Failure Handling in RPL Implementations: An Experimental Qualitative Study

Agnieszka Paszkowska and Konrad Iwanicki

Abstract The IPv6 Routing Protocol for Low-power and Lossy Networks (RPL) is a recognized standard for routing packets in low-power wireless networks. Its two popular implementations—TinyRPL and ContikiRPL—have been used for both research and commercial purposes. However, despite their wide adoption, qualitative studies of their behavior under various types of failures are essentially lacking.

Therefore, in this chapter, we aim to bridge this gap in a manner that may be of interest to both researchers and practitioners. More specifically, we evaluate the two implementations of RPL in a range of link and node failure scenarios. We show that whereas the implementations handle well some classes of failures, for others they exhibit undesirable behaviors or even fail completely. The results thus identify failure scenarios handling which may require additional attention before employing the implementations in real-world dependable embedded systems.

1 Introduction

A routing protocol is practically indispensable in embedded systems involving wireless low-power and lossy networks (LLNs). It allows network nodes that are out of each other's radio range to communicate by forwarding data packets via other, intermediate nodes. In effect, it alleviates several real-world problems associated with implementing and deploying LLN-based embedded systems, notably communication obstacles in the target environment or large dimensions of the environment

Agnieszka Paszkowska (✉) · Konrad Iwanicki
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warszawa, Poland, e-mail: ap321142@students.mimuw.edu.pl

H. M. Ammari (ed.), *Mission-Oriented Sensor Networks and Systems: Art and Science*,
Studies in Systems, Decision and Control 163,
https://doi.org/10.1007/978-3-319-91146-5_3

itself. If, in addition, the protocol is self-organizing, it can also significantly reduce the administrative costs inherent in configuring and maintaining such systems.

The IPv6 Routing Protocol for Low-power and Lossy Networks (RPL) is meant as such a protocol [36]. RPL is IETF's standard, incorporating state-of-the-art solutions, developed specifically to address the peculiar requirements of LLNs. It has both proprietary and open-source implementations, among which TinyRPL for TinyOS [27] and ContikiRPL for ContikiOS [6] are arguably the best known ones [23]. In particular, TinyRPL and ContikiRPL, have become inherent components of LLN protocol stacks in both research-oriented deployments and commercial solutions. In short, to deliver their functionality, multiple embedded systems rely on RPL-based networking provided by these implementations.

Nevertheless, despite the growing adoption of RPL's implementations, many of their aspects remain largely unexplored. One of such aspects is the behavior of the implementations in various failure scenarios. As we elaborate in subsequent sections, whereas the performance of RPL's implementations has been studied quantitatively, qualitative studies of their behavior under specific types of failures are essentially lacking. Without such studies, however, it is not clear to what extent the implementations can be relied upon in dependable embedded systems.

For this reason, here we aim to bridge this gap. To this end, we evaluate the two aforementioned state-of-the-art implementations of RPL—TinyRPL and ContikiRPL—in a range of failure scenarios. As the evaluation environment, we employ low-level simulators: TOSSIM [26] and COOJA [31], respectively. The scenarios cover in turn crashes of specific nodes and links, so as to affect the network topology in a particular manner, which would be hard to achieve with experiments studying robustness purely statistically. We show that whereas some of the considered failures are handled well by the implementations, for others the implementations exhibit undesirable behaviors or even fail completely. We also identify causes of such behaviors, which may be of particular interest to practitioners who are planning to use the implementations in their embedded systems. All in all, our results shed new light on the dependability of RPL's popular implementations.

The rest of this chapter is organized as follows. Section 2 gives an overview of RPL. Section 3 surveys related work. Section 4 presents our experimental methodology. Sections 5–8 discuss the results for the subsequent failure scenarios. Section 9 concludes and outlines possible future work.

2 Overview of RPL

To study the behavior of RPL's implementations, let us first give an overview of RPL itself, including details on how it provisions routes, what data structures it uses to maintain the routes and how it handles failures and topology changes, to name just a few examples. In addition to discussing the standardized aspects of the protocol, we highlight unspecified issues, which are left open to implementations.

Although many routing techniques exist for LLNs, including shortest-path routing [9], compact routing [28], hierarchical routing [19], and routing by means of geographic [22] or virtual [8] coordinates, to name a few representative examples, RPL is in principle a centralized protocol. Despite a potentially inferior performance of this technique [20], its choice can likely be attributed to simplicity.

Being a centralized protocol, RPL combines solutions for two basic traffic patterns: so-called *upward routing*, in which all low-power nodes forward their packets to a common (central) destination node, typically a border router, with so-called *downward routing*, from the border router to the nodes. These two basic patterns enable more complex ones, in particular, any-to-any routing between arbitrary nodes. However, it is upward routing, implemented by means of a distance-vector algorithm, that is the foundation of the protocol: if it does not work correctly, the other patterns will not work either. Consequently, for brevity, in our studies we focus solely on upward routing.

2.1 Preliminary Information

Recall that a routing protocol is responsible for directing packets in a LLN. More specifically, a node receiving a packet from a given *source node* to a given *destination node* needs to select the *next-hop node*, to which the packet will be forwarded, so that it can finally reach its destination. In LLNs, the next hop for a node is chosen from the nodes within the node's radio range. In RPL, such nodes are called the node's *neighbors*.

However, not every neighbor is a good candidate for the node's next hop. This is because transmitting a packet to a next hop should bring the packet closer to the destination, or, stated more formally, it should decrease a cost of the packet reaching the destination.

Therefore, each node in RPL's upward routing distinguishes a subset of neighbors that have a lower cost of sending a packet to the destination than this node. Such neighbors are called the node's *parents*. The node uses such a parent set to select a single *preferred parent*: the primary parent chosen as a default next hop for packets forwarded by the node to the given destination. The cost of sending a packet from a node to the destination is in turn called the node's *rank*.

Since when selecting a preferred parent, the node should in principle consider the neighbors with the lowest ranks, the rank aims to reflect the node's distance from the destination. However, RPL's specification does not state exactly how to measure this distance. In particular, the node's rank can depend on the requirements of a specific application and the choice of a metric to be optimized globally in the network when routing packets [35], for example, hop count, latency, or estimated transmission count, ETX [5].

Conceptually, nodes and the wireless links to their parents form a directed graph. Since when the network is stable a node's rank is always greater that its parents' ranks, the graph does not contain cycles and, consequently, is a directed acyclic

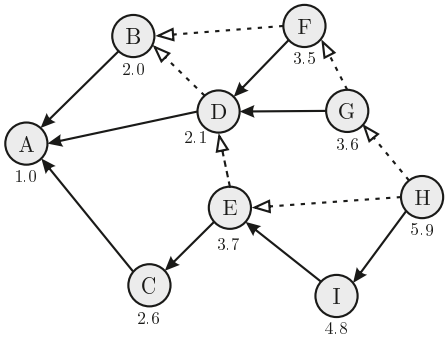


Fig. 1 An example of a DODAG

graph (DAG). The destination, for example, a border router, is the sink in the DAG: it has no outgoing edges. Therefore, the graph is often referred to as a destination-oriented DAG, abbreviated as *DODAG*. The destination—the sink of the graph—is in turn called the *root* of the DODAG.

Figure 1 presents a sample DODAG rooted at node A, with nodes' ranks displayed as numbers. Links between nodes and their parents are represented by lines with arrows pointing at the parent. A solid line leads to a preferred parent; a dashed one to an alternative parent. Node A's rank equals 1.0 and is the smallest rank in the network because A is the DODAG's root. For the same reason, A does not have any outgoing edges. D, an internal node, has in turn five neighbors: A, B, E, F, and G. Two of them, A and B, have their ranks lower than D's rank, 2.1, and, consequently, D considers them as parents: A is D's preferred parent and B is an alternative one. Although D has chosen as its preferred parent the neighbor with the smallest rank among its parents, such a choice is not imposed by the protocol. For example, E has C with rank 2.6 as its preferred parent, despite having D with a lower rank, 2.1, in its parent set. The rest of D's neighbors, E, F and G, have their ranks greater than D's rank, and, therefore, are D's children in the DODAG. Moreover, D is both F's and G's preferred parent. As a result, when, for instance, G wants to send a packet to the root node A, it first transmits the packet to D, which forwards the packet directly to A, yielding a hop count of 2. In contrast, the maximum number of hops to deliver a packet in the DODAG is 4 and is required when the packet originates at node H.

2.2 Packet Forwarding

RPL's DODAG thus describes all routes via which nodes can forward packets to the root node. However, the actual packet forwarding at each node is delegated by RPL to the node's IPv6 stack. To this end, RPL running at the node registers the node's preferred parent's IPv6 address as the default route in the node's IPv6 routing table. Based on this table, the node's IPv6 stack selects next hops for forwarded packets.

Nevertheless, in spite of the delegated packet forwarding, RPL's implementations should provide endpoints for inserting and verifying a special option in the headers of routable packets [15]. The option contains, among others, information used for loop and error detection: a forwarding node's rank and three flags: 'O', 'R', and 'F'.

Recall that a node's rank is meant to be greater than its preferred parent's rank, which aims to avoid routing loops during normal operation. However, such loops may appear under failures, notably when the node has outdated information on its neighbors' ranks. Therefore, an additional loop-detection mechanism is needed to quickly react to such rank inconsistencies. The mechanism detects an inconsistency whenever the direction of a forwarded packet (the 'O' flag in the option) does not match the rank relationship between the node transmitting the packet and the neighbor receiving it. One case is when the packet is going upward (the 'O' flag is clear) but the transmitter's rank in the packet is lower than the receiver's rank. Another case is the packet going downward (the 'O' flag is set) with the transmitter's rank in the packet being greater than the receiver's.

2.3 Parent Selection and Rank Computation

To register a default route, each node needs to select its preferred parent. This process is inherently coupled with the node's rank computation: the node's rank and preferred parent are chosen with a so-called *objective function*, which we abbreviate as *OF*. As far as objective functions are concerned, RPL's specification allows for much flexibility. First, new objective functions can be introduced, so that computed ranks and the method for selecting preferred parents would meet the requirements of particular applications. Second, OFs are configurable per node.

To illustrate how RPL uses objective functions, recall that each node maintains a parent set, a subset of neighbors from which the node chooses its preferred parent. An entry in the parent set contains, among others, a parent's address, rank and routing metric values for the parent and/or the link from the node to the parent. Since the node computes its own rank locally, it has to exchange information on its ranks with its neighbors by means of control traffic. The entries in the node's parent set are thus inserted and updated as a result of receiving control messages, which will be discussed in more detail shortly.

The parent set is an input to an objective function, which the node uses to choose its preferred parent. Parent selection should in principle be performed whenever the parent set has potentially changed, for example, upon reception of a control message with a neighbor's rank update or parent unreachability detection [30]. As soon as the node selects its preferred parent, it registers the route to the parent in its IPv6 routing table as the default route.

The objective function is also responsible for computing the node's rank after a change of the preferred parent or the metrics associated with the parent. The algorithm for rank computation is OF-dependent. Nevertheless, RPL's specification provides guidelines to be followed by every OF.

First of all, the rank value needs to reflect the node's distance from the root and has to be strictly increasing along the routes in the DODAG. To ensure the latter, the protocol uses a *MinHopRankIncrease* parameter, which defines the smallest permitted increase in rank per hop. As a result, the root's rank must be the smallest rank in the DODAG and every other node's rank must be higher than any of its parents' ranks and exceed its preferred parent's rank by at least *MinHopRankIncrease*. This feature is made use of by the aforementioned loop detection mechanism.

Furthermore, to enable breaking loops in the DODAG, which may occur upon failures, a node's rank must not increase by more than *MaxRankIncrease* within one so-called *version* of the DODAG. Namely, a node keeps track of the smallest rank it has computed in the current DODAG version, and, whenever its rank starts to exceed that smallest rank by more than *MaxRankIncrease*, it concludes that the DODAG is broken and disconnects from it by adopting a null preferred parent and an infinite rank. However, since the change in its rank may be permanent, for instance, as a result of a massive failure, DODAG versioning enables a complete reconstruction of the DODAG. Such as process is initialized by the root generating the DODAG's new version number, either periodically or as a result of an external event (e.g., a request from the network administrator). Upon discovering the new version, a node can choose a completely new rank, not constrained by its rank in the old version.

Although parameters *MinHopRankIncrease* and *MaxRankIncrease* exist irrespective of an objective function, their values may potentially depend on the particular OF employed. Currently, there are two popular OFs.

First, all implementations of RPL are required to implement the Objective Function Zero, OF0 [33]. In OF0, a node's rank equals the count of hops from the root to the node. The criteria taken into account in the parent selection process include both the candidate's rank and the quality of the link to the candidate. It is, however, the candidate's rank that influences the result the most.

Second, in practice the Minimum Rank with Hysteresis Objective Function, MRHOF [12] is commonly used. With MRHOF, a node uses a metric value retrieved from an additional option in control messages to evaluate parents and compute its rank. Various metrics can be used, for example, hop count, ETX, latency, though they need to be additive. The node's rank is calculated from the candidate parent's metric value incremented by the link metric to the candidate parent. The candidate with the lowest path cost is chosen as the node's preferred parent but only if it is significantly better than the current parent. This mechanism of not choosing the best parent unless truly beneficial is called *hysteresis* and is introduced in MRHOF to make the DODAG more stable.

2.4 Control Traffic

The algorithm for choosing parents and calculating ranks generates control traffic through which the nodes exchange information required by OFs. RPL introduces several types of control messages, implemented as ICMPv6 [4] messages.

The first one, DIO (DODAG Information Object) messages, is responsible for advertising nodes' ranks: a DIO message sent by a node is an advertisement that the node can serve for its neighbors as a next hop on their routes to the DODAG root. The root of a DODAG systematically sends such a message. A neighbor receiving the message can join the DODAG and select the sender node as its preferred parent. When it sends its own DIO, its neighbors can do the same, and so on.

A DIO message thus needs to contain all the information necessary for joining a DODAG, such as the DODAG's identifier, version number and configuration, and for considering the sender as a parent, for instance, information on the sender's rank and metric values. The configuration of a DODAG contains, among others, the identifier of the objective function to be used by the nodes to choose their parents and compute their ranks and the two aforementioned parameters used in rank computation and maintenance: *MinHopRankIncrease* and *MaxRankIncrease*.

There is a trade-off between keeping the time of reaction to network changes low and generating little control traffic for DODAG maintenance. To exploit this trade-off, each node employs a so-called Trickle timer [25] for coordinating the control traffic. After starting, the node sets its timer period to t_{min} (on the order of milliseconds) and doubles the next period whenever the previous one finishes until the period reaches t_{max} (on the order of minutes or even hours). However, the node can reset the period back to t_{min} whenever it observes an important change or an inconsistency in the DODAG.

The Trickle timer serves as the node's scheduler for DIO messages. At the beginning of each period, the node schedules the timer to a random time in the second half of the period. When the timer fires, the node broadcasts a DIO message to its neighbors unless the number of DIO messages it received in the current period exceeds some threshold. As a result, initially, nodes exchange many DIO messages and thus quickly construct a DODAG. However, after this short period, the frequency of DIO messages gradually decreases and stays low as long as the network is stable. In effect, not only can changes in the network be accounted for rapidly but also few messages are exchanged when the DODAG does not change, which minimizes the control traffic.

The second type of RPL control messages, DIS (DODAG Information Solicitation) messages, are used by nodes to actively ask their neighbors to provide information on the DODAGs they are aware of. This results in a quicker reaction to network changes at the cost of an increase in control traffic. A node broadcasts such DIS messages periodically until it joins a DODAG. In contrast, when the node receives such a message, it resets its Trickle timer so that it can quickly send back a DIO response.

Another use-case for DIS messages is probing. The aim of probing is to keep parent routing metrics up-to-date. Since probes are unicast messages, the response DIOs are sent directly to the probe's sender without a reset of the responding node's Trickle timer.

The other control messages introduced by RPL are not related to upward routing. Therefore, we omit them here for brevity.

2.5 Open Issues

RPL's specification does not provide solutions for all issues regarding the protocol, leaving them open to implementations. To give some example, while it is specified when the Trickle timer for DIO messages must be reset, the list is not exhaustive. Implementations may thus choose to reset the timer in other situations. Another example is the parent choice and rank computation, which are both OF- and implementation-dependent. Nevertheless, the specification provides guidelines that must be followed by all OF implementations. Finally, issues regarding routing metric maintenance and neighbor unreachability detection are unspecified and left open to implementations. All in all, the implementations have some freedom with respect to the way they provide RPL's functionality. This allows for adapting them to various applications, which is another advantage in embedded systems.

2.6 RPL Implementations

As a result of the standardization, RPL has been implemented in several operating systems for low-power wireless embedded devices. There thus exist both proprietary and open-source implementations of RPL, among which TinyRPL for TinyOS [27] and ContikiRPL for ContikiOS [6] are arguably the most widely known ones [23]. Therefore, we will focus on these two implementations in our analysis.

In short, both implementations are built on top of the IPv6 stacks of their underlying operating systems. They provide all basic features described in RPL's specification. In addition, supported by the research community, they try to integrate state-of-the-art solutions for the issues the specification leaves open. In particular, from the perspective of our study, it is worth to mention that for routing metric maintenance and neighbor unreachability detection, the implementations employ algorithms drawing from numerous studies on the performance low-power wireless communication and wireless link quality estimation.

What is more, the implementations have been widely tested and deployed in the real world. In effect, various embedded systems have appeared that rely on these implementations to deliver their own functionality.

3 Related Work

Because of this increasing popularity, RPL has been extensively evaluated in both simulations and the real world. However, whereas RPL's implementations have been studied thoroughly from the performance perspective, their behavior under specific types of failures has received significantly less research attention.

To start with, initial empirical studies of RPL focused on its efficiency. Results obtained in early experiments, both in simulations [34, 3] and on testbeds [10, 3],

were promising but also unveiled first problems resulting from RPL's underspecification [3]. In particular, it was shown that although RPL successfully detected simple failures [10], its DODAG reconstruction algorithms were not very efficient [34].

As RPL's specification was being clarified and improved, the protocol's performance was being further evaluated in various settings. To illustrate, Gaddour et al. [11] studied the efficiency of ContikiRPL during DODAG formation for different network topologies. Istomin et al. [17], in turn, evaluated the performance of actuation in downward routing. Likewise, RPL's performance was analyzed for different parameter configurations, including objective functions [2] and network-layer metrics [16], to name just two examples. Finally, studies on interoperability of the two aforementioned implementations of RPL—ContikiRPL and TinyRPL—showed that while both implementations perform well on their own, they demonstrate surprising performance artifacts when run together [23].

Furthermore, RPL's performance has been compared to that of other routing protocols, for instance, the Collection Tree Protocol [10], dissemination-based protocols [17], and protocols for Internet-enabled wireless sensor networks [32]. Recently, RPL has also been extended into new routing protocols or even entire protocol suites [7, 1, 29].

All in all, those research activities have made RPL an attractive solution for industry, where dependability is an important feature. As a result, RPL's implementations were evaluated in harsh conditions: under different levels of radio interference [13, 29] and in various failure scenarios, featuring local failures [24], failures of large groups of randomly selected nodes [14], of a few albeit critical nodes [14, 21], and of the DODAG root [18]. Although earlier of those results [14, 21, 24] suggested that RPL's implementations are by and large stable and correctly handle failures, our recent study [18] showed that those conclusions may have been too optimistic. More specifically, we observed that ContikiRPL has surprising problems dealing with a crash of the DODAG root, which is problematic in real-world systems because, being typically more complex than sensor nodes and relying on a tethered power supply, DODAG roots are prone to failures such as power outages.

Therefore, as a follow up on this observation, here we evaluate two popular implementations of RPL—TinyRPL and ContikiRPL—in a much broader range of failure scenarios and with different configuration parameters. Such a qualitative study may thus be of interest to both researchers and practitioners because it allows for a better understanding of conditions under which the implementations may be relied upon.

4 Experimental Methodology

Let us start our study by describing the experimental setup we used for the two implementations of RPL. We discuss the environments in which the implementations were evaluated, including the experimental application, configuration parameters,

and performance metrics. We also explain the link and node failure scenarios in which we tested the implementations.

4.1 Experimental Environments

To facilitate reproduction of our results, we evaluated the implementations in publicly-available simulators. For the TinyOS implementation of RPL, TinyRPL, we employed the TinyOS simulator, TOSSIM [26]. Similarly, for the ContikiOS implementation of RPL, ContikiRPL, we utilized the Contiki network simulator, COOJA [31]. Both environments are low-level simulators: they normally simulate actual implementations of protocols—not their simplified models—and the obtained results typically well predict the protocol's real-world behavior.

As to the implementations themselves, we analyze the latest version of TinyRPL, that is, one from January 5th, 2017. In contrast, when it comes to ContikiRPL, we focus mainly on an the last stable version, 3.0, available at the time of writing this chapter rather than on the latest one from the repository, that is, the one from January 5th, 2017. This is because the development version performs worse compared to the stable one, which we highlight in our experiments.

4.2 Experimental Settings

The experimental application, which generated network traffic to be routed by RPL, was designed to model a common communication pattern in LLNs: all-to-one data collection. More specifically, each node generated short, one-frame data packets that were forwarded by the network to the root. The interval between two consecutive packets generated by a node was chosen at random between T and $2T$ time units, where T was a configuration parameter. This resulted in relatively uniform multipoint-to-point traffic.

The experimental runs of the application presented here did not use any radio duty cycling techniques but we did verify that the results with duty cycling did not diverge from the presented ones. This choice is because duty cycling is a task of the link layer, not RPL, and, as such, should not influence the *correctness* of RPL's implementations. Unless the network operates close to its maximal capacity, which is usually not a normal situation in LLNs, and hence is not the case in our experiments, the only observable effects on RPL of employing duty cycling are higher latencies and lower throughput.

In order to test the implementations under various conditions, we conducted experiments with different configuration parameters. Nevertheless, we provide default values for the parameters that, unless noted otherwise, were used in the experiments. The default duration of an experiment was 1 simulated hour. Nodes generated packets to be forwarded to the root every 10–20 seconds ($T = 10s$). The distinguished

node acting as the DODAG root was the node with id 0. The highest possible increase in a node's rank within a DODAG version (*MaxRankIncrease*) was by default 7, while the maximum number of hop-by-hop retransmissions of a packet was 5 per hop. Finally, the experiments used either OF0 or MRHOF as the objective function.

We tested the implementations in so-called *unit-disk* topologies, which are often used in theoretical analyses. In our case, they consisted of 121 nodes evenly distributed over an 11-by-11 grid. The radio range of each node was a circle with the center at the node and radius r, where r ranged from 1 to 4. In other words, each node had a perfect link to every node at a distance lower than or equal to r, and did not have a link to any node at a distance greater than r. While this model is an idealized one, it is suitable for our purposes: if a protocol does not behave correctly in this idealized model, it is highly unlikely that it will behave correctly in the real world. Intuitively, in the unit disk model, a node can detect with a perfect accuracy whether a link is up or down, which is crucial for proper neighbor unreachability detection. In contrast, in the real world, there is no perfect failure detector. In particular, there can be false positives that trigger unnecessary topology changes.

The results presented in the remainder of this chapter were gathered during representative runs of the test application. Their analysis focuses mainly on the correctness but also on the efficiency of the evaluated implementations. The first group of metrics aims to examine the DODAG construction process. To this end, we tracked the number of nodes with a preferred parent and a valid path to the root with respect to time, and an average rank of a node in the network. Second, we evaluated the stability of the constructed DODAG by tracking the number of nodes' preferred parent changes, Trickle timer resets, and generated control messages. Finally, we used the last group of metrics, including metrics related to the network traffic and end-to-end delivery rates, to evaluate the efficiency of resource usage by the nodes in the network and the reliability of the network.

4.3 Experimental Scenarios

The experimental scenarios were designed to examine how the considered implementations of RPL behave in various conditions, in particular, under different link and node failures.

However, the first scenario was free of failures and aimed to analyze how long it takes the implementations to build the DODAG and how the implementations behave when the network is stable. These results are used as a base line for analyzing the results of subsequent experiments.

The second group was experiments with link failures. The scenarios in this group included failures of both individual links and sets of links but here we focus on single-link failures. None of the failures resulted in a network partition. Moreover, we analyzed the consequences of a link failure depending on the importance of the link. In particular, we considered experimental scenarios in which a failing link was

responsible for forwarding packets from either a large part of the network or just a single node.

The next group of experimental scenarios concerned failures of non-root nodes. Similarly to the previous group, the scenarios included failures of both individual nodes and sets of nodes. Like previously, none of the failures analyzed in this group resulted in a network partition. The failing nodes were either important ones, responsible for forwarding packets from a large part of the network, or leaf nodes, not forwarding any packets except for their own. For brevity, however, we focus on failures of only important nodes.

Finally, the last group encompassed scenarios with failures leading to network partitions. One example was failures of DODAG roots. Other examples are both link and node failures as a result of which all paths from some nodes to the root were broken. Nevertheless, we focus on the DODAG root failures as they are extreme cases of network partitions. Additionally, we tested scenarios in which the failing links or nodes recovered after failures, thus rebuilding the broken paths.

All in all, our experimental scenarios cover a broad spectrum of crash-failures that may occur in the real world. As such, they truly stress the considered implementations of RPL, so that we can study various claims about their fault tolerance.

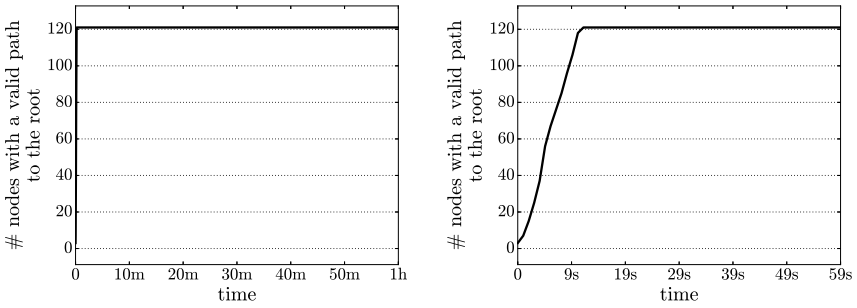
5 Experiments without Failures

As mentioned in the previous section, we begin our analysis by presenting the results of experiments without failures. Recall that the evaluation of the implementations in the failure-free environment aims to examine how quickly the implementations construct DODAGs and whether the DODAGs reach stable states.

To start with, Fig. 2 presents the number of nodes with a valid path to the DODAG root during the course of a representative experiment conducted with TinyRPL. We define a valid path from a node to the root as a sequence of nodes starting at the node and ending at the root in which: (1) all nodes are correct and (2) every two consecutive nodes are connected by a correct link and are in a child-preferred-parent relation. We consider a DODAG as constructed when each node has a valid path to the DODAG root and is thus able to forward packets to the root.

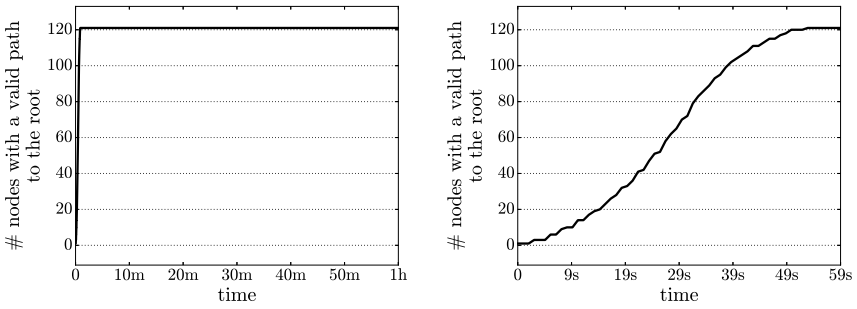
It can be observed in the figure that TinyRPL constructs a DODAG within seconds. More specifically, as shown in Fig. 2(b), plotting the metric values in the first minute of the experiment, the entire construction process took only 13 seconds for the most sparse of the analyzed topologies: the unit-disk topology with radius 1. Moreover, as can be observed in Fig. 2(a), once constructed, the DODAG was stable during the remaining period of the experiment, that is, all nodes always had valid paths to the root.

Figure 3 presents the same plots for ContikiRPL. It can be observed in Fig. 3(b) that constructing the DODAG for the same topology (i.e., the sparsest one) took ContikiRPL almost a minute, which is 4 times more than in the case of TinyRPL. The reason for this is a greater default minimum Trickle timer interval (t_{min}) in Con-



(a) unit disk, $r = 1$; duration = 1h; MRHOF **(b)** unit disk, $r = 1$; duration = 1m; MRHOF

Fig. 2 Duration of DODAG construction in TinyRPL



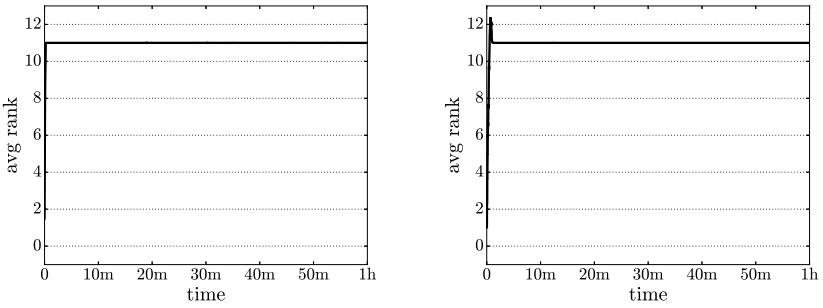
(a) unit disk, $r = 1$; duration = 1h; MRHOF **(b)** unit disk, $r = 1$; duration = 1m; MRHOF

Fig. 3 Duration of DODAG construction in ContikiRPL

tikiRPL than in TinyRPL and, consequently, a lower pace of DODAG information dissemination via DIO messages. Similarly to TinyRPL, the DODAG built by ContikiRPL remained stable, as shown in Fig. 3(a): after the initial construction period, all nodes had a valid path to the root throughout the rest of the experiment.

What is more, from Fig. 4, which plots the average DODAG rank of a node, it can be deduced that both implementations were able to construct optimal DODAGs. To explain, with the smallest allowed increase in rank per hop (*MinHopRankIncrease*) equal to 1 and the unit-disk links, a node's rank in an optimal DODAG would be equal to the node's Manhattan distance to the root plus 1. Consequently, the average rank of a node in an optimal DODAG would be equal to 11. It can be observed in the figure that the average node rank in the constructed DODAGs was indeed 11 (or at least close to 11 because of the limited resolution of the figure).

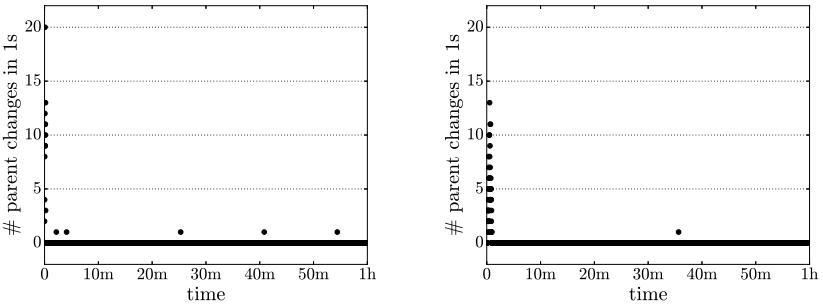
The metrics presented in the next three figures are used to analyze the stability of the DODAG after the initial construction period and the overhead of the control traffic in the period in which the DODAG should be stable.



(a) TinyRPL **(b)** ContikiRPL

Fig. 4 Average rank (unit disk, $r = 1$; MRHOF)

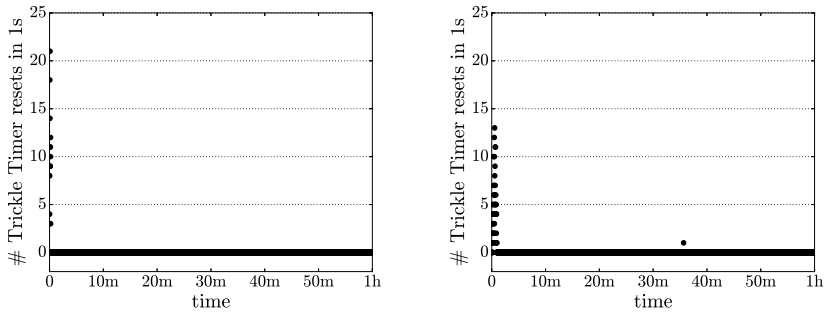
To start with, the plots in Fig. 5 present the number of preferred parent changes in an hour-long experimental run in the unit disk topology with radius 1. Figure 6, in turn, shows the number of node Trickle timer resets in the same experiment. Recall that a node resets its Trickle timer whenever it observes an inconsistency or a vital change in the network. A large number of Trickle timer resets is thus an indicator of a DODAG's instability.



(a) TinyRPL **(b)** ContikiRPL

Fig. 5 Parent changes (unit disk, $r = 1$; duration = 1h; MRHOF)

As can be observed in Fig. 5(b), a DODAG constructed by ContikiRPL in the initial period stabilized after 1–2 minutes and changed only once throughout the whole experiment. This matches well Fig. 6(b), which shows that all but one resets of node Trickle timers happened in the construction period. In contrast, TinyRPL did not build a final version of the DODAG in the initial period of the experiment. For this reason, a few preferred parent changes during the course of the experiment can be observed in Fig. 5(a). Nevertheless, these few changes were not vital enough

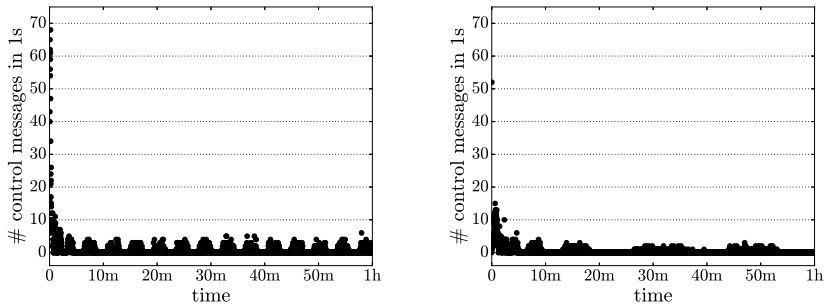


(a) TinyRPL **(b)** ContikiRPL

Fig. 6 Trickle timer resets (unit disk, $r = 1$; duration = 1h; MRHOF)

for the nodes to reset their Trickle timers. Consequently, there were no Trickle timer resets corresponding to these parent changes, as shown in Fig. 6(a).

Figure 7 illustrates the control traffic in the analyzed experiments. Control messages tracked in the plots are mainly DIO messages, scheduled by the nodes' Trickle timers. As a result, at the beginning of the experiment, when the Trickle periods were short, we can observe a large number of transmitted control messages. However, as soon as the DODAG stabilized and the periods of the node Trickle timers reached their highest values, the control traffic decreased and stayed low as long as the DODAG was stable.



(a) TinyRPL **(b)** ContikiRPL

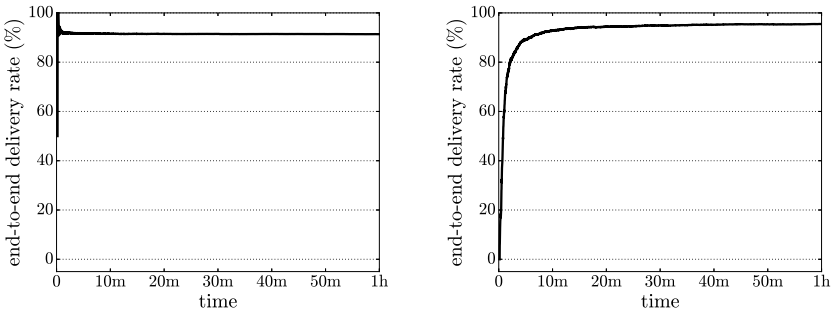
Fig. 7 Control messages (unit disk, $r = 1$; duration = 1h; MRHOF)

As can be observed in both plots in Fig. 7, the control traffic was not uniform. In other words, periods of control message transmissions were interleaved with periods when there was no control traffic. There are two reasons for this phenomenon. First, since all nodes were booted roughly at the same time and almost immediately

constructed the DODAG, their Trickle period starts were close in time. Second, the nodes scheduled their DIO messages in the second half of the Trickle period, thus leaving the aforementioned gaps in the control traffic. Moreover, the longest Trickle period for each implementation can be easily computed from the figures by summing the lengths of the longest gap and the following busy period. Accordingly, the longest Trickle period is about 4 minutes for TinyRPL and about 16 minutes for ContikiRPL, which matches their configurations. Consequently, when the DODAG is stable, out-of-the-box ContikiRPL generates less control traffic than out-of-the-box TinyRPL but this configuration can be changed.

The subsequent charts present results for metrics that were used to evaluate the reliability and resource consumption of the two implementations of RPL. The reliability was assessed by tracking *end-to-end delivery rate*. It was defined as the percent of all data packets generated by the nodes from the beginning of the experiment that were successfully delivered to the root. Resource consumption was in turn assessed by measuring *network traffic*, that is, the accumulated number of control and data packets generated by the nodes, the number of hops taken by those packets, as well as the number of physical radio transmissions of the packets. Since radio communication is typically the most resource-consuming activity of low-power wireless nodes, network traffic serves well as a proxy metric for resource consumption.

Figure 8 presents the end-to-end delivery rate for two hour-long experiments evaluating both implementations. As can be observed in Fig. 8(a), although the network did not suffer from any failures throughout the experiment, the end-to-end delivery rate for TinyRPL did not reach 100% but remained at the level of 90%. In other words, on average 1 data packet out of 10 was lost on its route toward the root and, consequently, did not reach the root. The reason for this was packet collisions caused by the small radio range of the nodes in the utilized topology and the high packet generation frequency ($T = 10s$). On the other hand, due to the quick DODAG construction in TinyRPL, even packets generated by the nodes at the beginning of the experiment were successfully delivered to the root, which is visible in Fig. 8(a) as the high end-to-end delivery rate in the first seconds of the experiment.

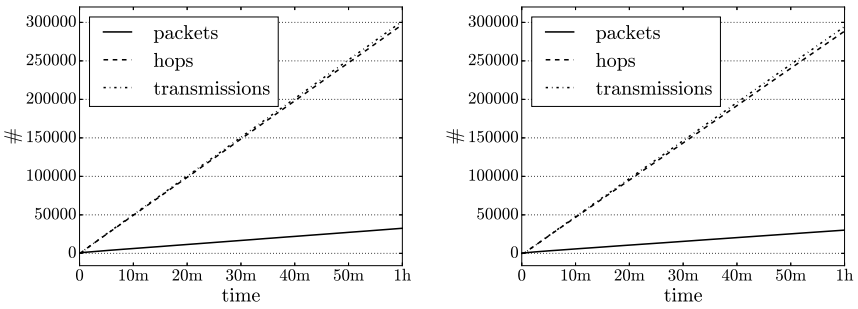


(a) TinyRPL **(b)** ContikiRPL

Fig. 8 End-to-end delivery rate (unit disk, $r = 1$; duration = 1h; MRHOF)

In Fig. 8(b), in turn, we can see that the final end-to-end delivery rate for ContikiRPL after one hour was slightly higher than that for TinyRPL. However, due to the longer DODAG construction period in ContikiRPL, the end-to-end delivery rate was low in the first seconds of the experiments and reached its highest value only after some period.

Figure 9, in turn, shows the network traffic in the two experiments. Packets, represented in the figure by a solid line, include both data packets generated every 10–20 seconds by each node and control packets carrying mainly DIO messages scheduled by the nodes' Trickle timers. Nevertheless, the number of control packets did not exceed 5% of the total number of packets. This also influenced the aggregate number of hops taken by the packets because the number of hops for a packet depends on the packet's type. In the analyzed configuration, it was always 1 for a control packet and on average 10 for a data packet. The transmissions in the figure, in turn, correspond to physical radio transmissions. Since control packets are broadcast and not acknowledged, the nodes never retransmit them. As a result, there was always 1 transmission for a single control packet. In contrast, due to transmission collisions, data packets were sometimes unacknowledged and required retransmitting. The maximum number of transmissions per data packet was limited by the value of the aforementioned configuration parameter, which was equal to 5. In practice, however, as can be observed Fig. 9, retransmissions were occasional in both implementations and did not incur much overhead on resource consumption.

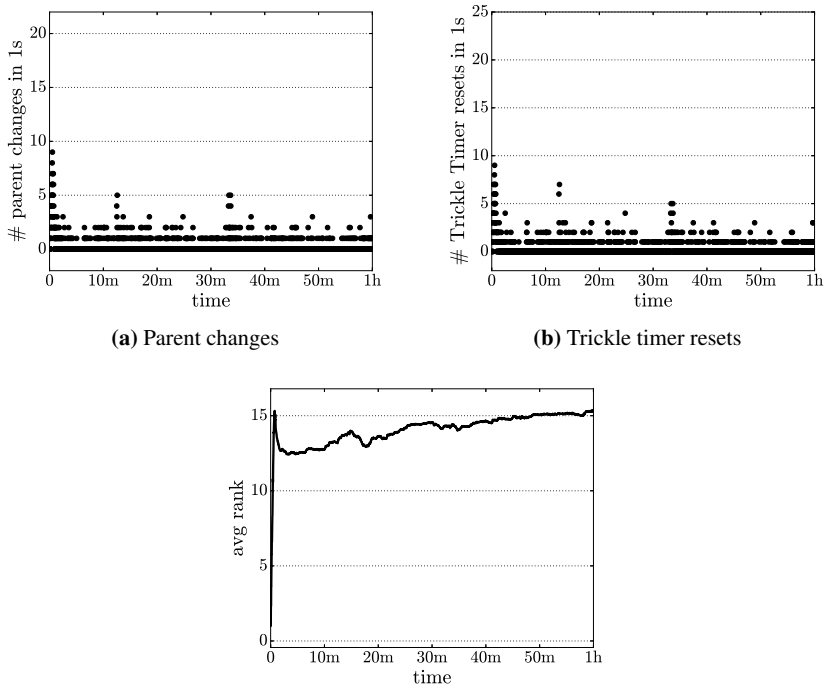


(a) TinyRPL **(b)** ContikiRPL

Fig. 9 Network traffic (unit disk, $r = 1$; duration = 1h; MRHOF)

As mentioned previously, the analysis hitherto has not concerned the latest development version of ContikiRPL available from the repository at the time of this writing (i.e., the one from January 5th, 2017). This is because that version of ContikiRPL does not construct a stable DODAG. More specifically, as can be observed in Fig. 10(a), which plots the same experiment but with the latest version of ContikiRPL, a few nodes changed their preferred parent almost every second of the experiment. In contrast, as we have already shown in Fig. 5(b), only one node changed

its preferred parent after the construction phase in the experiment with the earlier version, selected for our analyses. The increased number of parent changes also resulted in an increased number of observed Trickle timer resets, as can be verified in Fig. 10(b), and highly unstable ranks, visible in Fig. 10(c). This, in turn, increased the accumulated network traffic (not plotted), and hence, resource consumption. For these reasons, we settled on the earlier, yet more predictable and stable version of ContikiRPL.



(a) Parent changes

(b) Trickle timer resets

(c) Average rank

Fig. 10 ContikiRPL (latest version) (unit disk, $r = 1$; duration = 1h; MRHOF)

5.1 Summary

To sum up, the experiments in the failure-free environment have shown that both analyzed implementations of RPL quickly construct stable DODAGs. Moreover, when used for routing, the constructed DODAGs deliver the majority of packets without wasting network resources. It can thus be concluded that the implementations, at least some of their versions, behave as expected in the failure-free environment.

However, real-world networks are rarely free of failures. Therefore, in the next sections we proceed to failure-oriented experimental scenarios.

6 Experiments with Link Failures

We start our failure-oriented experiments with scenarios involving link failures. We consider a link as failed if every transmission over this link is lost, that is, it is not received by the target node. We evaluate how quickly the implementations react to link failures, depending on the chosen objective function and other configuration parameters. We also examine the consequences of link failures on the stability of the DODAG and reliability of packet forwarding.

Not all links in the network are equal. Depending on the network topology and the shape of the DODAG, some links are "important" in that packets from a significant fraction of nodes are forwarded through them to the DODAG root, whereas others are "unimportant" in that they are utilized by single nodes or not utilized at all. Moreover, the combinations of links that may fail are numerous. Since we are unable to test all link failure scenarios in this section, we analyze two selected ones: a failure of a single "important" link and a failure of a single "unimportant" link.

The experiments presented in this section lasted for two simulated hours; link failures occurred after the first hour. Such a timespan was chosen because it can be concluded from the previous section that an hour is enough for both implementations to construct DODAGs and stabilize.

6.1 Important Link Failure

In order to simulate a failure of an "important" link, we removed the link between nodes with identifiers 0 and 1 in the unit-disk topology with radius 1, as visualized in Fig. 11. The removed link was thus one of two links leading directly to the DODAG root and, as such, forwarded roughly half of data packets generated by all nodes.

MRHOF

Figures 12–14 compare the behavior of the two implementations of RPL with MRHOF as the objective function in response to the analyzed link's failure.

As can be observed in Fig. 12(a), ContikiRPL reacted to the failure quickly: within seconds. In contrast, as visible in Fig. 12(b), it took TinyRPL about 3 minutes to rebuild the DODAG, so that each node would again have a valid path to the root. This difference can be attributed mostly to the slightly different policies for resetting the Trickle timer in the two implementations.

More specifically, Fig. 13 presents the occurrences of Trickle timer resets caused by the failure. It can be observed in Fig. 13(b) that in TinyRPL the first node reset

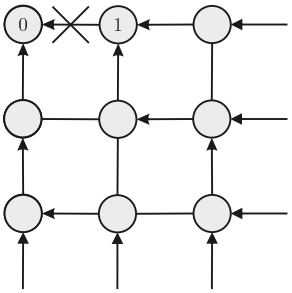
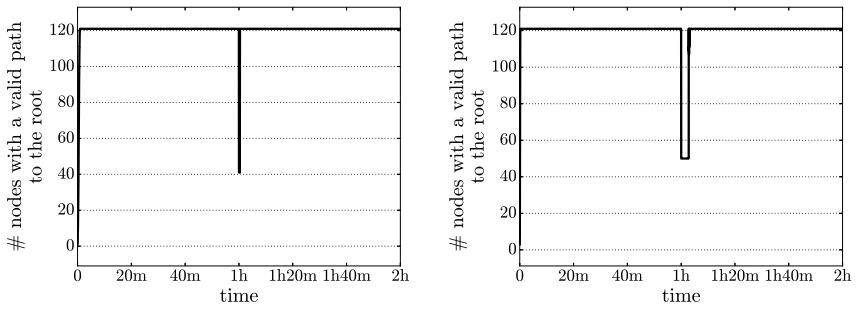


Fig. 11 Failure of an important link



(a) ContikiRPL **(b)** TinyRPL

Fig. 12 Nodes with a valid path to the root (unit disk, $r = 1$; MRHOF)

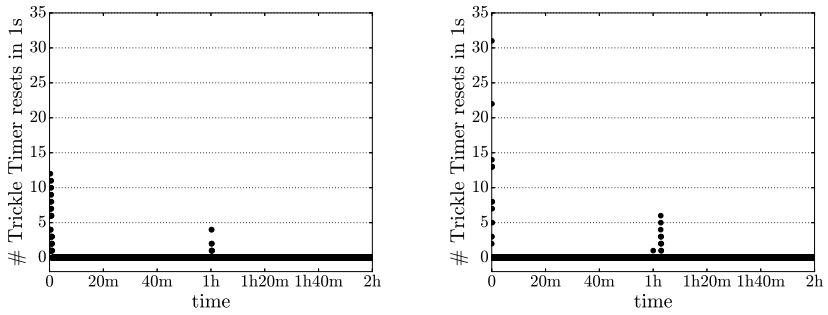
its Trickle timer immediately after the failure. However, the DODAG was not fully rebuilt until other nodes reset their timers 3 minutes later. In ContikiRPL, in turn, all additional resets of the node Trickle timers occurred immediately after the failure, as visible in Fig. 13(a), and, therefore, the failure was handled more quickly.

Figure 14 depicts an increase in control traffic following the Trickle timer resets after the failure. In both implementations, an increase in the control traffic can indeed be observed. Nevertheless, it is not significant compared to the stable period in the experimental scenarios without failures, as presented in Fig. 7.

OF0

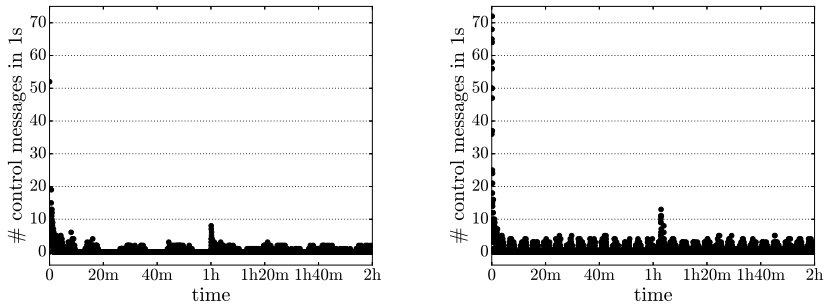
Although the reaction to the failure was not immediate in TinyRPL, both implementations managed to rebuild the DODAG when MRHOF was the objective function. Figures 15–17 present the results of the same experiments but conducted with OF0 as the objective function.

It can be observed in Fig. 15(a) that with OF0 rebuilding all nodes' paths to the root took TinyRPL about 6 minutes, twice as long as with MRHOF. In contrast, it can be concluded from Fig. 15(b) that ContikiRPL with OF0 as the objective



(a) ContikiRPL **(b)** TinyRPL

Fig. 13 Trickle timer resets (unit disk, $r = 1$; MRHOF)

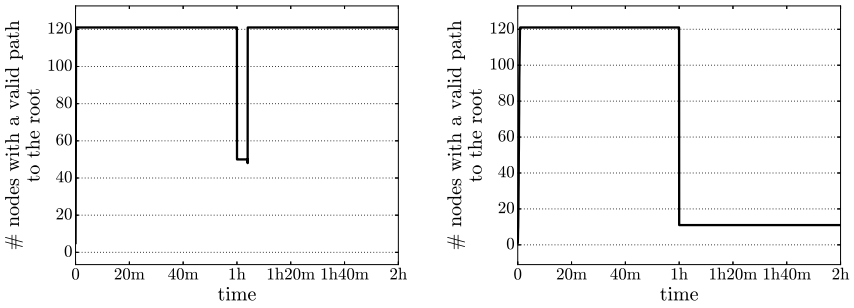


(a) ContikiRPL **(b)** TinyRPL

Fig. 14 Control messages (unit disk, $r = 1$; MRHOF)

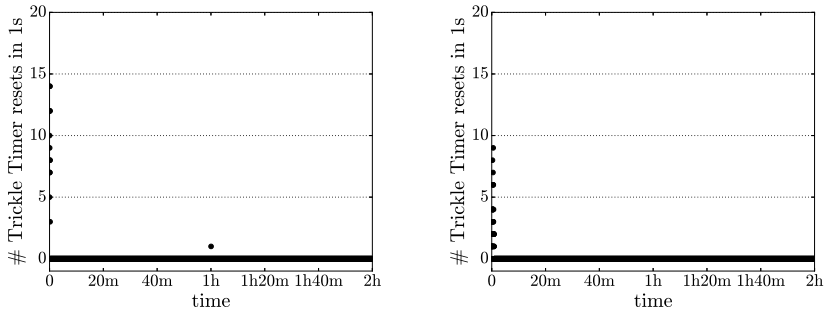
function did not manage to rebuild the DODAG at all. As a result, over 100 nodes, the nodes whose paths to the root contained the broken link, were not able to forward their packets to the destination throughout the rest of the experiment.

While the link failure in TinyRPL was followed by a Trickle timer reset, as can be observed in Fig. 16(a), it turns out that ContikiRPL with OF0 as the objective function did not even react to the failure, not to mention recovering from it. In other words, as can be seen in Fig. 16(b), no node in the network reset its Trickle timer in response to the failure. Moreover, it can be derived from Fig. 17(b) that since an average rank of a node in the network did not change as a result of the failure, then also the rank of the node with id 1 did not change. This, in turn, means that the node with id 1 did not notice for an hour that its link to the root was not working. The reason for this is that ContikiRPL does not estimate link metrics nor does it employ any neighbor unreachability detection mechanism when configured with OF0 as the objective function.



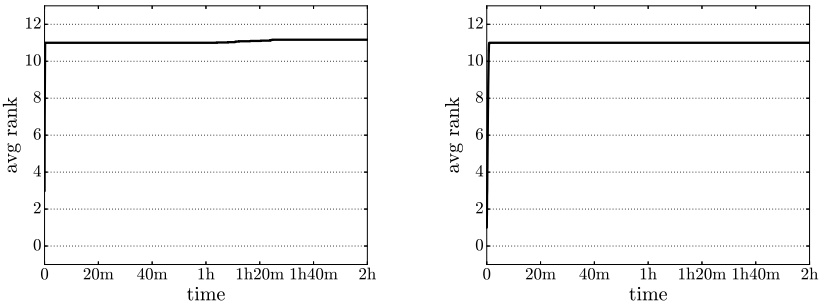
(a) TinyRPL **(b)** ContikiRPL

Fig. 15 Nodes with a valid path to the root (unit disk, $r = 1$; OF0)



(a) TinyRPL **(b)** ContikiRPL

Fig. 16 Trickle timer resets (unit disk, $r = 1$; OF0)



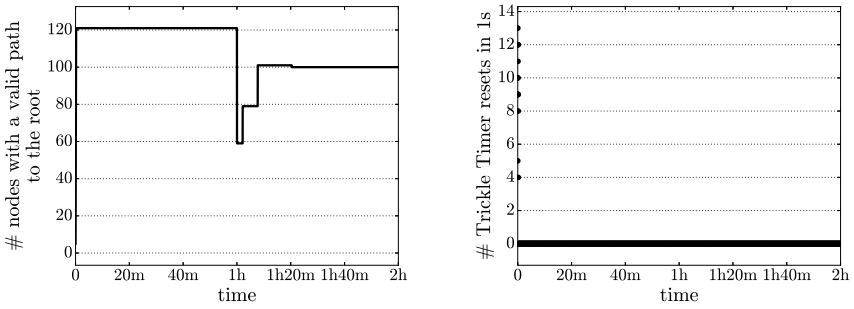
(a) TinyRPL **(b)** ContikiRPL

Fig. 17 Average rank (unit disk, $r = 1$; OF0)

Since ContikiRPL with OF0 as the objective function was not able to detect and react to the smallest possible failure, the failure of a single link, it would not be able to handle more complex failures either. As a result, we do not consider ContikiRPL in the configuration with OF0 in the remainder of the evaluation.

Although the analyzed experiments showed that, apart from ContikiRPL with OF0, all implementation-objective-function configurations were able to handle the failure and recover from it, this is not true for all possible values of configuration parameters. More specifically, decreasing the maximum number of retransmissions for a packet in TinyRPL made TinyRPL unable to rebuild the DODAG, irrespective of which objective function was used.

The results of an experiment with such a configuration are presented in Fig. 18. Similarly to ContikiRPL with OF0, not only did TinyRPL fail to rebuild the broken paths, which is visible in Fig. 18(a), but also no node reacted to the failure by resetting its Trickle timer, as can be observed in Fig. 18(b). We tracked this phenomenon to an emergent behavior of TinyRPL's code for neighbor unreachability detection but the explanation is out of the scope of this chapter.



(a) Nodes with a valid path to the root

(b) Trickle timer resets

Fig. 18 TinyRPL (unit disk, $r = 1$; OF0; max. retransmissions = 3)

6.2 Unimportant Link Failure

In order to simulate a failure of a link that is not responsible for forwarding a large part of all packets, the link between nodes with ids 0 and 1 in the unit-disk topology with radius 4 was removed, as depicted in Fig. 19.

Figure 20 presents the results of this experiment for both implementations with MRHOF as the objective function. As can be observed in Fig. 20, ContikiRPL reacted to the failure more quickly than TinyRPL, which matches the results of the experiments analyzed in the previous section. Nevertheless, the reaction times for

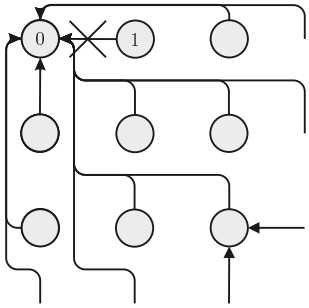
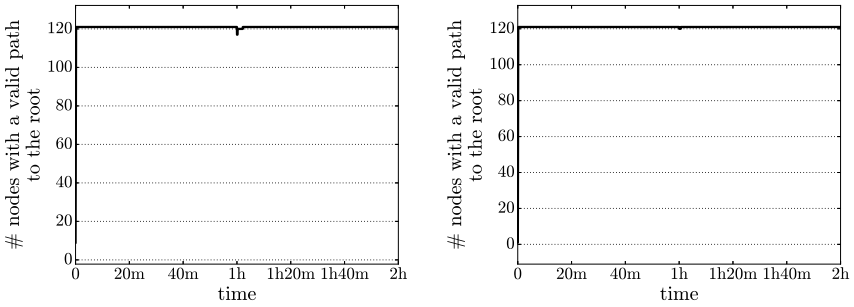


Fig. 19 An unimportant link failure.

both implementations were slightly higher than when the same link was subject to the failure in the previous experiment. This is because in this experiment the link was more loaded—at the network level—than in the previous experiment: since fewer packets failed to be forwarded over this link in a given time period, RPL was not able to detect the failure as fast as in the previous experiment.



(a) TinyRPL **(b)** ContikiRPL

Fig. 20 Nodes with a valid path to the root (unit disk, $r = 4$; MRHOF)

The results for TinyRPL with OF0 did not differ significantly from the presented results. For this reason, their analysis is omitted here.

6.3 Summary

To sum up, the experiments with the simplest type of failure, affecting a single link, show that while the default configurations of the two implementations are by and large able to handle the failure, there exist out-of-the-box configurations for which

this does not hold. What is more, in principle, it is not obvious why such configurations fail. This suggests that the two popular implementations of RPL may not yet be off-the-shelf solutions; on the contrary, their adoption in real-world embedded systems may require expertise.

7 Experiments with Node Failures

As the next step, we analyze the behavior of the implementations under different types of node failures. Since any node failure can be simulated by correlated link failures, we consider only those configurations from the previous section in which an implementation was able to properly handle a link failure. Therefore, we do not examine ContikiRPL with OF0 in this section.

Except for one 3-hour long experiment, the experiments analyzed in this section lasted for 2 simulated hours. Similarly to the link failures in the experiments from the previous section, all node failures in the experiments from this section occurred after the first simulated hour.

7.1 Important Node Failure

Like links, nodes may be more or less "important," depending on the network topology and the shape of a DODAG. Drawing from the previous section, we focus only on the failures of "important" nodes, that is, ones that forward packets from a large fraction of other nodes. In order to test the implementations under the failure of such a node, we turned off one of the two root's neighbors, the node with id 1, in the unit-disk topology with radius 1, as shown in Fig. 21.

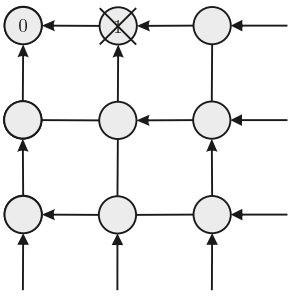
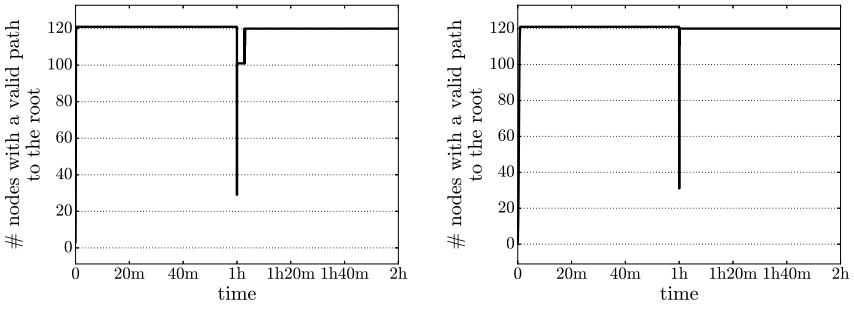


Fig. 21 Failure of an important node

The results turned out to be very similar to the results of the experiments with failures of important links, analyzed in the previous section. To illustrate, Fig. 22

presents the number of nodes with a valid path to the DODAG root for TinyRPL and ContikiRPL with MRHOF as the objective function.



(a) TinyRPL **(b)** ContikiRPL

Fig. 22 Nodes with a valid path to the root (unit disk, $r = 1$; MRHOF)

As can be observed in Fig. 22(a), it took TinyRPL about 3 minutes to rebuild paths from all working nodes to the DODAG root after the failure. ContikiRPL, in turn, more quickly reacted to the failure and reconstructed the DODAG, as visible in Fig. 22(b). Note that in both cases the number of nodes with a valid path to the root after the failure did not reach the level it had before the failure; it was lower exactly by 1. However, since we assume that non-working nodes do not have a valid path to the root, this is an expected result. All in all, these results match precisely those obtained for the failure of the link between the nodes with ids 0 and 1 (cf. Fig. 12).

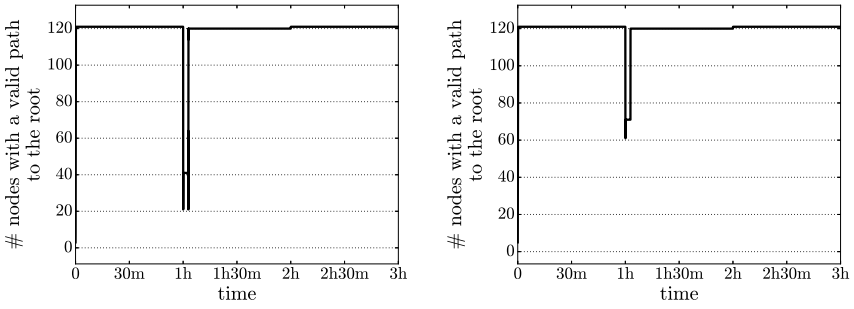
7.2 Node Failure and Recovery

Since failed nodes (and links) may recover, our next scenario aimed to evaluate the implementations' behavior in response to such a recovery. To this end, a 3-hour long experiment was conducted. Similarly to the previously described scenario, the node with id 1 was turned off after the first hour of the experiment (cf. Fig. 21). However, after the second hour the node was turned back on to see the implementations' reactions to the recovery. From the results of the previous experiment, we concluded that an hour is enough for both implementations to handle the failure.

TinyRPL

First, we discuss the results for TinyRPL. Figure 23 presents the number of nodes with a valid path to the root throughout the experiment for MRHOF, Fig. 23(a), and OF0, Fig. 23(b). For both objective functions, a 3–5-minute long decrease in the

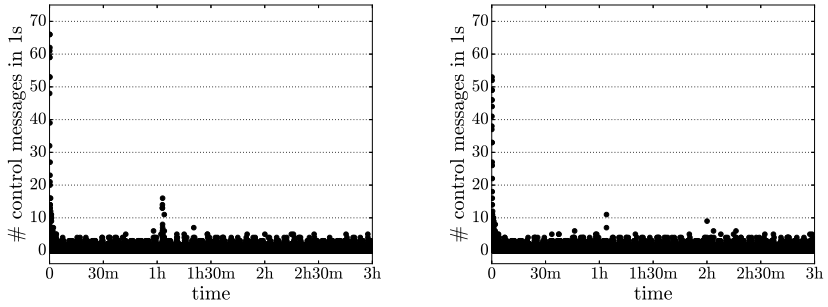
analyzed metric occurred after the first hour of the experiment, immediately after the failure, which is in line with the previous results. After the second hour, in turn, a small increase could be noticed. The increase was caused by the node with id 1 reentering the network and brought the metric back to the value before the failure.



(a) MRHOF **(b)** OF0

Fig. 23 Nodes with a valid path to the root (TinyRPL; unit disk, $r = 1$)

As can be observed in Fig. 24, neither the failure nor the recovery of the broken node resulted in a large increase in control traffic. Nevertheless, additional control messages transmitted as a result of the topology changes are visible in the plots for both objective functions.

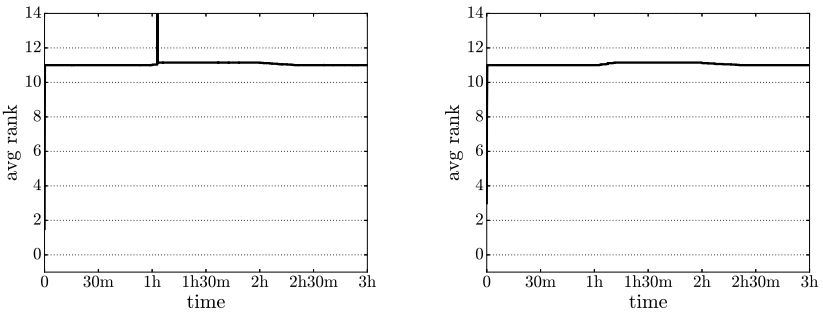


(a) MRHOF **(b)** OF0

Fig. 24 Control messages (TinyRPL; unit disk, $r = 1$)

An important result from this experiment is also presented in Fig. 25, showing the average rank of a node in the DODAG throughout the experiment. For both objective functions, an increase in the average rank can be observed after the first hour of the experiment, which can be attributed to detecting the failure. More specifically,

nodes that selected node 1 as their preferred parent before the failure had to find an alternative preferred parent after the failure. Since the alternative parent had a higher rank than node 1, the nodes' ranks increased as a result of the change, and so did the average rank. In contrast, after the recovery of the failed node, TinyRPL brought the average rank back to the value before the failure, irrespective of which objective function was used. This means that the DODAG restored by the implementation after the recovery was equally good as the one before the failure. Nevertheless, because of not generating much additional control traffic (cf. Fig. 24), this DODAG restoration process took some 20 minutes.

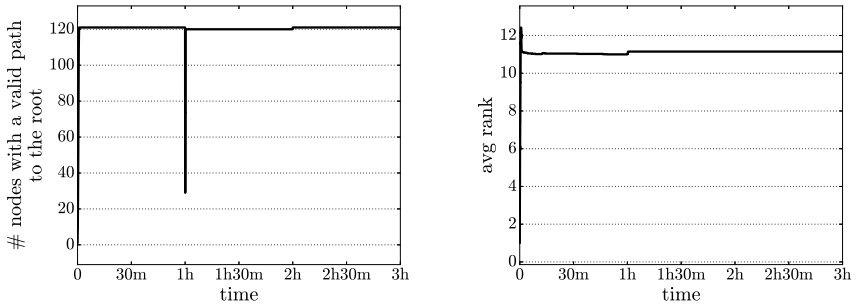


(a) MRHOF **(b)** OF0

Fig. 25 Average rank (TinyRPL; unit disk, $r = 1$)

ContikiRPL

Figure 26 presents the results of the same experiment but for ContikiRPL with MRHOF. The plot of the number of nodes with a valid path to the DODAG root for ContikiRPL in Fig. 26(a) resembles the one for TinyRPL in Fig. 23(a). However, a small difference can be observed in the plot for the average rank of a node in the DODAG for ContikiRPL in Fig. 26(b) and that for TinyRPL in Fig. 25(a). Namely, in ContikiRPL, contrary to TinyRPL, the average rank did not change after node 1 reentered the network. In particular, the routes built before the failure were not restored after the recovery of the failed node. As a result, some packets were routed through longer routes in the last hour of the experiment than in the first hour, thus generating slightly more network traffic than it was necessary. In other words, the DODAG restored by ContikiRPL was not as good as the one before the failure.



(a) Nodes with a valid path to the root **(b)** Average rank

Fig. 26 ContikiRPL (unit disk, $r = 1$; MRHOF)

7.3 Correlated Node Failures

To complete the picture of node failures, we analyze a more complex scenario, that is, correlated failures of multiple nodes. Figure 27 presents the part of the unit-disk topology with radius 1 affected by the failure. The failure of three nodes, marked in the figure with crosses, occurred after the first hour of the experiment. Recall that the node with id 0 is the root of the DODAG. As a result, one of the broken nodes was the root's direct neighbor. This, together with the proximity of the broken nodes, makes the failure potentially difficult to handle.

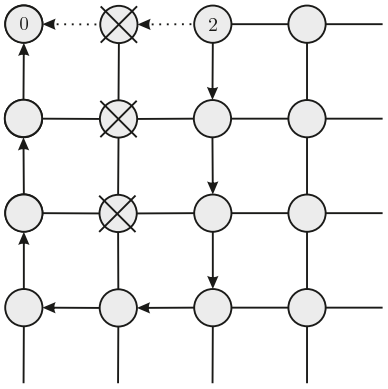


Fig. 27 Correlated node failures

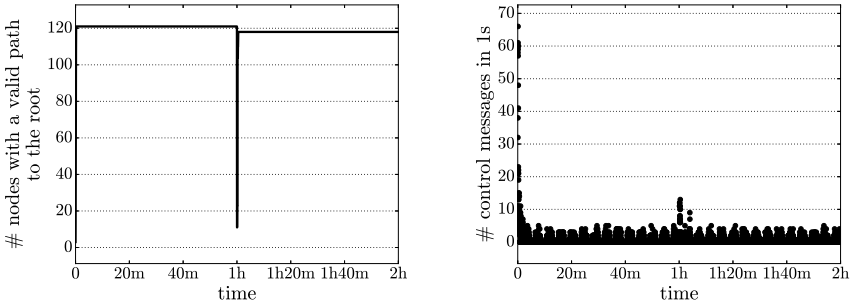
Before we analyze the results of the experiment, let us focus on the node with identifier 2. Since we could observe in Sect. 5 that all analyzed implementations constructed an optimal DODAG, we can assume that the dashed links in Fig. 27

mark the path from the node with identifier 2 to the root before the failure. The bold solid links with arrows, in turn, determine the shortest possible path between these nodes after the failure. As a result of the failure, the length of the path from node 2 to the DODAG root increased so that the hop count became 6. Since the links in the analyzed topology are perfect and retransmissions rare, the ETX difference between the paths should not exceed 6 either. We thus consider the implementations with two configurations of *MaxRankIncrease*: one in which the nodes' rank growths caused by the failure should not be higher than *MaxRankIncrease* and another in which the growths would be higher for some nodes.

New Rank Does Not Exceed MaxRankIncrease

With the default value of *MaxRankIncrease*, equal to 7, the rank growth caused by the failure for any node should not be higher. Consequently, for both OF0 and MRHOF, the implementations should reconstruct the DODAG after the failure.

We start the analysis from TinyRPL. Figures 28 and 29 present TinyRPL's reaction to the failure when MRHOF and OF0, respectively, were used as the objective functions.

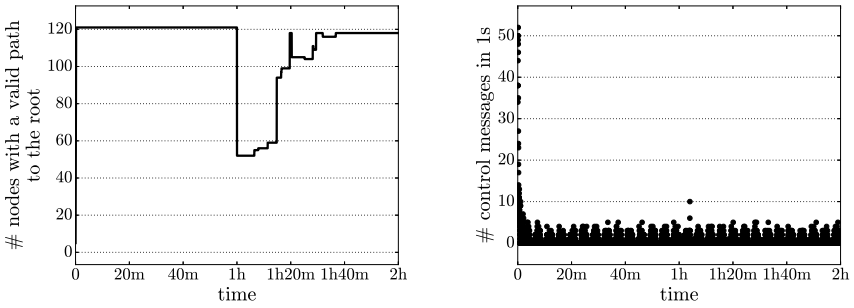


(a) Nodes with a valid path to the root **(b)** Control messages

Fig. 28 TinyRPL (unit disk, $r = 1$; MRHOF)

In the case of MRHOF, as can be observed in Fig. 28(a), the number of nodes with a valid path to the root sharply decreased immediately after the failure, but then it quickly increased to the value of 118, which is the number of all nodes except for the three broken ones. In other words, TinyRPL with MRHOF detected the failure and reacted to it within seconds. The quick reaction, however, resulted in a significant increase in the control traffic, as visible in Fig. 28(b) and not observed in the previous experiments in this configuration.

In contrast, as observed in Fig. 29(a), the recovery process with OF0 as the objective function took TinyRPL over half an hour, far longer than with MRHOF. This resulted in a considerably lower total end-to-end packet delivery rate. On the other

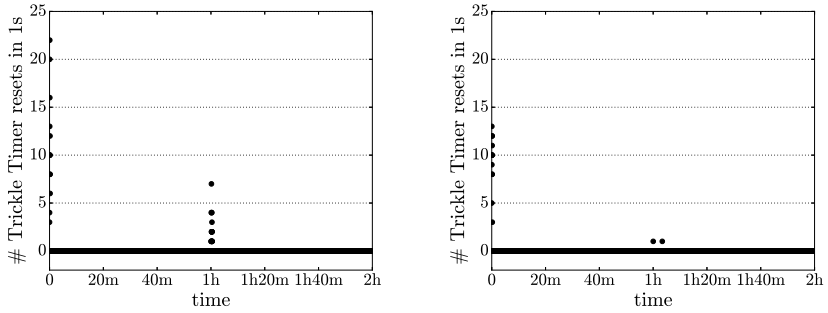


(a) Nodes with a valid path to the root **(b)** Control messages

Fig. 29 TinyRPL (unit disk, $r = 1$; OF0)

hand, the increase in control traffic presented in Fig. 29(b) is barely visible, and hence much lower when compared to that for MRHOF in Fig. 28(b).

The reason for the observed differences in the two metrics in the experiments with MRHOF and OF0 lies in the number of times the nodes reset their Trickle timers in response to the failure. As can be observed in Fig. 30(a), there were many Trickle timer resets when MRHOF was employed as the objective function. Consequently, the control messages traffic increased, the information on the failure was quickly disseminated in the network, and nodes could immediately handle the inconsistency. In contrast, when OF0 was used as the objective function, there were only two Trickle timer resets, as visible in Fig. 30(b). They thus did not cause a significant increase in the control traffic. This, in turn, resulted in the slow propagation of DODAG information in the network and, consequently, long recovery time.

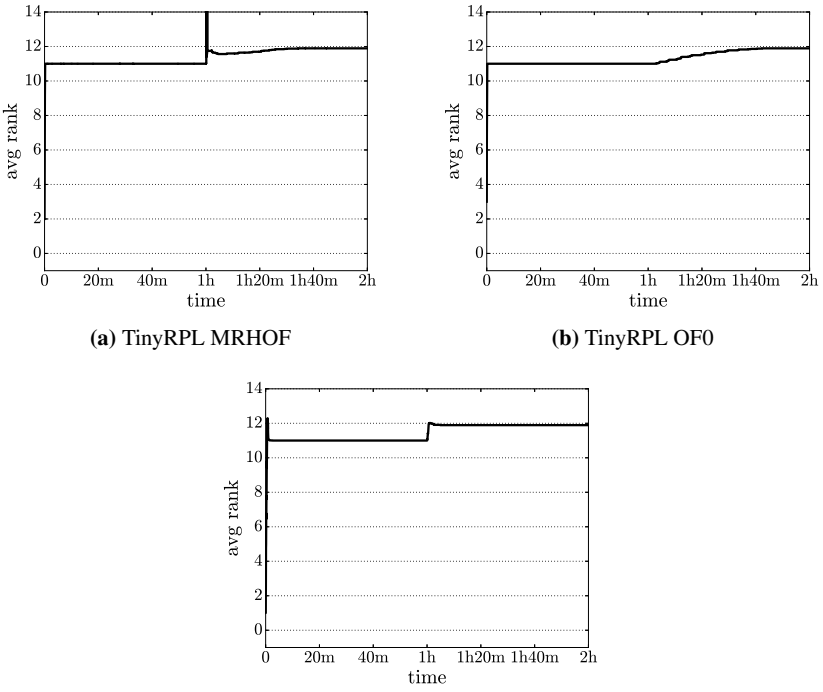


(a) MRHOF **(b)** OF0

Fig. 30 TinyRPL: Trickle timer resets (unit disk, $r = 1$)

The results for ContikiRPL with MRHOF are similar to those for TinyRPL with MRHOF. Therefore, we do not present them here for brevity.

The experiments hitherto showed that the implementations were capable of rebuilding the DODAG, so that all working nodes could successfully forward packets to the root. However, let us examine how close the reconstructed DODAG was to the optimal one. To this end, Fig. 31 compares the average rank of a node in the DODAG after the failure.



(a) TinyRPL MRHOF

(b) TinyRPL OF0

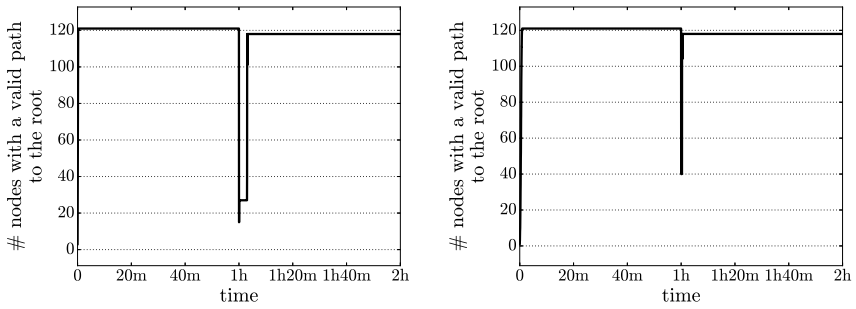
(c) ContikiRPL MRHOF

Fig. 31 Average rank (unit disk, $r = 1$)

It can be observed in Fig. 31(a) that although for TinyRPL with MRHOF there was a significant increase in the average rank immediately after the failure, the rank stabilized below 12 after several minutes. When OF0 was used, in turn, the average rank was slowly increasing for over half an hour after the failure, as visible in Fig. 31(b). Nevertheless, as soon as the DODAG was fully reconstructed, the average rank also stabilized before reaching 12. In contrast, in Fig. 31(c) for ContikiRPL with MRHOF, a sharp increase in the average rank was followed by a smooth decrease. The average rank, however, quickly dropped back below 12. It can thus be concluded that the DODAGs, once fully reconstructed, were equally close to optimal ones for both implementations.

New Rank Exceeds MaxRankIncrease

We reasoned at the beginning of this section that the increase in rank of the node with identifier 2 resulting from the failure is at least 6. Consequently, if we set *MaxRankIncrease* to 5 instead of 7, the node with identifier 2 should not be able to find a new valid path to the root after the failure because its rank would then exceed its minimum rank before the failure by more than *MaxRankIncrease*, which is forbidden according to RPL's specification. Figure 32 thus presents the number of nodes with a valid path to the root throughout the same experiment but with *MaxRankIncrease* set to 5.



(a) TinyRPL MRHOF **(b)** ContikiRPL MRHOF

Fig. 32 Nodes with a valid path to the root (unit disk, $r = 1$; *MaxRankIncrease*=5)

As can be observed in the figure, in contrast to the specification, both TinyRPL and ContikiRPL managed to rebuild the paths for all 118 working nodes, including the node with identifier 2. This result is incorrect and is a consequence of the differences between RPL's specification and both analyzed implementations in tracking the smallest rank in the current DODAG version. In the next section, we give a more detailed explanation for these differences.

7.4 Summary

The experiments with node failures confirmed the conclusions gathered in the previous sections. While the two implementations of RPL correctly handle some simple and more complex scenarios, there exist situations where they do not behave as expected; on the contrary, their behavior is incorrect. Moreover, although the implementations seem to be able to recover from node failures, in some configurations the recovery takes a long time during which the network is not fully reliable.

8 Experiments with Network Partitions

As a final step, we evaluate the implementations in scenarios where some nodes get disconnected from the DODAG root as a result of failures. Recall that according to RPL's specification, a disconnected node should first detect that it does not have any path to the root. It should then discard its preferred parent, set its rank to infinity, and stop forwarding packets.

The experiments presented in this section lasted for 2 or 3 simulated hours, depending on a particular scenario; the failure occurred always after the first hour.

8.1 Root Failure

One example of a failure resulting in a network partition is a failure of the DODAG root. As a result of such a failure, *all* nodes stop having a valid path to the root, which is visualized in Fig. 33. It is thus an extreme case of a network partition. We analyze it separately for each of the two implementations of RPL. For TinyRPL, we also consider the two objective functions, whereas for ContikiRPL—only MRHOF.

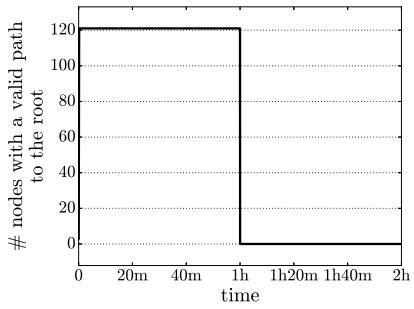
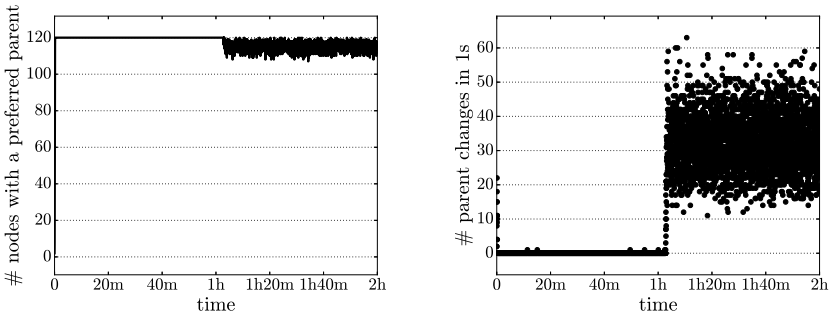


Fig. 33 Nodes with a valid path to the root under a failure of the root

TinyRPL with MRHOF

Figures 34–36 present the results of the experiment for TinyRPL with MRHOF as the objective function. It can be observed in Fig. 34(a) that, although no node had a valid path to the root after the failure, the majority of nodes continuously had nonnull preferred parents. Moreover, the nodes that discarded their preferred parents immediately selected new ones, thereby creating cycles in the DODAG. For this reason, a large number of parent changes can be observed in Fig. 34(b) after the failure. More specifically, the average number of parent changes in each second after

the failure was five times the number of parent changes in the initial second during the DODAG construction. After the failure, the DODAG was thus highly unstable.



(a) Nodes with a preferred parent **(b)** Parent changes

Fig. 34 TinyRPL (unit disk, $r = 1$; MRHOF)

The average rank of the nodes in the DODAG is, in turn, presented in Fig. 35. After the failure, it fluctuated a lot. The sharp changes can be owed to the nodes changing their ranks to infinity, which is represented as value 65 535, and back again to finite values. Nevertheless, a constant growth in the average rank can be observed. In particular, the average rank growth exceeded the maximum allowed growth for a single node, as specified by *MaxRankIncrease*, without any visible reaction from the implementation.

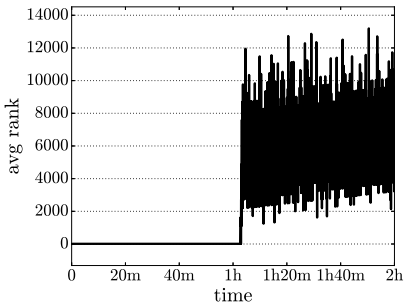
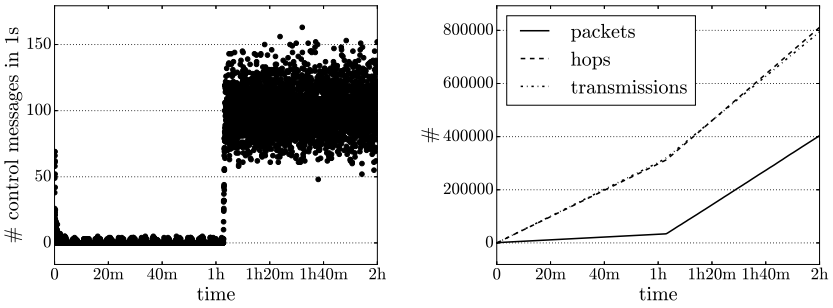


Fig. 35 TinyRPL: Average rank (unit disk, $r = 1$; MRHOF))

The large number of parent changes after the failure was also accompanied by a huge increase in control traffic, which is observable in Fig. 36(a). Namely, over 100 control messages per second were generated in the network after the failure, more than 10 times the number of data packets. This incurred a significant overhead

on the accumulated traffic, and hence, global resource consumption. As shown in Fig. 36(b), due to the increase in the control traffic, the total number of generated packets increased a few times. This, in turn, combined with forwarding data packets over cyclic routes, led to an increase of approximately 30% in the total number of transmissions. In conclusion, rather than reducing, TinyRPL actually increases the global network traffic and resource usage after a crash of the DODAG root.



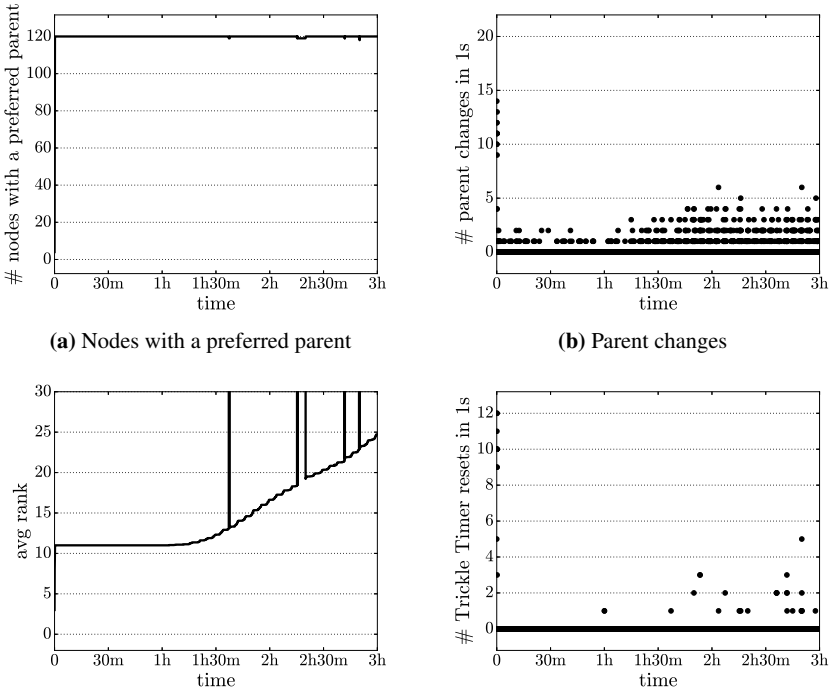
(a) Control messages **(b)** Network traffic

Fig. 36 TinyRPL (unit disk, $r = 1$; MRHOF)

TinyRPL with OF0

Figures 37 and 38 present the results of a 3-hour long experiment for TinyRPL with OF0. As can be observed in Fig. 37(b), the number of parent changes with OF0 was significantly lower than that with MRHOF, depicted in Fig. 34(b). On the other hand, for the majority of time after the failure, all nodes had a nonnull preferred parent, which is visible in Fig. 37(a). In particular, no node discarded its preferred parent until almost an hour after the failure, and a node that finally did it, immediately selected a new preferred parent. In other words, the DODAG included cycles that were not broken for a long time. This is, however, the consequence of the fact that TinyRPL does not implement the mechanism for loop detection when OF0 is employed as the objective function. Moreover, it can be concluded that such mechanism is indeed necessary for RPL to behave efficiently in the presence of network changes.

Similarly to the results for MRHOF, a stable growth in the average rank, exceeding *MaxRankIncrease*, can be observed for OF0 in Fig. 37(c). The growth rate is much lower in the case of OF0, though. This can be attributed to the slower reaction to network changes for TinyRPL with OF0 due to long Trickle timer intervals at the nodes. The intervals were long, in turn, as because of the lack of the loop detection mechanism, only a few nodes reset their Trickle timers in response to the failure, which can be verified in Fig. 37(d).



(a) Nodes with a preferred parent

(b) Parent changes

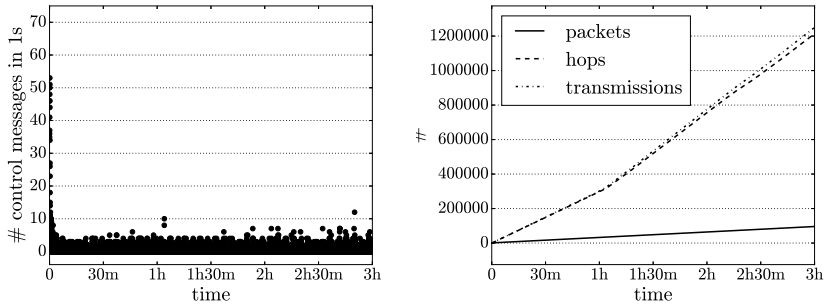
(c) Average rank

(d) Trickle timer resets

Fig. 37 TinyRPL (unit disk, $r = 1$; OF0)

Contrary to the experiments with MRHOF, an increase in the control traffic for OF0 is barely visible in Fig. 38(a). This is because no loops were detected and therefore, the nodes did not reset their Trickle timers in response. Nevertheless, an about 25% growth in the number of hops and transmissions can be observed in Fig. 38(b). The reason is that the nodes forwarded data packets through routes that contained the undetected cycles.

In conclusion, TinyRPL fails to correctly handle a crash of the DODAG root, irrespective of which objective function it is configured with. The main reason for this lies in TinyRPL's implementation of the enforcement mechanisms for the rank growth limit, described by *MaxRankIncrease*. As a reminder, according to RPL's specification, each node in the DODAG has to keep track of the smallest rank it has ever assigned in the current DODAG version. Whenever its rank starts to exceed the smallest rank by more than *MaxRankIncrease*, the node must conclude that the DODAG is broken, discard its preferred parent, and adopt an infinite rank. However, TinyRPL does not follow the specification in that the node keeps "forgetting" the smallest rank in the current DODAG version. In effect, the nodes' ranks may grow to infinity, as observed in our experiments.



(a) Control messages **(b)** Network traffic

Fig. 38 TinyRPL (unit disk, $r = 1$; OF0)

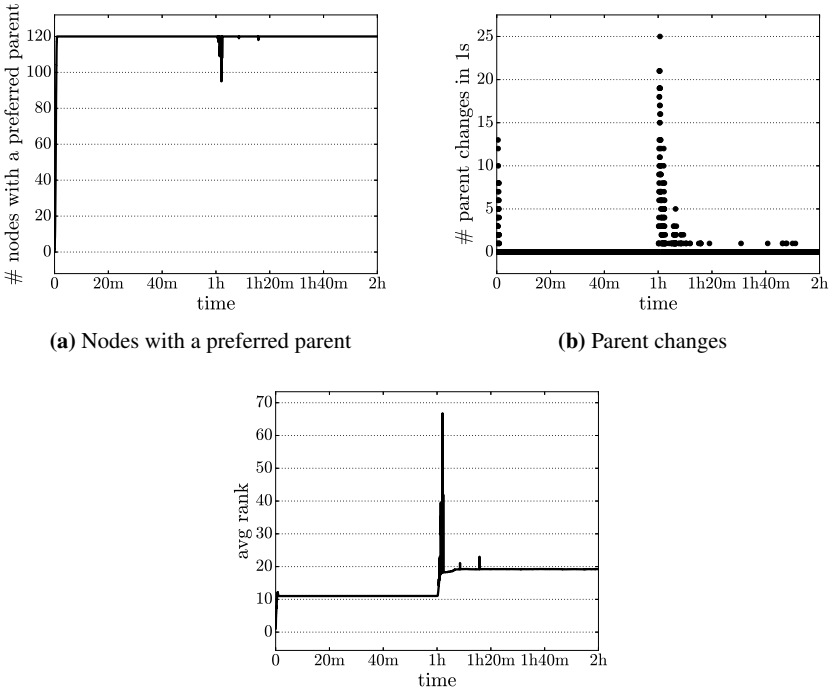
ContikiRPL with MRHOF

The results of the same experiment for ContikiRPL are presented in Fig. 39 and 40. As can be observed in Fig. 39(a), the failure resulted in some nodes discarding their preferred parents and selecting new ones. Consequently, a growth in the number of parent changes is visible in Fig. 39(b) and in rank in Fig. 39(c). Nevertheless, after a short period of changes, the DODAG stabilized again. More specifically, all nodes selected their preferred parents, the number of parent changes in one second dropped back to the level from before the failure and the average rank growth ceased. However, as visible in Fig. 39(c), the average rank grew by more than 7 compared to the one before the failure, and hence, *MaxRankIncrease* was exceeded.

In Fig. 40(a), in turn, it can be observed that although the failure resulted in an increase in control traffic, the increase did not last for more than a few minutes and hardly affected the accumulated network traffic, as can be verified in Fig. 40(b). Consequently, contrary to TinyRPL, handling the crash of the DODAG root by ContikiRPL did not cause a significant increase in global resource usage.

Nevertheless, while ContikiRPL's behavior is better than TinyRPL's, it is not fully correct either. In particular, despite the root being down, the nodes were still using their resources to forward generated data packets over the entire network to the root's neighbors, which, in turn, could not forward them further and were forced to drop them. Consequently, although the nodes did not generate extra traffic, the traffic they did generate was still far from optimal. Optimally, after the nodes detected the failure, they should have stopped forwarding any data packets as there was no route through which those packets could have been delivered to the root.

The reason for the observed behavior in ContikiRPL is the same as in the case of TinyRPL: improper management of the smallest rank assigned to a node in a given DODAG version. In the case of ContikiRPL, however, the incorrect changes to the value happen not virtually always but only occasionally. Nevertheless, in a network of a few tens of nodes, they are still a problem.

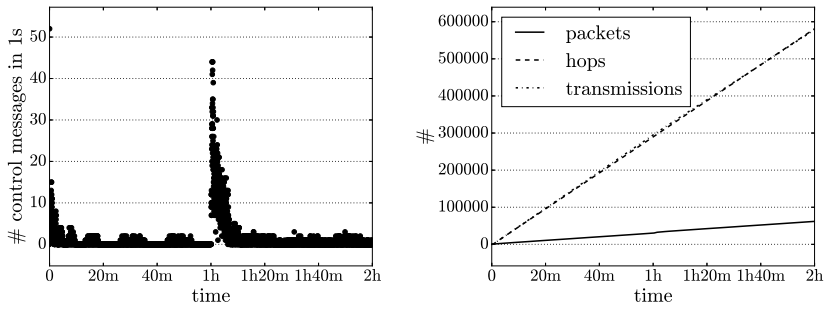


(a) Nodes with a preferred parent

(b) Parent changes

(c) Average rank

Fig. 39 ContikiRPL (unit disk, $r = 1$; MRHOF)



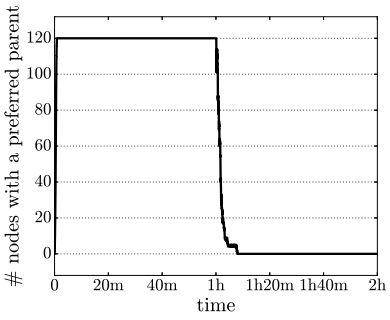
(a) Control messages

(b) Network traffic

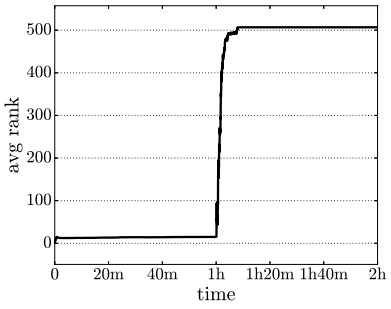
Fig. 40 ContikiRPL (unit disk, $r = 1$; MRHOF)

When it comes to the latest version of ContikiRPL, as of January 5th, 2017, it behaves correctly in this failure scenario. More specifically, in response to the failure, all working nodes discarded their preferred parents, as visible in Fig. 41(a), set their

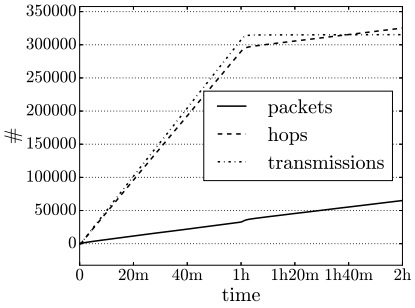
ranks to high values, as plotted in Fig. 41(b), and stopped forwarding data packets, as can be observed in Fig. 41(c). Consequently, there were only a few transmissions in reaction to the failure, which can be observed in Fig. 41(c), and they were all transmissions of control messages. Note that the number of hops after the failure increased faster than the number of transmissions. This was because a request from the test application to send a data packet counted as one hop even if the packet was not transmitted over the network because of, for example, a lack of the default route in the node's IPv6 routing table. Consequently, since the test application continued to generate data packets after the failure and RPL ceased to transmit them at some point, the growth rate of the number of hops exceeded that of the number of transmissions.



(a) Nodes with a preferred parent



(b) Average rank



(c) Network traffic

Fig. 41 ContikiRPL (latest version) (unit disk, $r = 1$; MRHOF)

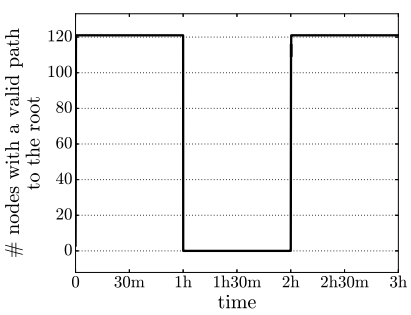
Nevertheless, recall that the DODAG constructed by the latest version of ContikiRPL is not stable. What is more, in contrast to the analyzed version, the new version fails in the next scenario, which we believe is even more important from a practical perspective.

8.2 Root Failure and Recovery

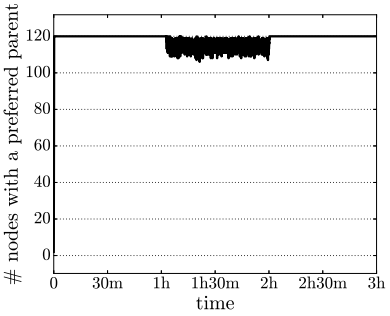
Hitherto, we have shown that neither TinyRPL nor ContikiRPL correctly handles a crash of the DODAG root. Our subsequent experiments aim to examine whether the implementations manage to rebuild a DODAG and return to a stable state when the root recovers from its failure. The experiments lasted for 3 hours. The root failure occurred after the first hour of each experiment, whereas the recovery after the second hour.

TinyRPL

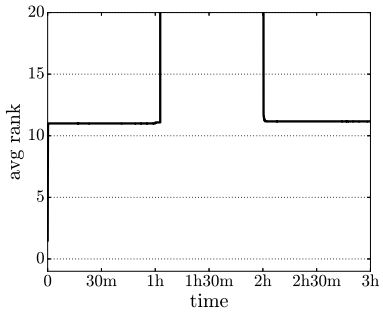
In the case of TinyRPL with MRHOF, as can be observed in Fig. 42(a) and 42(b), the DODAG was reconstructed immediately after the root had recovered. Moreover, since the average rank quickly returned to the level from before the failure, as shown in Fig. 42(c), it can be concluded that the reconstructed DODAG was as good as the initial one.



(a) Nodes with a valid path to the root



(b) Nodes with a preferred parent



(c) Average rank

Fig. 42 TinyRPL (unit disk, $r = 1$; MRHOF)

Figure 43 presents the network traffic throughout the experiment. Although an increase in the number of generated packets and performed transmissions can be observed between the failure and the recovery, the metrics returned to the values from before the failure as soon as the root recovered.

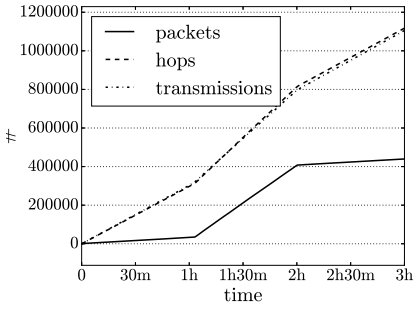


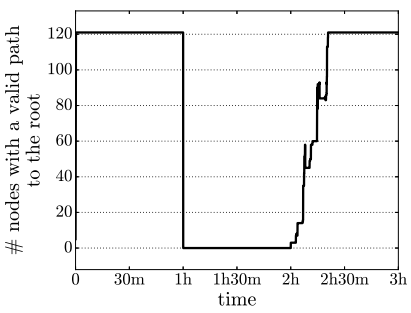
Fig. 43 TinyRPL: Network traffic (unit disk, r=1; MRHOF)

Figure 44 presents the results of the same experiment, but with OF0 as the objective function. Although TinyRPL with OF0 managed to rebuild the DODAG and bring the average rank of a node back to the value from before the failure, the reconstruction process took the implementation significantly longer than when MRHOF was used, about 20 minutes instead of a few seconds. Again, this can be attributed to the low number of nodes reseting their Trickle timers in response to network changes when OF0 is employed as the objective function, which can be verified in Fig. 44(c).

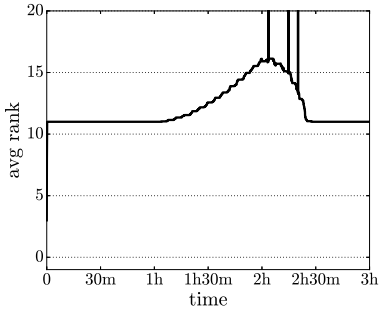
ContikiRPL

In ContikiRPL, in turn, all nodes rebuilt their paths to the root within seconds after the recovery, as shown in Fig. 45(a). It can be observed in Fig. 45(b), however, that it took the implementation more than half an hour to bring the average node's rank back to the value before the failure. The reason for this is that TinyRPL and ContikiRPL manage their parent sets in a different way, the details of which we omit here for brevity.

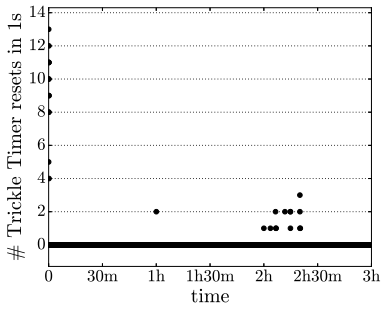
Finally, recall that while the analyzed version of ContikiRPL does not react correctly to the root failure, its latest version does. Let us thus check whether the latest version also correctly handles the DODAG root's recovery. As can be observed in Fig. 46, the latest version of ContikiRPL did not reconstruct the DODAG after the root had recovered from the failure. More specifically, neither did any nonroot node select a preferred parent, which can be verified in Fig. 46(a), nor did it adopt a low rank, which is visible in Fig. 46(b) in the third hour of the experiment. It can be thus concluded that the implementation is not capable of recovering from a failure of



(a) Nodes with a valid path to the root

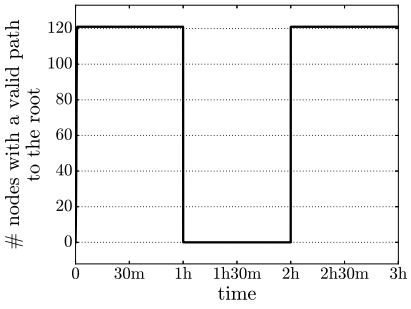


(b) Average rank

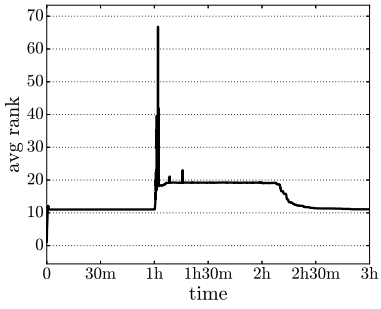


(c) Trickle timer resets

Fig. 44 TinyRPL (unit disk, $r = 1$; OF0)



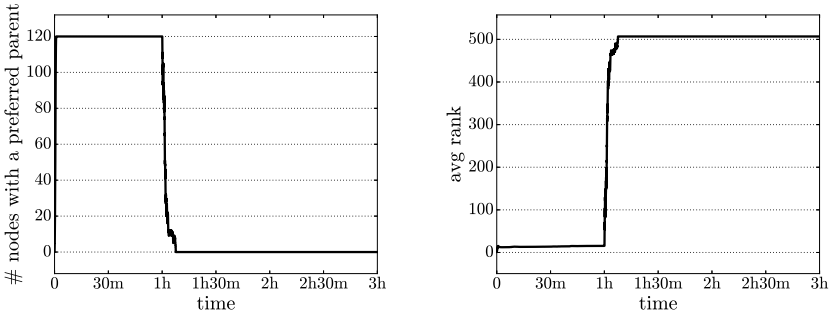
(a) Nodes with a valid path to the root



(b) Average rank

Fig. 45 ContikiRPL (unit disk, $r = 1$; MRHOF)

the DODAG root. Again, the reason lies in yet another method of maintaining node parent sets in the new version of ContikiRPL compared to the previous versions.



(a) Nodes with a preferred parent **(b)** Average rank

Fig. 46 ContikiRPL (latest version) (unit disk, $r = 1$; MRHOF)

8.3 Summary

The DODAG root failure scenario analyzed in this section was the most extreme case of a network partition. The results for other scenarios with network partitions are analogous to the presented ones. For this reason, we omit them here.

All in all, the results for scenarios involving network partitions are unfavorable for both analyzed implementations of RPL. Neither TinyRPL nor ContikiRPL correctly reacts to such failures: nodes keep forwarding data packets to the root even though all their paths to the root are broken, thereby unnecessarily wasting resources. In some cases, the failure additionally results in a huge increase in network traffic and, consequently, resource consumption. This, in turn, may be a major obstacle to deploying RPL in real-world embedded systems. Moreover, it turns out that the failure-handling behavior of the implementations may vary drastically between versions, which complicates deployments even more.

9 Conclusions

To sum up, the following general conclusion can be drawn from our dependability-oriented experiments with the two popular implementations of RPL. When a network is not subject to any failures, both implementations behave as expected: their control traffic volume gradually decreases to the minimal values and remains so, which corresponds to RPL's stable state. However, as soon as failures are introduced into the links and/or nodes, the implementations' behavior starts to diverge from the desirable one, sometimes with grave consequences for the network.

More specifically, it turns out that although the implementations are capable of handling simple link or node failures in the majority of the evaluated parameter settings, there exist configurations in which such failures are not even detected, not

to mention proper handling. The results for more complex failures, notably those leading to network partitions, are even more concerning. In both TinyRPL and ContikiRPL, nodes keep forwarding packets to the root node even if all their routing paths are broken. An effect of this inability to conclude that a major failure has taken place is that the nodes unnecessarily waste precious resources on transmitting packets that are never delivered to the root. In TinyRPL, the control traffic actually explodes after the failure, which could drain the energy of typical battery-powered nodes in a few days rather than months or years; in ContikiRPL, the increase is less pronounced. In any case, however, such futile transmissions combined with the lack of automatic nodes' reaction to the failure may give the network administrators an impression that everything functions properly, which may delay detecting the failure even by the human administrators. This, in turn, may be particularly problematic if the root node is an actuator that controls some important equipment. All in all, the two popular implementations of RPL are simply not robust against failures.

This is in stark contrast to the requirements of many LLN-based embedded systems. Because of the characteristics of LLNs, failures of both links and nodes are not uncommon. Consequently, it is crucial for RPL's implementations to handle such failures in a correct and efficient manner. The evaluated implementations thus have to be fixed before they can be utilized in real-world systems in which dependability is important. However, the fixing process may not be straightforward. The example of the different versions of ContikiRPL shows that addressing problems in one usage scenario may have unpredictable consequences in others. In other words, it may not be easy to determine whether a given change to the implementation is actually a "fix." We may thus need better methods of verifying the compliance of an implementation with RPL's specification. What is more, it is not clear whether fixing the implementations is possible without changing RPL's specification itself. It may well be that changes to the specification or novel algorithms [18] are necessary to improve failure handling. Importantly, we may also need formal methods for proving the correctness of the core protocol and such algorithms.

Acknowledgements This work was supported by the National Center for Research and Development (NCBR) in Poland under grant no. LIDER/434/L-6/14/NCBR/2015. K. Iwanicki was additionally supported by the Polish Ministry of Science and Higher Education with a scholarship for outstanding young scientists.

References

1. Boubekeur, F., Blin, L., Leone, R., Medagliani, P.: Bounding degrees on RPL. In: Q2SWinet '15: Proceedings of the 11th ACM Symposium on QoS and Security for Wireless and Mobile Networks, pp. 123–130. ACM (2015). DOI 10.1145/2815317.2815339
2. Brachman, A.: RPL objective function impact on LLNs topology and performance. In: Internet of Things, Smart Spaces, and Next Generation Networking: 13th International Conference, NEW2AN 2013 and 6th Conference, ruSMART 2013, St. Petersburg, Russia, August 28-30, 2013. Proceedings, pp. 340–351. Springer Berlin Heidelberg (2013). DOI 10.1007/978-3-642-40316-3_30

3. Clausen, T., Herberg, U., Philipp, M.: A critical evaluation of the IPv6 routing protocol for low power and lossy networks (RPL). In: 2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 365–372. IEEE (2011). DOI 10.1109/WiMOB.2011.6085374
4. Conta, A., Gupta, M.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol version 6 (IPv6) Specification. RFC 4443 (2006). DOI 10.17487/RFC4443
5. De Couto, D.S.J., Aguayo, D., Bicket, J., Morris, R.: A high-throughput path metric for multi-hop wireless routing. In: MobiCom '03: Proceedings of the 9th Annual International Conference on Mobile Computing and Networking, pp. 134–146. ACM (2003). DOI 10.1145/938985.939000
6. Dunkels, A., Gronvall, B., Voigt, T.: Contiki – A lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE International Conference on Local Computer Networks, pp. 455–462. IEEE (2004). DOI 10.1109/LCN.2004.38
7. Duquennoy, S., Landsiedel, O., Voigt, T.: Let the tree bloom: Scalable opportunistic routing with ORPL. In: SenSys '13: Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems, pp. 2:1–2:14. ACM (2013). DOI 10.1145/2517351.2517369
8. Fonseca, R., Ratnasamy, S., Zhao, J., Ee, C.T., Culler, D., Shenker, S., Stoica, I.: Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, NSDI '05, pp. 329–342. USENIX Association (2005)
9. Frey, H., Pind, K.: Dynamic source routing versus greedy routing in a testbed sensor network deployment. In: Proceedings of the 6th European Conference on Wireless Sensor Networks, EWSN '09, pp. 86–101. Springer-Verlag (2009). DOI 10.1007/978-3-642-00224-3_6
10. Gaddour, O., Koubâa, A.: RPL in a nutshell: A survey. Computer Networks **56**(14), 3163–3178 (2012). DOI 10.1016/j.comnet.2012.06.016
11. Gaddour, O., Koubâa, A., Chaudhry, S., Tezeghdanti, M., Chaari, R., Abid, M.: Simulation and performance evaluation of DAG construction with RPL. In: Third International Conference on Communications and Networking, pp. 1–8. IEEE (2012). DOI 10.1109/ComNet.2012.6217747
12. Gnawali, O., Levis, P.: The minimum rank with hysteresis objective function. RFC 6719 (2012). DOI 10.17487/RFC6719
13. Han, D., Gnawali, O.: Performance of RPL under wireless interference. IEEE Communications Magazine **51**(12), 137–143 (2013). DOI 10.1109/MCOM.2013.6685769
14. Heurtefeux, K., Menouar, H., AbuAli, N.: Experimental evaluation of a routing protocol for WSNs: RPL robustness under study. In: 2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 491–498 (2013). DOI 10.1109/WiMOB.2013.6673404
15. Hui, J., Vasseur, J.P.: The routing protocol for low-power and lossy networks (RPL) option for carrying RPL information in data-plane datagrams. RFC 6553 (2012). DOI 10.17487/RFC6553
16. Iova, O., Theoleyre, F., Noel, T.: Stability and efficiency of RPL under realistic conditions in wireless sensor networks. In: 2013 IEEE 24th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), pp. 2098–2102. IEEE (2013). DOI 10.1109/PIMRC.2013.6666490
17. Istomin, T., Kiraly, C., Picco, G.P.: Is RPL ready for actuation? A comparative evaluation in a smart city scenario. In: Wireless Sensor Networks: 12th European Conference, EWSN 2015, Porto, Portugal, February 9-11, 2015. Proceedings, pp. 291–299. Springer International Publishing (2015). DOI 10.1007/978-3-319-15582-1_22
18. Iwanicki, K.: RNFD: Routing-layer detection of DODAG (root) node failures in low-power wireless networks. In: 2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), pp. 13:1–13:12. IEEE (2016). DOI 10.1109/IPSN.2016.7460720
19. Iwanicki, K., van Steen, M.: On hierarchical routing in wireless sensor networks. In: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks, IPSN '09, pp. 133–144. IEEE Computer Society (2009)

20. Iwanicki, K., van Steen, M.: A case for hierarchical routing in low-power wireless embedded networks. ACM Trans. Sen. Netw. **8**(3), 25:1–25:34 (2012). DOI 10.1145/2240092.2240099
21. Khelifi, N., Kammoun, W., Youssef, H.: Efficiency of the RPL repair mechanisms for low power and lossy networks. In: 2014 International Wireless Communications and Mobile Computing Conference (IWCMC), pp. 98–103. IEEE (2014). DOI 10.1109/IWCMC.2014.6906339
22. Kim, Y.J., Govindan, R., Karp, B., Shenker, S.: Geographic routing made practical. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, NSDI '05, pp. 217–230. USENIX Association (2005)
23. Ko, J., Eriksson, J., Tsiftes, N., Dawson-Haggerty, S., Terzis, A., Dunkels, A., Culler, D.: ContikiRPL and TinyRPL: Happy together. In: Proceedings of the Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011) (2011)
24. Korte, K.D., Sehgal, A., Schönwälder, J.: A study of the RPL repair process using ContikiRPL. In: Dependable Networks and Services: 6th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2012, Luxembourg, Luxembourg, June 4-8, 2012. Proceedings, pp. 50–61. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-30633-4_8
25. Levis, P., Clausen, T., Hui, J., Gnawali, O., Jo, K.: The Trickle algorithm. RFC 6206 (2011). DOI 10.17487/RFC6206
26. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In: SenSys '03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, pp. 126–137. ACM (2003). DOI 10.1145/958491.958506
27. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: TinyOS: An Operating System for Sensor Networks, pp. 115–148. Springer Berlin Heidelberg (2005). DOI 10.1007/3-540-27139-2_7
28. Mao, Y., Wang, F., Qiu, L., Lam, S.S., Smith, J.M.: S4: Small state and small stretch routing protocol for large wireless sensor networks. In: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation, NSDI '07, pp. 8–8. USENIX Association, Berkeley, CA, USA (2007)
29. Mohammad, M., Guo, X., Chan, M.C.: Oppcast: Exploiting spatial and channel diversity for robust data collection in urban environments. In: 2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), pp. 19:1–19:12. IEEE (2016). DOI 10.1109/IPSN.2016.7460681
30. Narten, T., Nordmark, E., Simpson, W., Soliman, H.: Neighbor discovery for IP version 6 (IPv6). RFC 4861 (2007). DOI 10.17487/RFC4861
31. Osterlind, F., Dunkels, A., Eriksson, J., Finne, N., Voigt, T.: Cross-level sensor network simulation with COOJA. In: Proceedings. 2006 31st IEEE Conference on Local Computer Networks, pp. 641–648. IEEE (2006). DOI 10.1109/LCN.2006.322172
32. Radoi, I.E., Shenoy, A., Arvind, D.: Evaluation of routing protocols for Internet-enabled wireless sensor networks. In: ICWMC 2012: The Eighth International Conference on Wireless and Mobile Communications (2012)
33. Thubert, P.: Objective function zero for the routing protocol for low-power and lossy networks (RPL). RFC 6552 (2012). DOI 10.17487/RFC6552
34. Tripathi, J., de Oliveira, J.C., Vasseur, J.P.: A performance evaluation study of RPL: Routing protocol for low power and lossy networks. In: 2010 44th Annual Conference on Information Sciences and Systems (CISS), pp. 1–6. IEEE (2010). DOI 10.1109/CISS.2010.5464820
35. Vasseur, J.P., Kim, M., Pister, K., Dejean, N., Barthel, D.: Routing metrics used for path calculation in low-power and lossy networks. RFC 6551 (2012). DOI 10.17487/RFC6551
36. Winter, T., Thubert, P., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, J.P., Alexander, R.: RPL: IPv6 routing protocol for low-power and lossy networks. RFC 6550 (2012). DOI 10.17487/RFC6550