cttc-nr-demo tutorial

Release 2.6

Giovanni Grieco and OpenSim CTTC/CERCA

Sep 11, 2023

CONTENTS

CHAPTER

ONE

ACRONYMS

- 5th Generation (5G)
- Acknowledged Mode (AM)
- Acknowledgement (ACK)
- Bandwidth Part (BWP)
- Bearer Identifier (BID)
- Carrier Component (CC)
- Data/Control (D/C)
- Data Radio Bearer Identifier (DRBID)
- Downlink (DL)
- Downlink Control Indicator (DCI)
- Evolved Node-B (eNB)
- Evolved Packet Core (EPC)
- External Interface – User Plane (X2-U)
- Finite State Machine (FSM)
- General Packet Radio Service (GPRS)
- GPRS Tunneling Protocol – User data tunneling (GTP-U)
- Guaranteed Bit Rate (GBR)
- Head-of-Line (HOL)
- Hybrid Automatic Repeat Request (HARQ)
- Identifier (ID)
- International Mobile Subscriber Identity (IMSI)
- Internet Protocol version 4 (IPv4)
- Line of Sight (LoS)
- Listen Before Talk (LBT)
- Logical Channel Identifier (LCID)
- Long Term Evolution (LTE)
- Maximum Transmission Unit (MTU)

- Medium Access Control (MAC)
- New Radio (NR)
- Network Simulator 3 (ns-3)
- Next-Generation Node-B (gNB)
- Non-Access Stratum (NAS)
- Non-Guaranteed Bit Rate (NGBR)
- Non-Standalone (NSA)
- Packet Data Convergence Protocol (PDCP)
- Packet Gateway (PGW)
- Power Spectrum Density (PSD)
- Protocol Data Unit (PDU)
- Quality of Service Class Indicator (QCI)
- Radio Access Network (RAN)
- Radio Link Control (RLC)
- Radio Network Temporary Identity (RNTI)
- Radio Resource Control (RRC)
- Random Access Control Channel (RACH)
- Receive (RX)
- Round Robin (RR)
- Service Access Point (SAP)
- Service Data Unit (SDU)
- Serving Gateway (SGW)
- System Frame Number / Subframe Number (SFN/SF)
- Transmit (TX)
- Transmission Time Interval (TTI)
- Transparent Mode (TM)
- Transport Block (TB)
- Tunnel Endpoint Identifier (TEID)
- Unacknowledged Mode (UM)
- Uplink (UL)
- User Datagram Protocol (UDP)
- User Equipment (UE)

CHAPTER

TWO

OVERVIEW

This is a tutorial focused on the data plane operation of the example program cttc-nr-demo found in the `examples/` directory of the *ns-3* `nr` module. The objective of the tutorial is to provide a detailed, layer-by-layer walk through of a basic NR example, with a focus on the typical lifecycle of packets as they traverse the RAN. The tutorial points out all of the locations in the RAN model where packets may be delayed or dropped, and how to trace such events.

This document assumes that you have already installed *ns-3* with the `nr` module and you are familiar with how *ns-3* works. If this is not the case, please review the *ns-3* Installation Guide and Tutorial as needed. The Getting Started page of the `nr` module should also be reviewed.

The companion to this tutorial is the detailed manual for the `nr` module, which goes into more detail about the design and testing of each of the components of the 5G NR module.

To check if you are ready to work through this tutorial, check first if you can run the following program:

```
$ ./ns3 run cttc-nr-demo
```

and that it outputs the following output

```
Flow 1 (1.0.0.2:49153 -> 7.0.0.2:1234) proto UDP
    Tx Packets: 6000
    Tx Bytes:   768000
    TxOffered:  10.240000 Mbps
    Rx Bytes:   767744
    Throughput: 10.236587 Mbps
    Mean delay:  0.271518 ms
    Mean jitter:  0.030006 ms
    Rx Packets: 5998
Flow 2 (1.0.0.2:49154 -> 7.0.0.3:1235) proto UDP
    Tx Packets: 6000
    Tx Bytes:   7680000
    TxOffered:  102.400000 Mbps
    Rx Bytes:   7671040
    Throughput: 102.280533 Mbps
    Mean delay:  0.835065 ms
    Mean jitter:  0.119991 ms
    Rx Packets: 5993

Mean flow throughput: 56.258560
Mean flow delay: 0.553292
```

The tutorial also makes extensive use of the *ns-3* logging framework. To check if logs are enabled in your *ns-3* libraries, try the following command and check if it outputs some additional verbose output:

```
$ NS_LOG="CttcNrDemo" ./ns3 run cttc-nr-demo
```

2.1 Program Overview

From what it can be deduced, the demo simulates two downlink *Flows*, each of them relying on a unicast and unidirectional communication. Such flows rely on the UDP to carry application data from an origin with IPv4 `1.0.0.2` to two recipients with IPv4 `7.0.0.2` and `7.0.0.3` for *Flow 1* and *Flow 2*, respectively.

The purpose of this example is to simulate a downlink scenario. Two data flows originate from a remote host, with specific characteristics. One flow emphasizes low-latency communications, while the other focuses on achieving a higher throughput. In the provided demo output, it is evident that the former exhibits significantly lower mean delay and jitter compared to the latter, whereas the opposite is true for the achieved throughput. In the code, the low-latency communication is referred to as `LowLat` to indicate its low-latency nature, while the one that achieves higher throughput is referred to as `Voice` to reflect the traditional traffic associated with high-quality voice communications.

For this communication, the source is an IPv4 address, specifically `1.0.0.2`, which is referred to as the "remoteHost." The recipients of the data are two UEs.

To support such communications, a 5G Radio Access Network is configured, together with an LTE Core Network, referred to as the EPC. The entire architecture is defined as 5G NSA.

This demo is characterized by quasi-ideal conditions. For instance, the S1-U link, which interconnects the SGW with the gNB, has no delay. Furthermore, Direct Path Beamforming is used, which is an ideal algorithm based on the preassumption that the transmitters always know the exact location of the receivers. This particular beamforming technique is implemented in the spectrum module, part of the core of ns-3. Further information can be retrieved on the manual, at Section 2.3.9. Shadowing is not considered, as buildings and any other kind of obstacles that could impair normal LoS conditions are absent. Finally, the channel model is updated only once, at the start of the simulation, given that the scenario is static, i.e., it does not change over time.

While both UEs are characterized by a Uniform Planar Array of 2x4 isotropic antenna elements, the gNB has the same array with a configuration of 4x8.

In terms of spectrum, 2 bands are created to support such communications. The first one operates at 28 GHz, while the second one at 28.2 GHz, both with a bandwidth of 100 MHz. In terms of numerology, i.e., the sub-carrier spacing, the former is 4, while the latter is 2. This simplifies spectrum allocation, given that each communication will operate on a dedicated BWP, on a single CC that occupies the entire band, resulting in the spectrum organized as below:

```
* The configured spectrum division is:
* ------------Band1--------------|--------------Band2-----------------
* ------------CC1----------------|--------------CC2-------------------
* ------------BWP1---------------|--------------BWP2------------------
```

Given that there is only one gNB, a total transmission power of 4 dbW is spread among the two BWPs.

In terms of the BWP type and bearer, the former communication is configured to use a QCI with NGBR Low Latency, also known in the code as `NGBR_LOW_LAT_EMBB`, while the latter has a QCI with GBR and is named as `GBR_CONV_VOICE`. A list of other QCI types can be found at the ns-3 doxygen page on QCI.

On the top of the stack, two UDP applications are configued. The low-latency voice traffic is simulated to send 100 bytes, whereas the high data rate one sends 1252 bytes. Both of these applications send data every 0.1 ms. The `FlowMonitorHelper` is used to gather data statistics about the traffic.

Finally, the EPC's PGW is then connected to a remote host with an ideal Point-to-Point channel: 100 Gbps of data rate with 2500 bytes of MTU and no delay.

2.2 References

[cttc-nr-demo] cttc-nr-demo program. Available at: https://gitlab.com/cttc-lena/nr/-/blob/master/examples/cttc-nr-demo.cc

CHAPTER

THREE

END-TO-END OBSERVATIONS

We are mainly interested in observing the packet lifecycle as it moves through the RAN stack. We can make a few initial observations about the packet flow. The code is using `UdpClient` and `UdpServer` objects at the application layer. One client sends a stream of 1252 byte packets to the server, and the other client sends a stream of 100 byte packets to the server. This configuration can be seen in these lines of code:

```
UdpClientHelper dlClientLowLat;
...
dlClientLowLat.SetAttribute("PacketSize", UintegerValue(udpPacketSizeULL));
...
UdpClientHelper dlClientVoice;
...
dlClientVoice.SetAttribute("PacketSize", UintegerValue(udpPacketSizeBe));
```

Using the following logging-enabled command, generate the log information from the UdpClient and UdpServer objects in the simulation, and redirect the output to two files, as follows:

```
$ NS_LOG="UdpClient=info|prefix_time|prefix_node|prefix_func" ./ns3 run 'cttc-nr-demo
↪' > log.client.out 2>&1
$ NS_LOG="UdpServer=info|prefix_time|prefix_node|prefix_func" ./ns3 run 'cttc-nr-demo
↪' > log.server.out 2>&1
```

Looking at the first couple of lines of the `log.client.out` file, one can see:

```
+0.400000000s 6 UdpClient:Send(): TraceDelay TX 100 bytes to 7.0.0.2 Uid: 8 Time: +0.
↪4s
+0.400000000s 6 UdpClient:Send(): TraceDelay TX 1252 bytes to 7.0.0.3 Uid: 9 Time: +0.
↪4s
```

These first two packets were sent at the same time to two different UEs, from node 6. Next, observe the first packet arrivals on the UEs via the `log.server.out` file:

```
+0.400408031s 1 UdpServer:HandleRead(): TraceDelay: RX 100 bytes from 1.0.0.2␣
↪Sequence Number: 0 Uid: 8 TXtime: +4e+08ns RXtime: +4.00408e+08ns Delay: +408031ns
...
+0.401832140s 2 UdpServer:HandleRead(): TraceDelay: RX 1252 bytes from 1.0.0.2␣
↪Sequence Number: 0 Uid: 9 TXtime: +4e+08ns RXtime: +4.01832e+08ns Delay: +1.
↪83214e+06ns
```

The reception times (and packet delays) are quite different. One takes only 408 us to be delivered, the other takes 1832 us to be delivered. In this tutorial, we will explain why this is so.

This also illustrates that one does not have to let the simulation run for longer than 0.402 seconds if the focus is on these two packets. One can use a command-line argument to shrink the total simulation time, such as:

```
./ns3 run 'cttc-nr-demo --simTime=1s'
```

or one can edit the C++ program directly to change the default simTime value:

```
Time simTime = MilliSeconds(1000);
```

While working through this tutorial, we recommend the latter (temporarily editing the C++ program), to shorten the log files. The below example statements omit the `--simTime` argument because it is changed in the C++ program.

CHAPTER

FOUR

RAN LIFECYCLE

The figure *Packet lifecycle over the RAN from a data plane standpoint.* depicts the objects that each packet will traverse through the RAN. This tutorial will walk through each step of the way, starting with the entry point for these packets– the `EpcEnbApplication`.

4.1 EpcEnbApplication

The `EpcEnbApplication` is installed on the gNB. It is responsible for receiving packets tunneled through the EPC model and sending them into the `NrGnbNetDevice`. Conceptually, this is just an application-level relay function. It is possible to run the `cttc-nr-demo` scenario by enabling the `EpcEnbApplication` log component, which can also be configured to print messages at INFO level and prefix the message by stating the time of the simulation, the node ID of interest, and the routine that prints that message. In this way, with the following command

```
$ NS_LOG="EpcEnbApplication=info|prefix_all" ./ns3 run 'cttc-nr-demo' > log.out 2>&1
```

one can observe this relay function on the first packet, as follows:

```
+0.400000282s 0 EpcEnbApplication:RecvFromS1uSocket(): [INFO ] Received packet from␣
↪S1-U interface.
+0.400000282s 0 EpcEnbApplication:SendToLteSocket(): [INFO ] Forward packet from eNB
↪'s S1-U to LTE stack.
```

The file `src/lte/model/epc-enb-application.cc` contains the source code.

It is possible to get more information about how packet is being handled by enabling the DEBUG log level, which just requires a slight modification of the previous command:

```
$ NS_LOG="EpcEnbApplication=info|debug|prefix_all" ./ns3 run 'cttc-nr-demo' > log.out␣
↪2>&1
```

In this way, we ask the log component to print both INFO and DEBUG log messages. This produces the following output:

```
+0.400000282s 0 EpcEnbApplication:RecvFromS1uSocket(): [INFO ] Received packet from␣
↪S1-U interface.
+0.400000282s 0 EpcEnbApplication:RecvFromS1uSocket(): [DEBUG] Packet inspection:
  Packet tags:
    false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [32-140] FlowId=1 PacketId=0 PacketSize=128
  Packet structure:
    ns3::GtpuHeader ( version=1 [ PT  S  PN ], messageType=255, length=132, teid=2,␣
↪sequenceNumber=0, nPduNumber=0, nextExtensionType=0) ns3::Ipv4Header (tos 0
```

(continues on next page)

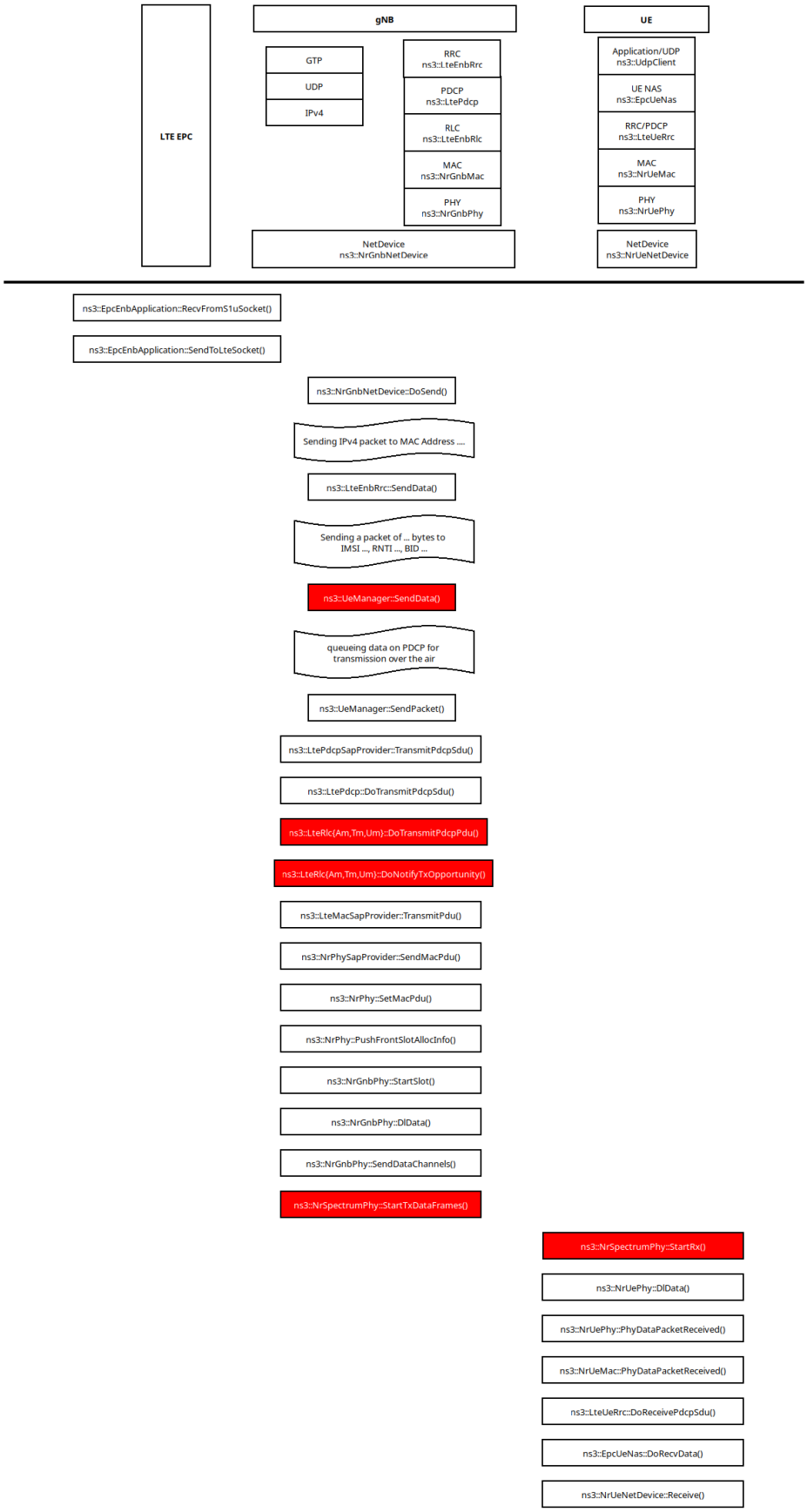

Fig. 1: Packet lifecycle over the RAN from a data plane standpoint.

(continued from previous page)

```
x0 DSCP Default ECN Not-ECT ttl 63 id 0 protocol 17 offset (bytes) 0 flags [none]␣
↪length: 128 1.0.0.2 > 7.0.0.2) ns3::UdpHeader (length: 108 49153 > 1234)␣
↪ns3::SeqTsHeader ((seq=0 time=+0.4s)) Payload (size=88)
+0.400000282s 0 EpcEnbApplication:SendToLteSocket(): [INFO ] Forward packet from eNB
↪'s S1-U to LTE stack.
+0.400000282s 0 EpcEnbApplication:SendToLteSocket(): [DEBUG] Packet inspection:
  Packet tags:
    rnti=2, bid=2 false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [20-128] FlowId=1 PacketId=0 PacketSize=128
  Packet structure:
    ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT ttl 63 id 0 protocol 17 offset␣
↪(bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.2) ns3::UdpHeader (length: 108␣
↪49153 > 1234) ns3::SeqTsHeader ((seq=0 time=+0.4s)) Payload (size=88)
```

Notice that now, between INFO messages, there are also DEBUG ones with their message starting with "Packet inspection". They are useful to understand how a packet is modified by a given object, which represents a layer of the gNB stack. The packet inspection does not only prints the structure of a packet per se, but also its metadata, which are tags and byte tags. Remember that tags are bound to the entire packet, while byte tags only to a portion of it and can survive fragmentation. More information about how packets and tags work can be found in the ns-3 manual.

It is now possible to better understand what this application does to the incoming packet by looking at the differences with the outgoing one.

First of all, take a look at the packet tags. The `EpcEnbApplication` adds the cell-specific UE ID (RNTI) and BID as tags. This greatly simplifies packet processing in later sections. The application is able to find the right RNTI and BID given the TEID present on the GTP-U header of the received packet structure. This process can be found at `EpcEnbApplication::RecvFromS1uSocket()` source code:

```
GtpuHeader gtpu;
packet->RemoveHeader(gtpu);
uint32_t teid = gtpu.GetTeid();
std::map<uint32_t, EpsFlowId_t>::iterator it = m_teidRbidMap.find(teid);
if (it == m_teidRbidMap.end())
{
    NS_LOG_WARN("UE context at cell id " << m_cellId << " not found, discarding packet
↪");
    m_rxDropS1uSocketPktTrace(packet->Copy());
}
else
{
    m_rxS1uSocketPktTrace(packet->Copy());
    SendToLteSocket(packet, it->second.m_rnti, it->second.m_bid);
}
```

Notice that the hashmap `<uint32_t, EpsFlowId_t>` links together a TEID, handled by `uint32_t`, with RNTI and BID, grouped by EpsFlowId_t data structure.

Secondly, there is a byte tag to trace the packet for flow statistics, which are bound to Flow Monitor and are relevant to the final results that are printed by the scenario. This byte tag is linked to the packet byte range `[32-140]`. It is possible to see that such range changes at the second inspection, as the packet structure is modified, i.e., GTP-U header is removed.

The packet and its modifications can be visually represented in this way, where removed portions of the packet are marked in red, whereas green ones are the added portions:

The new modified packet is finally sent to `EpcEnbApplication::SendToLteSocket()`:

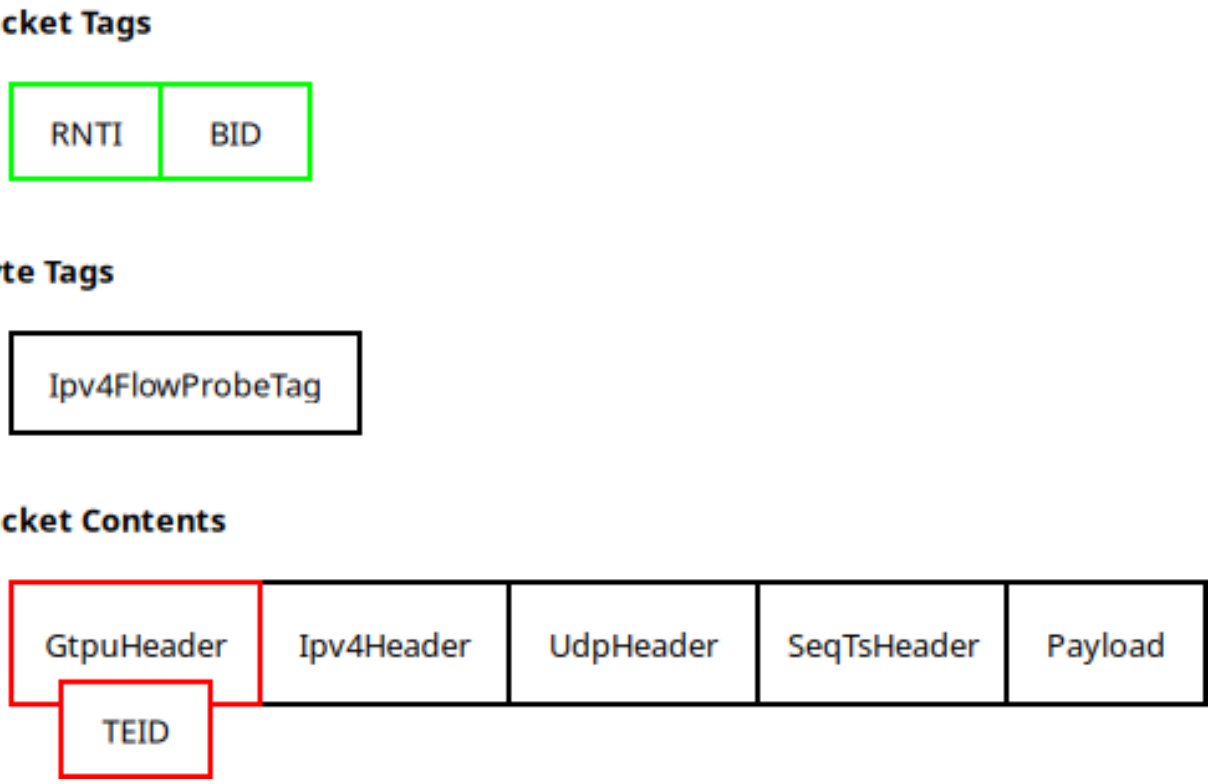

Fig. 2: Structure of the packet handled by `EpcEnbApplication`.

```
if (ipType == 0x04)
{
    sentBytes = m_lteSocket->Send(packet);
}
else if (ipType == 0x06)
{
    sentBytes = m_lteSocket6->Send(packet);
}
```

The LTE sockets, both IPv4 and IPv6 versions, are actually created in the EPC helper, specifically, in the file `src/lte/helper/no-backhaul-epc-helper.cc`. The socket is created and bound to a NetDevice; in this case, even though the variable name is `lteEnbNetDevice`, it will be of type `NrGnbNetDevice`:

```
// create LTE socket for the ENB
Ptr<Socket> enbLteSocket =
    Socket::CreateSocket(enb, TypeId::LookupByName("ns3::PacketSocketFactory"));
PacketSocketAddress enbLteSocketBindAddress;
enbLteSocketBindAddress.SetSingleDevice(lteEnbNetDevice->GetIfIndex());
...
```

As a final remark, the following traces can be used to track incoming and outgoing packets from this application:

- `RxFromEnb`: Receive data packets from LTE Enb Net Device
- `RxFromS1u`: Receive data packets from S1-U Net Device
- `RxFromS1uDrop`: Drop data packets from S1-U Net Device
- `TxToEnd`: Transmit data packets to LTE eNB Net Device

- `TxToS1u`: Transmit data packets to S1-U Net Device

4.1.1 Packet latency

Packets cannot incur latency in the `EnbEpcApplication`.

4.1.2 Packet drops

Packets can be dropped if there is no association between GTP-U TEID and gNB-UE link's RNTI and BID.

4.2 NrGnbNetDevice

After the `EpcEnbApplication` sends a packet, it is immediately received on the `NrGnbNetDevice::DoSend()` method. We can observe this in the logs:

```
$ NS_LOG="NrGnbNetDevice=info|prefix_all" ./ns3 run 'cttc-nr-demo' > log.out 2>&1
```

```
+0.400000282s 0 NrGnbNetDevice:DoSend(): [INFO ] Forward received packet to RRC Layer
```

The source code for this method is in the file `contrib/nr/model/nr-gnb-net-device.cc`. As an aside, notice how we are bouncing back and forth between the source code directories `src/lte/` (for the previous `EpcEnbApplication`) and `contrib/nr/` (for this object); this is true for the upper layers of the current `nr` module, which reuse pieces originally implemented for LTE.

The source code reveals that it is possible to trace the packet via `NrNetDevice/Tx`. Moreover, the packet is forwarded down to the RRC, if some sanity checks are passed:

```
NS_ABORT_MSG_IF(protocolNumber != Ipv4L3Protocol::PROT_NUMBER &&
                protocolNumber != Ipv6L3Protocol::PROT_NUMBER,
                "unsupported protocol " << protocolNumber
                                        << ", only IPv4 and IPv6 are supported");

NS_LOG_INFO("Forward received packet to RRC Layer");
m_txTrace(packet, dest);

return m_rrc->SendData(packet);
```

`NrGnbNetDevice` does not process the packet, but it just relays it to the RRC layer. To this end, packet analysis is skipped.

4.2.1 Packet latency

Packets cannot incur latency in the `NrGnbNetDevice`.

4.2.2 Packet drops

Packets cannot be dropped in the `NrGnbNetDevice`.

4.3 LteEnbRrc

The RRC layer of the gNB is handled by `LteEnbRrc` and its companion `UeManager`, both available in `lte/model/lte-enb-rrc.cc`. While packets incoming from the upper layers are processed by `LteEnbRrc::SendData()`, packets to be sent to UEs are handled by `UeManager::SendPacket()`. Indeed, if the simulation is started with the following command:

```
$ NS_LOG="LteEnbRrc=info|prefix_all" ./ns3 run cttc-nr-demo -- --simTime=1s &> out.log
```

it produces the following log messages:

```
+0.400000282s 0 LteEnbRrc:SendData(): [INFO ] Received packet
+0.400000282s 0 LteEnbRrc:SendPacket(): [INFO ] Send packet to PDCP layer
```

By taking a look at the source code, the RRC layer starts by extracting the RNTI, which is handled by the `EpsBearerTag` data structure. Thanks to the RNTI, it is possible to retrieve the corresponding `UeManager` instance, which keeps track of the gNB-UE link state through a FSM. The search is done on the `m_ueMap` property of `LteEnbRrc`, which is a hashmap that links each RNTI to a pointer of the `UeManager`. Once found, the matching `SendData()` method is called with the packet's BID of reference.

```
EpsBearerTag tag;
bool found = packet->RemovePacketTag(tag);
NS_ASSERT_MSG(found, "no EpsBearerTag found in packet to be sent");
Ptr<UeManager> ueManager = GetUeManager(tag.GetRnti());

ueManager->SendData(tag.GetBid(), packet);
```

The `SendData()` method can be quickly understood with the following code excerpt:

```
switch (m_state)
{
case INITIAL_RANDOM_ACCESS:
case CONNECTION_SETUP:
    NS_LOG_WARN("not connected, discarding packet");
    m_packetDropTrace(p, bid);
    return;

case CONNECTED_NORMALLY:
case CONNECTION_RECONFIGURATION:
case CONNECTION_REESTABLISHMENT:
case HANDOVER_PREPARATION:
case HANDOVER_PATH_SWITCH: {
    NS_LOG_LOGIC("queueing data on PDCP for transmission over the air");
    SendPacket(bid, p);
}
break;
// ...
}
```

Let's ignore the states related to handover, given that in the NR module the X2-U interface is not implemented, yet.

If the UE is ready to receive the packet from the gNB viewpoint, i.e. it's in `CONNECTED_NORMALLY`, the `SendPacket()` method is called to continue with the transmission:

```
LtePdcpSapProvider::TransmitPdcpSduParameters params;
params.pdcpSdu = p;
params.rnti = m_rnti;
params.lcid = Bid2Lcid(bid);
uint8_t drbid = Bid2Drbid(bid);
// Transmit PDCP sdu only if DRB ID found in drbMap
std::map<uint8_t, Ptr<LteDataRadioBearerInfo>>::iterator it = m_drbMap.find(drbid);
if (it != m_drbMap.end())
{
    Ptr<LteDataRadioBearerInfo> bearerInfo = GetDataRadioBearerInfo(drbid);
    if (bearerInfo)
    {
        NS_LOG_INFO("Send packet to PDCP layer");
        NS_LOG_DEBUG("Packet inspection:" << p->Inspect());
        LtePdcpSapProvider* pdcpSapProvider = bearerInfo->m_pdcp->
↪GetLtePdcpSapProvider();
        pdcpSapProvider->TransmitPdcpSdu(params);
    }
}
```

Over there, the packet is processed by creating a PDCP SDU structure with complementary information, known as `LtePdcpSapProvider::TransmitPdcpSduParameters`. These parameters are the RNTI and other two parameters which are dependent of the BID, such as LCID, which in this case is 4, and DRBID, which is 2. The latter is used to retrieve the `LteDataRadioBearerInfo` associated to a DRBID, if it is still in place. At last, the SDU is forwarded to the the active `LtePdcpSapProvider` through `TransmitPdcpSdu()`.

With this procedure in mind, it is possible to understand how the packet is modified. If we run the same simulation by putting `LteEnbRrc` log component in both DEBUG and INFO levels, we obtain the following message logs:

```
+0.400000282s 0 LteEnbRrc:SendData(): [INFO ] Received packet
+0.400000282s 0 LteEnbRrc:SendData(): [DEBUG] Packet inspection:
  Packet tags:
    rnti=2, bid=2 false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [20-128] FlowId=1 PacketId=0 PacketSize=128
  Packet structure:
    ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT ttl 63 id 0 protocol 17 offset␣
↪(bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.2) ns3::UdpHeader (length: 108␣
↪49153 > 1234) ns3::SeqTsHeader ((seq=0 time=+0.4s)) Payload (size=88)
+0.400000282s 0 LteEnbRrc:SendPacket(): [INFO ] Send packet to PDCP layer
+0.400000282s 0 LteEnbRrc:SendPacket(): [DEBUG] Packet inspection:
  Packet tags:
    false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [20-128] FlowId=1 PacketId=0 PacketSize=128
  Packet structure:
    ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT ttl 63 id 0 protocol 17 offset␣
↪(bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.2) ns3::UdpHeader (length: 108␣
↪49153 > 1234) ns3::SeqTsHeader ((seq=0 time=+0.4s)) Payload (size=88)
```

As it can be noticed, packet tags are removed, as the RNTI and BID are transferred to the `LtePdcpSapProvider::TransmitPdcpSduParameters` instance. Byte tags and packet structure remain unaltered.

4.3.1 Packet latency

Packets cannot incur latency in `LteEnbRrc`.

4.3.2 Packet drops

There may be a chance that the UE is still asking for resources on the RACH, i.e., `INITIAL_RANDOM_ACCESS`, or there is still pending set up, i.e., `CONNECTION_SETUP`. In that case, the packet cannot be transmitted. To this end, it is placed in the `UeManager/Drop` trace source and discarded.

4.4 LtePdcp

The `LteEnbRrc` sends the packet down the stack by referencing a `LtePdcpSapProvider`. This is an abstract interface in order to implement any kind of PDCP layer. In this case, the simulation makes use of the implementation provided by `LtePdcp`, available in `lte/model/lte-pdcp.cc`.

The `LtePdcp` log component can be enabled at INFO level to check out these log messages:

```
+0.400000282s 0 LtePdcp:DoTransmitPdcpSdu(): [INFO ] Received PDCP SDU
+0.400000282s 0 LtePdcp:DoTransmitPdcpSdu(): [INFO ] Transmit PDCP PDU
```

According to the `LtePdcpSapProvider` interface, the internal method used to receive an incoming packet from upper layers is called `DoTransmitPdcpSdu()`, which in this case is defined as the following:

```
Ptr<Packet> p = params.pdcpSdu;

// Sender timestamp
PdcpTag pdcpTag(Simulator::Now());

LtePdcpHeader pdcpHeader;
pdcpHeader.SetSequenceNumber(m_txSequenceNumber);

m_txSequenceNumber++;
if (m_txSequenceNumber > m_maxPdcpSn)
{
    m_txSequenceNumber = 0;
}

pdcpHeader.SetDcBit(LtePdcpHeader::DATA_PDU);

NS_LOG_LOGIC("PDCP header: " << pdcpHeader);
p->AddHeader(pdcpHeader);
p->AddByteTag(pdcpTag, 1, pdcpHeader.GetSerializedSize());

m_txPdu(m_rnti, m_lcid, p->GetSize());

LteRlcSapProvider::TransmitPdcpPduParameters txParams;
txParams.rnti = m_rnti;
txParams.lcid = m_lcid;
txParams.pdcpPdu = p;

m_rlcSapProvider->TransmitPdcpPdu(txParams);
```

As it can be observed, a PDCP packet tag, identified by `PdcpTag`, and its header, represented by `LtePdcpHeader`, are created and attached to the SDU. The former stores the current simulation time and the latter is set with a specified

sequence number and a D/C bit. While the sequence number is locally tracked through the `m_txSequenceNumber` property, the D/C bit is set to `DATA_PDU` to indicate that this is a data packet and not a control one.

Consequently, if we enable the DEBUG level of `LtePdcp`, we can take a look at how the packet is modified:

```
+0.400000282s 0 LtePdcp:DoTransmitPdcpSdu(): [DEBUG] Packet inspection:
  Packet tags:
    false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [20-128] FlowId=1 PacketId=0 PacketSize=128
  Packet structure:
    ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT ttl 63 id 0 protocol 17 offset␣
↪(bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.2) ns3::UdpHeader (length: 108␣
↪49153 > 1234) ns3::SeqTsHeader ((seq=0 time=+0.4s)) Payload (size=88)
+0.400000282s 0 LtePdcp:DoTransmitPdcpSdu(): [DEBUG] Packet inspection:
  Packet tags:
    false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [22-130] FlowId=1 PacketId=0 PacketSize=128 ns3::PdcpTag [1-
↪2] +4e+08ns
  Packet structure:
    ns3::LtePdcpHeader (D/C=1 SN=0) ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT␣
↪ttl 63 id 0 protocol 17 offset (bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.
↪2) ns3::UdpHeader (length: 108 49153 > 1234) ns3::SeqTsHeader ((seq=0 time=+0.4s))␣
↪Payload (size=88)
```

Notice that there is a new byte tag called `PdcpTag`, which stores the current timestamp of the simulation. Furthermore, most of the packet remains unchanged, if not for the introduction of `LtePdcpHeader` that encapsulates the SDU. This change can be visually represented with the following figure:

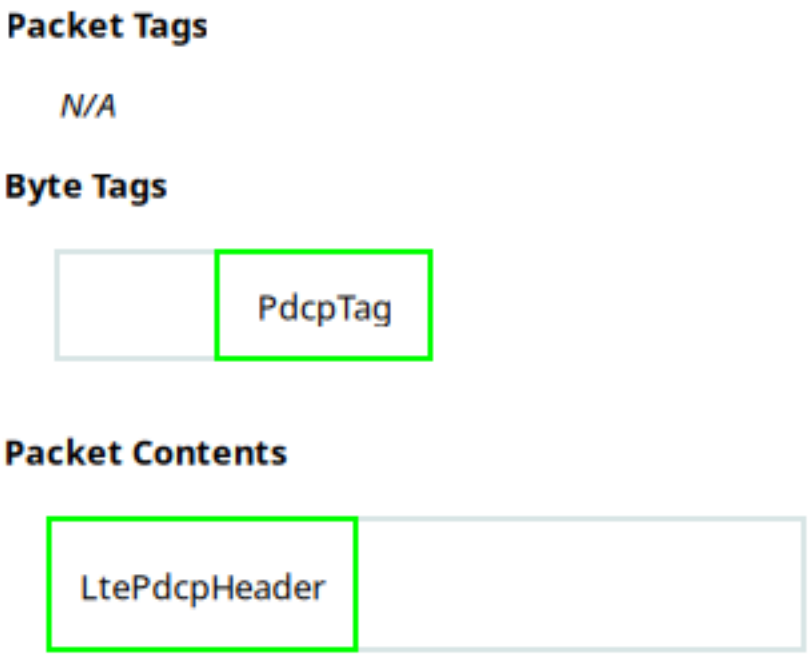

Fig. 3: Structure of the PDCP PDU processed by `LtePdcp`. Green blocks are added at the PDCP layer, while blue ones represent unchanged structure.

Finally, the packet is forwarded to the RLC layer through `LteRlcSapProvider::TransmitPdcpPdu()`, by providing some additional parameters to the PDCP PDU, such as the RNTI and the LCID, under the `LteRlcSap-`

`Provider::TransmitPdcpPduParameters` structure.

4.4.1 Packet latency

Packets cannot incur latency in the `LtePdcp`.

4.4.2 Packet drops

Packets cannot be dropped in the `LtePdcp`.

4.5 LteRlcUm

The RLC layer is designed upon the same templated structure of `LtePdcpSapProvider`, where multiple implementations of the RLC can be defined. On ns-3, such layer is implemented by `LteRlcAm`, `LteRlcTm`, and `LteRlcUm`, according to how the PDCP PDU is transferred to the UE, if on Acknowledged Mode (AM), Transparent Mode (TM), or Unacknowledged Mode (UM), respectively. For this simulation, the `LteRlcUm` is adopted, which is implemented in `lte/model/lte-rlc-um.cc`.

If the `LteRlcUm` log component is enabled at INFO level, it produces the following log messages:

```
+0.400000282s 0 LteRlcUm:DoTransmitPdcpPdu(): [INFO ] Received RLC SDU
+0.400000282s 0 LteRlcUm:DoTransmitPdcpPdu(): [INFO ] New packet enqueued to the RLC␣
↪Tx Buffer
```

As it can be deduced, the RLC SDU is received at `LteRlcUm::DoTransmitPdcpPdu()`, with its implementation that is:

```
if (m_txBufferSize + p->GetSize() <= m_maxTxBufferSize)
{
    if (m_enablePdcpDiscarding)
        // Ignored, as this flag is false by default

    /** Store PDCP PDU */
    LteRlcSduStatusTag tag;
    tag.SetStatus(LteRlcSduStatusTag::FULL_SDU);
    p->AddPacketTag(tag);

    NS_LOG_INFO("New packet enqueued to the RLC Tx Buffer");
    NS_LOG_DEBUG("Packet inspection:" << p->Inspect());
    m_txBuffer.emplace_back(p, Simulator::Now());
    m_txBufferSize += p->GetSize();
    NS_LOG_LOGIC("NumOfBuffers = " << m_txBuffer.size());
    NS_LOG_LOGIC("txBufferSize = " << m_txBufferSize);
}
else
{
    // Discard full RLC SDU
    NS_LOG_INFO("TxBuffer is full. RLC SDU discarded");
    NS_LOG_LOGIC("MaxTxBufferSize = " << m_maxTxBufferSize);
    NS_LOG_LOGIC("txBufferSize    = " << m_txBufferSize);
    NS_LOG_LOGIC("packet size     = " << p->GetSize());
    m_txDropTrace(p);
}
```

(continues on next page)

(continued from previous page)

```
/** Report Buffer Status */
DoReportBufferStatus();
m_rbsTimer.Cancel();
```

The packet remains unchanged, if not for a new tag managed by `LteRlcSduStatusTag`, which keeps track of the fragmentation status of the packet done by the RLC layer. Given that `DoTransmitPdcpPdu()` does not fragment the packet, the state is set to `LteRlcSduStatusTag::FULL_SDU` accordingly.

The packet is then appended to a buffer called `m_txBuffer`, which is a `vector<TxPdu>`, where `TxPdu` is a structure containing (i) a pointer to the packet, and (ii) the time when the packet has been added to the buffer, as dictated by the `Simulator::Now()` call in `m_txBuffer.emplace_back()` instruction.

At the end of the procedure, `DoReportBufferStatus()` is called to alert the queue size at the MAC layer, handled by `LteMacSapProvider`. The buffer report focuses on reporting the queue size and its HOL delay, together with the RNTI and LCID of reference, as it can be noticed by the following excerpt of the said method:

```
Time holDelay(0);
uint32_t queueSize = 0;

if (!m_txBuffer.empty())
{
    holDelay = Simulator::Now() - m_txBuffer.front().m_waitingSince;

    queueSize =
        m_txBufferSize + 2 * m_txBuffer.size(); // Data in tx queue + estimated␣
↪headers size
}

LteMacSapProvider::ReportBufferStatusParameters r;
r.rnti = m_rnti;
r.lcid = m_lcid;
r.txQueueSize = queueSize;
r.txQueueHolDelay = holDelay.GetMilliSeconds();
r.retxQueueSize = 0;
r.retxQueueHolDelay = 0;
r.statusPduSize = 0;

NS_LOG_LOGIC("Send ReportBufferStatus = " << r.txQueueSize << ", " << r.
↪txQueueHolDelay);
m_macSapProvider->ReportBufferStatus(r);
```

As it can be noticed, there is no mention of the actual `m_txBuffer`, which is kept at the RLC layer, until a transmission opportunity is found at the MAC layer, for which there is an upcall to `LteRlcUm::DoNotifyTxOpportunity()` done by the MAC scheduler. Indeed, the INFO log messages suggest that the buffer acquires two RLC SDUs before a transmission opportunity takes place:

```
+0.400000282s 0 LteRlcUm:DoTransmitPdcpPdu(): [INFO ] Received RLC SDU
+0.400000282s 0 LteRlcUm:DoTransmitPdcpPdu(): [INFO ] New packet enqueued to the RLC␣
↪Tx Buffer
+0.400002262s 0 LteRlcUm:DoTransmitPdcpPdu(): [INFO ] Received RLC SDU
+0.400002262s 0 LteRlcUm:DoTransmitPdcpPdu(): [INFO ] New packet enqueued to the RLC␣
↪Tx Buffer
+0.400062500s 0 LteRlcUm:DoNotifyTxOpportunity(): [INFO ] RLC layer is preparing data␣
↪for the following Tx opportunity of 81 bytes for RNTI=2, LCID=4, CCID=0, HARQ ID=19,
↪ MIMO Layer=0
```

The `LteRlcUm::DoNotifyTxOpportunity()` prepares the data to transmit in the following way:

```
if (txOpParams.bytes <= 2)
{
    // Stingy MAC: Header fix part is 2 bytes, we need more bytes for the data
    NS_LOG_INFO("TX opportunity too small - Only " << txOpParams.bytes << " bytes");
    return;
}

Ptr<Packet> packet = Create<Packet>();
LteRlcHeader rlcHeader;

// Build Data field
uint32_t nextSegmentSize = txOpParams.bytes - 2;
// ...
```

First of all, the opportunity window size is checked to ensure that we have more than two bytes, given that the MAC header requires that size. Next, an empty packet is created, called `packet`. Such packet must be equivalent to the size of the transmission opportunity, for which the variable `nextSegmentSize` is used to understand how much data contained in `m_txBuffer` can be transferred. In this case, we can transfer up to 79 bytes, which is the result of the `txOpParams.bytes`, set at 81 bytes, minus 2 bytes due to the MAC header size.

```
Ptr<Packet> firstSegment = m_txBuffer.begin()->m_pdu->Copy();
Time firstSegmentTime = m_txBuffer.begin()->m_waitingSince;
// ...
while (firstSegment && (firstSegment->GetSize() > 0) && (nextSegmentSize > 0))
{
    if ((firstSegment->GetSize() > nextSegmentSize) ||
            // Segment larger than 2047 octets can only be mapped to the end of the␣
↪Data field
            (firstSegment->GetSize() > 2047))
    {
        // The packet 'firstSegment' must be fragmented to fit in our segment being␣
↪prepared for MAC TX
    }
    else if ((nextSegmentSize - firstSegment->GetSize() <= 2) || m_txBuffer.empty())
    {
        // If the packet fits the segment and there are no other packets to TX, just␣
↪add it without fragmenting it
    }
    else // (firstSegment->GetSize () < m_nextSegmentSize) && (m_txBuffer.size () > 0)
    {
        // If there are still other packets to TX, add the current packet
↪'firstSegment' and update nextSegmentSize
    }
}
```

From the head of `m_txBuffer`, the first RLC SDU is taken and it is called `firstSegment`, together with its waiting time. Until we have space for the transmission opportunity, tracked by `nextSegmentSize`, the while loop is executed.

Of course, if the `firstSegment` is too big to fit in the transmission opportunity, it is fragmented. The `LteRlcSduStatusTag` is then updated to check if the fragment is the `FIRST_SEGMENT`, `MIDDLE_SEGMENT`, or `LAST_SEGMENT`, according to how many parts the packet is fragmented. Fragments that do not fit the opportunity window size are inserted back to the head of `m_txBuffer`. If such fragment is not the last, the RLC Header `DATA_FIELD_FOLLOWS` bit is set to alert that this packet does not contain the entire PDCP packet. This will be useful to schedule new transmission opportunities in the future and ensure correct packet integration.

There are other cases that could happen, which are handled by the conditional clause inside the while loop. If there is only

one packet left in the buffer and it fits the current segment, it is just placed to the segment. Furthermore, if there is still more space left on the segment and other packets are waiting in the `m_txBuffer`, the current packet `firstSegment` is placed in the segment and the left space, tracked by `nextSegmentSize`, is evaluated to add a new packet in the segment at the next while loop iteration.

This operation can be better visualized by observing the following figure:

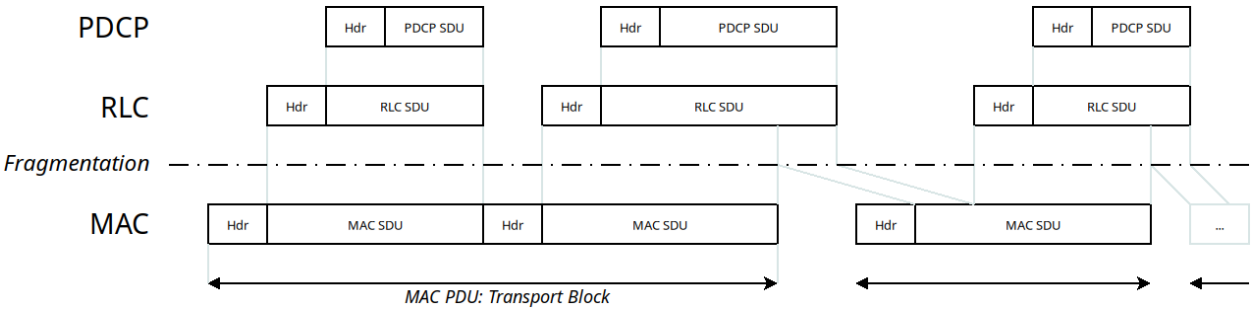

Fig. 4: Overview of how the packets buffered by the RLC layer are combined to prepare a segment with a size equal to the transmission opportunity granted at the MAC layer.

After the fragmentation procedure, the RLC header, handled by `LteRlcHeader` data structure, is set up. The sequence number is set and kept track via the local property `m_sequenceNumber`. Moreover, the framing information is set to declare if the last byte of the segment corresponds or not to the end of a RLC SDU. Finally, a `RlcTag` byte tag is added and linked to the RLC header, which saves the current simulation time.

It is possible to enable the DEBUG log level along the INFO one to inspect the packet sent to the MAC:

```
+0.400062500s 0 LteRlcUm:DoNotifyTxOpportunity(): [INFO ] Forward RLC PDU to MAC Layer
+0.400062500s 0 LteRlcUm:DoNotifyTxOpportunity(): [DEBUG] Packet inspection:
  Packet tags:
    false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [24-81] FlowId=1 PacketId=0 PacketSize=128 ns3::PdcpTag [3-
↪4] +4e+08ns ns3::RlcTag [1-2] +4.00062e+08ns
  Packet structure:
    ns3::LteRlcHeader (Len=2 FI=1 E=0 SN=0) ns3::LtePdcpHeader (D/C=1 SN=0)␣
↪ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT ttl 63 id 0 protocol 17 offset␣
↪(bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.2) ns3::UdpHeader (length: 108␣
↪49153 > 1234) ns3::SeqTsHeader ((seq=0 time=+0.4s)) Payload Fragment [0:37]
```

As expected, the incoming RLC SDU has been fragmented. The first RLC PDU that can be seen here contains:

- The `LteRlcHeader`, which requires 2 bytes, as it can be seen on the `Len` property in the packet inspection. Furthermore, the `FI` framing info is set to 1, which indicates that the first byte of this PDU corresponds to the first byte of a RLC SDU, but the last byte of this PDU does not correspond to the last byte of the SDU. The `SN` sequence number is zero, as expected, along the `E` extension bit, which means that the header does not have any extensions.
- The `LtePdcpHeader`, which covers 2 bytes.
- The `Ipv4Header`, which has 20 bytes.
- The `UdpHeader`, which occupies 8 bytes.
- The `SeqTsHeader`, which requires 12 bytes.

In total, this packet has 44 bytes reserved for the various headers, while the remaining 37 bytes are left to the payload, which is incomplete. Remember that the payload is of 88 bytes, this means that there is a fragment of 51 bytes waiting in the `m_txBuffer`.

Indeed, in the next RLC PDU forward to the MAC layer, it is possible to notice this other log message:

```
+0.400125000s 0 LteRlcUm:DoNotifyTxOpportunity(): [INFO ] Forward RLC PDU to MAC Layer
+0.400125000s 0 LteRlcUm:DoNotifyTxOpportunity(): [DEBUG] Packet inspection:
  Packet tags:
    false
  Packet byte tags:
    ns3::Ipv4FlowProbeTag [4-55] FlowId=1 PacketId=0 PacketSize=128␣
↪ns3::Ipv4FlowProbeTag [77-81] FlowId=1 PacketId=1 PacketSize=128 ns3::PdcpTag [56-
↪57] +4.001e+08ns ns3::RlcTag [1-4] +4.00125e+08ns
  Packet structure:
    ns3::LteRlcHeader (Len=4 FI=3 E=1 SN=1 E=0 LI=51 ) Payload Fragment [37:88]␣
↪ns3::LtePdcpHeader (D/C=1 SN=1) ns3::Ipv4Header (tos 0x0 DSCP Default ECN Not-ECT␣
↪ttl 63 id 1 protocol 17 offset (bytes) 0 flags [none] length: 128 1.0.0.2 > 7.0.0.
↪2) ns3::UdpHeader Fragment [0:4]
```

By observing the `LteRlcHeader` properties, it is possible to notice that its length `Len` is now of 4 bytes, due to the fact that the extension bit `E` is set to 1 and the length indicator `LI` indicates that there are 51 bytes belonging to a previous transmission. Indeed, it is possible to see that right after the RLC header properties, there is `Payload Fragment [37:88]`, which is the byte range indicating the remaining 51 bytes belonging to the original RLC SDU. The same reasoning can be applied on further transmissions, as in this RLC PDU there is another SDU fragment starting from `ns3::LtePdcpHeader`.

Once the RLC PDU is ready, it is pushed to the MAC SAP through the `TransmitPdu()` interface method, together with the RNTI, LCID, CCID, HARQ ID, and MIMO layer.

4.5.1 Packet latency

Packets cannot incur latency in the `LteRlcUm`.

4.5.2 Packet drops

Packets can be dropped if the RLC buffer `m_txBuffer` is full, or the `m_enablePdcpDiscarding` is enabled, which analyzes timing budget and conditions of the RLC SDU and may drop it.

4.6 NrGnbMac

The `NrGnbMac`, available in `nr/model/nr-gnb-mac.cc`, implements the MAC SAP for NR communications.

If we enable the `NrGnbMac` log component at INFO level, it is possible to see the connection initialization with the UEs, both in terms of RACH communication and RNTI allocation:

```
+0.016125000s 0  [ CellId 2, bwpId 0] NrGnbMac:DoSlotDlIndication(): [INFO ]␣
↪Informing MAC scheduler of the RACH preamble for RAPID 1 in slot FrameNum: 1␣
↪SubFrameNum: 6 SlotNum: 4; Allocated RNTI: 1
+0.016125000s 0  [ CellId 2, bwpId 0] NrGnbMac:DoSlotDlIndication(): [INFO ]␣
↪Informing MAC scheduler of the RACH preamble for RAPID 0 in slot FrameNum: 1␣
↪SubFrameNum: 6 SlotNum: 4; Allocated RNTI: 2
+0.016125000s 0  [ CellId 2, bwpId 0] NrGnbMac:SendRar(): [INFO ] In slot FrameNum: 1␣
↪SubFrameNum: 6 SlotNum: 2 send to PHY the RAR message for RNTI 1 rapId 1
+0.016125000s 0  [ CellId 2, bwpId 0] NrGnbMac:SendRar(): [INFO ] In slot FrameNum: 1␣
↪SubFrameNum: 6 SlotNum: 2 send to PHY the RAR message for RNTI 2 rapId 0
```

At first glance, it is possible to note that these logs present two key information, i.e., `CellId` and `bwpId`. These can be used as contextual information to better follow the traffic served by a certain cell on a BWP. To learn more how these IDs work, please refer to Section 2.2 of the NR manual.

Once the RLC transmission buffer is updated, the `NrGnbMac::DoReportBufferStatus()` intermediates with the MAC scheduler SAP to report the buffer status:

```
+0.400000282s 0  [ CellId 2, bwpId 0] NrGnbMac:DoReportBufferStatus(): [INFO ]␣
↪Reporting RLC buffer status update to MAC Scheduler for RNTI=2, LCID=4,␣
↪Transmission Queue HOL Delay=0, Transmission Queue Size=132, Retransmission Queue␣
↪HOL delay=0, Retransmission Queue Size=0, PDU Size=0
+0.400002262s 0  [ CellId 3, bwpId 1] NrGnbMac:DoReportBufferStatus(): [INFO ]␣
↪Reporting RLC buffer status update to MAC Scheduler for RNTI=1, LCID=4,␣
↪Transmission Queue HOL Delay=0, Transmission Queue Size=1284, Retransmission Queue␣
↪HOL delay=0, Retransmission Queue Size=0, PDU Size=0
```

By following these messages it is possible to track the `Transmission Queue Size`, which is filled by incoming RLC PDUs and emptied by the scheduled transmissions. By default, the `NrHelper` considers the TDMA RR as MAC Scheduler for this example, handled by `NrMacSchedulerTdmaRR`. Schedulers are extensively covered in both the NR manual, Sections 2.5.2 and 2.5.3, and the doxygen documentation.

The MAC layer receives scheduled allocations with the call to `NrGnbMac::DoSchedConfigIndication()`. All the scheduler parameters are packed in a `NrMacSchedSapUser::SchedConfigIndParameters` structure. Among these parameters, the HARQ Process ID is critical to fully trace (i) when an allocation is scheduled, (ii) when a frame is transmitted, and (iii) when the UE ACKs the correct reception of the frame. These are critical information that helps also evaluate the packet latency in this layer.

For instance, let's take a look at these INFO log messages, focusing the attention on finding correspondences with the `HARQ Process ID`:

```
+0.400062500s 0  [ CellId 2, bwpId 0] NrGnbMac:DoSchedConfigIndication(): [INFO ] New␣
↪scheduled data TX in DL for HARQ Process ID: 19, Var. TTI from symbol 1 to 13. 1␣
↪TBs of sizes [ 84 ] with MCS [ 0 ]
+0.400062500s 0  [ CellId 2, bwpId 0] NrGnbMac:DoSchedConfigIndication(): [INFO ]␣
↪Notifying RLC of TX opportunity for HARQ Process ID 19 LC ID 4 stream 0 size 81␣
↪bytes
+0.400062500s 0  [ CellId 2, bwpId 0] NrGnbMac:DoTransmitPdu(): [INFO ] Sending MAC␣
↪PDU to PHY Layer
...
+0.400374995s 0  [ CellId 2, bwpId 0] NrGnbMac:DoDlHarqFeedback(): [INFO ] HARQ-ACK␣
↪UE RNTI 2 HARQ Process ID 19 Stream ID 0
```

From the first message we can observe that a new transmission of just one stream of 84 bytes is scheduled for transmission. Consequently, from the second log message the MAC notifies this opportunity to the RLC layer. After that, the MAC PDU is prepared. After a while, the ACK from the UE arrives. This means that the frame took 312us to be sent and acknowledged.

Here the RLC PDU is made and forwarded to the MAC layer by calling its `TransmitPdu()` method, which in this case is implemented by `NrGnbMac::DoTransmitPdu()`. At the end of packet incapsulation, the PDU is forwarded to the PHY SAP, along with its SfnSf and the starting symbol index of the DCI, shown in the log messages as `Var. TTI from symbol 1 to 13`. To learn more how a variable TTI works, please refer to Section 2.5.1 of the NR manual.

Furthermore, observe how the TB size is larger for BWP with ID 1:

```
+0.400250000s 0  [ CellId 3, bwpId 1] NrGnbMac:DoSchedConfigIndication(): [INFO ] New␣
↪scheduled data TX in DL for HARQ Process ID: 19, Var. TTI from symbol 1 to 13. 1␣
↪TBs of sizes [ 348 ] with MCS [ 0 ]
...
```

(continues on next page)

(continued from previous page)

```
+0.401499997s 0  [ CellId 3, bwpId 1] NrGnbMac:DoDlHarqFeedback(): [INFO ] HARQ-ACK␣
↪UE RNTI 1 HARQ Process ID 19 Stream ID 0
```

but the latency has increased to 1,249us.

4.6.1 Packet latency

Packets does incur latency in `NrGnbMac`, as they can be left in the buffer until a successful HARQ ACK is received from the UE. Furthermore, there is a difference in latency, depending on the BWP configuration.

4.6.2 Packet drops

Packets cannot be dropped in `NrGnbMac`.

4.7 NrGnbPhy

`NrGnbPhy` tailors the `NrPhy` according to how the gNB should behave at its PHY layer. Its implementation can be found in (i) `model/nr-gnb-phy.cc`, `model/nr-phy.cc`, and `model/nr-phy-mac-common.cc`. The PHY layer is extensively covered in Section 2.3 of the NR manual. Furthermore, there is more technical description on how it works at the doxygen of NrGnbPhy and doxygen of NrPhy classes.

Please note that:

1. As the data plane is correctly simulated, the control plane does not have any kind of loss, thus making it an ideal channel for control information.
2. This simulation is configured by enabling both `NrGnbPhy` and `NrPhy` log components:

```
$ NS_LOG="NrGnbPhy=info|prefix_all:NrPhy=info|prefix_all" ./ns3 run cttc-nr-
↪demo -- --simTime=1s &> out.log
```

At the start of the simulation, the PHY layer is configured. As we are in a TDD context, its pattern is set. Furthermore, according to the bandwidth available, the number of RBs per BWP is set. Finally, the channel access is requested and obtained. Given the simplicity of this simulation, the channel is granted for a long time, but take into account that it is released as soon as the gNB does not use it for data transmission. All this information can be obtained in the following log messages:

```
+0.000000000s -1  [ CellId 0, bwpId 65535] NrGnbPhy:SetTddPattern(): [INFO ] Set␣
↪pattern : F|F|F|F|F|F|F|F|F|F|
+0.000000000s -1  [ CellId 0, bwpId 0] NrPhy:DoUpdateRbNum(): [INFO ] Updated RbNum␣
↪to 33
+0.000000000s -1  [ CellId 0, bwpId 1] NrPhy:DoUpdateRbNum(): [INFO ] Updated RbNum␣
↪to 133
+0.000000000s -1  [ CellId 2, bwpId 0] NrPhy:DoUpdateRbNum(): [INFO ] Updated RbNum␣
↪to 6
+0.000000000s -1  [ CellId 2, bwpId 0] NrPhy:DoUpdateRbNum(): [INFO ] Updated RbNum␣
↪to 6
+0.000000000s 0  [ CellId 2, bwpId 0] NrGnbPhy:StartSlot(): [INFO ] Channel not␣
↪granted, request the channel
+0.000000000s 0  [ CellId 2, bwpId 0] NrGnbPhy:ChannelAccessGranted(): [INFO ]␣
↪Channel access granted for +9.22337e+18ns, which corresponds to 147573952589675␣
↪slot in which each slot is +62500ns. We lost +62500ns
```

(continues on next page)

(continued from previous page)

```
...
+0.000062500s 0  [ CellId 2, bwpId 0] NrGnbPhy:EndSlot(): [INFO ] Release the channel␣
↪because we did not have any data to maintain the grant
```

The `F` in the TDD pattern means that for each TDD slot it will be possible to handle any type of frame, from DL to UL, which can be of `CTRL` or `DATA` type. More information can be found in the NR doxygen regarding the LteNrTddSlotType.

The gNB's PHY layer receives the MAC PDU through the `NrPhy::SetMacPdu()` method. First of all, the numerology is checked to understand if the PHY is working at that numerology at the moment. If such condition is true, the PDU is appended to a `ns3::PacketBurst` object, which is an abstraction to a list of packets. Such list is mapped to contextual information, which is composed by the stream ID and starting symbol.

Indeed, it is possible to notice this INFO log message, which signals the MAC PDU entrance in this layer, along with its properties, such as SfnSf (used to keep track of the frame, subframe, and slot number) and starting symbol:

```
+0.400125000s 0  [ CellId 2, bwpId 0] NrPhy:SetMacPdu(): [INFO ] Adding a packet for␣
↪the Packet Burst of FrameNum: 40 SubFrameNum: 0 SlotNum: 4 at sym 1
```

As the timeslot order is handled by the scheduler, the transmission is led by an event loop, which calls `NrGnbPhy::StartSlot()` and queries the packet burst through `NrPhy::PushFrontSlotAllocInfo()`. Packets are finally transmitted over the air via `DlData()`, which then interacts with the spectrum through `SendDataChannels()`. Upon the same logic, packets are received from `UlData()` and `PhyDataPacketReceived()`.

From this point onwards, the `NbGnbPhy` interacts with `NrSpectrumPhy::StartTxDataFrames()`, which acts as an interface between the gNB PHY layer and the channel. `NrSpectrumPhy` acts as a state machine to know what the PHY layer (at the BWP of interest) is currently doing, from transferring/receiveing data or control information or it is in idle. Once a pucket burst is given with its related set of control messages and duration of transmission, the structure `NrSpectrumSignalParameterDataFrame` is created. This object contains the aformentioned information, plus the cell identifier (`GetCellId()`) and the transmission PSD. Such information is then forwarded to the channel.

The log messages are quite verbose but in a constant pattern until there is data to transmit. Indeed, it is possible to notice these log messages once data arrive at the PHY layer:

```
+0.400187500s 0  [ CellId 2, bwpId 0] NrGnbPhy:RetrieveDciFromAllocation(): [INFO ]␣
↪Send DCI to RNTI 2 from sym 1 to 13
+0.400187500s 0  [ CellId 2, bwpId 0] NrGnbPhy:FillTheEvent(): [INFO ] Scheduled␣
↪allocation RNTI=0|DL|SYM=0|NSYM=1|TYPE=2|BWP=0|HARQP=0|RBG=[0;32] at +0ns
+0.400187500s 0  [ CellId 2, bwpId 0] NrGnbPhy:FillTheEvent(): [INFO ] Scheduled␣
↪allocation RNTI=2|DL|SYM=1|NSYM=12|McsStream0=0|TBsStream0=84|NdiStream0=1|RvS
tream0=0|TYPE=1|BWP=0|HARQP=19|RBG=[0;32] at +4464ns
+0.400187500s 0  [ CellId 2, bwpId 0] NrGnbPhy:FillTheEvent(): [INFO ] Scheduled␣
↪allocation RNTI=0|UL|SYM=13|NSYM=1|TYPE=2|BWP=0|HARQP=0|RBG=[0;32] at +58032ns
+0.400191964s 0  [ CellId 2, bwpId 0] NrGnbPhy:DlData(): [INFO ] ENB TXing DL DATA␣
↪frame FrameNum: 40 SubFrameNum: 0 SlotNum: 3 symbols 1-12 start +4.00192e+08ns end␣
↪+4.00246e+08ns
+0.400250000s 0  [ CellId 3, bwpId 1] NrGnbPhy:EndSlot(): [INFO ] Release the channel␣
↪because we did not have any data to maintain the grant
```

Here the `NrGnbPhy::FillTheEvent()` scheduled the Variable TTI in the simulator as events. The first scheduled information sends one symbol of control information in DL to the UE; the second one sends is the data TB, whereas the third one is to receive in UL the ACK from the UE.

4.7.1 Packet latency

At this layer each packet will get latency based on different features that characterize NR. For instance, the processing times affect packet latency and it is parametrized by `L1L2CtrlLatency` in timeslots. This parameter is hardcoded to 2, which indicates that the allocation requires two timeslots before seeing the packet going in the air.

It can be noticed that the packet gained 0.19 ms of latency by reaching the time scheduled for transmission and the total transmission of the data of interest requested 54 us.

Additionally, further propagation delay may be added by the pathloss models. In this case, it is assumed that the electromagnetic waves travel at the speed of light.

4.7.2 Packet drops

Packets cannot be dropped in the `NrGnbPhy`, neither in `NrSpectrumPhy`, but they may be dropped or distorted by the pathloss model if the SINR is not sufficient for a correct transmission.

4.8 NrUePhy

The other end of the channel, which in this case is the UE, is implemented by `NrUePhy`, available at `model/nr/nr-ue-phy.cc`. The log components can be enabled and correlated to that of the gNB to evaluate what happens at the PHY layer:

```
$ NS_LOG="NrGnbPhy:NrUePhy" ./ns3 run cttc-nr-demo -- --simTime=1s &> out.log
```

The following log excerpt can be analyzed:

```
+0.400191964s 0  [ CellId 2, bwpId 0] NrGnbPhy:DlData(): [DEBUG] Starting DL DATA TTI␣
↪at symbol 1 to 13
+0.400191964s 0  [ CellId 2, bwpId 0] NrGnbPhy:DlData(): [INFO ] ENB TXing DL DATA␣
↪frame FrameNum: 40 SubFrameNum: 0 SlotNum: 3 symbols 1-12 start +4.00192e+08ns end␣
↪+4.00246e+08ns
+0.400191964s 2  [ CellId 2, bwpId 0] NrUePhy:EndVarTti(0x1fd6a90)
+0.400191964s 2  [ CellId 2, bwpId 0] NrUePhy:EndVarTti(): [INFO ] DCI started at␣
↪symbol 0 which lasted for 1 symbols finished
+0.400191964s 2  [ CellId 2, bwpId 0] NrUePhy:TryToPerformLbt(0x1fd6a90)
+0.400191964s 2  [ CellId 2, bwpId 0] NrUePhy:TryToPerformLbt(): [INFO ] Not␣
↪scheduling LBT: the UE has a channel status that is GRANTED
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:EndVarTti(0x1fb2ac0)
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:EndVarTti(): [INFO ] DCI started at␣
↪symbol 0 which lasted for 1 symbols finished
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:TryToPerformLbt(0x1fb2ac0)
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:TryToPerformLbt(): [INFO ] This data␣
↪DCI ends at +4.00239e+08ns which is inside the LBT shared COT (the limit is +4.
↪00221e+08ns). No need for LBT
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:TryToPerformLbt(): [INFO ] Not␣
↪scheduling LBT: the UE has a channel status that is GRANTED
+0.400191964s 0  [ CellId 2, bwpId 0] NrGnbPhy:EndVarTti(0x1f91f10, 0.400192)
+0.400191964s 0  [ CellId 2, bwpId 0] NrGnbPhy:EndVarTti(): [DEBUG] DCI started at␣
↪symbol 0 which lasted for 1 symbols finished
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:StartVarTti(0x1fb2ac0)
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:DlData(0x1fb2ac0)
+0.400191964s 1  [ CellId 2, bwpId 0] NrUePhy:DlData(): [DEBUG] UE2 stream 0 RXing DL␣
↪DATA frame for symbols 1-12 num of rbg assigned: 33        start +4.00192e+08ns end␣
↪+4.00246e+08ns
```

During the reception, the UE will `TryToPerformLbt()` to do LBT before the transmission of an uplink control packet.

Upon reception of a packet, the `PhyRxDataEndOk` callback is executed. Consequently, `NrUePhy::PhyDataPacketReceived()` is called, which forwards the packet, along with node ID and the TB decode latency, to `NrUePhySapUser`, which reiterates through the SAP interface to to propagate the new packet to upper layers.

Furthermore, RX data can be traced through `NrSpectrumPhy/RxDataTrace` callback.

4.8.1 Packet latency

The same logic of `NrGnbPhy` applies.

4.8.2 Packet drops

The same logic of `NrGnbPhy` applies. The packet can be dropped if the TBs are corrupted.

4.9 NrUeMac

The `NrUeMac` is implemented in `model/nr-ue-mac.cc`. Thanks to the SAP and callback mentioned in `NrPhy`, `NrUeMac::DoReceivePhyPdu()` received the notification of a new packet.

```
LteRadioBearerTag tag;
p->RemovePacketTag(tag);

if (tag.GetRnti() != m_rnti) // Packet is for another user
{
    return;
}

NrMacHeaderVs header;
p->RemoveHeader(header);

LteMacSapUser::ReceivePduParameters rxParams;
rxParams.p = p;
rxParams.rnti = m_rnti;
rxParams.lcid = header.GetLcId();

auto it = m_lcInfoMap.find(header.GetLcId());

// p can be empty. Well, right now no, but when someone will add CE in downlink,
// then p can be empty.
if (rxParams.p->GetSize() > 0)
{
    it->second.macSapUser->ReceivePdu(rxParams);
}
```

Here it is possible to see the implementation of said method. It is clear that the packet is checked to ensure that we are handling a packet with an expected RNTI, otherwise we have received a packet intended for another UE. The MAC header is then removed and the PDU is forwarded, along the RNTI and LCID, through the SAP to be processed by upper layers.

4.9.1 Packet latency

Packets cannot incur latency in the `NrUeMac`.

4.9.2 Packet drops

Packets may be dropped by `NrUeMac` if the RNTI does not match with the expected one.

4.10 NrUeNetDevice

The incoming packet is quickly processed via `LteUeRrc::DoReceivePdcpSdu()`, which forwards it to the AS SAP of the User, which in this case is `EpcUeNas`. The latter is signaled via `DoRecvData()`, which does an up-callback to `NrUeNetDevice::Receive()`. The implementation of such callback is the same for both UE and gNB, and thus it can be found in `NrNetDevice`, i.e., `contrib/nr/model/nr-net-device.cc`:

```
Ipv4Header ipv4Header;
Ipv6Header ipv6Header;

if (p->PeekHeader(ipv4Header) != 0)
{
    NS_LOG_INFO("IPv4 stack...");
    m_rxCallback(this, p, Ipv4L3Protocol::PROT_NUMBER, Address());
}
else if (p->PeekHeader(ipv6Header) != 0)
{
    NS_LOG_INFO("IPv6 stack...");
    m_rxCallback(this, p, Ipv6L3Protocol::PROT_NUMBER, Address());
}
else
{
    NS_ABORT_MSG("NrNetDevice::Receive - Unknown IP type...");
}
```

If the `NrNetDevice` log component is enabled, the simulation produced the following message:

```
+0.400533031s 1 NrNetDevice:Receive(): [INFO ] Received 128 bytes on␣
↪00:00:00:00:00:09. IPv4 packet from 1.0.0.2 to 7.0.0.2
```

Finally, such packet is then available for the upper layers to consume, i.e., transport and application ones, thanks to the `m_rxCallback` up-call.

4.10.1 Packet latency

Packets cannot incur latency in the `LteUeRrc`, `EpcUeNas`, and `NrUeNetDevice`.

4.10.2 Packet drops

Packets cannot be dropped in the `LteUeRrc`, `EpcUeNas`, and `NrUeNetDevice`.