User Plane Function Offloading in P4 switches for enhanced 5G Mobile Edge Computing

Francesco Paolucci*, Davide Scano†, Filippo Cugini*, Andrea Sgambelluri†,
Luca Valcarenghi†, Carlo Cavazzoni‡, Giuseppe Ferraris‡, Piero Castoldi†
*CNIT, Pisa, Italy. †Scuola Superiore Sant'Anna, Pisa, Italy. ‡Telecom Italia, Turin, Italy
*Corresponding author: francesco.paolucci@cnit.it

2021 17th International Conference on the Design of Reliable Communication Networks (DRCN) | 978-1-6654-2234-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/DRCN51631.2021.9477338

***Abstract*—This demo shows a 5G X-haul testbed enhanced with P4 switches implementing the offloading of the User Plane Function module. The P4 code includes GTP protocol encapsulation/decapsulation function, fully configurable N3-N6-N9 steering, and advanced online monitoring of the experienced latency metadata.**

***Index Terms*—P4, SDN, UPF, offloading, BMv2, beyond-5G.**

I. Introduction

In the foreseen edge cloud architecture, selected 5G functions in the Multi-access Edge Computing (MEC) may be offloaded to dedicated programmable hardware, or, alternatively, to a programmable network device already existing in the 5G infrastructure (e.g., a programmable switch). Such function offloading strategies are of extreme interest for the deployment of 5G and beyond networks, where the overall offered network capacity will need to sustain extremely low latencies, not always achievable through software virtualizations at the IT platforms (edge, cloud). To this goal, the Software Defined Networking (SDN) network programmability at the data plane level, resorting to the platform-agnostic and high-level P4 language, may play a key role to enable 5G functions directly inside SDN network devices [1]. Recent research trends are exploring selected hardware offloading solutions. For example, FPGA implementation of offloading the GPRS Tunnelling Protocol (GTP) function in the MEC platforms are proposed in [2], or slicing solution based on programmable switch in [3] and, finally, stateless translations of GTP protocol in Segment Routing version 6 in [4]. Offloading of 5G virtualised Radio Access Network (vRAN) functions to programmable hardware has been also proposed [5].

In this demo, an implementation of UPF offloaded in a P4 switch running on the Behavioral Model version 2 (BMv2) [6] is shown. The demonstration includes the description of the main functions implemented at the UPF P4 switch: 1) the GTP User plane (GTP-U) encapsulation/decapsulation functions, 2) the automatic forwarding and steering functions serving all the required UPF interfaces resorting to simple flow entry configurations, 3) the configurable monitoring of selected GTP flows performance such as the online latency experienced at the switch. The same P4 code is utilized to deploy different P4 switch in a comprehensive network testbed to demonstrate the efficiency and the versatility of the UPF offload functions. During the demo, scalability performance are shown through latency monitoring when a high amount of 5G traffic flows are enforced.

This paper has been supported by TIM under the cooperation agreement with Scuola Superiore Sant'Anna for the 5G MISE trial in Bari and Matera 2018-2022, and by the BRAINE Project, funded by ECSEL Joint Undertaking under grant agreement No. 876967. This paper was carried out in the framework of the Department of Excellence in Robotics and Artificial Intelligence funded by MIUR to Scuola Superiore Sant'Anna.

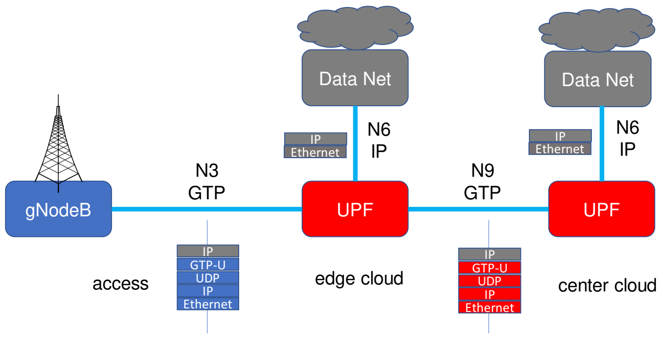
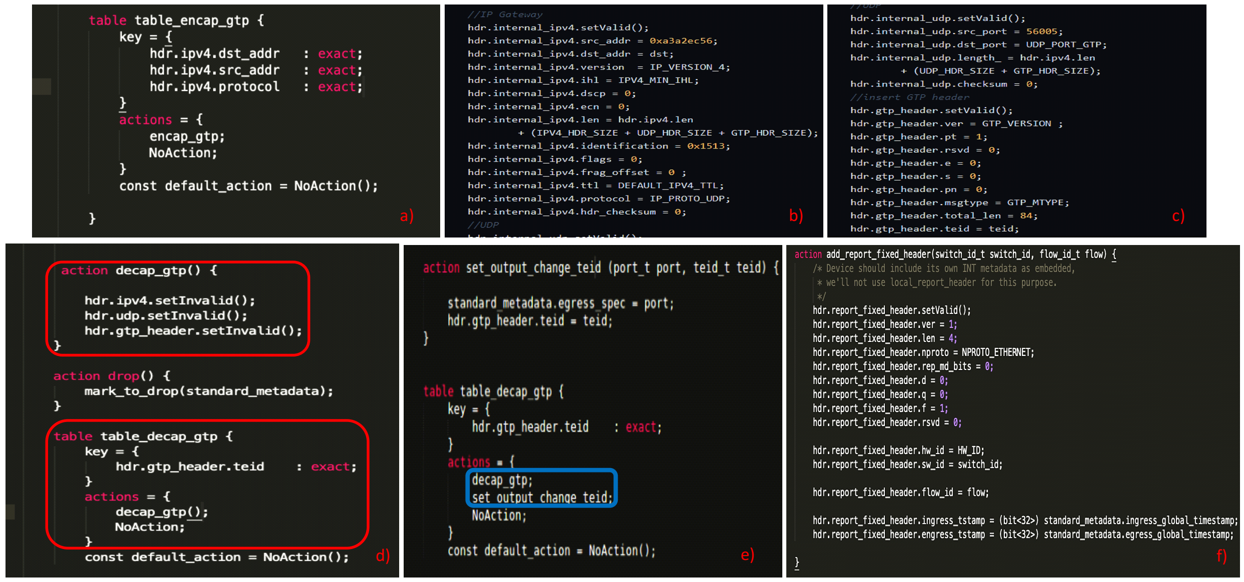


Fig. 1. Functional view of the User Plane Function: interfaces and connectivity in the 5G architecture.

II. P4 User Plane Function

Fig. 1 shows the UPF functional view in the 5G MEC architecture. Typically, UPF is implemented as a cloud/edge application (running as a virtual machine or a container in the IT space). The main role of the UPF is to map incoming and outgoing traffic connecting distributed gNodeB and the gateways to the edge-cloud segment and the metro/transport network. To guarantee the correct mapping between IP flows and the selected gNodeB in both downlink and uplink directions, each IP or layer-4 flow needs to be mapped in a GTP tunnel to reach the desired gNodeB, where the 5G stack will be applied to exploit the radio link to the mobile end user. Moreover, in the case of multiple gateways, as depicted in Fig. 1, different GTP tunnels are configured and swapped at each hop to map the correct destination gateway address. The interfaces involved in the GTP-U mapping and de-mapping operations are the following:

- *Interface N6*, connecting the Data Net to the first UPF gateway, carrying IP packets;
- *Interface N3*, connecting the gNodeB to the UPF, carrying GTP-U tunnelled packets;
- *Interface N9*, connecting different UPFs, carrying GTP-U tunnelled packets.



```
table table_encap_gtp {
    key = {
        hdr.ipv4.dst_addr   : exact;
        hdr.ipv4.src_addr   : exact;
        hdr.ipv4.protocol   : exact;
    }
    actions = {
        encap_gtp;
        NoAction;
    }
    const default_action = NoAction();

}
```
a)

```
//IP Gateway
hdr.internal_ipv4.setValid();
hdr.internal_ipv4.src_addr = 0xa3a2ec56;
hdr.internal_ipv4.dst_addr = dst;
hdr.internal_ipv4.version  = IP_VERSION_4;
hdr.internal_ipv4.ihl = IPV4_MIN_IHL;
hdr.internal_ipv4.dscp = 0;
hdr.internal_ipv4.ecn = 0;
hdr.internal_ipv4.len = hdr.ipv4.len
        + (IPV4_HDR_SIZE + UDP_HDR_SIZE + GTP_HDR_SIZE);
hdr.internal_ipv4.identification = 0x1513;
hdr.internal_ipv4.flags = 0;
hdr.internal_ipv4.frag_offset = 0 ;
hdr.internal_ipv4.ttl = DEFAULT_IPV4_TTL;
hdr.internal_ipv4.protocol = IP_PROTO_UDP;
hdr.internal_ipv4.hdr_checksum = 0;
//UDP
```
b)

```
hdr.internal_udp.setValid();
hdr.internal_udp.src_port = 56005;
hdr.internal_udp.dst_port = UDP_PORT_GTP;
hdr.internal_udp.length_ = hdr.ipv4.len
        + (UDP_HDR_SIZE + GTP_HDR_SIZE);
hdr.internal_udp.checksum = 0;
//insert GTP header
hdr.gtp_header.setValid();
hdr.gtp_header.ver = GTP_VERSION ;
hdr.gtp_header.pt = 1;
hdr.gtp_header.rsvd = 0;
hdr.gtp_header.e = 0;
hdr.gtp_header.s = 0;
hdr.gtp_header.pn = 0;
hdr.gtp_header.msgtype = GTP_MTYPE;
hdr.gtp_header.total_len = 84;
hdr.gtp_header.teid = teid;
```
c)

```
 action decap_gtp() {

    hdr.ipv4.setInvalid();
    hdr.udp.setInvalid();
    hdr.gtp_header.setInvalid();
}

action drop() {
    mark_to_drop(standard_metadata);
}

table table_decap_gtp {
    key = {
        hdr.gtp_header.teid     : exact;
    }
    actions = {
        decap_gtp();
        NoAction;
    }
    const default_action = NoAction();
```
d)

```
action set_output_change_teid (port_t port, teid_t teid) {

    standard_metadata.egress_spec = port;
    hdr.gtp_header.teid = teid;
}


table table_decap_gtp {
    key = {
        hdr.gtp_header.teid     : exact;
    }
    actions = {
        decap_gtp;
        set_output_change_teid;
        NoAction;
    }
    const default_action = NoAction();
```
e)

```
action add_report_fixed_header(switch_id_t switch_id, flow_id_t flow) {
    /* Device should include its own INT metadata as embedded,
     * we'll not use local_report_header for this purpose.
     */
    hdr.report_fixed_header.setValid();
    hdr.report_fixed_header.ver = 1;
    hdr.report_fixed_header.len = 4;
    hdr.report_fixed_header.nproto = NPROTO_ETHERNET;
    hdr.report_fixed_header.rep_md_bits = 0;
    hdr.report_fixed_header.d = 0;
    hdr.report_fixed_header.q = 0;
    hdr.report_fixed_header.f = 1;
    hdr.report_fixed_header.rsvd = 0;

    hdr.report_fixed_header.hw_id = HW_ID;
    hdr.report_fixed_header.sw_id = switch_id;

    hdr.report_fixed_header.flow_id = flow;

    hdr.report_fixed_header.ingress_tstamp = (bit<32>) standard_metadata.ingress_global_timestamp;
    hdr.report_fixed_header.engress_tstamp = (bit<32>) standard_metadata.egress_global_timestamp;

}
```
f)

Fig. 2. Excerpts of UPF P4 code: GTP encap table (a), GTP encap action(b)(c), GTP decap (d), GTP tunnel swap (e), monitoring report header insertion (f).

III. GTP-U ENCAPSULATION AND DECAPSULATION

In this section we describe the key parts of the P4 code utilized to implement the UPF GTP-U encapsulation and decapsulation. The code includes the Protocol header definitions (i.e., Ethernet, IP, UDP and GTP), the protocol parsers, the pipelines with flow tables ad actions.

All the involved protocol headers are defined in the code. By default, Ethernet is parsed and, in the case of IP header (checked by means of the ether_type field), the IP header is parsed as well. The code includes also the definition of the standard UDP and GTP protocol headers. Fig. 2(a) shows an excerpt of the implemented ingress pipeline. Table *table_encap_gtp* selects the IP flows to be GTP-encapsulated and performs packet encapsulation. The execution of this table is conditional, performed only if the incoming packet is not GTP-encapsulated (i.e., a standard IP or UDP packet). The match is performed against the IP addresses and the protocol type. Additional and finer matches are here possible (e.g., layer-4 protocol port) and the possible actions are *encap_gtp* and *NoAction*. Fig. 2(b) and (c) show the definition of the encapsulation action. Basically, a novel header set (IP+UDP+GTP) is created in the original packet, resorting to the P4 extra header handling feature. In the action, the *setValid()* P4 command allows to create a new header that will be inserted in the last offset position pointed by the BMv2 deparser (i.e., old IP header). In the same action, all the novel headers fields are updated. In particular specific fields are updated as parameters configurable through flow entry: the GTP gateways as IP addresses at the IP layer and the TEID value at the GTP layer.

Decapsulation is performed as follows. In the case of a GTP-U encapsulated packet received by the switch, the pipeline switches the packet to the *table_decap_gtp* table, triggering the *decap_gtp* action, both shown in Fig. 2(d). The action sets the headers of the GTP stack as invalid, thus imposing their removal from the packet. Preliminary encap/decap latency chain performance results on the BMv2 soft switch are shown in Fig. 3(a) and (b) under constant bit rate traffic scenario, in terms of SDN scalability, i.e., number of flow entries (range 1-10k) and GTP-U offload impact (F+GTP) with respect to baseline forwarding (F) at different packet lengths (128 and 1200 byte). Results show that in the P4 switch working range the full encap/decap latency is around 200µs, practically constant with respect to the flow entry size and with limited impact with respect to basic forwarding operation.

IV. FULL UPF STEERING AND MONITORING

Besides tunnelling, the P4-based UPF includes traffic steering options for each possible functional interface pair (N3, N6, N9) targeting a flow-based design decoupled from the particularly considered network interface, in order to meet the independency between logical and physical interfaces, ready for slicing-based solutions. To allow all the steering options, forwarding actions (output port selection) have been included in the existing encap and decap tables, utilizing the same flow match policies used for GTP operation (i.e., IP addresses+ IP protocol match). This way, the match is independent from the input network interface and includes the N6-N9 and N6-N3 GTP sessions. The general design of the ingress pipeline is unchanged. However, as shown in Fig. 2(e), we modify the structure of table *table_decap_gtp*, defining two mutually exclusive actions: the existing *decap_gtp* action providing decapsulation and a novel *set_output_change_teid* action. The new action performs two operations: sets the forwarding output port (*standard_metadata.egress_spec*) and swaps the existing GTP TEID with the TEID value of the new GTP tunnel, provided as parameter through flow entry. This simple action implements the N3-N9 swap between two GTP tunnels.

The P4 switch has the capability to extract selected packet metadata conveying monitoring information, such as the packet timestamp, the packet hop latency and other switch

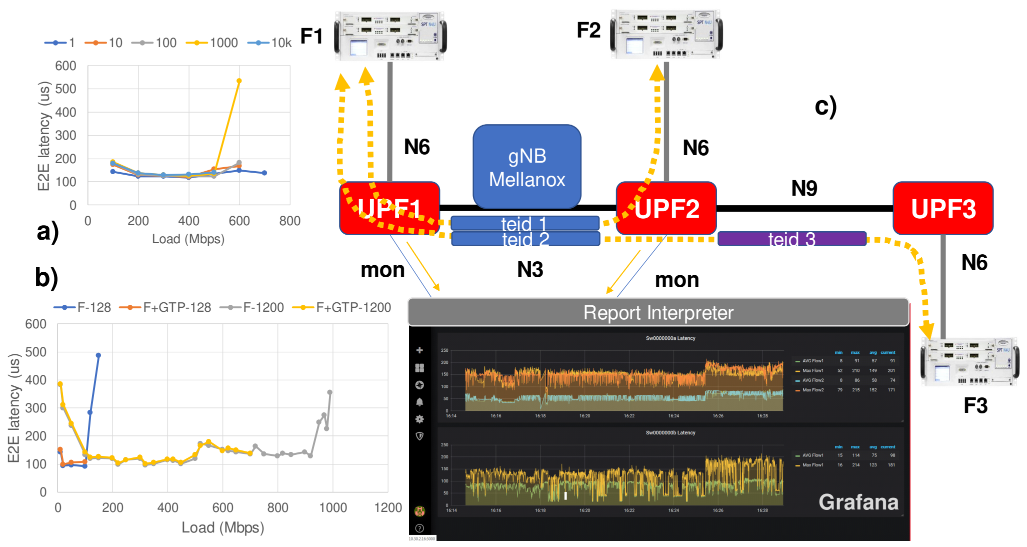


Fig. 3. P4 UPF: scalability (a), GTP impact versus baseline forwarding (b), testbed for the offloading demo (c).

parameter states. In this demo, we target the hop latency as a key monitoring parameter. The latency experienced at each UPF is a key metric for 5G infrastructures SLA. Thus, instead of using external generator/analyzers, we rely on extended P4 switches to extract latency directly. The implementation is based on *postcard-based telemetry*, a method used to extract features at each network node and provide results to a collector using a dedicated monitoring interface. The postcard-based telemetry has a simple implementation effort, since no complex in-band telemetry (INT) solutions are required.

The code selects the flows to be monitored through specific flow entry. Then, selected packets are mirrored to a monitoring interface. After mirroring, packets are transformed into Telemetry Report packets including the performance feature (e.g., the hop latency), see Fig. 2(e). Telemetry Report have been defined by the P4 consortium and the implementation is compliant with P4 INT specifications version 1.

V. P4-based Demo

The experimental demo showing the implemented P4 BMv2 switch employing UPF steering and monitoring functions is hereafter reported. The P4 code has been implemented, evaluated and validated over the BMv2 software switch [6], following the v1_model abstract model. The evaluation aims to validate the steering and provide the monitoring platform showing live monitored data (in particular, the live latency experienced at each switch by selected traffic flows). The validations are performed at the SSSA lab premises, utilizing BMv2 over bare metal Dell servers (Intel Xeon E5-2643 v3 6-core, 3.40GHz, 32GB RAM), connected by means of Gigabit Ethernet interfaces. Three UPF with the same implemented P4 code are deployed. Traffic flows are generated by the Spirent N4U. Bidirectional flows F1-F2 implement the N6-N3-N6 path, while F1-F3 follow the N6-N3-N9-N6 path. The topology includes all the UPF interfaces and directions (N3, N6, N9) to evaluate the steering and a Mellanox switch acts as a gNB transparent node. Monitoring interfaces are connected to a collector server running a python-based app. The app performs report packet dissection, timestamps extraction and hop latency computation. The hop latency is then associated with the *switch_id* and the *flow_id* of the report packet and stored in a InfluxDB database for Grafana GUI online visualization. The demo includes also a live evaluation of the scalability of the whole system, injecting a variable number of flow entries related to multiple GTP flows to evaluate the performance in terms of sustained throughput and transit latency. All the UPF steering directions have been validated and the Grafana shows the online hop latency for the selected flows (N3, N6, N9), successfully validating the P4 switch code. Such metadata info may be exploited by the SDN controller, in the case of excessive monitored values, to perform proactive GTP reconfigurations in order to meet SLA latency requirements. Note that each flow and interface may be easily reconfigured just resorting to online flow entry enforcement, without affecting the P4 code and the processing of other flows.

References

[1] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi, “P4 Edge Node enabling Stateful Traffic Engineering and Cyber Security,” *IEEE/OSA Journal of Optical Communications and Networking*, vol. 11, no. 1, pp. A84–A95, Jan. 2019.

[2] C. Shen, D. Lee, C. Ku, M. Lin, K. Lu, and S. Tan, “A programmable and fpga-accelerated gtp offloading engine for mobile edge computing in 5g networks,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops*, 2019, pp. 1021–1022.

[3] R. Ricart-Sanchez, P. Malagon, J. M. Alcaraz-Calero, and Q. Wang, “P4-netfpga-based network slicing solution for 5g mec architectures,” in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–2.

[4] C. Lee, K. Ebisawa, H. Kuwata, M. Kohno, and S. Matsushima, “Performance evaluation of gtp-u and srv6 stateless translation,” in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–6.

[5] F. Civerchia, M. Pelcat, L. Maggiani, K. Kondepu, P. Castoldi, and L. Valcarenghi, “Is opencl driven reconfigurable hardware suitable for virtualising 5g infrastructure?” *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 849–863, 2020.

[6] *BMv2: https://github.com/p4lang/behavioral-model.*