6LoWPAN Fragment Forwarding

Yasuyuki Tanaka, Pascale Minet, Thomas Watteyne

6LoWPAN Fragment Forwarding

Yasuyuki Tanaka, Pascale Minet, Thomas Watteyne *†

March 8, 2019

Abstract

Low-power wireless technologies have been applied to industrial fields not only to monitor facilities but also to control them. There is a legitimate requirement to integrate low-power wireless networks with existing IP-enabled networks such as the Internet. The 6LoWPAN standard makes this happen easily by enabling low-power wireless networks to transport IPv6 packets.

A challenge is that an IPv6 packet might not fit in a link-layer frame. The answer is fragmentation: the IPv6 packet is cut into fragments, each fitting in a frame. In a typical implementation, the IPv6 packet is fragmented and reassembled at every hop. Such per-hop reassembly causes low end-to-end reliability and high end-to-end latency. This article presents a new implementation technique which results in fragment forwarding without changing any of the standards. Simulation results show how, when going from per-hop reassembly to fragment forwarding, end-to-end reliability goes from 40% to 100%, memory requirements go from 1280 B to 160 B, and end-to-end latency is halved.

1 Introduction

Low-power wireless communication is a key technology for the Internet of Things. Products exist today which offer over 99.999% end-to-end reliability and over a decade of battery lifetime [1]. IPv6 allows those networks to seamlessly connect to the Internet: every node gets an IPv6 address, and writing an application to interact with a low-power wireless node now becomes very

*Y Tanaka, P. Minet and T. Watteyne are with *Inria, EVA team, Paris, France.* E-Mail: {yasuyuki.tanaka, pascale.minet, thomas.watteyne}@inria.fr.

†Manuscript received XXX, XXX, XXXX; revised XXX, XXX, XXX.

similar to interacting with another computer. The 6LoWPAN standard is what makes this possible: it compresses the IPv6 header so that IPv6 packets can flow on a low-power wireless network with only a small overhead.

Yet, transporting IPv6 packets over a low-power wireless network comes with its challenges. One of that is fragmentation, which we address in this article. An IPv6 packet can be up to 1280 octets long, but low-power wireless standards such as IEEE802.15.4 [2] have a maximum Payload Data Unit (PDU) of 127 octets. An IPv6 packet does not fit in an IEEE802.15.4 frame. Yet, if one wants to claim IPv6-compliance, the low-power wireless network must be able to transport long packets.

The mechanism developed by 6LoWPAN is fragmentation. It is defined in RFC4944 [3] and RFC6282 [4]. The base principle is that an IPv6 packet is divided into fragments, each of which fits into a link-layer frame. In a straightforward implementation, fragments are forwarded to the next hop, which reassembles the original IPv6 packet, possibly re-fragmenting it before forwarding it to the next hop after that. This reassembly/fragmentation process happens at each hop. We call this "per-hop reassembly".

Per-hop reassembly has two main issues. First, end-to-end latency is high as each node needs to wait for the last fragment before sending the first fragment to the next hop. But, perhaps more importantly, end-to-end reliability is affected by the fact that a node has limited (RAM) memory and cannot reassemble many packets at the same time. This means that, if a node is already reassembling 2 packets, it might have to drop fragments from a third packet, as it does not have enough RAM memory to allocate a new reassembly buffer.

In this paper, we present an implementation which results in intermediate nodes forwarding the fragments without reassembly. We call this "fragment forwarding". What makes fragment forwarding attractive is that it remains entirely standard compliant: no new standard protocols are needed to make it work, and a network can contain a mix of nodes that do and do not implement it.

The remainder of this article is organized as follows. Section 2 provides an overview of a typical per-hop reassembly implementation of 6LoWPAN fragmentation. Section 3 proposes fragment forwarding, a novel implementation technique of the same standard. Section 4 presents simulation results, comparing fragment forwarding to per-hop reassembly. Section 5 discusses limitations of fragment forwarding and possible enhancements. Section 6 concludes this article.

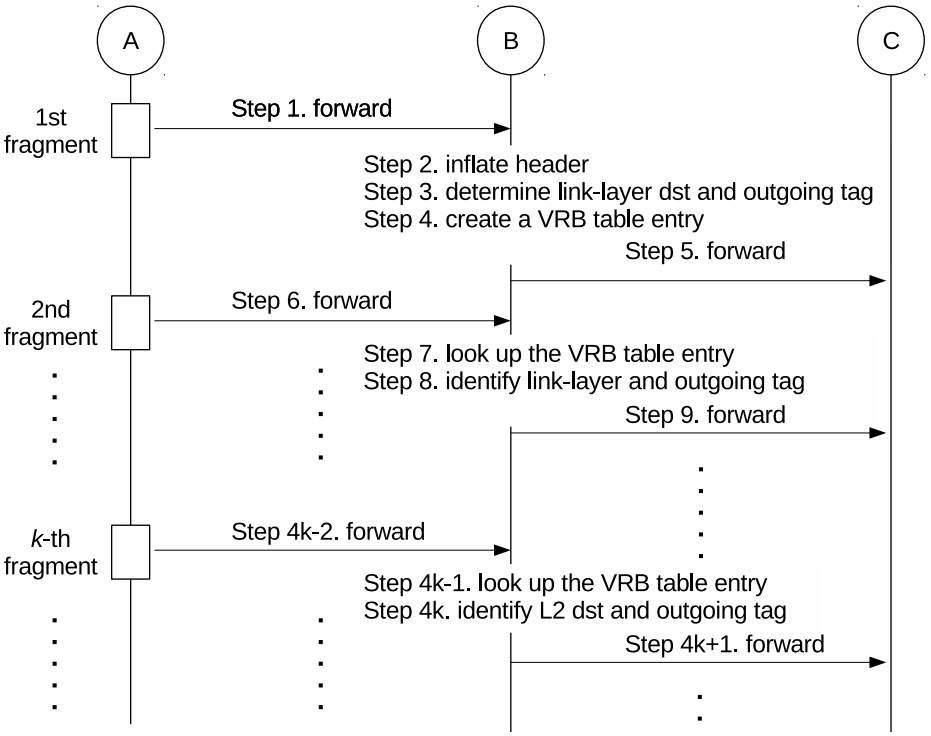

Figure 1: Illustrating the fragment forwarding implementation technique presented in this article. We assume a packet traveling from node A, to node B, then node C. Node B forwards each fragment to node C, without reassembling the entire packet.

2 6LoWPAN Per-hop Fragmentation

On any link which cannot convey a 1280-octet IPv6 packet in a single link-layer frame, link-specific fragmentation and reassembly must be provided for valid IPv6 communication [5]. IETF 6LoWPAN [3, 4] is an adaptation layer to transport long IPv6 packets (up to 1280 octets) into short IEEE802.15.4 [2] frames (at most 127 bytes). It provides link-specific IPv6 header compression, as well as fragmentation and reassembly. It consists of two main mechanisms.

First, it defines rules for compressing the IPv6 header. This is done by (1) removing fields that are not needed, (2) removing fields which always have the same contents, and (3) compressing the IPv6 addresses by inferring them from link-layer addresses. The result is that the 40-octet IPv6 header gets compressed down to 2 octets in the most favorable case. A Low-power Border Router (LBR) sits at the edge of the low-power wireless network and is responsible for doing transparent IPv6→6LoWPAN and 6LoWPAN→IPv6 translation. This allows a computer outside the low-power wireless network to interact with a low-power wireless device directly, using its IPv6 address.

Second, it defines fragmentation rules, so multiple IEEE802.15.4 frames can make up a single IPv6 packet. Each fragment has a MAC header, a fragment header, and a piece of the original IPv6 packet. The fragment header indicates the packet identifier, the length of the original packet, and the offset of the piece from the beginning of the packet. The packet identifier is called datagram tag, which is a 16-bit number locally unique between two link-layer nodes. All fragments of an IPv6 packet have the same datagram tag.

Fig. 2 illustrates a typical implementation of 6LoWPAN fragmentation. At the source node, the packet is cut into fragments small enough to fit into link-layer frames. When the next hop node receives the first fragment, it allocates a reassembly buffer big enough to fit the packet (the total length of the packet is found in the fragment header). It then fills the reassembly buffer as it receives fragments. Once all fragments are received, the node inflates the original packet, and hands it to its IPv6 layer. Depending on the IPv6 destination of the packet, it might get fragmented again, and be sent to the next hop. That is, fragmentation and reassembly happen at each node forwarding the packet.

Per-hop reassembly requires the mote to allocate sufficient memory for reassembly buffers, each potentially requiring 1280 B.

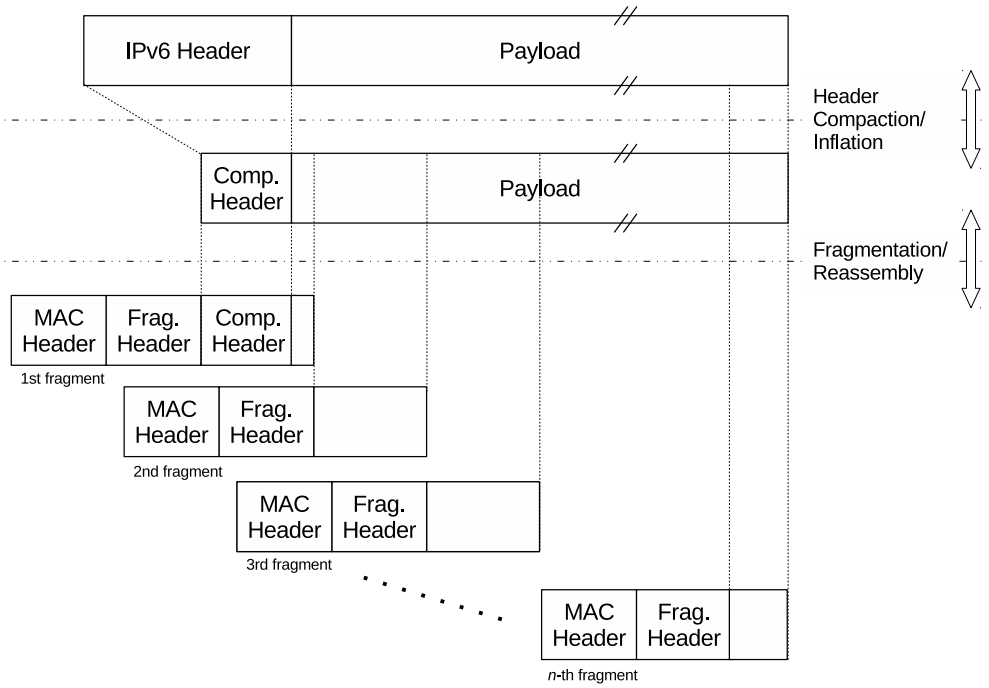

Figure 2: Typical implementation of 6LoWPAN fragmentation, resulting in per-hop reassembly. An outgoing IPv6 packet is compressed using header compression, and fragmented when the result does not fit into a single link-layer frame. Incoming fragments are reassembled into a compressed IPv6 packet, then inflated to form the original IPv6 header. A typical implementation fragments/reassembles at every hop.

3 Fragment forwarding

Fragment forwarding is an implementation technique of 6LoWPAN fragmentation which eliminates the need to fragment and reassemble at every hop. Its core idea is introduced by Shelby and Bormann [6], and is now the major focus of the IETF 6lo Fragmentation Design Team [7, 8].

The core idea of fragment forwarding is as follows. When receiving the first fragment, a node determines the next hop based on the destination IPv6 address in that fragment. It then forwards that fragment immediately to that neighbor, and remembers the datagram tag of that fragment. When receiving subsequent fragments (which have the same datagram tag), the node forwards them to the same next hop. Fragments are reassembled only at the destination node.

The memory needed for this forwarding technique is called "Virtual Reassembly Buffer" (VRB): the node behaves as if it were reassembling and fragmenting a packet from the viewpoint of the next hop, only without ever holding the entire IPv6 packet.

A node maintains a VRB table, each entry of which corresponding to a packet it is forwarding. Each VRB entry is a tuple with 4 elements: the source link-layer address of the incoming fragments, the datagram tag of the incoming fragments, the destination link-layer address of the outgoing fragment, and the datagram tag of the outgoing fragments. One VRB entry requires 20 B of memory, assuming 64-bit link-layer addresses.

When receiving a fragment from a neighbor with a datagram tag not present for that neighbor in the VRB table, the node creates a new VRB entry. It fills the source link-layer address and incoming datagram tag read from the incoming frame. It determines the destination link-layer address based on the next hop identified by the forwarding engine. It picks a datagram tag for the outgoing fragments that is unique for that neighbor (i.e. not yet in the VRB table). Once the VRB entry is created, it is used for all subsequent fragments of the same packet. Upon forwarding the last fragment, the node removes the VRB entry.

Fig. 1 illustrates how fragment forwarding works. A packet goes from node A to node B, then node C. On reception of the first fragment, node B determines the next-hop according to the destination IPv6 address in the IPv6 header. Node B creates the VRB entry and forwards the first fragment to node C. When node B receives the second fragment, it looks up its source link-layer address and datagram tag in the VRB table and finds the entry. It forwards the fragment using the destination link-layer address and the datagram tag from that VRB entry. All subsequent fragments go through the same process. After having forwarded the last fragment, node B clears

that VRB entry.

Fragment forwarding is attractive mainly for two reasons. First, it is an implementation technique, not a new protocol. That is, any node remains fully 6LoWPAN compliant when implementing this technique. Second, it does not require all nodes in the network to implement the technique. That is, a network can be composed of a mix of nodes that implement per-hop reassembly, and nodes that implement fragment forwarding. The nodes that implement per-hop reassembly simply reassemble the packet before fragmenting and forwarding it to the next hop.

4 Simulation Results

We compare the performance of per-hop reassembly and fragment forwarding by simulation.

4.1 Simulation Settings

We implement both per-hop reassembly and fragment forwarding on the 6TiSCH Simulator[1] [9]. This simulator is being maintained by the 6TiSCH working group, and implements the full behavior of the 6TiSCH stack.

In per-hop reassembly, a reassembly buffer has a maximum lifetime of 60 s. That is, the buffer is freed when either the reassembly is done, or the buffer has not been used for that period of time. Similarly, in fragment forwarding, a VRB has maximum lifetime of 60 s. That is, the entry is cleared when either the last fragment of a packet is forwarded, or when the entry has not been used for that period of time.

To witness the behavior of fragment forwarding, we run a simulation on the canonical 10-node topology shown in Fig. 3. It focuses on the situation of two flows of data converging to a "bottleneck" node (node I), which is the critical case as it forces node I to reassemble and forward multiple packets at the same time. This canonical topology allows us to precisely understand the behavior of both implementations, as in a "worst case" scenario. The same will happen in any topology at various degrees depending on whether there are bottlenecks present. That is, the results and lessons learned from this canonical topology are absolutely representative, and carry over to a more general topology.

In the topology of Fig. 3, each non-sink node generates a packet with an inter-packet period taken uniformly in [54 s, 66 s] (i.e. every minute with 10% randomization) and sends it to node J. The arrows indicate the routing

[1] `http://bitbucket.org/6tisch/simulator/`

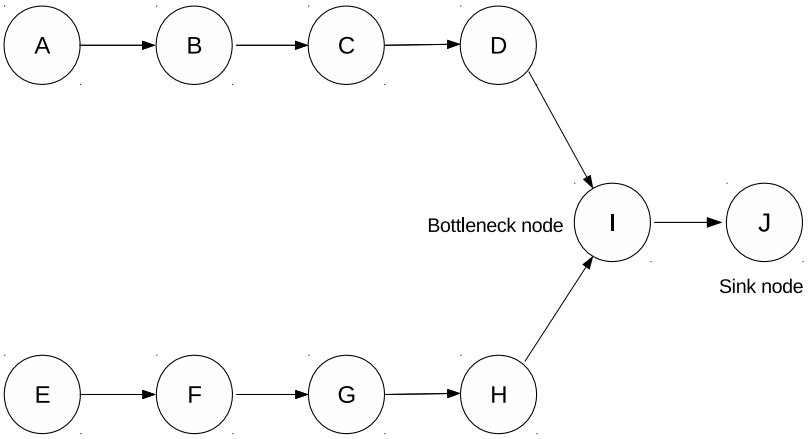

Figure 3: Canonical topology used in the simulation campaign. All non-root nodes periodically send packets to Node J. Exact simulation parameters are listen in Table 1.

paths, e.g. a packet from node B follows the B→C→D→I→J multi-hop route. Node I is the bottleneck, as all packets from nodes A to H flow through it.

Table 1 details the simulation parameters used. The goal of these parameters is to ensure we are measuring only the performance of per-hop reassembly and fragment forwarding, not other elements such as slot allocation, the length of TX queue, and the maximum number of link-lyare retransmissions. We use a 101 slot slotframe with 10 ms slots, the default values in RFC8180 [10]. The packet delivery ratio of all links is 100%, i.e. there are neither link-layer drops nor retries[2].

The cells in the TSCH schedule are statically allocated to ensure that there is enough bandwidth to transport all the fragments. On any node except for node J, one TX cell is allocated for its own traffic. In addition, as many TX cells and RX cells as the number of its descendants are allocated. This is to ensure that we measure only the performance of per-hop reassembly vs. fragment forwarding, not the performance of different cell allocation schemes. Slot offsets are randomly chosen in the slotframe, at every simulation run.

Low-power wireless devices are constrained in memory. With per-hop reassembly, we limit the number of reassembly buffers to 1. Similarly, with

[2] Having link-layer retries would have *no* influence on the results.

Table 1: Simulation parameters.

Parameter	Settings
TSCH slotframe length	101 slots
Slot duration	10 ms
Link reliability	100 %
Packet interval	Uniform in [54 s, 66 s]
Cell Allocation	Node A: 1 TX cell Node B: 2 TX cells and 1 RX cell Node C: 3 TX cells and 2 RX cells Node D: 4 TX cells and 3 RX cells Node E: 1 TX cell Node F: 2 TX cells and 1 RX cell Node G: 3 TX cells and 2 RX cells Node H: 4 TX cells and 3 RX cells Node I: 9 TX cells and 8 RX cell Node J: 9 RX cells
(per.hop reas.) # reassembly buffers	1 (1280 B of memory)
(frag. forwad.) # VRBs	8 (160 B of memory)
Num. fragment per packet	between 1 and 10
Number of simulation runs	100
Duration of one simulation run	7000 s

fragment forwarding, we limit the number of VRBs to 8. Note that, in both cases, about the same amount of memory is consumed. These are realistic numbers for today's micro-controllers. For example, the number of reassembly buffers is 1 in the latest release of Contiki open-source implementation[3] [11]. Increasing those numbers will "push the problem further", but not solve it.

We run simulations, and vary the number of fragments per packet between 1 and 10. When the number of fragments is 1, no fragmentation is happening. For each number of fragments, we run the simulation 100 times, and plot all results with a 95% confidence interval. The duration of one simulation run is 7000 s of network life.

4.2 End-to-end Reliability

We call end-to-end reliability the ratio of packets that reach their final destination. That is, if a fragment is lost, the packet is considered lost because it cannot be reassembled.

Fig. 4 shows the end-to-end reliability results. When there is no fragmentation (number of fragments is 1), the performance of both implementation techniques is the same, as expected, and end-to-end reliability is 100%. Yet, with per-hop reassembly, end-to-end reliability drops quickly with the number of fragments per packet. We confirm by looking at the simulation logs that packet loss is entirely due to fragments being dropped at node I because it runs out of reassembly buffer space. With fragment forwarding, end-to-end reliability stays at 100% in all cases.

4.3 End-to-end Latency

We call end-to-end latency the duration between the time the source node sends the first fragment, and the time the destination node (node J) receives the last fragment. Packets that do *not* reach the destination are not taken into account in this calculation.

Fig. 5 shows the end-to-end latency results. In both cases, latency increases linearly with the number of fragments per packet, as each fragment adds the same delay at each intermediate node. Fragment forwarding reduces end-to-end latency by roughly 50% when compared to per-hop reassembly.

[3] The development version of Contiki and the next generation of Contiki (contiki-ng, https://www.contiki-ng.org) can handle two packet reassemblies at the same time with about 1280 B memory, by default.

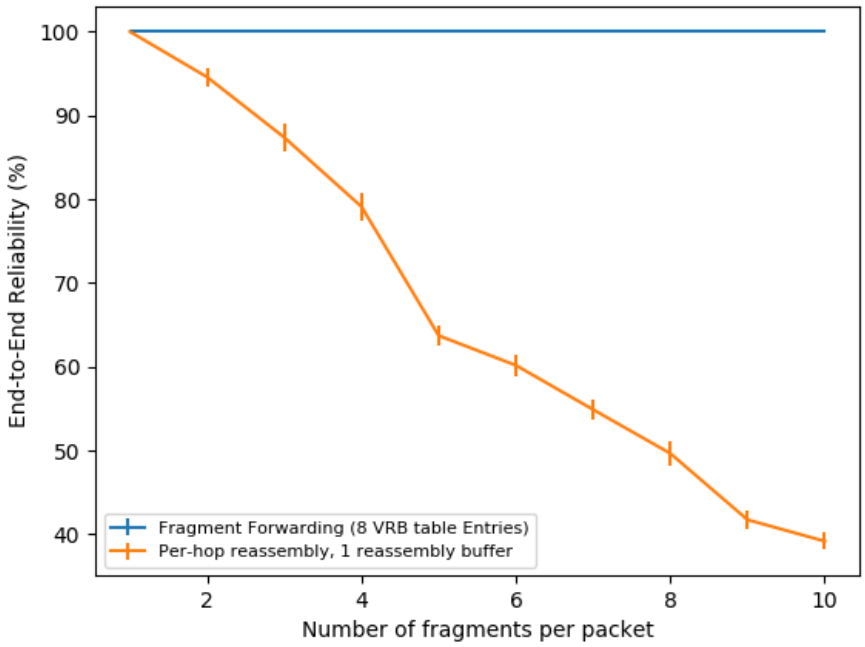

Figure 4: Comparing end-to-end reliability with per-hop reassembly and fragment forwarding. Results are averaged over 100 simulation runs and plotted with a 95% confidence interval. With per-hop reassembly, frames are dropped at node I because it runs out of memory for the reassembly buffer.

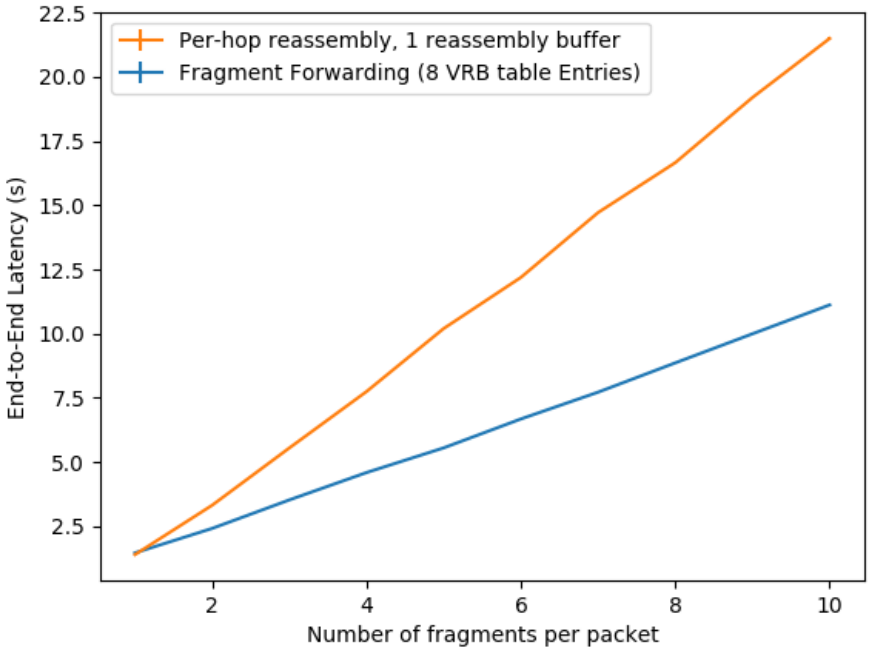

Figure 5: Comparing end-to-end latency with per-hop reassembly and fragment forwarding. Results are averaged over 100 simulation runs and plotted with a 95% confidence interval (which is too small to see in the plot). In this scenario, using fragment forwarding reduces end-to-end latency by roughly 50% when compared to per-hop reassembly.

5 Discussion

According to the results presented in Section 4, fragment forwarding outperforms per-hop reassembly on all fronts. For a fraction of the memory footprint of per-hop reassembly, fragment forwarding achieves 100% delivery, even with the maximum packet length, with half the end-to-end latency. We do not see a compelling argument *not* to implement fragment forwarding, and our main recommendation is that all 6LoWPAN implementations should use it.

That being said, there are some limitation to fragment forwarding, which we want to explicitly highlight here. First, packets can still be dropped. Each VRB entry occupies 20 B of memory. This is a memory footprint 2 orders of magnitude smaller compared to a 1280-byte reassembly buffer for each packet. Yet, the size of the VRB table necessarily remains finite. In the extreme case where a node is required to concurrently forward more packets than it has entries in its VRB table, packets are dropped. Second, there is no fragment recovery built in. There is no mechanism in fragment forwarding for the node that reassembles a packet to request a single missing fragment. Dropping a fragment requires the whole packet to be resent. This causes unnecessary traffic, as fragments are forwarded even when the destination node can never construct the original IPv6 packet. Depending on the networking technology used, it might be interesting to add a fragment recovery mechanism such as the one developed by Thubert [12]. Third, fragment forwarding does not allow per-fragment routing. All subsequent fragments follow the same sequence of hops from the source to the destination node as the first fragment.

6 Conclusion

This article presents an implementation technique for 6LoWPAN that results in fragment forwarding. Simulation results show that fragment forwarding is superior to per-hop reassembly (the typical implementation technique) in terms of end-to-end reliability and end-to-end latency, while having a memory footprint which is significantly lower. We evaluate fragment forwarding in the context of 6TiSCH, but it holds for any 6LoWPAN network.

Fragment forwarding is being standardized, as part of the 6lo Fragmentation Design team, part of the 6LoWPAN effort at the Internet Engineering Task Force (IETF)[4].

[4] This standardization effort is being led by one of the co-authors of this article.

Acknowledgments

The authors would like to thank Carsten Bormann and Pascal Thubert for the in-depth discussions about 6LoWPAN fragmentation, as well as Gabriel Montenegro and Samita Chakrabarti for creating the 6lo Fragmentation Design team that made this work happen.

References

[1] T. Watteyne, J. Weiss, L. Doherty, and J. Simon, "Industrial IEEE802.15.4e Networks: Performance and Trade-offs," in *IEEE International Conference on Communications (ICC), Internet of Things Symposium*. London, UK: IEEE, 8-12 June 2015.

[2] *IEEE Standard for Low-Rate Wireless Networks*, IEEE Std. 802.15.4-2015, April 2016.

[3] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, *Transmission of IPv6 packets over IEEE 802.15. 4 networks*, IETF Std. RFC4944, September 2007.

[4] J. Hui and P. Thubert, *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*, IETF Std. RFC6282, September 2011.

[5] S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, IETF Std. RFC8200, July 2017.

[6] Z. Shelby and C. Bormann, *6LoWPAN: The wireless embedded Internet*. John Wiley & Sons, Ltd, October 2009.

[7] T. Watteyne, C. Bormann, and P. Thubert, *LLN Minimal Fragment Forwarding*, IETF Std. draft-watteyne-6lo-minimal-fragment-01 [work-in-progress], 2018.

[8] C. Bormann, *Virtual Reassembly Buffers in 6LoWPAN*, IETF Std. draft-bormann-lwig-6lowpan-virtual-reassembly-00 [work-in-progress], 2018.

[9] E. Municio, G. Daneels, M. Vucinic, S. Latre, J. Famaey, Y. Tanaka, K. Brun, K. Muraoka, X. Vilajosana, and T. Watteyne, "Simulating 6TiSCH Networks," *Wiley Transactions on Emerging Telecommunications (ETT)*, 2018.

[10] X. Vilajosana, K. Pister, and T. Watteyne, *Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration*, IETF Std. RFC8180, May 2017.

[11] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *IEEE International Conference on Local Computer Networks*. IEEE, 2004, pp. 455–462.

[12] P. Thubert, *6LoWPAN Selective Fragment Recovery*, IETF Std. draft-thubert-6lo-fragment-recovery-00 [work-in-progress], 2018.