A Lesson in Scaling 6LoWPAN – Minimal Fragment Forwarding in Lossy Networks

Martine S. Lenders
Freie Universität Berlin
m.lenders@fu-berlin.de

Thomas C. Schmidt
HAW Hamburg
t.schmidt@haw-hamburg.de

Matthias Wählisch
Freie Universität Berlin
m.waehlisch@fu-berlin.de

***Abstract*—This paper evaluates two forwarding strategies for fragmented datagrams in the IoT: hop-wise reassembly and a minimal approach to direct forwarding of fragments. Direct fragment forwarding is challenged by the lack of forwarding information at subsequent fragments in 6LoWPAN and thus requires additional data at nodes. We compare the two approaches in extensive experiments evaluating reliability, end-to-end latency, and memory consumption. In contrast to previous work and due to our real-world testbed setup, we obtained different results and conclusions. Our findings indicate that direct fragment forwarding should be deployed with care, since higher packet transmission rates on the link layer can significantly reduce its reliability, which in turn can even further reduce end-to-end latency because of highly increased link layer retransmissions.**

***Index Terms*—Embedded networks, Internet of Things (IoT), Fragmentation**

I. Introduction

The advent of the Internet of Things (IoT) increased deployment of resource constrained wireless devices in a rapidly growing market. Always connected sensors and actuators advance business models concerning new products, process innovations, and data. Wireless operators have already started the wide-area outreach to the embedded edge, which facilitates operation of IoT gateways in the wild. Foreseeably, 5G technologies appear on the horizon with the promise of tailored technologies that can host vertical networks towards their users. Such vertical networks, or network slices, will allow public or private bodies and companies to create their own private 5G-based networks on site. This current trend will foster a strong increase of heterogeneous devices that join the wider Internet, but also a significantly widened range of heterogeneous access networks.

Besides the wireless IoT, other access technologies such as Power-line Communication (PLC) gather deployment, while offering a wide range of packet sizes [1]. These different technologies introduce a wide variety of maximum packet sizes in the link layer as visualized in Figure 1. On the network layer, nodes predominantly speak IPv6 [2] with a mandatory transparent Maximum Transmission Unit (MTU) size of at least 1280 bytes. Hence, fragmentation is necessary in order to communicate using these link layer technologies.

Some of these links—e.g. IEEE 802.15.4 [3]—only support a very limited number of bytes. For efficiency, information required to forward a packet cannot be encoded in every fragment but is only present in the first [4]. This is in contrast to transparent fragmentation such as in the IP protocols. Since many IoT networks form meshes, however, forwarding of packets is needed, and there are two concepts for forwarding fragmented datagrams. First, reassembly is performed at every hop (*hop-wise reassembly*), followed by re-fragmentation when forwarded on another constrained link. This is the simplest solution, due to the forwarding information only being stored in the first fragment. 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 [5, Section 2.5.2], [6].

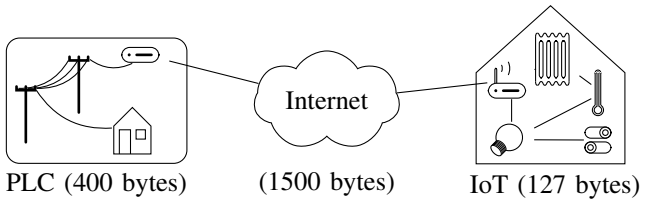


Figure 1. A typical scenario where datagram fragmentation is needed because of different maximum packet sizes.

Both approaches—hop-wise reassembly and fragment forwarding—have advantages and disadvantages. While direct forwarding can lead to lower latency, it also sends more packets on average over time, leading to a higher load on the medium. Hop-wise reassembly on the other hand is part of common network stacks, which can be a benefit on more constrained nodes, where program memory is scarce [7].

In this paper, we comparatively assess the performance and resource consumption of *hop-wise reassembly* and direct *fragment forwarding* over a thin IEEE 802.15.4 MAC layer. Our findings are ambivalent and reveal two sides of the coin. Depending on the MAC layer and packet frequency, hop-wise reassembly may perform much better than the prospective optimization introduced with fragment forwarding. Conversely, MAC layers with a slow coordinative function like IEEE 802.15.4e can profit from fragment forwarding. As part of this work, we also provide an independent implementation of fragment forwarding, which we showcase to allow deeper insights into our evaluation results.

The remainder of this paper is structured as follows. In Section II, the background of 6LoWPAN fragmentation and

forwarding is recapitulated along with related work. Section III describes our implementation for fragment forwarding, with which we obtain the results presented in Section IV. We discuss our findings in Section V, and close with a conclusion and an outlook in Section VI.

II. Problem Statement and Related Work

The IETF specified the 6LoWPAN protocol [4] to allow for transmissions of IPv6 packets over IEEE 802.15.4 [3] networks—a widely used link layer technology in the IoT. While IPv6 requires a Maximum Transmission Unit (MTU) of at least 1280 bytes [2], IEEE 802.15.4 is only able to handle link layer packets of up to 127 bytes—*including* the link layer header—which in the worst case only leaves 33 bytes for application data [8]. To enable IPv6 communication in such a restrictive environment, 6LoWPAN provides both header compression [9], [10] and datagram fragmentation [4]. The latter is the focus of this paper.

For completeness we note that the concept of 6LoWPAN (or more generally *6Lo*) is not limited to IEEE 802.15.4, but also can be used in other link-layer technologies such as PLC [1].

A. Basic Fragmentation and Reassembly in 6LoWPAN

In 6LoWPAN, datagram fragmentation implements the following common approach: 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. Each fragment includes a fragment header containing information to assemble the datagram [4]:

The fragmentation header of the first fragment contains an (uncompressed) datagram size in bytes as an 11-bit number and a 16-bit datagram tag to identify the fragment on the link. All subsequent fragments carry in addition to the header fields of the first fragment header an offset to this fragment in units of 8 bytes, see Figure 2. 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 three values: (*i*) the link layer source and destination addresses, (*ii*) the datagram size, and (*iii*) the datagram tag. Then, the receiver network stack stores all fragments of an incoming datagram in the *reassembly buffer* for up to 10 seconds. These identifying parameters to assign fragments to a datagram i we will refer to by $\mathbf{id}(i)$ in the following.

A brief back-of-the-envelope calculation shows that a node needs to allocate at least **1302 bytes** of memory **per reassembly buffer entry** to reassemble a fragmented datagram:

- At most **8 bytes** per address, plus **1 byte** per address to store their length as IEEE 802.15.4 supports both 64-bit EUI-64s and a 16-bit short addresses as addressing format,

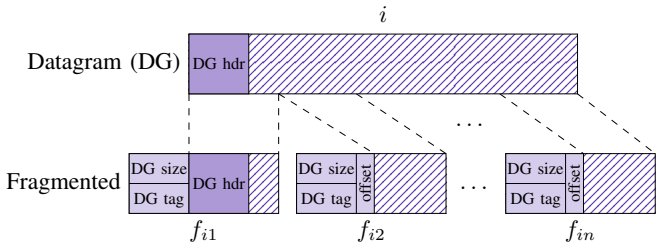


Figure 2. Fragmentation in 6LoWPAN.

- **2 bytes** for the datagram size,
- **2 bytes** for the datagram tag, and
- **1280 bytes** for the maximum expected size of an IPv6 datagram.

1302 bytes are significant memory requirements on constrained devices, which typically offer memory within the range of several kilobytes [7]. Especially in a multihop network—a common deployment scenario in the 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.

B. Fragment Forwarding for Low-power Lossy Networks

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)* [2], [4]. 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 4). Second, the memory requirements are unbalanced between nodes in the network. Considering highly connected nodes (see node e in Figure 3), 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.* [11] underscored these problems in more detail.

Fragment forwarding (FF) [6] tackles the drawbacks of HWR by leveraging a *virtual* reassembly buffer *(VRB)* [12], see Figure 5. In contrast to a reassembly buffer, a VRB only

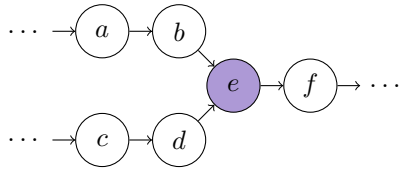


Figure 3. e represents a typical bottleneck for HWR.

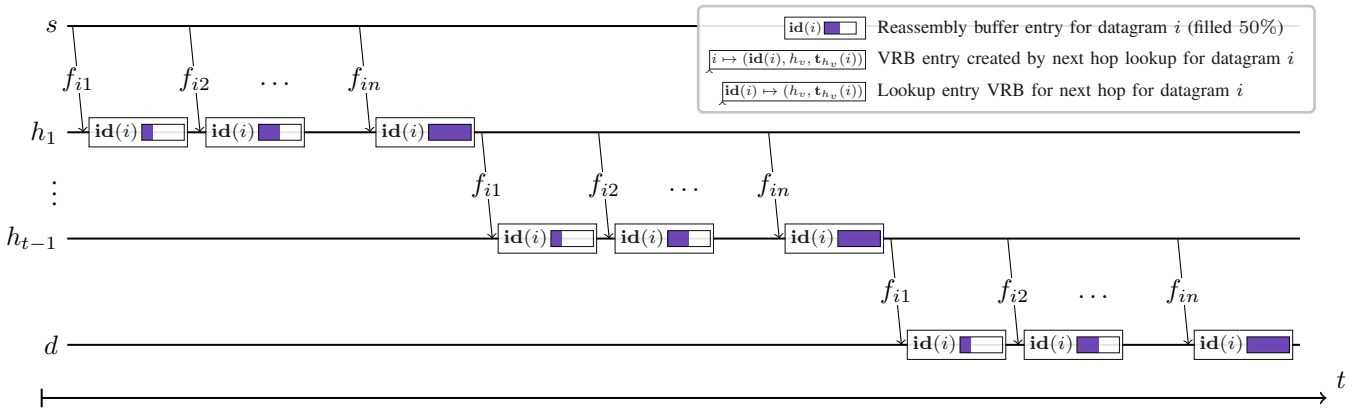


Figure 4. Hop-wise Reassembly (HWR) of a datagram i (s: source; $h_1, \ldots, h_{t-1}$: intermediate hops; d: destination; $\mathbf{t}_x(i)$: datagram tag to next hop x).

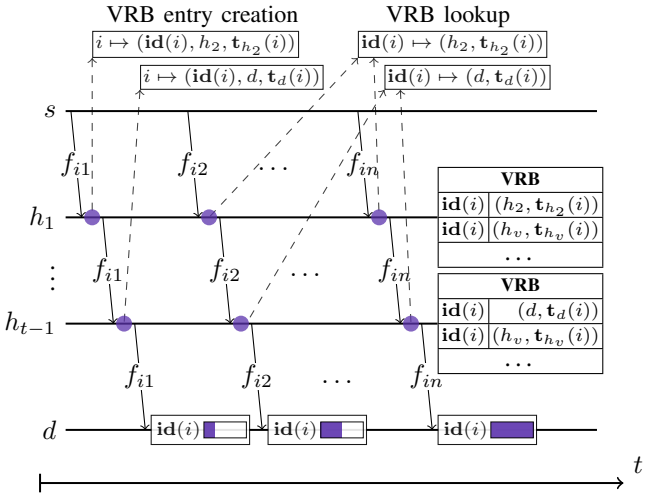


Figure 5. Fragment Forwarding (FF) of a datagram i using the Virtual Reassembly Buffer (VRB). Notations comply with Figure 4.

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 and destination addresses, the datagram size, the datagram tag ($\mathbf{id}(i)$, *cf.,* Section II-A), the next hop link layer address h_v, and the outgoing datagram tag $\mathbf{t}(i)$. 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.

C. Additional Related Work

Other approaches that use similar concepts as FF mainly focus on datagram prioritization [13], [14]. In addition to FF, the 6lo working group of the IETF is also working on a forwarding mechanism that includes *selective fragment recovery* [15]. Selective fragment forwarding is effectively a complete new fragmentation protocol introducing new header types for 6LoWPAN. As it allows for recovery of lost fragments and provides congestion control mechanisms it could help to mitigate the congestion problems we observed in our experiments. Exploring its advantages in more detail will be part of our future work.

Similar to this, Chowdhury *et al.* [16] proposed a standard compliant NACK-based approach for selective fragment recovery. Since those NACKs, however, are associated with $\mathbf{id}(i)$, this mechanism only allows for hop-wise recovery and does not cover the whole end-to-end path when using FF.

The work most closely related [17] to our study uses the 6TiSCH simulator [18] to analyze the performance of FF. The authors show that FF is a promising option in IEEE 802.15.4e (TSCH). As part of our experiments, we revisit the results of these experiments in a real-world setting and find that the abstraction in the simulation of [17] leads to misleading conclusions.

Awwad *et al.* [19] 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-B). 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 [4]—commonly known as *mesh-under* [5, Section 1.2]—because it is known that such a solution falls behind HWR [16].

III. Implementation

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 [20], [21], a common IoT operating system. By selecting an open source platform and making our software publicly available we enable reproducible research [22], [23]. 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. System Details on 6LoWPAN

RIOT provides a stable 6LoWPAN implementation as part of its default network stack, GNRC [24], [25]. 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 1302 bytes (*cf.,* Section II-A).

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.

B. Fragment Forwarding

We extend 6LoWPAN in GNRC to support direct *fragment forwarding*.[1] One crucial implementation choice relates to the creation of the first fragment. The first fragment may include the compression header [9], 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 [12], *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 6. 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.

[1] https://github.com/RIOT-OS/RIOT/pull/11068

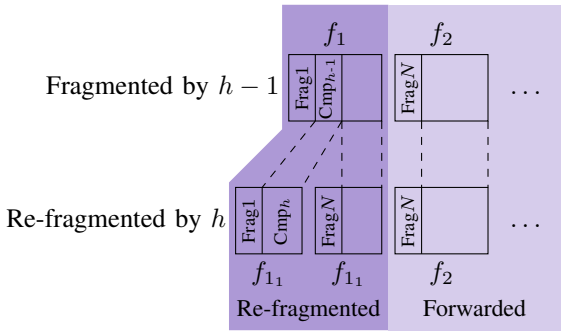


Figure 6. Compression header (Cmp) handling for fragment forwarding in the RIOT GNRC.

C. MAC Layer

In its default configuration, GNRC only provides a very slim MAC layer that benefits from radio drivers that support CSMA/CA, link layer retransmissions, and acknowledgement handling by default. Special care has to be taken for 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 [26] 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 send the packet from the top of the queue again.

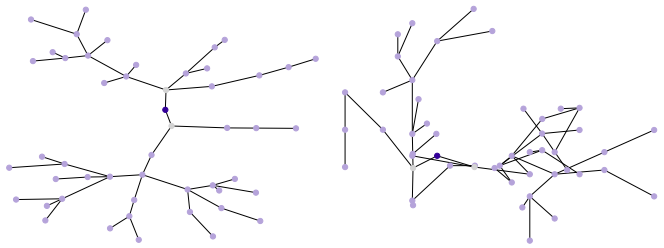
IV. Evaluation

Our evaluations are performed in a real-world testbed using class-2 IoT nodes [7] and real 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-B to stress hop-wise reassembly.

Our goal is to carefully explore the behavior of the competing fragmentation schemes and along this line to reproduce simulation results of [17]. From many previous experiences we know that simulation—although an important tool for network analysis—often produces misleading results in the complex and surprising world of low-power wireless communication.

A. Setup

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



(a) Logical topography. (b) Geographical topography.

Figure 7. Topography of the selected testbed network (dark-blue: sink, light-blue: source nodes).

provides the basic MAC layer features such as CSMA/CA, link layer retransmissions, and acknowledgements.

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. However, a careful selection of nodes is necessary to control side effects that may negatively affect our observations.

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 6.6 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.[2] 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 6.6 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-B.

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 7. Multiple paths have the same length. The longest path consists of 6 hops.

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

[2]We select node 55 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.

Table I
FRAGMENTS / UDP PAYLOAD SIZES MAPPING.

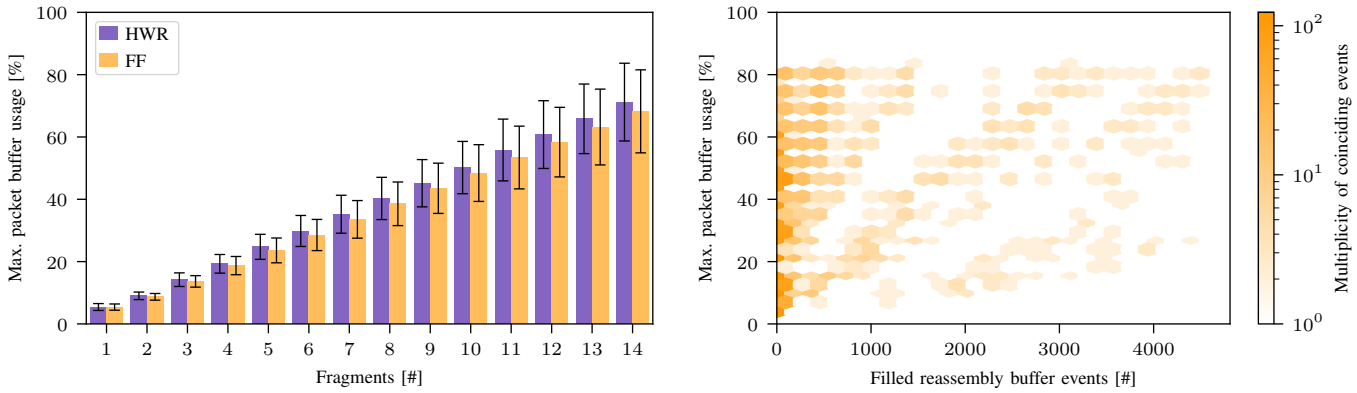
Fragments #	UDP Payload	Mean Reassembly Time	
		HWR	FF
1	16 bytes	(no reassembly)	
2	80 bytes	4.3 ms	5.8 ms
3	176 bytes	10.8 ms	13.7 ms
4	272 bytes	17.4 ms	19.6 ms
5	368 bytes	23.9 ms	26.2 ms
6	464 bytes	32.4 ms	33.5 ms
7	560 bytes	37.3 ms	39.1 ms
8	656 bytes	45.2 ms	47.5 ms
9	752 bytes	52.4 ms	54.5 ms
10	848 bytes	57.4 ms	60.6 ms
11	944 bytes	64.1 ms	67.1 ms
12	1040 bytes	71.3 ms	73.1 ms
13	1136 bytes	78.7 ms	80.7 ms
14	1232 bytes	85.2 ms	88.0 ms

All source nodes start sending UDP packets—using the same payload—to the sink in a uniformly distributed interval between 5 s and 15 s. The experiment ends after each source has sent 100 packets. In contrast to the reference simulation [17], we select a smaller interval to allow for a significant number of runs. Slower sending rates would lead to unfeasible durations in our real-world experiments. It is worth noting that our decision is made carefully: We conducted one experiment with exactly the same run times as described in the related work. The results are consistent with our experiments that adapt the improved parameter setting. The same is the case for smaller network sizes.

HWR and FF implement different fragmentation strategies (*cf.,* Section III-A). Consequently, the original UDP payload may lead to differently sized fragments resulting in varying overheads for the reassembly processes. To allow for the fair comparison of both approaches, we need to align the baseline depending on the UDP payload size. Table I shows the best results based on our empirical validation. We use these payload sizes in our subsequent experiments.

To evaluate the performance, our experiments measure the same metrics as the simulation [17]. This includes reliability, specifically the PDR, and the latency between the UDP sockets of source and sink. In addition, we also assess system complexity in terms of memory.

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 our testbed. (*ii*) We want to compare our results with related work that analyzed some aspects in simulation [17]. We document the changes of default values in the Appendix.



(a) Maximum packet buffer usage per fragment multiplicity.

(b) Maximum packet buffer usage versus reassembly buffer exhaustion for FF. The saturation of a hexagon indicates higher multiplicity of these events coinciding in its area.

Figure 8. Analysis of the packet buffer utilization.

In contrast to the parameters in [17], the default size of the virtual reassembly buffer in GNRC is 16 bytes. Since this only prefers direct forwarding, 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. Result 1: Memory Consumption

Table II shows both ROM and RAM usage of the 6LoWPAN layer at the source node for both forwarding approaches. When compiling the software we use `arm-none-eabi-gcc` v7.3.1 with `-Os` optimization (size-optimal) for ARM Cortex-M3 and the compile-time parameters we line out in Section IV-A. We use the `size` tool to extract the relevant module information. To make memory measurements compatible, we set the reassembly buffer size to the same value as the VRB size (16) for HWR. The anticipated memory advantage 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 in the asynchronous GNRC fragmentation buffer. More ROM is also needed for the possible refragmentation of the first fragment. The majority of the $\approx$ 500 bytes of additional ROM 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.

Table II
MEMORY SIZES [BYTES] FOR SOURCE NODES.

Module	HWR		FF	
	ROM	RAM	ROM	RAM
6LoWPAN	5950	6124	6472	4284
VRB	n/a	n/a	316	768
Forwarding	n/a	n/a	544	0
Sum	5950	6124	7332	5052

Figure 8 presents our analysis of the actual utilization of the 6144 bytes packet buffer. For FF the packet buffer is used just a little less than for HWR. This can be seen in Figure 8(a), which plots the maximum packet buffer utilization during the runtime of each experiment. 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.

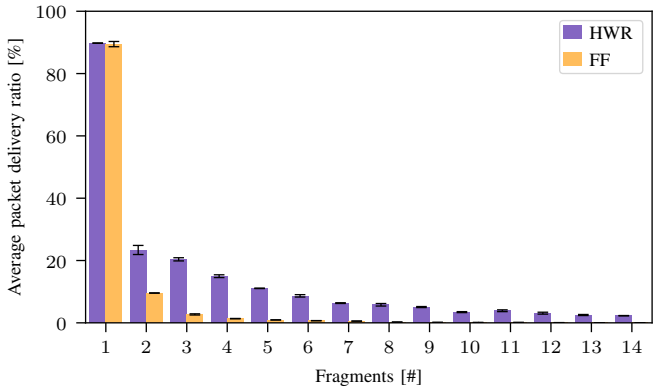
A clear correlation between events where a full reassembly buffer coincides with a high (or low) maximum usage of the packet buffer can be seen in Figure 8(b). This plot visualizes events taken from all nodes during three runs of the experiment. More saturated hexagons indicate higher multiplicities of events in this area. The occurrence of many full reassembly buffers tends to lead to high maximum packet buffers. Those coinciding events, however, are less likely in general. The observed clusters are in line with the hop distances from the nodes on which the coinciding events happen to the sink.

C. Result 2: Reliability and Latency

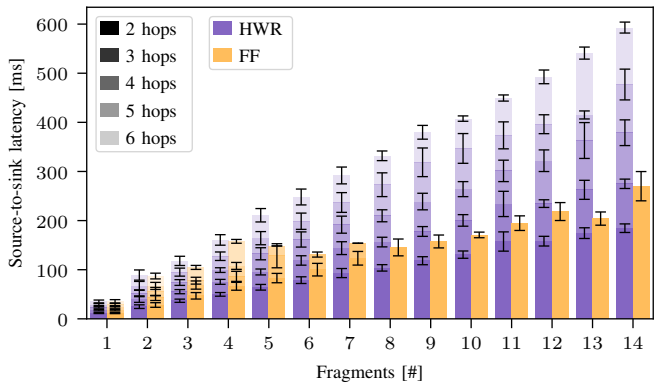
Figures 9(a) and 9(b) displays our results from measuring reliability and latency. Strikingly, FF admits poor reliability, which is in contrast to previous results [17]. Even for a small number of fragments, FF achieves less than half the PDR of HWR. Values then quickly approach zero with increasing number of fragments. HWR, though also performing poorly, manages to deliver at least some packets to the more distant nodes.

The latencies we measured for FF are also significantly higher than in the previous simulation work. HWR is expected to operate slower because each node needs to reassemble the entire frame prior to forwarding to the next hop.

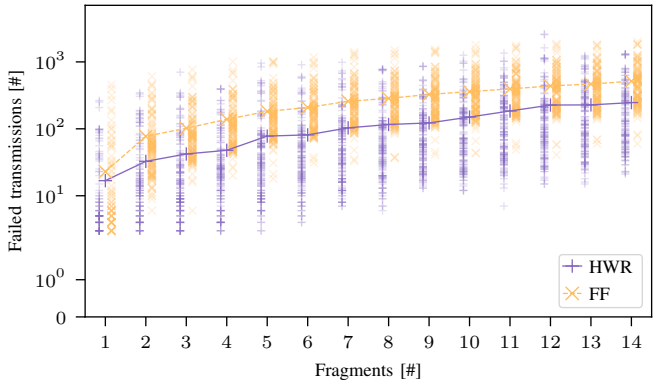
To explore the underlying reasons for the poor performance of FF, we analyse the radio transmission and media occupancy.



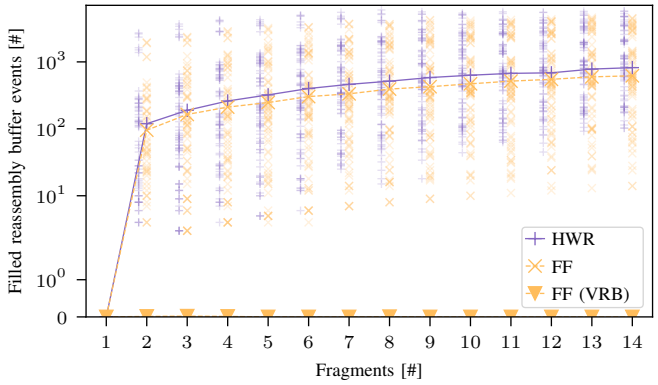
(a) Packet delivery ratio.



(b) Source-to-sink latency (socket-to-socket) by hop distance.



(c) Link layer retransmissions per node. Lines depict average values.



(d) Filled reassembly buffer events per node. Lines depict average values.

Figure 9. Measurement results for 100 packets every [5,10] s per node (3 runs).

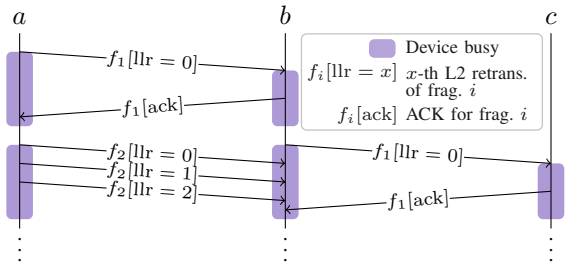


Figure 10. Example of L2 retransmissions caused by the device being busy, resulting in missing ACKs.

Figure 9(c) plots the number of link layer retransmissions that occurred for each node within the network over three experiment runs as a scatter plot with a logarithmically scaled y-axis. The line plot within the scatter plot represents the means of the respective data set.

In our experiments, we see significantly more link layer retransmissions per node with FF than with HWR. This is caused by much faster send and receive triggers on the device due to immediate fragment forwarding, which increases collisions and packet loss. 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 10. We are able to confirm with local measurements on a sister device of the nodes' radio (AT86RF233 [27]) that the device can remain busy for up to 4 ms utilizing a logic analyzer. With HWR the link is more relaxed due to the time it takes to reassemble and re-fragment a packet again, which leaves both the device and the medium non-stressed.

We can also see that packets are lost with FF when the respective reassembly buffers are full. In Figure 9(d), we plot these occurrences of reassembly buffer exhausts for each node over 3 experiment runs analog to Figure 9(c). The reassembly buffer with FF is only about 23% less often filled than with HWR. This hints at frequent transmissions that lose the first fragment and cause the FF implementation to fall back to normal reassembly, since the first fragment is missing or is received out of order (*cf.,* Section III-A). Indeed, in 50-60% of the cases, we observe that the reassembly buffer

expires because the first fragment of a datagram is missing. These transmission failures fill the reassembly buffer up with incomplete datagrams, especially when more fragments are lost. In this scenario a datagram never actually takes the full 10 s of reassembly timeout to reassemble at each hop (the plain source-to-sink latency is 600 ms at most). Hence it is unlikely that different strategies—for instance re-fragmenting partly reassembled datagrams and forwarding the rest as soon as the first fragment comes in—would noticeably increase reliability. Still, this process might save space in the reassembly buffer: When the first fragment just arrives after a subsequent fragment the reassembly buffer does not require the full space of the datagram.

To further verify that reliability problems are not caused by our implementation, we repeat the experiments with a modified version of FF. Our modification makes FF simulate the behavior of HWR by putting the fragments to forward in a VRB-associated queue instead of sending them. Only after all fragments belonging to the datagram pass the forwarding engine, all fragments queued in the VRB are sent. The performance of FF in those experiments is comparable to the HWR results we observed in our evaluation above. The number of link layer retransmissions also goes down to a comparable level. We consider this a strong indication of a consistent code base.

V. Discussion

In our testbed experiments, we were not able to reproduce the results for FF that are based on simulations as presented in [17]. One striking difference between the two settings is our faster, lightly coordinated CSMA/CA MAC layer, which is by no means uncommon. Corresponding problems have been already hinted at in [6], and are now substantiated.

We did not expect to see such disappointing results as revealed in this paper. In our given scenario, FF becomes more of a hindrance than an improvement over HWR, even though our implementation optionally falls back to HWR in case of fragment loss. The only advantage of FF we could clearly identify is its reduced RAM consumption. Evaluating whether alternative approaches to fragment forwarding such as Selective Fragment Recovery [15] could help to mitigate these problems will be part of our future work.

Nonetheless, the stress on the device can only be reduced by a more elaborate MAC protocol. In such attempts, however, care needs to be taken with the configuration of the experiment parameters: Preliminary experiments with an existing MAC protocol in RIOT [28] led to problems such as frequent packet buffer overflows, after the packets stayed much longer in the buffer queues of the MAC layer.

In the end, deployment scenarios and provider use cases should decide whether fragment forwarding is applicable and if so on which MAC protocol.

VI. Conclusion and Outlook

In this paper, we evaluated direct fragment forwarding with 6LoWPAN in comparison to hop-wise reassembly using large real-world experiments. We showed that with a thin MAC layer, hop-wise reassembly can be the better choice to achieve proper reliability and latencies. This contradicts previous results, but becomes clearer after careful analysis reveals that the medium is quickly exhausted by quicker fragment sending and retransmissions.

Further experiments are needed not only to evaluate more complex MAC layers and contrast with the results in [17], but also to empirically relate FF to other fragment forwarding techniques including the selective 6LoWPAN fragment recovery protocol [15]. A possible direction of further evaluation could also include end-to-end fragmentation such as performed by IP [2].

A Note on Reproducibility

We explicitly support reproducible research [22], [23]. Our experiments have been conducted in an open testbed. The source code of our implementations (including scripts to setup the experiments, RIOT measurement apps *etc.*) will be available on Github at https://github.com/5G-I3/IEEE-LCN-2019.

Acknowledgments

We would like to thank Jakob Pfender and the anonymous reviewers for their feedback on this paper. This work was supported in parts by the German Federal Ministry of Education and Research within the 5G project *I3* and the VIP+ project *RAPstore*.

References

[1] J. Hou, B. Liu, Y.-G. Hong, X. Tang, and C. Perkins, "Transmission of IPv6 Packets over PLC Networks," IETF, Internet-Draft – work in progress 00, February 2019.

[2] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," IETF, RFC 8200, July 2017.

[3] 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, New York, NY, USA, Tech. Rep. IEEE Std 802.15.4™–2015, April 2016.

[4] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," IETF, RFC 4944, September 2007.

[5] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*, 1st ed. Wiley Publishing, 2009.

[6] T. Watteyne, C. Bormann, and P. Thubert, "6LoWPAN Fragment Forwarding," IETF, Internet-Draft – work in progress 03, July 2019.

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

[8] S. L. Keo, S. S. Komar, and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 265–275, May 2014.

[9] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," IETF, RFC 6282, September 2011.

[10] C. Bormann, "6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)," IETF, RFC 7400, November 2014.

[11] G. Z. Papadopoulos, P. Thubert, S. Tsakalidis, and N. Montavont, "RFC 4944: Per-hop Fragmentation and Reassembly Issues," in *Proc. of the 2018 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, October 2018.

[12] C. Bormann and T. Watteyne, "Virtual reassembly buffers in 6LoWPAN," IETF, Internet-Draft – work in progress 01, March 2019.

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

[14] A. Weigel, M. Ringwelski, V. Turau, and A. Timm-Giel, "Route-over forwarding techniques in a 6LoWPAN," in *International Conference on Mobile Networks and Management*. Springer, 2013, pp. 122–135.
[15] P. Thubert, "6LoWPAN Selective Fragment Recovery," IETF, Internet-Draft – work in progress 05, July 2019.
[16] A. H. Chowdhury, M. Ikram, H.-S. Cha, H. Redwan, S. Shams, K.-H. Kim, and S.-W. Yoo, "Route-over vs Mesh-under Routing in 6LoWPAN," in *Proc. of the 2009 International Conference on Wireless Communications and Mobile Computing (IWCMC): Connecting the World Wirelessly*. ACM, June 2009, pp. 1208–1212.
[17] Y. Tanaka, P. Minet, and T. Watteyne, "6LoWPAN Fragment Forwarding," *IEEE Communications Standards Magazine*, vol. 3, no. 1, pp. 35–39, July 2019.
[18] M. Esteban, D. Glenn, M. Vucinic, and T. Watteyne, "Simulating 6TiSCH Networks," *Wiley Internet Technology Letters*, vol. 30, no. 3, 2018.
[19] S. A. 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 Personal Communications*, vol. 98, no. 1, pp. 1029–1053, Jan. 2018.
[20] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 79–80.
[21] 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 of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038
[22] Q. Scheitle, M. Wählisch, O. Gasser, T. C. Schmidt, and G. Carle, "Towards an Ecosystem for Reproducible Research in Computer Networking," in *Proc. of ACM SIGCOMM Reproducibility Workshop*. New York, NY, USA: ACM, August 2017, pp. 5–8.
[23] ACM, "Result and Artifact Review and Badging," http://acm.org/publications/policies/artifact-review-badging, Jan., 2017.
[24] H. Petersen, M. Lenders, M. Wählisch, O. Hahm, and E. Baccelli, "Old Wine in New Skins? Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices," in *1st Int. Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys15)*. Florence, Italy: ACM, May 2015.
[25] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündogan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," Open Archive: arXiv.org, Technical Report arXiv:1801.02833, January 2018. [Online]. Available: https://arxiv.org/abs/1801.02833
[26] *Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE, SP100, WirelessHART, and ISM Applications (AT86RF231)*, Microchip, September 2009, Rev.8111C.
[27] *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM Applications (AT86RF233)*, Microchip, July 2014, Rev. 8315E.
[28] S. Zhuo and Y. Song, "GoMacH: A Traffic Adaptive Multi-channel MAC Protocol for IoT," in *Proc. of the 42nd IEEE Conference on Local Computer Networks (LCN)*. Piscataway, NJ, USA: IEEE Press, October 2017, pp. 489–497.

APPENDIX
COMPILE TIME PARAMETERS IN RIOT

Table III
CHANGED COMPILE TIME PARAMETERS IN RIOT.

Compile-time Configuration Parameter	Value
GNRC_NETIF_PKTQ_POOL_SIZE	64
GNRC_SIXLOWPAN_FRAG_RBUF_SIZE	1 (src.) / 16 (sink)
GNRC_SIXLOWPAN_FRAG_RBUF_TIMEOUT_US	10000000
GNRC_SIXLOWPAN_FRAG_RBUF_AGGRESSIVE_OVERRIDE	0
GNRC_SIXLOWPAN_MSG_FRAG_SIZE	64