Programmable Low-End Networks: Powering Internet Connectivity for the Other Three Billion

André Scheibe, Willian Reichert, Luciano Gaspary, Weverton Cordeiro
Institute of Informatics (INF), Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
{first.last}@inf.ufrgs.br

***Abstract*—We propose the following exploratory research question: can we boost low-end networking through forwarding plane programmability? The implications of a positive answer to this question are manifold, unlocking low-end networks for innovative use cases, such as powering affordable internet access to remote communities and helping bridge the digital divide. As a first step towards answering this question, we introduce and discuss in this paper a conceptual architecture for programmable low-end networking. In summary, we take advantage of Low Power Wide Area Network (LPWAN) transceivers and use a reconfigurable pipeline of match+action stages to enable researchers and practitioners to write lightweight networking protocols that can seamlessly bring isolated communities to the Internet. We then exercise our architecture through the design and implementation of a Lightweight Tunnel Protocol, for optimized data communication over narrow-band links, with a throughput gain as high as 23%. Our results mainly provide evidence of programmable low-end networks' potentialities, helping us close the digital divide and bring affordable Internet for the other three billion.**

***Index Terms*—Low Power Wide Area Networks (LPWAN), Programmable Forwarding Planes, P4, Community Networks**

I. Introduction

Although Universal Internet Access was declared a fundamental human right [1], various challenges keep us far from providing affordable access for all, for example, high capital expenditure and service costs (e.g., satellite links for remote communities), operation complexity, and maintenance hurdles. Several public and private initiatives are attempting to mitigate these challenges and provide affordable connectivity solutions, including Internet.org [2], Loon [3], and A4AI [4]. There also various joint industry & academic initiatives, like IETF GAIA [5], as well as research contributions [6]–[8].

We argue in favor of *low-end networking* as a promising direction to help bridge the digital divide and provide affordable Internet access, mainly for remote communities with no telecommunication coverage (other than expensive satellite links). Low-end networking has been in the spotlight with the rapid growth of the Internet of Things (IoT) [9]. One key research challenge pursued in this context is how to provide connectivity to low power and low data rate devices used in several practical applications like asset tracking, smart cities and homes, and environmental monitoring [10]. In particular, Low Power Wide Area Networks (LPWAN) [11] are attracting significant attention, mostly because of their ability to make different trade-offs to address the diverse requirements of low-end networking (long-range, narrow band, low power, and high scalability), which are not met by prevalent (and legacy) wireless technologies and cellular networks [9], [12].

Several factors make LPWAN a promising technology to help connect remote communities and bridge the digital divide, with economic factors as the most prevalent. Because of the strict operational requirements of a majority of IoT devices and sensors (e.g., preserve battery life and communicate data over long distances), LPWAN mainly focuses on simplicity. Consequently, acquisition and operational costs (e.g., hardware and connectivity) are kept as low as possible. In contrast, satellite links and other wired/wireless infrastructures (e.g., optical fiber, directional antennas, or cellular networks) are often prohibitive because of their high acquisition, deployment, and operational costs, making them economically unfeasible for smaller and lower-income communities in remote areas [13].

Despite the potentialities, LPWAN integration with traditional networks is done in an *ad hoc* fashion, thus hampering widespread adoption for last-mile data communication in remote communities [14]. One integration approach relies on custom-tailored protocol gateways, which often depend on or provide closed and vendor-specific interfaces. IETF has a currently active WG (lpwan) that focuses on enabling IPv6 connectivity over selected LPWAN technologies.

We have seen in the past decade the advent of Software Defined Networks (SDN) [15], [16] and Programmable Data Planes (PDP) [17], [18]. Together, they enabled for the first time in decades a rapid evolution in the way we think and design computer networks, from closed-source and proprietary solutions to open management standards, and forwarding elements whose behavior can be freely redefined using Domain-Specific Languages (DSLs) like P4 [18] and Lyra [19]. SDN and PDP enabled researchers and practitioners to quickly introduce and test new ideas into networking, without depending on device vendors to implement them on their products. As a result, various fundamental networking questions have been asked, mostly targeting high-end data communication, such as data center and backbone networking [20].

In this paper, we take a step in the opposite direction of the status-quo of programmable networking research on high-end solutions by posing a radically different research question: *can we leverage the concept of programmable forwarding planes to boost low-end networking?* Note that a positive answer to this question will foster community networking research [8] by providing them access to a multitude of low-cost hardware whose behavior could be redefined using open and interoperable standards. More importantly, it may also redefine the IoT landscape by enabling the delivery of innovative protocols and services through open and highly programmable LPWANs for an increasingly interconnected world of things.

To answer the research question above, we propose in this paper a conceptual architecture for programmable low-end networking and use it in an exercise the design and implementation of a novel protocol for low-end networks. Our architecture introduces ideas for the design of virtually programmable LPWAN devices while promotes backward compatibility and enables the usage of LPWAN in the context of forwarding plane programmability. The protocol we designed using our architecture, Lightweight Tunnel Protocol (LTP), maximizes goodput on low-end networking devices with limited baud rate, performing well to provide Internet connectivity in coexistence with traditional TCP/IP protocols, with throughput gains as high as 23%. More importantly, our exercise in the design and implementation of LTP provides evidence of the feasibility of redefining packet parsing and processing semantics of legacy LPWAN devices to be more easily integrated with traditional networks without requiring specialized devices like gateways. In summary, we make the following contributions:

- A conceptual architecture for legacy LPWAN programmability, to enable researchers and practitioners to redefine the behavior of existing LPWAN devices and seamlessly integrate them with traditional networks;
- A Lightweight Tunnel Protocol (LTP), whose implementation exercises the concept of LPWAN programmability. LTP supports data communication over narrow-band links that is compatible with existing networking protocols and provides comparatively smaller overhead;
- An open-source implementation of LTP on GitHub [21].

The remainder of the paper is organized as follows. We cover background and related work in Sec. II. In Sec. III, we introduce our conceptual architecture for programmable low-end networking. In Sec. IV, we exercise our architecture with the design and implementation of a novel Lightweight Tunnel Protocol (LTP), suitable for networking using LPWAN devices, whereas in Sec. V we provide an extensive evaluation of our protocol. Finally, we close the paper in Sec. VI.

II. Background and Related Work

There are several technologies already suitable for community networking, like Cognitive Radio (CR) and TV white spaces (TVWS). CR is an adaptive radio that uses dynamic spectrum access techniques to reconfigure itself by detecting and using available wireless frequencies in its vicinity to avoid interference and enable concurrent communications [22]. TVWS, in turn, refers to the unused TV channels for both VHF and UHF spectrum. In the past, unused channels were placed between active ones to protect from broadcasting interference. It has since been shown that they can provide broadband communication links (and Internet access) while operating without affecting TV channels. The use of lower-frequency UHF signals also enables penetrating obstacles and covering uneven ground without additional infrastructure. For these reasons, developing and rural regions have been a key use case for TVWS [23]. Despite their potentialities, CR and TVWS often require licensing for operation and equipment that may not be affordable for some communities.

We posit that LPWAN may also become an exciting technology for deploying community networks, either substituting or complementing CR and TVWS, mostly because of its low cost and easy deployment. There have been investigations on using Software Defined Networking (SDN) for managing LPWAN, including flow forwarding management [24], application-aware service provisioning [25], and integration of heterogeneous networks [26]. In general, they are concerned with providing an open interface for managing the resource-constrained devices that compose the network. However, to the best of our knowledge, bringing forwarding plane programmability to LPWAN has not been approached in the literature yet.

The intrinsic nature of LPWAN makes it difficult to reshape them for forwarding plane programmability. Their design targets data communication over large geographical areas, often with poor logistical infrastructure, lack of electrification, challenging topography, etc., frequently trading-off complexity (and cost) for limited data rate [9]. In contrast, a design containing silicon for reconfigurable match-action tables would increase the cost of LPWAN devices substantially, likely making them economically prohibitive for their traditionally intended use. Nevertheless, there has been research on forwarding plane programmability in the context of LPWAN, e.g. for packet aggregation/using disaggregation in IoT [27].

As we bridge the gap towards programmable low-end devices, a remaining challenge for community networks will become the optimal usage of narrow-band communication channels for Internet traffic. There are several solutions that provide network traffic optimization through header compression [28]–[34], header field removal [35], coding [36], and packet aggregation [37]. Header compression solutions often rely on the RObust Header Compression (ROHC) Framework [38], which takes advantage of the fact that most data in packet header fields remain static during a flow (e.g., source and destination addresses, and most significant digits of packet identification numbers). ROHC may not be feasible for usage in scenarios with frequent packet loss, which is the case of wireless communication over long distances typical of community networks. Nevertheless, as we advance on programmable low-end networking, lessons learned with the design and implementation of such protocols will certainly benefit the emergence of a novel generation of community networking protocols designed and implemented for LPWAN.

III. Programmable Low End Networking

In this paper, we explore the possibility of using DSLs for programming the data plane [18], [19] to express the networking behavior of low-end networking devices. Our main goal is to use data plane programmability (PDP) to

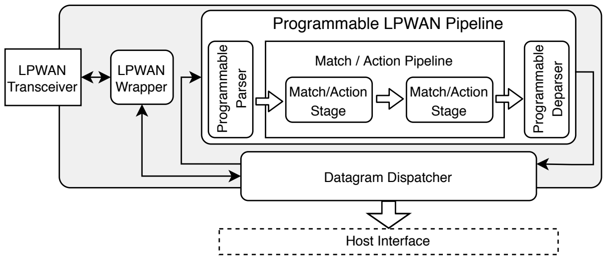


Fig. 1: Conceptual Programmable LPWAN architecture.

devise network protocols that can operate with devices with limited data transfer capabilities and under hostile networking conditions. A second but equally important goal is enabling network developers and practitioners to take advantage of PDP with existing (legacy) LPWAN devices, whose packet parsing and processing semantics are predefined, often closed, and cannot be changed with firmware upgrades.

To achieve these goals, we introduce a conceptual architecture for programming low-end devices. Fig. 1 provides an overview of our architecture, depicting its main components and their relationship. Our architectural design is inspired by off-the-shelf Programmable SmartNICs, simplified to a trade-off between robustness and implementation complexity/cost.

Our architecture includes an *LPWAN transceiver* (e.g., LoRa SX1276/SX1278 [39]), that sends/receives datagrams over the air and whose behavior we assume that cannot be redefined through firmware upgrades. To use such transceivers in a programmable environment, we adopt a *LPWAN Wrapper* component. The wrapper (a driver) interacts with the transceiver to read/write data, following the transceiver's specification. The wrapper also serves as an entry point for programming and configuring the transceiver, e.g., its baud rate, modulation scheme, TX power, RX sensitivity, etc., and can be developed using the vendor-provided SDK. An example is Ronoth LoStick [40], a USB LoRa Stick that provides an open-source interface that can be programmed using Python.

The *Datagram Dispatcher* is responsible for packet exchange between the *LPWAN Interface*, the host (through the *Host Interface*, which could be PCIe, USB, Ethernet, etc.), and the *Programmable LPWAN Pipeline*. The pipeline is inspired by the Very Simple Switch (VSS) model [41]. With such architecture, a network developer can then write a P4 program that redefines the packet parsing and processing semantics of the LPWAN pipeline and use a P4 compiler to generate the program/API that will define the device behavior.

In this paper, we take advantage of the conceptual architecture above to design and implement a networking protocol for low-end devices. Our instance of the conceptual architecture uses Ronoth LoStick [40] as LPWAN Transceiver, a custom-made Python program that interacts with LoStick to program it and read/write data, and the software switch bmv2 for emulating a Programmable LPWAN Pipeline. The following section provides an in-depth discussion of the design and implementation of our low-end networking protocol.

IV. Lightweight Tunnel Protocol

As an exercise for programmable low-end networking, we propose a lightweight protocol for data transfer over an unreliable wireless link, which we call Lightweight Tunnel Protocol (LTP) for Internet over LPWAN. The goal of LTP is to decrease the overhead in data communication between hosts over narrow-band links, by replacing L2/L3 headers with a protocol that uses tagging/labels to identify packets exchanged between specific pairs of hosts. Next, we discuss the design, implementation, and lessons learned with LTP. Later in Sec. V we assess the potentialities of using LPWAN devices for data communication over the Internet, by studying the technical feasibility of our implementation of LTP using P4 [18].

A. Design Considerations

In our exercise, we intended to devise a protocol that reduces the communication overhead due to link and network layer packet headers. Instead of sending Ethernet and IP headers for every packet of a TCP/IP flow, we send a smaller header containing only a tag identifier, which translates to data that devices can use to rebuild the original headers.

The protocol should also rely on the notion of virtual tunnels between pairs of devices. For this reason, its packets focus on single-hop communication, just like Ethernet. Finally, it should support any upper-layer protocols. To keep it simple, we assume that features like packet retransmission and checksuming are already handled by these protocols. In this context, LTP is akin to any L2 protocol, providing data communication over a virtual channel between pairs of devices.

B. LTP Overview

Without loss of generality, we describe LTP using an illustrative scenario two groups of users, each connected to a switch, as shown in Fig. 2. The uplink port of both switches is connected to a programmable LPWAN device, which the switches use to exchange data between them. We assume a fully distributed scenario; each programmable LPWAN device has its SDN controller, and SDN controllers cannot directly communicate with each other.

LTP enables data exchange over a narrow band wireless link, with minimal protocol overhead. To this end, we create tag identifiers for each pair of hosts that need to communicate over the air. These tags are single-hop only. Fig. 2 provides an overview of communication between Hosts A and B. Host A sends a packet to B using TCP/IP, through Programmable LPWAN Device (device) 1 (`d1`). Upon receiving the packet, `d1` looks up for a set of rules in its match+action tables that translate the IP addresses of A and B into a tag that univocally identifies the pair. In case such a tag does exist, `d1` strips the Ethernet and IP headers, builds a tag header (which will now contain the packet payload), and sends it over the air using the LPWAN antenna. After receiving the LTP packet, device 2 (`d2`) looks up the tag. In case `d2` does find a corresponding pair of IP addresses for A and B in its match+action tables, it strips the LTP header, reconstructs the Ethernet and IP headers, and sends the rebuilt TCP/IP packet to the target host B.

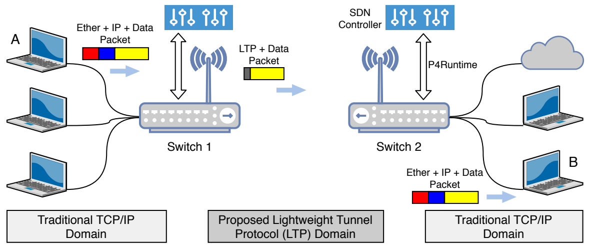


Fig. 2: Scenario using programmable LPWAN devices and a Lightweight Tunnel Protocol (LTP) for wireless networking.

LTP Packet Type	Device Id Number	Tag Id Number	Next Header Type

Fig. 3: LTP packet header.

C. LTP Packet Header Format

Fig. 3 shows LTP packet header format. A challenge we faced in the design of the packet parser for such protocol was devising a strategy to disambiguate standard TCP/IP packets from LTP ones. To make the parser (and protocol) agnostic of any environment particularities (e.g., port receiving LTP packets), we adopted in our packet header a special field `LTP Packet Type`. It will assume a known value that the parser can look ahead (using P4 `lookahead()`) to distinguish LTP from Ethernet frames. It is thus imperative that values for this field do not collide with valid MAC address prefixes [42].

Since we are dealing with data communication over the air, multiple programmable LPWAN devices may be initiating LTP tunnels between pairs of hosts at any instant. We thus need to ensure that created tag identifiers do not collide. In other words, we need to avoid that two devices, one initiating a tunnel for hosts A and B, and another initiating a tunnel for C and D, at the same instant, do not use the same identifier for both tunnels. We address this issue by including in the LTP header a `Device Id Number` field, which contains the identifier of the device initiating the tunnel. In this case, each device can locally decide on a tag identifier when creating a tunnel without risking using an already assigned tag identifier. Note that our design assumes that devices have a unique identifier in the network (defined *a priori*). We complete the LTP header with the `Tag Id Number` field, which univocally represents a pair of hosts A and B, and `Next Header Type` field, which identifies the next protocol header.

D. LTP Handshake

In case hosts A and B have not yet exchanged data, `d1` and `d2` will need to create a tunnel before any packet exchange between them can occur. Fig. 4 provides an overview of the handshake process for establishing a communication tunnel for A and B. Upon receiving a TCP/IP packet from Host A, `d1` generates a tag identifier Y and builds an LTP SYN packet, which includes its device identifier X. In addition to the tag identifier, this packet will contain the IP header as well as upper layer headers and data as payload. Device`d1` also

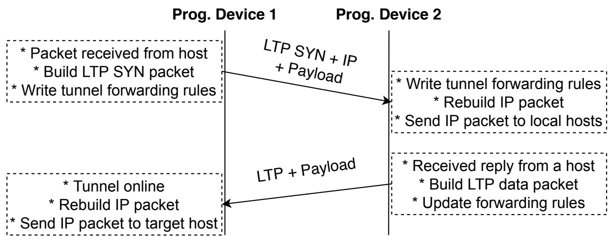

Prog. Device 1
Prog. Device 2
* Packet received from host
* Build LTP SYN packet
* Write tunnel forwarding rules
LTP SYN + IP + Payload
* Write tunnel forwarding rules
* Rebuild IP packet
* Send IP packet to local hosts
* Received reply from a host
* Build LTP data packet
* Update forwarding rules
LTP + Payload
* Tunnel online
* Rebuild IP packet
* Send IP packet to target host

Fig. 4: Lightweight Tunnel Protocol (LTP) handshake.

writes in its forwarding tables the set of rules to rebuild the original TCP/IP packet given the tag identifier extracted from received LTP packets. Upon receiving the LTP SYN packet with encapsulated IP header and payload, `d2` extracts from the LTP header the device identifier X (of the device that originated the LTP SYN packet) and the tag identifier Y. It also extracts from the encapsulated IP header the IP addresses of A and B. The device then matches the extracted information against its forwarding tables, sending to the controller for rule writing/update if necessary. The device then rebuilds the TCP/IP packet and sends it to host B.

In the event that B replies to the IP packet, `d2` will extract the IP addresses and look them up in its forwarding table for the corresponding tag identifier. The device will then build an LTP data packet and send it over the air. Finally, when `d1` receives the LTP data packet, it will extract the tag identifier, look it up in its forwarding table for the IP addresses, and rebuild the IP packet and send it to the target host A. At this point, both devices will see the tunnel as online.

E. LTP Implementation and State Machine

We depict in Fig. 5 an excerpt of the P4 code related to the implementation of LTP[1]. In Fig. 6 we depict a finite state machine (implemented in the excerpt code shown) designed for establishing tunnels under LTP, considering the scenario shown in Fig. 2, and highlighting the roles of the source and destination devices involved in the tunnel. Note that we maintain a state machine for each tunnel independently. Conceptually, any tunnel starts in the `TUNNEL_CLOSED` state.

Each device maintains two forwarding tables. The first is `ip_exact` table (lines 27-39 in Fig. 5), that matches the src and dst addresses of IP packets, and whose main action is `ltp_hdr_build()`. This action takes as input a packet with Ethernet and IP headers and builds an LTP header for the packet. The second is `ltp_exact` (lines 41-52), which matches the device and tag identifiers of the tunnel, and whose main action is `ip_hdr_build()`. The action takes as input an LTP packet with payload and rebuilds the Ethernet and IP headers. The devices also maintain a `dmac` table (lines 54-65), used for MAC learning and IP to MAC address translation.

1) Device 1 (Source): The tunnel negotiation for a pair of hosts A and B initiates when A sends to the source device (`d1`) a TCP/IP packet intended to B. The device checks that the

[1]Kindly note that the code shown in the paper differs in part from the actual code in our repository, for the sake of legibility and space constraints.

```
1   control MyIngress(inout headers hdr, inout metadata meta,
2                inout standard_metadata_t std_meta) {
3      action drop() {
4         // omitted for brevity
5      }
6
7      action ltp_hdr_build(devAddr_t devId, tagAddr_t tagId, egressSpec_t port) {
8         // omitted for brevity
9      }
10
11     action send_to_cpu() {
12        // omitted for brevity
13     }
14
15     action ip_hdr_build(ipAddr_t ipSrc, ipAddr_t ipDst) {
16        // omitted for brevity
17     }
18
19     action set_dmac(macAddr_t macDst, egressSpec_t sw_port, macAddr_t macSrc) {
20        // omitted for brevity
21     }
22
23     action set_dmac_bcast() {
24        // omitted for brevity
25     }
26
27     table ip_exact {
28        key = {
29           hdr.ip.srcAddr: exact;
30           hdr.ip.dstAddr: exact;
31        }
32        actions = {
33           ltp_hdr_build;
34           send_to_cpu;
35           drop;
36           NoAction;
37        }
38        default_action = send_to_cpu();
39     }
40
41     table ltp_exact {
42        key = {
43           hdr.ltp.devId: exact;
44           hdr.ltp.tagId: exact;
45        }
46        actions = {
47           ip_hdr_build;
48           drop;
49           NoAction;
50        }
51        default_action = drop();
52     }
53
54     table dmac {
55        key = {
56           hdr.ip.dstAddr: exact;
57        }
58        actions = {
59           set_dmac;
60           set_dmac_bcast;
61           drop;
62           NoAction;
63        }
64        default_action = set_dmac_bcast();
65     }
66
67
68
69     apply {
70        tagReg_t reg;
71        bit<8> current_state;
72
73        if (!hdr.ltp.isValid() && hdr.ip.isValid()) {
74           ip_exact.apply();
75           reg = ((devAddr_t)hdr.ltp.devId) << DEV_ID_SZ | (tagAddr_t)hdr.ltp.tagId;
76
77           if (std_meta.egress_spec == CPU_PORT) {
78              // do nothing. send_to_cpu() already triggered
79
80           } else {
81              tunnel_state_machine.read(current_state, (bit<32>)reg);
82              if (current_state == TUN_ST_PNDG) {
83                 hdr.ltp.type = LTP_TYPE_SYN;
84                 current_state = TUN_ST_SENT;
85                 tunnel_state_machine.write(reg, current_state);
86
87              } else if (current_state == TUN_ST_SENT) {
88                 hdr.ltp.type = LTP_TYPE_SYN;
89
90              } else if (current_state == TUN_ST_RECV) {
91                 current_state = TUN_ST_ACKD;
92                 tunnel_state_machine.write(reg, current_state);
93
94              } else if (current_state == TUN_ST_ACKD) {
95                 hdr.ip.setInvalid();
96              }
97           }
98        } else if (hdr.ltp.isValid()) {
99           ltp_exact.apply();
100
101          reg = ((devAddr_t)hdr.ltp.devId) << DEV_ID_SZ | (tagAddr_t)hdr.ltp.tagId;
102          tunnel_state_machine.read(current_state, reg);
103
104          if (std_meta.egress_spec == DROP_PORT) {
105             if (hdr.ltp.type == LTP_TYPE_SYN) {
106                send_to_cpu();
107             } else {
108                current_state = TUN_ST_DROP;
109                tunnel_state_machine.write(reg, current_state);
110             }
111          } else {
112             if (hdr.ltp.type == LTP_TYPE_SYN) {
113                current_state = TUN_ST_RECV;
114                tunnel_state_machine.write(reg, current_state);
115
116             } else (hdr.ltp.type == LTP_TYPE_DATA) {
117                if (current_state == TUN_ST_RECV) {
118                   current_state = TUN_ST_DROP;
119                   tunnel_state_machine.write(reg, current_state);
120
121                } else if (current_state == TUN_ST_SENT) {
122                   current_state = TUN_ST_ACKD;
123                   tunnel_state_machine.write(reg, current_state);
124                }
125             }
126             if (current_state == TUN_ST_DROP) {
127                drop();
128
129             } else {
130                dmac.apply();
131                hdr.ltp.setInvalid();
132             }
133          }
134       }
135    }
136 }
```

Fig. 5: Excerpt of the P4 code for the Lightweight Tunnel Protocol (LTP).

received packet is an ordinary TCP/IP packet (line 73) and tries to look up the src and dst IP addresses to find a tag identifier for this pair (line 74). In the case of a table miss, the default action (line 38) is sending the packet to the controller. It also marks the status of the tunnel as `TUN_ST_PDNG` (transition S.1 in Fig. 6, and line 75 in Fig. 5).

The controller keeps the id number of the device connected to it (X) and a list of active tunnels, which includes the tag identifier of the tunnel and pairs of endpoint hosts. Upon receiving a packet from the device (`PacketIn` event in the SDN literature), the controller opens it, extracts the src and dst IP addresses of A and B, assigns a tag identification number Y for them, and writes in the device tables the forwarding rules for the tunnel. More specifically, it writes the following rules:

- In the `ip_exact` table, a rule to match the IP of A and B and trigger action `ltp_hdr_build()`. The action receives as parameters the device id X, the created tag id Y, and the port where the LPWAN antenna is connected;
- In the `ltp_exact` table, a rule to match the device and tag identifiers X and Y and trigger action `ip_hdr_build()`. The action receives as parameters the IP addresses of hosts A and B.

The controller also takes the opportunity to learn the MAC address of the source host A, in case it is not yet present in the `dmac` table (defined in lines 54-65). The controller does so by writing a forwarding rule that triggers action `set_dmac()` in case a table entry matches the IP address of A.

After writing the rules above, the controller sends the packet back to `d1` (via `PacketOut` event). The device again applies the `ip_exact` table and, since it now has a table match (because of the rules that the controller just wrote), the device triggers action `ltp_hdr_build()` to build an LTP header for the packet. The device discards the original Ethernet and IP headers and sends the packet to the port where

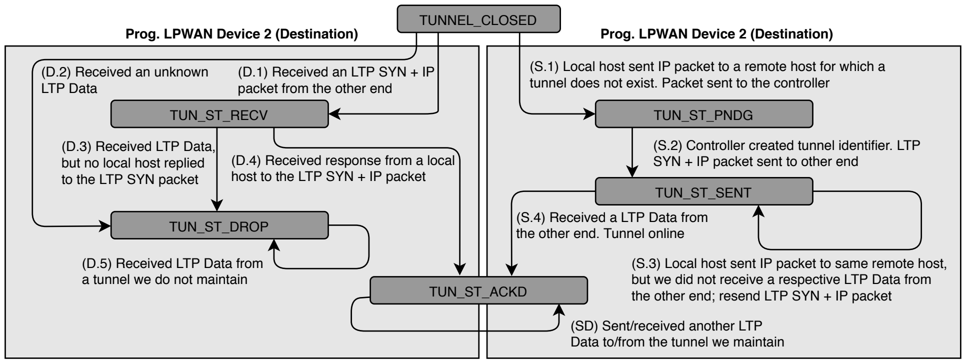

TUNNEL_CLOSED
Prog. LPWAN Device 2 (Destination)
Prog. LPWAN Device 2 (Destination)
(D.2) Received an unknown LTP Data
(D.1) Received an LTP SYN + IP packet from the other end
TUN_ST_RECV
(D.3) Received LTP Data, but no local host replied to the LTP SYN packet
(D.4) Received response from a local host to the LTP SYN + IP packet
TUN_ST_DROP
(D.5) Received LTP Data from a tunnel we do not maintain
TUN_ST_ACKD
(SD) Sent/received another LTP Data to/from the tunnel we maintain
(S.1) Local host sent IP packet to a remote host for which a tunnel does not exist. Packet sent to the controller
TUN_ST_PNDG
(S.2) Controller created tunnel identifier. LTP SYN + IP packet sent to other end
TUN_ST_SENT
(S.4) Received a LTP Data from the other end. Tunnel online
(S.3) Local host sent IP packet to same remote host, but we did not receive a respective LTP Data from the other end; resend LTP SYN + IP packet

Fig. 6: State machine for tunnel negotiation under LTP.

the LPWAN antenna is connected. The device also evolves the state machine for this tunnel to state `TUN_ST_SENT` (transition S.2 in Fig. 6 and lines 82-85 in Fig. 5).

LTP does not aim for packet retransmission. In this context, `d1` may receive another ordinary TCP/IP packet – a retransmission or a novel packet from another application – before the tunnel is active, i.e., before receiving a tunnel ACK packet. Device `d1` sends the received packet using LTP SYN again as packet type (transition S.3 and lines 87-88), remaining in state `TUN_ST_SENT`. This choice ensures that LTP behaves as an L2/L3 protocol and that any upper-layer protocol remains free to deal with lost packets and expired timeouts.

Upon receiving an LTP data packet from the other end, the device applies the `ltp_exact` table (lines 98-99). Since the device already has the forwarding rules for the tunnel, the packet will trigger action `ip_hdr_build()`, to rebuild the Ethernet and IP headers. The LTP header is discarded. The state machine for this tunnel then evolves to state `TUN_ST_ACKD` (transition S.4 and lines 121-124). For subsequent LTP packets sent/received, the tunnel remains in the `TUN_ST_ACKD` state (transition SD).

2) Device 2 (Destination): The tunnel negotiation in the destination device (`d2`) starts when it receives an LTP SYN from the source end (lines 98-99). If it is the first LTP SYN received, `d2` sends it to the controller, because of the `ltp_exact` table miss (transition D.1 and lines 104-106).

The controller then receives the packet from `d2`, extracts the tag id Y from the LTP SYN header and the IP addresses of A and B from the IP header, and writes in the `d2` `ltp_exact` table a forwarding rule to rebuild IP headers from subsequent LTP Data packets (that will not contain an IP header). The controller does not write the rules to build LTP packets yet since there is no confirmation that B is connected to `d2`. The controller then sends the packet back to the device.

The device receives the packet from the controller, which now matches an entry in the `ltp_exact` table and triggers `ip_hdr_build()`, to rebuild the original TCP/IP packet. At this point, the device evolves to state `TUN_ST_RECV` (transition D.1 and lines 112-114). The device applies the `dmac` table to populate the destination MAC address of host B in the packet. In case its MAC address is not yet known, the packet is sent in broadcast (default action, line 64).

Upon receiving a TCP/IP packet reply from host B, the device applies the `ip_exact` table to build the LTP Data packet. Note that, as the tunnel is in `TUN_ST_RECV` state, `d2` does not have the rules to build LTP packets from original TCP/IP packets yet. For this reason, the packet triggers a table miss, and the device sends it to the controller. The controller then writes the `ip_exact` rule that triggers action `tlp_hdr_build()`. The controller also learns the MAC address of B, if not yet recorded in the `dmac` table. Finally, the controller sends the TCP/IP packet back to `d2`.

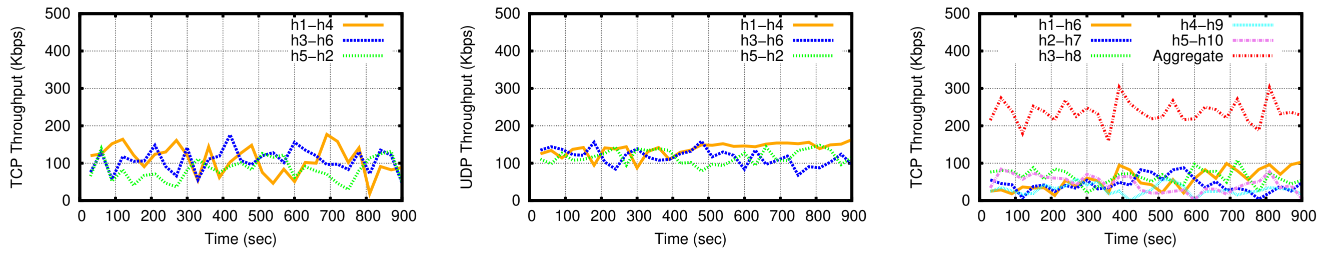
The TCP/IP packet received from the controller now triggers the `tlp_hdr_build()` action, for which the device builds the TLP Data packet to send to the other end of the tunnel. The tunnel transitions to state `TUN_ST_ACKD` (transition D.4 and lines 90-92). At this point, the destination device considers the tunnel online. Note that the tunnel must yet be acknowledged as online by the other end, which may not occur immediately because of packet loss. Such an event does not impact the destination device, which still can see the tunnel as online.

There are two particular cases.The first is when a device receives an LTP Data packet without having seen any prior LTP SYN. In this case, it evolves the state machine for this tunnel to `TUN_ST_DROP` (transition D.2). Further packets from this tunnel are dropped. The second is when the device sees an LTP Data packet for a tunnel to which its state machine is still in `TUN_ST_RECV` state. It means that the target host was connected to some other device (note that devices send LTP packets over the air and that there are multiple devices in the network). In this case, the device also evolves the state machine for this tunnel to `TUN_ST_DROP` (transition D.3).

Finally, note that, in any state, a tunnel may expire because no packets were seen after a given time period. In this case, the state machine for that tunnel evolves to state `TUNNEL_CLOSED`. The controller may implement a clean-up routine to remove entries of stalled tunnels.

V. Evaluation

We assessed the technical feasibility of our architecture for programmable low-end networks and evaluated the per-



(a) Star topology, TCP (b) Star topology, UDP (c) Single shared medium

Fig. 7: LTP feasibility in scenarios with limited link capacity and competing flows.

formance of our proof-of-concept implementation of LTP. We concentrated on the following research questions: (*i*) Can we effectively use LPWAN devices along with LTP for last-mile Internet access? (*ii*) What is the achievable throughput and overhead imposed to communication over such narrow-band links? (*iii*) What is the overall reduction in overhead compared to TCP/IP? And (*iv*) What is the overall performance for a standard file retrieval using HTTP for data transfer?

To answer these questions, we wrote LTP using P4_16, and used mininet and bmv2 switch for experiments. We wrote the controller in Python2.7, using scapy [43] for packet parsing. Our source code and results are on GitHub [21]. We used iperf3 for generating flows. We have a hardware prototype on assembly, based on a Raspberry Pi 3 and LoRa for LPWAN, with two antennas: LoStik USB LoRa device by Ronoth [40] and a LoRa/LoRaWAN Raspberry Pi SX127X HAT module by Dragino [39]. For these reasons, we used a 300 kbps narrow-band link, achievable with the SX127X HAT module.

A. LTP Feasibility

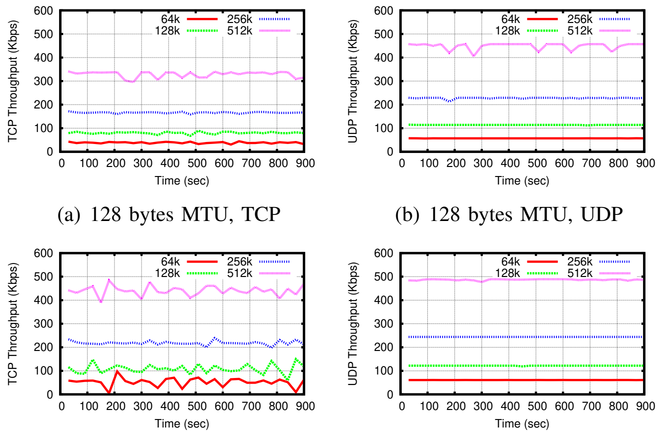
Here we concentrate on our the feasibility of LTP for last-mile communication. To this end, we considered two scenarios. The first one has 3 devices `d1..d3` and 6 hosts `h1..h6`, connected as follows: `h1..h2` to `d1`, `h3..h4` to `d2`, and `h5..h6` to `d3`. The devices are connected through a 256kbps link to a hub, in a star topology. The hub emulates the shared wireless communication. The second scenario also has 2 devices `d1..d2`, connected through a 256kbps link, and 10 hosts `h1..h10`, with `h1..h5` connected to `d1` and `h6..h10` to `d2`. In both scenarios, MTU is 1500 bytes.

Fig. 7(a) and Fig. 7(b) show the results obtained for the first scenario using TCP and UDP, respectively. Observe that LTP enables each flow to obtain a roughly fair of the communication medium, with smaller fluctuations observed for UDP. Fig. 7(b) show the results obtained for the second scenario, using TCP. Observe that, again, all flows obtain a fair share of the 256kbps link. The curve “Aggregate” (that depicts the overall throughput in the link) also shows that LTP enables hosts to use the available bandwidth to its capacity.

B. Overall Performance under Varying Conditions

Here we concentrate on the performance of LTP considering a variety of technical restrictions common to LPWAN devices: narrow band links and small payloads. We consider a topology with two hosts `h1..h2` and two devices `d1..d2`, with `h1` connected to `d1`, `d2` to `h2`. Fig. 8 depicts the results achieved.

Note that LTP enables each flow to obtain a throughput performance close to the nominal link capacity, regardless of the scenario, with a stable flow between the pairs of hosts. Table I presents the results achieved for various scenarios, measured with 99% confidence level.



(a) 128 bytes MTU, TCP (b) 128 bytes MTU, UDP

(c) 512 bytes MTU, TCP (d) 512 bytes MTU, UDP

Fig. 8: LTP performance with varying link speed and MTU.

C. LTP Overhead Compared to Standard TCP/IP

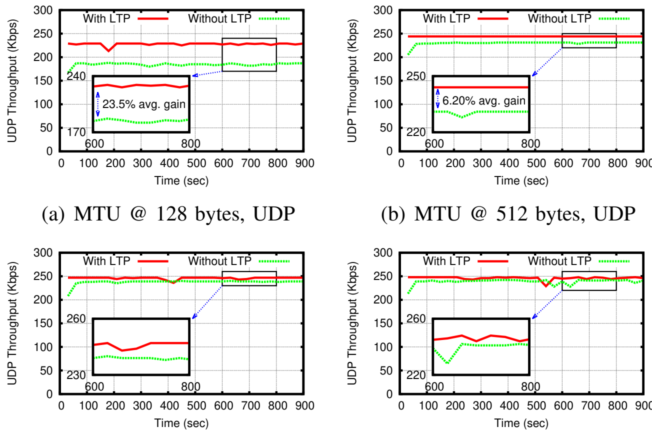
Here we consider the same topology as in the previous scenario, with a 256kbps link. Observe in Fig. 9 that LTP provides a flow performance superior to that of standard TCP/IP for every MTU considered, with smaller MTUs providing the highest gains. Considering that often LPWAN devices are capable of data transmission in small chunks, higher performance on small payloads is certainly a great advantage for LTP over standard TCP/IP, with throughput gains ranging from 6% to 23%, as shown in Fig. 9.

D. LTP Performance with Internet Traffic

In this scenario, we consider the same topology as in the previous scenario, except that `h2` is a router that provides access to the Internet. In this case, `h1` uses `wget` to retrieve

TABLE I: Experiments with various link speeds, payload lengths, and confidence level 0.99.

#	Link speed (kpbs)	Payload size (bytes)	LTP Avg	LTP SD	LTP CI	LTP Avg-CI	LTP Avg+CI	UDP/IP Avg	UDP/IP SD	UDP/IP CI	UDP/IP Avg-CI	UDP/IP Avg+CI	Gain %
1	64	128	57.1	0.15	0.08	57.03	57.18	46.3	3.05	1.54	44.73	47.80	23.44
2	64	512	61.1	0.09	0.05	61.01	61.10	56.5	2.75	1.39	55.11	57.88	8.07
3	128	128	114.0	0.41	0.21	113.76	114.17	93.1	2.58	1.30	91.83	94.42	22.38
4	128	512	121.9	0.55	0.28	121.62	122.18	113.0	4.95	2.49	110.51	115.49	7.88
5	256	128	227.8	3.04	1.53	226.27	229.33	184.4	3.82	1.92	182.48	186.32	23.54
6	256	512	243.8	0.91	0.46	243.37	244.29	229.6	4.73	2.38	227.22	231.98	6.20
7	512	128	447.3	7.52	3.79	443.55	451.12	364.7	4.19	2.11	362.59	366.81	22.66
8	512	512	487.8	1.52	0.77	487.00	488.53	453.4	7.89	3.97	449.43	457.37	7.58



(a) MTU @ 128 bytes, UDP (b) MTU @ 512 bytes, UDP

(c) MTU @ 1024 bytes, UDP (d) MTU @ 1500 bytes, UDP

Fig. 9: Flow performance with LTP and with standard TCP/IP.

a 16MB file from an HTTP server in the Internet. Name resolution (DNS) traffic also went through the link over LTP.

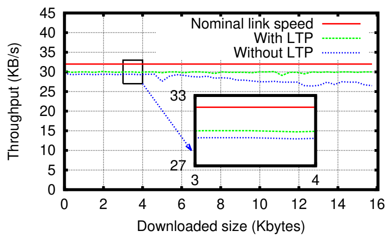


Fig. 10: File retrieval from the Internet using `wget`.

Fig. 10 depicts the results achieved. Observe that LTP provided a performance close to the nominal link speed again, consistently outperforming standard TCP/IP throughout the file transfer. The average throughput observed for LTP was 29.9 KB/s, whereas for TCP/IP was 28.2 KB/s, thus representing an average gain of around 6%.

VI. Final Considerations

The development of low-end technologies for community networks has long attracted the attention of the research community [8], and the rise of programmable forwarding planes offers an interesting novel path to be explored. In this context, the advent of *programmable low-end networking* would have great potential to unlock various networking technologies for a multitude of novel use cases, e.g., LPWAN for community networking deployment. In this paper, we contribute to the low-end networking research agenda by introducing a conceptual architecture for LPWAN programmability and exercise the architecture with the implementation of LTP, a novel protocol for data communication over the Internet that is compatible with existing TCP/IP protocols.

We emphasize that our architecture, while it does not provide the features of a fully-fledged programmable network interface, represents an important building block for experimenting with P4 for LPWAN, a topic not yet approached in the literature. About expected MAC layer, our goal is not being LoRa-compliant, but solely using available low-cost transceivers for data communication. Regarding LTP, we emphasize that our goal was exploring the power of LPWAN programmability to design a simple yet functional protocol for low-end networking. It was out of our scope to design a protocol superior to state-of-the-art solutions for data transmission optimization (like the family of solutions based on ROHC [38]). We leave this aspect, as well as security and reliability for LTP, for future research. Finally, we assume that controllers will have a unique id to avoid conflicting rules regardless of packet loss during tunnel setup. This assumption poses security challenges in case of misbehaving controllers, and a potential solution should likely use public-key infrastructure for tunnel setup (which we envisage as future work).

One main contribution of our paper is laying the ground for further research on programmable low-end networks so that they become a feasible option to connect remote communities and bridge the digital divide. All in all, such an option has the potential to make networks powered by low-cost devices flourish around the globe, fostering digital inclusion.

We are prototyping a pair of programmable low-end hardware devices for experimenting with our solution in the wild. As for future work, we intend to provide a hardware implementation of our conceptual architecture, using as a basis a Universal Software Radio Peripheral (USRP) B205mini-i Software Defined Radio (SDR) 70Mhz to 6Ghz, equipped with a Xilinx Spartan-6 XC6SLX150 FPGA, and a VERT900 Vertical Antenna (824-960 MHz, 1710-1990 MHz) dual-band.

Acknowledgements

This work has been supported in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001.

References

[1] United Nations, "The promotion, protection and enjoyment of human rights on the internet," *United Nations Digital Library*, vol. A/HRC/32/L.20, no. 32, pp. 1–4, June 2016. [Online]. Available: https://digitallibrary.un.org/record/845728

[2] Internet.org, "Internet.org website," 2020, available: https://info.internet.org/en/.

[3] Loon, "Project Loon website," 2020, available: https://loon.co/.

[4] A4AI, "A4AI Affordability Report 2018 [Online]," 2018, available: https://a4ai.org/affordability-report/report/2018/.

[5] IETF GAIA Research Group, "IETF GAIA Research Group website," 2020, available: https://datatracker.ietf.org/rg/gaia/about/.

[6] J. Crowcroft, A. Wolisz, and A. Sathiaseelan, "Towards an Affordable Internet Access for Everyone: The Quest for Enabling Universal Service Commitment (Dagstuhl Seminar 14471)," *Dagstuhl Reports*, vol. 4, no. 11, pp. 78–137, 2015. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/4971

[7] K. Heimerl, K. Ali, J. Blumenstock, B. Gawalt, and E. Brewer, "Expanding rural cellular networks with virtual coverage," in *10th NSDI 2013)*. Lombard, IL: USENIX Association, 2013, pp. 283–296. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/heimurl

[8] B. Braem, C. Blondia, C. Barz, H. Rogge, F. Freitag, L. Navarro, J. Bonicioli, S. Papathanasiou, P. Escrich, R. Baig Viñas, A. L. Kaplan, A. Neumann, I. Vilata i Balaguer, B. Tatum, and M. Matson, "A case for research with and on community networks," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 3, p. 68–73, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2500098.2500108

[9] U. Raza, P. Kulkarni, and M. Sooriyabandara, "Low power wide area networks: An overview," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 855–873, 2017.

[10] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer, "A comparative study of lpwan technologies for large-scale iot deployment," *ICT express*, vol. 5, no. 1, pp. 1–7, 2019.

[11] Datatracker, "IPv6 over Low Power Wide-Area Networks (lpwan)," 2020, available: https://datatracker.ietf.org/wg/lpwan/about/.

[12] M. Bembe, A. Abu-Mahfouz, M. Masonta, and T. Ngqondi, "A survey on low-power wide area networks for iot applications," *Telecommunication Systems*, vol. 71, no. 2, pp. 249–274, 2019.

[13] S. Nandi, S. Thota, A. Nag, S. Divyasukhananda, P. Goswami, A. Aravindakshan, R. Rodriguez, and B. Mukherjee, "Computing for rural empowerment: enabled by last-mile telecommunications," *IEEE Communications Magazine*, vol. 54, no. 6, pp. 102–109, 2016.

[14] P. Thubert, A. Pelov, and S. Krishnan, "Low-power wide-area networks at the ietf," *IEEE Communications Standards Magazine*, vol. 1, no. 1, pp. 76–79, 2017.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[16] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[17] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM 2013*. New York, NY, USA: ACM, 2013, p. 99–110.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[19] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics." New York, NY, USA: Association for Computing Machinery, 2020.

[20] W. L. da Costa Cordeiro, J. A. Marques, and L. P. Gaspary, "Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management," *Journal of Network and Systems Management*, vol. 25, no. 4, pp. 784–818, Oct 2017.

[21] UFRGS Networks Group, "GitHub - ProgrammableLowEndNetworks repo," 2021, available: https://github.com/ComputerNetworks-UFRGS/ProgrammableLowEndNetworks.

[22] J. Mitola and G. Q. Maguire, "Cognitive radio: making software radios more personal," *IEEE personal communications*, vol. 6, no. 4, pp. 13–18, 1999.

[23] Ofcom, "TV white spaces - A consultation on white space device requirements," 2012, available: https://www.ofcom.org.uk/consultations-and-statements/category-2/whitespaces.

[24] T. Luo, H.-P. Tan, and T. Q. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks," *IEEE Communications letters*, vol. 16, no. 11, pp. 1896–1899, 2012.

[25] S. Bera, S. Misra, S. K. Roy, and M. S. Obaidat, "Soft-wsn: Software-defined wsn management system for iot applications," *IEEE Systems Journal*, vol. 12, no. 3, pp. 2074–2081, 2016.

[26] P. Gallo, K. Kosek-Szott, S. Szott, and I. Tinnirello, "Sdn@ home: A method for controlling future wireless home networks," *IEEE Communications Magazine*, vol. 54, no. 5, pp. 123–131, 2016.

[27] S.-Y. Wang, C.-M. Wu, Y.-B. Lin, and C.-C. Huang, "High-speed data-plane packet aggregation and disaggregation by p4 switches," *Journal of Network and Computer Applications*, vol. 142, pp. 98–110, 2019.

[28] J. Sun, P. Dong, Y. Qin, T. Zheng, X. Yan, and Y. Zhang, "Improving bandwidth utilization by compressing small-payload traffic for vehicular networks," *International Journal of Distributed Sensor Networks*, vol. 15, no. 4, p. 1550147719843050, 2019.

[29] W.-E. Chen, W.-C. Chien, C.-F. Lai, and H.-C. Chao, "Promising framework of ethernet header compression in industrial internet of things," in *2019 International Conference on Intelligent Computing and its Emerging Applications (ICEA)*. IEEE, 2019, pp. 134–139.

[30] C. Feres and Z. Ding, "Low complexity header compression with lower-layer awareness for wireless networks," in *IEEE ICC 2019*. IEEE, 2019, pp. 1–7.

[31] J. S. da Silva, F.-R. Boyer, L.-O. Chiquette, and J. P. Langlois, "Extern objects in p4: an rohc header compression scheme case study," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2018, pp. 517–522.

[32] R. Garg and S. Sharma, "Modified and improved ipv6 header compression (mihc) scheme for 6lowpan," *Wireless Personal Communications*, vol. 103, no. 3, pp. 2019–2033, 2018.

[33] D. Kidston and P. Hugg, "Impact of header compression on tactical networks," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 1–6.

[34] Y. Niu, C. Wu, L. Wei, B. Liu, and J. Cai, "Backfill: An efficient header compression scheme for openflow network with satellite links," in *2016 International Conference on Networking and Network Applications (NaNA)*. IEEE, 2016, pp. 202–205.

[35] J. Saldana, F. Pascual, D. De Hoz, J. Fernández-Navajas, J. Ruiz-Mas, D. R. Lopez, D. Florez, J. A. Castell, and M. Nuñez, "Optimization of low-efficiency traffic in openflow software defined networks," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2014)*. IEEE, 2014, pp. 550–555.

[36] D. Gonçalves, S. Signorello, F. M. Ramos, and M. Médard, "Random linear network coding on programmable switches," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6.

[37] T.-P. Wang and Y.-C. Chen, "Adaptive packet aggregation for header compression in vehicular wireless networks," in *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE, 2011, pp. 935–939.

[38] L.-E. Jonsson, G. Pelletier, and K. Sandlund, "The robust header compression (rohc) framework," Internet Requests for Comments, RFC Editor, RFC 4995, July 2007.

[39] Dragino, "Raspberry Pi HAT featuring GPS and LoRa® technology," 2020, available: https://www.dragino.com/products/lora/item/106-lora-gps-hat.html.

[40] Ronoth, "LoSitck - USB LoRa Device – Ronoth," 2020, available: https://ronoth.com/products/lostik.

[41] P4-16, "P4-16 Language Specification," 2020, available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[42] Iana, "Ethernet Numbers," 2020, available: https://www.iana.org/assignments/ethernet-numbers/ethernet-numbers.xhtml.

[43] Scapy, "Scapy - Packet crafting for Python2 and Python3," 2020, available: https://scapy.net/.