Design, Development and Optimization of UPF Data Plane (5G)

A Report
Submitted in partial fulfillment of the requirements
of the degree of

Master of Technology
in *Computer Science and Engineering*

by

Diptyaroop Maji
Roll No. 183050016

Supervisor:
Prof. Mythili Vutukuru

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
June, 2020

Abstract

The 3GPP 5G technology can provide excellent user experience for a variety of new bandwidth/latency driven use-cases that are arising nowadays, ranging from video streaming services to self-driven cars. This is due to the several modifications suggested in the 3GPP standards — modular architecture, separation of the Control and the User Plane and customizable packet processing pipeline, to name a few. As a result, 5G is a very active research area. IIT-B is building a 5G test-bed from scratch to contribute to that research. Currently, a kernel-based User Plane Function (UPF) has been built for the Data Plane (DP), which performs fine under normal circumstances. However, it cannot handle very fast I/O due to the kernel and the implementation limitations, and thus cannot saturate line rate. This report briefly describes the current 5G Data Plane (DP) design, before moving on to its main contribution — talking about an optimized UPF Data Plane design. It describes a UPF design where the kernel is bypassed using Intel's Data Plane Development Kit (DPDK) [1]. Since the data plane does not use the kernel TCP/IP stack, packets are processed in the userspace. The report explores both run-to-completion (RTC) and pipeline models in the DPDK based UPF and compares them with other designs of UPF currently being developed. Unrestricted by the limitations associated with the kernel, the DPDK based UPF saturates the $10G$ line rate between the Radio Access Network (RAN), UPF and the Data Network Name (DNN) while supporting essential features like Quality of Service (QoS). The RTC design performs better currently and saturates 10 $Gbps$ from 256 B payload onwards. In contrast, the pipeline design saturates the uplink for payloads $>=$ 256 B and downlink from 512 B payload onwards (when a single core is dedicated to DP packet processing).

Contents

List of Figures

List of Tables

Abbreviations

3GPP	Third Generation Partnership Project
AMBR	Aggregate Maximum Bit-rate
AMF	Access and Mobility Management Function
ARP	Address Resolution Protocol
CP	Control Plane
DNN	Data Network Name
DP	Data Plane
DPDK	Data Plane Development Kit
eBPF	extended Berkeley Packet Filter
F-TEID	Full Tunnel Endpoint Identifier
GBR	Guaranteed Bit-rate
GTP	GPRS Tunnelling Protocol
IE	Information Element
LTE	Long Term Evolution
MTU	Maximum Transmission Unit
NF	Network Function
NRF	Network Repository Function
NUMA	Non Uniform Memory Access
PDU	Packet Data Unit
PFCP	Packet Forwarding and Control Protocol
QFI	QoS Flow Identifier
QoS	Quality of Service
RAN	(Radio) Access Network
SFP	Small Form-factor Pluggable
SMF	Session Management Function
TLB	Translation Lookaside Buffer
UE	User Equipment
UPF	User Plane Function
XDP	eXpress Data Path

1. Introduction

Cellular traffic is growing at an exponential rate due to a diverse set of services ranging from voice and video calling, gaming, video streaming (which require high bandwidth), to home IoT devices and autonomous cars (which require ultra-low latency). The 3GPP 5G technology is believed to be able to enhance the user experience for these and several other new use-cases, which the currently deployed 4G networks are not able to. This improvement is due to several modifications in both specification and implementation aspects in 5G, such as modular architecture with well-defined Network Functions (NF), a clear separation between the Control and the User Plane (the terms “Data Plane” and “User Plane” are used interchangeably in this report), and customized processing pipeline for different devices based on their high bandwidth or low latency requirements (network slicing). As a result, there is currently a massive drive to design, develop and understand the 5G technology in both academia and industry. Here at IIT-B, we are building a 5G test-bed from scratch so that we can explore various design options both in the Control Plane and the Data Plane, and contribute to the understanding of 5G.

1.1 Problem Statement

Based on the context provided above, the first goal was to build the User Plane Function (UPF) from scratch which would enable a User Equipment (UE) to send and receive data from the Internet. However, the current implementation of UPF in the 5G test-bed has several limitations which degrade its performance. For instance, the kernel stack cannot handle packet processing when the I/O rate is very high; there is a TUN device used between the UPF and the DNN which forms a bottleneck etc. These decreases the Data Plane (DP) throughput and increases the packet processing latency. The exciting thing to explore is how to remove the said limitations and overheads so that the UPF forwards packets at the $10G$ line rate in both uplink and downlink. It should be able to saturate the line rate even when multiple UEs are sending different sized packets simultaneously while supporting essential features like Quality of Service (QoS) for each UE.

1.2 Contribution

Stage-I was about developing the UPF from scratch. I am a part of the team that built the kernel-based UPF along with the GTP library (required for packet encapsulation/decapsulation in the path between the RAN and the UPF). The kernel-based implementation bottlenecked at 1.80 *Gbps* (uplink) and 1.77 *Gbps* (downlink) for 1422B packets (without any QoS implementation), due to the limitations mentioned above. Possible solutions to the problem(s) include:

1. Bypassing the kernel entirely to avoid the overheads and processing packets in the userspace.

2. Shifting the packet processing to hardware.

3. Processing in the Linux kernel but at the device driver level, skipping the kernel TCP/IP stack.

Since kernel bypass is one of the most popular techniques currently used in both academia and industry, Stage-I then moved on to optimizing the 5G test-bed DP by bypassing the kernel with the help of Intel's Data Plane Development Kit (DPDK) [1] framework, so that line rate is achieved while forwarding both uplink and downlink packets. At the end of Stage-I, UPF was saturating the $10G$ link for $1500B$ packets. However, the performance was significantly below par for smaller sized packets or when multiple UEs sent data simultaneously. The UPF also did not have essential features like ARP or Quality of Service (QoS). Stage-II was focused on optimizing the DPDK based UPF further, adding new features, and implementing Quality of Service (QoS). The current version of the DPDK based UPF enforces QoS on each UE based on pre-defined rules. It also supports features like ARP and verification of checksum for incoming packets. The current DPDK [1] based UPF implementation has been tested with $99k$ sessions and 2^{14} UE sending packets simultaneously from the RAN, wherein it saturates the $10G$ connection in the DP for payload sizes 256 B or more (when a single core is dedicated for DP packet processing). The maximum processing capacity of the UPF is 4.11 million packets per second (Mpps) in the downlink. This is not enough to saturate $10G$ for smaller payloads when a single core is used for packet processing. Of course, the line rate is saturated when multiple cores are used. Current and future work involves investigating techniques to scale across NUMA nodes, implementing more sophisticated QoS where each UE is guaranteed to have a minimum bitrate, further optimizing the UPF design using hardware accelerators etc.

The following sections in this report are organized as follows: Section 2 discusses the 5G architecture as given in the 3GPP specifications. Section 3 talks about how the data plane works in the 5G test-bed and what are the limitations associated with the kernel-based UPF. Section 4 addresses how to overcome those limitations and optimize the data plane by techniques like bypassing the kernel. It also describes the new DPDK-based UPF design and the challenges associated while developing the UPF. Section 5 elaborates on how Quality of Service (QoS) is implemented and enforced. Section 6 outlines the experimental setup, measures the performance of the UPF in terms of throughput and correctness and compares the DPDK based UPF with other designs. Sections 7 and 8 talk about related work that is going on currently and potential future work. Section 9 concludes the report.

2. Background

2.1 5G architecture

Figure 2.1 shows the 5G architecture as per 3GPP standards. UE to RAN is the wireless 5G Access Network, while the remaining is the wired 5G Core network.

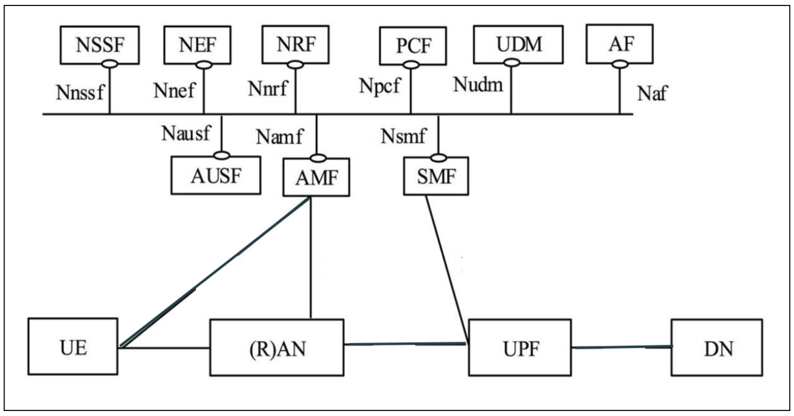


Figure 2.1: 5G architecture[1]

The 5G architecture is a modular structure, with different Network Functions (NFs) handling different well-defined functionalities. The Network Repository Function (NRF) supports service discovery. All other NFs must register to the NRF when they start, and the NRF provides information about other discovered NFs to that NF. The Access and Mobility Function (AMF) is the leading Control Plane (CP) component in 5G, which deals with registration and connection management for User Equipments (UE). The Session Management Function (SMF) deals with session management (eg., PDU session establishment, described later). SMF is also in charge of IP allocation to the UEs. All messages from the UE reach SMF via the AMF.

There is a clear separation between the Control Plane (CP) and the Data Plane (DP) in 5G. All the above components are in CP, whereas the main component in the DP is the User Plane Function (UPF), which processes and forwards the actual data packets. Section 3.1 talks about UPF in detail.

Before a UE sends/downloads any data from the Internet, there is an exchange of CP messages between the SMF and the UPF to set up a session. During session establishment, SMF installs rules in the UPF, and the UPF processes all packets following those rules. All data transfer takes place inside the session, which can be released if the UE no longer needs to send/receive any data. Sections 3.2 and 3.3.2 elaborate the session setup procedure and UPF packet processing respectively.

[1]3GPP TR 23.501, July 2017, Figure 4.2.3-1

3. Data Plane in the 5G Test-bed

This section describes the kernel-based implementation of the UPF in the 5G test-bed. The following subsections give a brief introduction about the UPF, PDU session establishment and GTP tunnelling. The final subsection explains the details of the end to end Data Plane.

3.1 User Plane Function (UPF)

UPF is the main NF in the 5G Data Plane (DP). As per specification, it has the following functionalities:

- In the Data Plane, it is the anchor point between the Access Network (AN) and the 5G Core Network (CN). It is also the gateway to the DNN for uplink packets.
- The main functionality of the UPF is to route and forward the packets based on some rules set by the SMF.
- Handling and enforcing Quality of Service (QoS).
- UPF can keep track of and report traffic usage statistics.
- When the UE is idle, UPF can also buffer downlink packets until the UE becomes active.

3.2 PDU Session Setup

Before actually sending any uplink/downlink Data Plane (DP) packets, the UE must request for a PDU session to be set up. The SMF initiates the session establishment procedure with the UPF wherein the GTP tunnel endpoint ID (TEID) is set up. TEID is required for creating a GTP tunnel between the UPF and the RAN through which the DP packets travel. The rules tell the UPF what to do when it receives an uplink/downlink packet — how to process it and whether to forward it, buffer it for the time being, or discard it. The current implementation does not support buffering.

3.2.1 Packet Forwarding Control Protocol (PFCP) interface

All communication between the UPF and the SMF is in the form of PFCP messages. A PFCP message consists of a PFCP header and a set of Information Elements (IE), which carry the rules that need to be installed at the UPF. The PFCP library, along with the session setup procedure (briefly described next) was developed by the project engineers associated with the 5G test-bed and has been optimized (implementation-wise) as a part of this project.

- **PFCP Association Setup**
  On discovering a UPF, the SMF sends out a PFCP association request to the UPF, which contains the features that the SMF supports along with the SMF IP. On receiving the request, UPF sends out an association response, containing its IP, the features it supports, and the TEID range that the SMF can allocate TEIDs from while establishing a session. Association is done before any PDU session establishment.

- **PFCP Session Establishment**
  When a UE requests for a PDU session, the SMF sends out a PFCP Session Establishment request to the UPF. This request contains the Packet Detection Rules (PDR) and corresponding Forward Action Rules (FAR) and QoS Enforcement Rules (QER) which the UPF needs to install and check against uplink packets arriving in the future. The request also contains a TEID (from the allowed range) for the UPF. The UPF tries to install the rules and sends out a success/failure message in the form of Session Establishment Reply.
  SMF and UPF also share the Session endpoint ID (SEID) while establishing the session, which distinguishes the current session from other sessions. All future session-related messages need this SEID.

- **PFCP Session Modification**
  After receiving the RAN-side GTP tunnel information, the SMF sends a Session Modification Request to the UPF. Session modification request contains the PDRs, FARs and QERs for downlink packets along with RAN side GTP tunnel info, which the UPF needs to create a GTP tunnel with the RAN and forward any GTP packets. UPF sends out a success/failure reply based on whether it was able to install the rules.

- **PFCP Session Release**
  On receiving a session release request from RAN/UE, the SMF sends out a PFCP session release request to the UPF to tear down the corresponding session. The UPF deletes all the PDRs/FARs related to that session along with the SEID and sends back a success/failure reply to the SMF.

3.2.2 Optimizing the PFCP library

The initial implementation of the PFCP library was sub-optimal. Vectors and ordered maps were being used to store PFCP messages. To search an IE, the data structures were traversed, which is $O(n)$ in the worst case (n being the number of elements stored). Although this was not becoming a bottleneck even when multiple sessions were created in the control plane (tested with $99K$ sessions), it had the potential to become one if the number of sessions went up to millions. In Stage-II, the vectors and the ordered maps were replaced by unordered maps (hashtables). As a result, the worst-case time complexity to find any IE has come down to $O(1)$, and the library is now optimized.

3.3 Data Plane procedures

After establishing a PDU session, Data Plane (DP) packets can now go from the UE to the DNN and vice versa. This section describes the GTP tunnelling between the RAN and the UPF, that is required for supporting UE mobility, before moving on to discuss how the current (kernel-based) Data Plane works.

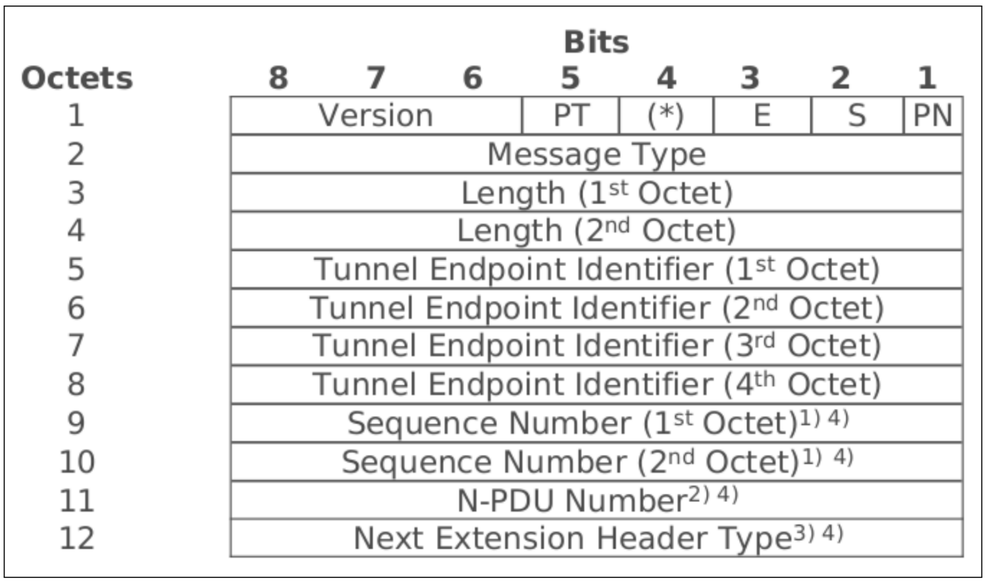
3.3.1 GTP tunnelling

Tunnelling is required to support UE mobility. GTP-U tunnels carry encapsulated data packets between a pair of NFs (RAN-UPF or UPF-UPF). Usually, both endpoints have a Tunnel Endpoint ID (TEID) associated with them for both ways communication. TEID (of the destination node) present in GTP header tells to which tunnel a particular packet belongs. This implementation uses GTP version 1 (GTPv1), as per specifications, which is a unidirectional tunnel. Hence, a pair of tunnels are needed for duplex connections.

GTPv1-U sending endpoint: GTP runs over UDP. So, an incoming IP packet at the sending endpoint shall be encapsulated with a GTP header, followed by UDP and IP headers. If the outer IP packet size is more than the MTU of the link between the endpoints, necessary fragmentation shall be done (currently not supported).

GTPv1-U receiving endpoint: Receiving endpoint shall reassemble all fragments to form a complete IP packet (reassembling not supported currently). Then, necessary decapsulation needs to be carried out to extract the inner IP packet by removing the outer IP, UDP and GTP headers. (GTP listens for incoming traffic on port 2152).

GTP-U header: The GTP-U header is a variable-length header with a minimum length of 8 *B* (refer Figure 3.1 for the format of a GTP-U header).

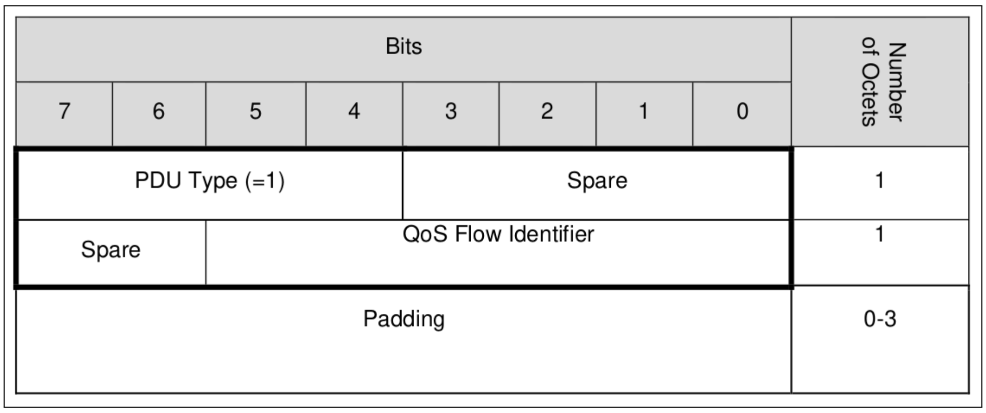


	Bits							
Octets	8	7	6	5	4	3	2	1
1	Version			PT	(*)	E	S	PN
2	Message Type							
3	Length (1st Octet)							
4	Length (2nd Octet)							
5	Tunnel Endpoint Identifier (1st Octet)							
6	Tunnel Endpoint Identifier (2nd Octet)							
7	Tunnel Endpoint Identifier (3rd Octet)							
8	Tunnel Endpoint Identifier (4th Octet)							
9	Sequence Number (1st Octet)1) 4)							
10	Sequence Number (2nd Octet)1) 4)							
11	N-PDU Number2) 4)							
12	Next Extension Header Type3) 4)							

Figure 3.1: GTP header[1]

[1]Figure taken from [12]

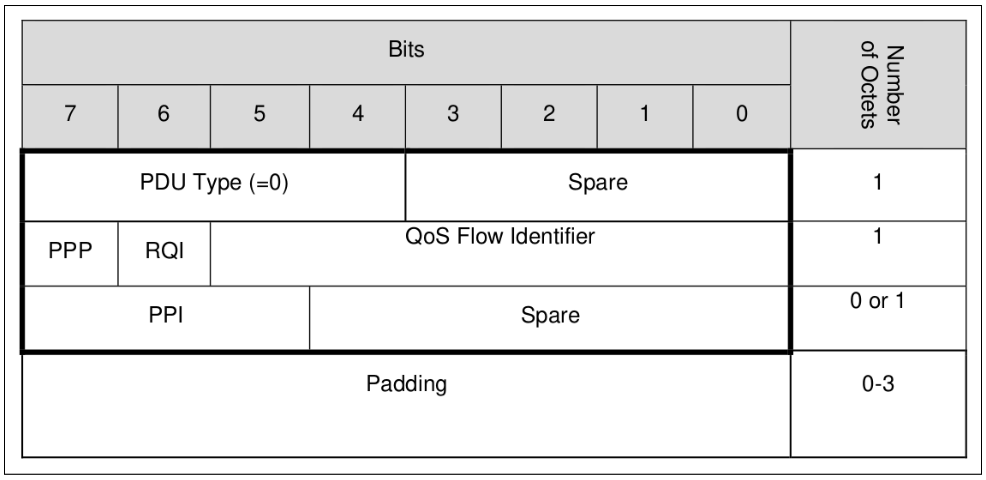
- **Tunnel Endpoint Identifier:** Identifies TEID at the receiving GTP endpoint.
- **Next Extension Header Type:** States the type of GTP extension header. It is interpreted only if the *E* flag is set in the GTP header. The UPF uses the *PDU Session Container* type extension header.



Bits								Number of Octets
7	6	5	4	3	2	1	0	
PDU Type (=1)				Spare				1
Spare		QoS Flow Identifier						1
Padding								0-3

Figure 3.2: UL PDU session container[2]

GTP-U extension header: The GTP-U extension header is a variable-length header with a minimum length of 4 *B*. GTP extension headers are linked (one after another if more than one) to the end of the main GTP header. In the current implementation, *UL PDU Session Container* is appended to the GTP header for up-link packets and *DL PDU Session Container* for downlink packets. Figures 3.2 and 3.3 show the two headers respectively. The extension headers contain the QoS Flow Identifier (QFI) field, which is used to differentiate QoS flows and enforce QERs to the incoming packets. Chapter 5 discusses QFI in details.



Bits								Number of Octets
7	6	5	4	3	2	1	0	
PDU Type (=0)				Spare				1
PPP	RQI	QoS Flow Identifier						1
PPI			Spare					0 or 1
Padding								0-3

Figure 3.3: DL PDU session container[3]

[2]Figure taken from [14]
[3]Figure taken from [14]

3.3.2 Packet Processing

Packet Detection Rules (PDR) are added during the PFCP Session Establishment or Modification procedure. Each PDR will have a Packet Detection Information (PDI) against which incoming packets will be matched. Only the source interface is a mandatory PDI field. All other fields are optional and implementation-specific. For uplink packets, optional PDI fields include the UE IP (i.e., inner IP source address), GTP-TEID and the QoS Flow Identifier (QFI). For downlink packets, only UE IP (IP destination address) is included currently. Packets matching the PDI will follow the Forwarding Action Rule (FAR) and the QoS Enforcement Rule (QER) stated in the PDR. A FAR has an "apply action" parameter, indicating whether a packet is to be forwarded/buffered/dropped. QER limits the maximum outgoing rate per session by specifying an Aggregate Maximum Bit Rate (AMBR).
Figure 3.4 shows the packet processing pipeline/sequence for an incoming packet at the UPF (URR is not currently supported). Once a UPF receives a packet, it will perform a session-wise lookup of the PDRs, in the decreasing order of PDR precedence within a PFCP session. The UPF will stop once a matching PDR is found (that is, only the highest precedence PDR shall be selected). A packet matches a PDR if all the fields in the corresponding PDI matches with the packet header fields. If a packet does not match any of the PDRs, it will be silently discarded.

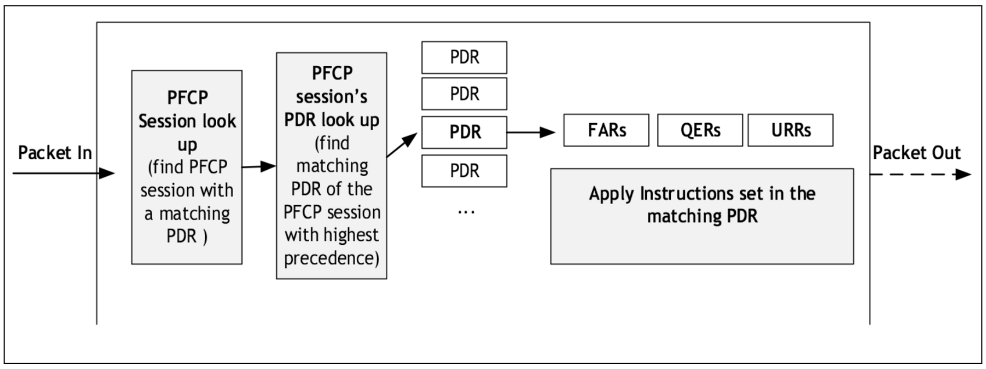

Packet In
PFCP Session look up (find PFCP session with a matching PDR)
PFCP session's PDR look up (find matching PDR of the PFCP session with highest precedence)
PDR
PDR
PDR
PDR
...
FARs
QERs
URRs
Apply Instructions set in the matching PDR
Packet Out

Figure 3.4: DP Packet processing pipeline[4]

For instance, suppose that the following 2 PDRs are installed for a particular session (assuming only 1 session).

PDR-1: Src iface: ACCESS, G-TEID: X, (Src) IP: <UE_IP1>, QFI: Y,
Remove GTP header, Run FAR-1
FAR-1: Dest iface: CORE, Action: FORW
QER-1: SEID: S1, AMBR: R1
PDR-2: Src iface: ACCESS, G-TEID: X, (Src) IP: <UE_IP2>, Run FAR-2
FAR-2: Dest iface: CORE, Action: BUFF
QER-2: SEID: S2, AMBR: R2

So, when a packet arrives at the UPF, PDR-1 will be checked first (assuming PDR-1 has higher precedence than PDR-2). If it matches, FAR-1 and QER-1 will be executed. Otherwise, PDR-2 will be checked. If the packet does not match any of the PDRs, it will be dropped.

[4]Figure taken from [13]

3.3.3 End to end data transfer

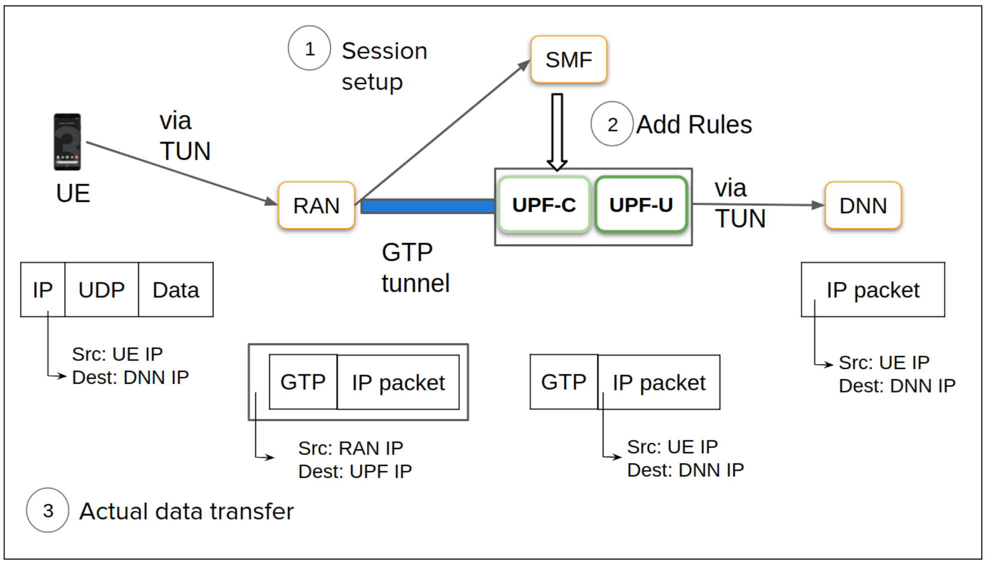


Figure 3.5: DP implementation in 5G test-bed

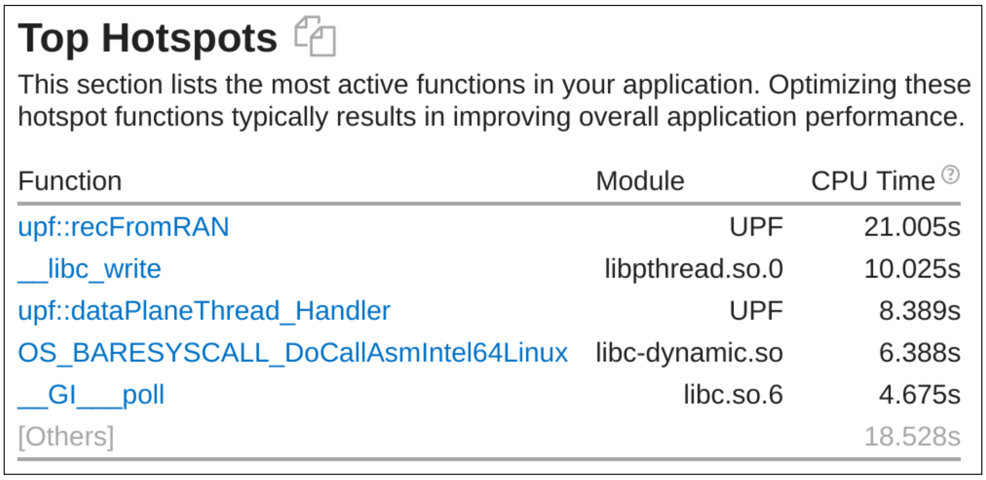
In the kernel-based implementation in the 5G test-bed, the UE and the RAN (emulator) are on the same machine. Hence, we use a TUN device there to mimic the wireless connection between the UE and the RAN to get an IP packet at the RAN. The UE thread in the system sends out a UDP packet to the DNN, which is re-routed to the TUN device, from where the RAN thread reads the IP packet. It encapsulates the packet with GTP using UPFs F-TEID and sends it out to the UPF. On receiving the packet, the UPF decodes the GTP message and processes the packet based on the previously set rules. Currently, only forwarding the packet to DNN is supported. After packet processing is done, UPF is left with an IP packet. If it sends out the packet via a normal UDP socket, another layer of IP/UDP headers will be added to it by the kernel stack, which is not desired. Hence, here also, a TUN device is used, which acts as a tunnel to send out the IP packet to the DNN. For this to work, *ip_forwarding* should be enabled at the UPF. Figure 3.5 captures the DP packet flow.

For the reverse path from the DNN to a UE, the DNN has a packet with the destination IP address as the UE IP address. As UE IP can be from another network, the kernel will drop the packet in the DNN itself as the current setup does not have any routers. So, we need to add a route stating that if the destination IP is that of the UE, the kernel should forward it to the UPF. The UPF again uses the same TUN device to get an IP packet from the kernel. It then processes the packet, adds a GTP header with RAN F-TEID, and forwards it to the RAN, which does the necessary processing before sending it to the UE thread.

3.4 Kernel based UPF limitations

Although the design is functionally correct, it can reach only up to 1.80 *Gbps* in uplink and 1.77 *Gbps* in the downlink (QoS disabled). This is due to the following limitations:

- When a packet comes in the NIC, an interrupt is generated. The packet is copied once in the kernel space and then copied again into the userspace. Memory is also allocated/deallocated per packet, and there are several system calls (and hence, context switching overhead) involved from when the NIC receives a packet to when it finally reaches the application and vice versa. All these factors increase the packet processing latency as well as decrease the throughput. For large packets, although the copying overhead is not noticed, other factors are still there.
- The TUN device used between the UPF and the DNN also becomes a bottleneck at high I/O rate, as it involves multiple packet copies both in the kernel and in the userspace.



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

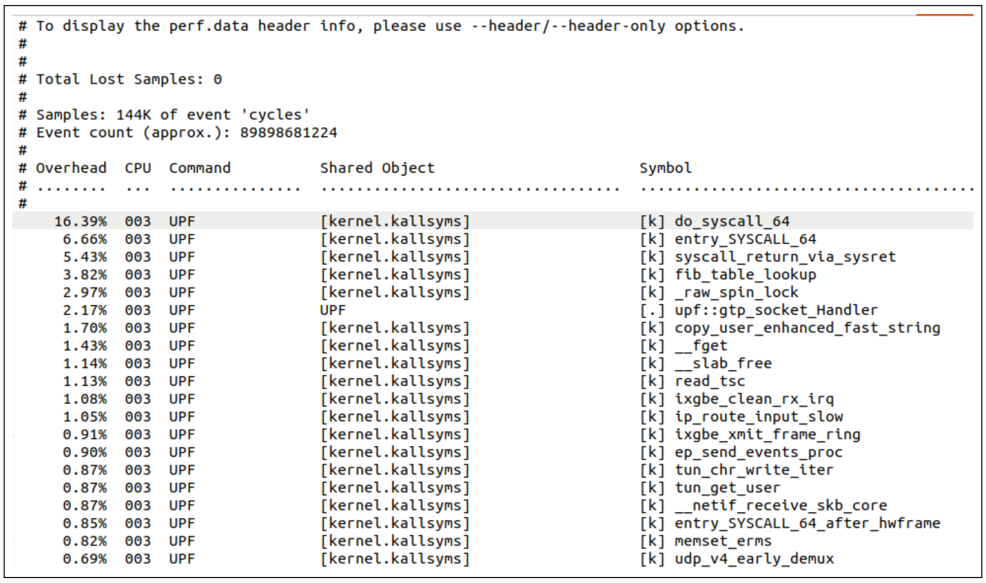
Function	Module	CPU Time
upf::recFromRAN	UPF	21.005s
__libc_write	libpthread.so.0	10.025s
upf::dataPlaneThread_Handler	UPF	8.389s
OS_BARESYSCALL_DoCallAsmIntel64Linux	libc-dynamic.so	6.388s
__GI___poll	libc.so.6	4.675s
[Others]		18.528s

Figure 3.6: Kernel UPF bottlenecks[5]

Figure 3.6 shows the functions which took the maximum CPU time in the UPF. Out of the 69.010 secs elapsed time, *recFromRAN* (underlying function: ___libc_recvfrom), which receives packets from the kernel, took the most time (21.005 secs, which equates to 30.44% of CPU time), while *___libc_write* used to write packets into the TUN device took the second most time (10.025 secs, which equates to 14.53% of CPU time).

To further prove that the kernel is indeed the bottleneck and not the userspace UPF function, the UPF is modified to forward any incoming packets from RAN without any processing. Only the data offset pointer is moved so that the inner packet is sent to the DNN. Figure 3.7 shows the top 20 hotspots in this situation, where all but one are kernel functions (profiling done using *perf* tool). This application can forward 1500 B packets at the rate of 0.2 $Mpps$ (equating to 2.4 $Gbps$), which is still well short of the 10 $Gbps$ line rate.

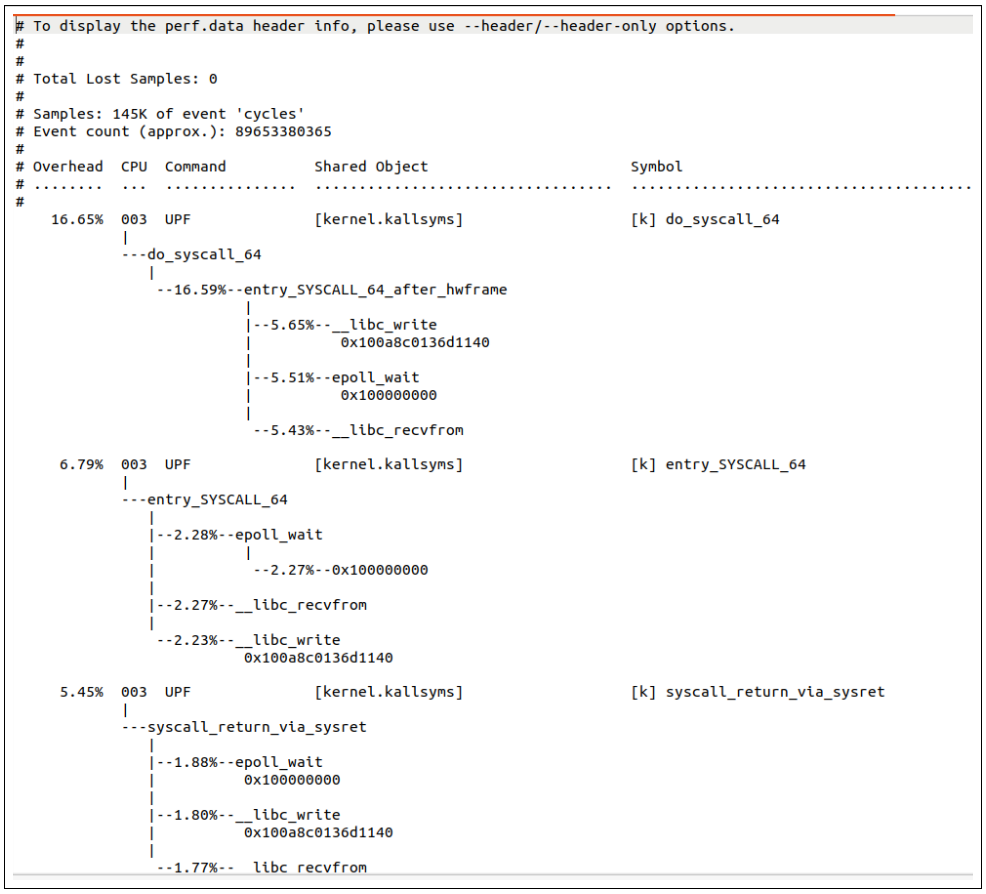
[5]Profiling was done using Intel vTune Amplifier



```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 144K of event 'cycles'
# Event count (approx.): 89898681224
#
# Overhead  CPU  Command          Shared Object                      Symbol
# ........  ...  ...............  .................................  .....................................
#
    16.39%  003  UPF              [kernel.kallsyms]                  [k] do_syscall_64
     6.66%  003  UPF              [kernel.kallsyms]                  [k] entry_SYSCALL_64
     5.43%  003  UPF              [kernel.kallsyms]                  [k] syscall_return_via_sysret
     3.82%  003  UPF              [kernel.kallsyms]                  [k] fib_table_lookup
     2.97%  003  UPF              [kernel.kallsyms]                  [k] _raw_spin_lock
     2.17%  003  UPF              UPF                                [.] upf::gtp_socket_Handler
     1.70%  003  UPF              [kernel.kallsyms]                  [k] copy_user_enhanced_fast_string
     1.43%  003  UPF              [kernel.kallsyms]                  [k] __fget
     1.14%  003  UPF              [kernel.kallsyms]                  [k] __slab_free
     1.13%  003  UPF              [kernel.kallsyms]                  [k] read_tsc
     1.08%  003  UPF              [kernel.kallsyms]                  [k] ixgbe_clean_rx_irq
     1.05%  003  UPF              [kernel.kallsyms]                  [k] ip_route_input_slow
     0.91%  003  UPF              [kernel.kallsyms]                  [k] ixgbe_xmit_frame_ring
     0.90%  003  UPF              [kernel.kallsyms]                  [k] ep_send_events_proc
     0.87%  003  UPF              [kernel.kallsyms]                  [k] tun_chr_write_iter
     0.87%  003  UPF              [kernel.kallsyms]                  [k] tun_get_user
     0.87%  003  UPF              [kernel.kallsyms]                  [k] __netif_receive_skb_core
     0.85%  003  UPF              [kernel.kallsyms]                  [k] entry_SYSCALL_64_after_hwframe
     0.82%  003  UPF              [kernel.kallsyms]                  [k] memset_erms
     0.69%  003  UPF              [kernel.kallsyms]                  [k] udp_v4_early_demux
```

Figure 3.7: Kernel UPF profiling[6]

Figure 3.8 expands the top 3 hotspots (the cumulative values are slightly different from Figure 3.7 as *perf* was run a second time). Similar to Figure 3.6, *__libc_write*, *epoll_wait* and *__libc_recvfrom* take the most time.



```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 145K of event 'cycles'
# Event count (approx.): 89653380365
#
# Overhead  CPU  Command          Shared Object                      Symbol
# ........  ...  ...............  .................................  .....................................
#
    16.65%  003  UPF              [kernel.kallsyms]                  [k] do_syscall_64
            |
            ---do_syscall_64
               |
                --16.59%--entry_SYSCALL_64_after_hwframe
                          |
                          |--5.65%--__libc_write
                          |          0x100a8c0136d1140
                          |
                          |--5.51%--epoll_wait
                          |          0x100000000
                          |
                           --5.43%--__libc_recvfrom

     6.79%  003  UPF              [kernel.kallsyms]                  [k] entry_SYSCALL_64
            |
            ---entry_SYSCALL_64
               |
               |--2.28%--epoll_wait
               |          |
               |           --2.27%--0x100000000
               |
               |--2.27%--__libc_recvfrom
               |
                --2.23%--__libc_write
                          0x100a8c0136d1140

     5.45%  003  UPF              [kernel.kallsyms]                  [k] syscall_return_via_sysret
            |
            ---syscall_return_via_sysret
               |
               |--1.88%--epoll_wait
               |          0x100000000
               |
               |--1.80%--__libc_write
               |          0x100a8c0136d1140
               |
                --1.77%--  libc recvfrom
```

Figure 3.8: Kernel UPF detailed trace[7]

[6]Profiling done with perf
[7]Profiling done with perf

4. Optimizing the 5G Data Plane

4.1 Intel Data Plane Development Kit

Intel's Data Plane Development Kit (DPDK) [1] is a framework that consists of libraries for processing packets very fast in data plane applications by bypassing the kernel processing stack and allowing applications to process Layer-2 packets in the userspace. DPDK [1] creates an Environmental Abstraction Layer (EAL) that hides the environment and architecture related complexities from the application and provides services (APIs) to the application for configuring NIC ports and queues, memory management, core affinity procedures etc. quickly and flexibly. NIC ports need to be bound to DPDK [1] before running an application. Once a NIC port is bound to DPDK [1], it is not visible to the kernel, and standard kernel-based networking applications cannot run on that particular NIC port.

Instead of traditional 4K pages, DPDK [1] needs hugepages (usually 2MB) to be reserved by the system. Hugepages are required for increased performance, as large page size means fewer pages will be needed and thus less TLB misses. Also, hugepages can support a larger memory pool to be used by DPDK [1] packet buffers (A fixed memory pool is pre-initialized from which the buffers are allocated, instead of allocating memory every time a packet is received). Figure 4.1 highlights the essential libraries provided by DPDK [1] for achieving high performance in I/O intensive applications.

- **rte_eal:** Provides the Environmental Abstraction Layer.
- **rte_mempool:** [2] Allocator of a pool of fixed-sized objects. A mempool handler stores the objects, which is by default a ring (circular queue).
- **rte_mbuf:** [3] Uses the mempool library to allocate and free buffers (*mbufs*). These buffers are used by the DPDK application to store incoming/outgoing packets.
- **rte_ring:** [4] It is a FIFO, lockless, multi-producer and multi-consumer data structure, enabling fast transfer of packets across different cores.
- **rte_hash:** [5] DPDK provides an optimized hash library for fast data lookup, update and delete.

Poll Mode Drivers (PMD) are userspace drivers in DPDK continuously polling the NIC for any incoming packets, thus preventing the need of per packet interrupts as in the case of kernel TCP/IP stack. DPDK applications can use the APIs provided by the DPDK libraries without any system calls, thus having no overhead of context switching associated with syscalls.

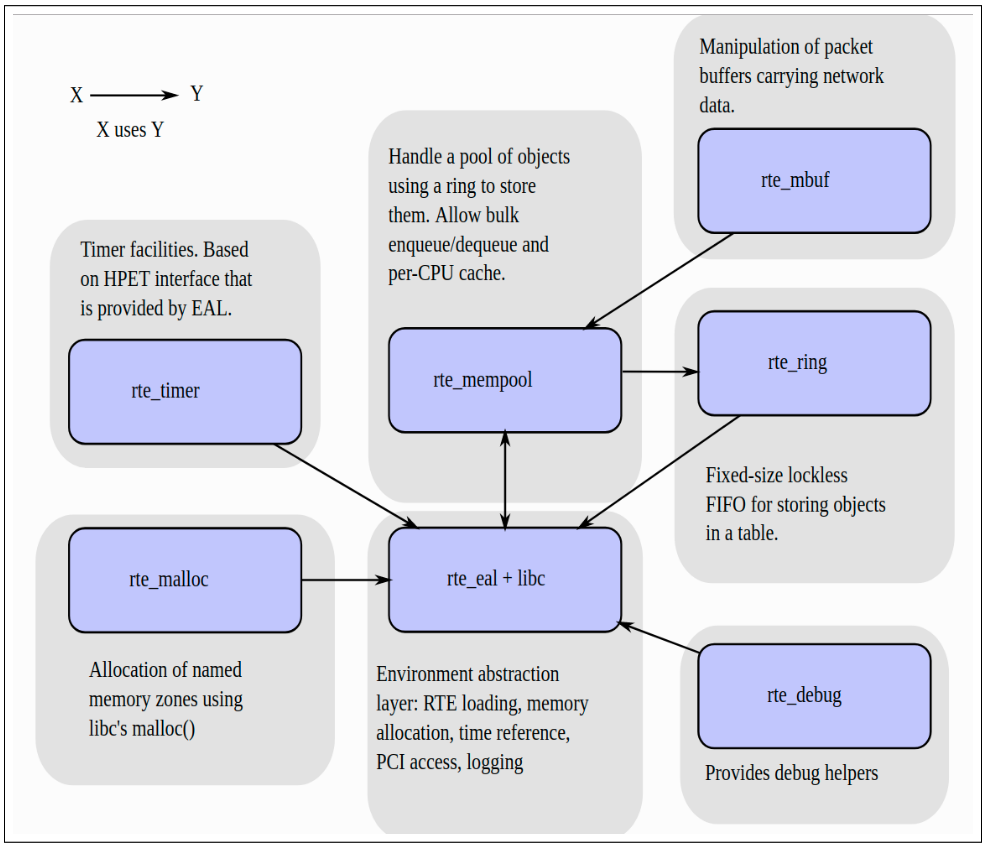

X → Y
X uses Y
Manipulation of packet buffers carrying network data.
rte_mbuf
Handle a pool of objects using a ring to store them. Allow bulk enqueue/dequeue and per-CPU cache.
rte_mempool
rte_ring
Timer facilities. Based on HPET interface that is provided by EAL.
rte_timer
Fixed-size lockless FIFO for storing objects in a table.
rte_malloc
rte_eal + libc
Allocation of named memory zones using libc's malloc()
Environment abstraction layer: RTE loading, memory allocation, time reference, PCI access, logging
rte_debug
Provides debug helpers

Figure 4.1: DPDK core components[1]

4.2 Kernel bypass design

The following subsections elaborate on how the DPDK-based UPF is designed and implemented. Although the title reads kernel bypass design, the TCP/IP stack of the kernel is still used for processing infrequent packets as well as for processing packets whose headers are complex (e.g., HTTP running over TCP), as can be seen below. However, there is scope in the future to shift processing of such packets also to the userspace, to bypass the kernel entirely.

4.2.1 ARP request and response

The UPF needs to register with the NRF as a part of the publish-subscribe model, wherein the UPF publishes that it is available as an NF and other NFs can subscribe to its services. However, before registration, the UPF needs to know where the other NFs are (NRF, SMF, RAN and DNN). So, it sends broadcast ARP requests for all the said NFs (except SMF) and updates its ARP table locally. Since the kernel TCP/IP stack is bypassed, ARP requests and responses need to be processed in the userspace. Also, as the SMF itself contacts the UPF during session setup, the UPF does not proactively send ARP requests to the SMF. Instead, when SMF broadcasts an ARP message to find the UPF, the UPF sends an appropriate reply and updates its ARP table.

The DPDK-based UPF has 2 ARP threads running on a dedicated core to handle

[1]Figure taken from [1]

ARP requests and responses respectively. Once an ARP message is received, the receiving core redirects the message to one of the threads (based on the message type) via s/w rings. For incoming ARP requests, the request-handling thread crafts an ARP reply and sends it out. Otherwise, the UPF has received an ARP reply for a request that itself had sent out, and so updates the ARP table. The ARP message processing threads keep on running in the background for the entire lifetime of the UPF to handle any future ARP request or responses (for example, gratuitous ARP messages).

4.2.2 Registering with NRF

The UPF sends an HTTP registration request to the NRF. Since processing TCP headers is much more complicated than UDP headers and these type of messages only appear at the start and once in every 100 seconds after that (as heartbeat messages), we let the kernel TCP/IP stack handle processing of those messages. We use a TUN device for redirecting the packets between the userspace and the kernel. As these messages are very few, it does not hamper performance of any kind, while taking care of the complexities associated with handling TCP headers (for instance, checking for correct sequence numbers, handling packet re-transmissions etc.). The TUN device is created when the UPF starts up, before the registration process begins.

When the UPF sends an HTTP registration request, as the NIC is already bound to DPDK, the message cannot go out after TCP/IP processing by the kernel stack. So, we reroute the packet to the TUN device after the kernel adds the TCP and IP headers, and read the packet from the TUN device in the userspace. Then, we add the Ethernet header and send it out with the help of APIs provided by DPDK. The MAC of NRF is read from the ARP table and put as the destination MAC in the Layer-2 header. On receiving a reply from the NRF, DPDK hands over a Layer-2 packet to the userspace for processing. We discard the Ethernet header and write the IP packet into the TUN device, which redirects the packet back to the kernel. Thus, the kernel processes the IP and TCP headers before our UPF application gets the HTTP response and act accordingly.

4.2.3 Handling Control Plane Messages

When a UDP packet is received in the userspace, it processes the UDP header to see the destination port field. Control Plane (CP) packets will have destination port as 8805, which is the dedicated PFCP port. All such packets are enqueued into a software ring dedicated to inter-core communication between the receiving core and the CP core. The CP core continuously checks whether there is a packet in the software ring and dequeues any such packet. Outer (L2-L4) headers are discarded so that now we have a PFCP packet. Peer information is collected from and saved for each such packet (that is, a unique peer token along with the IP and UDP port of the SMF which sent this message). This information is looked up while sending a corresponding reply so that the reply reaches the correct SMF.

Once peer information is saved, the PFCP message is decoded to see if it is a node related message or a session related message. Node related messages include PFCP association setup/update/release, while session-related messages include session establishment, modification, or deletion messages. A detailed description of node and session related PFCP messages is already provided in Section 3.2. If the PFCP message is a PFCP association setup request from the SMF (currently, only this is supported), the CP core extracts the necessary Information Elements (IE) from the message and triggers a PFCP association setup response where a PFCP reply message is crafted. Since the kernel is bypassed, the CP core creates and adds the Ethernet, IP and UDP headers, with correct information from the peer token in the userspace itself, and sends the packet out via APIs provide by DPDK. The destination MAC address in the Ethernet header is that of SMF, which is read from the ARP table. For session establishment, update and modification messages, first the session ID (SEID) is extracted from the PFCP header. If it does not match any existing session ids and the message is not a session establishment message, the packet is discarded. Otherwise, based on session id and message type, PFCP IE is extracted, and reply messages are crafted and sent out like the PFCP association response message. Figure 4.2 shows the CP packet handling flow.

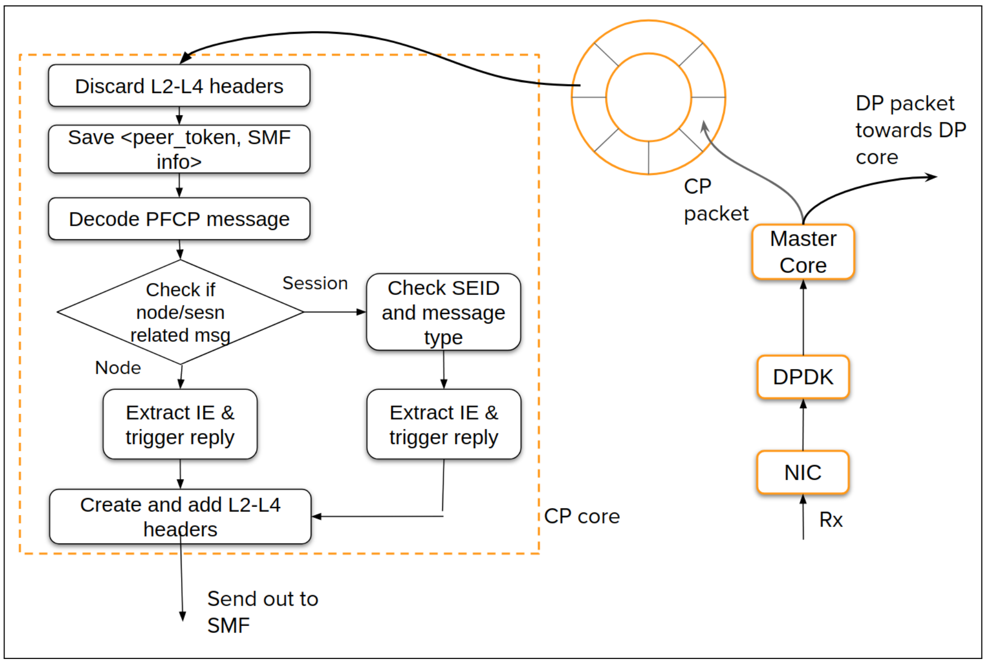

Discard L2-L4 headers
Save <peer_token, SMF info>
Decode PFCP message
Check if node/sesn related msg
Session
Node
Check SEID and message type
Extract IE & trigger reply
Extract IE & trigger reply
Create and add L2-L4 headers
Send out to SMF
CP core
DP packet towards DP core
CP packet
Master Core
DPDK
NIC
Rx

Figure 4.2: Control Plane packet flow[2]

Current DPDK-based implementation only has one core dedicated to handling control plane messages as they are few and far between (in the current design, control plane messages are passed between UPF and SMF only during session establishment and later if we want to tear down a PDU session). However, the number of dedicated Control plane cores can be increased easily with the DPDK-based design if and when required.

4.2.4 Designing the Data Plane

The current DPDK based UPF supports both the Run-to-completion (RTC) as well as the Pipeline based design for the data plane.

- **Run-to-Completion (RTC) design:** In RTC, each core receives a batch of packets from the NIC. Once received in the userspace, the core processes the batch of packets and sends them out. Only then the next batch of packets are

[2]PFCP encode/decode, IE check and crafting PFCP reply are done by the project engineers.

polled from the NIC and brought into the userspace. Scaling to multiple cores is easy. The same code should be launched on multiple cores, each core polling on a different RX queue of the NIC and transmitting via different TX queues for maximum performance. Since packets should be already distributed to different cores before they reach the userspace, h/w (NIC) support is required. More precisely, RSS should be applied on the packets in the NIC itself to redirect the packets to separate cores. For downlink packets, the RSS hash is on the IP destination address (i.e., the UE IP). For the uplink, currently, the hash function is applied on the outer UDP header. It is assumed (and implemented in RAN) that the source port of the outer UDP header is different for different UEs. Ideally, fields from the GTP or the inner IP header should be taken. However, the outer UDP header is considered as the NIC used in the experiments does not support RSS hash on GTP/inner headers. Outer IP source/destination address will be the same for all incoming packets in the UPF (source = RAN IP, destination = UPF IP) due to the GTP tunnelling used. Hence, the outer IP header cannot be considered for RSS. Figure 4.3 shows the RTC design with an uplink packet.

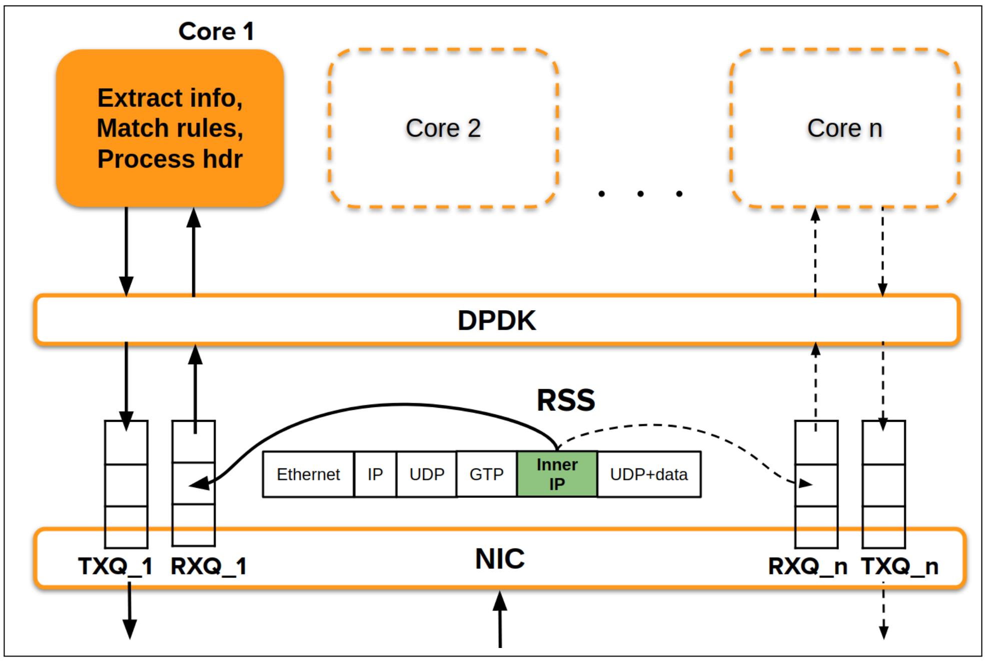


Figure 4.3: DPDK RTC design

- **Pipeline design:** In this design, a set of cores (henceforth called master cores) receive packets from the NIC. Once the packets are in the userspace, the master cores redirect the packets to a set of worker cores for processing via s/w rings provided by DPDK. This enables the master cores to receive more packets while the worker cores are processing the packets and transmitting them out. The design can be made so that the worker cores can themselves use s/w rings to transfer packets to another set of worker cores if the processing itself has multiple stages (can be visualised as stages of a pipeline). The bottleneck then is the slowest stage of pipeline processing. Apart from being able to poll continuously, another advantage is that no h/w support is required in this design. Although RSS on NIC is preferable, RSS can be done in the userspace itself if the NIC

does not support so. However, enqueuing and dequeuing from the rings incur an extra overhead not present in the RTC model, and this overhead is significant especially if the enqueue/dequeue is done for individual packets instead of in batches.
The current pipeline based design uses s/w based RSS once packets are received in the userspace. For uplink packets, the inner IP source address (i.e., the UE IP) is used to redirect packets to worker cores. For downlink packets, the IP destination address (again, the UE IP) is used for the same purpose. One point to note is that in the pipeline design, both h/w and s/w based RSS can also be used together when there are n master cores. This is to further divide a set of UEs received in a master core and redirect each such subset to different worker cores. Figure 4.4 shows the pipeline design with an uplink packet.

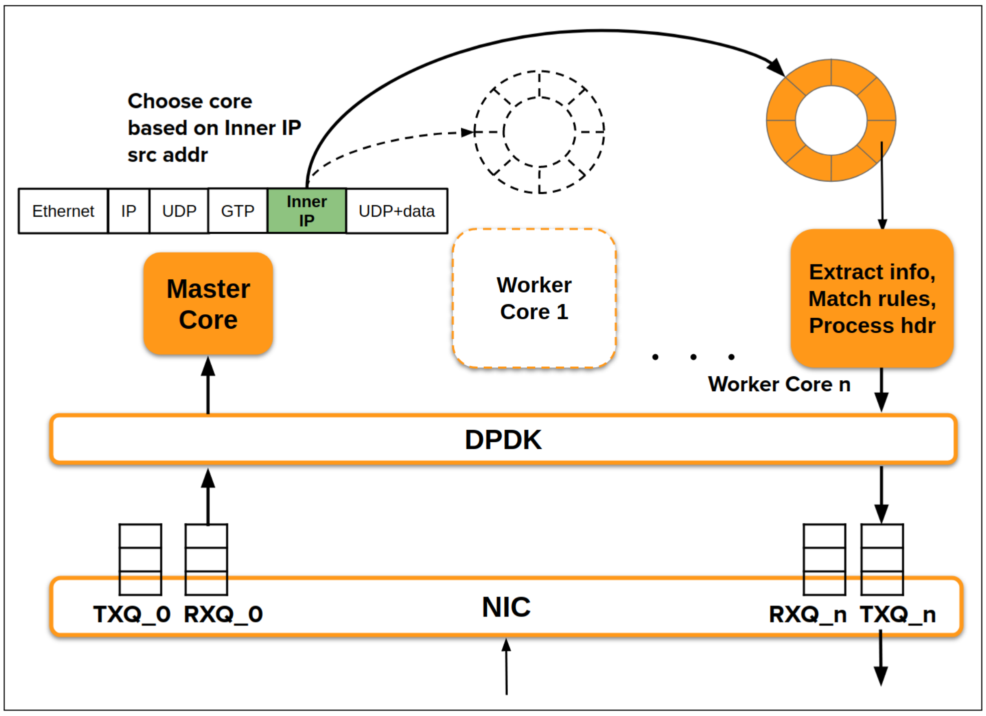


Figure 4.4: DPDK pipeline design

4.2.5 Handling Data Plane Messages

The current setup has the UPF machine connected to the RAN machine via 1 NIC port and to the DNN machine via another port of the same NIC. (In this report, the terms NIC port and NIC interface are used interchangeably). So, another CPU core (let us call it DNN core) is required in the UPF for receiving downlink Data Plane (DP) packets from the DNN, along with the master core, which receives uplink DP packets from the RAN. The DPDK API to receive packets (***rte_eth_rx_burst()***) maps to a *(NIC port, NIC RX queue)* pair and requires a whole CPU core to itself as it is continuously polling the NIC port. Performance is degraded if a core is shared between 2 such APIs, even if they are on separate threads (here, we need two calls because 2 NIC ports are being polled simultaneously). Hence, the design includes an additional dedicated CPU core.

In the **RTC based design**, once a core receives a packet, it needs to be seen whether the packet has correct checksum before proceeding with anything else. Since checksum verification is offloaded to the NIC, when a packet is received in the userspace, the packet metadata has information on the IP and UDP (wherever applicable) checksums. If the checksums are wrong, the packet is discarded. Otherwise, it is processed. One advantage of using the 2 NIC-port setup is that in the RTC design, the receiving core can determine whether the packet is an uplink/downlink packet without any hassle, based on the NIC it is polling. Thus, no further checks are required after checksum verification.

- **Processing Uplink Packets:**
  If the packet is an uplink packet, it will be a GTP packet. So, we extract Packet Detection Information (PDI) from the GTP and the inner IP headers. PDI is information embedded both in the packet and the PDR. A packet matching a PDR means that the PDI of the packet is the same as the PDI of that particular PDR. PDI fields are the GTP tunnel identifier (TEID), the source IP address in the inner IP header (i.e., UE IP), QoS Flow Identifier (QFI) and the source interface, that is, the interface from which the packet is coming. Source interface will be *Access* if the packet comes from RAN (access network), and *Core* if it is coming from another UPF (5G core network). The current implementation supports only 1 UPF, so an uplink packet always has source interface as *Access*. Once PDI is obtained for a packet, it will be matched against a set of PDRs. On finding a match, corresponding FAR and QER are obtained, and the packet is processed accordingly. The procedure is already shown in Figure 3.4 while describing the kernel-based implementation. For GTP packets (as per current implementation), GTP header should be removed, and the inner IP packet should be forwarded to the DNN by adding an Ethernet header to it. Note that no TUN device is required here. If no PDR is matched, or for a particular PDR there is no matching FAR, the packet is silently discarded.

- **Processing Downlink Packets:**
  For downlink packets also, we first extract the PDI. The only PDI field in the downlink packets is the destination IP address (i.e., UE IP). Packets coming from the DNN will not have GTP header. So, TEID and QFI fields are absent. Here, source interface will always be *Core* irrespective of whether it is coming from the DNN or another UPF. The PDI is matched against a set of PDRs, and corresponding FAR and QER is applied on the packet (again, silently discarded if no match is found). For downlink packets coming from DNN, FAR includes creating an outer GTP header and sending the packet to RAN in the current implementation. GTP header for downlink packets also has a GTP extension header appended to it, within which the QFI obtained from the matched PDR is encapsulated. This QFI value is propagated to the UE via the RAN.
  Figure 4.5 shows how a downlink packet is processed in the DPDK-based design. The FAR obtained tells to create a GTP header with TEID equal to the RAN side TEID. Then, the outer IP and UDP headers are created and added in the userspace itself. Initially, the UDP checksum is 0, which is then calculated from the pseudo (outer) IP header, the UDP header and the outer UDP payload (GTP header + inner IP packet). Since the UDP payload can be huge, calculating checksum trivially can become a bottleneck and hinder performance. Hence, the checksum calculation has been offloaded to the NIC. Finally, the Ethernet header is added to the packet before sending out to the

RAN via DPDK.

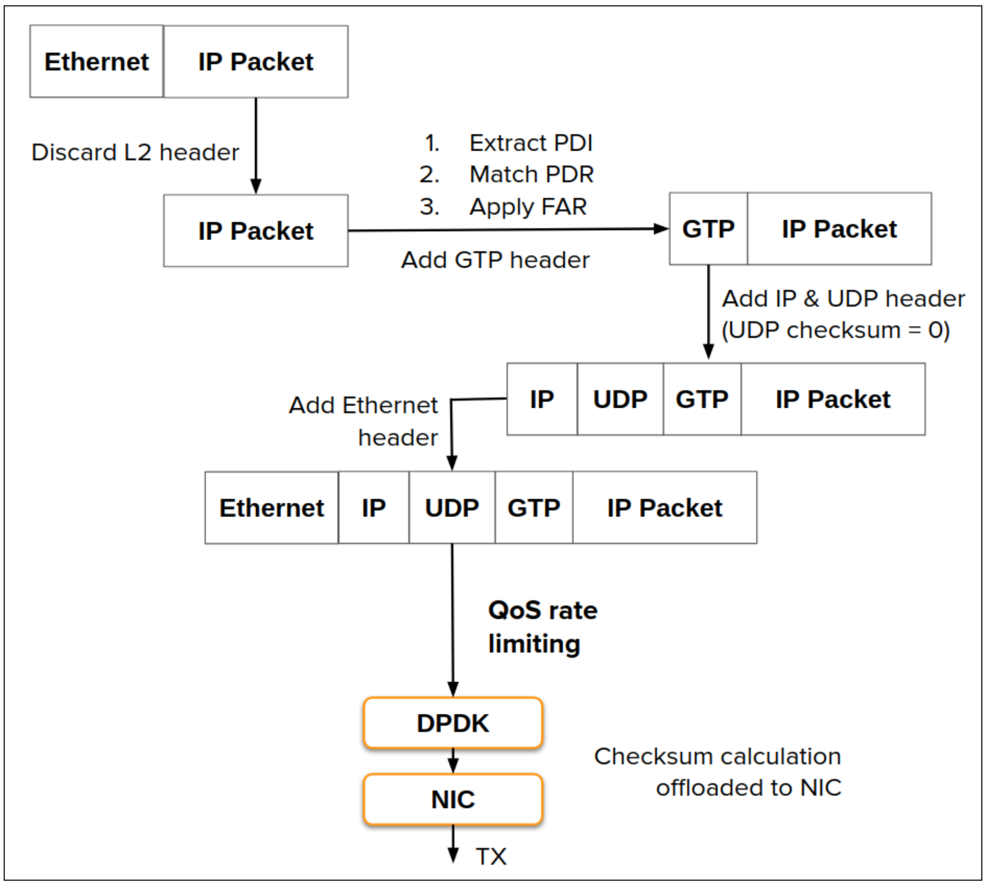


Figure 4.5: Downlink packet processing

In the **pipeline based design**, the master core checks the destination port field of the outer UDP header and sends it to 1 of the n worker cores dedicated to handling DP messages if the port number is 2152 (which is the dedicated GTP port). The DNN core checks if the packet is addressed to a UE before redirecting the packet to a worker core. The current design has one software ring per worker core, for inter-core communication between that core and the receiving core. The master/DNN core distributes the packets to each of the n worker cores based on the UE IP so that all packets from a UE are redirected to the same DP core. On receiving a DP packet, each worker core verifies the packet checksum (using information from the packet metadata) and discards the packet if the checksum is invalid. Otherwise, the packet is processed. Since a worker core can receive packets from both the master and the DNN core, it has to additionally check whether the packet is an uplink or a downlink packet by checking the source IP address — uplink packets will have source address equal to the RAN IP while downlink packets will have that equal to the DNN IP — and handles it accordingly. After the type of packet has been determined, processing the packet is exactly same as in the RTC design, as enumerated above. Figure 4.6 shows a simplified flow for the DP packets in the pipeline design (DP core = Worker core).

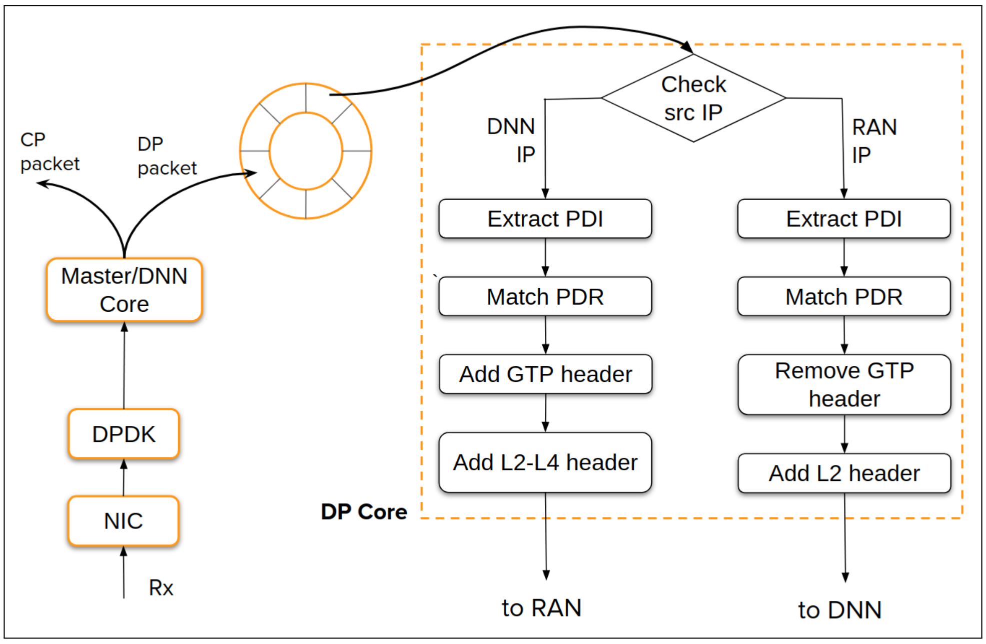


Figure 4.6: Data Plane packet flow in pipeline design

The current implementation does not create a new packet for forwarding. Instead, in both the designs, it manipulates the packet received from the RAN/DNN by changing its metadata and data-offset pointer (refer Figure 4.7) to overwrite the original contents of the packet with new headers in the packet headroom. As a result, the packet copying overhead is removed, which improves the performance.

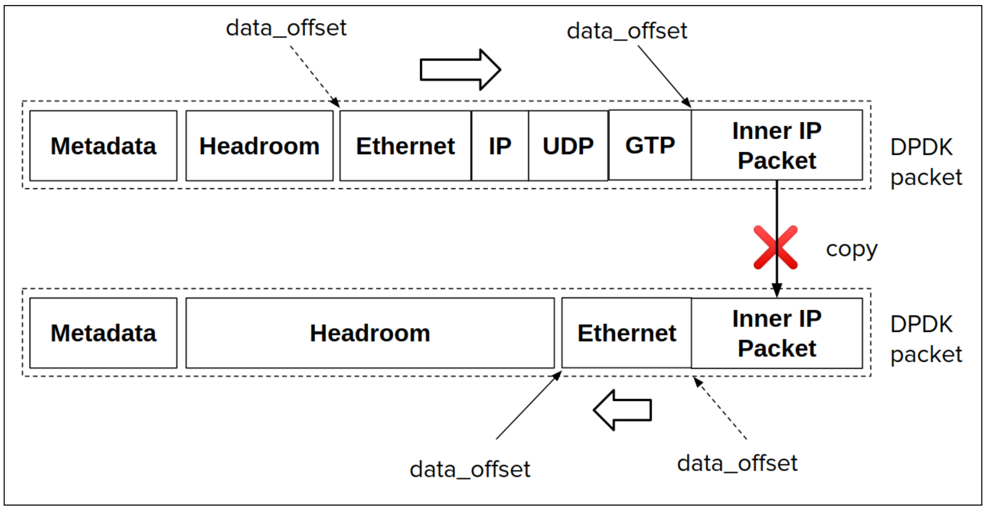


Figure 4.7: Reusing received packet

In both the designs, once the required packet is crafted, QoS is enforced on it, and it is sent to the transmit queue of the corresponding NIC port via ***rte_eth_tx_burst()*** API provided by DPDK from where the NIC can send it out (rate-limiting and QoS are discussed in detail in Section 5.2). There is an one-to-one mapping between the DP cores and the TX queues of the NIC, that is, DP core number i sends the packet to the i^{th} transmit queue of the NIC port, as shown in Figure 4.8. This $1:1$ mapping enables multiple cores to write into the TX queues of the NIC port concurrently without requiring any locks and allows both the designs to scale well with the increasing number of cores without degrading packet forwarding performance.

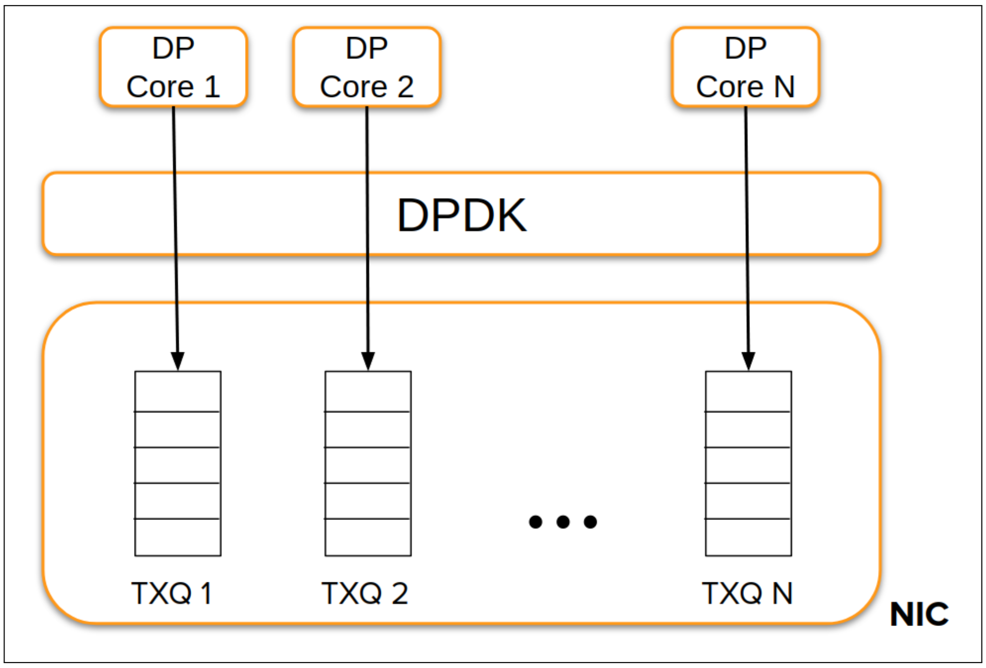


Figure 4.8: Core to TX queue mapping

4.3 Challenges

4.3.1 Design Choices

Following are the design and implementation choices that were taken while developing the DPDK-based UPF:

- **Run-to-completion vs. pipelining model:**
  DPDK applications can either be designed as a run-to-completion (RTC) model or as a pipelined model, as described in Section 4.2.5.
  While choosing a specific design will depend on the type of application or workload, table 4.1 summarises the pros and cons of each design to help developers choose the correct design as per requirement. Both the designs are easily scalable. In RTC, h/w support (for example, RSS on NIC) is necessary to receive packets in multiple RX queues of the NIC. There are no such restrictions in the pipeline design. If there is no RSS support on NIC, then one master core can receive the packets and distribute them to the worker cores in the userspace. However, the pipeline design has an extra overhead of en-queuing to and de-queuing from s/w rings.
  Additionally, in the pipeline design, m master cores receive the packets while n worker cores process them. In contrast, RTC can use all the $m + n$ cores to both receive and process packets. However, if packet processing is CPU intensive, RTC may receive a less number of packets because the time to process packets has now increased, which may lead to decreased per UE performance. For example, say RSS is used at the NIC level, and 2 UEs are redirected to the same CPU core, which can process packets at the rate of x $Mpps$ (and hence, receive at the same rate). Assuming uniform distribution between the 2 UEs, RTC design can now process $\frac{x}{2}$ $Mpps$ per UE. On the other hand, the pipeline

model has provision to further redirect the packets in the userspace and can distribute packets from these 2 UEs to 2 separate worker cores. As a result, both receiving and processing capability of the UPF (on a per UE basis) will be more than that of the RTC design, since each worker core can separately process $x\ Mpps$.

	RTC	Pipeline
Pros	1. Easily scalable. 2. Less userspace processing overhead. Throughput expected to be higher.	1. Easily scalable. 2. No h/w support needed to distribute packets to other cores.
Cons	1. H/W support needed (eg. for RSS) 2. Overall (per UE) performance may decrease if processing is CPU intensive.	1. More userspace processing overhead (ring enqueue-dequeue, s/w RSS). 2. More cores needed overall (m master + n worker cores).

Table 4.1: RTC vs Pipeline design — pros and cons

- **Optimizing packet processing for multiple sessions:**
  Since millions of UEs can send data simultaneously, with each UE having at least 1 PDU session, the packet processing algorithm needs to be designed carefully. The brute force design is that for each incoming packet, we traverse all sessions to fetch the correct one for a particular UE, and all PDRs within that session to get the correct PDR. This is an $O(nm)$ algorithm (n = no. of sessions, m = no. of PDRs in a session). As a result, performance will keep on decreasing as the no. of sessions (or, no. of PDRs in a session) increase and packet processing will become the bottleneck.

  The next few lines talk about a better algorithm that is implemented currently, to prevent the above scenario. When sessions are established or modified, we keep a *<PDI, SEID>* hashmap, where SEID is the unique identifier for any session and PDI is information obtained from any PDR installed in that session. Thus, if a session has 10 PDRs, there will be 10 *<PDI, SEID>* entries in the map with the same SEID. PDIs are unique both within and across sessions since $3GPP$ specification mentions that all PDRs should be non-overlapping, i.e., there will not be 2 PDIs having all fields same. Section 3.3.2 has details on the fields of a PDI. Thus, it can act as the key in the hashmap. Entries in this map are updated/deleted if any PDR is updated/deleted.

  When the first packet for any particular UE (/session) arrives at the UPF, the above hashmap is searched with the packet PDI. As the packet PDI for any legitimate packet must match the PDR's PDI, we get the correct SEID in $O(1)$. A packet not matching with any of the PDIs in the hashmap implies that the packet does not belong to any of the already established/active sessions. Such packets are discarded immediately. Once SEID is found, all PDRs in that session are traversed to get the exact PDR with which this packet matched along with the corresponding FAR and QER. This information is then stored in a new hashmap with PDI as the key, and a structure containing all the relevant information as the value. In the current implementation, the structure contains PDR, FAR, QER, SEID and the Latest Timestamp (LTS) for any packet in that particular session (Section 5.1 gives details about LTS). Currently, the PDIs are an exact match as there are no wildcard fields. However, later when wildcards will be introduced (for example, PDR's PDI field is a range of IP addresses and any packet PDI having IP within this range is considered a match), the algorithm and data structures need to be modified accordingly.

From the second packet onwards, this second hashmap is searched instead of traversing through all the PDRs every time. Thus, relevant FAR, QERs and other details are also obtained in $O(1)$, making the overall packet processing time complexity $O(1)$. Hence, packet processing is now independent of the no. of sessions or the no. of PDRs in a session, and the UPF performance will not be affected even with millions of sessions. This hashmap is also updated when any session is modified (for example, modifying a PDR in that session), or released.

Figure 4.9 represents a simplified view of the said algorithm.

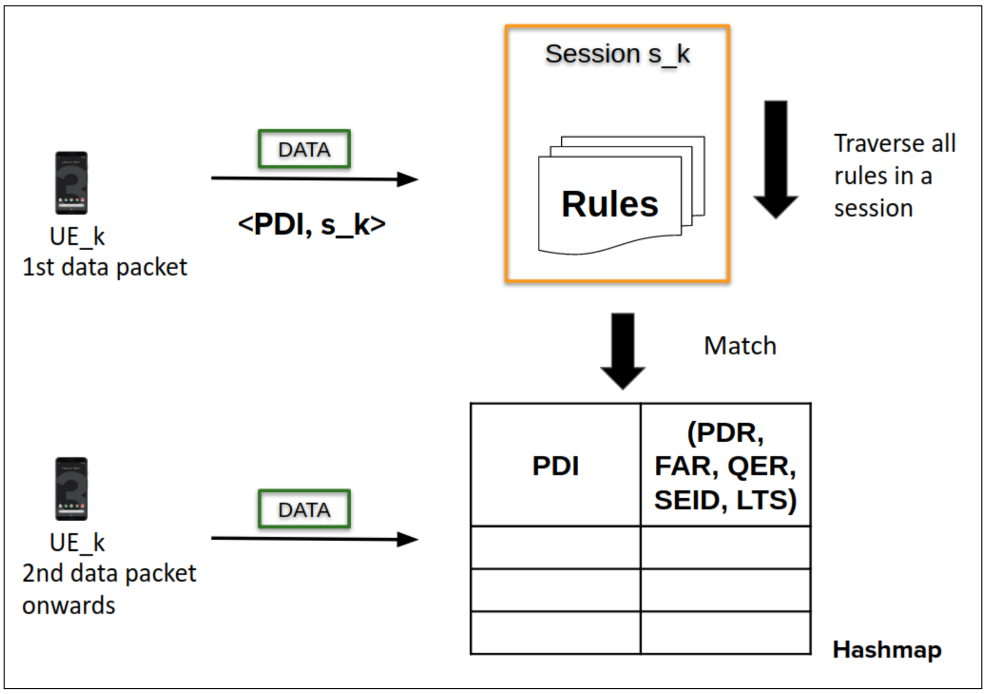


Figure 4.9: CP packet handling design

- **CP packet processing — TUN vs S/W Ring:**
  Since current implementation has CP messages only during PDU session establishment and teardown, a TUN device can be used to redirect those messages

to the kernel where the TCP/IP stack will process the headers, just like during NRF registration. However, the current implementation uses the ring-based design where CP packets are transferred to a dedicated core and are processed in the userspace itself, bypassing the kernel. The ring-based design ensures that the kernel/TUN does not become a bottleneck in scenarios in the future where the number of CP messages may be comparable to the number of DP messages, say in case of some IoT devices. The ring-based design can handle such scenarios and scale up if required. Figure 4.10 shows the design options.

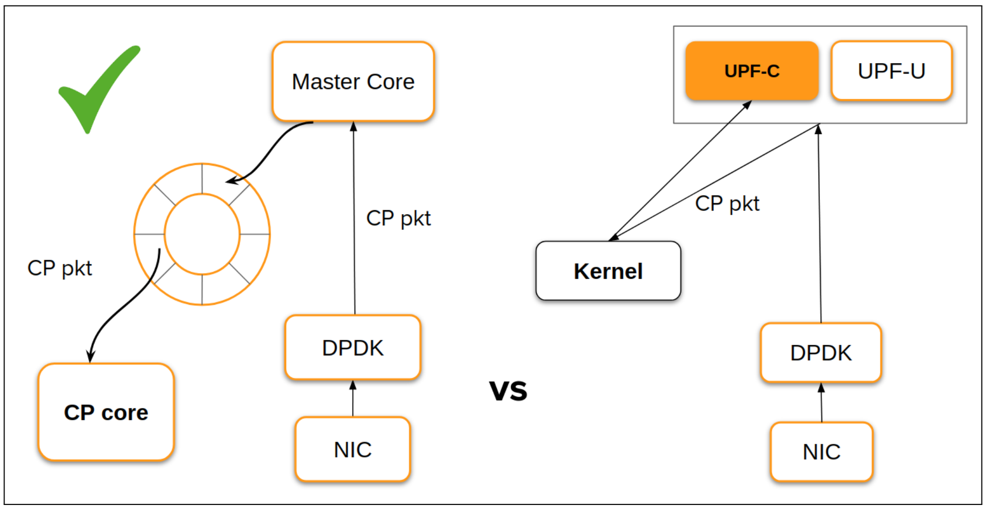


Figure 4.10: CP packet handling design

4.3.2 Scalability across NUMA nodes

Although the design is scalable across multiple cores, scaling across multiple NUMA nodes may become an issue, because communication between memory region associated with 1 NUMA node and the remote memory region associated with another NUMA node becomes a bottleneck. One way is to disable the NUMA architecture, but then the overall throughput in terms of packet forwarding rate is slower. Another solution is to use two cores in different NUMA nodes for receiving incoming packets and communicate only within the NUMA nodes, leaving inter-NUMA communication to a minimum. Currently, these design choices are still being explored and will be implemented if and when required.

4.3.3 Implementation challenges

Implementation challenges include learning to use the libraries and APIs provided by DPDK, bugs while coding, compiler issues, bottlenecks due to un-optimized code (removed via profiling), networking issues etc. Refer to Appendix A for a detailed list of such issues.

5. Enforcing Quality of Service (QoS)

QoS is an essential part of any real-life UPF. Users subscribe to different data plans and based on that they may want different customised services like higher bandwidth, guaranteed minimum speed even if the network is congested, and so on. Users paying more subscription fees will demand a higher bit-rate while users paying less may be satisfied with lower speeds.
As per *3GPP* specifications, the 5G QoS model supports both QoS flows that are capped at a specific bit-rate with no guaranteed minimum (Non-GBR) as well as flows that require a guaranteed minimum bit-rate (GBR). The QoS flows are differentiated by QoS Flow Identifier (QFI), which is unique within a PDU session. GBR and non-GBR flows have different QFIs. The QFI is encapsulated within the GTP extension header. Within a session, any UE packet with the same QFI receives the same QoS treatment. Any PDU session will have a QoS flow associated with the default QoS rule established throughout the session, which limits the aggregated maximum bit-rate within that session (Session-AMBR). This flow will be a non-GBR QoS flow. The current UPF implementation only supports one non-GBR flow which follows the default QoS Enforcement Rule (QER) for the session (1 QER for uplink packets, 1 for downlink packets). Of course, different UEs (having different sessions) will have separate QERs.

5.1 State of the art QoS

Usually, QoS implementations have one queue per flow for rate limiting. For each flow, if the incoming rate in the UPF is less than the allowed outgoing bitrate, packets are transmitted instantaneously. Otherwise, packets are queued and transmitted over a period, which allows the UPF to limit the rate for that particular flow. If the incoming rate is such that this queue becomes full, then further packets are dropped by the UPF. The problem with this approach is that when there are millions of UEs each having at least one QoS flow, millions of queues need to be traversed, which causes a massive overhead. As a result, the UPF performance degrades so much so that it cannot saturate the line rate. Saeed Ahmed et al. in their paper Carousel [16] propose a solution to this problem. Instead of millions of queues, they maintain only one queue for all flows. Figure 5.1 shows a simple representation of the queue and how the incoming packets are stored in it.
The paper uses certain terminologies, which are explained below for clarity:

- Timing Wheel (TW): The queue which stores the packets to enforce rate limiting. Each element/index in this TW denotes a time slot. A list of packets from different QoS flows is queued in each index of the TW. All packets in the current time slot of the TW are dequeued (FIFO order) and sent out via the NIC.

- Latest Timestamp (LTS): The time at which a certain packet needs to be dequeued from the TW and sent out. If the length of the packet is denoted by pkt_len and the outgoing rate limit (i.e., Session AMBR) is pkt_rate, then

$$LTS_{next\_pkt} = LTS_{cur\_pkt} + \frac{pkt\_len}{pkt\_rate} \tag{5.1}$$

- Slot Granularity (G): It is the time range each slot represents.
- Horizon: The furthest time till which packets will be queued. Any packets with LTS beyond the horizon are dropped. Hence,

$$\#TW\_slots = \frac{Horizon}{G} \tag{5.2}$$

- Front Timespatmp (FTS): FTS denotes the current time slot in the TW. Increasing FTS by 1 means time has progressed by G units.

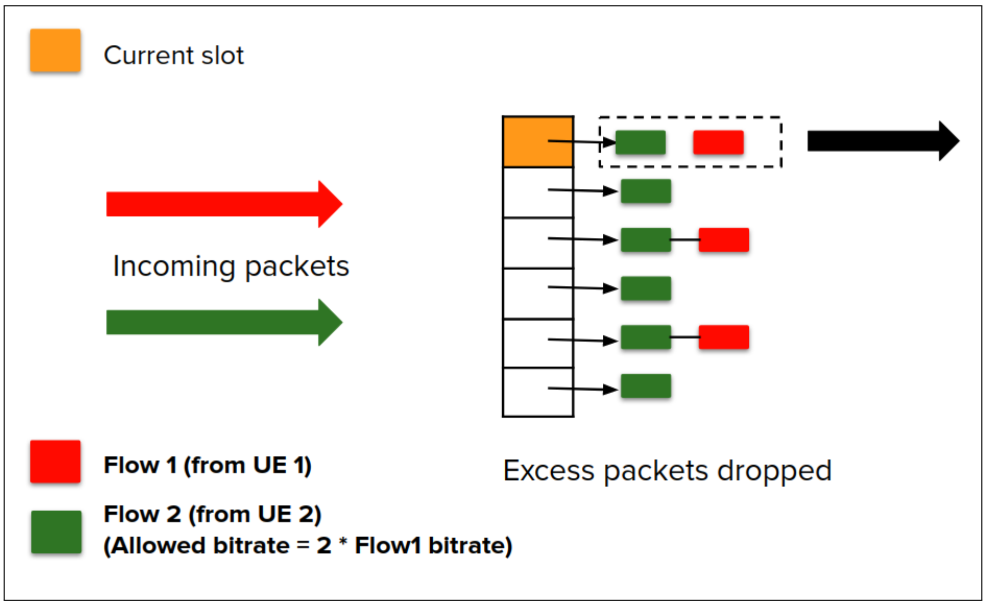


Figure 5.1: Carousel architecture

In Carousel [16], packets are inserted in slot number $(\frac{LTS}{G})\%(\#TW\_slots)$, provided LTS is between the FTS and the $Horizon$. For packets with $LTS < FTS$, LTS is set to FTS, i.e., they are inserted in the current slot. As shown in Figure 5.1, if a QoS flow has outgoing bit-rate double that of another flow, the first flow will have double the number of packets in the TW.
Packets are extracted every G period, and the next slot becomes the current one. The TW is a circular representation of time, i.e., once all packets are dequeued from the slot and it becomes an older one, that particular slot now represents the time $(cur\_time + Horizon)$.

5.2 QoS in the DPDK based UPF

The current QoS implementation in the UPF redirects all flows from a particular UE to a single core via RSS, to ensure that the performance does not degrade due

to inter-core communication in order to maintain the rate limit for a particular UE across cores. Of course, flows from 2 UEs may be directed to separate cores. Each core has a separate TW and separate FTS values to decouple one core from another and ensure scalability. Within a core, whenever a packet comes, it is assigned an LTS value (after processing PDR and FAR rules) depending on the aggregated maximum allowed bit-rate for that particular session/UE (session-AMBR). Once LTS is assigned to a packet, the LTS is checked against the FTS for that core. If $LTS <= FTS$, it implies that the packet can be sent out immediately, and it is added to the current slot of the TW to be sent out as a part of a batch next time *extract()* is called. In the current implementation, *extract()* is called after every 32 packets, or periodically every 100 us if no packets are coming. This periodical call ensures that any residual packets in the TW are sent out as well as the FTS remains up-to-date with the current time. Further details about *extract()* are explained later in this section.
If $LTS > Horizon$, where $Horizon = FTS + \#TW\_slots$, it means that the packet's transmission time is too far ahead in the future to be queued. This implies that the incoming rate is too high and the sender (UE) should slow down its transmitting rate as well as resend that particular packet, as the UPF drops these types of packets. If for a packet $LTS > FTS$ but within the $Horizon$, then such packets are queued and sent out at the allowed outgoing rate.

When the extract function (*extract()*) is called, all packets in the current slot are dequeued and sent out in a batch to the RAN or the DNN. If the FTS is behind the current time by at least G (slot granularity, explained in Section 5.1), then FTS is updated to the current time. Sometimes, a particular slot in the TW can be full due to too many packets being enqueued in that slot. If another packet comes which needs to be inserted in the same slot, then the *extract*() is called with an $URGENT$ flag and the respective slot number. All packets in that particular slot are then dequeued before inserting the new packet. As a result, there may be a slight rise in the outgoing rate, which may overshoot the rate limit for an instant. This is because the packets that were supposed to be sent at a particular time in the future (within the $Horizon$ value) are being sent at the current time. However, not emptying the slot would lead to no room for newer packets, and the outgoing rate may be decreased for an instant in the future. This is a trade-off, and future versions of the UPF QoS implementation need to handle this and design a better algorithm.

6. Evaluation

6.1 Experimental Setup

Figure 6.1 shows the experimental setup. All three servers have $10G$ NICs and are connected as shown. 1 NIC port of Server 2 is connected via a SFP cable to Server 1, and the other to Server 3. Server 2 hosts the UPF while Servers 1 and 3 host the RAN and the DNN respectively. All other NFs, namely the AMF, SMF, NRF, AUSF, UDM and UDR, are also on Server 1.

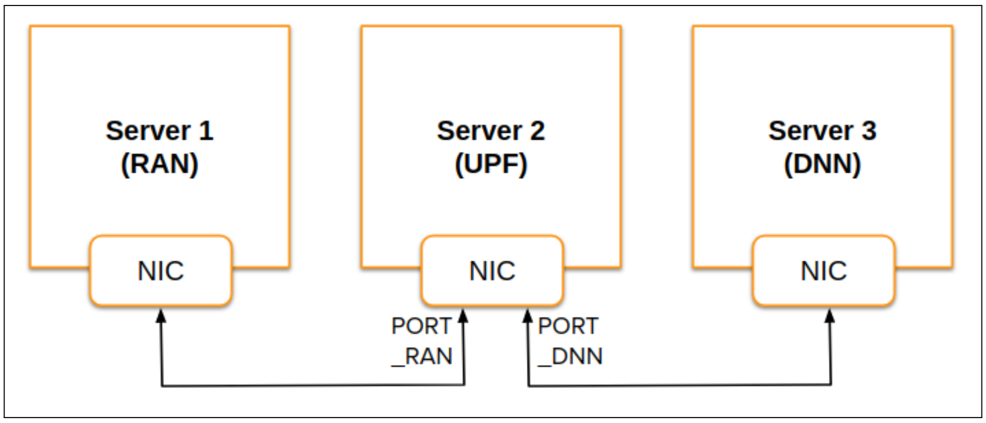


Figure 6.1: Experimental Setup

The server configurations are as follows:

- Intel Xeon(R) CPU $E5 - 2650\ v4$ @ $2.20GHz$ (24 CPU cores)
- 128 GB RAM and 2 TB HDD
- $10G$ NIC connected via cable (P2P)

8192 hugepages are allocated per NUMA node in Server 2 while loading DPDK. Server 3 runs DNN over DPDK. It has two modes — sink and echo. In sink mode, the DNN only receives packets and shows the throughput. In echo mode, the DNN mirrors the received packets back to the UPF. Echo mode simulates duplex connection, where both uplink and downlink traffic is arriving at the UPF simultaneously. In Server 1, the RAN is also DPDK-based. The DPDK-based RAN generates packets of different payload sizes at different rates from multiple UEs, as and when required.

In Server 2, there is one master core which receives packets from the RAN and another core which listens for packets coming from the DNN. One core is dedicated to CP processing, one for other miscellaneous functions (TUN handling, Heartbeat timer handling, ARP handling). The remaining cores can be used for processing Data Plane packets in the pipeline design. In contrast, in the RTC design, the two receiving cores themselves process DP packets and more cores receive (and process) packets from other RX queues of the NICs when required.

6.2 Results

This section shows how the DPDK based UPF performs for different sized packets in the uplink and downlink. It also shows how the UPF scales across cores, how it performs in a multi-UE setup, and finally how both the DPDK based UPF designs perform against other UPF designs. In essence, this section tries to answer the following questions:

1. How does DPDK based UPF perform for different payloads? When is it saturating the 10G line rate?
2. Are the designs (and implementations) scalable across cores?
3. How is the UPF performance affected with an increasing number of UEs/sessions?
4. Is QoS functioning correctly? How much overhead does the QoS implementation add to the UPF?
5. How does DPDK based UPF fare against other UPFs?

6.2.1 Single core performance

All experiments in this section assume that the incoming rate is always less than the session-AMBR (session-AMBR set to 10 $Gbps$). The aim is to determine the maximum achievable performance of the UPF in such a scenario.

- **Single UE throughput:**
  Figures 6.2 and 6.3 show the single-core uplink and downlink performance of the DPDK based UPF for payloads of different sizes (single UE). Payloads from 64 B to 1400 B are used for this experiment.

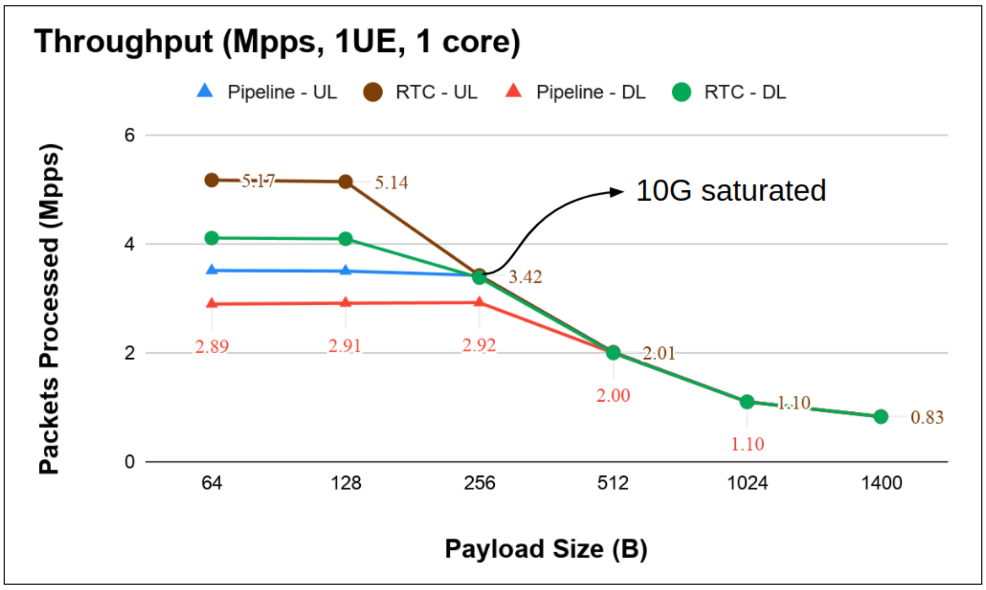


Figure 6.2: Single Core throughput (Mpps) for various payload sizes

Figure 6.2 shows how many packets are processed by the UPF per second. More packets are processed in the uplink than in the downlink. This is because

downlink traffic has the additional overhead of creating and attaching 62 B outer headers to each packet in the userspace (14 B Ethernet + 20 B IP + 8 B UDP + 8 B GTP + 12 B GTP extension header). The RTC design can process a maximum of 5.17 $Mpps$ in the uplink and 4.11 $Mpps$ in the downlink. In contrast, the pipeline design can process 3.51 $Mpps$ and 2.89 $Mpps$ in the uplink and downlink, respectively. The reason why RTC performs better than pipeline is discussed in section 6.2.6.1.
When data is sent simultaneously from both the RAN and the DNN (duplex traffic), the pipeline design can process 3.17 $Mpps$, which is almost halfway between purely uplink packet processing speed and purely downlink packet processing speed (as expected). The RTC design, however, processes uplink and downlink traffic on 2 separate cores due to the 2 NIC-port setup (1 port listening to traffic from RAN, the other listening to traffic from DNN). As a result, its processing speed is the aggregate of the uplink and downlink packet processing capability of the UPF (9.25 $Mpps$).
Note that packet processing capability does not depend on the size of the payload. This is because, for each packet, only the header fields are manipulated (added/removed) while keeping the payload as it is. Each packet has a 128 B headroom (empty by default) which is used if extra space is needed to add a header, and the data offset is moved accordingly. Since header size is fixed irrespective of the payload size, performance is the same across all payload sizes.

Figure 6.3 shows throughput in terms of $Gbps$. The RTC design saturates $10G$ in both uplink and downlink from 256 B payload onwards. The pipeline design saturates $10G$ for payload sizes >= 256 B in the uplink and >= 512 B in the downlink.

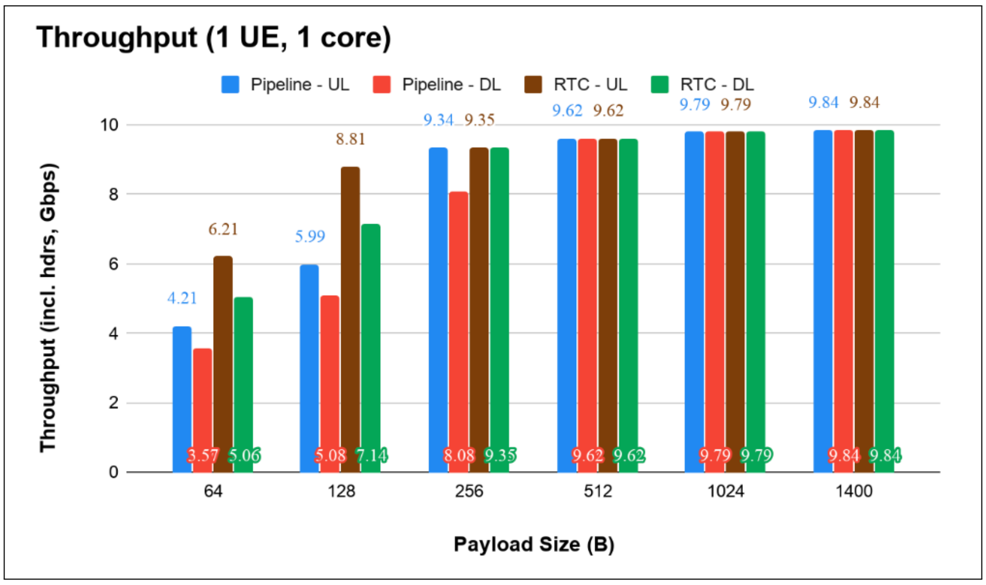


Figure 6.3: Single Core throughput (Gbps) for various payload sizes

- **Multiple UE throughput:**
  Figure 6.4 shows how both the designs perform when multiple UEs send data simultaneously. For this experiment, 99000 sessions are established before sending any data packets, with each session having 1 rule for uplink packets and 1 rule for downlink packets. Once the sessions are established, the number

of active UEs sending data packets is increased from 1 to 16384 (= 2^{14}). As shown, there is a negligible decrease in performance ($\sim$ 0.2 *Mpps*) even when 2^{14} UEs are sending data simultaneously. Thus, both designs can scale with multiple UEs/sessions.

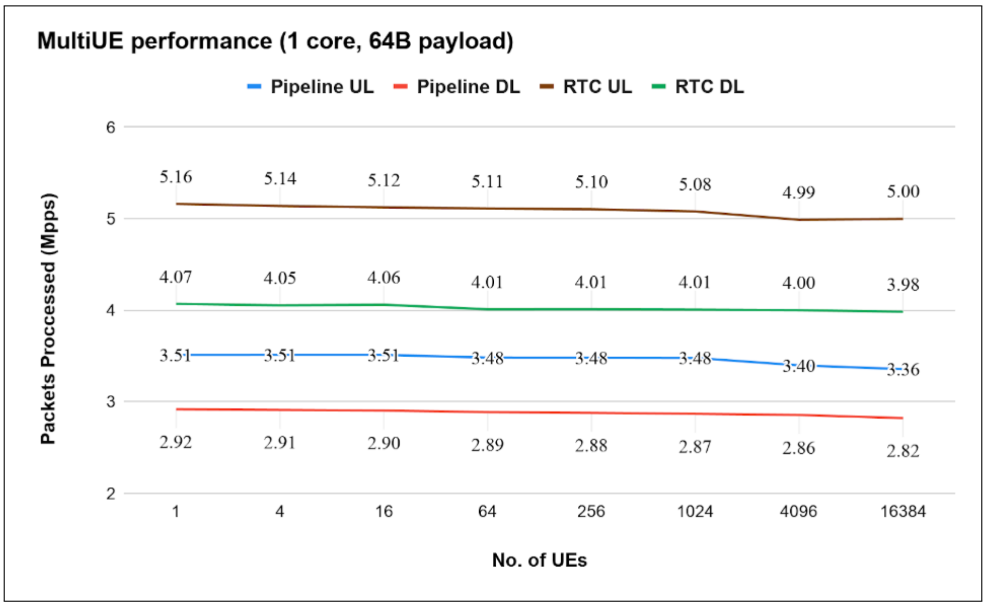


Figure 6.4: Performance with multiple UEs (Mpps)

6.2.2 Scalability with multiple cores

All experiments in this section also assume that the rate at which packets are coming from the RAN is always less than the session-AMBR (session-AMBR set to 10 *Gbps*).

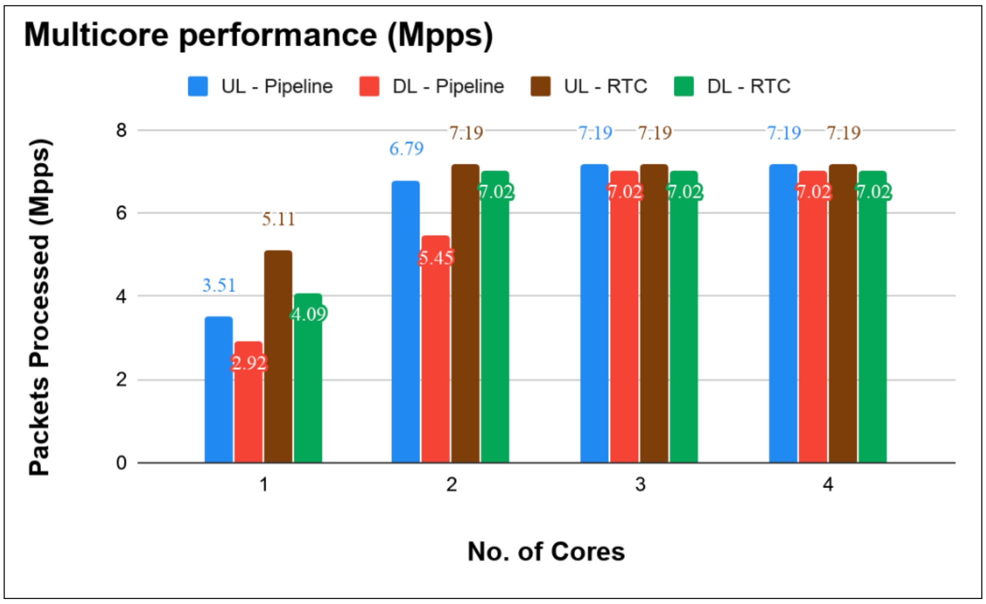


Figure 6.5: Multicore performance (Mpps)

Figure 6.5 shows how both the designs perform with an increasing number of packet processing cores.
Since the maximum possible processing rate in the 10 *Gbps* setup is 7.19 *Mpps* in

the uplink and 7.02 *Mpps* in the downlink, the RTC design easily saturates the line rate with 2 cores. It may be argued that from the graph, it is inconclusive whether the RTC design is scalable across cores. To say with certainty that the RTC design is scalable, the experiment should be done in a more advanced setup (say, with 40 *Gbps* NICs).

The pipeline design needed 3 processing (worker) cores to saturate the line rate. Here, the scalability is almost linear with an increasing number of cores. The slight decrease in the processing capability is because the worker cores themselves were receiving less number of packets (3.5 *Mpps* received when there was 1 worker core, 3.4 *Mpps* received with 2). Future work in the direction of increasing the s/w ring size or NIC RX-descriptor size may help to gain more insight into why the worker cores were receiving less number of packets.

Figure 6.6 shows the throughput in terms of Gbps when the number of cores is increased from 1 to 4. As stated earlier, the RTC design took 2 cores to saturate 10 *Gbps*, while the pipeline design needed 3 cores.

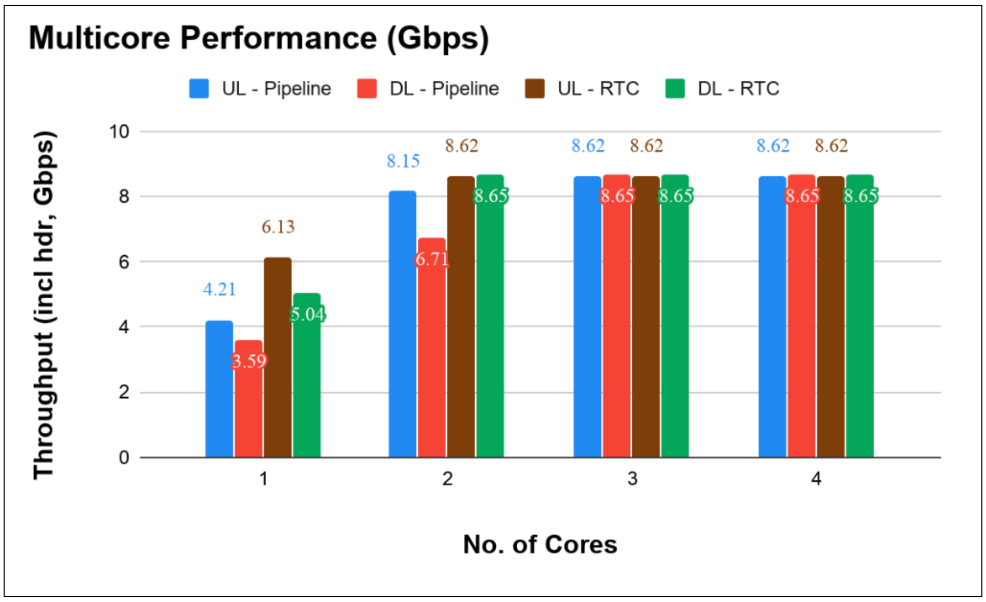


Figure 6.6: Multicore performance (Gbps)

6.2.3 Latency measurements

For all experiments in this section, the outgoing rate in the RAN (and hence the incoming rate in the UPF) is kept at 90% of the maximum processing capability of the UPF. Otherwise, latency increases due to queueing delays. The session-AMBR is set to 10 *Gbps*.
For calculating latency, a callback function adds a timestamp value each time a packet is received, and another callback function calculates the difference in timestamp when that packet is being transmitted. Then, the following formula is used:

$$Latency = \frac{\sum(Timestamp\_difference)}{\#\ packets} \tag{6.1}$$

Payload (B)	RTC (us)	Pipeline (us)
64	8.69	14.90
512	14.15	18.03
1400	52.34	33.89

Table 6.1: Latency for uplink packets

Tables 6.1 and 6.2 show the latency for uplink and downlink packets respectively. As expected, time taken to process downlink packets is slightly more than that of uplink packets. Larger payload sizes also take slightly more time. In general, latency of the RTC design is less when packet size is small. For larger packet sizes, the pipeline design has lower latency. However, latency results will become more clear and concrete once the experiments are done in a $40G$ NIC setup.

Payload (B)	RTC (us)	Pipeline (us)
64	12.66	34.44
512	16.75	17.94
1400	56.36	48.34

Table 6.2: Latency for downlink packets

6.2.4 QoS correctness

Figure 6.7 shows the timeline graph when packets are sent from a UE to the DNN. The RAN sends out 1024 *B* payloads at different speeds, alternating between 800 *Mbps* and 1200 *Mbps* (incl. headers) every 30 *secs*, while the session-AMBR (i.e., UPF outgoing rate limit) is set at 1 *Gbps*. Both the RTC and the pipeline design limit the outgoing rate correctly — matching the incoming rate when it is less than 1 *Gbps*, limiting the rate otherwise.

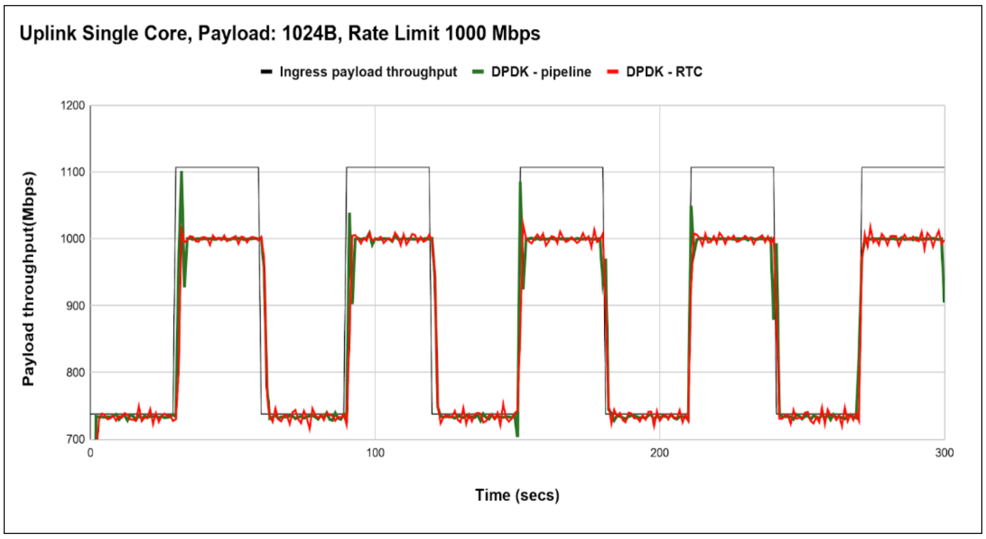


Figure 6.7: Time series graph showing rate limiting

Figure 6.8 shows the throughput when both uplink and downlink packets are received at the UPF. The session-AMBR is set at 1 *Gbps* for uplink packets and 500 *Mbps* for downlink packets. The RAN is transmitting packets from a UE with

1.5 *Gbps* speed. The UPF is sending them out at 1 *Gbps* rate to the DNN, which is mirroring back the packets at the same rate (1 *Gbps*). The UPF is then again restricting the downlink packets to 500 *Mbps*. Both the designs perform as expected. For 64 *B* packets, the pipeline throughput is slightly less than expected as it is bottlenecked by its single-core packet processing capability for duplex traffic. For larger packets, exact throughputs are obtained, as shown.

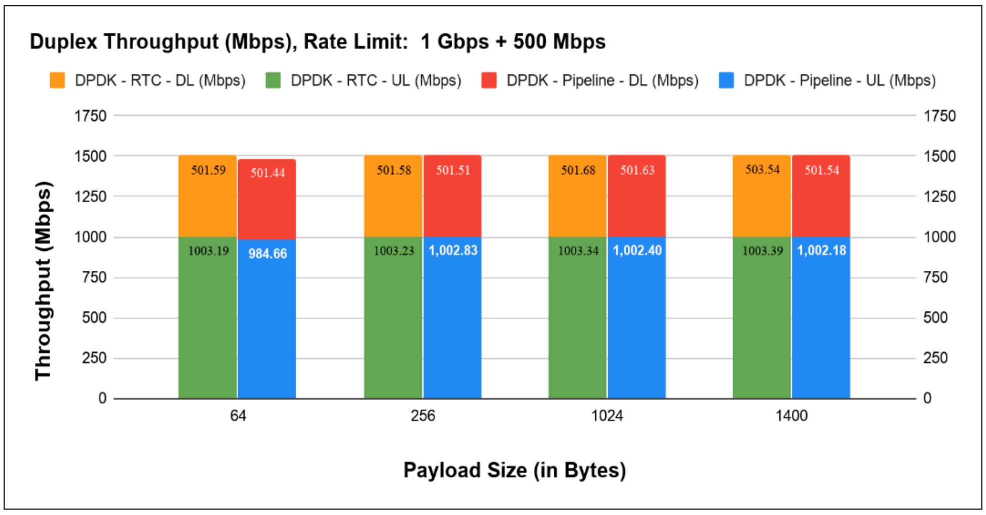


Figure 6.8: Rate limiting both UL and DL packets simultaneously

Figure 6.9 shows the rate-limiting functionality across multiple cores and multiple UEs. 4 UEs are sending data packets with speed > 250 *Mbps* each. Session-AMBR is set to 250 *Mbps* per UE in the UPF. RSS on NIC divides traffic from the UEs to 3 separate cores, such that 2 UEs are directed to 1 core (core 3), and the other 2 UEs are directed to the two other cores (cores 1 and 2). As expected, in core 3, total throughput is 500 *Mbps*. The output from both the UEs is limited to 250 *Mbps*, with rate limit on 1 UE independent of the rate limit on the 2*nd* UE. On the other two cores, outgoing speed is restricted to 250 *Mbps*.

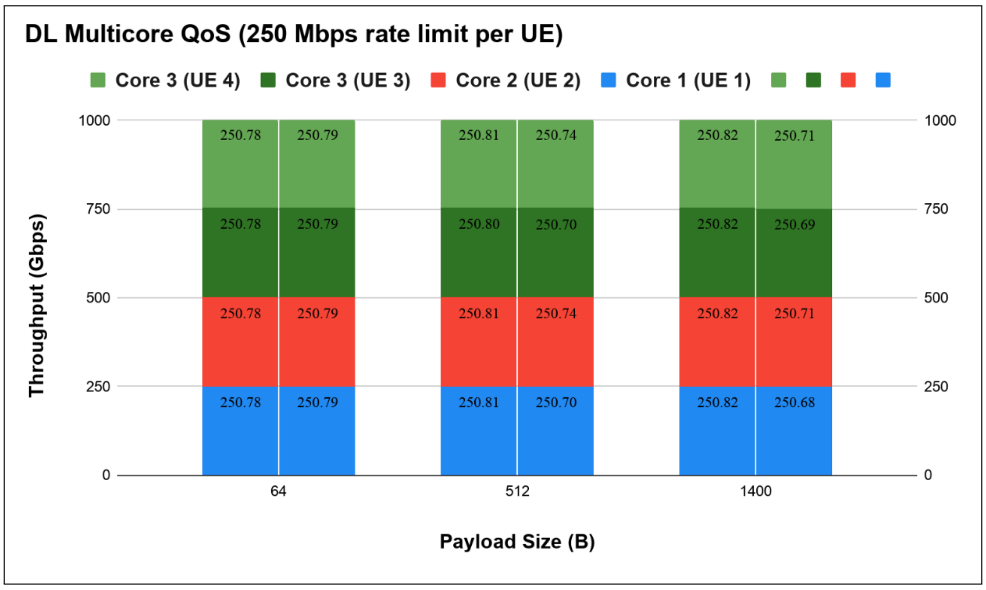


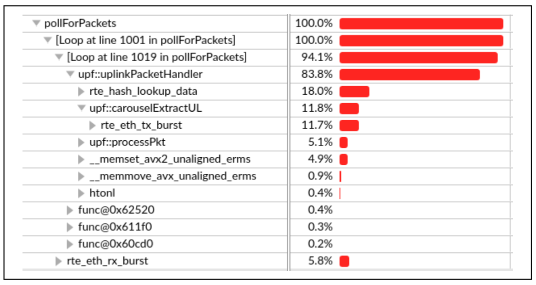
Figure 6.9: Rate limiting across multiple cores/UEs

6.2.5 QoS overhead

Any real-life UPF should support QoS. Results without enforcing QoS make little sense in the practical context. However, the performance results are briefly put here for the sake of completeness and to give an idea of how much overhead is caused by the QoS implementation.

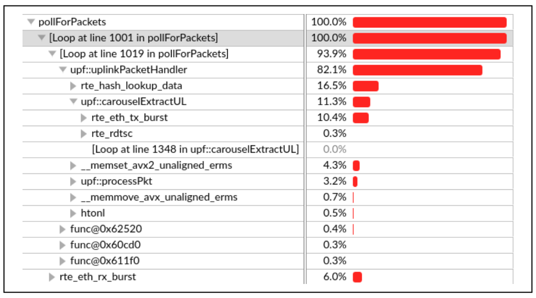
The RTC based design can process 5.51 $Mpps$ in the uplink and 4.34 $Mpps$ in the downlink when QoS is disabled. When QoS is enabled, the UPF can process 5.17 $Mpps$ in the uplink and 4.11 $Mpps$ in the downlink. This slight decrease in performance is due to the following additional overheads when QoS is enforced:

- Calculating current time periodically to update FTS and extract packets.
- Updating LTS after processing each packet.
- In addition to calling *extract()* after every 32 packets, *extract()* is also called periodically every 100 us when QoS is enabled. This is to ensure FTS remains up-to-date even when there are no incoming packets, as well as to transmit any residual packets that may have been previously enqueued.



```
pollForPackets                                   100.0%
  [Loop at line 1001 in pollForPackets]          100.0%
    [Loop at line 1019 in pollForPackets]         94.1%
      upf::uplinkPacketHandler                    83.8%
        rte_hash_lookup_data                      18.0%
        upf::carouselExtractUL                    11.8%
          rte_eth_tx_burst                        11.7%
        upf::processPkt                            5.1%
        __memset_avx2_unaligned_erms               4.9%
        __memmove_avx_unaligned_erms               0.9%
        htonl                                      0.4%
      func@0x62520                                 0.4%
      func@0x611f0                                 0.3%
      func@0x60cd0                                 0.2%
    rte_eth_rx_burst                               5.8%
```

(a) QoS disabled



```
pollForPackets                                        100.0%
  [Loop at line 1001 in pollForPackets]               100.0%
    [Loop at line 1019 in pollForPackets]              93.9%
      upf::uplinkPacketHandler                         82.1%
        rte_hash_lookup_data                           16.5%
        upf::carouselExtractUL                         11.3%
          rte_eth_tx_burst                             10.4%
          rte_rdtsc                                     0.3%
          [Loop at line 1348 in upf::carouselExtractUL] 0.0%
        __memset_avx2_unaligned_erms                    4.3%
        upf::processPkt                                 3.2%
        __memmove_avx_unaligned_erms                    0.7%
        htonl                                           0.5%
      func@0x62520                                      0.4%
      func@0x60cd0                                      0.3%
      func@0x611f0                                      0.3%
    rte_eth_rx_burst                                    6.0%
```

(b) QoS enabled

Figure 6.10: RTC: Profiling with and without QoS[1]

Figure 6.10 shows code profiling for the RTC design, both with (Fig. 6.10b) and without QoS (Fig. 6.10a). Due to the said overheads, *uplinkPacketHandler()* in Figure 6.10b — which is called for each packet — gets less percentage of CPU time. Also, some CPU usage is spent on *rte_rdtsc()*, which calculates the current time.

Although the pipeline design follows the same trend, there are some issues. In the downlink, maximum packets processed by the UPF is 2.96 $Mpps$ when QoS is disabled and 2.89 $Mpps$ when Qos is enabled. In contrast, the uplink throughput drops from 4.22 $Mpps$ to 3.51 $Mpps$ when QoS is enforced. The reason for this is yet to be determined. Further inspection is required to find out why the pipeline uplink packet handling code is behaving differently from the other cases (i.e., significant drop in performance when QoS is enabled) and will be done in the future.

[1]Profiling done using Intel vTune Amplifier

6.2.6 Comparison of different UPF designs

6.2.6.1 RTC vs Pipeline

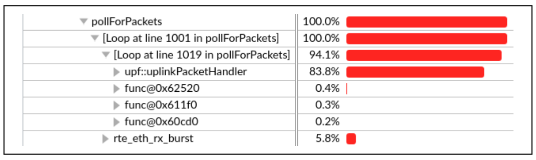
The RTC based design performs better than the pipeline based design, as already mentioned in section 6.2.1. Following are the reasons:

- As shown in figure 6.11, *uplinkPacketHanlder()* — which is called per packet — is getting more of the CPU time in the RTC based design. Since in the pipeline design, two cores listen for traffic from the RAN and the DNN and both the cores can enqueue packets in the s/w ring of the same worker core, the worker core has to check whether a packet is coming from the RAN or the DNN every time. This is not there in the RTC design, as depending on the NIC port a core listens to, it can determine the source of the packet.

- Overhead of enqueuing to and dequeuing packets from the s/w rings.

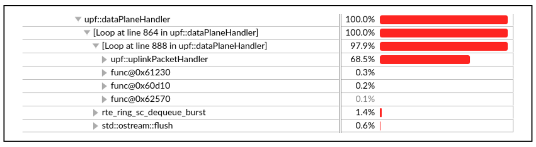
- Deciding where to redirect the packet in the userspace as opposed to RSS on NIC in the RTC design.

As a result, *uplinkPacketHandler()* only utilises 68.5% of the CPU time. Hence, the number of packets processed is less in the pipeline design.



```
▼ pollForPackets                                  100.0%
  ▼ [Loop at line 1001 in pollForPackets]         100.0%
    ▼ [Loop at line 1019 in pollForPackets]        94.1%
      ▶ upf::uplinkPacketHandler                   83.8%
      ▶ func@0x62520                                0.4%
      ▶ func@0x611f0                                0.3%
      ▶ func@0x60cd0                                0.2%
    ▶ rte_eth_rx_burst                              5.8%
```

(a) RTC design



```
▼ upf::dataPlaneHandler                                   100.0%
  ▼ [Loop at line 864 in upf::dataPlaneHandler]           100.0%
    ▼ [Loop at line 888 in upf::dataPlaneHandler]          97.9%
      ▶ upf::uplinkPacketHandler                           68.5%
      ▶ func@0x61230                                        0.3%
      ▶ func@0x60d10                                        0.2%
      ▶ func@0x62570                                        0.1%
    ▶ rte_ring_sc_dequeue_burst                             1.4%
    ▶ std::ostream::flush                                   0.6%
```

(b) Pipeline design

Figure 6.11: Code profiling: RTC vs Pipeline[2]

It would be interesting to see how both the designs perform when there is only 1 NIC port to listen to (instead of the 2 NIC-port setup currently). Then, the RTC design would also have to check the source of an incoming packet every time, and the difference in performance may reduce.

6.2.6.2 Comparing the UPFs in 5G testbed

This section compares the various UPF designs currently being developed in the 5G-testbed. DPDK-based UPF processes packets in the userspace. eBPF/XDP-based UPF processes packets in the Linux kernel device driver level. SmartNIC-based UPF processes and forwards packets from the NIC itself. Kernel UPF is the baseline design.

- **Single-core throughput for various payload sizes:**
  Figure 6.12 compares the uplink throughput for various payload sizes. The RTC based UPF has the highest packet processing rate. It processes $\sim 40\%$ more packets than the next best option — the eBPF based UPF. The pipeline design performs slightly worse than the eBPF based UPF. Kernel UPF can process only 0.06 $Mpps$.

[2] Profiling done using Intel vTune Amplifier

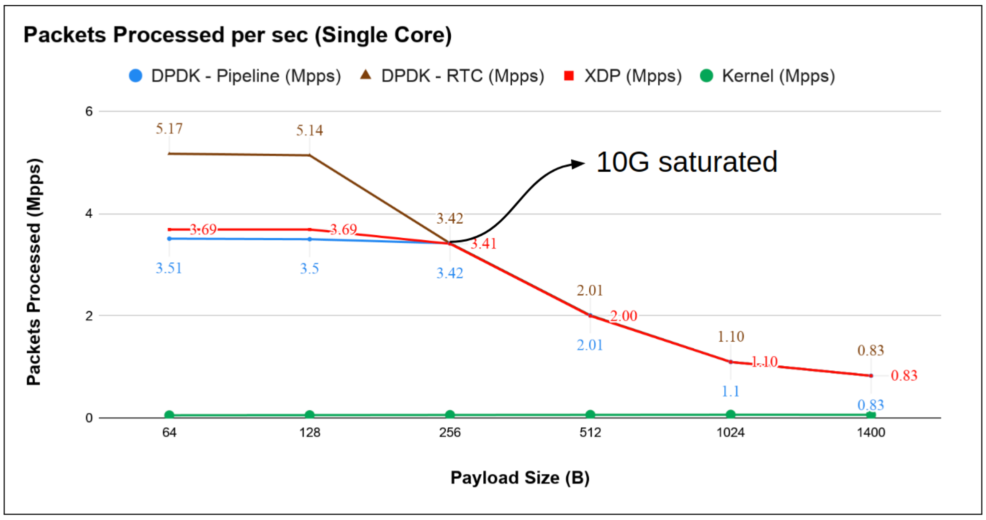


Figure 6.12: Single core performance of various UPFs (Uplink)

Figure 6.13 compares the downlink throughput for various payload sizes. Again, the RTC design outperforms all. However, the eBPF based UPF processing rate is only $\sim 13\%$ below the RTC design, while being $\sim 20\%$ above the pipeline design. This difference from the uplink comparison is because downlink involves more userspace packet processing (adding and copying headers) in the DPDK UPFs, while eBPF UPF can process packets in the driver level itself. Again, kernel UPF can process only 0.06 $Mpps$.

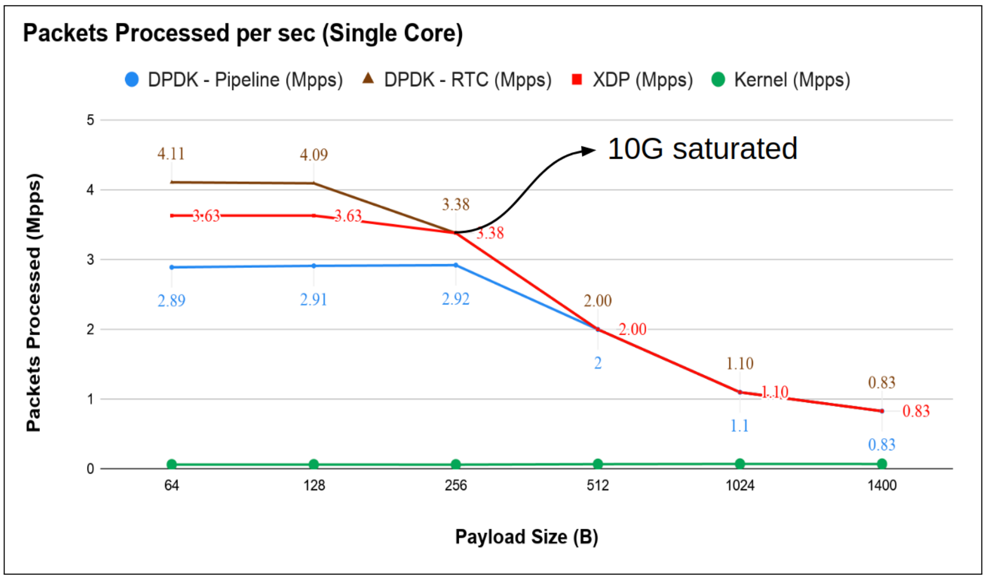


Figure 6.13: Single core performance of various UPFs (Downlink)

- **Performance with multiple UEs/sessions:**
  Figure 6.14 shows the downlink performance of various UPFs (single-core) when the number of active UEs are increased from 1 to 16384. $99k$ sessions were established before sending any data packets. Each session had 2 PDRs — 1 for uplink packets and another for downlink packets. All the UPF designs show negligible performance degradation even when 16384 UEs are downloading data simultaneously (~ 0.15 $Mpps$).

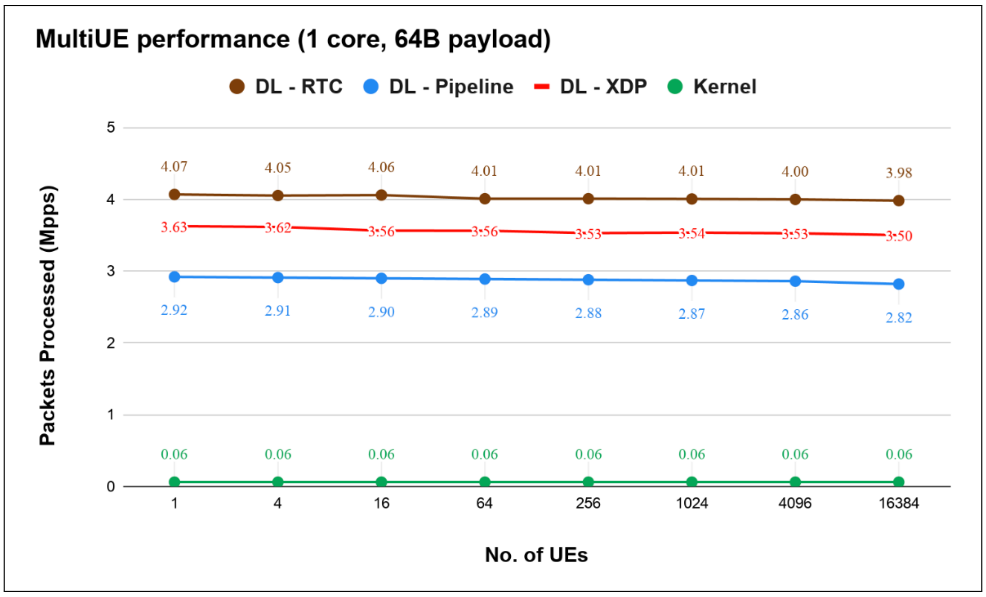


Figure 6.14: Performance of various UPFs with multiple UEs

- **Scalability across multiple cores:**
  Figure 6.15 shows downlink performance of the UPFs across multiple cores. The RTC design takes 2 cores to saturate 10 *Gbps*. In contrast, the eBPF UPF and the pipeline based design take 3 cores to do so. All designs scale almost linearly, although we can only say anything certain about the scalability of the RTC based design after moving to 40*G* NIC.

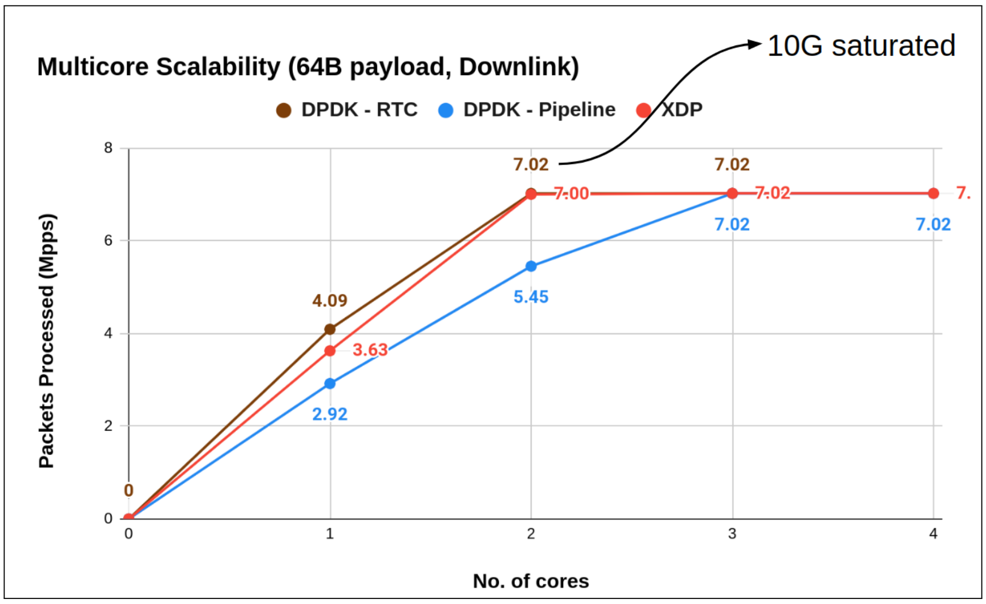


Figure 6.15: Multi-core performance of various UPFs (Downlink)

- **In-depth comparison of DPDK-UPF and eBPF-UPF:**
  The eBPF based UPF is faster than the pipeline design, but slower than the RTC design.

UPF design	CPU cycles used for packet processing (%)	Throughput (Mpps)
DPDK-RTC (Uplink)	83.64%	5.17
eBPF/XDP (Uplink)	56.03%	3.69
DPDK-RTC (Downlink)	71.16%	4.11
eBPF/XDP (Downlink)	59.16%	3.63

Table 6.3: Comparing packet processing times of DPDK and eBPF based UPF

  Table 6.3 shows how much CPU both the UPFs used for processing data plane packets (both uplink and downlink). Profiling was done using the *perf* tool. In the uplink, the DPDK based UPF gets $\sim$ 33% more processing time than the eBPF based UPF. This is because I/O operations are more costly in XDP than in DPDK. As a result, DPDK-UPF performance is better by $\sim$ 29%. The two percentages are not the same because DPDK-UPF takes around 230 ns to process a packet in the userspace while eBPF-UPF takes around 188 ns. Note that in both of the UPFs, this time is calculated from the moment a packet is received (in userspace in case of DPDK) till before it is sent out (just before calling the TX API in case of DPDK). Hence, this value is different from the values shown in Tables 6.1 and 6.2.

  Similarly, in the downlink, the difference in CPU cycles used is $\sim$ 17%. As a result, DPDK based UPF performance is $\sim$ 12% better than the eBPF based UPF.

- **Summary:**
  Finally, Table 6.4 gives an overall summary of the various UPFs (excluding the kernel-based UPF). Each UPF design has certain pros and cons, and developers should decide which UPF to use based on the workload, resources available and ease of implementation required.

Factors	**DPDK-UPF**	**XDP-UPF**	**SmartNIC-UPF**
NIC	Should be compatible	No restrictions (driver support required for maximum performance)	Should be compatible
Checksum calculation / verification	Offloaded to NIC	No offload feature	No offload feature
QoS implementation	Easier to implement in userspace	Difficult	Difficult
Handling non-UDP packets (ARP/HTTP)	Process in userspace (complex) / resend to kernel	Easier to handle	Easier to handle
Performance	RTC design better overall	Better than DPDK pipeline	Better when QoS is disabled

Table 6.4: UPF design comparison

Note that SmartNIC data is not shown in any of the graphs. Although SmartNIC performs best when QoS is disabled (saturating line rate for any payload size), there are some issues with the SmartNIC based UPF once QoS is enabled, due to which similar data is not available. Discussing those issues are beyond the scope of this report.

7. Related Work

DPDK based UPF: Lee et al. [9] have proposed a few methods in their white-paper to scale up the UPF. They use a Smart-NIC to redirect GTP encapsulated packets to separate cores by checking the inner IP address (i.e., UE IP). DPDK APIs are used to receive the packets from the NIC RX queues to the userspace. Once received, the cores follow the RTC model to process the packets and send them out. The paper is mainly focused on achieving high throughput ($\sim$ 200 *Gbps*, 40 cores) in the UPF irrespective of the type of workload. The authors state the techniques they have used without giving any background on why they chose such techniques.
While inspired by [9], the DPDK based UPF at IIT-B goes beyond what the paper did. This work also aims to saturate the line rate between the UPF and other NFs and can currently process packets $4X$ faster than [9] per core. However, this work first explains why frameworks like DPDK are actually required to build high-performance UPFs. The novelty in the current implementation is that it supports both the pipeline based design and the RTC based design and compares them, listing the pros and cons of each so that developers can choose the appropriate design based on the type of workload as well as the available resources.

Another DPDK based UPF design from Intel [8] uses a 3^{rd} party accelerator — Metaswitch's Composable Network Application Processor (CNAP) — for packet processing. CNAP uses a highly configurable match-action classifier which optimizes packet processing to provide very high throughput ($\sim$ 500 *Gbps*, 25 cores, 839 *B* payload). The current work does not use any such classifier for processing packets. Further comparisons are not possible at this moment due to very little information available on [8].

Alternate UPF designs: Additionally, the current work also performs a comparative study of the DPDK based design with the eBPF/XDP based UPF as well as the SmartNIC based UPF design, which are being currently developed in the 5G testbed. The main difference in each of these designs is the level at which the UE data packets are processed. In the SmartNIC based UPF, incoming packets are matched against the PDRs in the NIC itself and then forwarded to the DNN. The eBPF based UPF uses XDP [15], which provides a programmable and high-performance data path in the Linux kernel, at the device driver level. The TCP/IP stack is skipped to avoid the kernel-related bottlenecks. In contrast, the DPDK based UPF bypasses the kernel, and all packets are processed in the userspace. A detailed comparison among the UPFs is already discussed in Section 6.2.6.
Other than that, DPDK based UPF with eBPF/SmartNIC in the kernel/NIC level respectively is also being explored. In this model, the majority of the packets will be sent out directly via eBPF/SmartNIC (fast path), while redirecting remaining packets to the userspace via DPDK (slow path). These remaining packets can either be CP packets or packets which need to be queued for rate-limiting in some sophisticated QoS application only possible to implement in the userspace.

8. Current and Future Work

Following are a few of the things that are being implemented currently or will be interesting to explore in the future:

- **Fixing UPF issues**
  There are a few issues w.r.t both the pipeline and the RTC model. For example, the RTC model fluctuates above and below the rate limit too frequently if $Horizon > 100ms$. Currently, such issues are being fixed before moving on to adding new features.

- **Adding wildcards to packet rules**
  Currently, PDIs in the packets match exactly with the PDIs in the PDRs. $3GPP$ specification also allows wildcards in PDRs. For example, instead of a unique UE IP, the rule may state that all UE IPs within a range must perform the same action. Thus, wildcard matching must be incorporated in the future versions of the UPF.

- **Guaranteed Minimum Bit-rate (GBR) in QoS**
  GBR is an essential feature for any UPF, as it is natural for users to desire a minimum outgoing bit-rate for the subscription fees they pay. Thus, the QoS implementation needs to be revamped in the future. Note that Carousel [16] may not be a suitable fit for future QoS implementations and other state-of-the-art techniques may need to be looked at.

- **Optimizing the UPF further**
  Introducing h/w accelerators (for instance, DPDK flow library [7]), or combining the DPDK based UPF with SmartNIC (in the NIC level) / eBPF (in the Linux kernel level) is an interesting area to explore in the future, to improve the UPF performance further. Based on the current results, although the RTC design performs better than other UPFs, the pipeline design performance has the potential to be boosted when combined with eBPF/smartNIC.

- **Scaling across NUMA nodes**
  Future work involves designing the DP in a way that scaling up across the NUMA nodes does not become a bottleneck, as it is well established that inter-core communication when two cores are in different NUMA nodes is quite expensive.

- **Multiple UPF setup**
  Current 5G DP has only 1 UPF forwarding uplink packets to the DNN and downlink packets to the RAN. Future work may include building a data plane supporting multiple UPFs between the RAN and the DNN, with GTP tunnelling between 2 UPFs.

9. Conclusion

5G architecture can tackle various data/latency driven use-cases and the ever-increasing cellular traffic associated with it. This report presents an overview of the kernel-based 5G test-bed Data Plane design and the problems arising with fast packet I/O due to the kernel TCP/IP stack limitations (for instance per packet interrupt and TUN device overhead). The report then moves on to how kernel bypass techniques like DPDK optimizes the Data Plane, and details about the DPDK-based UPF design, where packet processing is done in the userspace, and the TUN device is removed. Currently, two models are supported — RTC and pipeline based design. Both the designs are multi-core scalable. Without the previous limitations, both have thus saturated the $10G$ link between the RAN, the UPF and the DNN. The UPF performance stays almost the same, even when multiple UEs are active simultaneously (tested with 2^{14} UEs). The RTC design performs better currently and saturates the downlink from 256 B payload onwards. In contrast, the pipeline design saturates from 512 B payload onwards when a single core is dedicated to DP packet processing.

This report also discusses the various observations made along the way regarding design and implementation aspects, evaluates the current DPDK based UPF, and compares it with other UPFs being implemented alongside currently (over kernel/XDP/SmartNIC). Currently, the RTC based design outperforms all other UPFs after enforcing QoS.

Acknowledgement

I thank Prof. Mythili Vutukuru, who has guided me throughout this project with crucial insights and valuable feedback. My gratitude towards the project engineers of the 5G test-bed team, who helped me become familiarized with the 5G standards, and also cleared my concepts. I also thank the SynerG faculty, the PhD students and my M.Tech colleagues for their inputs, feedbacks and help all along the way.

Bibliography

[1] DPDK overview. https://doc.dpdk.org/guides/prog_guide/overview.html

[2] DPDK mempool library. https://doc.dpdk.org/guides/prog_guide/mempool_lib.html

[3] DPDK mbuf library. https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html

[4] DPDK ring library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html

[5] DPDK hash library. https://doc.dpdk.org/guides/prog_guide/hash_lib.html

[6] DPDK timer library. https://doc.dpdk.org/guides/prog_guide/timer_lib.html

[7] DPDK generic flow library. https://doc.dpdk.org/guides/prog_guide/rte_flow.html

[8] Lighting Up the 5G Core with a High-Speed User Plane on Intel Architecture: https://www.metaswitch.com/blog/intel-and-metaswitch-lights-up-the-5g-core-with-a-500-gbps-cloud-native-upf

[9] Lee, D.J., Park, J.H., Hiremath, C., Mangan, J., Lynch, M., 2018. Towards achieving high performance in 5G mobile packet core's user plane function. [White paper from Intel & SK Telecom]

[10] 3GPP Ref #: 23.501. 2017. System architecture for the 5G System (5GS). https://www.3gpp.org/ftp/Specs/archive/23_series/23.501/

[11] 3GPP Ref #: 23.502. 2017. System architecture for the 5G System (5GS). https://www.3gpp.org/ftp/Specs/archive/23_series/23.502/

[12] 3GPP Ref #: 29.060. 2017. System architecture for the 5G System (5GS). https://www.3gpp.org/ftp/Specs/archive/29_series/29.060/

[13] 3GPP Ref #: 29.244. 2017. System architecture for the 5G System (5GS). https://www.3gpp.org/ftp/Specs/archive/29_series/29.244/

[14] 3GPP Ref #: 38.415. 2017. System architecture for the 5G System (5GS). https://www.3gpp.org/ftp/Specs/archive/38_series/38.415/

[15] Høiland-Jørgensen, Toke, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller."The express data path: Fast programmable packet processing in the operating system kernel." In Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, pp. 54-66. 2018.

[16] Saeed, Ahmed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. "Carousel: Scalable traffic shaping at end hosts." In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 404-417. 2017.

Appendices

A. Implementation bugs and fixes

The following list contains some of the bugs encountered and fixes provided, as well as things that degraded the UPF performance while implementing the DPDK based UPF. This is listed here to support any future DPDK based UPF, or any UPF/DPDK application in general.

- *rxnombuf* errors: DPDK was unable to allocate sufficient memory for incoming packets, due to no free space in the configured mempool. Previous packets may not have been appropriately freed. This issue also arose when the s/w ring size was set too large (2^{25}). Ring sizes till 2^{14} work fine.
- Multi-threaded setup does not work well with DPDK while receiving packets, as the application needs the whole core to itself for polling the NIC RX queue. Hence, any core using the receive API must not spawn any thread if it has to handle a large number of packet I/O. As a result, the 2 NIC setup must have two dedicated cores to receive packets from the NICs.
- In DNN, packets were seen on Wireshark, but not received by the application. This is because the source address of the inner IP (that is, UE IP) was not routable from the DNN and hence kernel was discarding the packet. The solution is to statically set up the route so that when such a packet comes, it should be routed via the UPF. Another solution is disabling *rp_filter* in the DNN, so that kernel does not check for routes. This problem is only evident if the DNN is an *iperf server*. No such issues with DPDK based DNN.
- While compiling, *fsanitize=address* flag was enabled for the compiler. While it checks for memory leaks by periodically monitoring the application in the background, it hinders DPDK performance. Disabling the flag increased throughput by $3X$. Similarly, *htop* or any profiling application running in the background in the same core will degrade DPDK performance.
- Profiling the code helped reduce a lot of bottlenecks — platform logs for debugging, functions taking high CPU time due to multiple addition/deletion of records in the program stack etc. — which made the code faster. For a single DP core forwarding 1422 B packets, throughput went up from 6.2 $Gbps$ to saturating line rate ($\sim$ 9.8 $Gbps$) by removing these bottlenecks in Stage-I.
- *rxerror* in RAN/DNN: After receiving a batch of packets, some packets were freed while some were sent ahead. Wrong checksum calculation can also cause this issue.
- Scaling issues: Per core variables and data structures should be used wherever applicable. Scaling issues arise mostly due to some race conditions where a particular variable is shared between the cores.