The IPv6 Routing Protocol for Low-power and Lossy Networks (RPL) under Network Partitions

Agnieszka Paszkowska
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
ap321142@students.mimuw.edu.pl

Konrad Iwanicki
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
iwanicki@mimuw.edu.pl

Abstract

We study the behavior of the IPv6 Routing Protocol for Low-power and Lossy Networks (RPL) under network partitions, failures that are notoriously difficult to handle. Our work combines experiments in simulators and on a ∼100-node testbed with formal reasoning methods. First, we show empirically that RPL's two popular implementations, TinyRPL and ContikiRPL, do not behave as expected under partitions. To establish whether this behavior is due to the implementations' defects or features of RPL's design itself, we model the protocol's dynamic operation, based directly on its specification, and use the model to formally prove how a compliant implementation must behave under a network partition. We then apply these theoretical results in practice by patching the two implementations, so that they satisfy all formal properties we defined for our model. Finally, we demonstrate experimentally that the behavior of the patched implementations is correct under partitions and in general.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*routing protocols, protocol verification*

General Terms

Experimentation, Reliability, Standardization, Theory

Keywords

RPL, routing protocol, network partition, partition tolerance, dependability, low-power wireless network

1 Introduction

The IPv6 Routing Protocol for Low-power and Lossy Networks (*RPL*) [33] is the current *de jure* standard for routing IPv6 packets in low-power wireless networks. Its popular implementations, notably ContikiRPL and TinyRPL [24], have been employed in numerous real-world embedded systems, both research-oriented and commercial ones.

This success is largely due to the way RPL's design fulfills the specific requirements of low-power wireless networks. In particular, one of the principal problems it addresses is how to efficiently handle continuous changes in link qualities and node population, which are inherent in these networks. In fact, the solutions for dealing with such network topology dynamics are RPL's major contribution.

However, as we elaborate in the next section, while there is sufficient evidence that these solutions handle well basic link and node dynamics, to the best of our knowledge, virtually no past work analyzes their behavior under topology changes that are notoriously difficult to deal with: *network partitions*. A network partition occurs when a group of nodes (or just one node) becomes isolated from the rest, such that no communication is possible between this group and other live nodes. It usually results from a correlated failure of multiple nodes and/or links but sometimes a deterioration of just one link or a crash of just a single node splits a network.

Proper partition handling by RPL is important for real-world embedded systems because partitions are not something uncommon in low-power wireless networks and their effects may be grave. For instance, in the widely cited, pre-RPL Sonoma deployment, a network split prevented roughly half of the nodes from reporting their data to a sink [31]. Likewise, in the deployments in which we participated, partitions were nothing extraordinary. Their predominant cause was power outages of so-called border router nodes, connecting low-power wireless networks to the Internet, as it was not economic to provision redundant power supply for such routers. Nevertheless, we experienced more involved partitions as well, notably in deployments for precision agriculture, where plants growing radically during vegetation tended to shadow wireless communication, a phenomenon confirmed by other researchers [16]. In general, since many factors affect wireless connectivity and the lifetime of low-power devices, various forms of splits and disconnections were observed also by other groups in past wireless sensing deployments [3]. Yet, we are not aware of any reports on RPL's behavior in such situations, notably whether it detects partitions or exhibits routing meltdowns. The lack of knowledge of this behavior can thus be a major risk when employing RPL in real-world systems requiring high reliability.

Consequently, in this paper, we conduct an in-depth study of RPL's partition handling behavior. After an overview of RPL and related work (Sect. 2), we experimentally evalu-

ate RPL's two popular implementations, ContikiRPL and TinyRPL, under network splits (Sect. 3). The experiments show that the implementations fail to correctly handle partitions. Our goal then is to formally establish whether this result is due to the implementations' flaws or inherent features of RPL's design. To this end, we build a model of RPL's dynamic behavior, based solely on the protocol's specification, and utilize it not only to formally prove that RPL itself should properly handle network partitions, but also to identify conditions that an implementation must satisfy for this to be true (Sect. 4). This allows us to identify violations of these conditions in the two implementations, fix these violations, and demonstrate experimentally, both in simulations and on a ∼100-node testbed, that the fixes correct the implementations' partition handling behavior (Sect. 5). Finally, we conclude by highlighting major lessons learned (Sect. 6).

2 Background

Let us start by formulating the network partition problem in RPL's terminology [33] and surveying related work.

2.1 Overview of RPL

RPL incorporates two routing techniques: distance-vector routing and link-state routing. Distance-vector routing is responsible for forwarding packets in a so-called *upward* direction: from low-power wireless nodes via a border router to the Internet. Link-state routing is utilized for forwarding packets in the opposite, so-called *downward* direction. A combination of these two techniques allows for packet forwarding between any two network nodes.

For upward routing to a given destination, usually a border router, RPL utilizes a *DODAG* (i.e., destination-oriented directed acyclic graph), which represents the available routes from the nodes to the destination. To this end, each node is assigned a *rank* reflecting its distance (e.g., hop count) from the destination. The node's *neighbors* (i.e., other nodes in the node's radio range) that have their ranks lower than the node's own rank are the node's *parents*: forwarding a packet to an arbitrary parent shortens its distance to the destination. Normally, however, a node forwards all packets to a single *preferred parent*. The destination, the DODAG *root*, has in turn no parent and a minimal rank. Globally, the nodes' links to preferred parents thus form a directed tree that is a subgraph of the DODAG and has the DODAG root as the sink.

The maintenance of the DODAG under network topology dynamics is RPL's important contribution. In essence, each node exchanges its rank with its neighbors by regularly broadcasting so-called *DIO messages*. The broadcasting is driven by a so-called *Trickle timer* [26]: normally it is rare but, when the DODAG changes, it may be reset to be more frequent and gradually return to its stable mode, unless reset again. It can also take place on demand, in response to so-called *DIS messages* from the node's neighbors. Beyond this, the management of the node's neighbor set, notably detecting which of the neighbors are still reachable, is outside RPL's specification and is referred to as *routing adjacency maintenance*. Nevertheless, given a neighbor set with the neighbors' ranks, reachability status, and other information, the node selects its preferred parent and rank. RPL delegates these tasks to a so-called *objective function*, like OF0 [30] or MRHOF [13], which, in particular, need not select the parent offering the node the lowest rank. All in all, RPL's DODAG maintenance in not trivial; yet it can be highly efficient.

In contrast, downward routing is far simpler. Each node periodically routes a so-called *DAO message* upward to the DODAG root. The message contains information on the node's parents. Being normally a border router with enough memory, the root collects such information from all nodes, so that it has a global view of the DODAG. Consequently, to route a packet downwards, it computes an entire route, embeds this route in the packet, and forwards the augmented packet to the first node on the route, which continues to the next node, and so on. In particular, in this way, with so-called *DAO-ACK messages*, the root acknowledges to the nodes the reception of their DAO messages. Finally, as an optimization, the nodes on the upward route of a DAO message from a node (i.e., the node's ancestors in the DODAG) may store information on the downward routes to the node, so that the routes need not be embedded into downward-routed packets. This, however, requires extra memory at these nodes.

2.2 Problem Statement

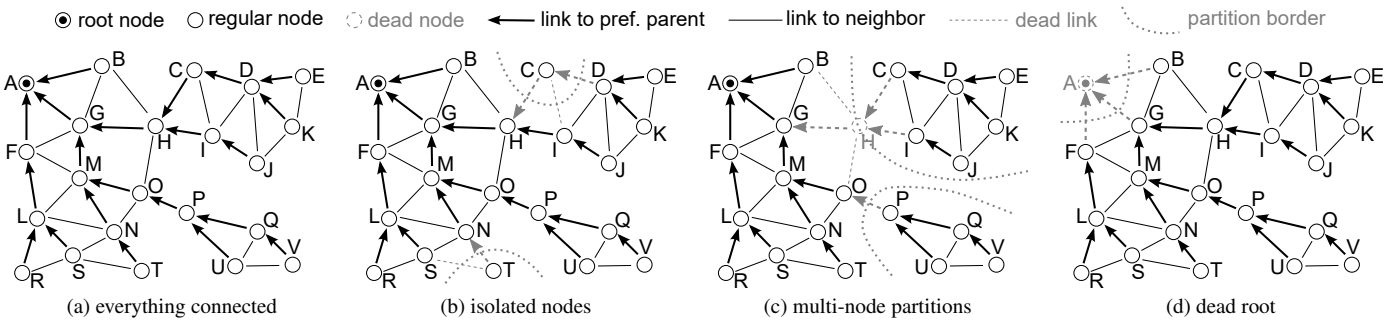
Traditionally, a network is considered partitioned if and only if (abbr. *iff*) it contains two live nodes between which no routing path exists. In RPL, however, routing is done over a DODAG and the DODAG root plays a key role: typically, every upward route ends and every downward route starts at the root but even with the aforementioned optimization, the root coordinates DODAG construction and maintenance. Therefore, here we use a stronger definition of partitioning: *a network/DODAG is partitioned/split/disconnected* iff it contains two live nodes between which no path exists *via the DODAG root* or, put differently, *iff there is no path between the DODAG root and some live node*. We will also refer to such a node as partitioned/split/disconnected.

There are many ways in which a connected DODAG, as in Fig. 1(a), can get partitioned. For example, in Fig. 1(b), nodes T and C become isolated from the root node, A, as a result of a correlated link deterioration. Note also that since node C is a DODAG ancestor of nodes D, E, and K, no routing is possible between these nodes and the root. However, this is temporary because a path exists between D, E, K, and the root (i.e., the nodes are not partitioned), and hence in principle they can be reconnected to the DODAG. In Fig. 1(c), in turn, a failure of a single link partitions nodes P, Q, U, and V from the root and, similarly, a crash of one node disconnects nodes C, D, E, I, J, and K. Finally, Fig. 1(d) presents a case that is a partition according to our definition but not the traditional one: a failure of the DODAG root.

2.3 Related Work

Proper partition handling is critical in Internet-scale systems [5]. As we argued previously, it is also important in low-power wireless networks. However, in prior studies concerning RPL's reliability, the protocol's behavior under network partitions has received little research attention.

Early work on RPL aimed to qualitatively identify flaws in the protocol's initial specification [7, 35] as well as to quantitatively assess the general performance of its simulated models [7, 32] and prototype implementations [11].



(a) everything connected (b) isolated nodes (c) multi-node partitions (d) dead root

Figure 1. Examples of DODAG partitions.

Consequently, partition handling was not a major concern then. Even though routing loops observed by Clausen et al. were attributed to DODAG partitions, the problem was not further investigated [7]. In other cases, scenarios with network partitions were avoided, sometimes explicitly [35].

Subsequent reliability-oriented studies of RPL—in simulations, on testbeds, and in the real world—focused in turn on various performance aspects under failures [12, 17, 19, 21, 22, 25], novel extensions and solutions for the protocol's open issues [4, 21], the impact of heavy radio interference [10, 15], interoperability of popular implementations [24], and certain application scenarios [20, 28]. To the best of our knowledge, however, apart from recent work on RNFD [21], network partitions were only mentioned briefly (e.g., isolated nodes [17])—if at all—without further studies. RNFD, in turn, is an extension to RPL that improves the handling of DODAG root node crashes, which is just one type of network partition. Moreover, it is a heuristic in that it just helps but *does not guarantee* correctly handling such failures, instead relying on RPL itself to this end. In this view, this paper complements the work on RNFD: it establishes formal guarantees on RPL's behavior under all types of network partitions. The utility of our results thus stretches beyond RNFD.

Finally, handling network partitions was also studied from other perspectives. Kleinberg et al. [23] and Shrivastava et al. [29] developed theoretical bounds and approximation algorithms for selecting "sentinel" nodes that guarantee detecting disconnected network regions. For the same problem, Barooah et al. [1] introduced an algorithm whose operation resembles iteratively computing electrical potentials. Won and Stoleru [34] devised a solution for a more general problem, involving multiple destination nodes. Devi and Manickam [8] showed how to reconnect partitions with mobile nodes. Gregorczyk et al. [14], in turn, presented probabilistic algorithms for estimating partition sizes under mobility. Some of those solutions could potentially extend RPL [8, 14]; others themselves require reliable routing [23, 29]. All in all, RPL does not use those solutions and we are not aware of any previous studies of network partitions from RPL's perspective.

3 RPL's Implementations under Partitions

Since network partition handling in RPL has received little research attention, we start our study with experimental results illustrating the behavior of RPL's two popular implementations, TinyRPL for TinyOS and ContikiRPL for ContikiOS, under partitions. To this end, we employ the most recent version of TinyRPL from the main repository of TinyOS[1] (i.e., from June 7, 2017) and the latest stable version of ContikiRPL available from the ContikiOS website[2] (i.e., Contiki 3.0). We evaluate the implementations in publicly available simulators: TOSSIM and Cooja, respectively.

3.1 Experimental Setup

For the experimental scenarios, let us observe that handling a network partition is trivial for downward routes. This is because, when a node becomes disconnected from the DODAG root, no DAO message from the node reaches the root. In effect, the parent information for the node at the root eventually expires, and hence the root starts considering the node as absent: no more downward routes involving the node are generated. Because of this mechanism's simplicity, downward routing in the two implementations behaves correctly under network partitions. Consequently, in the rest of this paper, we focus on upward routing, the behavior of which is more intricate under partitions, and thus may have led to defects both in RPL's design and its implementations.

Accordingly, we evaluate the implementations with a data collection application utilizing all-to-one upward routing. More specifically, each node generates UDP data packets to be forwarded to the root. The inter-packet delays at the node are chosen at random between T and $2T$ time units. This results in relatively uniform traffic, which helps the reactive routing adjacency maintenance solutions, as employed in the implementations, to detect dead links [21]. The presented experimental runs of the application do not involve radio duty cycling. Nevertheless, we did verify that results for duty cycling do not diverge from the demonstrated ones.

We have tested the implementations in various network topologies. However, for brevity, here we present results only for 11×11 grids with unit-disk communication: the radio range of each node is a circle with its radius equal to 1; a node thus has perfect links to up to 4 neighbors and no links to the other nodes. While unit-disk communication is an idealized model, we have selected it for three reasons. It allows for precisely controlling the network topology, which is desired for illustrative purposes. It is supported by both simulators, TOSSIM and Cooja, which facilitates directly comparing results for TinyRPL and ContikiRPL. It is also favor-

[1] https://github.com/tinyos/tinyos-main

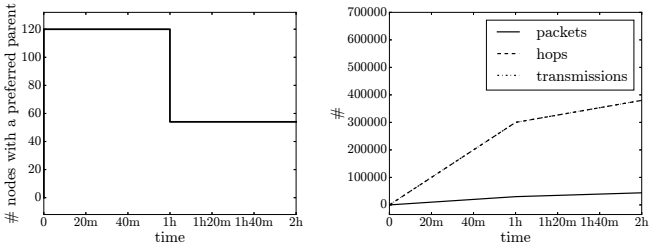
[2] http://www.contiki-os.org/download.html

able for the implementations because—with perfect links—they can reliably detect whether a given link is dead or live, whereas in the real world, such detection entails one- or two-sided errors. Nevertheless, we did verify that our results hold also for non-unit-disk communication models. In particular, in Sect. 5.3, we discuss experiments on a real-world testbed.

The selected experiment is 2 simulated hours long with a network partition occurring after 1 simulated hour, values chosen with a large safety margin based on preliminary experiments. We have evaluated many partition scenarios, obtaining consistent results. The one presented involves correlated link failures along the diagonal of the grid that cause roughly a half (i.e., 66) of the 121 nodes to permanently lose their paths to the DODAG root; the root is in one corner of the grid and these nodes are connected to the node in the opposite corner. Each node continuously generates packets to be forwarded to the root with $T = 10$ s. The maximum number of retransmissions is 5 per hop. As the objective function, both implementations utilize MRHOF [13], which is more advanced than OF0 [30]. The remaining configuration parameters for the implementations have default values.

3.2 Experimental Results

Before discussing the results of the experiment for each of the implementations, let us explain the expected behavior after a network partition. Since partitioned nodes have no paths to the DODAG root, RPL instances running on them should eventually detect this and stop forwarding any packets upward. Instead, the higher layers, in particular the application, should learn about the lack of the default route. This is to allow them, for instance, to buffer any important data that, if sent during the partition, would be lost. In general, doing any upward forwarding to the root by disconnected nodes is bound to fail and thus should be prevented to avoid wasting global network resources. To address all these issues, upon detecting that it is partitioned, each node running RPL should adopt an *infinite rank* and a *null preferred parent*, which reflect its disconnection from the DODAG root. Figure 2 shows an idealized run of RPL with this behavior.



(a) nodes with preferred parents (b) accumulated network traffic

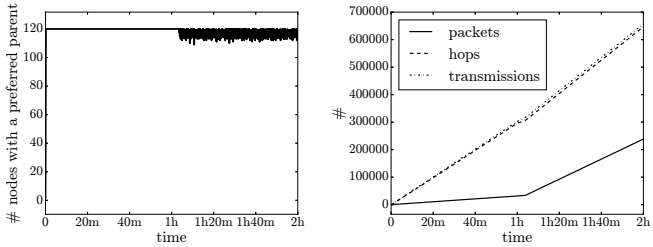
Figure 2. RPL's expected behavior under a partition.

Figure 2(a) depicts the number of nodes with a non-null preferred parent. Ideally, immediately after the first hour of the experiment, that is, after the partition occurs, this number should drop from 120 by 66, reaching the value of 54. In contrast to the idealized run, in practice, the drop need not be immediate because of the time to detect the link deaths.

Figure 2(b) shows in turn the total (i.e., control and data) traffic accumulated since the beginning of the experiment and measured with three metrics: the number of packets generated by the nodes that are passed for radio transmission, the total number of hops these packets are forwarded over, and the total number of transmissions necessary to cover these hops. The number of generated packets should level off after the partition by approximately $66/\big((10+20)/2\big) = 4.4$ per second. This is because the packets generated by each of the 66 partitioned nodes every 10–20 s should not be passed for transmission, as the nodes lack preferred parents to which they could be transmitted. The drop in the number of hops and transmissions should be even greater. The reason for this is that the disconnected nodes are further away from the DODAG root than the other nodes, and hence their packets need more hops (and transmissions) to reach the root.

In practice, a slight increase in all three traffic metrics immediately after the partition could be observed, which would be a result of resetting the aforementioned Trickle timers for DIO messages to quickly react to the network turbulence. However, we have decided not to account for this increase: in our idealized run, Trickle timers are always in their stable modes, generating minimal control traffic. Likewise, we assume that in the run, the number of transmissions equals the number of hops. Yet, in practice, this need not be the case because of collisions and retransmissions (over dead links).

Figure 3 shows the results of the experiment for TinyRPL. They do not match the reference ones from Fig. 2. As can be observed in Fig. 3(a), the number of nodes with non-null preferred parents starts to fluctuate after the partition. This suggests that the disconnected nodes do not permanently discard their preferred parents but, instead, repeatedly select new ones. What is more, as visible in Fig. 3(b), rather than dropping, the network traffic actually permanently increases after the partition. There are two possible reasons for this. First, as the disconnected nodes do not discard their preferred parents, they also continue forwarding packets. Second, frequent preferred parent changes lead to Trickle timer resets, and hence an explosion of DIO messages. All in all, not only is this behavior incorrect but, what is arguably more problematic, it also results in an increased use of network resources.

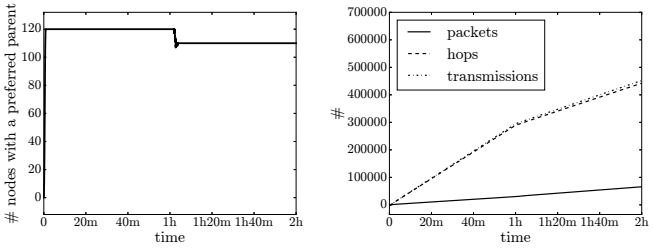


(a) nodes with preferred parents (b) accumulated network traffic

Figure 3. TinyRPL's behavior under a partition.

Figure 4 presents the corresponding results for ContikiRPL. Although they seem better than for TinyRPL, they are not correct either. It can be observed in Fig. 4(a) that only 10 nodes permanently discard their preferred parents after the partition. Consequently, the other 56 nodes without a path to the DODAG root continue to generate packets to the root, and hence there is no drop in the number

of packets in Fig. 4(b). However, what can be observed in Fig. 4(b) is some drop in the number of hops and transmissions. The possible reason for this is that a packet generated by a partitioned node is not forwarded all the way up to the DODAG root but only within the partition. Nevertheless, the network traffic is still not optimal because the partitioned nodes should not forward any packets at all, as in Fig. 2(b).



(a) nodes with preferred parents (b) accumulated network traffic

Figure 4. ContikiRPL's behavior under a partition.

To sum up, none of the two implementations correctly handles the network partition. Neither in TinyRPL nor in ContikiRPL is the application guaranteed to get feedback when the route to the DODAG root is lost. Moreover, while in terms of network traffic, ContikiRPL's behavior seems slightly better than TinyRPL's behavior, it is not as expected either, which may be a symptom of further hidden problems. We have obtained similar results for experiments with different partition scenarios and in different system configurations.

4 Formal Analysis of Partition Handling

Since TinyRPL's and ContikiRPL's partition handling behavior diverges from the expected one, we analyze whether this is due to RPL's design or flaws in these two implementations. To this end, we model RPL—based directly on its specification—to formally prove its dynamic behavior under partitions. For the process, we adopt a methodology founded on linear temporal logic (LTL), a popular formalism for verifying concurrent algorithms [2]. Although deriving proofs in LTL can often be automated with generic [18] or domain-specific [27] model checkers, apart from proving RPL's behavior, an arguably more important goal of this paper is explaining this behavior and the underlying assumptions. For this reason, we derive the proofs manually and, rather than LTL's symbolic notation, we employ a textual one, much like Cachin et al. [6] for classic distributed algorithms.

In the process, we strive to ensure that we neither oversimplify nor overspecify RPL. In particular, we deliberately try to avoid any simplifying assumptions that would make our analysis easier but are false in the real world. This is because making such assumptions, we could fail to capture important phenomena that affect RPL's partition handling. Likewise, for RPL's open issues, we avoid assuming anything beyond the specification, even though the implementations rely on particular solutions. This is to model the protocol not its particular realizations, which may be incorrect. All in all, we are confident that the model and analysis presented in this section precisely capture RPL's behavior under real-world network partitions—a claim that is further reinforced by the practical application of our results in Sect. 5.

4.1 System Model

We model a RPL-based system intuitively: as a directed graph of nodes and links. A *live node* runs RPL; a *dead node* does nothing. A *live link* allows a packet sent by one live node to be received by the other live node; a *dead link* lets no packets through. However, even packets transmitted over a live link may be lost or duplicated, as explained shortly. The subset of nodes to which a node has links—live or dead—corresponds to the node's neighbors. A neighbor *is adjacent* to a live node iff the neighbor and the link from the node to the neighbor and in the opposite direction are all live; otherwise, the neighbor is *non-adjacent*.

To focus our reasoning on partition handling, we abstract out RPL's local state at each node to four variables: `rank`, `prefpar`, `neighborset`, and `minrank`. `Rank` is a non-negative integer denoting the node's rank in the DODAG; it can be infinite. `Prefpar` is the identifier of the node's preferred parent in the DODAG or `null`. `Neighborset` is a set of records describing the node's local view of its neighbors: each record, `n`, represents a neighbor and holds the neighbor's identifier (`n.id`), rank (`n.rank`), as known by the node (which may be different from the neighbor's present rank), and the node's view of whether the neighbor is reachable over the corresponding link or not (`n.reachable`), that is, whether the node believes that the the neighbor is adjacent (again, this belief may be different than the actual situation). Finally, `minrank` is the minimal value of `rank` that the node has ever had.[3] Overall, this abstraction of a node's local state is sufficient for the considered problem.

Likewise, we model packets in transit (i.e., on air or in communication queues anywhere in the network stack) as delivery multi-sets: one multi-set per node, denoting the packets to be received by the node. Packet loss and duplication are intuitive: a packet, respectively, does not appear or appears more than once in the target multi-set(s). Moreover, for our purposes, we need to consider only packets carrying DIO messages. Each such DIO message, `d`, is a record consisting of the transmitting neighbor's identifier (`d.id`) and rank (`d.rank`). Again, this granularity of modeling RPL's communication is adequate for the analyzed problem.

To sum up, globally, the system's state comprises all nodes' and links' live/dead statuses, as well as the values of all nodes' local variables and delivery multi-sets. We can thus formulate properties of such a state, for example, *"node B's* `rank` *is infinite and node A's* `prefpar` *is* `null`*."*

Given this, RPL's operation at the nodes, which changes the state, is modeled as a LTL *computation*: an infinite sequence of global system states representing the flow of time. To express properties that hold only at specific moments in time, we utilize LTL's temporal operators *always*, *never*, and *eventually*. Their semantics is intuitive. Property "always/never *P*" holds in state *i* of a computation iff property "*P*" holds in all/no states $j \geq i$ of the computation. Property "eventually *P*" holds in state *i* iff property "*P*" holds in

[3] Actually, RPL's specification states that this is the minimal rank value in a so-called DODAG version. However, a version change is initiated at the root and corresponds to a complete DODAG reconstruction. Therefore, partitioned nodes are unable to learn about the change. Since our analyses focus on such nodes, we need not consider DODAG version changes here.

some state $j \geq i$. By convention, if the state is omitted for a property, then the first state is implied. For example, property *"node A is eventually always adjacent to node B"* means that, from some moment in time forever, *A* is adjacent to *B*. In contrast, *"node A is always eventually adjacent to node B"* means that, for each moment, *A* is adjacent to *B* at this or some future moment, that is, *A* is repeatedly adjacent to *B* but not necessarily permanently. Note that properties formulated in LTL involve no timing guarantees. However, this is deliberate as RPL is not a real-time protocol, and hence the actual timings must not influence its correctness.

Iteratively, we have devised a number of properties—based directly on RPL's specification—that describe the various facets of RPL's behavior in terms of the model. Here we present the final minimal subset of those properties that is sufficient for analyzing the considered problem. These properties are divided into three groups related to routing adjacency maintenance, objective functions, and control traffic.

4.1.1 Routing Adjacency Maintenance Properties

The properties for routing adjacency maintenance formalize the management of a node's `neighborset`. Property RA1 states that any change to the `neighborset` triggers `prefpar` and `rank` reselection. Yet, it does not specify when exactly the reselection occurs, thereby not constraining implementations beyond RPL's specification, which permits both on-demand and periodic approaches. Property RA2, in turn, states that the node's local view of its neighbors' `rank`s is copied from the DIO messages received by the node, which again conforms to the specification. Finally, property RA3 formalizes neighbor non-adjacency detection. Since in RPL's specification this issue is largely open, our requirements for the detector are extremely weak: they imply no particular approach (e.g., reactive vs. proactive) and allow for mistakes before finally marking a non-adjacent neighbor as such. They are thus satisfied by practical failure detectors.

RA1. *Always, if a node's* `neighborset` *changes (i.e., entries are added or removed, or their fields are modified), then the node will eventually reselect its* `prefpar` *and* `rank`*.*

RA2. *Always, for each entry,* `n`*, in a node's* `neighborset`*,* `n.rank` *is infinite, if the node has received no DIO message,* `d`*, with* `d.id` *equal to* `n.id` *since the entry was added to the* `neighborset`*, or* `n.rank` *is equal to* `d.rank`*, where* `d` *is the last DIO message with* `d.id` *equal to* `n.id` *received by the node since the entry was added to the* `neighborset`*.*

RA3. *If a node's neighbor is eventually always non-adjacent, then an entry,* `n`*, with this neighbor's identifier as* `n.id` *will eventually always either be absent from the node's* `neighborset` *or have its* `n.reachable` *set to* `false`*.*

Apart from these properties, we do not further formalize routing adjacency maintenance not to overspecify the protocol. In particular, we do not specify when an entry is added to a node's `neighborset` or how the node decides which subset of its neighbors is represented by the `neighborset` entries, issues that are both outside RPL's specification.

4.1.2 Objective Function Properties

The same is true for Properties OF1–OF5, which describe the requirements for an objective function: they are formulated based precisely on RPL's specification. The positive constant `MinHopRankIncrease` is the minimal difference RPL aims to enforce between a node's `rank` and its preferred parent's `rank`; it is also the DODAG root's `rank`. The constant `MaxRankIncrease` denotes in turn the maximal acceptable increase of a node's `rank` in the DODAG.[4]

OF1. *A node's* `prefpar` *and* `rank` *change only as a result of reselection or the node's death and (re)start; otherwise, they remain unmodified.*

OF2. *A node's* `minrank` *is always equal to the minimal value of the node's* `rank` *so far.*

OF3. *Always, when reselecting its* `prefpar` *and* `rank`*, the root node adopts* `null` *and* `MinHopRankIncrease`*, respectively. These are also the initial values of the two variables at the root node when the node (re)starts.*

OF4. *Always, when reselecting its* `prefpar` *and* `rank`*, a non-root node adopts* `null` *as* `prefpar` *iff it also adopts infinite* `rank`*. These are also the initial values of the two variables at the non-root node when the node (re)starts.*

OF5. *Always, when reselecting its* `prefpar` *and* `rank`*, a non-root node adopts* `null` *and infinity, respectively, iff its* `neighborset` *does not contain an entry,* `n`*, for which a rank,* `r`*, can be computed (in an objective-function-specific way), such that* `n` *and* `r` *satisfy all the following constraints:*
(a) `n.rank` $<$ *infinity,*
(b) `n.reachable` $=$ `true`*,*
(c) `r` $<$ *infinity,*
(d) `r` $\geq$ `n.rank` $+$ `MinHopRankIncrease`*,*
(e) `r` $\leq$ `minrank` $+$ `MaxRankIncrease`*.*
Otherwise, the node adopts `n.id` *and* `r` *as its* `prefpar` *and* `rank`*, respectively, for some neighbor* `n` *and rank* `r`*, such that* `n` *and* `r` *satisfy all conditions (a)–(e).*

4.1.3 Control Traffic Properties

Finally, RPL's specification contains no requirements on the delivery of DIO messages between neighboring nodes. In contrast, since our analysis necessitates such requirements, we formalize them as properties DIO1–DIO3. Property DIO1 states that DIO messages are not spoofed: any DIO message received by a node comes from the node's actual neighbor. Property DIO2, in turn, describes message loss. It permits a loss of any number of DIO messages at arbitrary moments in time, as long as for adjacent neighbors DIO message delivery never stops permanently. Finally, property DIO3 formalizes message duplication. It allows a DIO message to be duplicated an arbitrary, albeit finite number of times, so that its reception eventually finishes. All in all, the properties for DIO message loss and duplication, are rather pessimistic. This is to avoid oversimplifying the real-world intricacies of low-power wireless communication.

[4] Actually, in a DODAG version, with the same remarks as in footnote [3].

DIO1. *If a node receives a DIO message, `d`, then the message must have been sent earlier by the node's neighbor: `d.id` equals the neighbor's identifier and `d.rank` equals the neighbor's `rank` from the moment of sending `d`.*

DIO2. *If a node's neighbor is always eventually adjacent, then the node will always eventually receive a DIO message from this neighbor.*

DIO3. *If a node's neighbor sends a DIO message, then the node will eventually never receive the message.*

4.2 Main Hypothesis and Its Analysis

Given the model, we can formalize this paper's main hypothesis on RPL's behavior under network partitions:

HYPOTHESIS 1. *If an always-eventually-live node is eventually always partitioned from the DODAG root, then eventually always, whenever the node is live, its `prefpar` and `rank` will be equal to `null` and infinity, respectively.*

The formulation of the hypothesis is general in that it does not preclude failures (and recoveries) of the considered node: the node is not required to be always live but only always eventually. This is again to avoid any simplifying assumptions, so that we can check if RPL handles partitions even under additional failures. However, the possibility of nodes (and links) failing and recovering also requires clarifying what it means that "a node is eventually always partitioned."

To this end, consider an arbitrary RPL computation. Let us take an arbitrary node, P, and the set of its neighbors, N_P. Each neighbor, $Q \in N_P$, is either eventually never adjacent to P (i.e., from some moment in time $t^d_{P,Q}$, Q is forever non-adjacent to P) or always eventually adjacent to P (i.e., for every moment in time, Q is adjacent to P at this or some future moment, which we represent as $t^d_{P,Q} = null$). We can thus divide the set of P's neighbors, N_P, into two fixed subsets: N^a_P, the neighbors that are always eventually adjacent to P, and N^n_P, the remaining neighbors, which are eventually always non-adjacent to P. As the definition of being adjacent is symmetric, for any node Q, we have: $Q \in N^a_P$ iff $P \in N^a_Q$.

Intuitively, P's *partition*, denoted π^{a*}_P, is a subset of the nodes that contains: node P itself, P's always-eventually-adjacent neighbors (i.e., the nodes in N^a_P), their always-eventually-adjacent neighbors, and so on. To formalize this, consider the following function: $Reach(X) = X \cup \bigcup_{Q \in X} N^a_Q$, where X is an arbitrary subset of the nodes. P's partition is equal to that fixed point of function *Reach* that contains P. Since the number of the nodes is finite, say k, we have:

$$\pi^{a*}_P = Reach^k(\{P\}) = \underbrace{Reach(Reach(\ldots Reach}_{k \text{ times}}(\{P\})\ldots)).$$

Similarly to N^n_P, π^{n*}_P is the subset of the remaining nodes, that is, the ones that do not belong to π^{a*}_P. Again, for any node S, we have $S \in \pi^{a*}_P$ iff $P \in \pi^{a*}_S$. However, while $N^a_P \subseteq \pi^{a*}_P$, we need not have $N^a_P = \pi^{a*}_P \cap N_P$ because P's eventually-always-non-adjacent neighbor can still be in P's partition, *Reach*able from P via other nodes. Figure 5 visualizes these sets.

Accordingly, *node P is eventually always partitioned from the DODAG root* iff the root is not in π^{a*}_P. Like previously, this definition is extremely permissive to maximize the generality of our conclusions. In particular, two nodes that are never live at the same time need not be partitioned.

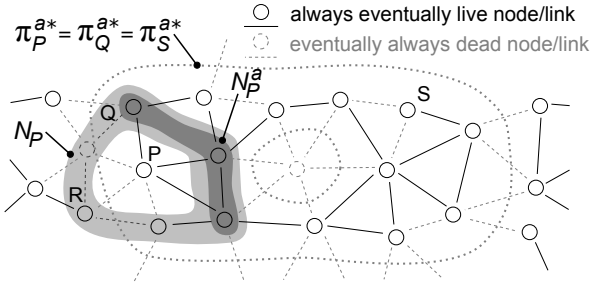


Figure 5. A classification of nodes after a partition.

In our analysis, we consider an arbitrary RPL computation in which some node(s) are eventually always partitioned from the DODAG root. Therefore, for any node P, let $t^d_P = max_{Q \in N_P}(t^d_{P,Q})$ denote the moment in time, possibly *null*, at which some of P's neighbors becomes permanently non-adjacent to P for the last time, that is, starting from t^d_P, no other P's neighbors ever get non-adjacent to P forever—only temporarily—if at all. Moreover, let $t^d = max_{P \in nodes}(t^d_P)$ be the last moment at which any two neighboring nodes in the entire network become permanently non-adjacent ($t^d \neq null$ because some node is eventually always partitioned). This implies that after t^d no new permanent network splits occur.

As the first step of our analysis, we will show that nodes' permanently non-adjacent neighbors will eventually cease to be the nodes' preferred parents, as formalized in Lemma 1.

LEMMA 1. *Eventually always (i.e., starting from some moment t^s forever), each node will* not *have as its `prefpar` any neighbor that is eventually always non-adjacent. In other words, if R is ever P's `prefpar` after t^s, then $R \notin N^n_P$.*

Proof of Lemma 1. To show that such a moment t^s exists, consider an arbitrary node, P, and P's arbitrary neighbor that is eventually always non-adjacent, $R \in N^n_P$ (see Fig. 5). From the definition of N^n_P, at some moment $t^{na}_{P,R} \leq t^d$, R becomes forever non-adjacent to P. Therefore, from property RA3, at some moment $t^{nr}_{P,R}$, the entry n_R corresponding to R in P's `neighborset` (n_R`.id` $= R$), will forever be removed from the `neighborset` or have its n_R`.reachable` flag `false`, if n_R exists at all. In effect, from property OF5, starting from $t^{nr}_{P,R}$ forever, node R cannot be selected as P's `prefpar`: an entry for R either is absent from P's `neighborset` or violates condition (b) of OF5.

What is more, from property RA1, the change to P's `neighborset` at $t^{nr}_{P,R}$ will cause P at a later moment, $t^{np}_{P,R} \geq t^{nr}_{P,R}$, to reselect its `prefpar` to `null` or a neighbor different from R forever, if R has ever been P's `prefpar`. In other words, starting from $t^{np}_{P,R}$, R will never be P's `prefpar`.

Therefore, $t^{np}_P = max_{R \in N^n_P}(t^{np}_{P,R})$ is the moment starting from which node P will never have as its `prefpar` any neighbor that is permanently non-adjacent. Moreover, $t^{np} = max_{Q \in nodes}(t^{np}_Q)$ represents the moment starting from which no network node does ever have as its `prefpar` any eventually-always-non-adjacent neighbor. Finally, let $t^s =$

$max(t^d, t^{np})$, that is, starting from t^s, no new partitions occur globally and no nodes have eventually-always-non-adjacent neighbors as their `prefpar`s, which ends the proof. □

In the remainder of our analysis, let us denote as π an arbitrary partition without the DODAG root and as π' the remaining nodes. From Lemma 1, starting from t^s, any node in π always has as its `prefpar` some of its always-eventually-adjacent neighbors or `null`. Since for any $P \in \pi$, $N_P^a \subseteq \pi$, we can conclude that, starting from t^s, the nodes in π always have their `prefpar`s equal to `null` or selected only among themselves. If a node selects a `null prefpar`, then, from property OF4, it also sets its `rank` to infinity. Therefore, to prove Hypothesis 1, we have to show that the situation where nodes from π have other nodes from π as their `prefpar`s is temporary, that is, that all nodes from π will eventually always have their `prefpar`s and `rank`s equal to `null` and infinity, respectively. To this end, since OF4 is symmetric for `prefpar` and `rank`, we focus on the nodes' `rank`s.

Intuitively, if we proved that, starting from t^s, `rank` for each node in π only grew, we would have Hypothesis 1 largely proved. This is because a node's `prefpar`'s `rank` must not exceed the limit described by properties OF2 and OF5(e). In contrast, continuously growing integer `rank`s for each of the node's adjacent neighbors would eventually forever exceed the node's limit, thereby forcing the node to adopt an infinite `rank`, and hence a `null prefpar`. Unfortunately, this intuition is false as a node's `rank` can decrease after t^s. For example, the node can receive a DIO message, which reduces `n.rank` for one of its neighbor entries, `n`. In effect, `r` computed for `n`, as in property OF5, may decrease, which may in turn lead to a drop in the node's `rank`. Consequently, a more elaborate approach is required.

To this end, we define the following multi-sets for any moment in time $t \geq t^s$:

- $R_\pi^V(t) = \bigcup_{X \in \pi} \{rank_X\}$, encompassing the values at t of the `rank` variables for all nodes in π;
- $R_\pi^N(t) = \bigcup_{X \in \pi} \bigcup_{Y \in N_X^a} \{n.rank | n.id = Y \text{ and } n \in X\text{'s } neighborset\}$, containing the values at t of the `n.rank` fields for all `neighborset` entries, `n`, in π that can potentially be `prefpar`s of the nodes in π;
- $R_\pi^D(t) = \bigcup_{X \in \pi} \bigcup_{Y \in N_X^a} \{d.rank | d.id = Y \text{ and } d \in X\text{'s delivery multi-set}\}$, comprising the values at t of the `d.rank` fields for all DIO messages, `d`, in transit to the nodes in π from their their potential `prefpar`s.

Note that $R_\pi^V(t)$ and $R_\pi^N(t)$ cover only `rank`s and `neighborset`s of those nodes in π that are live at t; for nodes dead at t, we assume the variable values to be *null*s. In the case of $R_\pi^D(t)$, in turn, the situation is more complex. Although properties DIO1–DIO3 formalize how DIO reception behaves over time, they do not exhaustively specify what happens to each individual message. This is deliberate as it allows for modeling DIO message loss and duplication in a more general way: at scale not per message. In particular, in our model, it is perfectly valid that a DIO message from a neighbor is present in a node's delivery multi-set whereas the link from the neighbor and/or the node itself are dead. This is also justified as it models the case when, for instance, the message is still in the neighbor's transmission queue. As a consequence, however, when analyzing the impact of node and link deaths on $R_\pi^D(t)$ from the perspective of a particular DIO message, we have to consider the cases in which both the message disappears and remains in the target multi-set as a result of the death. In any case, however, from the global perspective, we assume that properties DIO1–DIO3 hold.

Given these multi-sets, we introduce a concept of *globally minimal rank* (*GMR*) in π, which we define as follows:

$$GMR_\pi(t) = min_{r \in R_\pi^V(t) \cup R_\pi^N(t) \cup R_\pi^D(t) \cup \{\infty\}}(r)$$

Intuitively, $GMR_\pi(t)$ is the minimal value of some node's `rank` (possibly from the past), which at the moment t resides anywhere in the π partition and may affect (at t or some later moment) the `rank`s of the nodes in the partition.

Since for any node in π, the node's `rank` at any moment t is always greater than to equal to $GMR_\pi(t)$, our goal will be to show that, GMR_π will always eventually permanently increase. We formalize this as Lemmas 2 and 3.

LEMMA 2. *For $t \geq t^s$, $GMR_\pi(t)$ never decreases.*

LEMMA 3. *For $t \geq t^s$, $GMR_\pi(t)$ always eventually increases, unless it is already infinite.*

Proof of Lemma 2. To prove that after t^s, GMR_π will never decrease, we need to analyze all events in the system that affect $GMR_\pi(t)$, that is, events that affect the three multi-sets: $R_\pi^V(t)$, $R_\pi^N(t)$, and $R_\pi^D(t)$. $R_\pi^V(t)$ is affected by a death and recovery of a node in π (properties OF3 and OF4), and by `prefpar` and `rank` reselection at a node in π (OF3, OF4, and OF5). No other events affect $R_\pi^V(t)$ (property OF1). $R_\pi^N(t)$ is affected by a reception by a node in π of a DIO message from an always-eventually-adjacent neighbor, a death and recovery of a node in π, and an addition and removal of entries for always-eventually-adjacent neighbors to/from the `neighborset` of a node in π. The effects of these events on $R_\pi^N(t)$ are all captured by a single property (RA2). Finally, $R_\pi^D(t)$ is affected by a transmission, reception, and loss of a DIO message from an always-eventually-adjacent neighbor to a node in π, as well as a death and recovery of a node in π or an in-π link. As mentioned previously, these effects are formalized by a suite of properties (DIO1–DIO3). Table 1 lists precisely how each of the aforementioned events affects each of the multi-sets $R_\pi^V(t)$, $R_\pi^N(t)$, and $R_\pi^D(t)$.

The events that only remove values from the sets, that is, node death, link death, neighbor entry removal, and DIO message loss, cannot lead to a decrease of GMR_π: at best GMR_π remains as it was before the event; at worst it increases. Likewise, the events that add infinite values to the sets, that is, node (re)start and neighbor entry addition cannot reduce GMR_π. The same is true for the event that copies values between the sets, that is, DIO message transmission, and the event that moves values between the sets, that is, DIO message reception: the copied/moved values already exist, and hence contribute to GMR_π. Since link recovery does not affect GMR_π at all, the only event that could potentially reduce GMR_π is rank reselection to a new value according

Table 1. The Effects of the Possible Events in π on $GMR_\pi(t)$

Event	Effect
node death	removal from $R^V_\pi(t)$ of the node's `rank` value; removal from $R^N_\pi(t)$ of `n.rank` values for the node's all neighbor entries `n`; possible removal from $R^D_\pi(t)$ of `d.rank` values for none/some/all DIO messages `d` both in the node's delivery multi-set and, for `d` sent by the node, in the node's neighbors' delivery multi-sets;
node (re)start	addition to $R^V_\pi(t)$ of the node's `rank`, which is *infinite* per property OF4;
link death	possible removal from $R^D_\pi(t)$ of `d.rank` values for none/some/all DIO messages `d` in the target node's delivery multi-set;
link recovery	none
neighbor entry addition	addition to $R^N_\pi(t)$ of the new entry's `n.rank` value, which is *infinite* per property RA2;
neighbor entry removal	removal from $R^N_\pi(t)$ of the evicted entry's `n.rank` value;
DIO message transmission	addition to $R^D_\pi(t)$ (possibly multiple times if the message is multicast) of the transmitted message's `d.rank` value, which, per property DIO1, is equal to the transmitting node's `rank`, and hence already exists in $R^V_\pi(t)$; in other words, copying the value from $R^V_\pi(t)$ to $R^D_\pi(t)$, possibly multiple times;
DIO message reception	removal from $R^D_\pi(t)$ of the received message's `d.rank` value; possible replacement in $R^N_\pi(t)$ of `n.rank` value for the `d.rank` value, per property RA2, if a neighbor entry `n` with `n.id` = `d.id` exists in the receiving node's `neighborset`;
DIO message loss	removal from $R^D_\pi(t)$ of the lost message's `d.rank` value;
parent & rank reselection	replacement in $R^V_\pi(t)$ of the reselecting node's old `rank` with the new value `r`, computed according to property OF5;

to property OF5. However, from OF5(d), the new value is greater at least by `MinHopRankIncrease` than some existing value that contributes to GMR_π. The reselection thus cannot reduce GMR_π. Since no other events affect GMR_π, its value never decreases after t^s, which ends the proof. □

Proof of Lemma 3. Having proved that any increase of GMR_π after t^s is permanent, we just have to show that such increases will repeatedly occur until GMR_π becomes infinite. Therefore, let $t_i \geq t^s$ be an arbitrary moment. To prove that, unless infinite, GMR_π must increase after t^s, we have to show that from some later moment $t_j > t_i$, there will forever be no occurrences of $GMR_\pi(t_i)$, unless $GMR_\pi(t_i)$ is infinite, in any of the three multi-sets: $R^V_\pi(t)$, $R^N_\pi(t)$, and $R^D_\pi(t)$.

For $R^V_\pi(t)$, consider an arbitrary node $P \in \pi$, such that P's `rank` at t_i equals $GMR_\pi(t_i)$. Suppose that $t^V_P > t_i$ is the earliest moment at which any event affecting P's `rank` occurs.

If P is dead at t_i, then it does not contribute its `rank` to $R^V_\pi(t_i)$, and the event at t^V_P must be P's recovery. In this case, a new value will appear in $R^V_\pi(t^V_P)$ as P's `rank`, but this value is infinity per OF4, and hence will not affect $GMR_\pi(t^V_P)$.

Therefore, assume that P is live at t_i and its `rank` = $GMR_\pi(t_i) = r_i$. The event at t^V_P is thus either P's death or `prefpar` and `rank` reselection. If the event is P's death, then P's `rank` will be absent from $R^V_\pi(t^V_P)$, that is, an occurrence of r_i will disappear from $R^V_\pi(t^V_P)$. If, in turn, the event is `prefpar` and `rank` reselection, then it abides by property OF5. Consequently, if P selects infinite `rank`, then an occurrence of r_i will be replaced in $R^V_\pi(t^V_P)$ by infinity. If, in turn, P selects a finite `rank`, equal to some r', then $r' \geq r_i$+`MinHopRankIncrease`$> r_i$. This is because, from property OF5(d), $r' \geq$`n.rank`+`MinHopRankIncrease` for some entry, `n`, in P's `neighborset`. However, from the definition of t^s, the neighbor corresponding to `n` must be always eventually adjacent to P. Therefore, `n.rank` must belong to $R^N_\pi(t^V_P)$, which, from Lemma 2, implies that at t^V_P, `n.rank`$\geq r_i$. In short, also in this case, the occurrence of r_i for P in $R^V_\pi(t^V_P)$ will be replaced by a larger value, r'.

What is more, no occurrence of r_i corresponding to P's `rank` will ever reappear in $R^V_\pi(t)$ after t^V_P, unless r_i is infinite. Suppose, by contradiction, that r_i is finite and P's `rank` becomes r_i at some moment $t'_P > t^V_P$. This can only be due to some of the events affecting P's `rank`. Neither P's death nor P's recovery can bring P's rank to a finite value, though. Therefore, this must be due `prefpar` and `rank` reselection. The selection of r_i as P's `rank` at t'_P implies in turn the existence at t'_P of a entry `n` in P's `neighborset`, such that `n.rank` $\leq r_i -$`MinHopRankIncrease` (property OF5(d)). Since $t'_P > t^s$, then the neighbor corresponding to `n` must be always eventually adjacent to P, which entails `n.rank` $\in R^N_\pi(t'_P)$. This, however, means that $GMR_\pi(t'_P) \leq$`n.rank`$< r_i = GMR_\pi(t_i)$, that is, GMR_π must have decreased from t_i to t'_P, which contradicts Lemma 2.

To sum up, indeed $GMR_\pi(t_i)$ equal to some finite r_i and corresponding to P's `rank` forever disappears from $R^V_\pi(t)$, but only if there exists the moment $t^V_P > t_i$ at which some of the events affecting P's `rank` occurs. What thus remains to be shown is that such a moment does exist. Since P's liveness is beyond RPL's control, we cannot guarantee an occurrence of P's death and recovery after t_i. Let us thus focus on P's `prefpar` and `rank` reselection events. By contradiction, assume that no such event takes place after t_i. As $GMR_\pi(t_i)$ = r_i, for every entry `n` in P's `neighborset` corresponding to an always-eventually-adjacent neighbor, we must have `n.rank`$\geq r_i$ at t_i. Since $t_i \geq t^s$, only an always-eventually-adjacent neighbor can be P's `prefpar` at and after t_i. This means that at t_i there is no entry `n` in P's `neighborset`, for which `r` computed according to property OF5, notably condition (d), would be equal to P's `rank` $= r_i$. Therefore, P's `neighborset` must have changed at or before t_i and no P's `prefpar` and `rank` reselection event has taken place since then. However, from property RA1, such an event is guaranteed to take place, which contradicts our assumption that no such event occurs after t_i. In other words, the moment t^V_P at which $GMR_\pi(t_i)$ corresponding to P's `rank`, unless infinite, forever disappears from $R^V_\pi(t)$ indeed exists for P.

All in all, we have shown that for any finite $GMR_\pi(t_i)$ and any node $P \in \pi$, there exists a moment $t^V_P > t_i$ at which the occurrence of $GMR_\pi(t_i)$ corresponding to P's `rank` disappears forever from $R^V_\pi(t)$, if it has ever existed. We can thus take $t^V_j = max_{P \in \pi}(t^V_P)$ as the moment from which any occurrence of $GMR_\pi(t_i)$, if finite, is forever absent from $R^V_\pi(t)$.

Given this, proving the same for the set of DIOs in transit, $R^D_\pi(t)$, is straightforward. The only event producing new

Table 2. Violations of RPL's Properties Relevant to Partition Handling in the Implementations' Sources

Properties	Are properties satisfied?	
	TinyRPL	ContikiRPL
RA1	NO: reselection is not triggered when a node receives a DIO message advertising an infinite rank and removes the message's sender from its `neighborset`	YES
RA2	YES	YES
RA3	YES	NO: non-adjacent neighbors are not marked as unreachable at all
OF1, OF3, OF4, OF5	YES	YES
OF2	NO: `minrank` is equal to a node's `rank` when it selected its `prefpar` for the last time	NO: `minrank` is equal to a node's `rank` when it changed a `null prefpar` to a non-`null prefpar` for the last time
DIO1, DIO3	YES	YES
DIO2	NO: a node stops transmitting DIO messages when it selects a `null prefpar` and adopts an infinite `rank`	YES

values in $R^D_\pi(t)$ is DIO message transmission, and the values produced, the `d.rank` fields of the DIO messages `d`, are copied from the nodes' `rank`s (property DIO1). From t^V_j forever, no node in π has its `rank` equal to $GMR_\pi(t_i)$, unless $GMR_\pi(t_i)$ is infinite. Therefore, from this moment no new occurrences of $GMR_\pi(t_i)$ will ever appear in $R^D_\pi(t)$, unless $GMR_\pi(t_i)$ is infinite. Moreover, each DIO in transit is eventually received or lost for the last time (property DIO3), and thus the corresponding `d.rank` value disappears from $R^D_\pi(t)$. Consequently, there exists a moment in time, t^D_j, from which $GMR_\pi(t_i)$, if finite, will never appear in $R^D_\pi(t)$.

A similar reasoning can be used for $R^N_\pi(t)$. From property RA2, the value of the `n.rank` field of an entry `n` for a neighbor in a node's `neighborset` either is copied from the `d.rank` field of the last DIO message `d` received by the node from the neighbor or is infinite if the node has received no message from the neighbor since the entry was added to the `neighborset`. From t^D_j, no DIO messages received by the nodes in π have their `d.rank` fields equal to $GMR_\pi(t_i)$, if $GMR_\pi(t_i)$ is finite. Therefore, no new neighbor entries of the nodes will have their `n.rank` fields changed to $GMR_\pi(t_i)$, unless $GMR_\pi(t_i)$ is infinite. What is more, as $R^N_\pi(t)$ covers only entries to always-eventually-adjacent neighbors and, from property DIO2, a node always eventually receives a DIO message from such a neighbor, any existing neighbor entries `n` with their `n.rank` fields equal to $GMR_\pi(t_i)$ will eventually have these fields forever changed to different values, unless $GMR_\pi(t_i)$ is infinite. In other words, there exists a moment, t^N_j, at which the last occurrence of $GMR_\pi(t_i)$, if finite, disappears from $R^N_\pi(t)$ forever.

All in all, there exists a moment, $t_j = max(t^V_j, t^N_j, t^D_j)$, starting from which $GMR_\pi(t_i)$, unless infinite, is absent from all three multi-sets: $R^V_\pi(t)$, $R^N_\pi(t)$, and $R^D_\pi(t)$. This changes $GMR_\pi(t)$ forever, unless it is infinite. Since from Lemma 2, $GMR_\pi(t)$ never decreases after t^s, the change must be an increase, which ends the proof of Lemma 3. □

The last guarantee we need for our reasoning is that a node's `rank` is bounded, as formalized in Lemma 4.

LEMMA 4. *Always, if a node is live, then its* `rank` *is infinite or does not exceed* `minrank` + `MaxRankIncrease`.

Proof. Recall that the only events affecting a node's `rank` are the node's death, restart, and `prefpar` and `rank` reselection (property OF1). The node's death is irrelevant because the lemma assumes the node is live. Upon the node's recovery, in turn, the node's `rank` is set to infinity (OF4). Finally, upon reselection, the node's `rank` is set to infinity or a value that, among others, is not greater than the node's `minrank` plus `MaxRankIncrease` (OF5). □

Given the four lemmas, we are able to prove Theorem 1.

THEOREM 1. *Hypothesis 1 is true.*

Proof. Consider an arbitrary, always-eventually-live node *P* that is eventually always partitioned from the DODAG root. From Lemmas 1–3, unless infinite, GMR_π will repeatedly permanently increase. Since *P*'s `rank` is always at least GMR_π, it will repeatedly permanently increase as well. From Lemma 4, in turn, the number of such increases is limited. This is because, first, *P*'s `rank` is an integer value, and thus any increase is at least by one, and second, *P*'s `rank`, if finite, is upper-bounded by *P*'s `minrank` plus `MaxRankIncrease`, where `minrank` is the minimal value so far of *P*'s `rank` (property OF2). The number of *P*'s `rank` increases will thus be at most `MaxRankIncrease` before *P* will be forced to make its `rank` infinite forever. Finally, from properties OF4 and OF1, *P*'s `rank` is infinite iff *P*'s `prefpar` is `null`. Therefore, indeed, eventually always, when live, *P* will have its `prefpar` and `rank` equal to `null` and infinity, respectively, which ends the proof. □

5 Fixing RPL's Implementations

Let us summarize our findings hitherto. On the one hand, Sect. 3 demonstrates that the two popular RPL's implementations fail to correctly handle network partitions. On the other hand, Sect. 4 formally proves that RPL itself is designed to behave correctly in these situations. It can thus be concluded that the implementations do not satisfy some of the formal properties from Sect. 4.1 that describe RPL's operation.

Therefore, here we analyze the implementations' sources, to identify the properties that they violate, and propose fixes for these defects. We then check, both in simulations and on a ∼100-node testbed, that the corrected implementations behave as expected under network partitions—and in general.

5.1 Implementation Defects and Fixes

Because of space constrains and the intricacy of the identified implementation defects, instead of explaining each of the defects, we just give a summary in Table 2. As can

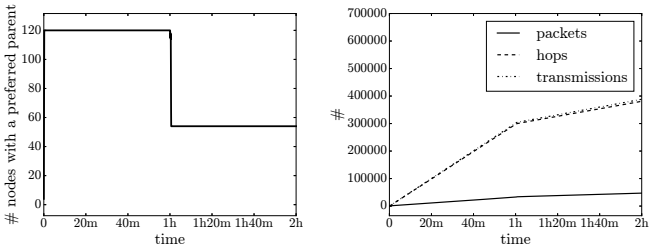
be observed in the table, both implementations do not satisfy property OF2. Yet, the property is crucial for limiting a node's rank growth, which is RPL's major failure detection mechanism. Nevertheless, this is not the only reason for the implementations' misbehavior, as Table 2 shows that both implementations also fail to satisfy other properties.

There are several possible reasons for the property violations. Property OF2 is covered in RPL's specification but is not stated as explicitly as in our model. In addition, in the implementations' sources, it is affected in multiple places. Our work, in turn, entails global verification of its precise formulation, which facilitates identifying violations. In contrast, it is symptomatic that the other violated properties concern the issues that RPL's specification leaves open. The developers of the solutions to those issues must have invented custom suites of properties. Apparently, doing this correctly is not straightforward. Therefore, again, with provable guarantees on their utility, the properties introduced here can be of value.

In particular, we have developed source code patches ensuring that all properties from Sect. 4.1 are satisfied in both TinyRPL and ContikiRPL. We have verified empirically, in simulations and on a testbed, that the patches actually correct the implementations' behavior and that they do not impair the implementations' performance in other scenarios. Here, we present just an illustrative subset of these experiments.

5.2 Simulation-Based Evaluation

Figure 6 shows, for instance, the results of the same experiment as in Sect. 3 but for TinyRPL with our patches. They match the expected results from Fig. 2. Within seconds after the partition, the 66 nodes without paths to the DODAG root discard their preferred parents, as visible in Fig. 6(a), and adopt infinite ranks. Moreover, a significant drop in the network traffic can be observed in Fig. 6(b), which is due to the 66 nodes ceasing to forward packets to the root.



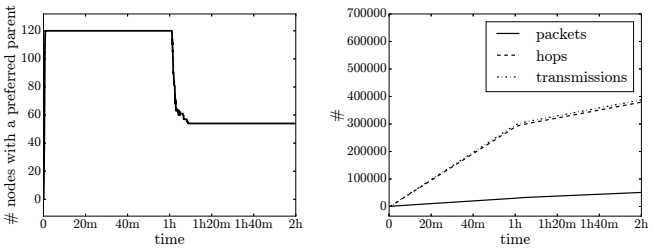
(a) nodes with preferred parents (b) accumulated network traffic

Figure 6. Patched TinyRPL under a partition.

Figure 7 shows the same results but for ContikiRPL. Like TinyRPL, ContikiRPL after fixes behaves in accordance with RPL's specification. All 66 nodes discard their preferred parents, as visible in Fig. 7(a), and stop forwarding packets to the root, as can be observed in Fig. 7(b). The difference in the reaction time compared to TinyRPL is due to differences in the solutions for routing adjacency maintenance and Trickle timer configurations in the two implementations.

5.3 Testbed-Based Evaluation

In addition to the simulations, we demonstrate representative results from a real-world indoor testbed, Indriya [9]. The



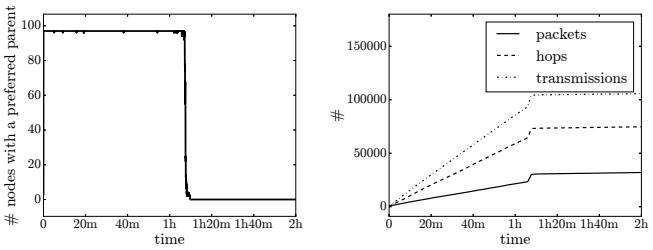
(a) nodes with preferred parents (b) accumulated network traffic

Figure 7. Patched ContikiRPL under a partition.

experiment involved 98 TelosB nodes, dispersed across three floors and connected with low-power radio links of varying quality. The nodes ran TinyRPL, reporting relevant statistics through their wired serial interfaces, so as not to interfere with the radio traffic. The DODAG root was the node with identifier 1, located in the corridor of the first floor.

Like the simulations, the experiment lasted 2 hours, with a partition after 1 hour and 5 minutes. Those 5 minutes were to account for node programming, which was not immediate. Moreover, because of the different network topology, we would not be able to mimic on the testbed the partition scenarios adopted hitherto. Therefore, the scenario of the experiment was different: it emulated the extreme case of network partition—a crash of the DODAG root—causing all live nodes to become disconnected, as in Fig. 1(d). The remaining configuration parameters were as in the simulations.

Figure 8 shows the results. They match well those obtained for TinyRPL and ContikiRPL in simulations (cf. Fig. 6 and 7). As can be observed in Fig. 8(a), all nodes quickly discarded their preferred parents after the failure. Likewise, a significant drop in network traffic is visible in Fig. 8(b) when the nodes ceased to forward packets to the root, exchanging only DIO messages at a low frequency.



(a) nodes with preferred parents (b) accumulated network traffic

Figure 8. Patched TinyRPL under a partition on Indriya.

There are a few differences from the simulations, though. As visible in Fig. 8(a), the nodes were occasionally discarding their preferred parents for short periods of time. Nevertheless, this phenomenon was due to the instability of some wireless links, which is expected in the real world. Moreover, the drop in traffic in Fig. 8(b) was more pronounced but preceded by a larger temporary growth than in simulations. This is again expected as all nodes got disconnected: they all had to reset their Trickle timers in reaction to the partition, hence the larger growth, and none of them was forwarding any packets to the root when the partition was

handled, hence the more pronounced drop later. The same can be observed when the DODAG root crash is simulated. All in all, the testbed results confirm the ones from simulations. Likewise, the results of further experiments, both on testbeds and in simulations, confirm that the patches indeed correct the implementations' behavior under partitions, without negative effects in other scenarios.

6 Conclusions

In conclusion, we showed that TinyRPL and ContikiRPL fail to handle network partitions. None of them detects when the host node gets disconnected from the DODAG root, which precludes correctly notifying higher layers about the lack of a default route. In effect, these layers waste network resources, generating packets that never reach their destinations. Moreover, in TinyRPL, a partition leads to an increase in control traffic, which further drains the resources.

To remedy this problem, we developed RPL's formal model, based directly on the protocol's specification, and used it to prove that—contrary to the implementations' observed behavior—RPL's design does guarantee correct partition handling. An important contribution of the process was not only the guarantees but also a set of formal properties that an implementation must satisfy for the guarantees to hold.

As an application of these theoretical results in practice, we analyzed the two implementations' source code with the focus on violations of the properties. In effect, we identified a number of such code defects and patched them. Because of the nature of these defects, identifying them would have been difficult without our results. By reevaluating the implementations experimentally, we showed that, in effect of our work, they became capable of handling network partitions.

From a broader perspective, in turn, our results confirm that, despite the seeming maturity of RPL's implementations, their reliability leaves room for improvement. We have been addressing this issue in our recent research activities.

7 Acknowledgments

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 also supported by the Polish Ministry of Science and Higher Education with a scholarship for outstanding young scientists.

8 References

[1] P. Barooah, H. Chenji, R. Stoleru, and T. Kalmar-Nagy. Cut detection in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):483–490, 2012.

[2] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer-Verlag London, 3rd edition, 2012.

[3] J. Beutel, K. Römer, M. Ringwald, and M. Woehrle. Deployment techniques for sensor networks. In G. Ferrari, editor, *Sensor Networks: Where Theory Meets Practice*. Springer Berlin Heidelberg, 2009.

[4] A. Brachman. RPL objective function impact on LLNs topology and performance. In *NEW2AN'13*, 2013.

[5] E. Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2):23–29, February 2012.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer-Verlag Berlin Heidelberg, 2nd edition, 2011.

[7] T. Clausen, U. Herberg, and M. Philipp. A critical evaluation of the IPv6 routing protocol for low power and lossy networks (RPL). In *Proc. WiMob'11*, 2011.

[8] E. A. Devi and J. M. L. Manickam. Detecting and repairing network partition in wireless sensor networks. In *Proc. ICCPCT '14*, 2014.

[9] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda. Indriya: A low-cost, 3D wireless sensor network testbed. In *Proc. TridentCom '11*, 2012.

[10] S. Duquennoy, O. Landsiedel, and T. Voigt. Let the tree bloom: Scalable opportunistic routing with ORPL. In *Proc. SenSys'13*, 2013.

[11] O. Gaddour and A. Koubâa. RPL in a nutshell: A survey. *Computer Networks*, 56(14):3163–3178, 2012.

[12] O. Gaddour, A. Koubâa, S. Chaudhry, M. Tezeghdanti, R. Chaari, and M. Abid. Simulation and performance evaluation of DAG construction with RPL. In *ComNet'12*, 2012.

[13] O. Gnawali and P. Levis. The minimum rank with hysteresis objective function. RFC 6719, 2012.

[14] M. Gregorczyk, T. Pazurkiewicz, and K. Iwanicki. On decentralized in-network aggregation in real-world scenarios with crowd mobility. In *Proc. DCOSS '14*, 2014.

[15] D. Han and O. Gnawali. Performance of RPL under wireless interference. *IEEE Communications Magazine*, 51(12):137–143, 2013.

[16] R. Hartung, U. Kulau, B. Gernert, S. Rottmann, and L. Wolf. On the experiences with testbeds and applications in precision farming. In *Proc. FAILSAFE '17*, 2017.

[17] K. Heurtefeux, H. Menouar, and N. AbuAli. Experimental evaluation of a routing protocol for WSNs: RPL robustness under study. In *WiMob'13*, October 2013.

[18] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[19] O. Iova, F. Theoleyre, and T. Noel. Stability and efficiency of RPL under realistic conditions in wireless sensor networks. In *PIMRC'13*, 2013.

[20] T. Istomin, C. Kiraly, and G. P. Picco. Is RPL ready for actuation? A comparative evaluation in a smart city scenario. In *Proc. EWSN'15*, 2015.

[21] K. Iwanicki. RNFD: Routing-layer detection of DODAG (root) node failures in low-power wireless networks. In *Proc. IPSN'16*, 2016.

[22] N. Khelifi, W. Kammoun, and H. Youssef. Efficiency of the RPL repair mechanisms for low power and lossy networks. In *IWCMC'14*, 2014.

[23] J. Kleinberg, M. Sandler, and A. Slivkins. Network failure detection and graph connectivity. In *Proc. SODA '04*, 2004.

[24] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-Haggerty, A. Terzis, A. Dunkels, and D. Culler. ContikiRPL and TinyRPL: Happy together. In *Proc. IP+SN '11*, 2011.

[25] K. D. Korte, A. Sehgal, and J. Schönwälder. A study of the RPL repair process using ContikiRPL. In *Proc. AIMS'12*, 2012.

[26] P. Levis, T. Clausen, J. Hui, O. Gnawali, and K. Jo. The Trickle algorithm. RFC 6206, 2011.

[27] L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi. Anquiro: Enabling efficient static verification of sensor network software. In *Proc. SESENA '10*, 2010.

[28] I. E. Radoi, A. Shenoy, and D. Arvind. Evaluation of routing protocols for Internet-enabled wireless sensor networks. In *Proc. ICWMC'12*, 2012.

[29] N. Shrivastava, S. Suri, and C. D. Tóth. Detecting cuts in sensor networks. *ACM Trans. Sen. Netw.*, 4(2):10:1–10:25, 2008.

[30] P. Thubert. Objective function zero for the routing protocol for low-power and lossy networks (RPL). RFC 6552, 2012.

[31] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. EWSN '05*, 2005.

[32] J. Tripathi, J. C. de Oliveira, and J. P. Vasseur. A performance evaluation study of RPL: Routing protocol for low power and lossy networks. In *Proc. CISS'10*, 2010.

[33] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J.-P. Vasseur, and R. Alexander. RPL: IPv6 routing protocol for low-power and lossy networks. RFC 6550, 2012.

[34] M. Won and R. Stoleru. A destination-based approach for cut detection in wireless sensor networks. *International Journal of Parallel, Emergent and Distributed Systems*, 28(3):266–288, 2013.

[35] W. Xie, M. Goyal, H. Hosseini, J. Martocci, Y. Bashir, E. Baccelli, and A. Durresi. Routing loops in DAG-based low power and lossy networks. In *Proc. AINA '10*, 2010.