropped by UPF or DNN. It sent a packet with a unique inner IP packet identifier and waited for it to return from DNN. The difference between *rx* and *tx* timestamp was computed and averaged out in the end. However, if somehow this packet got lost in the middle, the time difference would significantly increase, and also rest of the latency packets were not send due to the blocking nature of latency function.

To overcome above problem, different designs were taken into consideration. One design was to embed tx timestamp into the optional field of the inner IP header and, during its return packet, was parsed to retrieve the information. Some problem with this design was that adding fields and retrieving information from the optional field is cumbersome; moreover, checksum needs to be recalculated for inner and outer packets. Also, in this approach, if latency and data packets are simultaneously sent, there would be a difference in packet sizes.

Current design 4.7 has a *tx* timestamp array that stores timestamp whenever a latency packet leaves the NIC. Latency packets have a range of different inner packet identifier, same as the size of tx timestamp array. A latency packet is created with a new packet identifier and sent to UPF before leaving the NIC; tx callback

function records the timestamp against the packet ID in the array. For the latency function to work properly, DNN should be mirroring 100% of the incoming packets. When the packet returns to RAN emulator, rx callback retrieves the packet identifier and computes the difference between current timestamp and recorded tx timestamp to compute latency. If a packet is lost in the middle of the trip to DNN, latency function will wait until the next 2^{16} latency packet before discarding old timestamp corresponding to that id and will send a new packet of that packet ID.

4.7 Controlling Outgoing data traffic

DPDK based RAN emulator was designed to pump as much as traffic to the UPF check what amount of traffic it can handle. As the UPF was equipped with Quality of Service (QoS) functionalities, it was necessary to test UPF for a particular incoming rate. RAN emulator, by default, sends packets at the maximum rate possible through a given number of cores. The control over the outgoing data traffic was achieved via two ways:-

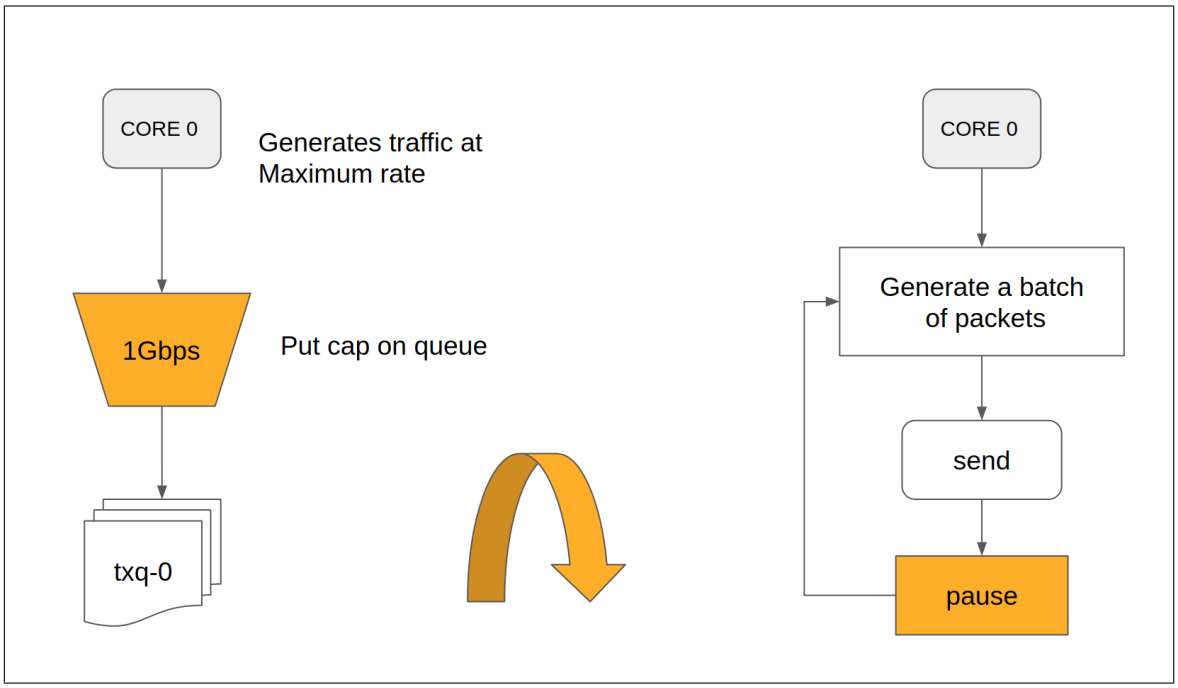

Figure 4.8: Controlling Outgoing data traffic

- DPDK provides traffic management APIs for Quality of Service (QoS) traffic management of ethernet devices. One of them is rte_eth_set_queue_rate_limit() [6], which can be used to set rate limitation for a particular queue on an ethernet device. The main problem with this is that it requires supports from NIC and compatible drivers. This API considers the size of the packet with all headers, also including the ethernet header for rate limiting. This function takes queue and particular rate (in Mbps) and sends only that much packets through the given queue; the rest of the packets are dropped as shown in left figure in 4.8.
- Unfortunately, rte_eth_set_queue_rate_limit() is not supported on Netronome's smartNIC. To control rate on the smartNIC only way was to generate less traffic to send in the first place. This was achieved by adding a delay in between transmitting two consecutive batches s shown in right figure in 4.8. The

delay is configurable to the granularity of microsecond by using DPDK API rte_delay_us() [7].

4.7.1 Variable data traffic rate

In order to emulate real-life fluctuating traffic for testing, QoS enforced UPF, variable data traffic with respect to time was needed.
This was achieved in the norma RAN emulator by changing the transmit rate at particular CPU ticks. Time to switch the rate and by how much granularity can be configured by the user. The time given by the user is converted to CPU tick so that the DPDK APIs for timer can be used to avoid the overhead of reading time by traditional system calls.
Unfortunately, this method does not work on Netronome's smartNIC, as the API is not at all supported. In the RAN emulator for smartNIC, an extra thread is spawned, which changes the inter batch delay according to the given parameters. The parameters in the orange colour shown in figure 4.8 are changed periodically to achieve the desirable variable traffic example: figure 6.7.

4.8 Maintaining packet count

If the flag for packet per count is enabled, the inner IP packet is parsed, and the corresponding packet count of the UE IP is increased. Earlier, this was implemented by using an unordered map. Unordered map look times are comparatively very high to the incoming traffic rate, resulting in a decrease in receiving performance and leading to NIC dropping incoming packets. DPDK hash was used to compensate this issue, resulting in lower hash lookup times and better performance.

4.9 Challenges

4.9.1 Delivering existing control plane messages over DPDK

- **KNI module:** The DPDK Kernel NIC Interface (KNI)[8] allows userspace applications access to the Linux control plane. There are some advantages of using KNI module than existing TUN/TAP interfaces like a reduced number of system calls. The kernel can see the DPDK interface in ifconfig and can be managed by tools like ethtool. Moreover, applications running over sockets can use user kernel network stack via this interface. The reasons for dropping the idea of KNI, is that it requires an additional kernel module, for a virtual interface to become "up" another DPDK based application needs to be running in the background, and only one DPDK application can run at a time.

- **Dtap / Dtun:** DPDK libraries support DPDK TUN (Dtun) and DPDK TAP (Dtap)[9] virtual interfaces. These interfaces are compatible with DPDK APIs and are optimized for fast packet IO. But unfortunately, none of the socket programs were able to send or receive packets via Dtap/Dtun. The exact reason for the failure of this design is unclear.

- **TUN device:** The existing TUN interface is used in the final design, as shown in figure 4.4. The tun is created with an IOCTL call, and a perfectly

made packet with correct checksum is written to the TUN device. For sending packet out of the NIC, a packet is read from the TUN and L2 header is added according to destination IP and is sent out of the NIC via DPDK as shown in figure 4.4

4.9.2 Scaling performance with multiple cores

- DPDK uses poll mode drivers to keep polling on NIC for incoming packets. Polling avoids the overhead of interrupt handling per packet. When multiple threads are spawned on the same core as that of DPDK polling thread, DPDK performance suffers. All the packets which come at the time when DPDK polling thread is not scheduled are dropped at NIC. To overcome this issue, DPDK RX polling thread is spawned and pinned to a separate CPU core. Same issue can be seen when DPDK tx is scheduled with another thread on a core. As DPDK tx does not get enough CPU time, it sends fewer packets than its potential.
  Another issue comes when different DPDK tx from different cores try to write to the same tx queue of NIC. Most of the packets are dropped in such a situation, to avoid this, different DPDK tx should write on different tx queues.

4.9.3 Failing to register a large number of UEs

- The DPDK application tries to detect all available cores and spawns a slave-thread on them. However, When the application was giving all the available cores, including those which were on another NUMA node, the application was crashing as the slave-threads were interfering with the shared memory of the RAN emulator. The main culprit, for this issue, was not yet found, but for completing the rest of the experiments, this issue was temporarily avoided by running the application on a single NUMA node.

- After handling the above issue, the application was no able to register a large number of UEs in less amount of time. The reason behind this was as UDM tries to connect with redis-server frequently, the previous connections are still in TIME_WAIT state. To get around this situation:-

```
#echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
```

Chapter 5

DPDK based DNN Emulator

5.1 Architecture

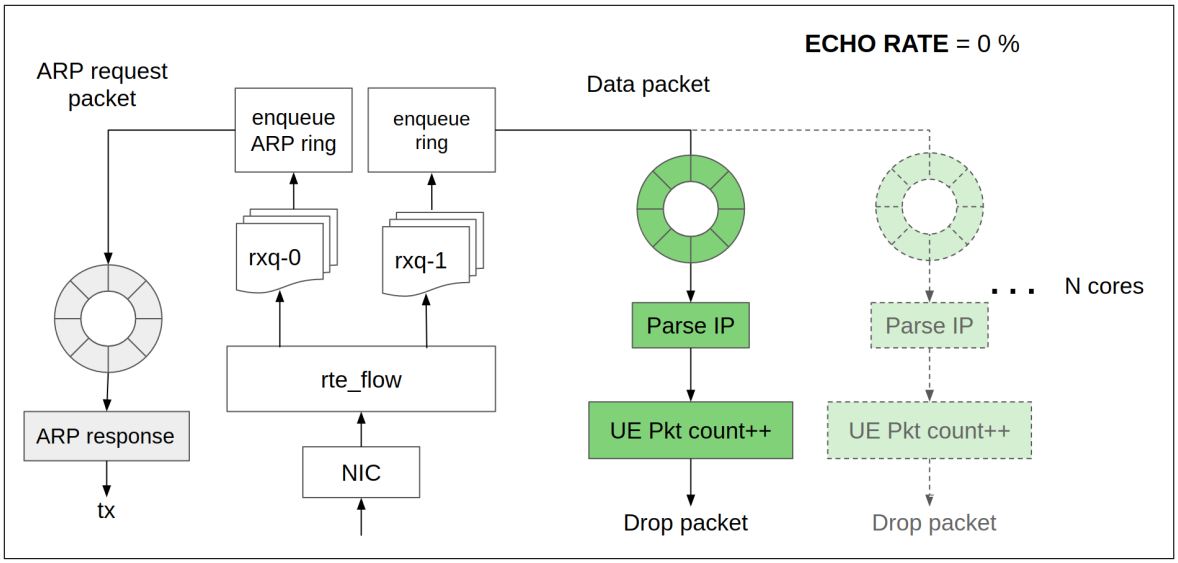

Figure 5.1: DPDK DNN Emulator rx architecture

As the figure 5.1 shows, for bifurcating, the incoming packets rte_flow has been used. As DNN has no control packets what so ever, there are only two types of packets that come, i.e., ARP packets and DP packets. In the earlier designs, the parsing was performed in the userspace. DNN was able to handle most of the incoming traffic rate with that design. Only a small chunk of packets was dropped in this design. In userspace, parsing of packets was done to only identify and separate ARP and DP packets. This processing can also be entirely shifted to hardware with the help of DPDK rte_flow [10]. All the ARP packets are now queued to rx queue 0 and rest packets to rx queue 1.

ARP request packets are handled by already running ARP request handle thread. If the received ARP packet is valid and meant for DNN, the thread will form an appropriate ARP response and sent it out.

DNN can be used as a sink to receive all data packets and drop them or can be configured to echo incoming data packets with a particular percentage. This can be configured by giving echo percentage as a parameter and giving enough CPU cores to handle the incoming traffic. Figure 5.1 shows when DP packet flow when DNN echo percentage is set to 0. As soon as the DP packets are received on the rx queue

0 all of them are enqueued to the DPDK ring (lockless FIFO queue). The number of rings is the same as the number of core configured to handle incoming traffic. The enqueuing of the packets are in round-robin fashion so that all CPU cores have work to do. The handle packet cores dequeue from their respective ring and parse the DP packets. As the echo rate is 0, only DP packet-count is increased, and packets are dropped. If the per UE count flag is set, then packet IP addresses are parsed, and the respective packet count of UE is increased.

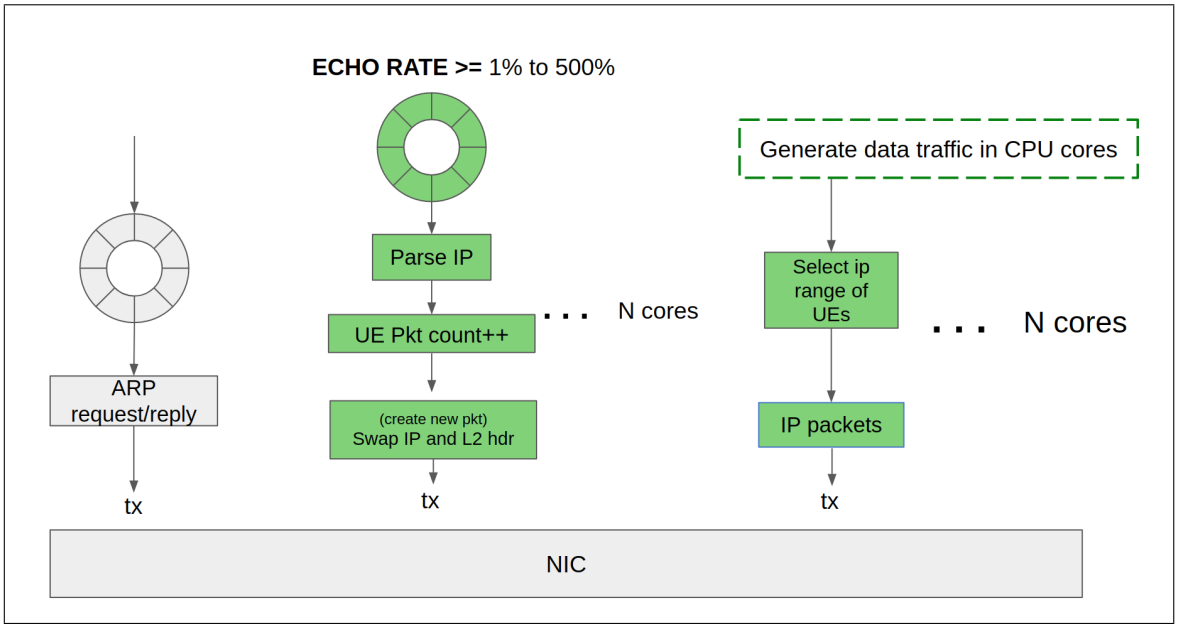

Figure 5.2: DPDK DNN Emulator tx architecture

As shown in figure 5.2 DPDK based Data Network Name (DNN) Emulator sends out packets in the following cases.

DPDK DNN needs UPF's MAC address in order to send the data packets. Similar to RAN Emulator, DNN sends ARP request packets periodically to the UPF until the valid ARP response is received.

When DNN is configured to run on echo percentage > 0, the given percentage of packets needs to send back. When the percentage is given is less than 100, for accurate mirroring of packets handle packet function running on each core waits for 100 packets to be enqueued in their respective ring. Only 100 packets are dequeued from the ring at a time. All dequeued packets are parsed and taken into consideration for packet count per UE. Given echo percentage, packets are processed further to swap their source and destination IP address. This process needs no recalculation of UDP and IP checksum, so there is no extra overhead. Also, source and destination MAC address also needs to swapped for send packets back to UPF. The rest of the packets are dropped, and thus one batch of 100 packets is processed.

DNN can also be configured for an echo percentage of greater than 100 upto 500% of packets. In such a situation, swapping IP and MAC addresses of the incoming would only generate 100% of the downlink data traffic. Hence additional new packets are needed to allocated memory, and already IP and MAC address swapped packets are copied to match the required percentage.

5.2 Mirroring incoming packets

In the real-world, the DNN would be a server or proxy server connecting to the user's desired website. For every user's request, there would be a response back from the server. This was emulated by echoing the packets back to the user. Echoing of packets was chosen because it needs very low overhead just to swap MAC and IP addresses without changing the payload or header. If any other packet content is changed, all checksums need to be recalculated.

Also, the uplink and downlink traffic from any UE would vary; it would not always be equal, so to emulate such traffic, a configuration parameter is there, which controls how much of the incoming traffic would be echoed back. Earlier design was trying to send back packets as soon as it was received, but as the rate was not consistent, precise mirroring of packets was not possible. Now current design waits for 100 packets to be received and then echoes the packets back or drops according to the given configuration. If the mirroring percentage is greater than 100, then new packets need to be created, and the incoming packet needs to be copied to new packets. This work will require more CPU computation, and if only one CPU core is doing the job, it might result in no free memory for incoming packets or sending the packets less then the required rate. So, this job was divided among n cores depending on NIC bandwidth as shown in figure 5.2.

5.3 Throughput visualization and retention

To quickly cross-check whether the QoS enforced by UPF for a particular speed is correctly working, displaying speed per second is very useful. Calculating every packet received and determining its length requires significant CPU computations and will result in a decrease in application receiving capability. To avoid this altogether, a dedicated core probes the NIC every second and takes the information about packets directly form NIC. The information is converted to packets per second (pps), bytes per sec (Bps), and Megabits per second (Mbps) for incoming and outgoing packets as shown in figure 6.9. Also, for long-running Qos experiments wth variable data rates, all the collected statistics are also logged in a file 6.10.

5.4 Maintaining packet count

Unlike the RAN incoming packets are not encapsulated, only outer headers are needed to parse, and also handling of packets is divided into a number of mirroring core to generate instantaneous downlink traffic. Every CPU has enough time to parse the packet and increase corresponding into an unordered array. As of now, the aforementioned design is working on 10G; for higher bandwidth NICs further optimizations can be added, such as using the DPDK hash library instead of unordered hash map.

5.5 Emulating multiple UEs

The design of this is more or less like RAN Emulator section 4.5. The packet does not require GTP encapsulating, and hence only the UE related information different per-packet id the IP address if UE. This packet is made per UE and stored in a map. As mentioned in section 4.5 for emulating multiple UEs per CPU core packet are sent

multiplexed with respect time. For faster access to the stored packet, an array is used per CPU core.

5.6 Controlling outgoing data traffic

The controlling rate of the traffic generated from the DNN is techniques already mention in section 4.7. For intel NICs DPDK rate-limiting APIs [6] are used, and for smartNIC the generation of the traffic is limited by adding delay.
The variable data plane traffic rate is also implemented in the same way as mentioned in section 4.7.1.

5.7 Challenges

5.7.1 Packets dropped at high incoming rate

All the received packets get polled by the single CPU core and then are checked if the packet is ARP or data packet. This design works for almost all range of traffic. At very high incoming, some of the packets are dropped, and this was mitigated by using rte_flow library, which divides the ARP and data packets at hardware-level into separate queues. The current design is working for 10G NIC, but for higher bandwidth, NICs more optimization can be added, such as enabling RSS to distribute incoming traffic in various rx queues.

5.7.2 Echoing all packets at greater than 100% echo rate

If only one CPU core is receiving packets and also transmitting the packets, it needs to perform the following computations. MAC and IP addresses need to be swapped,the new packets need to be allocated memory, and received packets need to copy into the new packets, and then all packets will need to be transmitted. This design works only if incoming traffic is very low, and the mirroring rate is very low. This design drops packet at high incoming rate.
To avoid the dropping of packets, the receiving and transmitting of the packets should be on different cores. Also, if enough cores are not allotted mirroring the traffic (echo percentage ≥ 400), then the actual traffic will be less leading to inaccuracies in experiments.

Chapter 6

Evaluations

6.1 Experimental Setup

Figure 6.1 shows the server configuration used for experiments. The servers are connected point to point by SFP cables. The first server hosts the DPDK RAN Emulator and all other NFs except UPF. The second server has UPF running; it can be any UPF (DPDK based or eBPF based). The last server acts as DNN; it either receives packets while uplink or may itself send messages in downlink.

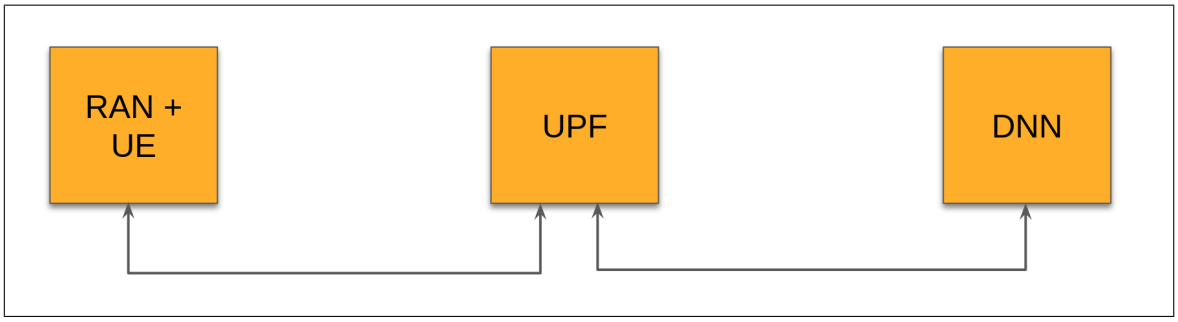

Figure 6.1: Server setup

Server Configuration: –

- **CPU :** Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
- **RAM :** 128G
- **NIC :** 10G / 40G

The DPDK based RAN Emulator can be configured for various throughput rates and packet sizes. It is configured with 8192 huge pages of size 2048MB, variable TX queues and CPU cores, single UE mode, or multiple UE mode.

6.2 Results

This section shows the data traffic generated by RAN Emulator with various packets sizes on 10G and 40G. Some of the experiments are done to answer the following questions:–

- How do different packet creation methods affect the throughput of RAN emulator?
- Will emulating multiple UEs per core affects the degrades the throughput?
- Does the RAN emulator scalable across scores?
- Do the batch size while sending packets affect throughput?
- How many CPU cores are required to saturate a 10G & 40G NIC for different payload sizes?

This section ends by showing some of the features of RAN and DNN emulator.

6.2.1 Comparision between different methods of creating packets

Data traffic can be generated by creating a complete packet in userspace and sending it via DPDK. This packet can be created in different ways and sent in batches. As the packet has 2 UDP headers, UDP checksum needs to be calculated twice. Table 6.1 First-row describes when UDP checksum is calculated by the normal c program. The second row shows that when DPDK API rte_checksum is used. The third row shows when UDP checksum is entirely disabled. The fourth one is creating a complete packet once and storing in a map. As it is clear from the table that UDP checksum is the bottle neck in packet creation.

Method of creating traffic	Payload size 1422		Payload size 64	
	Gbps	Mpps	Gbps	Mpps
Creating each packet (with UDP checksum)	1.93	0.17	0.57	1.12
Creating each packet (with UDP checksum using rte_cksum)	2.65	0.23	0.61	1.20
Creating each packet (without UDP checksum)	12.83	1.13	0.99	1.94
Crafting packet once (with UDP checksum)	20.06	1.76	3.78	7.38

Table 6.1: Throughput comparison when using different methods of packet creation

6.2.2 Throughput vs packet sizes

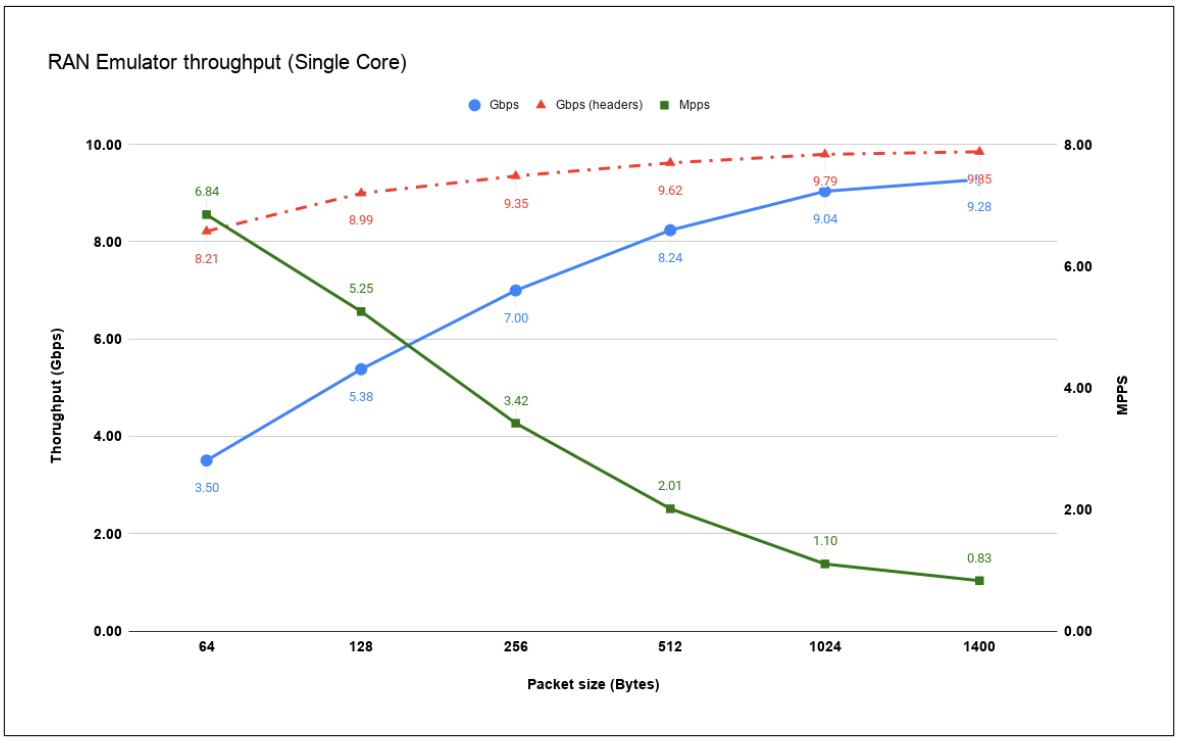

Figure 6.2: Throughput of various packet size on 10G NIC

When using 64Byte payload size, the number of packets generated is the maximum. As payload size increases, the number of packets decreases due to the NIC bandwidth. For larger payload size, the throughput with headers saturates the NIC with only one core.

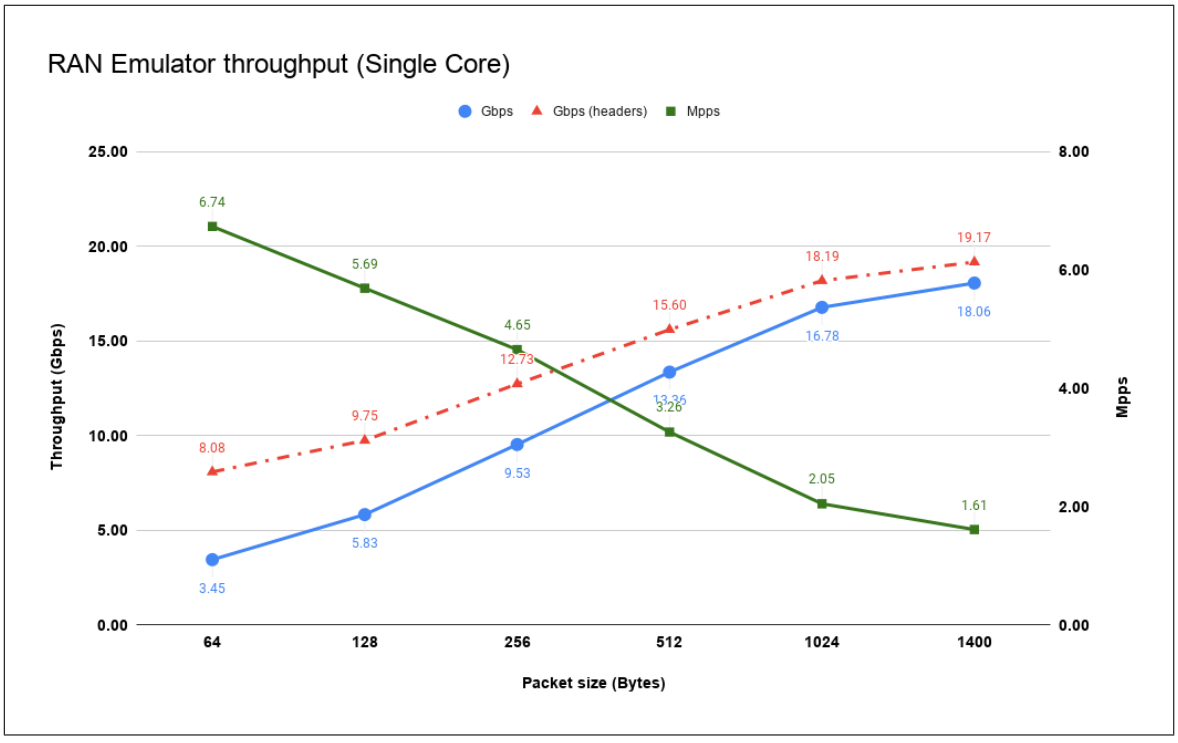

Figure 6.3: Throughput of various packet size on 40G NIC

In 40G NIC the throughput per core is no longer bottle-necked by 10G NIC and it reaches upto 19.17Gbps on single for 1400 packet size. Also for small packets size the packets generated per second are 6.74 Mpps

6.2.3 Single UE vs Multiple UEs throughput on single core

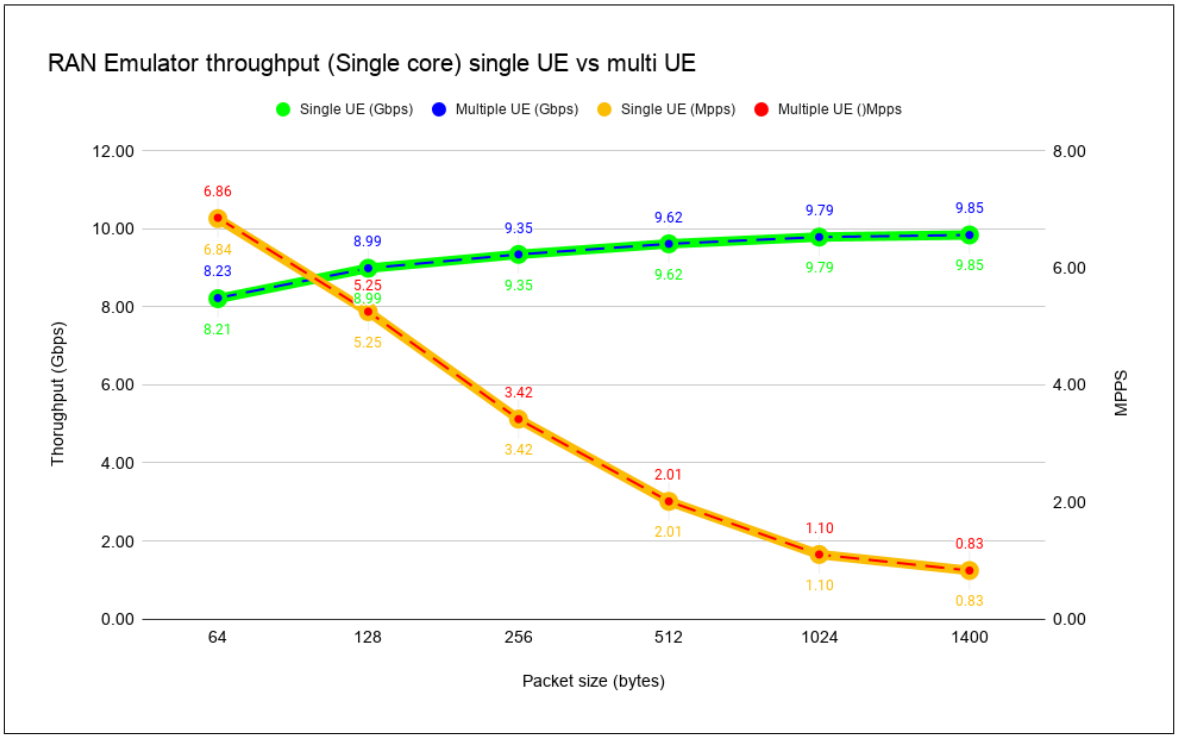

Figure 6.4: Single UE vs Multiple UEs throughput on 10G NIC

Figure 6.4 shows that even if more than one UE is emulated per core, the throughput generated by single CPU core is same as that of when only one UE is emulated.

6.2.4 Scalability

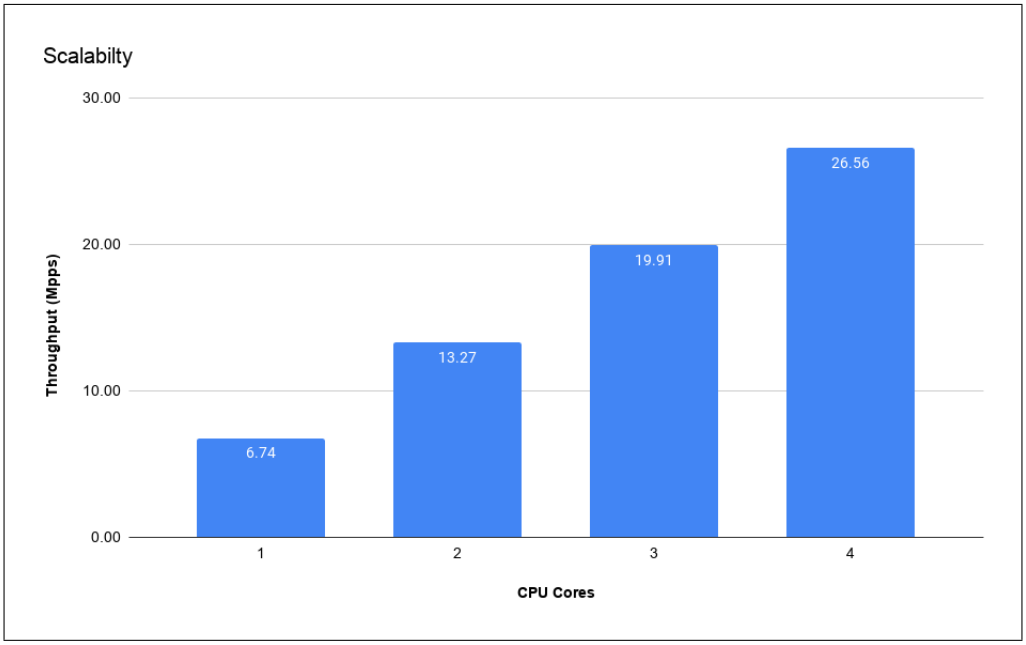

Figure 6.5: Scalability across CPU cores on 40G NIC

DPDK uses lockless and per core data structures and other optimization, which enables to scale the throughput of the application with respect to CPU cores. Figure 6.5 shows data traffic generated by the RAN Emulator, which scales almost linearly.

6.2.5 Batch size vs Throughput

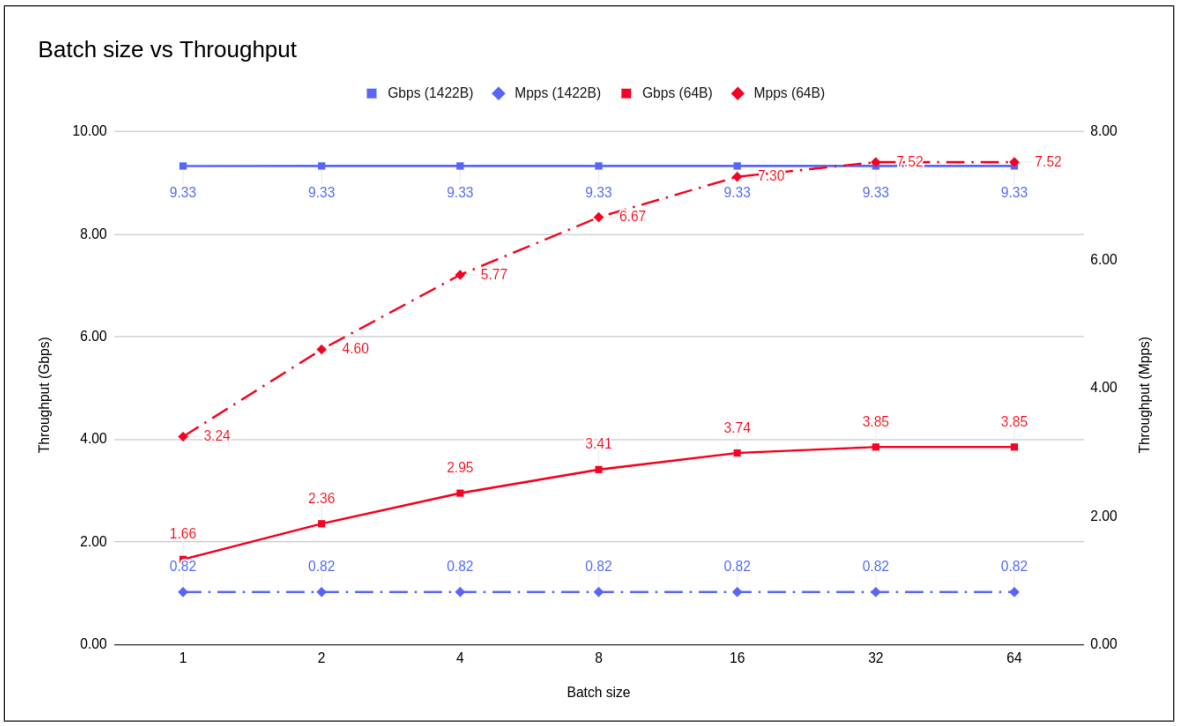

Figure 6.6: Batch size vs Throughput on 10G NIC

Batch size does not play a major role when it comes to sending packets with high payload. For packets having smaller packet sizes as the number of batch size increases throughput increases as shown in figure6.6. After batch size of 16 the throughput is almost same.

6.2.6 CPU cores vs packet size

Payload size	Cores needed for saturation	
	10G NIC	40G NIC
64 Bytes	2	5
128 Bytes	1	4
256 Bytes	1	4
512 Bytes	1	3
1024 Bytes	1	3
1422 Bytes	1	3

Table 6.2: Number of cores required to saturate NIC corresponding to packet size

Table 6.2 shows that saturating NIC with small packet sizes require more cores than that with larger packet sizes.

6.3 Proof of Correctness

6.3.1 Variable Data traffic

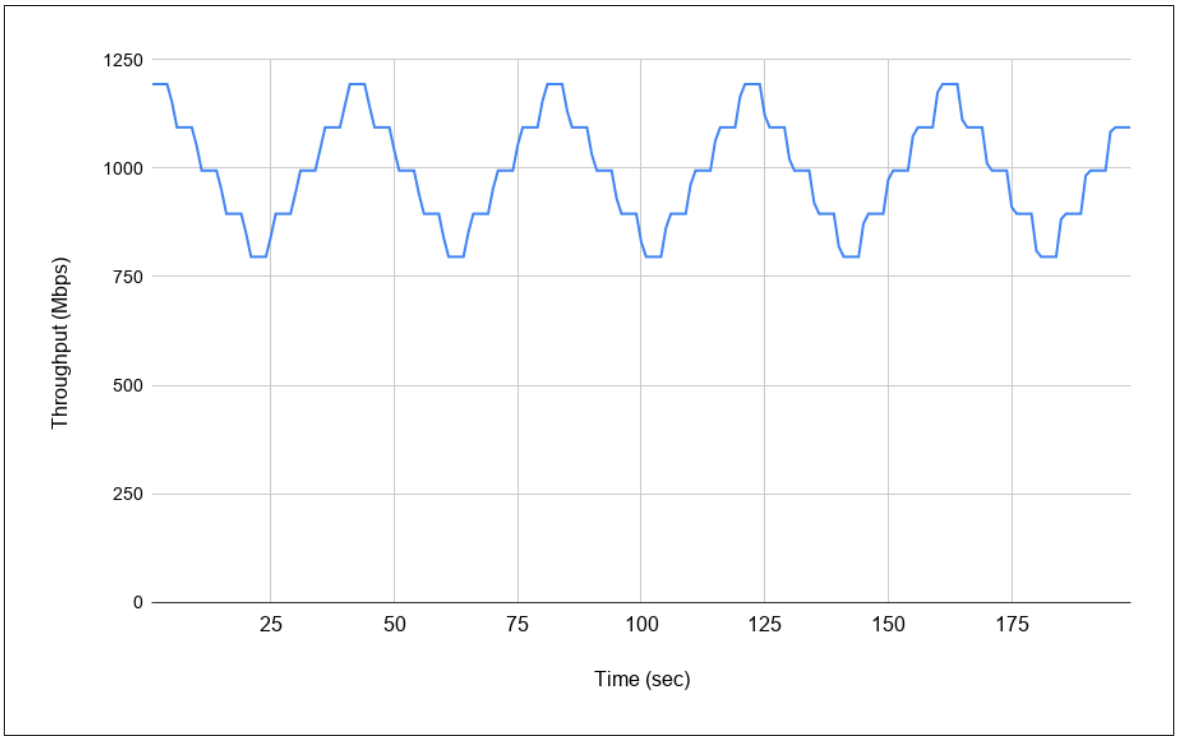

Figure 6.7: Variable data transmit rate with time

The figure 6.7 shows output from the RAN Emulator when variable data rate is enabled. In the above scenario, RAN emulator is sending data with rate of upper

bound of 1200Mbps and lower bound of 800Mbps. After every 5 seconds the data rate is changed by 100Mbps. Once the data rate hits upper or lower bound it adds or subtracts the given change rate after particular toggle duration to keep data rate within bounds.

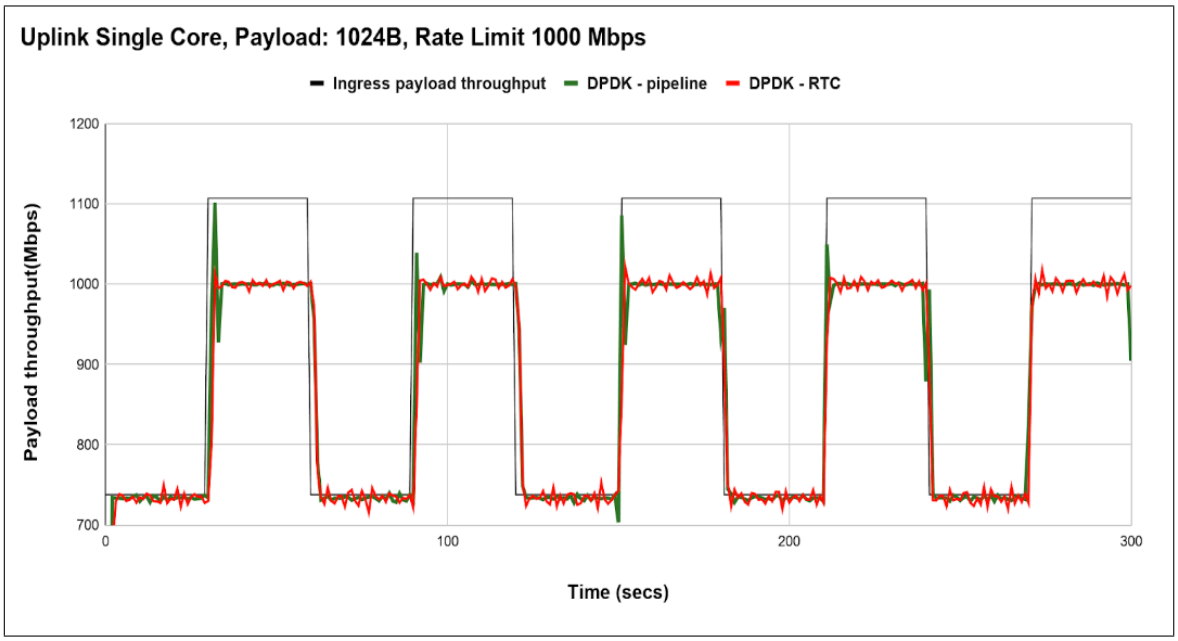

Figure 6.8: Time series graph showing rate limiting[1]

Figure 6.8 shows the timeline graph when packets are sent from a UE to the DNN. The RAN sends out 1024B payloads at different speeds, alternating between 800Mbps and1200Mbps (incl. headers) every 30secs, while the UPF rate is set at1Gbps.

6.4 Extra features

6.4.1 Real time throughput display

```
######################## NIC statistics for port 0  ########################
RX-packets: 519339      RX-errors:  0          RX-bytes:  748884074
RX-nombuf:  0
TX-packets: 259502      TX-errors:  0          TX-bytes:  374199120
Rx-pps:       106242   Rx-Kbps:      1225606   Rx-Mbps:         1225
Tx-pps:        53113   Tx-Kbps:       612717   Tx-Mbps:          612
############################################################################
```

Figure 6.9: Throughput is displayed per second

Figure 6.9 shows a snapshot of a real-time display of the per second display of incoming and outgoing traffic. It shows packets per second, bytes per second, and also cumulative packets and bytes received. This feature can be enabled in both RAN and DNN emulator.

[1]Data collected of DPDK UPF developed by Diptyaroop Maji

6.4.2 Throughput logging periodically

Elapsed seconds	Rx-Mbps [including headers]	Tx-Mbps [including headers]
0	0	0
1	0.000952	0.000952
2	869.6922	434.357088
3	1293.436744	644.606864
4	1293.437808	646.966376
5	1293.466456	647.233752
6	1225.6066	612.717
7	1194.048616	598.33604
8	1194.036392	595.453328
9	1194.040016	598.343264
10	1125.096496	560.943088

Figure 6.10: Throughput is logged per second

Above figure 6.10 shows a throughput log file generated by DNN. It contains the rx and tx throughput in Mbps, including headers. Such a log file is useful while performing experiments of long duration with varying rates of data traffic for example:- experiments like 6.8 and 6.7.

6.4.3 Maintaining packet counts

```
========== PER UE PKT COUNT ==========
UE IP                    Packet Count
192.168.0.3              49026
192.168.0.5              48435
192.168.0.9              48485
192.168.0.8              47772
192.168.0.4              49414
192.168.0.2              47916
192.168.0.6              48340
192.168.0.10             49674
192.168.0.7              49398
192.168.0.1              48679
```

Figure 6.11: Maintaining per UE packet count

Above snapshot 6.11 shows the per UE count at DNN. These packet counts help to determine the UPF's QoS enforcement correctness.

Chapter 7

Future Work

- **Exploring scalability across NUMA nodes**: The RAN Emulator is currently able to utilize all cores of one NUMA node when working with multiple UEs. Scaling performance across NUMA nodes and debugging why application crashes while using all cores will further increase the performance.
- **Handling IP fragmentation in DPDK**: Current end to end data plane of 5G core has not considered an increase in the size of data packets greater than MTU. As the l2 frames are handled by all UPF based on fast packet processing techniques, IP fragmentation will need to be handled manually.
- **Exploring DPDK KNI module**: Now tun devices are being used to send and receive packets from the NFs which are using sockets. DPDK provides KNI module, which might be useful in such scenarios. Exploring the exact working of KNI might be helpful in determining its worth in RAN Emulator.
- **Testing DPDK RAN + UPF on VMs with the help of SRIOV**: As of now, three server machines are required to test the end to end 5G data plane as DPDK runs on direct NIC. The testing can be done in one server machine if we get to run the DPDK based application on VMs with a virtual function of SRIOV.
- **Emulating CP traffic at high pps**: IOT devices can generate high control plane traffic than data traffic. To determine how well UPF handles parallel data and control plane traffic, CP traffic needs to be emulated at enough higher rate to study the impact on UPF.

Chapter 8

Conclusions

This report gives a brief overview of the limitation of the existing implementation RAN Emulator using TUN device. After removing TUN device dependency and shifting all data plane processing to userspace with the DPDK, these limitations are mitigated, and performance increased for all types of packets. The DPDK based RAN Emulator is scalable with available cores and perform very well even for 64B size packets. For receiving and handling such huge traffic from UPF, DPDK based DNN was also developed. RAN and DNN Emulator can generate multiple UE data traffic multiplexed in time. These applications are equipped with features like support for ARP, fine control over outgoing traffic, variable traffic generation with respect to time, etc. Both applications can saturate the 10G NICs and are suitable for UPF testing. Using DPDK, a testing infrastructure is built to stress the various capabilities of UPF.

Acknowledgement

I am grateful to Prof. Mythili Vutukuru for guidance, insightful discussions, and feedback. I would also like to thank 5G project team who helped me to get familiarized with 5G specs and testbed. I would also express my gratitude towards Diptyaroop Maji and Priyanka Naik for their help in understanding and implementing the program over DPDK. I would also like to thank M.Tech colleagues who gave their valuable feedback on the project.

References

[1] "3gpp ref #: 23.501 ystem architecture for the 5g system (5gs)." https://www.3gpp.org/ftp/Specs/archive/23_series/23.501/, 2017.

[2] "3gpp ref #: 23.502 system architecture for the 5g system (5gs)." https://www.3gpp.org/ftp/Specs/archive/23_series/23.502/, 2017.

[3] Intel DPDK, "Data plane development kit." https://www.dpdk.org/, 2014.

[4] Intel DPDK, "Blog on dpdk." https://blog.selectel.com/introduction-dpdk-architecture-principles/.

[5] "Kernel sk_buff strcuture." https://www.kernel.org/doc/html/v4.16/networking/kapi.html.

[6] Intel DPDK, "Dpdk rate limiting api." https://doc.dpdk.org/api/rte__ethdev_8h.html.

[7] Intel DPDK, "Dpdk timer apis." https://doc.dpdk.org/api/rte__cycles_8h.html.

[8] Intel DPDK, "Dpdk kni." https://doc.dpdk.org/guides/prog_guide/kernel_nic_interface.html.

[9] Intel DPDK, "Dpdk tun/tap." https://doc.dpdk.org/guides/prog_guide/kernel_nic_interface.html.

[10] Intel DPDK, "Dpdk rte_flow." https://doc.dpdk.org/guides/prog_guide/rte_flow.html.

[11] Intel DPDK, "Dpdk overview." https://doc.dpdk.org/guides/prog_guide/overview.html.

[12] Intel DPDK, "Dpdk api." http://doc.dpdk.org/api-19.05/.

[13] Intel DPDK, "Dpdk rte_mbuf." https://doc.dpdk.org/api/structrte__mbuf.html.

[14] Intel DPDK, "Dpdk rte_ring." https://doc.dpdk.org/api/structrte__ring.html.

[15] Intel DPDK, "Dpdk rte_hash." https://doc.dpdk.org/api/rte__hash_8h.html.