Pushing Services to the Edge Using a Stateful Programmable Dataplane

Angelo Tulumello, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Bianchi
CNIT/University of Rome Tor Vergata
Rome, Italy
{name.surname@uniroma2.it}

***Abstract*—Offloading to the edge a subset of the cloud services requested by users is a very appealing solution to reduce the bandwidth pressure, minimize latency and improve the overall quality of experience of mobile users. Achieving this goal is technically challenging, since the network functionalities needed to manage this offloading are not trivial, and present significant requirements in terms of scalability and speed. We propose to realize these network functionalities directly in the dataplane, exploiting the characteristic of FlowBlaze, a novel stateful programmable dataplane. Furthermore, we show how the network functions for the traffic offload can be easily expressed using an ad-hoc domain specific language developed for the description of per-flow stateful network functions.**

I. Introduction

The fifth generation of mobile networks is deemed to provide relevant advances and revolutions in the telecommunication sector both for end-users and carriers.

The envisioned 5G network aims at seamlessly supporting the widest range of services and applications ever witnessed in past wireless mobile networks. To realize such a vertical integrated vision of multiple end to end services deployed on the same mobile infrastructure, future 5G networks will embrace the vision of customer-facing on-demand network slicing. As described in the 5G-PPP architecture white paper [1] a network slice is nothing but a composition of adequately configured network functions, network applications, and the underlying cloud infrastructure (physical, virtual or even emulated resources, RAN resources etc.), that are bundled together to meet the requirements of a specific use case, e.g., bandwidth, latency, processing, and resiliency, coupled with a business purpose. To fulfill the continuously growing performance requirements, it appears necessary in given use cases to distribute application components at the network edge. Such concepts is known as Mobile Edge Computing [2]. Due to the dynamic requirements of network slicing, as well as the roaming nature of 5G mobile users, the logical network functions deployed in the MEC components need to be dynamically instantiated, interconnected, orchestrated, scaled, migrated and terminated. Thus, it appears clear why a programmable virtualized infrastructure is considered the most obvious supporting such an architectural vision.

Even if research in fast packet processing brought significant performance improvements in software environments, the stringent 5G performance requirements about latency, throughput and processing power reveal the need to offload some Network Functions to dedicated and powerful processing units. In this direction, the 5G-PPP consortium envisaged a logic block called Traffic Offloading Function (TOF) that permits the applications to re-route traffic from the core network to the services running into the MEC.

Unfortunately, the actual implementation of a TOF presents challenging design requirements that are hard to satisfy. In particular, an efficient TOF should provide high scalability (i.e. must be able to manage a significant number of concurrent connections) and low latency (one of the main reason for mobile edge cloud is to achieve ultra low latency, thus the overhead due to the traffic rerouting must be minimized). Furthermore, the TOF must implement complex and dynamic Network Functions like NAT, GTP-U encapsulation and de-encapsulation, and should be able to be adapted to different network scenario (e.g supporting different encapsulation types, or integrating the TOF with QOS policy or billing and accounting functionalities). Therefore, we propose to realize the TOF using programmable dataplanes, improving the performance and reducing the computational load of the virtualized network functions. In particular, this paper presents the design and implementation of a Traffic Offloading Function on top of FlowBlaze [3], a novel stateful programmable dataplane for fast packet processing. The dataplane is programmed by using an ad-hoc domain specific language, called XL (XFSM language), which is able to describe network functions in form of eXtended Finite State Machine (XFSM) [4], an abstraction suitable for the description of stateful functionalities.

The paper is organized as follows: section II provides a background of the stateful dataplane used to provide the Traffic Offloading Function and describes XL, the domain specific language used to program the stateful dataplane. Section III introduces the Mobile Edge Cloud scenario and gives the details of the Traffic Offload Function that has been realized. In section IV the actual implementation of the TOF using XL is presented, while conclusions are drawn in Section V.

II. Background

A. FlowBlaze

In this section we give a brief description of FlowBlaze [3], the stateful programmable dataplane that has been used to provide the TOF functionalities.

FlowBlaze leverages some well-known and widely accepted abstractions originally proposed for OpenFlow [5]. First of all,

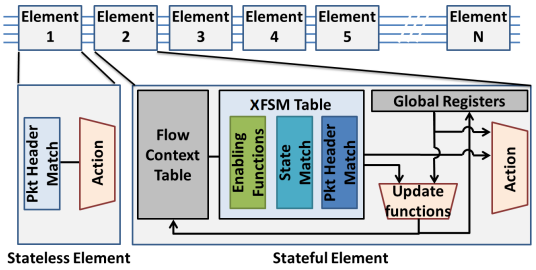

Fig. 1. FlowBlaze architecture

it is based on the Match Action Table (MAT) abstraction to define and select the rules that identify which actions must be executed to a packet. The match table is filled by a set of rules, which permit to classify the incoming packets, and the actions associated to the highest priority matching rule are executed. The MAT therefore acts as a set of *if-then* statements, where the rules corresponds to the *if* part, and the actions to the *then* part. Usually, a pipeline of MAT stages is used to further enhance programmability of an OpenFlow datapath. The packet headers (including packet metadata) are processed through the sequence of MAT stages and the relevant actions are applied to the packet while it is traversing the pipeline. Unfortunately, apart from the ability to maintain some counters (i.e. count how many times a rule is matched), the processing that can be executed by this datapath is inherently stateless, since there is no clear way to store inside the dataplane the history associated to the packets belonging to the same flow. The stateful processing is usually managed involving an external controller, which is in charge to update the MAT rules, in order to modify the datapath behavior. This approach greatly affect the latency of the state update and limits the scalability of this solution, due to the control link bottleneck. As stated in the introduction, this is a significant limitation when we want to deploy traffic offloading functionalities, which requires fast and scalable tunneling and binding operations. Instead, the proposed FlowBlaze [3] abstraction pushes directly in the dataplane flexible, but hardware efficient, stateful stages able to express the stage behaviour as an instead of a simple *if-then* rule in a more flexible eXtended Finite State Machine (XFSM) representation [4]. As a result, a pipeline can combine both stateless and stateful elements.

Fig. 1 presents a scheme of the FlowBlaze pipeline. A detailed description of the pipeline is available in [3]. Here we only present the main elements of a stateful stage: i) *Flow Context Table*, linking incoming packets to the corresponding set of state variables, ii) *XFSM Table* which evaluates the state transitions, iii) *Update Functions*, which update the state variables using arithmetic logic instructions, and iv) *Action*, which applies actions on the packet header.

B. XL

Since FlowBlaze is an abstraction which implements multiple XFSMs in hardware, the natural language for implementing our Network Functions within this abstraction is XL (XFSM Language). XL is a domain specific language, created to describe stateful and stateless network functions

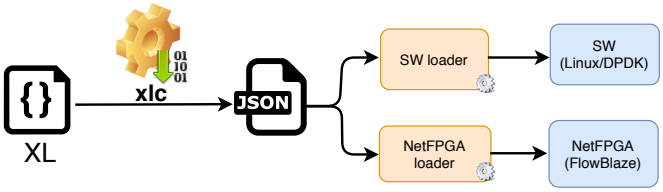

Fig. 2. XL development workflow

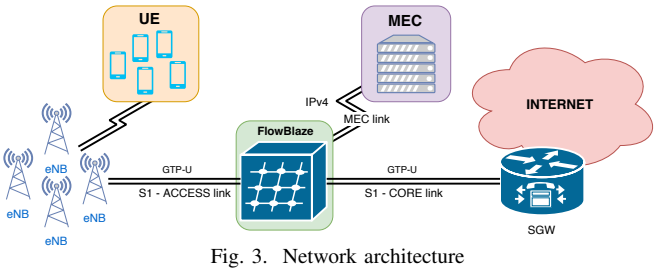

Fig. 3. Network architecture

using the XFSM model. The first advantage of using XL, is to have a *platform independent* portable code, which can be deployed on every platform capable of executing an XFSM (see Fig. 2). Moreover, another useful feature of XL is its clear concept of state, i.e. for every flow a *flow key* is defined, allowing to execute a set of lookup and update actions to the corresponding state variables. In order to write a Network Function, the developer has to write an XL program describing the behaviour of the function. This code can be compiled using *xlc* (i.e. XL compiler developed in JAVA using the ANTLR library[1]), into a JSON representation of the XFSM. This intermediate representation, platform independent, can be loaded into hardware or software based implementations, using a specific loader for each platform.

III. Traffic Offloading Function

A. Scenario

Our application is set into the context of a generic mobile metro network, in particular handles the traffic between User Equipement (UE) and Service Gateway (SGW). UEs are connected to the base stations (eNBs) through the Radio Access Network (RAN). User traffic is encapsulated in a tunnel between eNB and SGW, namely S1 bearer [6]. Our application resides in a middlebox placed among the S1 bearer. In particular, the FlowBlaze node will have (I) a link to the various eNBs, (II) a link to the SGW and (III) a link to a MEC node, e.g. an Internet Service Provider facility equipped with a server cluster. The objective of our FlowBlaze middlebox application will be to transparently forward user traffic to MEC, based on arbitrary policies to discriminate against specific services.

In Fig. 3 is depicted the network topology of the proposed application. The FlowBlaze node is placed in the middle and handles tunneled traffic on the interfaces connected to eNBs and SGW. The encapsulation technology taken into account in this work is the User GPRS Tunneling Protocol (GTP-U) [7], as it is one of the most relevant encapsulation protocols

[1]https://www.antlr.org/

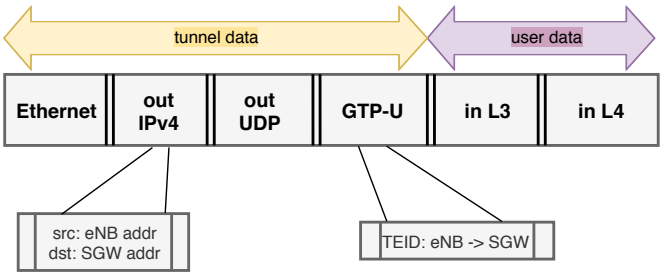

Fig. 4. Example of a GTP-U encapsulated packet.

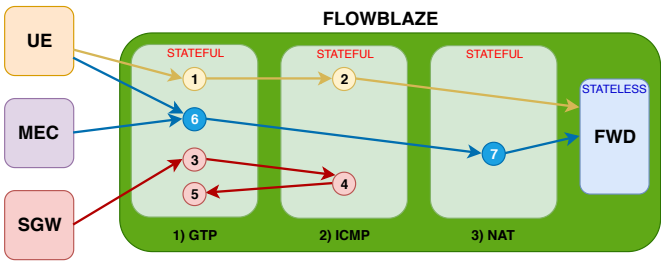

Fig. 5. TOF Pipeline.

used in deployed mobile backbones. A representation of a tunneled packet is depicted in Fig. 4. This example represents a segment carrying UE data in the direction eNB to SGW and it is structured as follows:

- the outer IPv4 header has as destination the SGW address and the eNB address as source and defines the connection between the two nodes;
- the outer UDP header has the special destination port (2152) which is used to identify that the next protocol would be GTP-U;
- the GTP-U header contains the 32 bit Tunnel Endpoint IDentifier (TEID) field that specifies the unique id for the tunnel in the direction eNB to SGW;
- the L3/L4 headers encapsulated after the GTP-U header represent the actual user data with public IPv4 address.

In the considered mobile network architecture, the tunnel identifier for the S1 bearer is different for the two directions since each direction is a distinct tunnel. In the link between FlowBlaze and MEC, all the traffic is without GTP-U encapsulation and carries only user data. The FlowBlaze application manages the de-encapsulation and the encapsulation with the correct outer headers needed to reconstruct the tunnels.

B. Design challenges

As stated before, an user equipment flow is identified by a couple of GTP-U TEIDs, one for each direction. This association allocates two tunnels in the S1 bearer (how the bearer is setup is out of the scope of this paper, so we assume that users has already a tunnel assigned). From the perspective of our FlowBlaze node, when an UE starts a new connection to the Internet, the TEID associated to the return tunnel is unknown. Since the objective of the application is to offload specific traffic to the MEC, we have to consider the condition in which an unknown user initiates a connection that has to be pushed to the edge. To retrieve the return TEID (from SGW to eNB), we need to get a packet belonging to the return path from which extract the return TEID. Since we do not want to introduce a control plane managing these issues, we choose to forge an ICMP echo request (ping) directly in the dataplane intended to a replying node after the Service Gateway. That node would then respond to the ICMP echo request, making FlowBlaze able to bind the outgoing TEID with the return one. Once the dataplane has the stateful association for the two tunnels, it can resolve the encapsulation headers needed to forward the packets correctly to the UE. The data needed to reconstruct the tunnel is stored in the FlowBlaze per-flow memory. This enables (1) the ability to NAT traffic directed to the services offloaded by the MEC and (2) reconstruct the return GTP-U header (and the outer encapsulating stack) to send the traffic back to the user. In the following section, we consider the implementation details of the three FlowBlaze stages required to implement the application:

1) extracts and identifies the GTP-U TEIDs for the outward traffic (eNB to SGW) and associates the backward TEID;
2) manages the ICMP echo requests and replies used to acquire the TEID couple; This stage is used only for the acquisition of the TEID and return TEID tunnels.
3) selects the registered offloading services and performs NAT and de-encapsulate/encapsulate operations.

In Fig.5 it is showed the pipeline containing the three stateful stages described above and the steps followed by packets traversing the stages.

C. Network Function Design

Echo Request Generation. The first stage classifies packets coming from UEs based on the GTP-U TEID field. Within the FlowBlaze architecture, this means selecting a *Flow Key* which is used to extract a *flow* depending on a packet field, in this case the TEID. Each *flow* has its memory context stored in the *Flow Context Table*. When an unknown packet is received from eNB link, FlowBlaze extracts the TEID and the flow's corresponding entry is allocated in the *Flow Context Table*. For this first packet of a new flow the state is *DEFAULT*. After the reception of the first packet, to retrieve the return TEID associated with the extracted flow, an ICMP echo request is forged and sent to the reply node (step 1 in Fig.5). The forged ICMP packet has the same outer headers of the incoming packet *(IPv4, UDP, GTP-U)*, while the inner data is the actual ICMP echo request intended to the reply node. The encapsulated packet is stored at FlowBlaze configuration time as a template in the *Flow Context Table*. The only field in the ICMP header that is modified is the *ICMP Identification field*, set according to a global variable that is incremented for each ICMP echo request sent. We use this field to identify the ICMP reply needed to bind the return tunnel to the right ingress TEID. The forged packet is parsed by the the second stage and the flow state is changed to *PENDING*.

The second stage has the *ICMP Identification field* as Flow Key, while the TEID is stored in a register of the corresponding *Flow Context Table* entry. Also the flow state for this second

stage is changed from *DEFAULT* to *PENDING*, and finally the packet is forwarded to SGW (step 2).

Regarding the ingress packet, there are three possible actions that can be applied when the state is *PENDING*: (1) the packet is dropped as eventual subsequent packets belonging to that flow; (2) the packet (and eventual subsequent packets) is stored in a per-flow buffer for future forwarding to MEC; (3) all the packets are forwarded to the SGW.

Echo Reply and Tunnel Setup. When the echo reply is received, it is first considered by the first FlowBlaze stage, which evaluates whether the packet is an ICMP echo reply by matching *type* and *code* fields (step 3). After a successful match the packet is passed to the second stage without any stateful processing. The second stage extracts the flow context associated with the *ICMP Identification field* (state is *PENDING*) and is able to bind the return TEID to the corresponding TEID coming from the UE. Now we need to push back this association in stage 1[2]. Thus, stage 2 modifies the packet's TEID field with the initial TEID previously stored in the flow register and saves the return TEID in the packet's metadata. Then, the packet is recirculated to stage 1 (step 4). In this way, the first stage can extract the TEID and store the return TEID in a flow context register, then the echo response is dropped (step 5). State is now *ESTABLISHED*: every new packet having that specific GTP-U TEID has in its flow context the matching return TEID identifying the backward tunnel associated to that flow.

NAT function. Stage 3 performs the Network Address Translation in order to forward de-encapsulated traffic to the services in the MEC. Services can be configured as a static set of match fields of the inner data packet, e.g. the combination of a destination IP address and a destination TCP port. Since distinguishing a service is an OpenFlow-like stateless operation, the traffic that may be offloaded may be processed by a stateless stage before the stateful stages which actually perform the Traffic Offloading Function. Therefore, only the selected traffic will be processed by stage 3, while other packets will simply be transparently forwarded to the SGW without being offloaded to MEC. We assume that an offloaded connection is initiated by an UE. Consequently, a packet matching a specific service is initially processed by stage 1 that matches the packet in *ESTABLISHED* state and puts in the metadata field the GTP-U return TEID and the outer IP addresses (step 6). Then the packet is de-encapsulated and passed to stage 3. The *Flow Key* of stage 3 extracts the 5-tuple *(IP destination address, IP source address, IP protocol, Transport destination port, Transport source port)*, a generally sufficient set of fields to differentiate Internet services. Afterwards, the metadata containing the return TEID is copied in a register as well as the outer IP addresses (later, this information will be used to reconstruct the encapsulating headers for the return path). The packet can now be forwarded to the MEC (step 7). In FlowBlaze forwarding can be carried out by a following stateless stage dedicated exclusively to that purpose, or by stage 3 itself, depending on the details of the network configuration taken into account. In the return path, traffic is plain IP without encapsulation. Stage 1 passes the packets straight to stage 3 that performs GTP-U encapsulation by setting outer IP addresses and GTP-U TEID fields with the three registers previously stored in memory.

[2]We could not do this binding when the reply packet has arrived at stage 1 since this stage is conceived to have as flow-key the TEID, not the ICMP Identification field

```
Stage gtp {
  Register global c;
  Register returnTEID;
  setFlowKey(gtp.teid);

  State initial default {
     if (pktRcvd.in_port == UE) {
       setField(in_icmp.id, FORGED, c);
       c += 1;
       setNextStage(icmp, FORGED);
       setNextState(pending);
       sendPkt(CURR, SGW);
     }
     if (pktRcvd.in_port == SGW) {
       sendPkt(CURR, UE);
     }
     if (pktRcvd.in_port == MEC) {
       setNextStage(nat);
     }
  }
  State pending {
     if (pktRcvd.metadata[0] != 0) {
       returnTEID = pktRcvd.metadata[1];
       setNextState(established);
     }
     if (in_icmp.type == 8, in_icmp.code == 0) {
       setNextStage(icmp, CURRENT);
     }
  }
  State established {
     if (pktRcvd.in_port == UE) {
       pktRcvd.metadata[0] = returnTEID;
       pktRcvd.metadata[1] = ip.dst;
       pktRcvd.metadata[2] = ip.src;
       decapGTP(CURRENT);
       setNextStage(nat, CURRENT);
     }
  }
}
```

Fig. 6. Stage 1.

```
Stage icmp {
 Register teid;
 setFlowKey(icmp.id);

 State initial default {
   teid = gtp.teid;
   setNextState(pending);
   sendPkt(CURRENT, SGW);
 }
 State pending {
   pktRcvd.metadata[0] = gtp.teid;
   setField(gtp.teid, CURRENT, teid);
   setNextStage(gtp);
   deleteInstance();
 }
}
```

Fig. 7. Stage 2.

IV. Application deployment using XL

In this section we analyze the deployment of the application using XL: the code of the three stages is depicted in Fig.6, Fig.7 and Fig.8.

First Stage: GTP. In the first stage we declare one global register, that is the counter used to increment the ICMP identification field, and one per-flow register devoted to store

```
Stage nat {
 Register ipsrc, ipdst, teid;
 setFlowKey(ip.src,ip.dst,ip.proto,tp.sport,tp.dport);
 State initial default {
   if (pktRcvd.in_port == UE) {
     teid = metadata[0];
     ipsrc = metadata[1];
     ipdst = metadata[2];
     sendPkt(CURRENT, MEC);
   }
   if (pktRcvd.in_port == MEC {
     encapGTP();
     setField(gtp.teid, CURRENT, teid);
     setField(ip.src, CURRENT, ipsrc);
     setField(ip.dst, CURRENT, ipdst);
     sendPkt(CURRENT, UE);
   }
 }
}
```

Fig. 8. Stage 3.

the return TEID for the backward tunnel. We set the *Flow Key* with the GTP-U TEID field. The first state *(default)*, has three *if clauses*, each representing three entries in which the match conditions evaluate respectively whether the input port was the one from UE, SGW or MEC. The first entry handles the generation of ICMP echo requests. To manage packet field modifications, we use the *setField()* primitive that has three parameters: the first selects the field to change in the packet; the second selects the packet to which the action should be applied; the last is the value of the field to be set. As introduced before, the forged packet is pre-allocated at configuration time and consists of a template representing an ICMP echo request GTP-U encapsulated and stored in a local buffer. Then the forged packet is passed to the *icmp* stage throguh the *setNextStage()* action and the state is set to *PENDING* through the *setNextState()* primitive. The second state processes the incoming ICMP echo reply. In the first entry it is evaluated whether the metadata is set or not to distinguish between the packet storing the return TEID and the initial ICMP packet for which the metadata should still be set. We store the return TEID in the appropriate register and change the state to *established*. The second entry simply matches an ICMP packet and passes it to the second stage. In the third state (*established*) the GTP-U roundrip TEIDs are finally bound. Every packet entering from the UE port and belonging to that flow will match the only entry present in that stage. The packet's metadata is filled with the header fields needed to reconstruct the encapsulating header: source and destination IP addresses and TEID. Then the packet is de-encapsulated and sent to the *nat* stage.

Second Stage: ICMP. This element processes ICMP packets received from the first stage. We declare the *teid* Register needed to store the return TEID and set the *Flow Key* to ICMP identification field. There are only two states to discriminate the echo request (*DEFAULT* state) with the echo reply (*PENDING* state). In the first there are no *if clauses* and the TEID is stored in the appropriate register, the state is changed to *PENDING* and the packet is sent to the SGW. In *PENDING* state the TEID of the received ICMP echo reply is stored in the metadata while the TEID field of the packet is replaced with the one stored in *teid* register. FInally, the packet is passed back to the *gtp* stage and the instance destroyed.

Third Stage: NAT. In the third and last stage we implement the NAT function to actually offload traffic to MEC. We define three registers *(ipsrc, ipdst, teid)* to store the packet fields needed to encapsulate the traffic from MEC to UE. We set *Flow Key* with the 5-tuple defined by the IP source and destination addresses, IP protocol and transport layer source and destination ports. The first entry processes the packets received from the UE port and updates the three registers with the proper values received as metadata from the *gtp* stage and forwards the packet to the MEC. The second entry handles traffic coming from MEC port. The plain IP traffic is encapsulated and the outer packet fields are set according to the three registers storing TEID and IP addresses. At last, the packet is sent to UE.

V. Conclusion and Future Work

This paper presented a Traffic Offloading Function implemented in FlowBlaze, a novel stateful programmable architecture for fast packet processing. We showed how a complex network function involving packet generation, tunnel bindings and encapsulation/de-encapsulation functions can be implemented entirely in the dataplane without the intervention of an external controller, thus reducing the possible controller bottlenecks and delays. The high level programming language and the FlowBlaze abstraction permit the deployment of advanced and stateful network functionalities in hardware nodes, thus enabling performance benefits with respect to software implementations of network functions. Moreover, the FlowBlaze pipeline model enables the ability to flexibly add other network functions just cascading multiple XFSM elements implementing, e.g. firewalls, QoS policy or billing and accounting functions.

Acknowledgment

This work is partially supported by the EU Commission in the frame of the H2020 project 5G-PICTURE (grant #762057).

References

[1] (2017) 5g-ppp architecture white paper. [Online]. Available: https://5g-ppp.eu/wp-content/uploads/2018/01/5G-PPP-5G-Architecture-White-Paper-Jan-2018-v2.0.pdf

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computinga key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[3] S. Pontarelli *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association*, 2019.

[4] V. S. Alagar and K. Periyasamy, *Extended Finite State Machine*. London: Springer London, 2011, pp. 105–128.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[6] (2016) Etsi ts 136 300. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/136300_136399/136300/13.02.00_60/ts_136300v130200p.pdf

[7] (2018) Etsi ts 129 281. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/129200_129299/129281/09.03.00_60/ts_129281v090300p.pdf