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 forwarding plane and the relevant network functions are briefly reviewed in 1.1. PFCP protocol and its significance is discussed in 1.2. The layout of the report is outlined in 1.3.

1.1 5G Forwarding Plane

The major difference in data packet processing between 5G and earlier standards is control-user plane separation and the use of network function virtualization. Forwarding of packets (user plane), authentication of mobile devices (control plane), session establishment and management (control plane) 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 (preferably) for easier migration/scaling.

This project is mainly concerned with data forwarding plane. The network functions in our implementation will run as separate processes. The network functions relevant for forwarding plane are described further.

1.1.1 Radio Access Network (RAN)

RAN is a point of contact for all the user equipments (UEs) like handsets, IOT devices, industrial machine controllers etc. RAN runs on all the mobile towers and UEs communicate with the one in their vicinity. RAN is responsible for talking to Access Mobility Function for authenticating the UEs, registering the new session. The

session establishment request is further forwarded to Session Management Function (SMF) which establishes a new session and forward session information to the User Plane Function (UPF).

1.1.2 User Plane Function (UPF)

User plane function (UPF) is responsible for forwarding packets from user equipments to the Internet and vice versa. The uplink direction is defined as the flow of the packets from user equipments to the Internet. The downlink direction is defined as the traffic coming from the Internet to the user equipments/RAN.

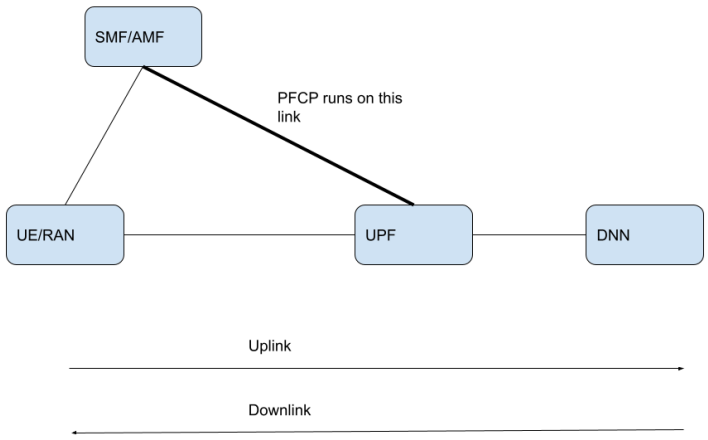

Figure 1.1: 5G Forwarding Plane

1.1.3 Data Network Name (DNN)

This network function is the gateway to the public Internet. All incoming packets from outside the local network are received by this NF and are subsequently forwarded to the user equipment through the UPF and the RAN.

1.2 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

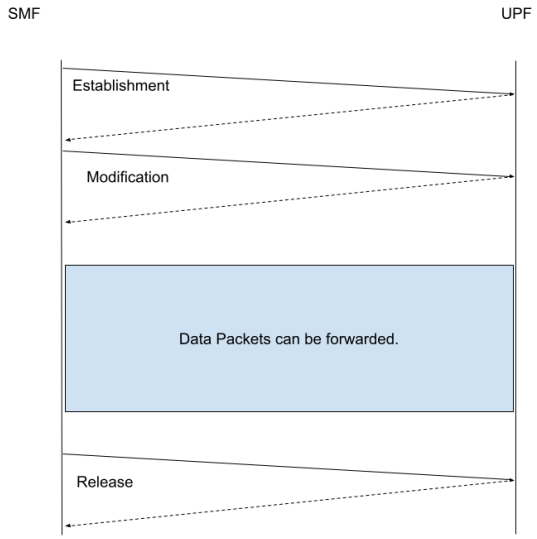

Figure 1.2: PFCP Session Messages

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-

- **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.

1.3 Organization

The report is organized as follows. The chapter 2 defines the problem statement. Chapter 3 discusses the design of the RAN emulator describing the core layout, how control plane messages are forwarded and what are the different modes of operation. Control plane latency calculation mechanisms are explained in the Chapter 4. Chapter 6 describes the simultaneous forwarding of control plane and data plane traffic. Chapter 5 describes the steps taken to clean the code and make it easy to extend for future developers. The results generated from control plane latency experiments with different models of the UPF and simultaneous transfer of control plane and data plane traffic are reported in Chapter 7.

Chapter 2

Problem Statement

- Feature Implementation
  - Control Plane Latency Calculation.
  - Simultaneous forwarding of Data Plane and Control Plane Traffic.
- Refactor, clean and make the RAN code easy to understand and further extend.

Chapter 3

RAN Design

3.1 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.

- **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.

3.2 Forwarding of Control Plane Messages

In a 5G-conforming RAN, the session related messages are sent by SMF to UPF. This RAN-emulator's primary role is that of a load generator. Session related messages - session establishment, modification and release messages are directly sent by RAN to the UPF. There are however some functions defined before the start of the load testing mode in the main function that interacts with AMF and to SMF through AMF. These are not removed as they act as a stub for interaction with AMF. They can be deleted after rigorous testing.

The session establishment, modification and release packets have a standard format of the header as well as the payload. These packets are defined as static arrays in the source file dpdk_ran.cpp - pfcpSessionEstablishmentRequestPacket, pfcpSessionModificationRequestPacket and pfcpSessionReleaseRequestPacket. Before sending these packets, the fields pertaining to a particular session are modified in these packets. These fields include source IP, tunnel end point identifier (teid) and session ids. There are counter variables defined in the dpdk class which maintain the next session related fields. Examples include nextUEIPEstablishment, nextTEIdEstablishment etc. These counters are updated after packet is sent. Only one control plane packet is sent at a time i.e. batch size is one.

3.3 Modes of Operation

The load generator can be used in different modes to characterize the different behaviors of the user plane function. The various modes are

- Setup Sessions and Send Data.
- Send only control plane traffic.

- Send Control Plane and Data Plane traffic simultaneously.
- Helper functions.
- QoS functions.

3.3.1 Setup sessions and send data

Initially sessions are setup by sending session establishment and session modification messages directly from the RAN to the UPF. Once the sessions are setup the data packets are forwarded from CORE_TX_START- CORE_TX_END (inclusive). Control plane latency is calculated during the initial setup for all the session related packets. Throughput is also logged in the log file. The number of UEs/sessions that are used for sending data per core is asked to the user. The sessions are partitioned in disjoint sets among the cores before forwarding.

3.3.2 Send only control plane traffic

This mode is used to send only control plane traffic and no data plane forwarding takes place. This is a synthetic traffic i.e. it does not represent any real world scenario. The main motive behind this mode was to saturate control plane and measure control plane handling capabilities of the different designs of the UPF. The latency and throughput values are logged in the file as earlier. This is an open load test in which session establishment, release and modification packets are sent one by one with a user provided inter packet delay (in us). These packets are sent from CORE_CP_TRAFFIC. The open load means that the RAN does not wait for response packets before sending the next packet. Ideally modification message should be sent only once the session establishment response is received. However, open load is a good approximation of the actual behavior and is easy to implement.

3.3.3 Send Control Plane and Data Plane Traffic Simultaneously

This feature is discussed in Chapter 6

3.3.4 Helper Functions

There is only one helper function right now. This helper function - Delay Estimator - helps in mapping inter batch delay with the data forwarding throughput of the load generator for a given number of cores and sessions. This mapping is used to set the rate of forwarding of the data packets. Intel 40Gbps NICs that we have do not have rate limiting APIs like 10Gbps NIC.

3.3.5 QoS Functions

These functions were developed to check the correctness of QoS algorithms deployed at the UPFs. These functions were tested on 10Gbps NIC systems and may require modification for other NICs.

For QoS testing, it was required to test whether the packet forwarding rate is actually limited by the algorithm. For this, data is forwarded at a low rate and at a high rate. The low rate is lower than the rate limit imposed on the sessions. The high rate is higher than the limit. So when the rate is higher, the output at UPF is limited to the rate limit.

Lower rate - 700 Mbps, High rate - 1200 Mbps, Rate limit -1000 Mbps. When the incoming rate at the UPF is 1200 Mbps, outgoing rate is 1000 Mbps if the QoS algorithm is correct.

Chapter 4

Control Plane Latency

4.1 Latency Calculation Algorithms

4.1.1 Key Concept

When packets are transmitted from RAN, the timestamp when the packet is forwarded is stored for the outgoing packet. When the response for the packet arrives at the receiving side of RAN, the difference in time values of the current time and the stored timestamp is calculated and then reported in the log file.

Storing of Timestamp

- **Data Plane** The packet identification field in the IP header is used to identify the packet. These latency packets are sent from the core CORE_RTT. All the established sessions/UEs are used for forwarding the traffic. When the packet is transmitted, a callback function stores the outgoing packet timestamp in a hashmap with packet id as the key. This field is generally used in the fragmentation and reassembly of packet data. The data plane latency packet throughput is substantially reduced by introducing sleep commands. The option of disabling the calculation of latency is removed as latency packets do not significantly affect any other metrics. Data plane latency is independently logged in a different column in the log file.

- **Control Plane** This requires deep packet inspection of control plane packets. These packets are sent from the core CORE_CP_TRAFFIC. There are

three types of control plane packets - session establishment, session modification and session release packets - all of them are used for measuring control plane latency. These packets have session ids stored at different offsets in the payload.

Earlier, a hash map was used for storing the timestamps. The use of hashmap slows down the call backs and unnecessary limits the rate of data forwarding.

Subsequently, fixed length array was used to store timestamps - **TSArray** . The array size is $65536 * 3$ and stores the value of timestamp register **rdtsc** of 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.

A callback function is registered on the transmit queue corresponding to CORE_CP_TRAFFIC which stores the timestamp in the hashmap.

4.1.2 Calculation of Latency

A single callback function is registered on the CORE_RX_POLL which receives the incoming packet.

- **Data Plane** The same outgoing packet is reflected by the DNN packet and stored timestamp is retrieved from the hashmap and the difference is the end to end latency of each packet.

- **Control Plane** Every control plane packet has a response packet - establishment response (51), modification response (53), release response (55). The control plane latency is the time period from when the request packet is sent and the response packet is received. The response packets have message type and either session IDs in their payload. For modification and deletion responses, the session ids have a difference of 3000. Using these information, the index in **TSArray** can be calculated and the corresponding timestamp can be retrieved.

4.1.3 Why do we need different techniques for control plane and data plane?

- **Why can't we use packet identifier for control plane packets?** Both types of packets are received on the same rx queue/core. So, only the packet identifier in ip header is not enough to differentiate control plane and data plane packets. And you will need deep packet inspection to differentiate among the two packets.
- **Why don't we inspect packet payload for data plane packets?** Data plane packet has no useful payload and when the ip header identification field is unused till now, we can use it.
- **Development Sequence** Data plane latency was implemented earlier when the packet id field was unused. Control plane latency calculation is done later.

4.2 Callback functions

Call back functions are registered on receive and transmit queues for latency calculation.

- **CPStoreTimestampCallback** Registered on CORE_CP_TRAFFIC for storing the timestamp of outgoing control plane packets.
- **DPStoreTimestampCallback** Registered on CORE_RTT for storing the timestamp of outgoing data plane latency packets.
- **CPLatencyCallback** Registered on CORE_RX_POLL for inspecting responses to control plane requests. The difference in the current time and the timestamp stored during the tx callbacks gives the latency values.
- **DPLatencyCallback** Also registered on CORE_RX_POLL for inspecting the mirrored data plane latency packets.

Chapter 5

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.

5.1 Naming of Variables

5.1.1 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.

5.1.2 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.

5.2 Stale Comments

5.2.1 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.

5.2.2 Resolution

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

5.3 Dead Code

5.3.1 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.

5.3.2 Resolution

Dead code mentioned in the issue sections is cleaned.

5.4 Deprecated Options

5.4.1 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.

5.4.2 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.

5.5 Copied Code

5.5.1 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.

5.5.2 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.

5.6 Global Variables

5.6.1 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.

5.6.2 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.

5.7 Directory Structure

5.7.1 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.

5.7.2 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.

5.8 Poor Refactoring

5.8.1 Issues

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

- The DRY 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.

5.8.2 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.

5.9 Unnecessary Offloads

5.9.1 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.

5.9.2 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.

5.10 Technical Improvements

5.10.1 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 number comes into play much before that and high latency packet throughput causes unnecessary callback overhead.
- The statements like **if (argc == 0)**, for loop running once, calling an internal function of the dpdk API when external call.

5.10.2 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.

Chapter 6

Data Plane + Control Plane Traffic

6.1 Algorithm

$n1$ 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.

6.2 Issues

- **Pthreads vs. Lthreads** There are two kind of threading models available - Pthreads and Lthreads.
  - **Pthreads** This model is fully implemented and works correctly when locks are used in both data plane and control plane functions. The names of control plane procedures to be used in this model end with 'Pthread'.
  - **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. This yielding happens 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.

Chapter 7

Results

Bibliography