Extending RAN with a focus on Understandability and Extensibility

A Project Report Submitted

in the Partial Fulfilment of the Requirements

for the Degree of

Master Of Technology

by

Prateek Agarwal

Computer Science and Engineering

Indian Institute of Technology Bombay

June, 2021

Contents

List of Figures

Chapter 1

Introduction

5G technology aims to satisfy the increasing requirements of the customers of mobile service in terms of higher throughput and low processing latency. The major difference in the processing of data and signalling packets in the core network between 5G and earlier standards is control-user plane separation and the use of network function virtualization. Forwarding of packets , authentication of mobile devices, session establishment and management are some of the network functions required in the core of a telecommunication network. These network functions run on different or same physical machines as virtual machines for easier migration/scaling.

This project is mainly concerned with the data forwarding plane. The network functions in our setup ran as separate processes. The high speed data plane is required to meet the ever increasing demands of higher bandwidth and lower latency. The Linux kernel stack cannot meet this demand as it is a general purpose stack catering to different protocols and there is a packet copy overhead from kernel space to the user space for reading the packet payload. The technqiues used to overcome this limitation of the Linux stack is the use of kernel bypass frameworks like Data Plane Development Kit (DPDK), and the use of programmable NICs to offload processing functions in the hardware.

DPDK runs in software and there is a single copy of packets from NIC buffer to the user space. DPDK libraries are optimized for high speed data processing. The use of superpages, lockless data structures and customized device drivers for various NICs are among some of the salient features of DPDK framework which enables high speed processing of data.

The main network functions in the data plane are

- **Radio Access Network (RAN)** RAN is the main point of contact for all the user equipments and is an entry point to Non Access Stratum in the 5G network.
- **User Plane Function (UPF)** UPF is responsible for forwarding the incoming packets from the RAN to the public internet. UPF ensures QoS guarantees, collects usage statistics useful for charging purpose, buffers packets (if required).
- **Data Network Name** DNN is the gateway to the public internet. DNN forwards the uplink packets from UEs to the public internet and all the incoming packets for the UEs (downlink) are received at the DNN.

1.1 Problem Statement

- **User Plane Function** Implement the packet steering offload mechanism in the UPF.
- **Radio Access Network** Extend the existing DPDK based RAN by adding new features and refactor, clean the code to make RAN extensible in the future.

1.2 Major Contributions

- UPF with RSS based redirection on the inner packet headers was implemented.
- Two new features were added in the RAN - measurement of control plane latency and simultaneous transfer of control plane and the data plane traffic.
- The source code of the existing RAN was entirely refactored and restructured.
- Conducted experiments for benchmarking different models of the UPF for measurement of throughput and latency metrics under different load conditions.

Chapter 2

Background

A 5G network hash RAN with user equipments and

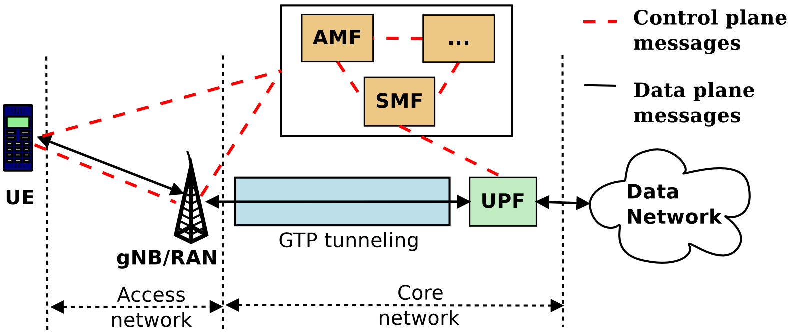

Figure 2.1: 5G Architecture.

2.1 PFCP Protocol

PFCP stands for Packet Forwarding Control Protocol. There are two types of messages sent using PFCP - node related and session related. The discussion here describes session related messages.

Session Management Function (SMF) interacts with the User Plane Function (UPF) to setup sessions related information at the UPF. This information enables UPF to identify data packets of different sessions coming from RAN and provide forwarding, usage report, charging, buffering, QoS related service to the sessions. Each session is identified by a unique session ID which helps in differentiating among different UEs/sessions at the UPF.

There are three kinds of messages sent from SMF to the UPF in PFCP session related messages-

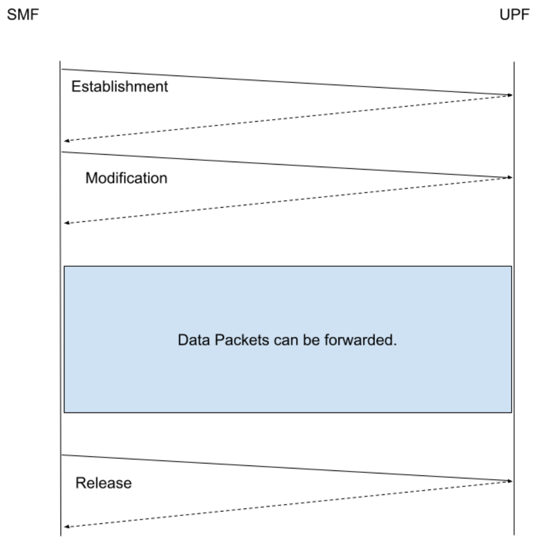

Figure 2.2: PFCP Session Messages

- **Establishment Request** This message has all the forwarding, usage report, QoS etc. related information for a new UE session.
- **Modification Request** Once the establishment response is received from the UPF, the UPF sends the modification request. The important field in this message is the intimation of identifier that will be used for data plane packets coming from the UPF. The data forwarding can not start before the modification response is received.
- **Release Request** Once the session is not required, th release request is sent to the UPF signaling the tear down of the session and UPF may remove this session's related information.

UPF gives response to each of the three messages.

2.2 GTP Protocol

The raw packets meant for communicating outside the netowrk are encapsulated with GTP application level header. GTP runs over UDP/IP stack. This GTP header (shown in **??**)contains various fields for regulating the session parameters. The relevant fields are

./fig/Introduction/intro2.png

Figure 2.3: GTP-U header [?]

- **Tunnel Endpoint Identifier (TEID)** identifies the packet with a particular session which may have different Quality of Service parameters.
- **Extension Headers** These are required for providing differentiated services to a given packet in the same or different session.

Chapter 3

Stage 1 Summary

Chapter 4

RAN Design and Implementation

4.1 Design Objectives

The RAN-emulator which acts as load generator in both the data plane and control plane has to satisfy the following objectives -

- **High Throughput** The data plane packets should be generated at a sufficient pace to saturate 40 Gbps link.

- **Flexible Modes** The data should be sent at a varying rate from a very low load to the saturation throughput either in a single run or across different runs. It should also be possible to support various modes e.g. both the data plane and control plane packets are sent together, correctness of QoS at UPF can be checked, find a mapping between inter-batch delay and throughput.

- **Latency Measurement** The measurement of latency for both the data plane and control plane packets should be possible.

- **Logging** The latency and throughput should be measured and logged at the every fixed interval (parameter for a given run).

4.2 Prior Work

4.2.1 Kernel-Based RAN

The major limitation of this RAN is that it is not able to saturate max line rate of 40 Gbps. This is due to the fact that there are multiple copies of packet when the packet is received or transmitted - from user space to kernel space and from kernel space to the NIC buffers when the packet is forwarded.

4.2.2 DPDK-based RAN

This RAN is based on DPDK. This RAN satisfied the requirements of high data throughput and low processing latency. The current work extends on this emulator. This emulator had satisfied most of the objectives mentioned in **??** except for the following -

- Measurement of Control Plane Latency.
- Sending Control Plane and Data Plane Packets simultaneously.

4.3 RAN Architecture

4.3.1 Threading Model

DPDK pins one thread to each core for avoiding the overhead of context switch. This is especially important for both the data forwarding cores and the polling cores to get the maximum throughput per core. The auxillary threads like ARP messages handler, heart beat messages, timers etc. can run on available free core. The cores which send data and control plane latency packets have to handle the callbacks associated with the storage of timestamps of outgoing packets. Running them on separate cores is preferred if the sufficient number of cores are available.

The NIC queues/buffers for storing the outgoing (TX-queue) and the incoming (Rx-queue) packets are one-to one mapped to the data forwarding/polling cores. The threads pinned to these cores process the incoming packets and prepare the packets for forwarding.

4.3.2 Core Layout

There are 24 cores on the available machines. 12 cores are available on each NUMA socket. Although there are 2 CPUs (2 hardware threads) available on each core, hyperthreading is kept off for reducing the non deterministic behavior attributed to contention of hardware units and getting repeatable results.

NUMA- node 1 with starting core 12 is currently used by the RAN.

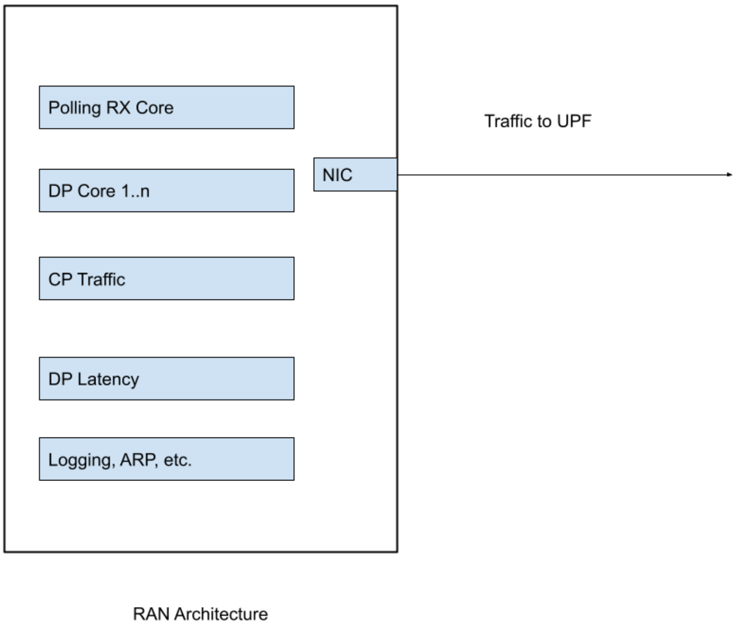

Figure 4.1: RAN Architecture

- **CORE_RX_POLL**: Receives all the incoming packets from UPF. These includes the latency packets or data packets in the downlink direction.

- **CORE_TX_START - CORE_TX_END** These cores are used to send data packets in the uplink direction i.e. from RAN to UPF.

- **CORE_RTT** This core sends the data plane latency packets in the uplink direction. These packets are reflected back by the DNN so that end to end latency can be measured.

- **CORE_CP_TRAFFIC** This core sends the control plane traffic. A separate core was used so that a call back can be registered for storing timestamps. These timestamps are used for calculating control plane latency.

- **CORE_MISC and CORE_STAT** These cores handles miscellaneous functions like ARP handling, timer and logging of stats.

Using this layout, a maximum of 7 cores are available for the data forwarding on the same numa node. To increase this number, the functions running on CORE_MISC and CORE_STAT can be run parallely on the same core. Although 7 cores have been found sufficient till now to saturate 40 Gig line with 64 byte packets.

4.4 Latency Calculation Mechanisms

4.4.1 Key Idea

The sample packets that are used to measure latency are sent from a specific core. When these packets are copied into a NIC buffer, a callback function is called. This callback function stores the timestamp of the outgoing packet in an array. The location at which the timestamp is stored is identified by anentifier present in the outgoing packet. When the response to a sample packet arrives at the receiving core, a call back function is called to measure the end to end latency or response time of the UPF. This callback function extracts the timestamp stored previously and subtracts it from the current time to get the latency measure.

4.4.2 Defintion of Control Plane Latency

For control plane, the measured latency is the difference between two events:

- Forwarding of PFCP requests - session establishment, modification, and deletion requests.
- Receipt of PFCP responses to the establishment, modification and response requests.

This time period includes the following major delays -

- **Processing Delay** This includes the processing of request packets, updation of local data structures at UPF and preparation of response packets.
- **Queueing Delay** This is the time when the PFCP packets remain queued in the NIC or any userspace buffers present in the UPF and the RAN.

4.4.3 Implementation Issues

As discussed in **??**, the main issues to address for measuring control plane latency are-

1. **Identify the identifier** All kinds of PFCP messages have a session identifier field in Forwarding Action rules that is used by the UPF to classify the data plane packets across different UEs/sessions. This field is present at different offsets for different PFCP messages. Each message is also identified by a unique number, for example - 51 for establishment request, 52 for establishment response etc. So we have 6 different kinds of messages for a given session id. A combination of both the message type and session identifier uniquely identifies the packet.

2. **Manage the Callback Overhead** It becomes difficult to process all the callbacks if the packets are sent at a sufficiently high rate. The array is a performant data structure in the sense that it is random access and data is stored contiguously. The access pattern in our use case an excellent spatial locality and temporal locality i.e. all the consecutive entries are accesssed one after the other (session IDs increase sequentially).

4.4.4 Details

A fixed length array is used to store timestamps - **TSArray** . The array size is $65536 * 3$ and stores the value of timestamp register **rdtsc** for the outgoing packets. The first 65536 values are used to store establishment packet related timestamps, next 65536 modification packet related timestamps and then release packet timestamps are stored. The bitwise operations can be easily used as 65536 is a power of 2.

When the corresponding response packets are received, the matching entry is retrieved from the array and latency is calculated.

4.5 Control Plane + Data Plane Traffic

$n1$ sessions are established before the data forwarding takes place. These sessions remain established throughout the run.

$n2$ sessions are established, modified and released while the data forwarding is also taking place. The data packets are sent from all the currently established sessions. The minimum value of currently established sessions is $n1$. The maximum value is $n1 + n2$.

$t1$ is the total duration of the experiment. $t2$ is the duration of the establishment, modification and release cycle of each of the $n2$ sessions.

The duration $t1$ for which all the static sessions $n1$ and the dynamic sessions $n2$ are used is also asked to the user.

Data forwarding starts from the $n1$ sessions at the start. After sleeping for a time (currently 5 seconds), $n2$ threads are started. The role of each thread is to sleep for a random amount of time $t3$ ($< t2$), establish and modify the session, and then sleep for some time $t4$ and release the session. The invariant is $t3 + t4 = t2$. The session is available for data forwarding in the period $t4$. The threads with new session Ids are started once all the previous threads have joined i.e. have finished their task. Each of the data packet forwarding cores use all the existing established sessions/UEs to forward the data. Note that this is different from the case when sessions were partitioned among cores.

4.5.1 Issues

- **Pthreads, Lthreads** There are two kind of threading APIs available. - Posix-threads and lthreads. Lthreads is a user level api which comes with dpdk.
  - **Pthreads** This completely implemented and works correctly when locks are used in both data plane and control plane functions.
  - **Lthreads** DPDK has lthread API available for spawning threads on the same core. This is a user space threading API with a user space scheduler. This is not preemptive and is based on cooperative scheduling -threads yield control for scheduler to schedule another thread. The yielding occurs on certain points when functions like **lthread_sleep** are called. An

alternate set of control plane procedures for defining the dynamic session functionality is defined on an experimental basis. This may be used if there are performance hits in pthread implementation.

- Data plane latency packets are currently sent only from first $n1$ sessions. The dynamically created sessions ($n2$) are used to send data but not the latency packets. This is not difficult to modify and technique that is used to send normal data packets can be used here as well.

Chapter 5

Evaluation

5.1 Experimental Setup

The setup is illustrated in Figure **??**. The main features of the setup are:

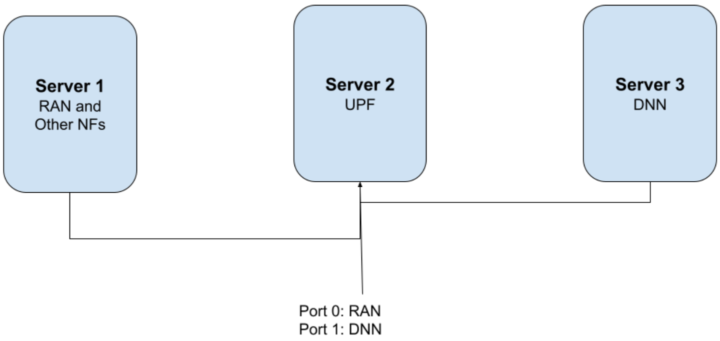

Figure 5.1: Experimental Setup

- **Server's Layout**

  1. **Server 1** The radio access network (RAN) and other network functions responsible for authentication, session establishment, charging functions are simulated on server 1. The user equipments are also simulated on the same machine. The load generation in the uplink direction is the main function of this server in the data forwarding plane.

2. **Server 2** This server hosts the user plane function (UPF). Two ports on this server are used to communicate with RAN (server 1) and DNN (server 3).

3. **Server 3** This server simulates the data network name (DNN) network function. This network function is the gateway to public Internet for 5G telecommunication. This server is used to generate downlink traffic. This server is also used to mirror packets received from uplink and forwarded back in the downlinj direction. The mirroring of latency packets help in measuring end-to-end latency (Round Trip Time) at the RAN.

- **Hardware Configuration**

  1. **CPU** Intel Xeon Core i5 @2.20GHz. 12 cores on every NUMA node. Only a single NUMA node is used in the experiments. Hyperthreading is kept off to facilitate repeatability of results.

  2. **Memory** 8192 superpages of size 2MB are reserved initially for the whole run.

  3. **Cache** 32 KB L1i-cache, 32 KB L1d-cache, 256 KB L2 cache per core. 30 MB L3 cache per NUMA node.

  4. **NIC** Netronome Agilio CX 2x10GbE smartNIC.

5.2 UPF

The primary role of user plane function is described in **??**. UPF is the main network function which takes the centre stage in the core network. The RAN load generator is used to test the processing capability of UPF. The metrics used are throughput and latency of data plane and control plane traffic.

Different designs of the UPF are studied:

- **Software UPF**

  This design uses kernel bypass framework DPDK for userspace processing of the incoming packets. All the control plane and data plane packets are received on the master core which redirects these packets into different worker cores.

- **Offloaded Data Plane** The complete data plane processing is offloaded on the smartNIC. The control plane messages are still processed in the software. Once a session is established (released), the rules are inserted (removed) in the smartNIC.
- **Offloaded Control Plane** In this design, the control plane handling is offloaded to the NIC.

The detailed description of these designs are available in [].

Performance metric	**SoftUPF**	**DPOffload**	**CPOffload**
Throughput (messages/sec)	5.1K	666	2.08M
Latency (μs)	113	1646	24

Table 5.1: Control plane performance.

5.3 Results

5.3.1 Control Plane Performance

The latency v/s. throughput graphs for different UPF designs are illustrated in **??**, **??**, **??**. The summary of the results is shown in **??**. The table **??** shows the latency around the control plane saturation. The control plane throughput is defined in terms of sessions handled per second - the whole establishment, modification and response cycle for a given session is counted once.

When the control plane is offloaded, we can see the best performance in terms of latency and throughput. With the control plane offload, the latency is only around 24 us and 2.08 M sessions can be handled per second. The performance deteriorates for data plane offload when compared to software UPF. This is attributed to the bottleneck created between offloaded data plane and user space control plane - once the control plane messages are processed in the user space, the rules are installed in smartNIC.

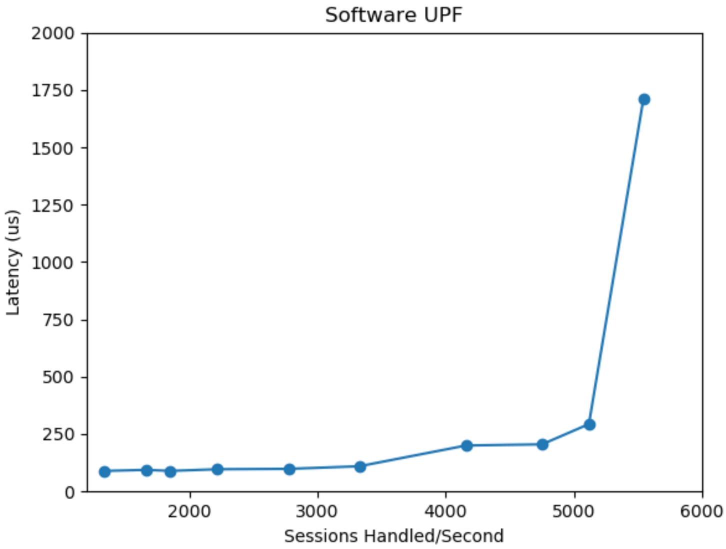

Figure 5.2: Control Plane Performance for Software UPF

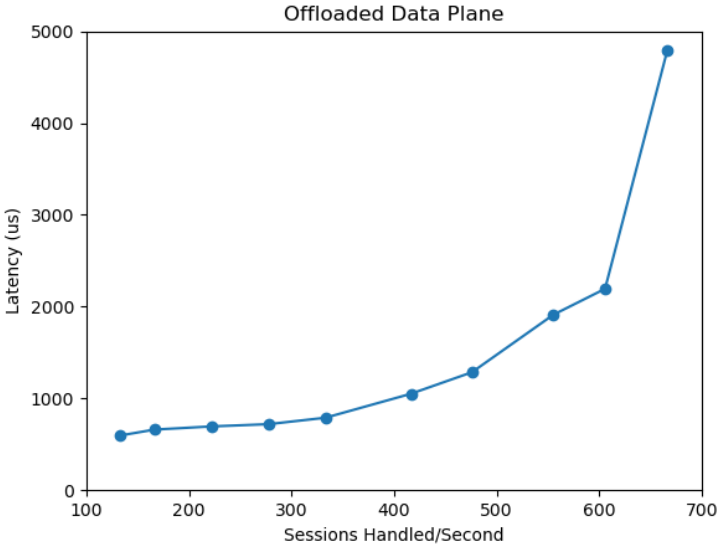

Figure 5.3: Control Plane Performance for Offloaded Data Plane

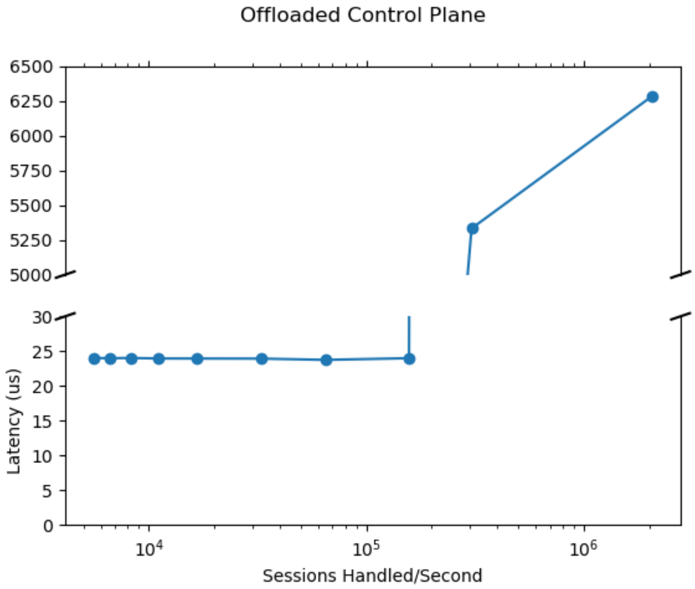

Figure 5.4: Control Plane Performance for Offloaded Control Plane

Chapter 6

Introduction

Chapter 7

Clean Code

This portion of the problem statement took the maximum effort and time. The inherited code was never cleaned and had various issues which are discussed below.

7.1 Naming of Variables

7.1.1 Issues

- **Wrong Case** camelCase should be used in all the 5GCore network functions. PascalCase is OK too. However snake_case, snake_camelCase were also rampantly used in the RAN code.
- **Redundancy** Having test or dpdk in the name of every function is not helpful when it is already a testing/experimental setup and DPDK APIs are used.

7.1.2 Resolution

All new functions that were defined do not have these issues. The name of many past functions are also changed. The complete revamp is avoided as past functions might be familiar to the team by the old names.

7.2 Stale Comments

7.2.1 Issues

- **TODOs** There were many todos lying around from very long. The one who would have these TODOs on their list might have completed them on their branch or have never bothered after writing the TODOs.
- **Misleading Comments** The comments were not updated with the change/removal of the lines of the code.

7.2.2 Resolution

Whatever TODOs and misleading comments that came in my way have been erased.

7.3 Dead Code

7.3.1 Issue

- There were many functions in the code which were never called in any of the data forwarding mode. Some of them were aptly identified as beta functions and many were not.
- Hundreds of lines of code was defined in the conditional statement $if(0)$- they were never called/compiled.

7.3.2 Resolution

Dead code mentioned in the issue sections is cleaned.

7.4 Deprecated Options

7.4.1 Issue

There were 20 modes of data forwarding available. Hardly 3-4 were used for testing purposes. Many options were redundant as same options were implemented using dpdk APIs and kernel based functionalities.

7.4.2 Resolution

The number of modes are reduced to 5. They are appropriately categorized as main data forwarding modes, helper functions and QoS related modes. The code that is not based on DPDK APIs is removed for clarity.

7.5 Reused Code

7.5.1 Issues

DPDK provides a lot of examples showing the usage of their API. It is quite extensive and easy to read and excerpts of code can directly be used in our applications. The code should be cleaned when it is lifted from these applications. The declaration of functions which are never used should be removed. The variables should be named according to the convention that is followed in our source. The header dpdk_ran.h had around 600-800 lines of declared functions which were never defined and used.

7.5.2 Resolution

Most of the declarations discussed here are removed. There might be some declarations in other header files which were not removed. Future developers may clean whenever they come across them.

7.6 Global Variables

7.6.1 Issue

Around 150 variables, data structures were declared globally in ranMain.cpp. The use of global variables made the code highly coupled and made it difficult to change one procedure without affecting the other. Infact some of them were declared but never used. This made it also difficult to discern the scope of variable usage inside a procedure - whether is s global variable, local variable or a class member.

7.6.2 Resolution

- The unused global variables are deleted. The variables which were called in only a few functions are made local and passed as parameter.
- The remaining global variables are defined in a new namespace **Global**. This helps in identifying global variables in the procedures' definitions.

7.7 Directory Structure

7.7.1 Issues

- The earlier RAN directory was not created as a different folder in 5GCore folder as other NFs. It was defined inside AMF. This was very misleading and suggested coupling between AMF and RAN when there never was. It was primarily done to avoid creating a new CMakeLists and reuse the build functionality of AMF.
- All log files were generated in the parent folder itself. This makes it difficult to read, delete , transfer log files.

7.7.2 Resolution

- A different folder DPDK_RAN is created inside 5GCore directory.
- Log files are now generated in subdirectories. Two log files are generated in each run - throughput log files and debugging information related log files.

7.8 Poor Refactoring

7.8.1 Issues

- The main function had a switch statement for different modes of operation. This switch statement was approximately 2000 lines long.

- The DRY (Don't Repeat Yourself) principle was violated multiple times. If all modes require setting of time duration of the run, it is better to define a procedure asking for time rather than repeating it 20 times.
- The files were unusually long. The ranMain.cpp and dpdk_ran.cpp were more than 10k, 8k lines of code respectively. The file in which main routine is defined should be small enough for readers of the code to grasp.

7.8.2 Resolution

The code was extensively refactored. Differnt procedures were defined for each of the switch statement option. Different modes were refactored to make them shorter and easy to understand. ranMain.cpp is bifuracted into ranMain.cpp and ran.cpp (for lack of a better name). The header ranMain.h is defined for inclusion into both the translation units. Further refactoring can be done of the data forwarding procedures if required. It will be a good exercise in understanding of the code.

7.9 Unnecessary Offloads

7.9.1 Issues

As some sections of the code were directly copied from dpdk applications, there were some offloads which are unrelated to our application - VLAN, QINQ and MACSEC offloads. The entire code in dpdk_ran.cpp was infested with these offloads. These offloads were present with IP and UDP checksum related offloads which were used by our application.

7.9.2 Resolution

These lines of code were removed from the dpdk_ran.cpp . However if the surgeon is not an expert, one may remove healthy and important part with tumors as well. Thankfully, it was possible here to reinstall the healthy part back. This took enormous amounts of time and it was a good learning experience for future.

7.10 Technical Improvements

7.10.1 Issues

- High throughput of data plane latency packets. 100 Mbps of data plane latency packets were sent for measuring the end to end latency besides the load. Law of large numbers kicks in for much smaller values and high latency packet throughput causes unnecessary callback overhead. This can further help in moving different functions running on different cores to a single core. This increases the number of cores available for data forwarding.

- The statements like **if (argc == 0)**, for loop running once, comparison of floating point number with equality (==), calling an internal function of the dpdk API when external call.

7.10.2 Resolution

The mentioned throughput was reduced to 0.4 Mbps. This can be further reduced if required. The incorrect statements that I came across were removed.