An End-to-End DPDK-Integrated Open-Source 5G Standalone Radio Access Network: A Proof of Concept

Abhishek Bhattacharyya[1], Shunmugapriya Ramanathan[2,*], Andrea Fumagalli[2], Koteswararao Kondepu[1]

Abstract

The fifth-generation mobile networks are designed to accommodate billions of users worldwide with numerous applications and different service level requirements by providing significantly high data rates and availability with reduced latency. These requirements can be achieved by exploiting Next Generation - Radio Access Network (NG-RAN) architectures. One such famous architecture is Cloud Radio Access Network, whose Next Generation NodeB functions are physically decoupled into different entities — Radio Unit, Distributed Unit, and Central Unit. The Central Units are connected to the 5G Core Network, and all the components are likely to be virtualized and run on the commercial off-the-shelf hardware in the micro/macro data centers.

However, the inherent drawbacks of traditional kernel-based networking approaches, including context switches, interrupt handling, and memory copies, pose significant challenges in achieving the above challenges faced by the 5G mobile network operators. To address these overheads in the Kernel packet processing, Kernel bypass techniques like Data Packet Development Kit (DPDK) must be incorporated into the 5G architecture. In this work, we present the DPDK framework integrated into the publicly available open-source OpenAirInterface software modules. The design and implementation features are incorporated into the 5G radio access network modules, namely DU and CU, and the core network module, User Plane Function (UPF). The measurements taken in the 5G network show performance improvement when deploying the DPDK techniques over the network interface in the 5G midhaul and backhaul communication.

Keywords: 5G, Standalone, RAN, DPDK, Core Network, Midhaul, Backhaul.

1. Introduction

5G technology has the potential to revolutionize a wide range of industries and enable various usage scenarios due to its enhanced speed, capacity, low latency, and reliability. Some of the key 5G usage scenarios are Enhanced Mobile Broadband (eMBB), massive Machine-Type Communication (mMTC) and Ultra-Reliable Low Latency Communication (URLCC). eMBB is one of the most common and straightforward use cases for 5G. It offers significantly faster download and upload speeds than 4G LTE, making it ideal for streaming high-definition video, online gaming, and other data-intensive applications. In mMTC, 5G can support many low-power, low-data-rate devices simultaneously. This makes it suitable for connecting a wide range of IoT devices, such as smart meters, sensors, and wearable devices, which require long battery life and efficient use of network resources. URLLC applications require near-instantaneous response times, such as autonomous vehicles, remote surgery, and industrial automation, which benefit from 5G low latency and high reliability. URLLC ensures that data is transmitted and received with minimal delay.

Network Functions Virtualization (NFV) and Software-Defined Networking (SDN) are two key enablers that play an essential role in the 5G network deployment. NFV technology involves virtualizing network functions traditionally performed by specialized hardware commodities. It decouples the network function from the proprietary specialized hardware by virtualizing it and allows the software-based virtualized instance to run on the commercial off-the-shelf (COTS) servers. With the advances in general-purpose computing hardware, it is now feasible to run virtualized LTE and 5G network functions on the COTS servers. This feature provides tremendous benefits to telecommunication service providers, where multiple instances of virtualized 5G functions can run on the same high-volume server simultaneously, leading to higher flexibility, scalability, and cost efficiency. However, in the COTS server, a significant number of considerations need to be taken

*This work received funding from DST SERB Startup Research Grant (SRG-2021-001522), the SGNF project ("Reliability Evaluation of Virtualised 5G"). This work is also supported in part by NSF grant CNS-1956357.

*Corresponding author

Email addresses: 211011001@iitdh.ac.in (Abhishek Bhattacharyya), sxr173131@utdallas.edu (Shunmugapriya Ramanathan)

[1]Indian Institute of Technology Dharwad, Dharwad, Karnataka, India.

[2]Open Network Advanced Research lab, The University of Texas at Dallas.

into account to match the carrier-grade performance and reliability features of the highly optimized custom hardware platform.

When 5G Virtualized Network Functions (VNFs) are deployed on Virtual Machines (VMs) hosted in a cloud infrastructure, the amount of network traffic handled by a single server is expected to increase significantly. So when multiple VNFs run simultaneously, processing a significant number of packets per second may lead to performance challenges. As a result, the performance of the VNFs handling the client's requests depends hugely on the system's architecture features, such as memory access, task resource allocation, network I/O, etc. Considering the significance and importance of I/O performance in cloud infrastructure, various software and hardware optimizations are required to enhance the system's performance to improve the performance and efficiency of I/O (Input/Output) operations.

In addition, there are some inherent drawbacks of traditional packet processing when a packet arrives at the Network Interface Card (NIC). These drawbacks are in the form of severe performance bottlenecks faced by traditional network stacks due to the limitation of the hardware platform in the process of sending and receiving packets —- remains one of the significant challenges that need to be addressed. Various overheads in the form of *context switches*, *locking mechanisms on shared data structures*, *interrupt handling*, and *memory copies* have caused modern-day high-performance NICs to fail to achieve the desired line rate. Therefore, acceleration techniques becomes very necessary in 5G networks to achieve the desired level of QoS required by billions of users worldwide. Various software acceleration techniques are proposed in the literature that can enhance network performance. Data Plane Development Kit (DPDK) [1] is one such acceleration technique that bypasses the typically feature-rich – thus bulky – networking stacks realized in an operating system (OS) kernel, dropping unneeded features. DPDK is one of the most common kernel bypass techniques that provides a set of data plane libraries for implementing user space poll-mode drivers and making the communication between NIC and applications in user space possible without kernel involvement and eliminating kernel overheads, as discussed above.

In our proposed work, we incorporated one of the well-known DPDK frameworks named Accelerated Network Stack (ANS) to accelerate communications in the 5G User Plane (UP), which is responsible for the proper flow of data packets from the User Equipment (UE) to the Data Networks (DN). We aim to optimize user data communications on the 5G midhaul User Plane tunnel between DU and CU components and the backhaul User Plane tunnel between CU and UPF components to achieve the lowest possible end-to-end latency. This is beneficial considering the critical latency requirements imposed by various 5G use cases. The detailed paper contributions are highlighted as follows: (i)designing and integrating the DPDK framework on the publicly available open-source OpenAir-Interface (OAI) Radio Access Network and Core Network on the 5G midhaul and backhaul user data traffic communication; (ii) Control User Plane Separation (CUPS) implementation in the open source OAI to make end-to-end DPDK integration successful; (iii) Two different designs are considered based on the CUPS architectures defined in 3GPP 38.401, where multiple DUs are connected to a single CU on User Plane. Thus, there is a strong need for acceleration at the User Plane. This paper provides midhaul DPDK-based acceleration as a proof-of-concept to address the acceleration at the User Plane. In addition, the paper provides insights and design challenges on end-to-end DPDK acceleration; (iv) The paper also provides extensive experimental evaluation by considering the real-time video transmission impact on the design implementation and also the impact of different network parameters — Throughput, Latency, Jitter, and Packet loss. (v) The obtained results are compared with Kernal-based implementation and observed the performance improvements when deploying the DPDK-based techniques over the network interface in both midhual and backhaul communication.

2. Related Work

In the past extensive research and studies have been done in the field of DPDK for achieving enhanced performance. [2] developed UDPDK, a DPDK-based middleware to address the critical shortcomings of existing kernel-based networking solutions to achieve high-performance networking. Experimental results on the testbed showed that by incorporating the UDPDK framework in their applications they can achieve a reduction of 69% in the overall end-to-end latency. Some well-known DPDK frameworks exist in literature (mTCP, F-Stack, etc) which are discussed in Section 3.1. In [3], a DPDK-based NFV architecture is presented where monolithic VNFs are disaggregated into lightweight "micro" VNFs, enabling a finer-grained resource allocation and reducing redundancy in the network stack. Authors in [4] are able to obtain a latency reduction of nearly an order of magnitude by the adoption of DPDK in-game servers. They showed how latencies can be reduced by using the packet processing framework DPDK to replace the operating system's network stack. In [5] authors are able to achieve a significantly higher packet throughput performance using Single Root I/O VIrtualization (SR-IOV) and DPDK in unison compared to the traditional processing with the Native Linux Kernel network stack.

In the field of Long Term Evolution (LTE) and 5G networks, researchers started exploring the usefulness of DPDK to evaluate performance improvements. In [6], the authors investigated USRP Hardware Driver (UHD) driver with DPDK in a Software Defined Radio (SDR) based environment and applicable to SDRs using UHD and an Intel NIC to transfer radio samples between the SDR and the host computer. The results presented using the iperf3

network performance application showed performance improvements when a Kernel bypass framework like DPDK is used to facilitate data transfer over the network interface between the SDR application and the radio hardware.

To the best of our knowledge, prior research has not demonstrated the integration of DPDK frameworks within open-source repositories that implement the 3GPP 5G protocol stacks, such as OpenAirInterface (OAI). This paper presents a comprehensive integration of the renowned DPDK-ANS framework into the OAI repository, encompassing the DU, CU, and CN components. Our findings reveal a significant performance boost, with increased throughput and decreased latency, compared to the conventional Kernel-based implementations of the 5G protocol stack.

3. Background

3.1. DPDK Overview

Data Plane Development Kit (DPDK) is an open-source software project managed by the Linux Foundation [1]. DPDK consists of a set of libraries and user space poll mode drivers for faster packet processing and provides the framework and Application Programmable Interfaces(APIs) for high-speed networking applications. As shown in Fig. 1, DPDK process packets in the user-space without involving the kernel network stack by a bypass mechanism, thus eliminating the overheads involved in kernel packet processing. One such overhead is the *interrupt mechanism* which happens when there is a context switch from kernel space to user space and vice versa. When a packet is received by the Network Interface Card (NIC), it passes to the receive queue or RX. From there, it gets copied to the main memory via Direct Memory Access (DMA) mechanism. Afterwards, the system needs to be notified of the new packet and pass the data into a specially allocated buffer which is dynamically allocated by the Linux Kernel for each packet that is received by the network. To perform the aforementioned operation, Linux Kernel uses an interrupt mechanism to perform a context switch from user space to kernel space for every packet which arrives in the system for a kernel read operation. Once the processing is done, the packet needs to be transferred to the user space which involves a context switch back from the kernel space to user space. The two main advantages of using DPDK: (i) to eliminate *this context switching back and forth* leading to minimized overheads compared to the Linux Kernel; (ii) to eliminate the need for dynamic allocation of memory buffers when a packet arrives and also eliminates the need for packet copy between kernel and user space. This is achieved by making the network card interact with DPDK special drivers and libraries without involving kernel overhead.

3.2. DPDK Framework

There are various high-performance user-space network stacks developed on top of high-performance low-level frameworks like DPDK or netmap and, sometimes relying on code "borrowed" from widespread operating systems like Linux or FreeBSD. mTCP [7] is one such highly scalable user-level TCP stack designed for scalability on multicore systems. F-Stack [8] is another open-source high-performance network stack based on DPDK which provides ultra-high network performance that the network card can achieve under full load supporting a million number of connections. This framework also provides an Epoll/Kqueue interface that allows various kinds of applications to easily use F-Stack. Another well-known popular stack named FD.io [9] Transport Layer Development Kit (TLDK) includes a set of libraries for L4 protocol processing (UDP, TCP etc) for both IPV4 and IPV6. It is maintained to support the development of Vector Packet Processing (VPP), a high-performance software switch. Since it is highly focused to meet VPP requirements, its API is not meant to be compatible with POSIX sockets, limiting its applicability. SeaStar [10] is another event-driven framework with its own TCP/IP stack allowing us to write non-blocking, asynchronous code in a relatively straightforward manner. The Linux TCP/IP stack is also ported to the DPDK platform in IPAugenblick [11] having POSIX-like API, which relies on a background process to act as a glue component between poll-managed devices and applications. However, this project has not seen any active development since 2016. Accelerated Network Stack (ANS) [12] is one of the most popular native TCP/UDP and IP protocol stack that provides a user-space TCP/UDP stack for use with Intel DPDK. UDPDK [2] is another UDP stack developed on top of DPDK similar to ANS but lacking in the ability to provide non-blocking API (such as epoll, event) support. 5G user plane relies on UDP and therefore our initial focus is limited to ANS, UDPDK, TLDK in FD.io frameworks that support UDP/IP stack over DPDK.

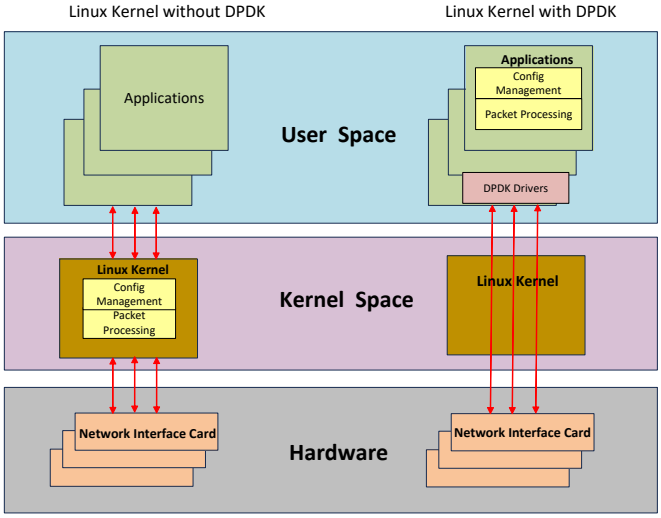

Figure 1: Kernel vs DPDK Mode

4. Design

4.1. 5G User Plane and GTP-U Tunnel

Fig. 2 shows 5G RAN User Plane Protocol Stack that uses GPRS Tunneling Protocol (GTP-U) for carrying the Protocol Data Units (PDUs) for the end-users. In 4G LTE and 5G system, to provide mobility to the UE and cope with the resulting network topology dependencies, the UE uplink and downlink IP packets are routed through a GTP tunnel. Tunnel Endpoint Identifier (TEID) values are mutually exchanged between the base station and UPF to ensure the proper flow of data traffic. As shown in Fig. 2, the 5G User Plane uses UDP for the transport layer protocol that is placed in the GTP header. For instance, considering the UE uplink communication, the IP data packet is first encapsulated at the gNB_vDU by adding its IP/UDP/GTP header and transmitted in the GTP tunnel to reach the gNB_vCU. The gNB_vCU replaces the outer header with its IP/UDP/GTP header and sends it to the User Plane Function (UPF). The UPF decapsulates the outer header and passes the original UE IP data packet to the Internet/Packet Data Network (PDN). The GTP-U communication, along with the UDP/GTP header addition is illustrated in Fig. 3. The OAI software modules (gNB_vCU, gNB_vDU and UPF) implement the above-mentioned data plane connectivity by using the Linux Kernel module. Our design strategy involves using DPDK with higher layer user-space stack to enable the GTP user plane communication for a better performance.

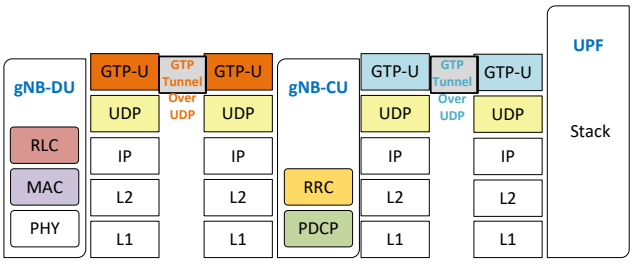

Figure 2: 5G RAN User Plane Protocol Stack

4.2. DPDK Integration with Open Source Stack (OAI) 5G: Challenges Encountered

The following section lists the integration challenges faced by the user-space stack selection (for DPDK) and integrating with the 5G standalone module (i.e., OAI 5G RAN, SRS 5G RAN).

- Socket IO Event Notification: 5G user plane uses the UDP socket programming for communication among the modules, namely gNB_vDU, gNB_vCU and UPF. There are several ways to handle the event notification between the server and the client. Epoll is a Linux kernel system that calls for a scalable I/O event notification mechanism [13]. The traditional select and poll based system calls operate in O(n) time complexity to poll for the IO events. For a demanding application (i.e., 5G network stack) that has a larger number of watched file descriptors, epoll based event notification mechanism provides a better performance improvement. The epoll operators in O(1) [14] time complexity by monitoring multiple file descriptors to check if any I/O event is possible on any of them. The open-source 5G modules make use of epoll based event notification for the GTP user data communication. From the available open-source UDP over DPDK user-space stacks namely ANS, UDPDK and TLDK, TLDK are not compliant with the traditional BSD Socket based API. And, UDPDK does not support epoll based event notification. Since ANS over DPDK works on both TCP and UDP-based communication and supports the epoll event notification, we selected ANS over DPDK for integration with the 5G stack.
- *CMake/Make Build System*: CMake is cross-platform free and open-source software for build automation, testing, packaging and installation of software by using a compiler-independent method [15]. CMake is driven by the *CMakeLists.txt* files written for a software project. CMakeLists.txt file contains a set of directives and instructions describing the project's source files and targets (executable, library, or both). The open-source 5G stack (OAI, SRS 5G) uses *CMakeLists.txt* file to compile and build the executable files. In contrast, ANS over DPDK uses Makefile to build the executable files from the source files — which have to include numerous Makefiles [16].

  Our challenge comes in the way of integrating *Makefile* from ANS to *CMakeList* in OAI 5G to use the ANS API calls for the GTP-based UDP communication.

  We started with the integration of ANS header and library files to the OAI *CMakeLists.txt*. As the first step of integrating the DPDK-ANS APIs in the OAI, it is required to initialize the DPDK-ANS Socket by the function call *anssock_init*. But when the ANS socket initialization API was called, it led to unsuccessful initialization of DPDK tail Queues. Tail queues are data structures used by DPDK to manage different types of objects efficiently and are a fundamental part of DPDK's runtime environment used to store and manage its data structures. The root cause of this problem was primarily due to *CMakeLists.txt* not being able to include the dependant DPDK *Makefiles* which generates *.map* files along with the target executable for the proper functioning of the DPDK ANS libraries. To overcome this issue, we developed our own *Makefile* to create our target executable for gNB-CU and gNB-DU which was earlier created by the *CMakeLists.txt* file in OAI. With the proposed Makefile, we are able to include other numerous dependant DPDK Makefiles in our custom

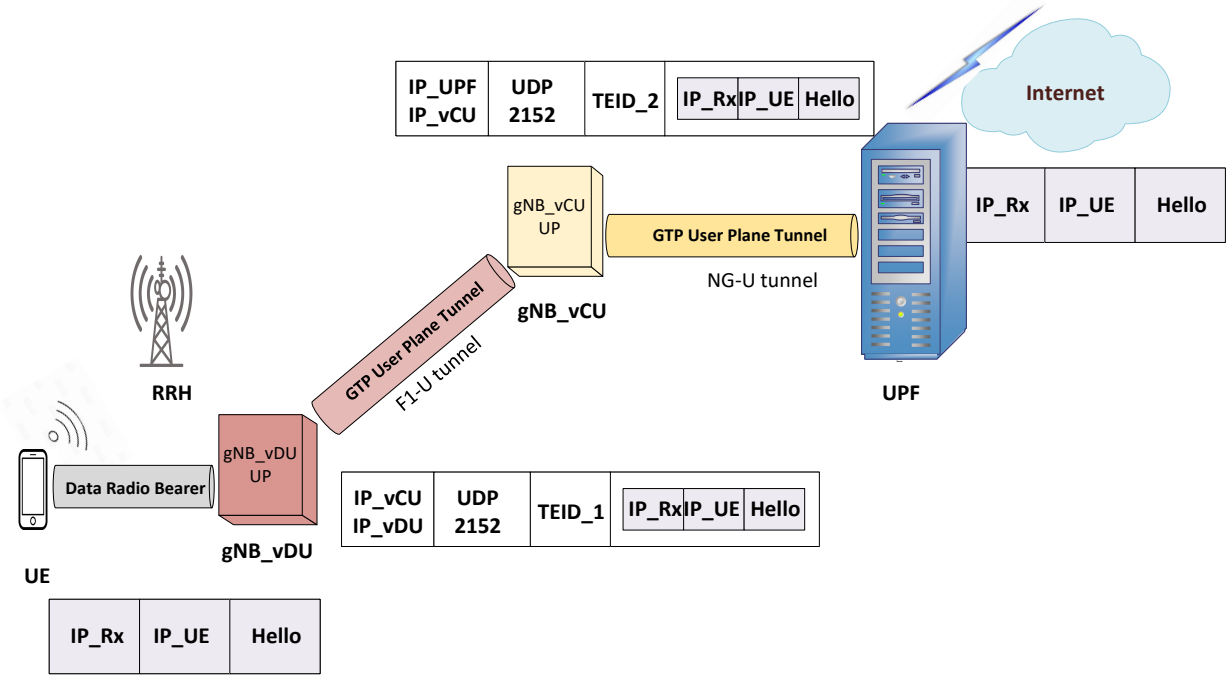

Figure 3: GTP Tunnel Uplink data traffic

Makefile to generate the *.map* executables along with our gNB target executables.

- *Control/User Plane Separation*: Control/User Plane Separation (CUPS) was another challenge we faced while integrating DPDK libraries into our publicly available open-source OAI software. The OAI uses a single interface for transferring both Control Plane and User Plane-oriented messages between the gNB-CU and gNB-DU sides. But our focus is to accelerate the User Plane communications using DPDK and let the signalling/control oriented messages use the traditional in-kernel networking stack. To achieve this, two different interfaces for the Control Plane and User Plane oriented message transfer are needed so that the Control Plane interface is bound to the kernel drivers using the traditional kernel stack and the User Plane is bound to DPDK poll mode drivers (VFIO). To achieve the aforementioned objective, CUPS related code changes are contributed by us at gNB-CU and gNB-DU sides, respectively.

  The architecture with these dual interfaces is used for implementing the DPDK (as shown in Fig. 6). To make CUPS integration successful, the following contributions are introduced in the OAI source code: (i) *Port Number and IP address:* Port Number is used to create socket file descriptors for control plane-oriented message transfers. IP Address specifies the address for these socket file descriptors. These parameters are incorporated in the configuration file of OAI and code modifications are made in the *gnb_config.c* and *gnb_paramdef.h* to allow communication between the gNB-CU and gNB-DU for the signaling messages. The gNB-CU and gNB-DU leverage these configuration parameters to create a socket for signaling-oriented message transfer. Specifically, the function *send_sctp_init_req* in the F1AP file is responsible for creating this socket. (ii) *gNB-CU modifications:* For accelerating data transfer using DPDK, the existing parameters from the configuration file are utilized. These sockets are distinct from the signaling-oriented ones and optimized for efficient User Plane communication. (iii) *gNB-DU modifications:* Similar steps are followed for the gNB-DU side, albeit with some variations. In the *f1ap_du_task.c* file, the two parameters defined in the files *gnb_config.c* and *MACRLC_nr_paramdef.h* are used. The function responsible for creating sockets in the gNB-DU is send_sctp_association_req. This subsection highlighted the CUPS implementation specifically related to the DPDK, however, the detailed information on CUPS design and implementation is discussed extensively in [17].

4.3. General Design

Fig. 4 shows the GTP User Plane data communication in the uplink direction. Once the UE is attached after the RRC connection setup, a default radio bearer (tunnel) is created exclusively for the UE internet communication. When the UE sends an IP packet to the internet, a sequence of actions is performed in various modules with our emphasis on the GTP tunnel part, as simplified below.

- In the *gNB_vDU* module:

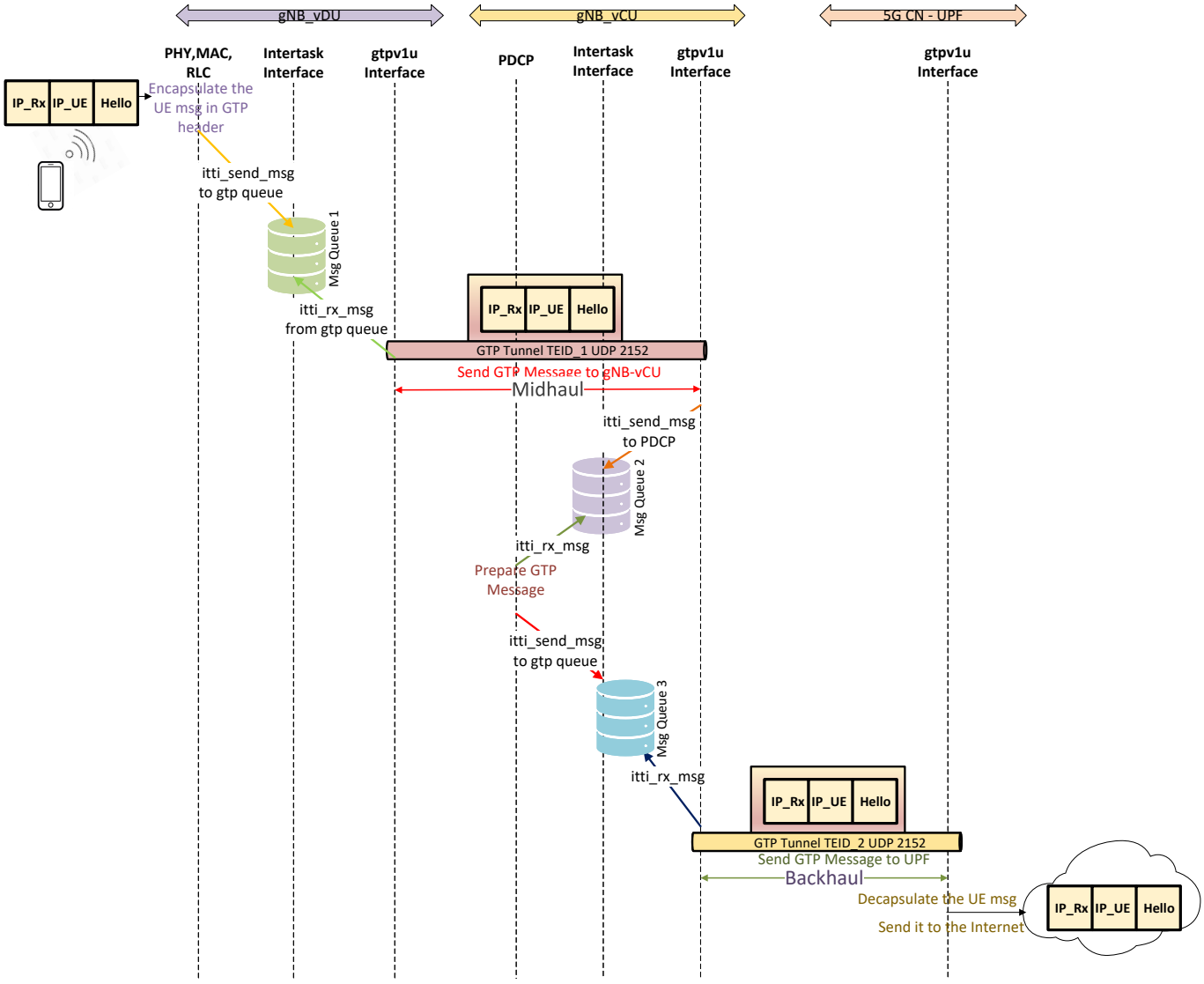

Figure 4: GTP User Plane data Communication in the Uplink direction

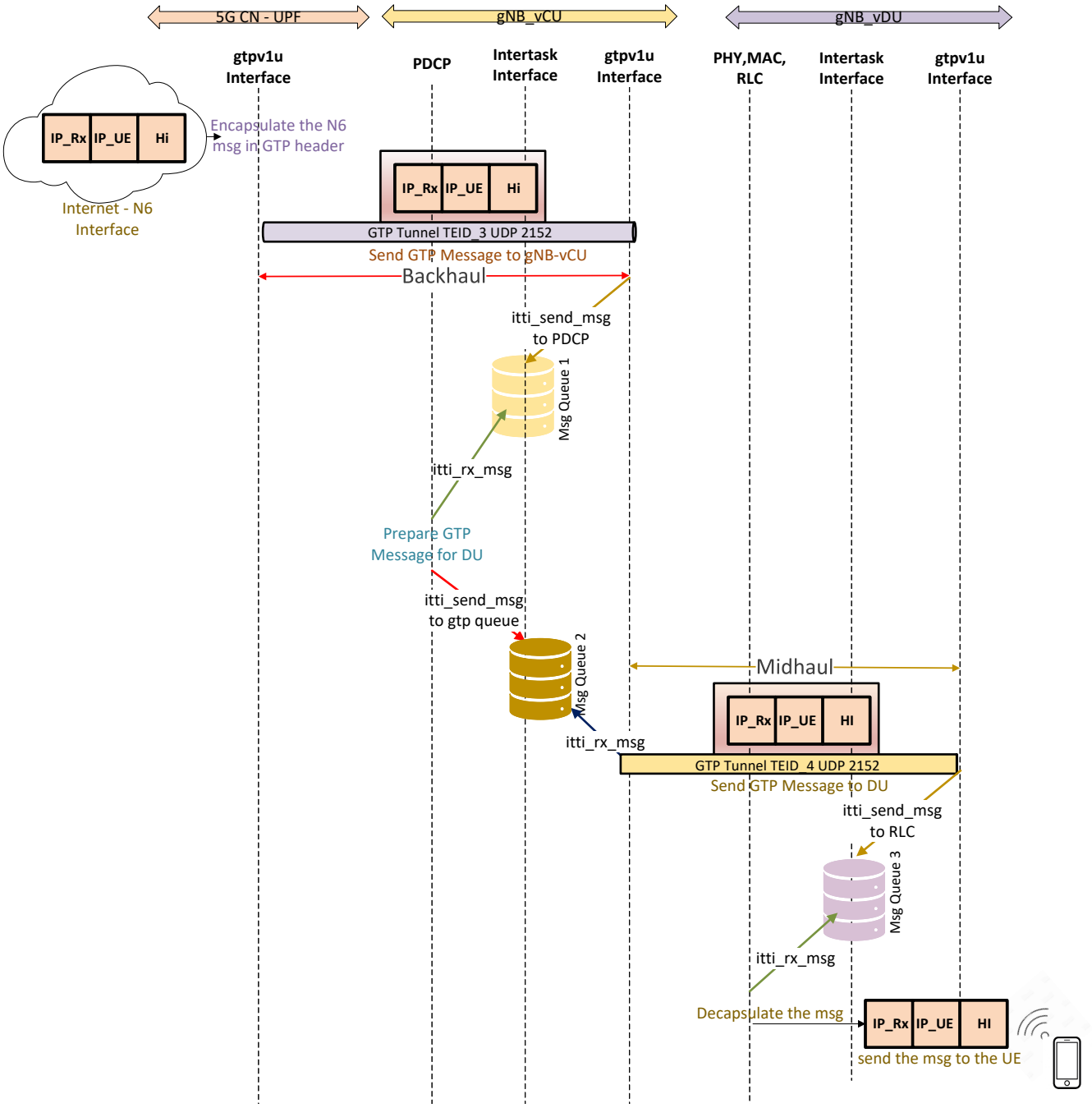

Figure 5: GTP User Plane data Communication in the downlink direction

1. PHY layer receives the UE message.
2. After the internal processing by the MAC and RLC layers, it encapsulates the UE message in a GTP header, as detailed in Sec. 4.1.
3. In the interprocess communication and synchronization handling, the RLC task sends the message to the GTPV1u handler through the message queue
4. GTPV1u handler receives the encapsulated message in the gtp message queue
5. GTPV1u handler sends the message to the gNB_vCU through a tunnel as an external event. The tunnel is identified by the corresponding Tunnel End Pointer Identifier (TEID).

- In the *gNB_ vCU* module:
  1. GTPV1u handler receives the message from the gNB_vDU module as an external event (from the tunnel).
  2. GTPV1u handler sends the message to the PDCP layer through the message queue.
  3. PDCP layer prepares the GTP header information and informs the GTPV1u handler to forward the message to the 5G Core Network.
  4. GTPV1u handler sends the message to the 5G_CN through a tunnel as an external event.

- In the *5G_ CN* module:
  1. The UPF of 5G_CN receives the GTP message from the gNB_vCU module.
  2. It decapsulates the original UE message from the GTP header.
  3. Sends the UE message to the Internet.

Similarly, to send an IP message from Internet to the UE, the following sequence of actions will be performed in the downlink direction as shown in Fig. 5.

- In the *5G_ CN* module:
  1. The UPF receives the message from the Internet.
  2. It encapsulates the Internet message into the GTP header with the TEID information.
  3. It sends the message to the gNB_vCU through the GTP tunnel.

- In the *gNB_ vCU* module:
  1. The GTPV1u handler receives the GTP message from the 5G_CN.
  2. It forwards the message to the PDCP layer.
  3. The PDCP layer changes the GTP header information and sends it back to GTPV1u handler.
  4. The GTPV1u handler sends the message to the gNB_vDU through the GTP tunnel.

- In the *gNB_ vDU* module:
  1. The GTPV1u handler receives the GTP message from the gNB_vCU module.
  2. It forwards the message to the RLC layer.
  3. The RLC layer decapsulates the GTP message.
  4. After the internal layers processing, the Internet message is sent to the UE.

In this GTP user data communication, the gNB_vCU plays a critical role in receiving GTP messages either from the gNB_vDU and/or from the 5G_CN UPF. Therefore, the GTPV1u handler will always be in a blocking call to check for any external event from gNB_vDU and 5G_CN UPF. Two design strategies are considered to realize DPDK based 5G user data communication.

The following subsections describe the considered DPDK designs, namely Hybrid DPDK Model and End-to-End DPDK Model.

4.3.1. Design 1 — Hybrid DPDK Model

In this model, we envisioned the DPDK integration in the midhaul interface (the GTP F1-U tunnel) between the gNB_vDU and gNB_vCU modules. This model is referred as hybrid since for the same user-plane, the gNB_vCU communicates with the gNB_vDU using the DPDK APIs using DPDK poll mode driver and the gNB_vCU communicates with the 5G_CN UPF using the Kernel APIs using Kernel network driver. This hybrid model acts as a proof of concept of DPDK integration in the 5G midhaul interface.

The gNB_vCU design complexity involves the GTPV1u handler polling for vDU messages using the DPDK API call and polling for CN messages using the different Kernel API call. Therefore, a single blocking-based epoll mechanism will not work in this hybrid module. This hybrid model is envisioned using the timed-based epoll wait mechanism since the GTP message could be received from either of the two adjacent modules for the vCU. The GTPv1U handler module is designed as follows in the hybrid mode:

1. Timeout Configuration Option - wait time to receive GTP tunnel messages from the neighboring modules
   - kernel epoll_wait timeout addition -
   - ANS over DPDK epoll_wait timeout addition
2. If EPOLL_IN is enabled in kernel epoll_wait, receive the GTP message from the CN through the kernel API receive command.
3. If EPOLL_IN is enabled in ANS DPDK epoll_wait, receive the GTP message from the distributed network (gNB_vDU) through ANS API receive command.
4. When receiving an internal event from the message queue, check the destination address of the GTP message

- If the destination address matches with the 5G_CN address, send the GTP message through the kernel API send command.
- If the destination address matches with the gNB_vDU address, send the GTP message through ANS API send command.

Note that the interprocess communication uses the semaphore based epoll_wait and it will be based on the kernel system call for the communication across multiple tasks within the same module.

4.3.2. Design 2 — End-to-End DPDK Model

In this model, we envisioned the DPDK integration for both the midhaul interface (the GTP F1-U tunnel) and the backhaul interface (the GTP NG-U tunnel). This model is called end-to-end since all three modules (gNB_vDU, gNB_vCU and the 5G CN UPF) have the DPDK integration enabled for the GTP tunnel communication. As shown in Figs. 4 and 5, both the GTP tunnels in the midhaul and backhaul network communicate using the DPDK poll mode drivers to accelerate the packet processing improvement to a greater extent. In this design, the ANS over DPDK integration is made in all three modules with the GTP socket programming communication changes along with the CMakeFile changes. The UPF in the 5G CN communicates with the SMF using the standard kernel API calls and with the CU socket using the DPDK API calls. To achieve this separation, two separate interfaces are created in the UPF container — one for SMF communication and another for CU communication. As mentioned in sec. 4.3.1, the interprocess communication still makes use of kernel system call for the semaphore-based epoll_wait, since these messages are internal and will not interact with the NIC card for any performance optimization.

5. Implementation

The design in Section 4.3 is implemented by extending *OAI*[3] — an open-source software implementation of the 5G Standalone project. OAI project implements the 3GPP technology on the general purpose x86 computing hardware combined with the software-defined radio cards. . Our proof-of-concept implementation is open-source available in Box[4]. Most of the DPDK support-related code changes in userspace are done in the GTPv1-U (GPRS Tunneling Protocol v1 - User) folder. GTPv1-U exchanges user data over GTP tunnels between network nodes. It uses User Datagram Protocol (UDP) with the IPv4/IPv6 support. The version history file lists changes applied to all other files to achieve DPDK integration support.

[3]https://gitlab.eurecom.fr/oai/openairinterface5g
[4]https://app.box.com/folder/225989759397

5.1. GTP Tunnel with DPDK Integration between DU and CU - Hybrid Mode

In the hybrid mode, the DPDK integration is realized only on the midhaul User Plane communication. The CU code changes are incorporated to communicate in the DPDK mode with the DU and in the kernel mode with the CN module as shown in Algorithm 1. The gtp_interface module in the gtpv1U is altered to provide this hybrid support based on the *IP Address* and *Port Number* parameters read from the configuration file (see line 1). After getting the peer Port information, the Port Number is checked whether it matches with the DU or the CN Port (see lines 4-6). If the peer Port is matched with the DU Port, then by using DPDK ANS APIs — create epoll file descriptors, create and bind sockets for UDP communication, and then add the sockets to them (see lines 7-9). Otherwise, the conventional kernel-based ones for CN ports are created (see lines 11-13). Finally, the *socket_create_status* is assigned to the output *staus* variable (see line 15).

Algorithm 1 UDP ServerSocket Initialization

1: **Input:** peerAddress Info
2: **Output:** status
3: **Function Name**: udpServerSocket
4: Read the $DUIP$ and $DUPort$ details from the config file
5: Get the $peerPort$ from peerAddress structure
6: **if** $peerPort == DUPort$ **then**
7: Create anssock epoll
8: Create and bind ans socket for UDP communicate
9: Create EPOLLIN event and add the ans socket to the EPOLL_CTL_ADD
10: **else**
11: Create kernel epoll
12: Create and bind kernel socket for UDP communicate
13: Create EPOLLIN event and add the kernel socket to the EPOLL_CTL_ADD
14: **end if**
15: $status = socket\_create\_status$

After the socket initialization, whenever a message from the PDCP layer, the function call *gtpv1uCreateAndSendMsg* is invoked (see in line 3) as described in Algorithm 2. Upon triggering the function, the DU IP address and DU port details are obtained from the configuration file (see line 4), and checks to select either DPDK-based UDP tunnel or Kernel-based UDP tunnel (see line 5). If the peer IP and Port is of DU then DPDK ANS API calls are used for sending the packets (see line 6) otherwise Kernel-based API calls are used (see line 8). Finally, the *send_status* is assigned to the output *status* variable (see line 10).

The *gtpv1uTask* function call as stated in Algorithm 3 is based on the epoll wait mechanism. The function call iteratively checks if the *IntertaskInterface_event* is true

Algorithm 2 GTPV1U Send Message

1: **Input:** peerIP, peerPort, teid, msg
2: **Output:** status
3: **Function Name**: gtpv1uCreateAndSendMsg
4: Read the *DUIP* and *DUPort* details from the config file
5: **if** *peerIP* == *DUIP* AND *peerPort* == *DUPort* **then**
6: *anssock_send* command is initiated on the tunnel
7: **else**
8: *kernel_send* command is initiated on the tunnel
9: **end if**
10: *status* = *send_status*

Algorithm 3 GTPV1U Receive Message

1: **Input:** None
2: **Output:** None
3: **Function Name**: gtpv1uTask
4: **while** true **do**
5: **if** IntertaskInterface_event is TRUE **then**
6: Check the msg-type received from PDCP layer
7: **if** msg-type is GTPSendMsg **then**
8: Call GTPV1U Send Message Function
9: **end if**
10: **end if**
11: Read the Kernel_timeout and DPDK_timeout from the configuration file
12: Wait for CN messages with the kernel_timeout
13: **if** Kernel_epoll_events **then**
14: Call Kernel_receive command and process the received message
15: **end if**
16: wait for DU messages with the DPDK_timeout
17: **if** DPDK_epoll_events **then**
18: Call anssock_receive command and process the received message
19: **end if**
20: **end while**

— the type of message received from the PDCP layer (see lines 4-6). If the type of the message is *GTPSendMsg*, then the CU calls *GTPV1U Send Message Function* (defined in Algorithm 2) to send packets to either DU/CN (see lines 7-8). Otherwise, the CU waits to receive messages from either DU/CN.. As stated above, our implementation added the timer values for each event wait in the configuration file. Based on the timer configured, the CU *gtpv1uTask* function waits for that predetermined period on the network nodes (see line 11). After a designated period, when the CU sends data packets to either CN/DU, subsequently calls either the Kernel-based APIs to receive and process UDP packets from CN (see lines 12-14), or the ANS DPDK APIs to handle UDP packets received from DU (see lines 16-18). The steps outlined in lines 8-16 accomplish the desired functionality and procedure for receiving and processing these packets accordingly.

Note that similar types of functional changes in the send and receive are performed on the DU side of the gtp code to support the DPDK-based midhaul GTP communication.

5.2. GTP Tunnel with DPDK Integration among DU, CU and CN — End-to-End DPDK Mode

In the end-to-end DPDK mode, the DPDK integration is realized on both the midhaul and backhaul user data communication. The UPF module in the CN is modified to accommodate DPDK-based GTP tunnel communication with the CU module. The docker build command builds the docker container image for the UPF module. The CU module interacts with both DU and UPF of the CN using ANS DPDK communication. In the CU module, Algorithms 1-3 are modified in the GTPv1u task to accommodate the socket creation, bind, send, and receive using the ANS DPDK libraries running in the user space to communicate with DU and UPF. In the UPF module, the GTPv1u task uses the ANS DPDK API calls for the UDP communication, as mentioned below. After reading the IP and port details from the yaml file, it creates and binds kernel socket for Control Plane communication with the SMF and ANS DPDK socket for GTP User Plane communication with the CU (see lines 4-6). Once the *IntertaskInterface_event* is true, UPF checks the message type received. If it is of type *GTPU*, then *GTPU ANS DPDK Send Message Call* is called else *Kernel Send Message Call* is called (see lines 9-14). Otherwise, the UPF module waits for a predefined amount of time based on the *Kernel_timeout* and *DPDK_timeout* values configured (see line 17-19). If the *DPDK_epoll_events* is enabled it calls ANS DPDK API to receive the message else it receives and processes the message using the normal Kernel-based API (see lines 20-23).

6. Performance Evaluation

6.1. Experimental Setup

This section reports the experimental setup for testing the end-to-end connectivity in our DPDK-enabled UTD lab testbed. The testbed shown in Fig. 6 includes a User Equipment, a gNB-DU, a gNB-CU, and a 5G CN component. These 5G RAN and CN components are implemented using the OAI software packages. The radio functionality is implemented using an RF simulator but can also be extended using software-defined radio boards such as NI B210 or USRP N310 with a Faraday cage. The OAI 5G-NR provides different split options as defined in [18], and we make use of option 2 split between gNB-DU and gNB-CU, with the Control and User Plane separation. The F1-U interface and the N3 interface in Fig. 6 are connected using a switch to have the same CU NIC communicate with the DU and the UPF NICs of the User Plane.

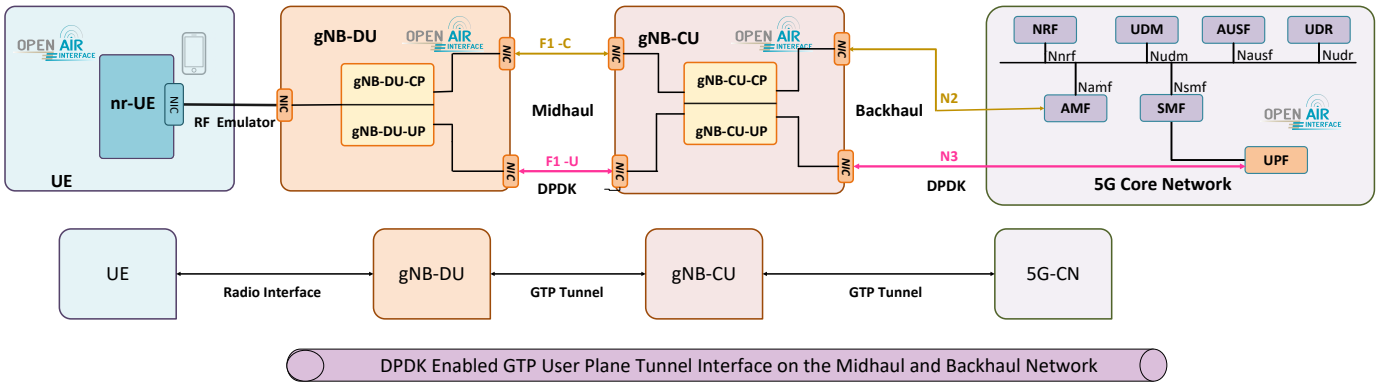

Figure 6: Experimental Setup

Algorithm 4 UDP_GTPU l4 stack

Input: None
Output: None
Functions Name: create_socket, udp_read, send_to
Read the *IP* and *Port* details from the yaml file
Create and bind Kernel socket for SMF communication
Create and bind ANS DPDK socket for GTP user plane communication
Enable DPDK epoll event
while true **do**
    if IntertaskInterface_event is TRUE **then**
        Check the msg-type received
        if msg-type is GTPU **then**
            Call GTPU ANS DPDK Send Message Call
        else
            Call Kernel Send Message Call
        end if
    end if
    Set the Kernel_timeout and DPDK_timeout
    Wait for SMF messages with the kernel receive
    Wait for CU messages with the DPDK_timeout
    if DPDK_epoll_events **then**
        Call anssock_receive command
    end if
    Process the received message
end while

The implemented 5G Core Network (CN) components include: Unified Data Repository (UDR), Unified Data Management (UDM), Authentication Server Function (AUSF), Network Repository Function (NRF), Access and Mobility Management Function (AMF), Session Management Function (SMF), and User Plane Function (UPF) (i.e., SPGW-U). For the backhaul interface to support the DPDK mode, the UPF module is updated to enable the GTP-U communication using ANS-DPDK API calls with the gNB-CU module. These functions are deployed as multiple docker containers using the OAI 5G CN. The system configuration details and 5G network parameters are reported in Table 1 and Table 2.

Table 1: System Configuration

Description	**UTD Lab**
Product	APEX S3
CPU	16-core Intel® Core™ i7
Intel Architecture	Skylake
RAM	32GB DDR5
Memory	500GB SSD
Kernel NIC Drivers	ixgbe, e1000e
DPDK Driver	VFIO
OAI RAN	Develop version
OAI CN	v1.2.1

6.2. Use-case Scenario

Video traffic is one of the most essential use case scenarios in the 5G network, exerting a substantial influence on resource consumption and network efficiency. For instance, live video streaming has gained significant traction recently, especially during the coronavirus pandemic. One of the most essential applications of video streaming is Online Learning, which requires streaming video and audio to deliver content to online learners without buffering and packet loss and at a higher bit rate. To mimic this type of traffic, our study uses UDP transmission of video files from the CN to the UE using the open-source, cross-platform multimedia player VLC application whose

Table 2: 5G Network Parameters

Description	Value
NR Release	3GPP Release 16
NR Band	Band 78
NR Frequency	3.6 GHz
RAN type	5G standalone gNB
CU/DU split	Option 2
Physical Resource Block (PRB)	106
Radio Channel Bandwidth	40 MHz
Midhaul Capacity	10 Gbps Ethernet
Backhaul Capacity	1 Gbps Ethernet
UE	OAI based 5G SA UE

settings are shown in Table 3. Video streaming is started at the CN once the end-to-end connection is established and GTP tunnel is created from UE to the CN. The content bitrate of the video application running in the UE is monitored in two different scenarios - one in ANS-DPDK enabled testbed and the other in the Kernel one.

Table 3: Codec Configuration in the VLC

Description	CloudLab
Video Codec	MPEG-4-V
Video resolution	320x240 px
Buffer dimensions	320x256 px
Frame rate	15 fps
Audio Codec	ADTS
Audio bit rate	128 kbps
Communication	UDP
Audio Sample rate	48000 Hz
Stream output muxer caching	1500 ms

6.3. Results

This section reports the experimental data collected from the UTD lab testbed. Each experiment is repeated ten times evaluating performance metrics such as throughput, jitter, packet loss, round trip time, CPU utilization on midhaul, backhaul, and end-to-end 5G network on DPDK and Kernel mode totaling three hundred trials. Results are reported with the mean value along with the 95% confidence interval accounting for the stochastic variations due to the network, I/O, and processing delays. Experiments with ANS use DPDK version 18.11 and the DPDK implementation of the Virtual Function I/O driver, and for the Kernel POSIX socket API, the device driver used was ixgbe. ANS application is started as the primary process and OAI code is started as a secondary process to initiate the ANS API calls.

6.4. Midhaul Performance Analysis - Hybrid Mode

In the 5G standalone testbed, three interfaces are configured in the CU application, accounting for Control Plane communication with the DU, User Plane communication with the DU and AMF communication with the CN. In this hybrid mode, DU Control Plane and AMF communication calls are initiated using the kernel API calls. For the midhaul performance evaluation, the results are captured by initiating the User Plane tunnel communication in the kernel mode. Then, the next set of results is captured with the User Plane tunnel communication with the ANS DPDK API calls. Both scenarios are tested with the 10G Network Interface Card for the User Plane communication.

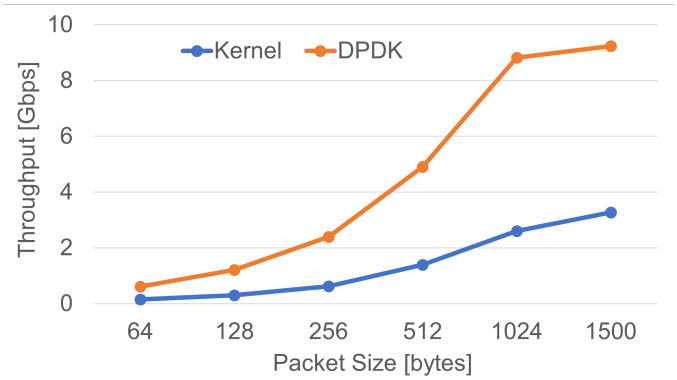

Figure 7: 5G Midhaul Throughput Comparison between POSIX Socket and ANS-DPDK

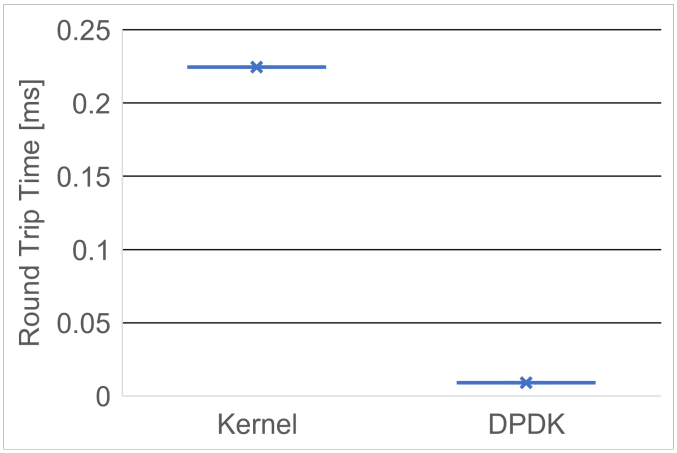

Figure 8: 5G Midhaul RTT Comparison between POSIX Socket and ANS-DPDK

Fig. 7 reports the throughput results captured in the midhaul User Plane tunnel communication between the DU and CU. The packet size refers to the size of the ethernet frame transmitted on the end-to-end 5G network, and the throughput values are computed on the midhaul communication by varying the packet size. The throughput result shows that ANS-DPDK outperforms Kernel, achieving, on average, up to about 182.236% throughput improvement. Fig. 8 reports the RTT results captured in the midhaul communication and the result clearly shows that ANS-DPDK RTT reduction is 95.963% when compared to that of Kernel based posix call accounting for faster user data communication.

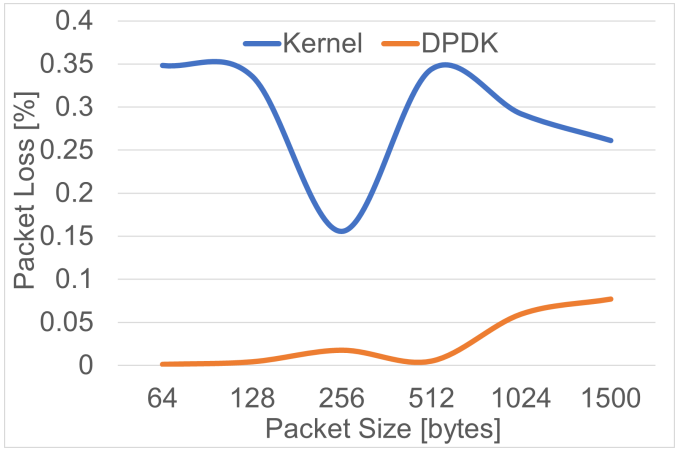

Figure 9: 5G Midhaul Packet Loss Comparison between POSIX Socket and ANS-DPDK

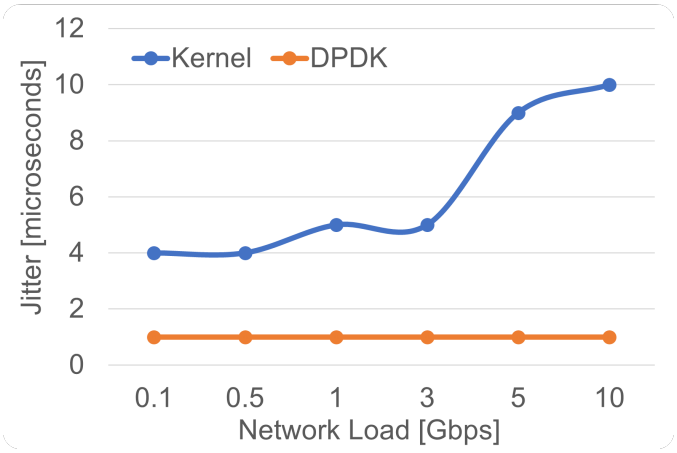

Figure 10: 5G Midhaul Jitter Comparison between POSIX Socket and ANS-DPDK

In addition, parameters such as packet loss and jitter are measured to check the system's performance. All the experiments are repeated for ten trials, and the average result is considered. Packet loss occurs when the network packet fails to reach the destination, causing information data loss. One of the several causes of packet loss would be network congestion, where the network is operating at higher capacity, and it was simulated in this experiment. Jitter measures the inconsistency of the data packet arrival rate. Higher Jitter would cause the packets to be received out of order or discarded, resulting in packet loss.

Figs. 9 and 10 show the packet loss and jitter calculated in the midhaul interface for varying network load. The maximum transmission unit (MTU) is set to 1500 bytes in both the Kernel and ANS-DPDK mode. MTU is the largest packet size, specified in octets that can be sent in a packet-based network. As shown in Figs. 9 and 10, the network load increases, the jitter value increases for the Kernel mode. The jitter value is comparatively lower in the DPDK mode and stays in the same range for the change in the network load, thereby reducing the packet loss. During the experiment, it is noted that the packet loss is not linear with the UDP tunnel communication, which is unreliable, and therefore, the drops are unpredictable. Similar patterns of UDP packet loss in the kernel are observed in [19]. However, in all our trials, it is noted that the packet loss percentage is lower in ANS-DPDK mode than in the Kernel mode.

To analyze the CPU utilization behavior, the Linux top command was run in batch mode. The top command will show the real-time running processes and their corresponding threads to view the system resource utilization. Only the threads involved in the OAI CU communication and those that produce more than one percent of CPU utilization are listed in Table 4. The default Linux scheduling is performed without any CPU affinity for the OAI CU application. The ANS application is started with the CPU Core 3. It is observed that the OAI CU application uses three CPU cores to schedule the work in the kernel mode. In the DPDK mode, the ANS utilizes one CPU core, so overall, four CPU cores are used for running the application.

Table 4 clearly shows that the CPU utilization is lower for the Task_GTP_v1_U, ru_thread and nrsoftmodem threads in the ANS-DPDK mode. It is because of the kernel bypass support provided by the DPDK which allows the thread to perform other intertask_interface operations. On the contrary, the ans_io thread utilizes the cpu — lcore 3 —- effectively upto 99.7% in the ANS-DPDK mode. The ANS-DPDK IO framework uses the CPU affinity to assign threads to the lcore 3 for the considered receive port. It uses the poll mode driver to configure the device and to check for any messages received, constituting higher CPU utilization. Also, for all the threads, the memory utilization is higher in ANS-DPDK mode because, in the user space, ANS-DPDK uses the memory pool from the Non-Uniform-Memory-Access (NUMA) for its I/O operations.

6.4.1. End-User Performance Analysis — End-to-End Mode

In this setup, DPDK is integrated for both the midhaul interface connecting gNB-vCU and gNB-vDU (the GTP F1-U tunnel) and the backhaul interface connecting gNB-vCU and UPF (the GTP NG-U tunnel) on a 5G standalone testbed. Four interfaces are created in the configuration file on the CU side, accounting for the Control Plane communication with the DU, User Plane communication with the DU, Control Plane communication with AMF, and User Plane communication with UPF of the CN. In this End-to-End mode, DU Control Plane and AMF communication calls are initiated using the kernel API calls same as in the hybrid mode. In contrast, DU User Plane and UPF communication calls are created using ANS DPDK API calls in the DPDK-enabled setup and kernel API calls in the kernel-enabled one. Due to the hardware and ANS limitations, a switch with the Gigabit ethernet port is configured for the same CU interface to communicate with the DU and the UPF of the user data traffic. This switch configuration setup generates the midhaul and backhaul traffic

Table 4: CPU and Memory Utilization for the OAI CU application

OAI CU	Posix API		ANS-DPDK API	
[Thread Name]	% CPU	% Memory	% CPU	% Memory
Task_GTP_V1_U	95.3	1.8	94.7	1.9
ru_thread	79	1.8	68.1	1.9
nr-softmodem	15.7	1.8	9.6	1.9
Tpool0_-1	15.3	1.8	15.3	1.9
Tpool1_-1	15	1.8	15	1.9
Tpool2_-1	2.3	1.8	1.7	1.9
Tpool3_-1	2.3	1.8	1.7	1.9
Tpool4_-1	2.3	1.8	1.7	1.9
Tpool5_-1	2.3	1.8	1.7	1.9
Tpool6_-1	2.3	1.8	1.7	1.9
Tpool7_-1	2.3	1.8	1.7	1.9
pdcp_timer	0.7	1.8	0.7	1.9
ans_io	-	-	99.7	-

Table 5: End-to-End Iperf Comparison between Kernel and DPDK

Iperf direction	Bandwidth (Mbps)	
	Kernel	DPDK
Uplink	13.37	13.67
Downlink	9.98	10.01

Table 6: End-to-End Throughput Comparison for Packet Sizes in Kernel and DPDK

Packet Size (Bytes)	Throughput	
	Kernel (Mbps)	DPDK (Mbps)
64	6.49	6.64
128	8.06	8.09
256	9.07	9.1
512	9.93	9.98
1024	9.98	10.01

at 1 Gbps speed, even though the NIC capacity is 10 Gbps at the midhaul network. For the End-to-End performance evaluation, the results are captured by initiating the user plane tunnel communication both in the kernel mode and in the DPDK mode.

Fig. 11 shows the RTT results captured end-to-end and the result indicates that there is RTT reduction in the ANS-DPDK based setup than that of Kernel-based one, accounting for a comparatively faster user data communication. Table. 5 reports the iperf results captured in both the uplink and downlink direction from UE to CN and vice versa once the end-to-end connectivity is established. The table shows that ANS-DPDK setup shows a marginal improvement over the Kernel one. Only a limited bandwidth improvement is noticed because of the radio channel bandwidth constraint and configured switch speed. Increasing the radio channel bandwidth and the switch speed will correspondingly increase the bandwidth. DPDK performance improvements can more visible when the incoming packets reach the NIC threshold.

Table. 6 reports the throughput results captured end-to-end from the UE to the GTP tunnel interface of the CN (i.e the UPF). The packet size refers to the size of the ethernet frame transmitted end-to-end on the 5G network, and the throughput values are computed by varying the packet size. The throughput result shows that the ANS-DPDK enabled setup performs better than the Kernel one. The results will become similar to the midhaul throughput result as shown in Fig. 7 once the radio channel bandwidth increases and the incoming data packets reach the NIC threshold. This increase in data flow makes the Kernel mode reach its bottleneck, whereas the DPDK mode handles the traffic efficiently, making the DPDK mode ideal for the 5G use cases. The bottleneck scenario is interpreted in Fig. 7 for the GTP midhaul communication.

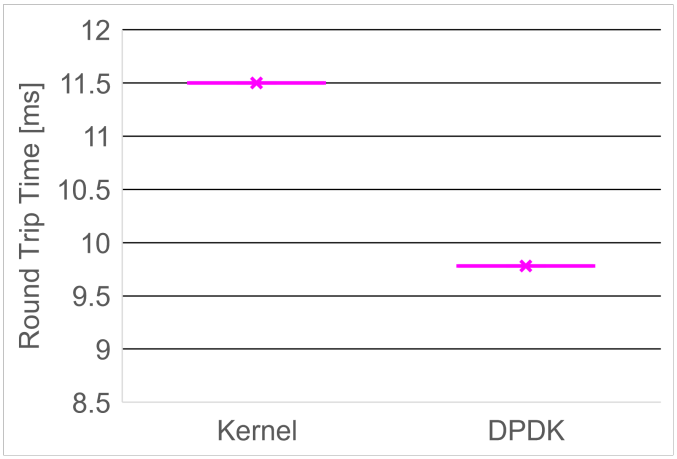

Figure 11: 5G End-User RTT Comparison between POSIX Socket and ANS-DPDK

6.4.2. End-User Application Performance Analysis

Fig. 12 reports the content bitrate results captured in two different testbeds namely the ANS-DPDK and the Kernel-enabled one. The results highlight that there is an increase in the content bitrate in the case of ANS-DPDK enabled setup compared to the traditional Kernel-based one. The high bitrate for the video application running in DPDK enabled testbed can be attributed to the Kernel bypass technique eliminating the underlying overheads and challenges like a significant amount of context switching, which occurs in the traditional Kernel packet processing. It should be noted that by increasing the radio channel bandwidth with the high speed application, the efficiency of the DPDK enabled mode can further increase and become more apparent than the Kernel-based mode.

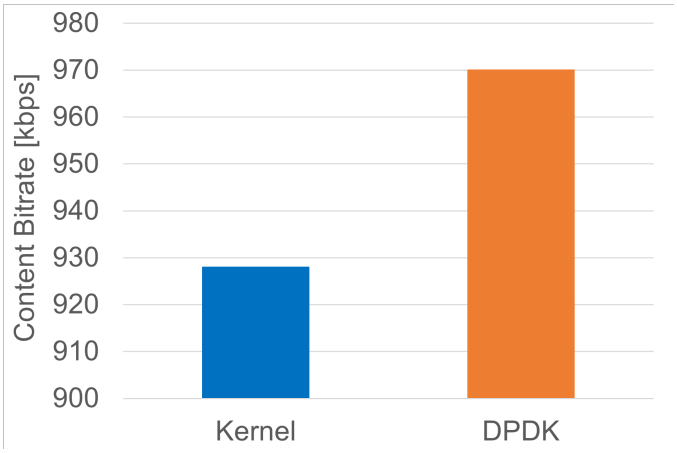

Figure 12: 5G End-User Video Streaming Content Bit Rate

7. Conclusion and Future Work

In this paper, we presented a solution to enable end-to-end open-source 5G deployment with the DPDK feature incorporated on the commercial off-the-shelf (COTS) hardware suitable for various vertical use cases like Industry 4.0, Smart Cities, Healthcare, etc. Different network performance metrics are discussed, while abstracting the underlying hardware complexity. Our solutions are verified both in the midhaul and backhaul GTP tunnel traffic for the user data plane with the considered two design models. Our obtained results show the improvement in crucial network parameters such as latency, throughput, packet loss and jitter over the ANS-DPDK framework on the OpenAirInterface software components.

The possible subject of future work involves the study of the data plane development kit involving low latency applications that facilitate the development of high-speed data plane use cases for the 5G user plane. Our experimentation reveals that DPDK-related performance improvements are particularly noted on heavily loaded network applications. We focus on increasing the midhaul and backhaul user data traffic with the increased number of UE connections and improving the radio channel bandwidth. In addition, the resiliency feature for the DPDK-supported containerized radio access network components also comes under the scope of our study.

APPENDIX: List of abbreviations

AMF Access and Mobility Management Function in 5G Core Netork

AUSF Authentication Server Function in 5G Core Netork

CU Central Unit in the Next Generation base station

DPDK Data Plane Development Kit, a set of libraries used for implementing UserSpace drivers for the Network Interface Controllers

DU Distributed Unit in the Next Generation base station

gNB-vCU virtualized Central Unit in the Next Generation base station

gNB-vDU virtualized Distributed Unit in the Next Generation base station

GTP GPRS Tunneling Protocol that carries the general packet radio service

MAC Medium Access Control in LTE/5G protocol stack

NRF Network Repository Function in 5G Core Network

OAI OpenAirInterface, an open-source wireless technology platform for the LTE/5G system

PDCP Packet Data Convergence Protocol in LTE/5G protocol stack

PDU Protocol Data Unit, a single unit of information transmitted among peer entities of a computer network

PHY Physical Layer in LTE/5G protocol stack

RLC Radio Link Control in LTE/5G protocol stack

SMF Session Management Function in 5G Core Netork

TEID Tunnel Endpoint Identifier transferred among endpoints for communication amongst them

UDM Unified Data Management in 5G Core Network

UDR Unified Data Repository in 5G Core Network

UPF User Plane Function in 5G Core Network

USRP Universal Software Radio Peripheral, a software-defined radio device

VLC VideoLAN Client, an open-source free multimedia player

References

[1] Intel, Data Plane Development Kit, https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html.

[2] L. Lai, G. Ara, T. Cucinotta, K. Kondepu, L. Valcarenghi, Ultra-low Latency NFV Services Using DPDK, in: IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2021, pp. 8–14. doi:10.1109/NFV-SDN53031.2021.9665131.

[3] S. R. Chowdhury, Anthony, H. Bian, T. Bai, R. Boutaba, A Disaggregated Packet Processing Architecture for Network Function Virtualization, IEEE Journal on Selected Areas in Communications 38 (6) (2020) 1075–1088. doi:10.1109/JSAC.2020.2986611.

[4] P. Emmerich, D. Raumer, F. Wohlfart, G. Carle, A study of network stack latency for game servers, in: 13th Annual Workshop on Network and Systems Support for Games, 2014, pp. 1–6. doi:10.1109/NetGames.2014.7008960.
[5] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, F. Liberal, Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration, in: IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), 2015, pp. 74–78. doi:10.1109/NFV-SDN.2015.7387409.
[6] D. M. Brennan, V. Marojevic, UHD-DPDK Performance Analysis for Advanced Software Radio Communications, in: 18th International Conference on Distributed Computing in Sensor Systems (DCOSS), 2022, pp. 420–425. doi:10.1109/DCOSS54816.2022.00076.
[7] mTCP, https://github.com/mtcp-stack/mtcp.
[8] F-Stack, https://github.com/F-Stack/f-stack.
[9] FD.io, https://fd.io.
[10] SeaStar, https://github.com/scylladb/seastar.
[11] PAugenblick, https://github.com/vadimsu/ipaugenblick.
[12] Accelerated-Network-Stack, https://github.com/ansyun/dpdk-ans.
[13] Epoll, https://man7.org/linux/man-pages/man7/epoll.7.html.
[14] Daytong, The Implementation of epoll, https://idndx.com/the-implementation-of-epoll-1/.
[15] CMake, Mastering CMake, https://cmake.org/cmake/help/book/mastering-cmake/chapter/Why%20CMake.html.
[16] Make, Learn MakeFiles, https://makefiletutorial.com/.
[17] A. Bhattacharyya, S. Ramanathan, A. Fumagalli, K. Kondepu, Towards Disaggregated Resilient 5G Radio Access Network: A Proof of Concept, in: IEEE 9th International Conference on Network Softwarization (NetSoft), 2023, pp. 396–401. doi:10.1109/NetSoft57336.2023.10175418.
[18] 3GPP TR 38.801 , Study on new radio access technology: Radio access architecture and interfaces, Release 14 (Mar. 2017).
[19] K. M. S. Soyjaudah, P. C. Catherine, I. Coonjah, Evaluation of UDP tunnel for data replication in data centers and cloud environment, in: International Conference on Computing, Communication and Automation (ICCCA), 2016, pp. 1217–1221. doi:10.1109/CCAA.2016.7813927.