RPL-mcaster: Overcoming Memory Limitations in RPL Point-to-Multipoint Routing

Csaba Kiraly
Bruno Kessler Foundation – IRST
kiraly@fbk.eu

Timofei Istomin, Oana Iova, Gian Pietro Picco
University of Trento – DISI
{timofei.istomin, oanateodora.iova, gianpietro.picco}@unitn.it

***Abstract*—RPL, the IPv6 Routing Protocol for Low-Power and Lossy Networks, supports both upward and downward traffic. However, support for the latter is compromised by memory limitation in the nodes. In RPL storing mode, nodes store routing entries for each destination in their sub-graph, limiting the size of the network, and often leading to unreachable nodes and protocol failures. We propose here RPL-mcaster, a mechanism that overcomes the scalability limitation by mending storing mode forwarding with multicast-based dissemination. Our modification has minimal impact on code size and memory usage. RPL-mcaster is activated only when memory limits are reached, and affects only the portion of the traffic and the segments of the network that have exceeded memory limits. We evaluate our solution using Cooja emulation over different synthetic topologies, showing a six-fold improvement in scalability.**

I. Introduction

RPL [1], the standard IPv6 Routing Protocol for Low-Power and Lossy Networks, has been designed to connect thousands of resource-scarce devices. The protocol creates a Destination-Oriented Directed Acyclic Graph (DODAG) topology starting from the border router, called *root*. This topology enables nodes in the network to send packets to the root while keeping only minimal routing state information in memory: by default, all traffic is simply forwarded through a preferred parent. For the root to send packets to the nodes, however, additional control messages have to be used to create the downward routes and, even more important, much more routing state needs to be stored. In the so-called *storing mode*, each node keeps a routing table with all the destinations reachable in its sub-DODAG. As it was highlighted in early deployments [2], this is the *Achilles' heel of RPL*: when used with resource-scarce devices, the routing tables fill quickly (especially near the root), impairing its scalability.

Problem statement. RPL creates downward paths by using additional control messages called DAOs: Destination Advertisement Objects. The paths are built in a reverse order as DAO messages, initiated by each potential destination node, propagate towards the root. In storing mode, DAOs are sent to a node's preferred parent, and optionally can be acknowledged by it. Then, the preferred parent is in charge of further propagating the DAOs, ultimately ensuring the sender's reachability. For example, in Fig. 1a, D announces itself as a destination to the root by sending a DAO message to its preferred parent C, which adds it to its routing table, acknowledges it, and then forwards it further up in the DODAG.

When the routing table of a node becomes full, incoming DAO messages from new destinations are simply dropped. The RPL standard specifies an optional DAO-ACK message with a *Rejection* status code (we call this a DAO-NACK) to notify the DAO sender that the recipient is unwilling to act as a DAO forwarder (e.g., because it has no more space in its routing table). Still, the standard does not define any mechanism to handle this rejection. In fact, in popular Contiki and TinyOS implementations of RPL, this DAO-NACK is not even sent. As a result, the DAO is dropped, and the path remains partially built, but useless, since the destination remains unknown to all nodes higher in the DODAG, including the root. Therefore, if there is an incoming packet destined to the destination node, the root has no choice than to drop the packet.

Back to our example in Fig. 1b, node E wants to announce itself as a destination to the root, as D previously did. However, assuming that B's routing table is limited to 2 entries, it cannot store this new destination. In current open source implementations, C has no idea that the information about E is not known higher in the DODAG. Consequently, all the packets destined to E are dropped. Even if B were to send a DAO-NACK, C would not know what to do with it.

Existing solutions [3] [4] have focused on mixing the storing mode of RPL with its non-storing mode, where the root is the only node keeping routing information. However, the binding between the mode of operation and the node is done statically, which is too rigid and potentially non-optimal.

The solution we propose in this paper, RPL-mcaster, enables downward packets to bypass the path-agnostic area of the DODAG using a multicast mechanism. All the nodes that failed to advertise a DAO (for themselves or for someone in their sub-DODAG) join a special multicast group, whose address is used by the root to send data to destinations for which it does not have a route. The normal RPL unicast operation is resumed as soon as the packet reaches a group member with a route to the destination. Compared to the previously mentioned solutions, RPL-mcaster dynamically adapts to appearance of "hot spots" in the topology. We present a detailed description of RPL-mcaster in the next section, and its performance evaluation using Cooja in Section III.

II. Our Solution: RPL-mcaster

Let us go back to our example in Fig. 1c. When C receives a DAO-NACK from B, C becomes the last node on the path

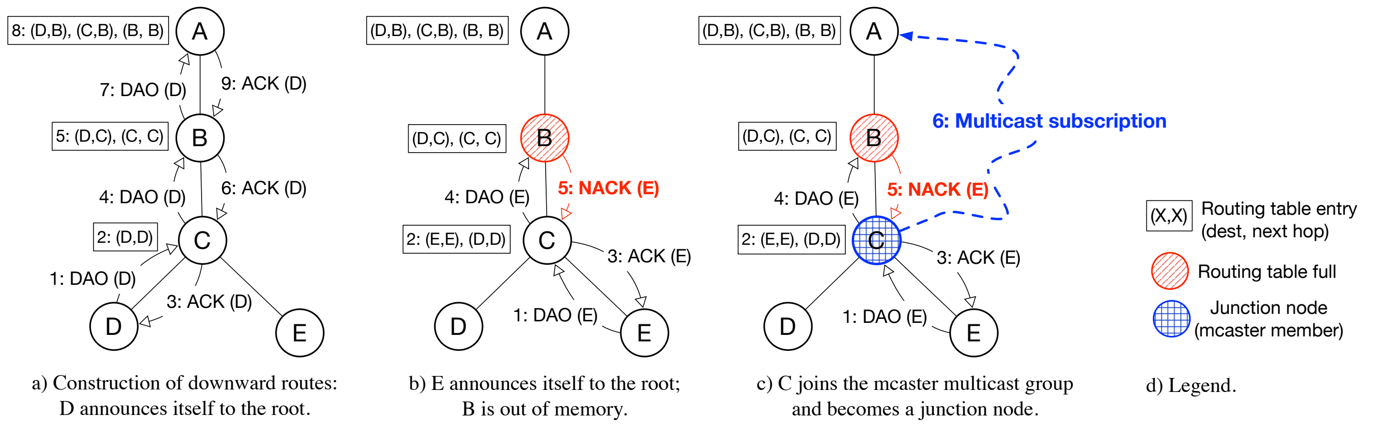

Fig. 1. The mcaster algorithm (routing table of B is limited to 2 entries).

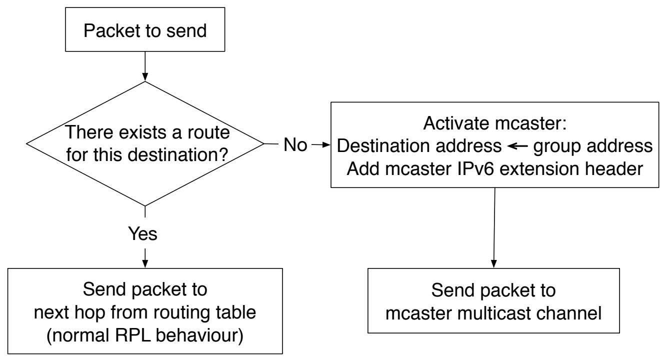
Packet to send
There exists a route for this destination?
No
Yes
Activate mcaster:
Destination address ← group address
Add mcaster IPv6 extension header
Send packet to next hop from routing table (normal RPL behaviour)
Send packet to mcaster multicast channel

Fig. 2. Flow chart of a RPL root that implements RPL-mcaster

knowing the route to E. As such, C has the critical role of ensuring E's reachability from the root. We propose that instead of making the root aware of E's existence, C rather subscribes to a special multicast group for rejected nodes: we call this the *mcaster multicast channel*. When the root wants to send a packet to E, or to any other node for which it does not have a route, it simply sends it to all the nodes subscribed to this multicast group. Since C is situated at the crossroad of two delivery mechanisms, multicast dissemination and forwarding based on routing table, we call it a *junction node*.

In essence, RPL-mcaster enhances RPL's storing mode by modifying the root's behaviour, and by proposing a straight-forward mechanism to handle DAO-NACK messages.

Root node behavior. For each packet, the root first checks its routing table for normal IP forwarding. RPL-mcaster is activated only if there is no route entry for the destination address. In this case, instead of dropping the packet, the root forwards it to the mcaster multicast channel. As we can see in Fig. 2, the root first changes the destination address to the group address, and adds a special mcaster IPv6 extension header that contains the original destination address[1]. Routing table based forwarding is resumed at the moment the packet reaches a junction node that knows the packets original destination.

Junction node behavior. By joining the mcaster multicast channel, a junction node enables the delivery of packets from the root to all the destinations in its routing table, even those for which it received a DAO-NACK. If a node receives several DAO-NACKs for different destinations, it joins the group only once. Note that a junction node should keep track of destinations for which it received DAO-NACK messages by *marking* corresponding routing table entries, for the following reason: if all the junction nodes were to forward all the packets received on the mcaster group for which they know (i.e. have a routing table entry to) the original destination, they would introduce duplicates in the network. For example, let us assume that, in Fig.1c, there is a node C' between C and E, which is a junction node for other destinations, but not for E. In this case, both C and C' would forward the packet, thus E would receive all the packets from the root in duplicate. To overcome this problem, the junction node should only act upon packets whose destination received a DAO-NACK. Otherwise, the message is dropped. Using this simple rule in our example, C is the only junction node in the network to resume routing table based forwarding for the packets that have E as a destination.

To reduce traffic overhead in space and time to the necessary minimum, nodes that have no more marked route entries can leave the mcaster group.

Implementation. In principle, RPL-mcaster could use any multicast protocol, or build on the multicast already embedded in the RPL implementation, significantly reducing the need for extra code. We have implemented RPL-mcaster on top of Contiki RPL, using the SMRF (Stateless Multicast RPL Forwarding) [5] protocol that is already part of the Contiki codebase. SMRF has no per-packet state, and only very little structure-related state. In SMRF, multicast DAO messages are initiated by joining nodes and propagated up the RPL DODAG. Nodes receiving these messages, even if they did not join the group, store the multicast address in their routing table and forward the DAO message upwards. Thus, nodes that have exhausted their memory can still use SMRF, given that only a single multicast routing table entry is reserved for the RPL-mcaster multicast address.

Our implementation of RPL-mcaster includes some mod-

[1]Other possibilities to achieve the same functionality could be the use of an IPv6 destination option header or to use generic IPv6 tunneling.

TABLE I
MEMORY USAGE

	Flash [Bytes]	RAM [Bytes]
ContikiRPL	41498	8246
SMRF	948 (+2.3%)	296 (+3.6%)
improvements	172 (+0.4%)	0
Mcaster	722 (+1.7%)	124 (+1.5%)
RPL-mcaster	43340 (+4.4%)	8666 (+5.1%)

TABLE II
SETTINGS

L3	L2: ContikiMAC	MRM
60 routing entries	125 ms sleep period	noise (dBm):
ETX metric	5 TX attempts	-90 mean, $\sigma = 1$;
MRHOF obj. func.	20 neighbor entries	path loss exponent: 3

ifications to ContikiRPL and SMRF. First of all, we have extended ContikiRPL with the sending and reception of DAO-ACK and NACK messages: a requisite for RPL-mcaster operation, but also an optional part of the standard. We have also corrected the handling of sequence numbers in forwarded DAO messages, necessary to match an ACK to the corresponding DAO. In SMRF, we improved the propagation of DAO ACK messages, and changed the separation between packet reception and forwarding, improving the group join time and reliability of SMRF, respectively. While the above changes are beneficial for RPL-mcaster, they do not affect the performance of ContikiRPL.

The code specific to RPL-mcaster includes a hook in the IPv6 packet forwarding logic to divert packets on the mcaster channel and to add the mcaster extension header; the handling of the mcaster extension header in the incoming packet processing code; an extension of routing table entries with DAO-ACK/NACK status field; and the corresponding logic in the RPL DAO-ACK handling code.

Table I shows the overall code and data memory increase and its breakdown. The cost of RPL-mcaster alone is only 722 B of Flash and 124 B of RAM, which is very small. Of course, we need to consider also the cost of the multicast layer, however, in applications that already use it, this is amortized.

III. PERFORMANCE EVALUATION

We evaluate RPL-mcaster performance using the Cooja emulator, which allows us to test exactly the binary that would run on real TelosB WSN nodes, provides full freedom in evaluating radio environments and topologies, and endless simulation of networks. We fixed the routing table to 60 entries, which is the maximum value that fits in RAM if all the RAM is allocated for the network stack, without leaving space for the application itself. ContikiMAC is used as the underlying duty-cycling MAC layer. For the radio model, we use Cooja's MRM code supporting SINR-based (Signal-to-Interference-plus-Noise Ratio) reception and capture effect, with the addition of propagation path loss exponent. Table II summarizes the most important simulation parameters.

Nodes are deployed in a 2D grid with 20 m distance from each other, and a small (± 2 m in both dimensions) randomization to avoid tie situations due to unrealistic regularity. We study the two extreme cases of root node placement in the grid: *center*, leading to a larger neighborhood for the root node, and *corner*, leading to larger hop counts. After a 4 minute warm-up period, a message with 8 B of application data is sent with UDP over IPv6 every 10 s to a randomly selected node. 500 messages are being sent in each run. Simulations are repeated 5 times with different random seeds for statistical relevance; curves show averages, bars show minimum and maximum.

Scalability. The left plot of Fig. 3 shows the packet delivery ratio (PDR) averaged over all nodes and all messages sent[2], as a function of the number of nodes in the network. ContikiRPL performance starts to degrade at 60 nodes, since the root node, and with larger network sizes also nodes near the root, run out of routing table entries. Degradation is severe, proportional to the ratio of the routing table size to the number of nodes, and independent of root node placement. RPL-mcaster yields considerably higher performance for the whole studied range. With the root in the center, PDR is still at 95% with 360 nodes, compared to a mere 13% for ContikiRPL. PDR is somewhat lower with the root in the corner, due to larger hop counts; nevertheless it exceeds 75% on average, and all the nodes remain reachable even with 360 nodes.

The center plot of Fig. 3 provides more insights into the differences in protocol operation by showing average radio duty-cycle (including both listening and TX time), and average latency (end-to-end delivery delay). For less than 60 nodes, the two protocols behave almost the same, since mcaster does not get activated. Slight differences are due to the extra signaling traffic of DAO-ACKs in the RPL-mcaster case.

Above 60 nodes, two important changes can be noticed: ContikiRPL curves flatten out, but this is due to the fact that it is delivering messages to only a small portion of the nodes. From a network operational point of view these curves, corresponding to very low PDR values, are almost irrelevant. The modified protocol operation of RPL-mcaster instead, kicks in for an increasing portion of the traffic, and its effect can clearly be seen on both curves.

The average duty-cycle of RPL-mcaster is increasing with the network size, since a growing portion of the messages are sent as link-local broadcasts instead of unicasts that, due to the way duty-cycling MACs operate, means that longer packet trains are being transmitted in each hop. Further, messages diverted to the mcaster channel are spawned to multiple junction forwarders, also increasing duty cycle; this is the price to pay for scalability in a network operating with memory-constrained devices.

Finally, the right plot of Fig. 3 shows that latency of RPL-mcaster is also increasing. Naturally, larger networks mean longer delivery paths, however a part of the increase is due to SMRF delaying packet forwarding with 1–4 cycles (125–500 ms) in each hop to avoid collision of broadcast messages, a well-known inefficiency in duty-cycled networks.

To summarize, RPL-mcaster does not alter performance in the conditions when ContikiRPL works, but it can make the

[2]For lack of space, we do not discuss per-node spatial distribution of PDR.

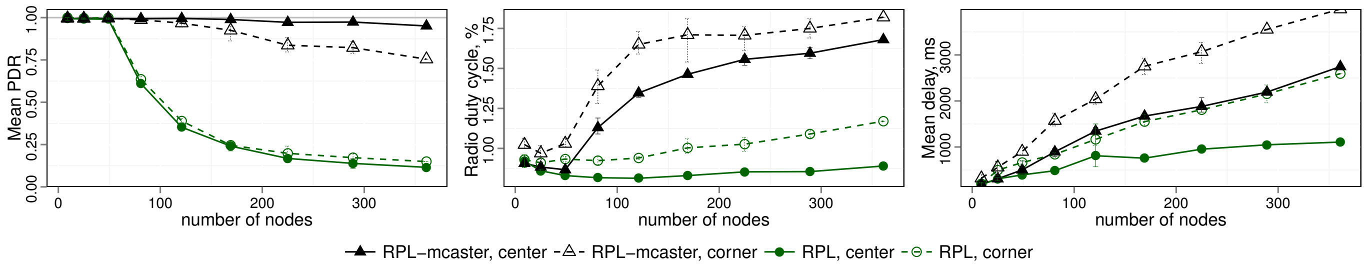

Fig. 3. Average PDR, radio duty-cycle and delivery delay vs. network size (2D grid).

network operate at scales where ContikiRPL would fail. In these latter cases, the relative cost of packets delivered by mcaster is higher than that of packets delivered by RPL, but this cost is only paid for packets that would otherwise not be delivered at all.

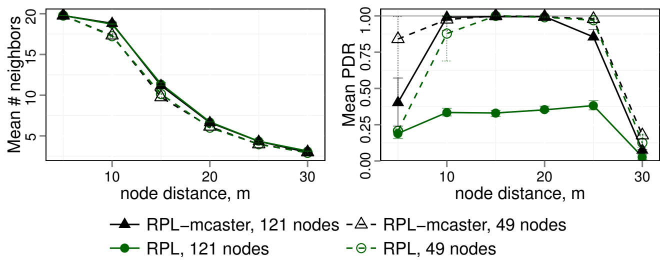

Fig. 4. Average PDR vs. network density.

Node density. Network operation—including signal interference, L2 behavior, the structure of the constructed DODAG, and the hop count of downward routes—is also greatly influenced by node density. Fig. 4 compares the performance of RPL-mcaster with ContikiRPL on 2D grids with different distances between neighboring grid nodes. To put our performance results in perspective, the left side shows average neighborhood size (N) as a function of grid distance between nodes (D). Instead of trying to give a geometric interpretation of our SINR-based radio model, we show N as seen by the Contiki protocol stack itself, i.e. as the number of nodes in the neighbor table. N is limited to 20 by default in ContikiRPL. Although this could be increased, we have intentionally kept this limit since additional neighbor entries would take away valuable RAM resources from the system. With $D = 20$ m, i.e. the value used earlier, N is around 9, i.e. nodes can communicate with their grid neighbors, diagonal neighbors, and 1–2 other nodes. $D = 25$ m corresponds to seeing the four grid neighbors only, while with $D \leq 15$ m nodes can communicate directly to multiple grid hops in each direction.

The right side of Fig. 4 shows average PDR for two network sizes: 49 nodes, where routing could work even without RPL-mcaster, and 121 nodes, where ContikiRPL is expected to fail.

At $D = 30$ m, the topology gets disconnected, thus neither protocol can deliver packets. There are also issues with dense networks, as shown for $D \leq 10$ m. When the neighbor table is saturated, ContikiRPL operates incorrectly due to the asymmetry in neighbor relations [6]. Our RPL-mcaster inherits this issue from its underlying ContikiRPL implementation. Still, it manages to alleviate the effects due to the introduction of the junction node. This solves the problem of not enough space in both the routing and the neighbor tables, improving protocol performance.

For middle densities (10 m $< D <$ 25 m), instead, density has no significant impact on average PDR.

IV. Conclusions and Future Work

Downward routing in RPL is constrained by the amount of memory allocated to routing table entries; with current open source implementations and hardware, a routing table of 50-60 entries fits in RAM if all the RAM is allocated for the network stack, without leaving space for the application itself. This is a serious limitation for a protocol that targets the LLN (low-power and lossy network) ecosystem.

We proposed the RPL-mcaster extension to the standard, promising improved scalability when the above limits are exceeded. In this initial evaluation, our fully functional prototype showed at least 6-fold scalability improvement.

Moreover, through an example of ContikiRPL's issue with large node densities, we showed that RPL-mcaster can also improve RPL's robustness.

Whether RPL-mcaster shows the same scalability improvements in real deployments with different radio condition and topologies, and how the protocol works under higher traffic loads, are still open for future evaluation.

References

[1] T. Winter *et al.*, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," IETF, RFC 6550, 2012.
[2] T. Clausen, U. Herberg, and M. Philipp, "A Critical Evaluation of the IPv6 Routing Protocol for Low Power and Lossy Networks (RPL)," in *IEEE WiMob*, 2011.
[3] J. Ko, J. Jeong, J. Park, J. A. Jun, O. Gnawali, and J. Paek, "DualMOP-RPL: Supporting Multiple Modes of Downward Routing in a Single RPL Network," *ACM Transactions on Sensor Networks*, vol. 11, no. 2, pp. 39:1–39:20, 2015.
[4] W. Gan, Z. Shi, C. Zhang, L. Sun, and D. Ionescu, "MERPL: A More Memory-Efficient Storing Mode in RPL," in *IEEE ICON*, 2013.
[5] G. Oikonomou and I. Phillips, "Stateless Multicast Forwarding with RPL in 6LowPAN Sensor Networks," in *IEEE PERCOM Workshops*, 2012.
[6] T. Istomin, C. Kiraly, and G. Picco, "Is RPL ready for actuation? A comparative evaluation in a smart city scenario," in *EWSN*, 2015.