Received July 30, 2021, accepted September 25, 2021, date of publication October 19, 2021, date of current version October 28, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3121557

Fragment Forwarding in Lossy Networks

MARTINE S. LENDERS[1], **THOMAS C. SCHMIDT**[2], **(Member, IEEE), AND MATTHIAS WÄHLISCH**[1], **(Member, IEEE)**
[1]Department of Mathematics and Computer Science, Freie Universität Berlin, 14195 Berlin, Germany
[2]Department Informatik, HAW Hamburg, 20099 Hamburg, Germany
Corresponding author: Martine S. Lenders (m.lenders@fu-berlin.de)

This work was supported in part by the German Federal Ministry of Education and Research within the projects RAPstore and PIVOT, and in part by the Open Access Publication Fund of the Freie Universität Berlin.

ABSTRACT This paper evaluates four forwarding strategies for fragmented datagrams in the Internet of Things (IoT). We focus on classic end-to-end fragmentation, hop-wise reassembly, a minimal approach to direct forwarding of fragments, and direct forwarding utilizing selective fragment recovery. To fully analyze the potentials of selective fragment recovery, we include four common congestion control mechanisms. We compare all fragmentation strategies comprehensively in extensive experiments to assess reliability, end-to-end latency, and memory consumption on top of IEEE 802.15.4 and its common CSMA/CA MAC implementation. Our key findings include three takeaways. First, direct fragment forwarding should be deployed with care since higher packet transmission rates on the link layer can significantly reduce reliability, which can even further increase end-to-end latency because of highly increased link layer retransmissions. Second, selective fragment recovery can mitigate the problems underneath. Third, congestion control for selective fragment recovery should be chosen such that small congestion windows grow together with fragment pacing. In case of fewer fragments per datagram, pacing is less of a concern but the congestion window is limited by an upper bound.

INDEX TERMS Embedded networks, Internet of Things (IoT), IP networks, fragmentation, multihop wireless mesh networks.

I. INTRODUCTION

The advent of the Internet of Things (IoT) increased deployment of resource constrained, heterogeneous, and wireless devices that join the wider Internet. This change also increased the deployment of heterogeneous access networks, which introduce a variety of maximum packet sizes on the link layer, many of them below 250 bytes, see Figure 1. On the network layer, however, nodes predominantly speak IPv6 [6] using a mandatory transparent Maximum Transmission Unit (MTU) size of at least 1280 bytes. Since header compression cannot fully mitigate this problem, fragmentation on the lower layer is necessary to support communication based on IoT link layer technologies.

One of the most popular IoT link layer technologies, IEEE 802.15.4 [1], only allows for the transmission of a very limited number of bytes. For efficiency reasons, information required to forward a packet cannot be encoded in every fragment but is only present in the first fragment, based on the IPv6 adaptation layer 6LoWPAN [7]. There are two concepts for forwarding fragmented datagrams in 6LoWPAN. First, reassembly is performed at every hop (*hop-wise reassembly*), followed by re-fragmentation when forwarded on another constrained link (see Figure 2b). As the forwarding information is only being stored in the first fragment this is the simplest solution. Second, individual fragments are forwarded (*fragment forwarding*) by recording the forwarding information required from the first fragment on all participating nodes. This recorded information then can be used to forward all subsequent fragments to the next hop [8, Section 2.5.2], [9] (see Figure 2c). Since losing a single fragment requires resending of the whole datagram on an upper layer, fragment forwarding can be extended by *selective fragment recovery* that allows the reassembling end-point to cumulatively acknowledge received fragments to the fragmenting end-point (see Figure 2d). Due to the higher traffic load by forwarding fragments directly, *selective fragment recovery* also provides the basis to deploy congestion control.

Fragmentation enables packet delivery because it splits packets to comply with the maximum sized datagram that can be transmitted on a link. Choosing the right fragmentation

The associate editor coordinating the review of this manuscript and approving it for publication was Tawfik Al-Hadhrami.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

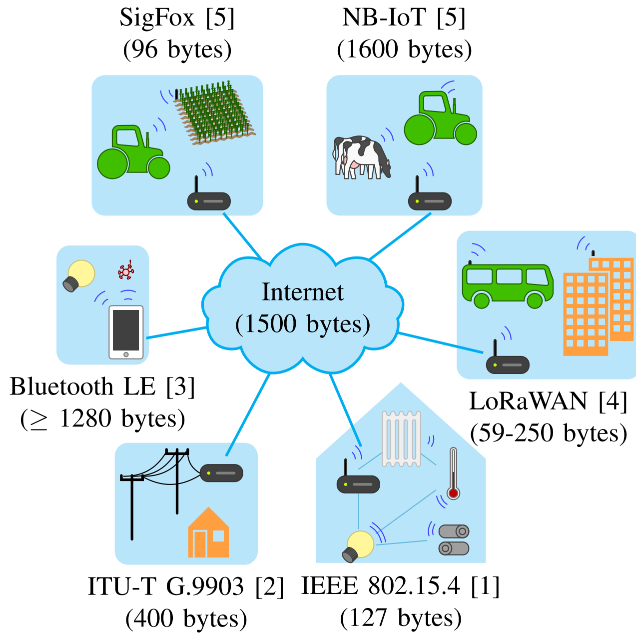


FIGURE 1. Common IoT link layer technologies deployed in typical IoT scenarios and their maximum Service Data Units. Link layer technologies include LoRaWAN, SigFox, and NB-IoT representing low-power wide-area networks, Bluetooth Low-Energy and IEEE 802.15.4 for low-power personal-area networks, and ITU-T G.9903 (power-line communication) for local-area networks.

strategy is intricate since the strategy affects packet delivery ratio, latency, and system resources. In this paper, we comparatively assess the performance and resource consumption of *hop-wise reassembly*, direct *fragment forwarding*, and *selective fragment recovery* over the very thin IEEE 802.15.4 MAC layer. As a point of reference, we include *end-to-end fragmentation* as well. As part of this work, we also provide an independent implementation of both simple fragment forwarding and its selective fragment recovery variant, which we showcase to allow deeper insights into our evaluation results.

To evaluate which improvements an optional congestion control could offer to cope with fragment retransmissions in lossy networks, we compare four common congestion control mechanisms in *selective fragment recovery* on the same MAC layer: (*i*) A simple mechanism as proposed in the appendices of RFC 8931, (*ii*) a TCP-Reno-like approach that only takes loss into account for congestion control, (*iii*) a TCP-Reno-like approach with *Alternative Exponential Backoff* (ABE) that introduces special behavior for *Explicit Congestion Notification* (ECN), and finally (*iv*) a QUIC-like approach that adds packet pacing on top of that. To evaluate the different approaches we developed *CongURE*, a framework for *Cong*estion Control *U*tilzing *R*eusable *E*lements. CongURE allows for drop-in replacements of the congestion control mechanism in a protocol.

Our findings reveal the drawbacks of using the widely deployed very thin MAC layer below 6LoWPAN fragment forwarding. While hop-wise reassembly manages to transmit at least a fraction of the data, even with a higher number of fragments, direct fragment forwarding techniques quickly drop to 0% performance. Selective fragment recovery helps to mitigate those drawbacks, but only when it comes to packet delivery ratio and with low window sizes. Classic congestion control mechanisms based on *Additive Increase, Multiplicative Decrease* (AIMD), slow start, congestion avoidance, and recovery help to find a balance between latency and high packet delivery ratio with SFR over the very thin MAC layer when combined with RTT-based pacing.

In summary, our work on a comprehensive picture on fragment forwarding in lossy networks makes the following contributions:

1) Comparative evaluation of four fragment forwarding approaches (*end-to-end fragmentation*, *hop-wise reassembly*, direct *fragment forwarding*, and *selective fragment recovery*) on top of a very thin MAC layer.
2) The analysis of congestion control in the context of *selective fragment recovery*. To this end, we design the integration of three common congestion control mechanisms (*TCP Reno*, *TCP ABE*, and *QUIC* congestion control) in SFR, and implement these mechanisms.
3) Comparative evaluation of congestion control mechanisms in SFR on top of a very thin MAC layer.
4) *CongURE*, a lightweight congestion control framework for low-end IoT devices, which allows for the drop-in replacement of congestion control mechanisms independently of a specific protocol.

The remainder of this paper is structured as follows. In Section II, we provide background on 6LoWPAN fragmentation and forwarding as well as congestion control options for SFR. In Section III, we outline our implementations of the different fragment forwarding options and the CongURE framework. We present the results of our experiments on fragment forwarding and congestion control in Section IV and Section V, respectively. In Section VI, we summarize related work. We discuss our overall findings in Section VII, and close with a conclusion and an outlook in Section VIII. Table 1 summarizes the key abbreviations used throughout this paper.

TABLE 1. Abbreviations used throughout this paper.

Abbreviation	Meaning
ARQ	Automatic Repeat Request
BLE	Bluetooth Low Energy
CC	Congestion Control
E2E	Classic End-to-End (IPv6) Fragmentation
FF	(Direct) Fragment Forwarding
HWR	Hop-wise Reassembly
IFG	Inter-frame Gap
IPHC	IPv6 Header Compression
RFRAG	Recoverable Fragment
PDU	Protocol Data Unit
SCHC	Static Context Header Compression and Fragmentation
SDU	Service Data Unit
SFR	Selective Fragment Recovery
VRB	Virtual Reassembly Buffer

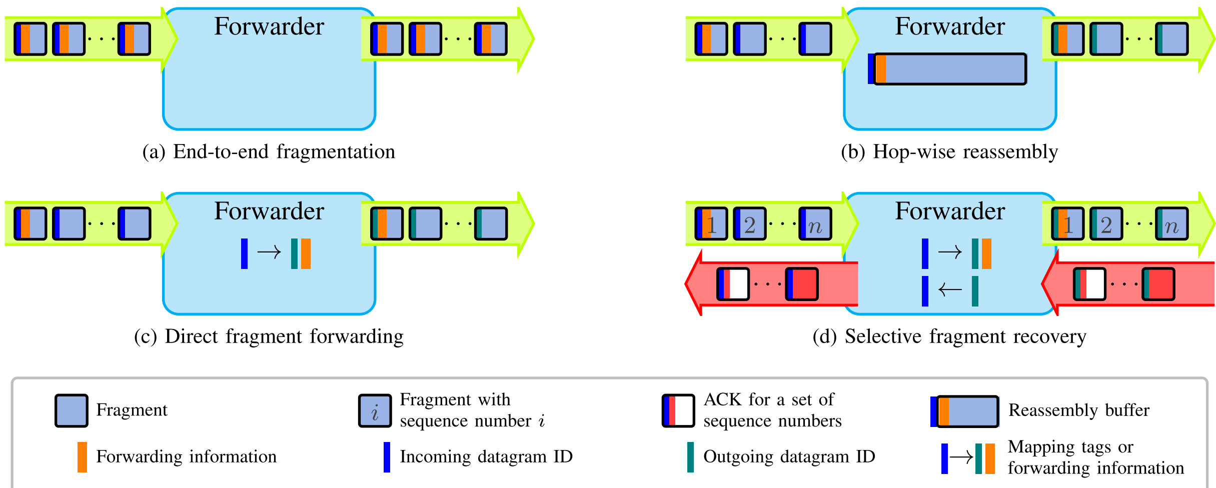


FIGURE 2. The four approaches for fragment forwarding in low-power and lossy networks compared in this paper.

II. BACKGROUND AND PROBLEM STATEMENT

The IETF specified the 6LoWPAN protocol [7] to allow for transmissions of IPv6 packets over IEEE 802.15.4 [1] networks, a widely used link layer technology in the IoT. While IPv6 requires a Maximum Transmission Unit (MTU) of at least 1280 bytes [6], IEEE 802.15.4 is only able to handle link layer frames of up to 127 bytes—*including* the link layer header. Enabling link layer security, this leaves 81 bytes for the network layer [10]. Considering 48 bytes for the IPv6 and UDP headers, this leaves only 33 bytes for application data per frame. To enable IPv6 communication in such a restrictive environment, 6LoWPAN provides both header compression [11], [12] and datagram fragmentation [7].

Header compression is applied to a datagram before it is sent, even before it is fragmented. Two types of header compression are supported: (*i*) the classic approach of field elision based on a bit mask as specified in RFC 6282 [11] and (*ii*) Generic Header Compression as described in RFC 7400 [12] based on LZ77-style compression [13]. Option (*i*) is expected to be supported by all 6LoWPAN nodes. Option (*ii*) is not widely deployed yet and requires an exchange of capabilities between nodes. The impact of both compression schemes on performance is less compared to fragmentation: Header compression aims to add a slight processing overhead to reduce the need for fragmentation while fragmentation introduces performance issues for each single fragment a datagram is separated into. To better understand performance challenges with larger impact, fragmentation is the focus of this paper.

It is worth noting that the concept of 6LoWPAN (or more generally *6Lo*) is not limited to IEEE 802.15.4 but can also be used in other link layer technologies such as PLC [2] or Bluetooth Low Energy [3] (see Figure 1). There is also Static Context Header Compression and Fragmentation (SCHC) [14] for even more restrictive link layer technologies such as LoRaWAN [15], SigFox [16], and NB-IoT [17]. The fragmentation modes of SCHC, while incompatible to the fragmentation approaches of 6LoWPAN, are based on the same principles. As such, the same conclusions of this paper apply, as we will discuss later in this paper.

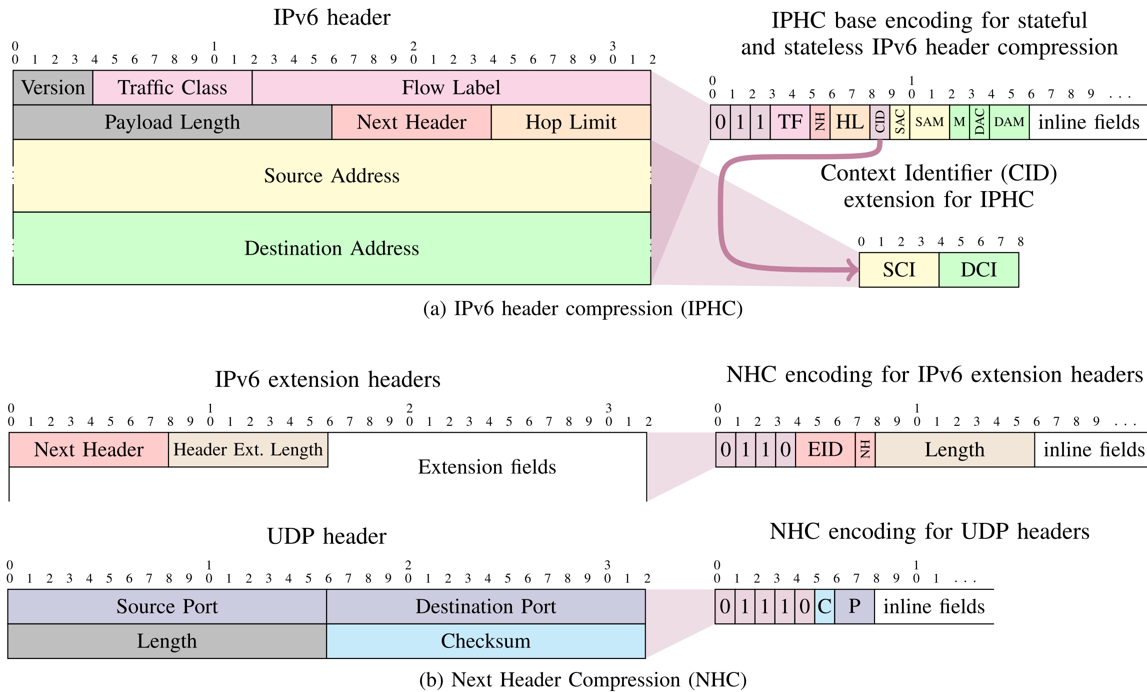
A. 6LoWPAN HEADER COMPRESSION

In this paper, we will focus on fragmentation. To understand, however, how fragmentation decisions are made and also why packets cannot simply carry a compressed header in every fragment, a basic understanding of header compression is needed. We will discuss classic 6LoWPAN header compression [11] because this scheme is widely supported; in the context of fragmentation, similar results apply to Generic 6LoWPAN header compression [12].

Classic 6LoWPAN header compression supports two modes, *IPv6 Header Compression* (IPHC) (see Figure 3a) and *Next Header Compression* (NHC) (see Figure 3b). Both modes use the elision or replacement of fields and use lower layer information and bit-fields signifying which fields are elided or replaced.

IPHC is applied at every hop in a network. Both the 4-bit version field and the 16-bit length field of the IPv6 header are elided in any case. The version field is always set to the value 6 and is thus redundant, and the length field can be derived from the lower layers (*i.e.,* 6LoWPAN fragmentation headers or the link layer). The 8-bit traffic class field and the 20-bit flow label field of the IPv6 header can be elided: The TF flags encode the information which of those fields or their subfields are elided. Special values of the 8-bit hop limit field of the IPv6 header mapped to the 2-bit HL field: 00 means the hop-limit is carried inline, 01 means the hop limit is implied to be 1, 10 means the hop limit is implied to be 64, and 11 means the hop limit is implied to be 255. In all but the first case the hop limit field is elided.

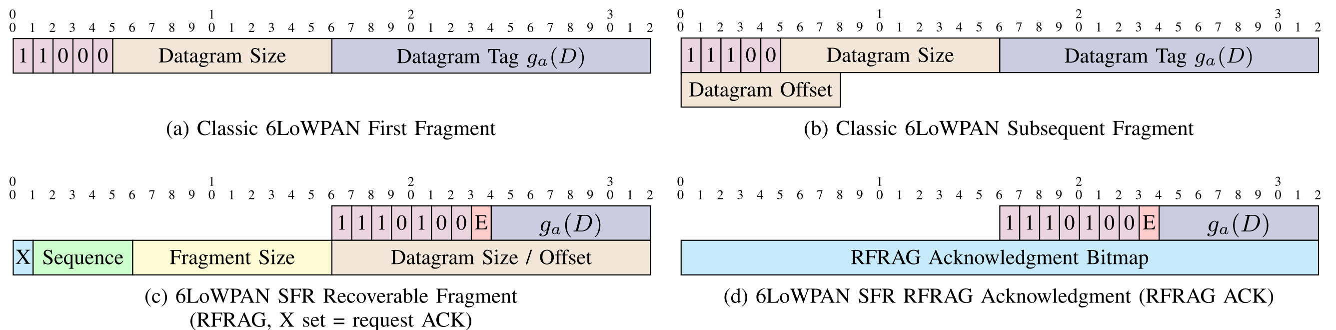
IPHC allows for both stateless and stateful address compression of the 128-bit source and destination address fields



(a) IPv6 header compression (IPHC)

(b) Next Header Compression (NHC)

FIGURE 3. Encodings for 6LoWPAN header compression. Compression encodings are marked in the corresponding color of the field they compress.



(a) Classic 6LoWPAN First Fragment

(b) Classic 6LoWPAN Subsequent Fragment

(c) 6LoWPAN SFR Recoverable Fragment (RFRAG, X set = request ACK)

(d) 6LoWPAN SFR RFRAG Acknowledgment (RFRAG ACK)

FIGURE 4. 6LoWPAN fragmentation headers.

within the IPv6 header. The combination of SAC (source address compression), SAM (source address mode), M (multicast compression), DAC (destination address compression), and DAM (destination address mode) flags reflect the respective mode; alternatively, the address is carried fully inline. Details are described in RFC 4944 [7]. In short, when only considering unicast communication, stateless address compression is used for link-local addresses (eliding the `fe80::/64` prefix) and stateful address compression can be used for all other addresses. The prefix elided by stateful compression is encoded for both source and destination address based on an optional 4-bit context identifier (CID) extension. If the identifiers for both source and destination address are 0 or if both addresses use stateless address compression that extension can be elided, signified by clearing the CID flag in the IPHC encoding. How the CID-to-prefix mapping is shared between the nodes is out of scope of the IPHC specification [11]. One option is discussed in the Neighbor Discovery optimization for 6LoWPAN [18].

The suffixes or interface identifiers (IIDs) of addresses can be elided when they are based on a link layer address [7] and if one of the nodes involved in the communication is the node with that link layer address. There are no special signifier bits for IID elision.

NHC supports IPv6 extension headers and UDP headers. If NHC is used, the NH bit in the IPHC encoding is set. Otherwise, the 8-bit next header field is carried inline and the remaining headers are uncompressed.

IPv6 extension header compression allows for the removal of padding within the extension headers and thus internal fragmentation of the headers by re-defining the length field in units of bytes and not 8 bytes as specified in [6].

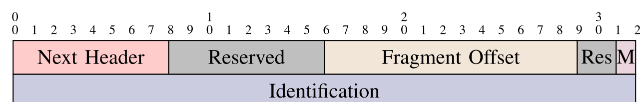


FIGURE 5. IPv6 fragment extension header (M set = more fragments, M unset = last fragment).

If an extension header exceeds the length of 255 and thus overflows the 8-bit length field, NHC cannot be used for that extension header. The type of extension header is signified by the EID (Extension Header ID) flags in the encoding. It also allows for encapsulated IPHC encodings and thus compressed IPv6-in-IPv6 tunneling. The 8-bit next header field of an extension header can also be elided, if the following header is also compressed by NHC.

UDP header compression allows for the elision of the checksum if and only if the upper layer provides other means of integrity checks of the payload. If the checksum can be elided, the C flag in the UDP compression header is set to 1. Moreover, port compression is supplied for special port ranges. These port ranges are encoded with the 2-bit P flag field. For more details about those port range encodings, we refer to [11].

B. WHY NOT IPv6 END-TO-END FRAGMENTATION (E2E)?

IPv6 fragmentation requires the lower layer to be able to support a minimum MTU of 1280 bytes. Many IoT link layer technologies, however, allow only for much smaller MTUs (*e.g.,* 127 bytes in IEEE 802.15.4 with and 400 bytes in ITU-T G.9903). Carrying IPv6 source and destination addresses already requires space for 32 bytes (2×128 bits) of data when compression is not possible, see Section II-A. 32 bytes are more than a fifth of the frame SDU of IEEE 802.15.4. Because of that, IPv6 fragmentation alone is not suitable on top of IoT link layer technologies. This justifies a solution that contains the forwarding information only in the first fragment.

IPv6 fragmentation also requires a 32-bit identifier to associate fragments to a complete datagram end-to-end [6], see Figure 5. A link-wise identifier of shorter length can help to further save space in a constraint link layer frame.

Furthermore, some extension headers have to be carried in every fragment [6] and the capabilities of next header compression for extension headers is very limited, see Section II-A.

As we will see in Sections II-C and II-D, other fragment forwarding approaches require overhead in form of new data structures to save space in the transmitted fragment. To compare IPv6 end-to-end fragmentation with IoT-specific fragmentation solutions, we will use a modified version of IPv6 fragmentation that allows for such small fragments in our analysis. We simply set the MTU of the interface so that a compressed IPv6 fragment fits the link-layer PDU at all hops in the network, and configure the network stack as such to not use the 6LoWPAN module. We will call this approach *end-to-end fragmentation* (E2E, see Figure 8a) in the remainder of this paper.

C. BASIC FRAGMENTATION AND REASSEMBLY IN 6LoWPAN

In 6LoWPAN, datagram fragmentation implements the following common approach, similar to IPv6 fragmentation (see Section II-B): Before sending a datagram to the underlying link layer, the network layer checks whether the data exceeds the maximum payload length (commonly referred to as SDU, Service Data Unit) of the link layer. If the data size complies with the SDU, a single datagram is sent without any modification. If the data size does not comply with the SDU, a datagram is divided into multiple fragments such that the content of each fragment matches the SDU, see Figure 6. Each fragment includes a fragment header containing information to assemble the datagram [7]:

The fragmentation header of the first fragment contains a 16-bit datagram tag to identify the fragment on the link and an (uncompressed) datagram size in bytes as an 11-bit number, see Figure 4a. The datagram size can be encoded as a 11-bit number, because the MTU for IPv6 over 6LoWPAN is capped at 1280 bits, which is less than $2^{11} - 1 = 2047$ bits but greater than $2^{10} - 1 = 1024$ bits. This way we can keep the header as small as possible, *i.e.,* the 5-bit dispatch to identify the header type, the 11-bit datagram length, and the 16-bit datagram tag form a header of only 32 bits (or 4 bytes). After the first fragment, all subsequent fragments carry—in addition to the header fields of the first fragment header—an offset to refer to this fragment in units of 8 bytes, see Figure 4b. Consequently, all payloads in a fragment must be of a length that is a multiple of 8.

The receiver identifies multiple fragments that belong to the same datagram by comparing two values: the link layer source addresses and the datagram tag. Then, the receiver network stack stores all fragments of an incoming datagram in the *reassembly buffer* for up to 10 seconds. In the following, we will refer to identifiers that map fragments to a datagram D by $(a, g_a(D))$ with a being the link layer source address and $g_a(D)$ being the datagram tag for D on the link to a (*cf.,* Figure 4).

A brief back-of-the-envelope calculation shows that a node needs to allocate at least **1291 bytes** of memory **per reassembly buffer entry** to reassemble a fragmented datagram. In detail:

- At most **8 bytes**, plus **1 byte** to store link layer source address and its length, since IEEE 802.15.4 supports two addressing formats (64-bit EUI-64s and a 16-bit short addresses),
- **2 bytes** for the datagram tag, and
- **1280 bytes** for the maximum expected size of an IPv6 datagram.

1291 bytes are significant memory requirements on constrained devices, which typically offer memory within the range of several kilobytes [19]. Especially in a multihop network—a common deployment scenario in the

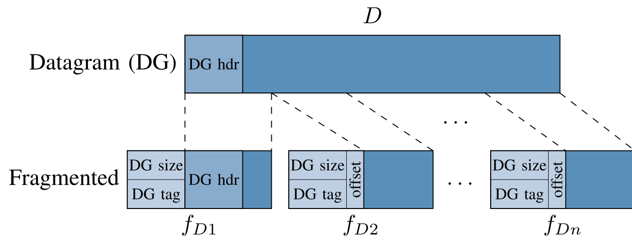


FIGURE 6. Fragmentation in 6LoWPAN.

IoT—it becomes challenging to provide enough resources to store a sufficient number of reassembly buffer entries. In Section III-A, we show how to save memory in a concrete implementation.

It is worth noting that RFC 4944 [7] specifies four components to identify all fragments that belong to a datagram (*e.g.,* the link layer destination address and the datagram size). RFC 8930 [9], however, argues that only the link layer source address and the datagram tag are necessary for such an identification. As this paper compares classic hop-wise reassembly based on RFC 4944 [7] with approaches defined in RFCs 8930 [9] and 8931 [20], we decided to use the reduced $(a, g_a(D))$ identifier in all approaches.

D. HOP-WISE REASSEMBLY (HWR) VS. DIRECT FRAGMENT FORWARDING (FF)

The destination address in the IPv6 header guides forwarding. In 6LoWPAN fragmentation, however, the IPv6 header is only present in the first fragment. To enable intermediate nodes in a multihop network to forward fragments without this context information, two solutions are proposed: hop-wise reassembly and direct fragment forwarding.

The naive approach to handle fragmented datagrams in a multihop network is *hop-wise reassembly (HWR)* [6], [7]. In HWR, each intermediate hop between source and destination assembles and re-fragments the original datagram completely. This leads to three drawbacks. First, each intermediate hop needs to provide enough memory resources to store all fragments in the reassembly buffer (see Figure 8b). Second, the memory requirements are unbalanced between nodes in the network. Considering highly connected nodes (see node e in Figure 7), these nodes need to cope with the reassembly load of all their downstream nodes. Third, datagram delivery time is bound by the time needed to receive all fragments of the datagram. Papadopoulos *et al.* [21] underscored these problems in more detail.

Fragment forwarding (FF) [9] tackles the drawbacks of HWR by leveraging a *virtual* reassembly buffer *(VRB)* [22], see Figure 8c. In contrast to a reassembly buffer, a VRB only stores references to link the subsequent fragments to the first fragment such that intermediate nodes can determine the next hop. In detail, the VRB is applied as follows. Each entry represents the source address and the incoming datagram tag $(a, g_a(D))$ (*cf.,* Section II-C), the next hop link layer address b,

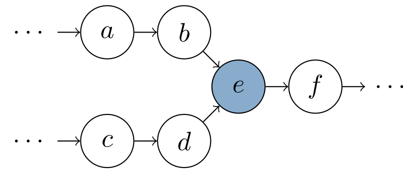


FIGURE 7. *e* represents a typical bottleneck when using HWR.

and the outgoing datagram tag $g_b(D)$:

$$(a, g_a(D)) \mapsto (b, g_b(D))$$

This has two implications. First, an intermediate node can ensure that datagram tags are unique between a node and its neighbors. Second, all fragments belonging to the same datagram will travel the same path.

E. SELECTIVE FRAGMENT RECOVERY (SFR)

In case of HWR or FF, losing fragments is costly when the complete datagram needs to be retransmitted by an upper layer. To that end, Selective Fragment Recovery (SFR) was introduced to 6LoWPAN within the IETF [20]. SFR utilizes the same mechanisms of FF but introduces new header formats, the recoverable fragment (RFRAG) header and RFRAG acknowledgment. SFR is thus a completely new protocol. In addition to datagram tag and datagram offset, those headers include a 5 bit sequence number, see Figure 4c, which allows for lightweight cumulative acknowledgments (ACKs) based on a 32 bit bitmap, see Figure 4d. Those acknowledgments can be requested by the fragmenting endpoint using the X flag in the RFRAG header. Using a configurable window size for setting this flag and an Explicit Congestion Notification (ECN) flag E, optional congestion control can be provided in SFR.

As SFR uses the source address and the datagram tag to identify a datagram, a reverse lookup in the VRB by next hop address and outgoing tag can be used to send ACKs to the datagram source, see Figure 8d. Another crucial difference to HWR and FF is that datagram size and offset share the same field: while in the first fragment that field denotes the size of the datagram, in all subsequent fragments it denotes the offset. Therefore, the true size of a datagram can only be known by the reassembling endpoint, when the first fragment arrived.

To recover fragments, the fragmenting end-point triggers a timeout—the Automatic Repeat Request (ARQ) timeout—whenever it sends out a fragment marked with the ACK request flag. If an ACK is not received within that timeout, or a received ACK with the same datagram tag compared to the ACK of the requesting fragment marks a previously sent fragment as not received in its 32 bit bitmap, the fragmenting end-point may resend the fragment. If a pre-configured number of resends fails, the fragmenting endpoint can either try to resend the whole datagram a pre-configured number of times or give up on sending the datagram. The maximum number of fragments in flight can be either configured statically or adaptively controlled by an optional congestion

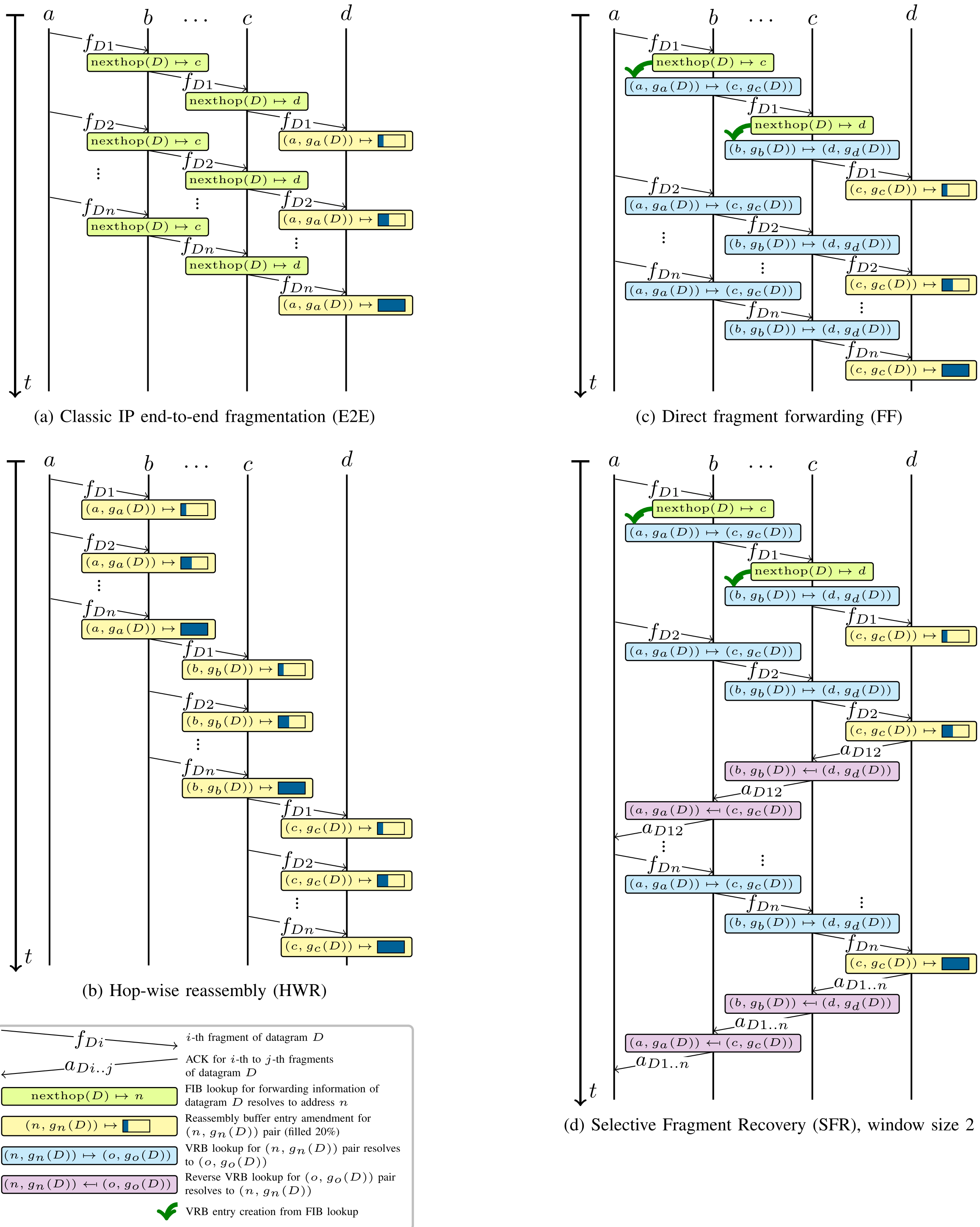


FIGURE 8. The fragment forwarding approaches compared for a datagram *D* (*a*: source; *b* and *c*: intermediate hops; *d*: destination; $g_n(D)$: datagram tag of *D* as generated by node *n*).

control mechanism. To further reduce congestion within the network, RFC 8931 [20] specifies a configurable inter-frame gap (IFG), which defines the time an SFR-capable node has to wait between sending either fragments or ACKs.

F. CONGESTION CONTROL FOR 6LoWPAN FRAGMENTATION

Since the acknowledgment mechanism in SFR is cumulative, multiple fragments can be sent at once. Considering the

resource restrictions of the constraint devices on path, the number of fragments should be limited by a congestion control (CC) mechanism, though. Moreover, as we will show, the single-antenna radio used by such devices can itself be a point of congestion in case fragments are directly forwarded. To find a balance between not exhausting the resources en-route and reducing latency penalties due to the waiting for an ACK after a number of fragments have been sent, CC should be an integral part of SFR.

SFR supports an *Explicit Congestion Notification* (ECN) mechanism. RFC 8931, however, defines only that a reassembling end-point that receives any RFRAG with the ECN flag E set, must set the E flag in the ACK to acknowledge that fragment as well (*cf.,* Figures 4c and 4d). As the reasons for a loss (congestion, interferences, *etc.*) are not clearly distinguishable in LLNs, using ECN as part of CC is crucial. Furthermore, in the appendices of RFC 8931 [20], some considerations are made how CC can be provided in SFR, but the specification also states that more experimentation is needed. This paper contributes these experiments.

To analyze CC from different angles, we consider four CC mechanisms specified by the IETF: (*i*) the base mechanism described in the appendices of RFC 8931, (*ii*) *TCP Reno* as described in RFC 5681 [23] to use the same mechanism for CC mostly deployed in the wider Internet, (*iii*) an extension of the latter using an *Alternative Exponential Backoff* (ABE) [24] and ECN to adapt the congestion window, and (*iv*) the congestion control mechanism of QUIC [25] to analyze the impact of its adaptive pacing mechanism. It is worth noting that the CC in 6LoWPAN fragmentation operates per link and not end-to-end, which decouples CC from the original source and final destination.

G. MAC LAYER

The MAC layer plays a crucial role in preventing packet losses because it controls the medium access. IEEE 802.15.4 provides support for a variety of MAC modes. The base specification [1] only supports the *non-beacon enabled* (NBE) mode and the *beacon enabled* (BE) mode. IEEE 802.15.4e extends IEEE 802.15.4 by *Time Slotted Channel Hopping* (TSCH) and the Deterministic and Synchronous Multi-channel Extension (DSME) [26]. While for BE, NBE, and TSCH specifications an IETF-compatible network layer exists [7], [27], IPv6 over DSME is not specified.

Even though open source implementations for both TSCH [28] and DSME [29] are available, operating systems such as Contiki, RIOT, or Zephyr provide only a very thin MAC layer by default. This thin MAC layer supports CSMA/CA, link layer retransmissions, and acknowledgments. Many other crucial features to improve coordination among nodes, such as joining procedures, scanning, or indirect transmissions, are not implemented because most IEEE 802.15.4 radios do not allow operating system developers access to those features.

In this paper, we analyze fragmentation on top of a thin, unreliable link layer for two reasons. First, such a link layer is common in IoT deployments. Second, in this setting, fragmentation challenges network reliability since the loss of a single fragment may amplify load and thus congestion.

III. SYSTEM DESIGN AND IMPLEMENTATION IN RIOT

A thorough experimental evaluation of protocols requires sound software implementations. For the sake of comparison, the protocols under investigation should be analyzed on the same system. Unfortunately, there is no software basis available which assembles all required components for constrained devices. In this paper, therefore, we extend RIOT [30], a common IoT operating system. By selecting an open source platform and making our software publicly available we enable reproducible research [31], [32]. Based on our extensions, we gain detailed insights into system and network performance.

In the remainder of this section, we present design, implementation, and configuration choices to better understand the subsequent evaluation.

A. 6LoWPAN AND MAC LAYER

RIOT provides a stable 6LoWPAN implementation as part of its default network stack, GNRC [30]. Instead of statically allocating packet space for each reassembly buffer, it uses the preconfigurable packet allocation arena of GNRC, called `gnrc_pktbuf`, to dynamically allocate packet buffer space of varying length within it. This allows for high resource efficiency and flexibility. By storing the major part of the IPv6 datagram (1280 bytes) only in the packet buffer, the 6LoWPAN stack requires **22 bytes** (plus some additional bytes for management), instead of allocating the complete 1291 bytes (*cf.,* Section II-C).

To provide low delays and high throughput, the fragmentation is done asynchronously. For this purpose, the reference to the datagram that needs to be fragmented is stored in a fragmentation buffer. The data of the datagram resides in `gnrc_pktbuf`. In addition to the datagram, the fragmentation buffer also contains meta-information needed for fragmentation, including the original datagram size and its tag. The software architecture is summarized in Figure 9.

GNRC only provides a very thin MAC layer that benefits from radio drivers that support CSMA/CA, link layer retransmissions, and acknowledgment handling by default. Special care has to be taken in case of hardware platforms that use ‘‘blocking wait on send’’ whenever the device is in a busy state. When deploying fragment forwarding, this may cause race conditions within the internal state machine of the device [33] because of the faster interchange of simultaneous sending and receiving events. To solve this problem, we provide a simple mechanism to queue packets whenever the device signals that it is in a busy state. As soon as the device becomes available again (and not later than 5 ms), the MAC layer tries to resend the packet from the top of the queue.

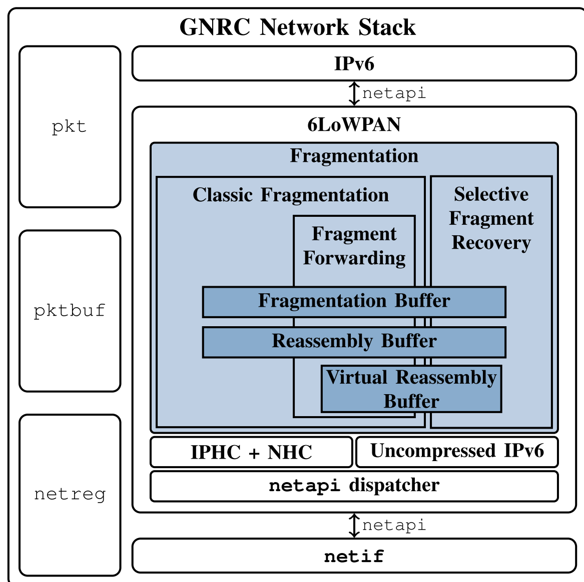


FIGURE 9. 6LoWPAN system architecture of RIOT.

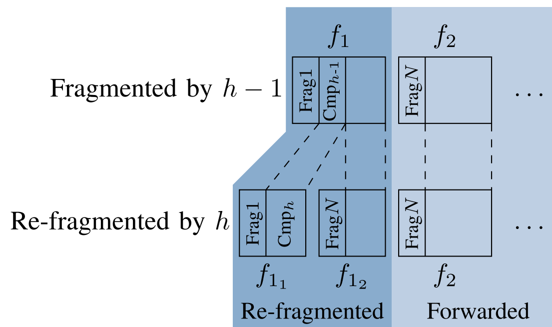


FIGURE 10. Compression header (Cmp) handling for fragment forwarding in the RIOT GNRC.

B. FRAGMENT FORWARDING

We extend 6LoWPAN in GNRC to support direct *fragment forwarding*. One crucial implementation choice relates to the creation of the first fragment. The first fragment may include the compression header [11], which may change size during network traversal as compression contexts such as link-layer addresses change. Because of that, the compression may be less or more effective depending on header updates made by intermediate forwarders. In the worst case, the packet becomes less compressed, leading to additional fragmentation. To tackle this problem, we apply a well-known approach by keeping the first fragment as minimal as possible [22], *i.e.,* the original sender includes only the fragment and compression headers and pushes the payload to the subsequent fragment. It is worth noting that this approach does not increase the overall number of fragments compared to a naive approach that minimizes the size of the last fragment. In fact, it will reduce the likely creation of additional fragments.

We support this mechanism not only on the original sender but also on intermediate forwarders for the case that the original sender did not provide enough space for the expanding compression header, see Figure 10. This is possible, as all subsequent fragments also contain an offset, which indicates fragmentation relating to the first fragment. Furthermore, it simplifies the implementation greatly, which in turn saves ROM. Since the fragmentation buffer is used for this, its default size of 1 needs to be increased so that the node is able to handle multiple datagrams—forwarded datagrams and datagrams sent by the node itself—at the same time.

To keep the implementation simple, we only forward fragments when the first fragment is received in order, otherwise we reassemble the packet completely. This can be considered a fall-back to hop-wise reassembly. We are able to do this in contrast to just dropping the fragments as proposed in RFC 8930 [9] since in RIOT (*i*) the reassembly buffer is not as expensive for static memory as assumed in RFC 8930 due to the dynamic allocation arena for packets and (*ii*) later incoming datagrams are preferred over incomplete reassembly buffer entries when the reassembly buffer is full.

C. SELECTIVE FRAGMENT RECOVERY

We also extend 6LoWPAN in GNRC to support *selective fragment recovery* (SFR). Due to the different dispatches of SFR, our implementation can be run together with classic 6LoWPAN fragmentation, *i.e.,* also with simple fragment forwarding. To that end, our SFR implementation reuses the fragmentation buffer and reassembly buffer of classic 6LoWPAN (*cf.,* Section III-A) and the virtual reassembly buffer of our direct fragment forwarding implementation (*cf.,* Section III-B) with slight adaptations to handle the new features, such as tallying up the sequence numbers of fragments in the reassembly buffer to be able to generate the cumulative acknowledgments.

In contrast to our fragment forwarding implementation, we do not only send the compression header in the first fragment, because the specification of SFR accounts for the compression header to change. If the VRB is full, our implementation falls back to reassembly, making the node on which that error occurs the reassembling endpoint.

D. CongURE: CONGESTION CONTROL FRAMEWORK

To evaluate different congestion control mechanisms for SFR in a modular way, we designed the *CongURE* (*Cong*estion Control *U*tilizing *R*eusable *E*lements) framework as part of the `2021.04` release of RIOT. CongURE is designed to be unit agnostic so it can be used in several use cases, such as TCP or QUIC where the window size is in bytes [23], [25], [34] or SFR where the window size is in fragments [20].

CongURE provides two operations to fetch the current congestion state:

1) *cwnd(): **congure_wnd_size_t*** to get the current congestion window size in user defined units.
2) *inter_msg_interval(msg_size: **unsigned**): **int*** to get the currently calculated inter-message interval in

milliseconds for pacing. The size of the next message *msg_size* can be used to go into the calculation of the interval. If pacing is not supported by the congestion control mechanism, the operation returns −1.

There are six operations to report various congestion events:

1) *report_msg_sent(msg_size: **unsigned**)* to report a message as sent. The message size *msg_size* is to be provided in user-defined units. This method is used to increase the internal count of in-flight messages.
2) *report_msg_discarded(msg_size: **unsigned**)* to report a message as discarded for any other reason than timeout or loss. The message size *msg_size* is to be provided in user-defined units. This method decreases the internal count of in-flight messages.
3) *report_msgs_timeout(msgs: **congure_msg_t[]**)* to report a collection of messages *msgs* times out. A collection of messages is used, as timed out messages are reported in bulk for many congestion control mechanisms.
4) *report_msgs_lost(msgs: **congure_msg_t[]**)* to report a collection of messages *msgs* is lost. In case the congestion control mechanisms does not distinguish between loss or timeout, *report_msgs_lost()* is an alias of *report_msgs_timeout()*.
5) *report_msg_acked(msg: **congure_msg_t**, ack: **congure_ack_t**)* to report an acknowledgment *ack* for a previously sent message *msg*.
6) *report_ecn_ce(time: **time_t**)* to report an ECN congestion encounter for a message sent at *time*.

To deploy CongURE with SFR, a simple but optional adapter was written that provides a CongURE object for every fragmentation buffer entry. The reporting operations of the CongURE object are then called for each corresponding congestion event.

IV. COMPARISON OF FRAGMENT FORWARDING METHODS

In this section, we compare four fragment forwarding methods. Our goal is to carefully explore the behavior of the competing fragmentation schemes. Along this line we show that using fragment forwarding over the very thin MAC layer that many IoT systems deploy does more harm than good. The key MAC layer components (CSMA/CS, link layer retransmissions, and acknowledgments) are not sufficient to cope with interferences in fragmentation scenarios.

Our experiments are conducted in a real-world testbed using class-2 IoT nodes [19] and 802.15.4 radio communication. One important aspect of the experiment design is the underlying network topology, which we consider by selecting specific nodes from the testbed. We want to assure that (*i*) the network is widespread enough and not too crowded, but also that (*ii*) it contains multiple bottlenecks as described in Section II-D to stress hop-wise reassembly.

A. SETUP

1) EXPERIMENT TESTBED AND NODE SELECTION

We deploy our experiments on the FIT IoT-LAB testbed and use 50 nodes of the Lille site. These are constrained IoT devices with Cortex-M3 MCUs, 64 kB of RAM, 512 kB of ROM (STM32F103REY), and IEEE 802.15.4 radios (Atmel AT86RF231). The radio chip provides the basic MAC layer features such as CSMA/CA, link layer retransmissions, and acknowledgments.

The Lille site features a challenging multihop network. Nodes are not only distributed in a dedicated room in a grid but also located in multiple offices spread over different floors. The site therefore provides a realistic scenario for different types of heterogeneous deployment. In our experiments we focus on a static network topology. To ensure the static setting, we disabled any routing protocol.

To select nodes for our experiment, we first measure basic properties of the testbed. By correlating the geographic distance and the packet delivery ratio (PDR) between two nodes, we found that two hops should be in range of 6.5 m or less. This ensures that the PDR is at least 97.5%, which we argue is acceptable. Lower PDRs do not contribute to a better understanding of the problem space in this paper. The network is then constructed by a breadth-first search over all available nodes of the testbed site, starting at the sink s. We select node 57 as the sink as it is located centrally between the more crowded nodes in the dedicated room and the more sparse nodes in the office space at the Lille site. This ensures that a balanced set of both network deployment scenarios is included. To prevent a bias towards specific nodes, our network construction algorithm works as follows.

1) Collect all neighbors within the range of 2.2 m and 6.5 m as potential node candidates in set N. This selection expands the network as much as possible under our PDR requirement.
2) Get a randomized, uniformly distributed sample M of 1 to 3 members in N; s always selects 2 neighbors.
3) Add M to the network, and continue for each member of M until 49 nodes are found.

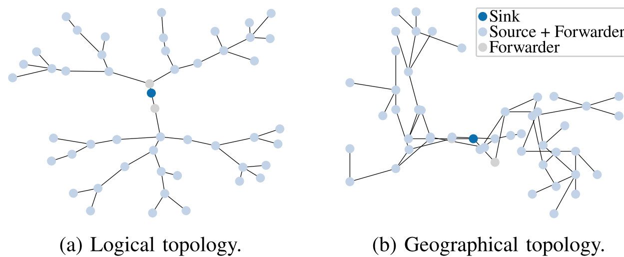
The selection of 1 to 3 downstream neighbors per node assures the inclusion of reassembly bottlenecks into the network, as described in Section II-D.

After constructing the network, we used the same set of nodes in all of our experiments to ensure comparability. The resulting logical and geographical topologies are visualized in Figure 11. Multiple paths have the same length. The longest path consists of 6 hops.

2) COMMUNICATION SETUP

We configured all routes based on the breadth-first search described above. Except for the sink and its neighbors, we configured all other nodes as data senders to ensure the need for forwarding.

All source nodes start sending UDP packets—using the same payload—to the sink in a uniformly distributed interval



(a) Logical topology. (b) Geographical topology.

FIGURE 11. Network topology of the selected testbed.

between 5 s and 15 s. The experiment ends after each source has sent 50 packets.

For each fragment forwarding approach we evaluated as many UDP payload length as possible to see the impact of different packet sizes. As all the fragment forwarding strategies we pick for our evaluation must contain payloads divisible by 8 in all but the last fragment except for SFR, we do not need to cover all payload sizes. As such, to further decrease the overall run-time of our experiments we decided on an increment of 16 for our range of payload size inclusively between 16 and 1024 bytes.

To evaluate the performance, our experiments measure system complexity in terms of memory usage, reliability, specifically the PDR, and the latency between the UDP sockets of source and sink.

3) SOFTWARE PARAMETERIZATION

RIOT offers a variety of compile-time configuration parameters to adapt to use cases. In most of the experiments, we can use default configurations. For the following reasons, however, we have to change some default values: (*i*) The default configurations assume rather small networks. This conflicts with efficient forwarding in large-scale mesh networks, such as the testbed. (*ii*) We originally wanted to compare our results of the fragment forwarding performance with related work that analyzed some aspects in simulation [35]. We document the changes of default values for the `2021.04` release of RIOT in the Appendix.

We also adapted the configuration parameters for selective fragment recovery to the needs of our experiments. In all runs, we selected the number of fragment retransmissions to be four instead of two. For the direct comparison with the other approaches we deactivated congestion control and configured a window size of 1 and 5 respectively to account for different window sizes but negate potential side effects on latency by congestion control.

We will specify further parameters in the evaluation of the specific metric, when they differ from the default configuration.

To evaluate end-to-end transport of the forwarding information in every fragment (E2E) we use a modified version of IPv6 fragmentation (see Section II-B) with 6LoWPAN header compression (see Section II-A). To be comparable to the 6LoWPAN fragmentation approaches we configure the IPv6 MTU to be a non-standard compliant value of the link layer PDU + 22 bytes. Those 22 bytes are the bytes gained by elision of fields due to 6LoWPAN header compression [11]; see Table 2 for a detailed tally up.

TABLE 2. Bytes gained for IPv6 MTU through 6LoWPAN header compression. IPv6 address IIDs and hop limit field cannot be elided in all cases due to the multi-hop environment.

Compressed Field / Dispatch	Bytes elided
IPv6 header compression (IPHC)	
IPHC dispatch	-2
Version, traffic class, and flow label	$+4$
Length	$+2$
Next header (due to NHC being used)	$+1$
Addresses (2×64 bit prefix $\mapsto$ context 0)	$+16$
IPv6 extension next header compression (NHC)	
IPv6 extension header NHC dispatch	-1
Length	$+1$
UDP next header compression (NHC)	
UDP NHC dispatch	-1
Length	$+2$
Sum	$+22$

TABLE 3. Memory sizes [bytes] for source nodes.

Module	HWR		FF		SFR	
	ROM	RAM	ROM	RAM	ROM	RAM
Fragmentation Buffer	104	770	100	1026	104	2562
Reassembly Buffer	1112	176	1554	1840	1144	184
Virtual Reassembly Buffer	n/a	n/a	532	640	532	704
Protocol implementation	5153	388	5865	388	8353	2500
Sum	6369	1334	8051	3894	10133	5950

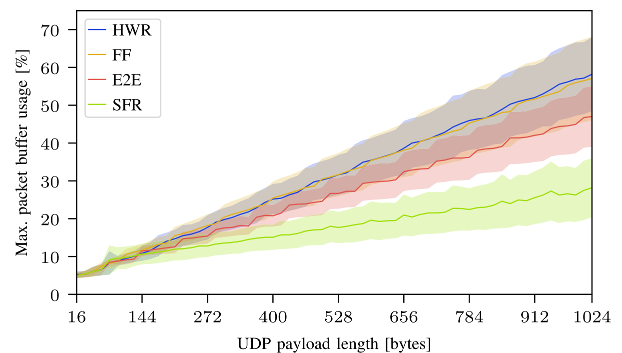


FIGURE 12. Mean packet buffer utilization over each run (window size 1 for SFR). Areas around line represent standard deviation.

The default size of the virtual reassembly buffer in GNRC is 16 entries. Since this only prefers direct forwarding and selective fragment recovery, we do not need to adapt its size. Furthermore, we have to increase the size of the common reassembly buffer of the sink. Without this adaptation the reliability decreases significantly, even for the smallest number of fragments.

B. MEMORY CONSUMPTION

1) OVERALL MEMORY CONSUMPTION

Table 3 shows both ROM and RAM usage of the 6LoWPAN layer at the source node for HWR, FF, and SFR. E2E is not included, as it is implemented in the IPv6 layer and thus

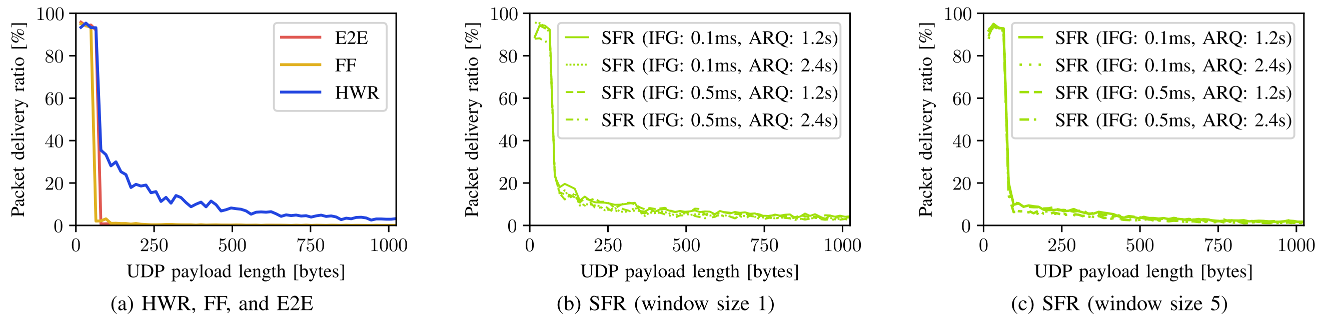


FIGURE 13. Packet delivery ratio for 50 UDP packets sent with a [5, 15] s interval per source (3 runs, IFG: Inter-frame gap, ARQ: ARQ timeout). Split to make different SFR variants more distinguishable.

not comparable. As the fragmentation buffer size is dependent on the window size in SFR, we fixed the window size for that binary to 1 fragment. When compiling the software we use `arm-none-eabi-gcc` v9.3.1 with `-Os` optimization (size-optimal) for ARM Cortex-M3, all debug information stripped, and the compile-time parameters we line out in Section IV-A are applied. We use the `size` tool to extract the relevant module information. To make memory measurements compatible, we set the reassembly buffer size for HWR to the same value as the VRB size (16) for FF and SFR. The anticipated memory advantage for FF does indeed exist, even with the GNRC strategy to not allocate 1280 bytes IPv6 MTU for every reassembly buffer entry but using the central packet buffer instead (*cf.,* Section III).

FF adds a small amount of RAM to keep the meta-data required for refragmentation. Our implementation utilizes (see Section III-B) in the asynchronous GNRC fragmentation buffer to store that meta-data. More ROM is also needed for the possible refragmentation of the first fragment. The majority of the $\approx$ 500 bytes of additional ROM in the reassembly buffer for FF in 6LoWPAN is explained by the overhead required to distinguish whether packets need to be handled by a VRB entry creation or put into the regular reassembly buffer.

SFR of course uses more ROM and RAM (both $\approx$2 kilobytes) than FF, as the recovery mechanism adds extra complexity and requires new data structures to be stored.

2) PACKET BUFFER USAGE

Figure 12 presents our analysis of the mean utilization of the 6144 bytes packet buffer during the overall runtime of each experiment in percent on the y-axis for each evaluated UDP payload size on the x-axis. To show the extremes, for SFR we deliberately used only the window size 1 run. FF and HWR have similar usages, but a slight advantage of $\approx$1% for FF. E2E uses less space in the packet buffer compared to FF and HWR, as the overhead of 6LoWPAN is not required. SFR genuinely mostly sees single fragments instead of full datagrams on most of the nodes due to the window size being 1. As such, its packet buffer usage is the lowest of all approaches with higher fragmentation.

The high packet buffer usage for FF is mostly caused by the fallback to regular reassembly as we describe in more detail in Section IV-C. A clear correlation between this fallback and packet buffer usage was described in [36].

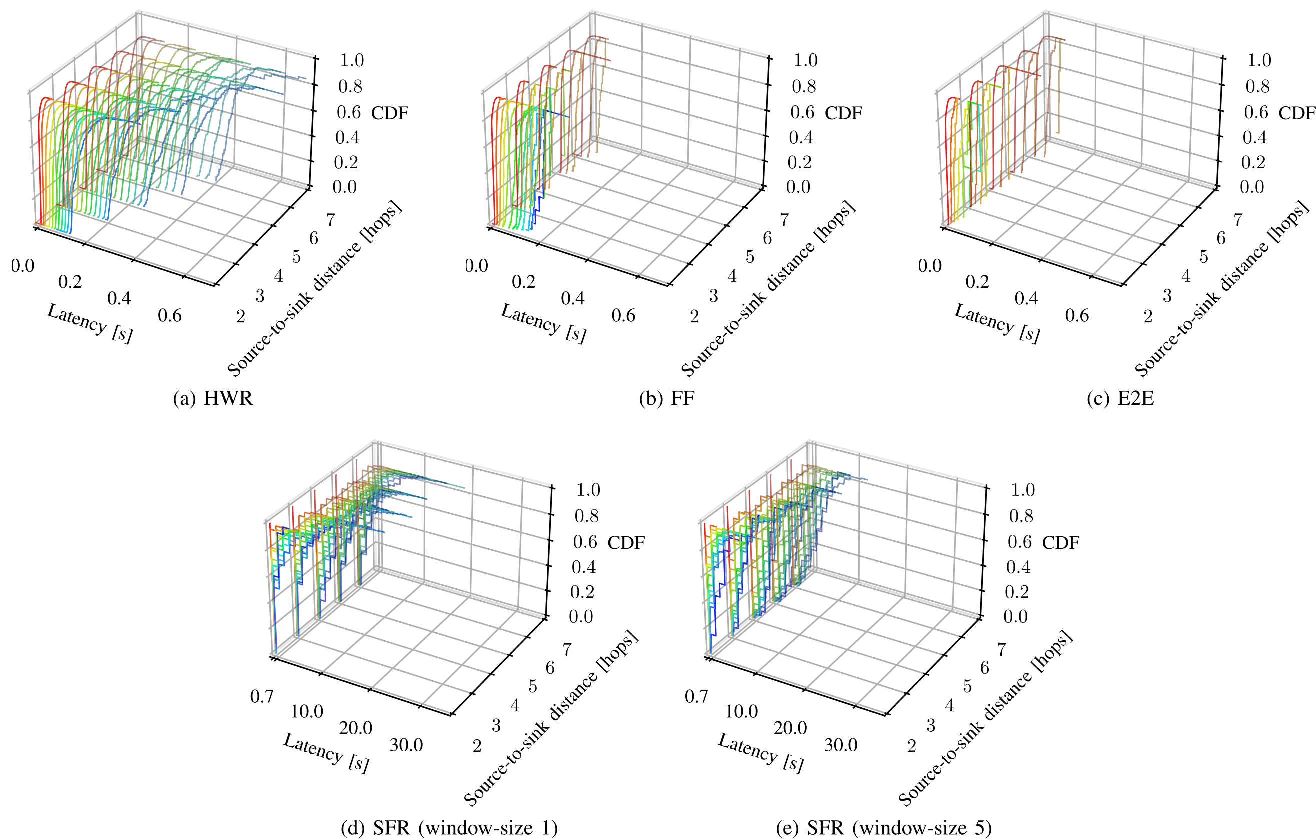
C. RELIABILITY AND LATENCY

Figures 13 and 14 display our results from measuring reliability and latency. Figure 13 shows the reliability in terms of the packet delivery ratio (PDR) depending on the UDP payload length in bytes. To provide a clearer picture, we divided the data sets into three sub-figures: Figure 13a shows the approaches that do not allow for recovery, Figure 13b shows 4 different configurations for SFR with window size 1, and Figure 13c shows the same configurations for SFR with window size 5. The window sizes were chosen to compare between two extremes: window size 1 to measure the effects of waiting for an ACK per fragment (stop-and-wait) and window size 5 to measure the impact of cumulative ACKs. At higher window sizes than 6, we were able to confirm that the PDR dropped drastically in our setup (not shown).

FF and E2E admit poor reliability. For FF this is in contrast to previous results where simulation and a coordinated TSCH MAC layer was used [35]. Even for a small number of fragments, both achieve less than a quarter of the PDR of HWR. The PDR then quickly approach zero with increasing number of fragments. HWR, though also performing poorly, manages to deliver at least some packets to the more distant nodes. The PDR of SFR on the other hand shows comparable to HWR. For window size 1 it is even slightly better than HWR with higher payload length—even if only by 1-2 %, due to the recovery mechanisms of SFR.

Figure 14 depicts the latency as a 3-dimensional CDF, showing the latency in seconds on the x-axis, the CDF on the y-axis, and the source-to-sink distance in hops on the z-axis. The UDP payload lengths are binned together based on the respective number of fragments each protocol requires for that payload length shown in each dedicated plot. Due to its larger latency in SFR, the x-axis is scaled differently and shows 0 to 35 seconds.

The latencies we measured for FF (see Figure 14b) are also significantly higher compared to previous simulation work.



Line color															
Fragments [#]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
UDP payload length [bytes] by variant															
HWR	< 96	< 176	< 288	< 384	< 480	< 576	< 672	< 768	< 864	< 960	≤ 1024				
E2E	< 96	< 144	< 224	< 288	< 368	< 432	< 512	< 576	< 656	< 720	< 800	< 864	< 944	< 1008	≤ 1024
FF	< 64	< 112	< 208	< 304	< 400	< 496	< 592	< 688	< 784	< 880	< 976	≤ 1024			
SFR	< 80	< 160	< 256	< 352	< 448	< 544	< 640	< 736	< 832	< 928	< 1024	$= 1024$			

FIGURE 14. Latency for 50 UDP packets sent with a [5, 15] s interval per source (3 runs). For the SFR results an inter-frame gap of 0.1ms and an ARQ timeout of 1.2s is used.

HWR (see Figure 14a) is expected to operate slower because each node needs to reassemble the entire frame prior to forwarding to the next hop. E2E (see Figure 14c) performs similar to HWR with low fragmentation, but as reliability quickly drops to 0% with higher fragmentation, the latency becomes infinite for higher payloads. SFR (see Figures 14d and 14e) has the highest latency compared to the other protocols, due to its recovery mechanisms and the induced inter-frame gap. In the given plots we only show the results for inter-frame gap 0.1ms and an ARQ timeout of 1.2s. When increasing inter-frame gap and ARQ timeout the latency grows. With a window size of 1 fragment (see Figure 14d) we see significantly higher latencies with higher fragmentation compared to a window size of 5 fragments (see Figure 14e). This is because the sender does not have to wait for an ACK for each fragment, but only every fifth fragment, thus fitting more fragments into a smaller timeframe.

In our experiments, we see significantly more link layer retransmissions per node using FF compared to HWR [36], and the same holds for E2E and SFR (not shown). This is caused by much faster send and receive triggers on the device due to immediate fragment forwarding, which increases collisions and packet loss on the single antenna radio. Moreover, this results in straining the single buffer of a device, which far more often needs to discard unacknowledged incoming packets while it is busy with either sending or receiving a different packet. This invokes link layer retransmissions and eventually contributes to packet loss. An example for these occurrences is illustrated in Figure 15. Based on local measurements using a logic analyzer on a sister device that deploys the same radio (AT86RF233 [37]), we are able to confirm that the device can remain busy for up to 4 ms. Additional measurements on the same device type show that even the shortest fragmented datagram with HWR (88 bytes of UDP payload length) requires ≈8 ms from entering the reassembly buffer to leaving the fragmentation buffer on re-fragmentation. This leaves both the device and the medium more relaxed with HWR.

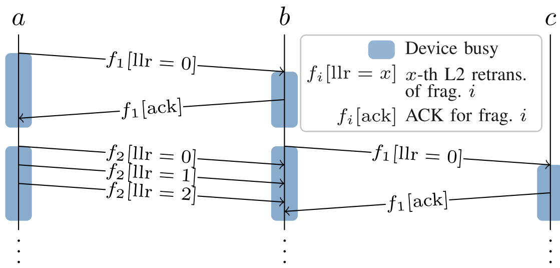


FIGURE 15. Example of L2 retransmissions caused by the device being busy, resulting in missing ACKs.

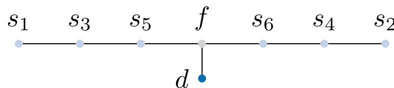


FIGURE 16. Network topography of the selected testbed nodes for congestion control evaluation with d being the sink, s_i being both sources and forwarders and f being only a forwarder.

In previous evaluations [36], [38], it was also shown that packets are lost with FF and SFR when the respective reassembly buffers are full.

V. EVALUATION OF CONGESTION CONTROL WITH SFR

Our evaluation of congestion control with SFR were made in a similar but much smaller setup than we used for comparison of the fragment forwarding methods in Section IV. The forwarding bottlenecks we described in Section II-D also are a point of congestion so the basics of our network topology choices still stand.

Our goal is to explore a suitable congestion control mechanisms for SFR on top of a MAC layer consisting only of CSMA/CS, link layer retransmissions, and acknowledgments by looking into common mechanisms used in the wider Internet.

A. SETUP

1) EXPERIMENT TESTBED

For our congestion control evaluation for SFR we picked a much smaller network than the one used for Section IV to have a better control over how and where congestion occurs. Here we used 8 nodes of the Grenoble side, which are the same type of node as the once at the Lille side we described in Section IV-A. We selected a T-shaped set of nodes (see Figure 16) to assures that the nodes are at least 1 m spaced from another and that the forwarder f would provide a bottleneck for hop-wise reassembly.

2) COMMUNICATION SETUP

We configured all routes statically to form the T-shape described above. Except for the sink and its neighbors, we configured all other nodes as data senders to ensure the need for forwarding.

For our evaluation are only interested in the effects that different number of fragments have on our results. As such, we pick the UDP payload such that SFR increases the number of fragments: We start at 8 bytes for 1 fragment and increment the payload size by 96 bytes for each run, until we reach 1169 bytes, or 13 fragments, just under the IPv6 MTU in IEEE 802.15.4 for of 1232 bytes [7] (1280 bytes minus the 48 bytes for the IPv6 and UDP headers). For each payload we repeat the run 10 times.

To cause as much congestion on the resources of the nodes as possible, all source nodes send UDP packets—using the same payload—to the sink in a uniformly distributed interval between 250 ms and 750 ms. The experiment ends after each source has sent 1200 packets.

3) SOFTWARE PARAMETERIZATION

We mostly use the same software parameterization as in Section IV-A with the difference that congestion control is now activated. For the congestion control comparison, to have a good base-line for all congestion control mechanisms we set the initial window size for SFR to 2. Due to the higher frequency of packets we increase the size of the packet buffer arena of RIOT's default network stack GNRC from 6 to 40 kilobytes. This assures that loss is not happening due to packets not being able to be allocated. While a full packet buffer could be interpreted as a point of congestion, it skews the results especially with larger UDP payloads, where a small packet buffer is filled by only a 2-3 reassembling datagrams at the sink.

B. MEMORY CONSUMPTION AND SYSTEM ANALYSIS

In Figure 17, we compare the build sizes of modules relevant to SFR with CC in our implementation: 6LoWPAN SFR itself, the 6LoWPAN fragmentation buffer (containing state variables for CC), and the CongURE module. To see the cost of abstraction due to the CongURE module, we unrolled each CongURE implementation into the adapter code of the SFR implementation. The binary was generated using `arm-none-eabi-gcc` v9.3.1 with `-Os` optimization (size-optimal) for ARM Cortex-M3, all debug information stripped, and the compile-time parameters we line out in Section IV-A. We use the `size` tool to extract the relevant module information.

All implementations use about 2.5 kbytes of RAM and 4-5 kbytes of ROM, depending on complexity. RAM mostly increases based on the state variables. ROM expectedly increases by a few hundred of bytes with each level of complexity in CC.

Abstraction using the CongURE framework only adds a few bytes of RAM compared to the unrolled version with at most 62 bytes in the QUIC implementation. In ROM the abstraction can add up to about 500 bytes, mostly due to both the indirection introduced and the way CongURE initializes constants in a reusable way.

C. RELIABILITY

To analyze the impact of congestion control, we only looked at PDR as the latency varies quite widely depending on the

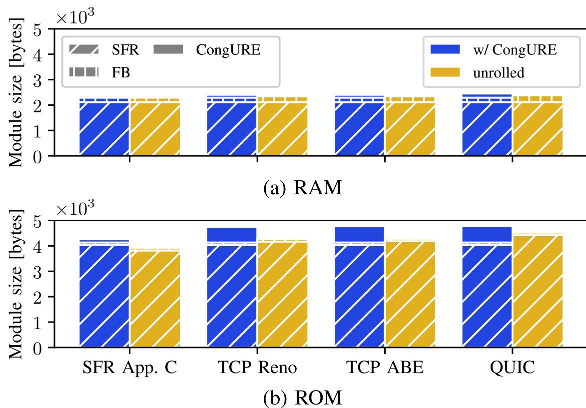


FIGURE 17. Module size in bytes for 6LoWPAN SFR, 6LoWPAN fragmentation buffer (FB) and CongURE compared to CongURE unrolled into SFR.

congestion in the network. Figure 18 shows PDR by number of fragments. Note, that we left out 1 fragment on the x-axis intentionally, as fragmentation is not triggered for a single frame and the far higher PDR would make the remaining results less visible. For reference, see Figures 13b and 13c. However, note that the actual PDRs are not comparable due to the smaller network and the higher sending rate.

While for 2 fragments the performances of all CC mechanisms is comparable around 2%, they vary quite drastically in higher fragmentations: *SFR App. C* only outperforms both TCP variants especially in very low fragmentations, but at 6 fragments it is often comparable or under the two TCP variants. An interesting observation is the saw-tooth like shape of its PDR plot, with even number of fragments being outperformed by the next higher odd number of fragments. This is due to the congestion window defaulting to 2 and only being able to shrink to 1 on timeout with this mechanism: Even numbered fragments oftentimes need a timeout to shrink the window for the transmission to succeed. For odd numbered fragments, on the other hand, this is not necessary for at least the last fragment, when amortizing the transmission sequence.

D. CONGESTION EVENTS

We need to look very selectively on the dataset to analyze the influence on the congestion window and inter-frame gap (IFG) of the congestion events that are reported by the *CongURE* API. Figure 19 shows two selected transmissions for each CC mechanism —one succeeding, one timing out— for the transfer of 12 fragments. We selected the transmission based on the following criteria, in that order, (*i*) having at least two timeouts in the timeout plots to better show its influence on congestion window and inter-frame gap, (*ii*) IFG changes, if any exist, (*iii*) ECN events occurring, if any occurred, and (*iv*) similarity of the course events. We selected s_5 (see Figure 11) for the node under observation, as both s_5 and s_6 show the most ECN events.

For *SFR App. C* we see the expected decrease of the congestion window on timeout in Figure 19a, while on ACK,

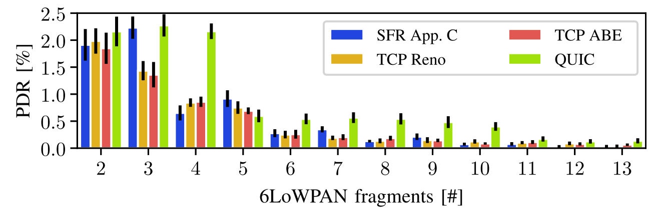


FIGURE 18. SFR packet delivery ratio (PDR) with congestion control for 1200 packets sent with a [250, 750] ms interval per source (10 runs).

it stays at 2 in Figure 19e. As there is no support for pacing, the IFG stays at the default of 170 ms.

For *TCP Reno* and *TCP ABE* (see Figures 19b, 19c, 19f and 19g), we often see spikes in the congestion window, which are explained by the window being increased by ACK'd fragments and then immediately decreased by fragments marked lost within the same ACK. However, ACKs lead to a rapid increase in the window size, which is not as easily mitigated even by the multiplicative decrease due to loss, timeout or ECN. As such, the congestion windows stay rather high, risking a lot of loss, may it be due to congestion or intrinsic loss of the LLNs. Furthermore, we see that both start already with a window size of 4. This is due to the initial window size being defined by the maximum segment size (set to 1 fragment for SFR) in RFC 5681 [23], rather than having a constant initial window size as the other two approaches. This makes both TCP variants rather optimistic, but not very suitable for our scenario. However, with the erratic increase of the congestion window we see with initial window size 4, there is doubt, adopting the mechanism for 6LoWPAN, by setting the window to our default of 2 statically, would make any difference. As with *SFR App. C*, there is no support for pacing, so the IFG stays at the default of 170 ms.

For *QUIC* we saw in Section V-C, that pacing allows for a better PDR. Sadly, for 12 fragments we rarely see the IFG change, as oftentimes we only see one timeout that deletes the fragment buffer, while the other resends are already expended by negative ACKs for the fragment. For the timeout event we were able to see it a few times, for which we present the transmission in Figure 19d. In the success case (see Figure 19h), we see similar behavior as for *TCP ABE*, with the special ECN-behavior giving a slight advantage over *TCP Reno*.

VI. RELATED WORK

In prior work [36], we compare HWR and FF. In this paper, we extend the scope and contribute the comparison of E2E fragmentation and SFR as well as the evaluation of different congestion control mechanisms in the context of fragmentation. This comprehensive analysis allows us to guide discussions why the use of direct forwarding approaches should be discouraged in scenarios that deploy very thin MAC layers.

Kent *et al.* [39], [40] discussed potential harms already at the beginning of the Internet and thus paved the way for proper protocol design.

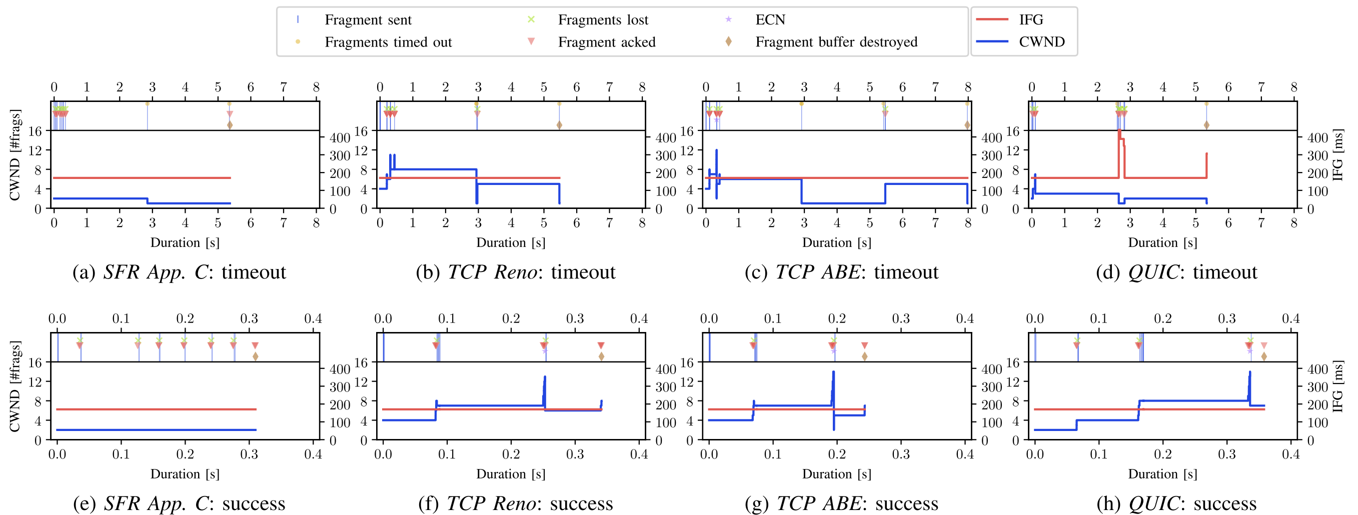


FIGURE 19. Consistent selection of datagram transmissions on one of the forwarding sources at 12 fragments with their respective congestion events (50% opacity, *i.e.,* darker events mean multiple rapidly repeating events), congestion window (CWND) and calculated or static inter-frame gap (IFG). For each congestion control mechanism two transmissions are shown, one succeeding and one timing out. The selection was made based on (*i*) by at least two timeouts in the timeout graphs, (*ii*) by, if any, IFG changes (*iii*) ECN events occurring, if any, and (*iv*) similar event occurrences, in descending priority.

Papadopoulos *et al.* [41] discussed the ongoing work within the IETF on 6LoWPAN fragment forwarding and selective fragment recovery already. In contrast to their work, our work provides an in-depth analysis of these approaches.

Other approaches that use similar concepts as FF mainly focus on datagram prioritization [42], [43]. Similar to SFR, Chowdhury *et al.* [44] proposed a standard compliant NACK-based approach for selective fragment recovery. Since those NACKs, however, are associated with the datagram identifier $(a, g_a(D))$ (see Section II-C), this mechanism only allows for hop-wise recovery and does not cover the whole end-to-end path when using FF.

Tanaka *et al.* [35] used the 6TiSCH simulator [45] to analyze the performance of FF compared to HWR. The authors show that FF is a promising option in IEEE 802.15.4e (TSCH) and that as such also SFR could yield better results.

Awwad *et al.* [46] also compared FF to HWR in a simulator and conducted experiments in a testbed. They used a topology consisting of 4 nodes in a line. This setup ignores challenging bottlenecks, which occur in real deployments (see Section II-D). Furthermore, they only compared their proprietary solution of fragment forwarding with HWR in the testbed evaluation. In contrast to this, we evaluate standard compliant protocols in a complex testbed setup.

In our work, we did not consider the frame delivery mode for link-layer meshes of 6LoWPAN [7]—commonly known as *mesh-under* [8, Section 1.2]—because it is known that such a solution falls behind HWR [44].

Hummen *et al.* [47] analyzed the security implications of 6LoWPAN fragmentation. They considered both hop-wise reassembly and direct fragment forwarding, but also mesh-under mode. In their paper they present 2 possible attack vectors utilizing 6LoWPAN fragmentation: A fragment duplication attack and a buffer reservation attack. Buffer reservation attack only effects not performing hop-wise reassembly and the reassembling end-points. The attacker spams the victim with first fragments with changing datagram tags, so that the reassembly buffer is quickly exhausted. All fragment forwarding schemes are however susceptible to a fragment duplication attack in which the attacker sends bogus subsequent fragments that identify to belong to a different fragment chain, effectively invalidating the data within the reassembled datagram. As a solution the authors propose an extension to 6LoWPAN fragmentation utilizing content chaining against fragment duplication attacks and a split reassembly buffer approach against buffer reservation attacks.

Combining fragmentation, selective acknowledgments, and congestion control is an approach novel to 6LoWPAN Selective Fragment Recovery. As such, not much research considered that combination yet.

VII. DISCUSSION

A. IS FRAGMENT FORWARDING A VIABLE OPTION WITHOUT A COORDINATED MAC LAYER?

Our testbed experiments clearly indicate that direct fragment forwarding is outperformed by HWR if direct fragment forwarding is based on the widely deployed, very thin CSMA/CA MAC layer. Our results systematically confirm prior assumptions [9] in practice. Radios that are busy sending a fragment are not able to listen for the next fragment, leading to high losses. In contrast to this, HWR allows for a datagram to be received fully, causing the radio to be in receive mode first and only to switch to send mode when the node is fragmenting the datagram again. In [35] the authors show that with direct fragment forwarding an advantage can be gained when using a coordinated MAC layer such as

IEEE 802.15.4e TSCH. The only advantage of FF over CSMA/CA we could clearly identify is its reduced RAM consumption.

SFR reduces the bottleneck of HWR while providing a similar packet delivery ratio. Using lower window sizes may improve the results even further. However, a higher latency is to be expected due to the recovery mechanism.

In practice, whether fragment forwarding is applicable and if so on which MAC protocol depends on the deployment scenarios and use cases. Our work helps to assess the potential deployment space.

B. IS END-TO-END FRAGMENTATION A VIABLE SOLUTION?

In our E2E setup using a modified version of IPv6 fragmentation, we show that the performance disadvantages cannot be reduced when carrying the forwarding information in every fragment. The only advantage we observe is a smaller memory footprint due to the VRB and the 6LoWPAN fragmentation layer, which are not required, on the cost of using bytes of the link layer PDU to carry the forwarding information. At least for classic IEEE 802.15.4 where the link layer PDU is very restricted, this cost is not negligible. Larger SDUs (*e.g.,* 400 bytes in the PLC protocol ITU-T G.9903) might not have to deal with these restrictions.

It should be noted, however, that we used IPv6 fragmentation in a very ideal configuration in our experiments. Other extension headers such as routing headers or hop-by-hop options might also need to be carried in every fragment. While the 6LoWPAN Routing Header specified in RFC 8138 [48] may be able to compress some of those, the remaining headers can quickly exceed an MTU that is smaller than the required 1280 bytes specified in [6].

C. WHAT KIND OF CONGESTION CONTROL SHOULD BE USED WITH SFR?

Providing optional congestion control in SFR has the potential to provide a good balance between high packet delivery ratio and low latency. We showed that a highly sophisticated congestion control mechanism utilizing pacing, such as the one used by QUIC, can yield benefits for SFR. Some adaptations tailored to LLNs are required, though: The window size should be increased very conservatively and the initial window size should be kept as minimal as possible.

Using ECN has little impact on the overall performance: Both TCP variants perform very similar, despite *TCP ABE* using an exponential backoff for the window decreases on ECN. This is surprising as the common metric, packet loss, does not serve as a good heuristic to detect congestion in *lossy* and low-power networks. Explicit congestion indicators such as ECN would have been expected to improve the situation when congestion occurs in the network. However, the very nature of LLNs weakens this advantage, since ECN in SFR is marked in the fragments by the forwarders on congestion but delivered to the fragmenting end-point in the ACK. Both messages can easily get lost after the ECN flag was set in the headers. In fact, we observed the ECN flag being set on the forwarders in *SFR App. C*, but we did not observe a corresponding ECN event at the fragmenting end-point.

D. HOW DOES FRAGMENTATION WORK WITH OTHER LINK LAYER TECHNOLOGIES?

Bluetooth Low Energy (BLE) provides fragmentation and reassembly, implemented in its Logical Link Control and Adaption Protocol (L2CAP). It is comparable to a hop-wise SFR. This means nodes face not only similar bottleneck problems compared to HWR but also latency issues as introduced by SFR. However, the hop-wise reassembly occurs much closer to the device and within the coordinated MAC layer of Bluetooth Low Energy. As such, buffer space and air time can be allocated much more tailored to the specific datagram. Since acknowledgments are already part of the link layer, the latency issues we saw in this paper might be much smaller. Due to the coordinated MAC layer in BLE, all remaining latency penalties because of ACKs might also be negligible when compared to a similar approach in IEEE 802.15.4e TSCH (6TSCH) or DSME [26]. Other low latency modes of 802.15.4e such as LLDN or LECIM only allow for star topologies, so they do not fit our initial use case for fragment forwarding.

It should also be noted that for low-power wide area networks (LPWANs) such as LoRAWAN, SigFox, or NB-IoT deployment scenarios face much more extreme constraints than in most other low-power and lossy network technologies [5]. As such, the IETF specified a whole different suite of IPv6-over-X adaptation layer protocols—Static Context Header Compression and Fragmentation (SCHC) [14]—which is not compatible with 6LoWPAN. SCHC introduces different fragmentation schemes, tailored to the underlying link layer technologies, called SCHC Fragmentation/Reassembly (SCHC F/R). SCHC uses fragment forwarding but very tailored to the infrastructure of the underlying LPWAN. As such the forwarding can be much more reliable. In [14] three SCHC F/R modes are defined. Those may be used for different link layer technologies (*i*) *No-ACK mode*, which is comparable to FF in this paper, (*ii*) *ACK-Always mode*, which is comparable to SFR in this paper, and (*iii*) *ACK-on-Error mode*, which is a NACK approach, comparable to the one proposed in [44]. Later SCHC extensions might add more modes.

All the proposed modes are comparable to approaches evaluated in this paper. This is the reason why we argue that the same conclusions apply for the different SCHC modes. This is bolstered in RFC 8724 [14], which clearly states that the selection of the mode is dictated by the underlying link layer.

VIII. CONCLUSION AND OUTLOOK

In this paper, we compared four different fragment forwarding schemes—end-to-end fragmentation, hop-wise reassembly, direct fragment forwarding, and selective fragment recovery—using large real-world experiments, as well as

congestion control mechanisms for selective fragment recovery. We showed that hop-wise reassembly is the preferred choice to achieve proper reliability and latencies, even if deployed on top of a thin MAC layer. Our careful analysis reveals that the radio of a node cannot handle the fast reception and sending of fragments, which occurs in case of direct fragment forwarding but is far more relaxed in hop-wise reassembly.

Strictly coordinating MAC layer schemes such as Time-Slotted Channel Hopping (TSCH) or the Deterministic and Synchronous Multi-channel Extension (DSME) of 802.15.4e have the potential to improve reliability without sacrificing data rates. Prior work [35] focused on simulating TSCH in small topologies of ten nodes but showed promising results for that MAC layer using direct fragment forwarding. Integrating those access schemes into existing 6LoWPAN implementations to analyze whether they can cope with large-scale sender scenarios will be part of our future work.

A NOTE ON REPRODUCIBILITY

We explicitly support reproducible research [31], [32]. Our experiments have been conducted in an open testbed. The source code of our implementations (including scripts to set up the experiments, RIOT measurement apps *etc.*) is published under doi:10.5281/zenodo.5575035.

APPENDIX A: ARTIFACTS

Our implementations have been provided to the RIOT community. Fragment forwarding and selective fragment recovery are officially supported in RIOT since release 2021.01. The CongURE framework is officially supported in RIOT since release 2021.04. Open Pull requests are available via the following URLs:

App. C: https://github.com/RIOT-OS/RIOT/pull/16158
QUIC: https://github.com/RIOT-OS/RIOT/pull/16159
Reno: https://github.com/RIOT-OS/RIOT/pull/16170
ABE: https://github.com/RIOT-OS/RIOT/pull/16171

APPENDIX B: COMPILE TIME PARAMETERS IN RIOT

TABLE 4. Changed compile time parameters in RIOT 2021.04. Values with asterisk (*) only apply for the congestion control evaluation in Section V.

Compile-time Configuration Parameter	Value
CONFIG_GNRC_IPV6_EXT_FRAG_RBUF_SIZE	1 (source) / 16 (sink)
CONFIG_GNRC_IPV6_EXT_FRAG_SEND_SIZE	64 / 4*
CONFIG_GNRC_IPV6_EXT_FRAG_RBUF_TIMEOUT_US	10000000
CONFIG_GNRC_IPV6_NIB_NO_RTR_SOL	1
CONFIG_GNRC_NETIF_PKTQ_POOL_SIZE	64 / 4*
CONFIG_GNRC_SIXLOWPAN_FRAG_RBUF_DEL_TIMER	250
CONFIG_GNRC_SIXLOWPAN_FRAG_FB_SIZE	64
CONFIG_GNRC_SIXLOWPAN_FRAG_RBUF_DO_NOT_OVERRIDE	1
CONFIG_GNRC_SIXLOWPAN_FRAG_RBUF_SIZE	1 (source) / 16 (sink)
CONFIG_GNRC_SIXLOWPAN_FRAG_RBUF_TIMEOUT_US	10000000
CONFIG_GNRC_SIXLOWPAN_SFR_FRAG_RETRIES	4
CONFIG_GNRC_SIXLOWPAN_SFR_INTER_FRAME_GAP_US	17000*
CONFIG_GNRC_SIXLOWPAN_SFR_OPT_ARQ_TIMEOUT_MS	2500*

ACKNOWLEDGMENT

The authors would like to thank Jakob Pfender and the anonymous reviewers for their feedback on this paper.

REFERENCES

[1] *IEEE 802.15 Working Group, IEEE Standard for Local and Metropolitan Area Networks—Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE Standard 802.15.4–2015, New York, NY, USA, Apr. 2016.

[2] J. Hou, B. Liu, Y.-G. Hong, X. Tang, and C. Perkins, *Transmission of IPv6 Packets Over PLC Networks*, document IETF, Internet-Draft–Work in Progress 06, Apr. 2021.

[3] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby, and C. Gomez, *IPv6 Over BLUETOOTH(R) Low Energy*, document IETF, RFC 7668, Oct. 2015.

[4] L. Alliance, ''LoRaWAN 1.0.3 regional parameters,'' LoRa Alliance, Fremont, CA, USA, Tech. Rep. RP002-1.0.0, Jul. 2018.

[5] S. Farrell, *Low-Power Wide Area Network (LPWAN) Overview*, document IETF, RFC 8376, May 2018.

[6] S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, document IETF, RFC 8200, Jul. 2017.

[7] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, *Transmission of IPv6 Packets Over IEEE 802.15.4 Networks*, document IETF, RFC 4944, Sep. 2007.

[8] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*, 1st ed. Hoboken, NJ, USA: Wiley, 2009.

[9] T. Watteyne, P. Thubert, and C. Bormann, *On Forwarding 6LoWPAN Fragments Over a Multi-Hop IPv6 Network*, document IETF, RFC 8930, Nov. 2020.

[10] S. L. Keo, S. S. Komar, and H. Tschofenig, ''Securing the Internet of Things: A standardization perspective,'' *IEEE Internet Things J.*, vol. 1, no. 3, pp. 265–275, May 2014.

[11] J. Hui and P. Thubert, *Compression Format for IPv6 Datagrams Over IEEE 802.15.4-Based Networks*, document IETF, RFC 6282, Sep. 2011.

[12] C. Bormann, *6LoWPAN-GHC: Generic Header Compression for IPv6 Over Low-Power Wireless Personal Area Networks (6LoWPANs)*, document IETF, RFC 7400, Nov. 2014.

[13] J. Ziv and A. Lempel, ''A universal algorithm for sequential data compression,'' *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[14] A. Minaburo, L. Toutain, C. Gomez, D. Barthel, and J. Zúñiga, *SCHC: Generic Framework for Static Context Header Compression and Fragmentation*, document IETF, RFC 8724, Apr. 2020.

[15] O. Gimenez and I. Petrov, *Static Context Header Compression and Fragmentation (SCHC) Over LoRaWAN*, document IETF, RFC 9011, Apr. 2021.

[16] J. Zuniga, C. Gomez, S. Aguilar, L. Toutain, S. Cespedes, and D. Torre, *SCHC Over Sigfox LPWAN*, document IETF, Internet-Draft–Work in Progress 07, Jul. 2021.

[17] E. Ramos and A. Minaburo, *SCHC Over NB-IoT*, document IETF, Internet-Draft–Work in Progress 05, Jul. 2021.

[18] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann, *Neighbor Discovery Optimization for IPv6 Over Low-Power Wireless Personal Area Networks (6LoWPANs)*, document IETF, RFC 6775, Nov. 2012.

[19] C. Bormann, M. Ersue, and A. Keranen, *Terminology for Constrained-Node Networks*, document IETF, RFC 7228, May 2014.

[20] P. Thubert, *IPv6 Over Low-Power Wireless Personal Area Network (6LoWPAN) Selective Fragment Recovery*, document IETF, RFC 8931, Nov. 2020.

[21] G. Z. Papadopoulos, P. Thubert, S. Tsakalidis, and N. Montavont, ''RFC 4944: Per-hop fragmentation and reassembly issues,'' in *Proc. IEEE Conf. Standards Commun. Netw. (CSCN)*, Oct. 2018, pp. 1–6.

[22] C. Bormann and T. Watteyne, *Virtual Reassembly Buffers in 6LoWPAN*, document IETF, Internet-Draft–Work in Progress 02, Mar. 2020.

[23] M. Allman, V. Paxson, and E. Blanton, *TCP Congestion Control*, document IETF, RFC 5681, Sep. 2009.

[24] N. Khademi, M. Welzl, G. Armitage, and G. Fairhurst, *TCP Alternative Backoff With ECN (ABE)*, document IETF, RFC 8511, Dec. 2018.

[25] J. Iyengar and I. Swett, *QUIC Loss Detection and Congestion Control*, document IETF, RFC 9002, May 2021.

[26] D. De Guglielmo, S. Brienza, and G. Anastasi, ''IEEE 802.15.4e: A survey,'' *Comput. Commun.*, vol. 88, pp. 1–24, Aug. 2016.

[27] P. Thubert, *An Architecture for IPv6 Over the Time-Slotted Channel Hopping Mode of IEEE 802.15.4 (6TiSCH)*, document IETF, RFC 9030, May 2021.

[28] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "OpenWSN: A standards-based low-power wireless development environment," *Trans. Emerg. Telecommun. Technol.*, vol. 23, no. 5, pp. 480–493, 2012.

[29] M. Köstler, F. Kauer, T. Lübkert, and V. Turau, "Towards an open source implementation of the IEEE 802.15. 4 DSME link layer," in *Proc. GI/ITG KuVS Fachgespräch Sensornetze*. Augsburg, Germany: Univ. of Applied Sciences Augsburg, Dept. of Computer Science, Sep. 2016, pp. 1–4.

[30] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An open source operating system for low-end embedded devices in the IoT," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018, doi: 10.1109/JIOT.2018.2815038.

[31] Q. Scheitle, M. Wählisch, O. Gasser, T. C. Schmidt, and G. Carle, "Towards an ecosystem for reproducible research in computer networking," in *Proc. Reproducibility Workshop*, New York, NY, USA, Aug. 2017, pp. 5–8.

[32] ACM. (Jan. 2017). *Result and Artifact Review and Badging*. [Online]. Available: http://acm.org/publications/policies/artifact-review-badging

[33] *Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE, SP100, WirelessHART, ISM Appl. (AT86RF231)*, document Rev.8111C, Microchip, Sep. 2009.

[34] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, pp. 314–329, Aug. 1988.

[35] Y. Tanaka, P. Minet, and T. Watteyne, "6LoWPAN fragment forwarding," *IEEE Commun. Standards Mag.*, vol. 3, no. 1, pp. 35–39, Mar. 2019.

[36] M. S. Lenders, T. C. Schmidt, and M. Wahlisch, "A lesson in scaling 6LoWPAN–minimal fragment forwarding in lossy networks," in *Proc. IEEE 44th Conf. Local Comput. Netw. (LCN)*, Piscataway, NJ, USA, Oct. 2019, pp. 438–446, doi: 10.1109/LCN44214.2019.8990812.

[37] *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, ISM Appl. (AT86RF233)*, Rev. 8315E, Microchip, Jul. 2014.

[38] M. S. Lenders, C. Gündoğan, T. C. Schmidt, and M. Wählisch, "Connecting the dots: Selective fragment recovery in ICNLoWPAN," in *Proc. 7th ACM Conf. Inf.-Centric Netw.*, New York, NY, USA, Sep. 2020, pp. 70–76, doi: 10.1145/3405656.3418719.

[39] C. A. Kent and J. C. Mogul, *Fragmentation Considered Harmful*, vol. 17, no. 5. Palo Alto, CA, USA: Western Research Laboratory, 1987.

[40] C. A. Kent and J. C. Mogul, "Fragmentation considered harmful," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 1, pp. 75–87, Jan. 1995, doi: 10.1145/205447.205456.

[41] G. Z. Papadopoulos, R. A. Jadhav, P. Thubert, and N. Montavont, "Updates on RFC 4944: Fragment forwarding and recovery," *IEEE Commun. Standards Mag.*, vol. 3, no. 2, pp. 54–59, Jun. 2019.

[42] A. Weigel, "Forwarding strategies for 6LoWPAN-fragmented IPv6 datagrams," Ph.D. dissertation, Inst. Telematics, Technische Universität Hamburg-Harburg, Hamburg, Germany, 2017.

[43] A. Weigel, M. Ringwelski, V. Turau, and A. Timm-Giel, "Route-over forwarding techniques in a 6LoWPAN," in *Proc. Int. Conf. Mobile Netw. Manage.* Cham, Switzerland: Springer, 2013, pp. 122–135.

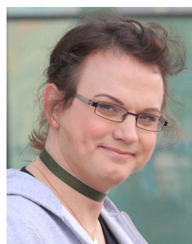
[44] A. H. Chowdhury, M. Ikram, H.-S. Cha, H. Redwan, S. M. S. Shams, K.-H. Kim, and S.-W. Yoo, "Route-over vs mesh-under routing in 6LoWPAN," in *Proc. Int. Conf. Wireless Commun. Mobile Comput. Connecting World Wirelessly (IWCMC)*, 2009, pp. 1208–1212.

[45] E. Municio, G. Daneels, M. Vučinić, S. Latré, J. Famaey, Y. Tanaka, K. Brun, K. Muraoka, X. Vilajosana, and T. Watteyne, "Simulating 6TiSCH networks," *Trans. Emerg. Telecommun. Technol.*, vol. 30, no. 3, p. e3494, Mar. 2019.

[46] S. A. B. Awwad, N. K. Noordin, B. M. Ali, F. Hashim, and N. H. A. Ismail, "6LoWPAN route-over with end-to-end fragmentation and reassembly using cross-layer adaptive backoff exponent," *Wireless Pers. Commun.*, vol. 98, no. 1, pp. 1029–1053, Jan. 2018.

[47] R. Hummen, J. Hiller, H. Wirtz, M. Henze, H. Shafagh, and K. Wehrle, "6LoWPAN fragmentation attacks and mitigation mechanisms," in *Proc. 6th ACM Conf. Secur. Privacy Wireless Mobile Netw. (WiSec)*, New York, NY, USA, 2013, pp. 55–66, doi: 10.1145/2462096.2462107.

[48] P. Thubert, C. Bormann, L. Toutain, and R. Cragie, *IPv6 Over Low-Power Wireless Personal Area Network (6LoWPAN) Routing Header*, document IETF, RFC 8138, Apr. 2017.



MARTINE S. LENDERS received the B.Sc. and M.Sc. degrees in computer science from Freie Universität Berlin, in 2011 and 2016, respectively, with a focus on the development and comparison of several IP-based network stacks for embedded devices, where she is currently pursuing the Ph.D. degree. She is currently a Researcher with the Internet Technologies Research Group, Freie Universität Berlin. She is also one of core developers of the operating system RIOT and maintains large parts of the RIOT networking infrastructure. Her research interests include the Internet of Things, information-centric networking, and API design.

THOMAS C. SCHMIDT (Member, IEEE) received the Ph.D. degree from FU Berlin, in 1993. He is currently a Professor of computer networks and internet technologies at the Hamburg University of Applied Sciences (HAW), where he heads the Internet Technologies Research Group (iNET). Prior to moving to Hamburg, he was the Director of the Scientific Computer Centre, Berlin. He studied mathematics, physics, and German literature at Freie Universitaet Berlin and the University of Maryland. Since then, he has continuously conducted numerous national and international research projects. He was a principal investigator in a number of EU, nationally funded and industrial projects as well as a Visiting Professor at the University of Reading, U.K. His interests include in the development, measurement, and analysis of large-scale distributed systems, like the Internet. He serves as a co-editor and a technical expert for many occasions and actively involved in the work of IETF and IRTF. Together with his group, he pioneered work on an information-centric Industrial IoT and the emerging data-centric Web of Things. He is the co-founder of several large open source projects and the coordinator of the community developing the RIOT operating system—the friendly OS for the Internet of Things.

MATTHIAS WÄHLISCH (Member, IEEE) graduated from Freie Universität Berlin after studying computer science and contemporary German literature. He received the Ph.D. degree (Hons.) in computer science, in 2016. He is currently an Assistant Professor of computer science at Freie Universität Berlin and heads the Internet Technologies Group. His research and teaching focus on efficient, reliable, and secure internet communication. This includes the design and evaluation of networking protocols and architectures, as well as internet topology measurements and analysis. He has published more than 150 peer-reviewed papers. He has been active in internet standardization, since 2005, and co-founded some successful open source projects in the Internet of Things (RIOT) and secure internet routing (RTRlib), where he is still responsible for the strategic development. His research results have been distinguished multiple times, such as the Best of ACM SIGCOMM CCR (2018 and 2019).