| | |
|---|---|
| Networking Working Group | T. Winter, Ed. |
| Internet-Draft | |
| Intended status: Standards Track | ROLL Design Team |
| Expires: February 5, 2010 | IETF ROLL WG |
| | August 04, 2009 |

# RPL: Routing Protocol for Low Power and Lossy Networks
**draft-ietf-roll-rpl-00**

## Abstract

This document specifies the Routing Protocol for Low Power and Lossy Networks (RPL), in accordance with the requirements described in [I-D.ietf-roll-building-routing-reqs] (Martocci, J., Riou, N., Mil, P., and W. Vermeylen, “Building Automation Routing Requirements in Low Power and Lossy Networks,” February 2009.), [I-D.ietf-roll-home-routing-reqs] (Porcu, G., “Home Automation Routing

Requirements in Low Power and Lossy Networks,” November 2008.), [I-D.ietf-roll-indus-routing-reqs] (Networks, D., Thubert, P., Dwars, S., and T. Phinney, “Industrial Routing Requirements in Low Power and Lossy Networks,” June 2009.), and [RFC5548] (Dohler, M., Watteyne, T., Winter, T., and D. Barthel, “Routing Requirements for Urban Low-Power and Lossy Networks,” May 2009.).

**Requirements Language**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” March 1997.) [RFC2119].

---

**Table of Contents**

## 1. Introduction

The defining characteristics of Low Power and Lossy Networks (LLNs) offer unique challenges to a routing solution. The IETF ROLL Working Group has defined application-specific routing requirements for a Low Power and Lossy Network (LLN) routing protocol [I-D.ietf-roll-building-routing-reqs] (Martocci, J., Riou, N., Mil, P., and W. Vermeylen, “Building Automation Routing Requirements in Low Power and Lossy Networks,” February 2009.) [I-D.ietf-roll-home-routing-reqs] (Porcu, G., “Home Automation Routing Requirements in Low Power and Lossy Networks,” November 2008.) [I-D.ietf-roll-indus-routing-reqs] (Networks, D., Thubert, P., Dwars, S., and T. Phinney, “Industrial Routing Requirements in Low Power and Lossy Networks,” June 2009.) [RFC5548] (Dohler, M., Watteyne, T., Winter, T., and D. Barthel, “Routing Requirements for Urban Low-Power and Lossy Networks,” May 2009.). RPL is a new routing protocol designed to meet these requirements.

## 1.1. Design Principles

TOC

RPL was designed with the objective to meet the requirements spelled out in [I-D.ietf-roll-building-routing-reqs] (Martocci, J., Riou, N., Mil, P., and W. Vermeylen, “Building Automation Routing Requirements in Low Power and Lossy Networks,” February 2009.), [I-D.ietf-roll-home-routing-reqs] (Porcu, G., “Home Automation Routing Requirements in Low Power and Lossy Networks,” November 2008.), [I-D.ietf-roll-indus-routing-reqs] (Networks, D., Thubert, P., Dwars, S., and T. Phinney, “Industrial Routing Requirements in Low Power and Lossy Networks,” June 2009.), and [RFC5548] (Dohler, M., Watteyne, T., Winter, T., and D. Barthel, “Routing Requirements for Urban Low-Power and Lossy Networks,” May 2009.). Because those requirements are heterogeneous and sometimes incompatible in nature, the approach is first taken to design a protocol capable of supporting a core set of functionalities corresponding to the intersection of the requirements. (Note: it is intended that as this design evolves optional features may be added to address some application specific requirements). All "MUST" application requirements that cannot be satisfied by RPL will be specifically listed in the Appendix A, accompanied by a justification. The core set of functionalities is to be capable of operating in the most severely constrained environments, with minimal requirements for memory, energy, processing, communication, and other consumption of limited resources from nodes. Trade-offs inherent in the provisioning of protocol features will be exposed to the implementer in the form of configurable parameters, such that the implementer can further tweak and optimize the operation of RPL as appropriate to a specific application and implementation. Finally, RPL is designed to consult implementation specific policies to determine, for example, the evaluation of routing metrics.
A set of companion documents to this specification will provide further guidance in the form of applicability statements specifying a set of operating points appropriate to the Building Automation, Home Automation, Industrial, and Urban application scenarios.

TOC

## 2. Terminology

The terminology used in this document is consistent with and incorporates that described in `Terminology in Low power And Lossy Networks' [I-D.ietf-roll-terminology] (Vasseur, J., “Terminology in Low power And Lossy Networks,” March 2010.). The terminology is extended in this document as follows:

**Autonomous:** Refers to the ability of a routing protocol to independently function without requiring any external influence or guidance. Includes self-organization capabilities.

**DAG:** Directed Acyclic Graph- A directed graph having the property that all edges are oriented in such a way that no cycles exist. In the RPL context, all edges are contained in paths oriented toward and terminating at a root node (a DAG root, or sink-typically a LBR).

**DAGID:** DAG Identifier- A globally unique identifier for a DAG. All nodes who are members of a DAG have knowledge of the DAGID. This knowledge is used to identify peer nodes within the DAG in order to coordinate DAG Maintenance while avoiding loops.

**DAG Parent:** A parent of a node within a DAG is one of the immediate successors of the node on a path towards the DAG root. For each DAGID that a node is a member of, the node will maintain a set containing one or more DAG Parents. If a node is a member of multiple DAGs then it must conceptually maintain a set of DAG Parents for each DAGID.

**DAG Sibling:** A sibling of a node within a DAG is defined to be any neighboring node which is located at the same depth, or rank, within a DAG. Note that siblings defined in this manner do not necessarily share a common parent. For each DAGID that a node is a member of, the node will maintain a set of DAG Siblings. If a node is a member of multiple DAGs then it must conceptually maintain a set of DAG Siblings for each DAGID.

**DAG Root:** A DAG root is a sink within the DAG graph. All paths in the DAG terminate at a DAG root, and all DAG edges contained in the paths terminating at a DAG root are oriented toward the DAG root. There must be at least one DAG Root per DAGID, and in some cases there may be more than one. In many use cases, source-sink represents a dominant traffic flow, where the sink is a DAG root. Maintaining default routing towards DAG roots is therefore a prominent functionality for RPL.

**Grounded:** A DAG is grounded if it contains a DAG Root offering a default route.

**Floating:**
A DAG is floating if it contains a DAG root that does not offer a default route.

**Inward:** In the context of RPL, inward refers to the direction from leaf nodes towards DAG roots, following the orientation of the edges within the DAG.

**Outward:** In the context of RPL, outward refers to the direction from DAG roots towards leaf nodes, going against the orientation of the edges within the DAG.

**P2P:** Point-to-point. This refers to traffic exchanged between two nodes.

**P2MP:** Point-to-Multipoint. This refers to traffic between one node and a set of nodes. This is similar to the P2MP concept in Multicast or MPLS Traffic Engineering ([RFC4461] (Yasukawa, S., "Signaling Requirements for Point-to-Multipoint Traffic-Engineered MPLS Label Switched Paths (LSPs)," April 2006.) and [RFC4875] (Aggarwal, R., Papadimitriou, D., and S. Yasukawa, "Extensions to Resource Reservation Protocol - Traffic Engineering (RSVP-TE) for Point-to-Multipoint TE Label Switched Paths (LSPs)," May 2007.)).

**MP2P:** Multipoint-to-Point; used to describe a particular traffic pattern. A common RPL use case involves MP2P flows collecting information from many nodes in the DAG, flowing inwards towards DAG roots. Note that a DAG root may not be the ultimate destination of the information, but it is a common transit node.

**OCP:** Objective Code Point. In RPL, the Objective Code Point (OCP) indicates which routing metrics, optimization objectives, and related functions are in use in a DAG. It is recommended that a companion document define instances of the Objective Code Point and request the creation of a registry to manage them.

Note that in this document, the terms `node' and `LLN router' are used interchangeably.

---

## 3. Protocol Model

The aim of this section is to describe RPL in the spirit of [RFC4101] (Rescorla, E. and IAB, "Writing Protocol Models," June 2005.). An architectural protocol overview (the big picture of the protocol) is provided in this section. Protocol details can be found in further sections.

## 3.1. Problem

TOC

Some well-defined LLN application-specific scenarios are Building Automation, Home Automation, Industrial, and Urban; for which the unique routing requirements have been detailed respectively in [I-D.ietf-roll-building-routing-reqs] (Martocci, J., Riou, N., Mil, P., and W. Vermeylen, “Building Automation Routing Requirements in Low Power and Lossy Networks,” February 2009.), [I-D.ietf-roll-home-routing-reqs] (Porcu, G., “Home Automation Routing Requirements in Low Power and Lossy Networks,” November 2008.), [I-D.ietf-roll-indus-routing-reqs] (Networks, D., Thubert, P., Dwars, S., and T. Phinney, “Industrial Routing Requirements in Low Power and Lossy Networks,” June 2009.), and [RFC5548] (Dohler, M., Watteyne, T., Winter, T., and D. Barthel, “Routing Requirements for Urban Low-Power and Lossy Networks,” May 2009.). Within these application-specific scenarios there are some common elements required of routing. RPL intends to address the requirements of these application-specific scenarios, and it is further intended to be flexible enough to extend to other application scenarios.

## 3.2. Protocol Properties Overview

TOC

RPL demonstrates the following properties, consistent with the requirements specified by the requirements documents.

### 3.2.1. IPv6 Architecture

TOC

RPL is strictly compliant with layered IPv6 architecture. Further, RPL is designed with consideration to the practical support and implementation of IPv6 architecture on devices which may operate under severe resource constraints, including but not limited to memory, processing power, energy, and communication. The RPL design does not presume high quality reliable links, and should be able to operate over lossy links (usually low bandwidth with low packet delivery success rate).

TOC

### 3.2.2. Path Properties for LLN Traffic Flows

Multipoint-to-point (MP2P) and Point-to-multipoint (P2MP) traffic flows from nodes within the LLN from and to egress points are very common in LLNs. Low power and lossy network Border Router (LBR) nodes may typically be at the root of such flows, although such flows are not exclusively rooted at LBRs as determined on an application-specific basis.

As required by the aforementioned routing requirements documents, RPL supports the installation of multiple paths. The use of multiple paths include sending duplicated traffic along diverse paths, as well as to support advanced features such as Class of Service (CoS) based routing, or simple load balancing among a set of paths (which could be useful for the LLN to spread traffic load and avoid fast energy depletion on some nodes).

---

### 3.2.3. Constraint Based Routing

The RPL design supports constraint based routing, based on a set of routing metrics. The routing metrics supported by RPL are specified in a companion document to this specification, [I-D.ietf-roll-routing-metrics] (Vasseur, J., Kim, M., Networks, D., and H. Chong, “Routing Metrics used for Path Calculation in Low Power and Lossy Networks,” April 2010.). RPL signals the metrics and related objective functions in use in a particular implementation by means of an Objective Code Point (OCP).

RPL supports the computation and installation of different paths in support of and optimized for a set of application and implementation specific constraints, as guided by an OCP. Traffic may subsequently be directed along the appropriate constrained path based on traffic marking within the IPv6 header. For more details on the approach towards constraint-based routing, see Section 4 (Constraint Based Routing in LLNs).

---

### 3.2.4. Autonomous Operation

Nodes running RPL are able to independently and autonomously discover a network topology and compute and install routes, without requiring further administrative interaction.

---

## 3.3. Protocol Operation

LLN nodes running RPL will construct Directed Acyclic Graphs (DAGs) rooted at designated nodes that generally provide default routes. The DAG is sufficient to support inward MP2P traffic flows, flowing inward along the LLN towards a sink (DAG Root), which is one of the dominant traffic flows described in the requirements documents ([I-D.ietf-roll-building-routing-reqs] (Martocci, J., Riou, N., Mil, P., and W. Vermeylen, "Building Automation Routing Requirements in Low Power and Lossy Networks," February 2009.), [I-D.ietf-roll-home-routing-reqs] (Porcu, G., "Home Automation Routing Requirements in Low Power and Lossy Networks," November 2008.), [I-D.ietf-roll-indus-routing-reqs] (Networks, D., Thubert, P., Dwars, S., and T. Phinney, "Industrial Routing Requirements in Low Power and Lossy Networks," June 2009.), and [RFC5548] (Dohler, M., Watteyne, T., Winter, T., and D. Barthel, "Routing Requirements for Urban Low-Power and Lossy Networks," May 2009.)).

By utilizing a DAG for dominant MP2P flows, RPL allows each node to select and maintain potentially multiple successors capable of forwarding traffic inwards towards the root. The DAG does not present as many single points of failure as a tree, and in addition can offer a node a set of pre-computed successors in support of, e.g. local route repair in case of a temporary failure, load balancing, or short term fluctuations in link characteristics.

A DAG also serves to restrict the routing problem on the nodes when it is used as a reference topology. This allows nodes to determine their positions in a DAG relative to each other and provides a means to coordinate route repair in a way that endeavors to avoid loops. These mechanisms will be described in more detail later in this specification.

As DAGs are organized, RPL will use a Destination Advertisement mechanism to build up routing state in support of outward P2MP traffic flows. This mechanism, using the DAG as a reference, `paints' the underlying LLN graph, guided along the DAG, such that the routes toward destination prefixes in the outward direction may be set up. As the DAG undergoes modification during DAG maintenance, the Destination Advertisement mechanism can be triggered to update the outward routing state.

Arbitrary P2P traffic MAY flow inward along the DAG until a common parent is reached who has stored routing state and is capable of directing the traffic outward along the correct outward path. In the present specification RPL does not specify nor preclude any additional mechanisms that may be capable to compute and install more optimal routes into LLN nodes in support of arbitrary P2P traffic. (Note that in some application scenarios it may be important to support arbitrary P2P traffic along more optimal paths `across' the DAG). This functionality is to be investigated further in a future revision.

This section further describes the high level operation of RPL.

### 3.3.1. DAG Construction

TOC

#### 3.3.1.1. Overview of a Typical Case

TOC

RPL constructs one or more base routing topologies, in the form of DAGs, over gradients defined by optimizing cost metrics along paths rooted at designated nodes.
DAGs may be grounded, in which case the DAG Root is offering a default route. A typical goal for a node participating in DAG Construction will be to find and join a grounded DAG.
In the context of a particular LLN application one or more nodes will be capable of offering a default route and thus be provisioned to act as DAG roots. These nodes will begin the process of constructing a grounded DAG by occasionally emitting Router Advertisements containing the necessary information for neighboring nodes to evaluate the DAG Root as a potential DAG parent. This information will include a DAGID and an Objective Code Point (OCP). The OCP provides information as to which metrics and optimization goals are being employed across the DAG. Note that a single DAG Root may conceptually root different DAGs with different OCPs as required to support different sets of routing constraints. Note that if multiple DAG roots are rooting the same DAG, i.e. presenting the same DAGID, then they must have some means of coordinating with each other when emitting Router Advertisements. This is described further below.
Nodes who hear Router Advertisements, advertising a specific DAGID and OCP, will take into consideration several criteria when processing the extracted DAG information. A node may seek a DAG advertising a specific OCP, reflecting the implementation specific routing constraints understood by the node. In particular, a node will be seeking to find a least cost path satisfying some objective function as indicated by the OCP according to some routing metrics defined in [I-D.ietf-roll-routing-metrics] (Vasseur, J., Kim, M., Networks, D., and H. Chong, “Routing Metrics used for Path Calculation in Low Power and Lossy Networks,” April 2010.). For example, the least cost path may be determined in part by minimizing energy along a path, or latency, or avoiding the use of battery powered nodes. Based on the evaluation of such criteria, a node may determine if the node who emitted the Router Advertisement should be considered as a potential DAG parent. If so, then the node may add the advertising node to its set of DAG parents for the advertised DAGID, and can be considered to have joined the DAG designated by DAGID.
When a node adds the first DAG parent to the set of DAG parents for a particular DAGID, the node is said to have joined, or attached to, the DAG designated by DAGID. Adding additional DAG parents beyond the first

simply increases path diversity inwards toward the DAG root. When a node removes the last DAG Parent from the set of DAG parents for a particular DAGID, the node is said to have left, or detached from, the DAG designated by DAGID. RPL will coordinate the joining, leaving, and movement of nodes within a DAGID in such a way so as to avoid the formation of loops, as described further below.

As nodes join the DAG they are able advertise the fact by beginning to multicast the DAG information in Router Advertisements. In this way, nodes are able to join the DAG at ever-increasing depths outward from the DAG root. As nodes continue to receive DAG multicasts they may continue to expand their set of DAG parents, while employing loop avoidance strategies as describe below, in order to build path diversity inwards toward the DAG root.

Using the information conveyed in the metrics of its most preferred DAG parent, its own metrics, and the conventions and functions indicated by the OCP, a node is able to compute a depth value within the DAG which it will use to coordinate its DAG maintenance.

In addition to identifying DAG parents, a node also may hear the Router Advertisements of other neighboring nodes at the same depth within the DAG. In this way a node can discover DAG Siblings.

A node may order its set of DAG parents according to some implementation specific preference. To this list the node may also append a similarly ordered set of DAG siblings. By forwarding traffic intended for the default destination towards the DAG parents, the node is able to support the main Multipoint-to-point (MP2P) traffic flows required by a typical LLN application. By using the ordered set of DAG parents and DAG siblings the node is able to take advantage of path diversity. For example, preferring to forward traffic towards parents guarantees to get the traffic inwards, closer to the DAG root, by definition, regardless of which parent is selected. In this example, if forwarding towards parents is not possible, perhaps due to a transient phenomena, then a node may then choose to forward traffic towards siblings, moving across the DAG at the same level (neither inwards or outwards). When receiving traffic forwarded from a sibling, the traffic should not be forwarded back to the same sibling in order to avoid a 2-node loop. In a further example, a forwarding implementation may choose to decrease the hop limit more quickly when forwarding along sibling paths than along parent paths. A forwarding engine may behave in a manner similar to these examples, however the specific implementation of a forwarding engine and related path diversity strategies is beyond the scope of this specification.

Note that the further interaction of the routing solution and the forwarding engine, in particular how they utilize and react to changes in metrics, and how the forwarding engine may use the constrained set of successors provided by the routing engine based on L2 triggers and metrics, is under investigation.

By employing this procedure, the LLN is able to set up a path-constrained DAG, rooted at designated nodes, with other nodes organized along paths leading inward toward the DAG root. MP2P traffic intended

for the default destination flows inward along the DAG towards the root, and nodes forwarding traffic are able to leverage the path diversity of the DAG as necessary.
The DAG is then used by RPL as a reference topology, constraining the LLN routing problem, on which to build additional routing mechanisms.

---

### 3.3.1.2. Further Operation

TOC

The sub-DAG of a node is the set of other nodes of greater depth in the DAG that might use a path towards the DAG root that contains this node. Depth in the DAG is defined such that nodes contained in the sub-DAG of a specific node should tend to have a greater depth than the node. Paths through siblings are not contained in this set.
As a further illustration, consider the DAG examples in Appendix B (Examples). Consider Node (24) in the DAG Example depicted in Figure 12 (Example DAG). In this example, the sub-DAG of Node (24) is comprised of Nodes (34), (44), and (45).
A DAG may also be floating, in which case the node rooting the DAG is not offering a default route. Floating DAGs may be encountered, for example, during coordinated reconfigurations of the network topology wherein a node and its sub-DAG breaks off the DAG, temporarily becomes a floating DAG, and reattaches to a grounded DAG at a different (more optimal) location. (Such coordination endeavors to avoid the construction of transient loops in the LLN). A DAG, or a sub-DAG, may also become floating because of a network element failure.
A node will generally join at least one DAG, typically (but not necessarily) to or through a LBR. This specification does not preclude a node from joining multiple DAGs. In one such case, a particular application may require the node to maintain membership in multiple DAGs in order to satisfy competing constraints, for example to support different types of traffic, similar to the concept of MTR (Multi-topology routing) as supported by other routing protocols such as IS-IS [RFC5120] (Przygienda, T., Shen, N., and N. Sheth, “M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs),” February 2008.) or OSPF [RFC4915] (Psenak, P., Mirtorabi, S., Roy, A., Nguyen, L., and P. Pillay-Esnault, “Multi-Topology (MT) Routing in OSPF,” June 2007.), although the RPL mechanisms will significantly differ from the ones specified for these protocols. (Note that not all constrained traffic cases may require multiple DAGs). In support of such cases the RPL implementation must independently maintain requisite information and state for each DAG in parallel. In cases where a competing constraints must be satisfied toward the same DAG root, the OCP should differ by definition and may serve to coordinate the maintenance of the multiple DAGs.

---

#### 3.3.1.3.  Router Advertisement - DAG Information Option (RA-DIO)

TOC

The IPv6 Router Advertisement mechanism (as specified in [RFC4861] (Narten, T., Nordmark, E., Simpson, W., and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” September 2007.)) is used by RPL in order to build and maintain a DAG.
The IPv6 Router Advertisement message is augmented with a DAG Information Option (DIO) in order to facilitate the formation and maintenance of DAGs. The information conveyed in the DIO includes the following:

- A DAGID used to identify the DAG as sourced from the DAG Root. Typically the (potentially compressed) IPv6 address of the DAG Root. May be tested for equality.

- Objective Code Point (OCP) as described below.

- Depth information used by nodes to determine their relationships in the DAG relative to each other, i.e. parents, siblings, or children. This is not a metric, although its derivation is typically closely related to one or more metrics as specified by the OCP. Used to support loop avoidance strategies and in support of ordering alternate successors when engaged in path maintenance.

- Sequence number originated from the DAG root, used to aid in identification of dependent sub-DAGs and coordinate topology changes in a manner so as to avoid loops.

- Indications for the DAG, e.g. grounded or floating.

- DAG configuration parameters

- A vector of path metrics. As discussed in [I-D.ietf-roll-routing-metrics] (Vasseur, J., Kim, M., Networks, D., and H. Chong, “Routing Metrics used for Path Calculation in Low Power and Lossy Networks,” April 2010.) such metrics may be cumulative, may report a maximum, minimum, or average scalar value, or a link property.

- List of additional destinations prefixes reachable via the DAG root.

The Router Advertisements are issued whenever a change is detected to the DAG such that a node is able to determine that a region of the DAG has become inconsistent. As the DAG stabilizes the period at which Router Advertisements occur is configured to taper off, reducing the steady-state overhead of DAG maintenance. The periodic issue of Router

Advertisements, along with the triggered Router Advertisements in response to inconsistency, is one feature that enables RPL to operate in the presence of unreliable links.
The RA-DIO and related mechanisms are described in more detail in Section 5 (Specification of Core Protocol).

#### 3.3.1.4. Objective Code Point (OCP)

TOC

The OCP is seeded by the DAG Root and serves to convey and control the optimization functions used within the DAG. The OCP is envisaged to serve as an reference into a TBA Registry. Each instance of an allocated OCP MUST indicate:

*The set of metrics used within the DAG

*The objective functions used to determine the least cost constrained paths in order to optimize the DAG

*The function used to compute DAG Depth

*The functions used to construct derived metrics for propagation within a DIO

For example, and objective code point might indicate that the DAG is using ETX, that the optimization goal is to minimize ETX, that DAG Depth is equivalent to ETX, and that DIO propagation entails adding the advertised ETX of the most preferred parent to the ETX of the link to the most preferred parent.
By using defined OCPs that are understood by all nodes in a particular implementation, and by conveying them in the DIO, RPL nodes may work to build optimized LLN using a variety of application and implementation specific metrics and goals.
NOTE: A NULL OCP MUST be specified with a well-defined default behavior. The NULL code point will subsequently be used to define RPL behaviors in the case where a node encounters a DIO containing a code point that it does not support.

#### 3.3.1.5. Selection of Feasible DAG Parents

TOC

The decision for a node to join a DAG may be optimized according to implementation specific policy functions on the node as indicated by one or more specific OCP values. For example, a node may be configured for one goal to optimize a bandwidth metric (OCP-1), and with a parallel goal to optimize for a reliability metric (OCP-2). Thus two

DAGs in parallel may be constructed and maintained in the LLN, DAG-1 would be optimized according to OCP-1, whereas DAG-2 would be optimized according to OCP-2. A node may then maintain two parallel sets of DAG parents. Note that in such a case traffic may directed along the appropriate constrained DAG based on traffic marking within the IPv6 header.
As a node hears RAs from its neighbors it may process their DIOs. At this time the node may be able to take into consideration, for example, the following:

* Is the neighboring node heard reliably enough, and are the metrics stable enough, that a local degree of confidence may be established with respect to the neighboring node? Should the neighboring node be considered in the set of candidate neighbors?

* In consultation with implementation specific policy (OCP goal), is the neighboring node a feasible parent from a constrained-path perspective?

* According to the implementation specific policy (OCP), does the neighboring node offer a better optimized position into the DAG?

* Is the neighboring node a peer (sibling) within the DAG?

Based on such considerations, the node may incorporate the neighboring node into the set of DAG parents.
When the node inserts the first DAG parent into the empty DAG parent set, it is able to join the DAG. After the DAG parent set is updated, the node will consult a depth computation function indicated by the OCP for the DAG in order to determine its depth value, which it will subsequently advertise when it emits its own DIOs. A general property of the depth value presented by the node is that it should be greater than that presented by any of its DAG parents. It is recommended that a node maintain its DAG Parent set such that its most preferred parent from the OCP goals also has the greatest depth value in the DAG parent set. All reliable neighboring nodes of a lesser depth then the node may then be considered as potential DAG parents. All neighboring nodes of equal depth may come to be considered as siblings within the DAG (even though they may not have parents in common, they may still provide path diversity towards the DAG root).
The computation of depth, and related properties, are further described in Section 3.4.1 (DAG Depth and Loop Avoidance).

---

#### 3.3.1.5.1. Example

TOC

For example, suppose that a node (N) is not attached to any DAG, and that it is in range of nodes (A), (B), (C), (D), and (E). Let all nodes be configured to use an OCP which defines a policy such that ETX is to be minimized and paths with the attribute `Blue' should be avoided. Let the depth computation indicated by the OCP simply reflect the ETX aggregated along the path. Let the links between node (N) and its neighbors (A-E) all have an ETX of 1 (which is learned by node (N) through some implementation specific method). Let node (N) be configured to send Router Solicitations to probe for nearby DAGs.

- Node (N) transmits a Router Solicitation.
- Node (B) responds. Node (N) investigates the DIO, and learns that Node (B) is a member of DAGID 1 at depth 4, and not `Blue'. Node (N) takes note of this, but is not yet confident.
- Similarly, Node (N) hears from Node (A) at depth 9, Node (C) at depth 5, and Node (E) at depth 4.
- Node (D) responds. Node (D) has a DIO that indicates that it is a member of DAGID 1 at depth 2, but it carries the attribute `Blue'. Node (N)'s policy function rejects Node (D), and no further consideration is given.
- This process continues until Node (N), based on implementation specific policy, builds up enough confidence to trigger a decision to join DAGID 1. Let Node (N) determine its most preferred parent to be Node (E).
- Node (N) adds Node (E) (depth 4) to its set of DAG Parents for DAGID 1. Following the mechanisms specified by the OCP, and given that the ETX is 1 for the link between (N) and (E), Node (N) is now at depth 5 in DAGID. 1.
- Node (N) adds Node (B) (depth 4) to its set of DAG Parents for DAGID 1.
- Node (N) is a sibling of Node (C), both are at depth 5.
- Node (N) may now forward traffic intended for the default destination inward along DAGID 1 via nodes (B) and (E). In some cases, e.g. if nodes (B) and (E) are tried and fail, node (N) may also choose to forward traffic to its sibling node (C), without making inward progress but with the intention that node (C) or a following successor can make inward progress.

#### 3.3.1.6. DAG Maintenance

When a node moves within a DAG, the move is defined as updating the set of DAG Parents for a particular DAGID, i.e. adding or deleting DAG Parents. Not all moves entail changes in depth.
A jump in the context of a DAG is attaching to a new DAGID, in such a way that an old DAGID is replaced by the new DAGID. In particular, when an old DAGID is left, all associated parents are no longer feasible, and a new DAGID is joined.
When a node in a DAG follows a DAG parent, it means that the DAG parent has changed its DAGID (e.g. by joining a new DAG) and that the node updates its own DAGID in order to keep the DAG parent.
A frozen sub-DAG is a subset of nodes in the sub-DAG of a node who have been informed of a change to the node, and choose to follow the node in a manner consistent with the change, for example in preparation for a coordinated move. Nodes in the sub-DAG who hear of a change and have other options than to follow the node do not have to become part of the frozen sub-DAG, for example such a node may be able to remain attached to the original DAG through a different DAG Parent. A further example may be found in Section 3.4.1.1 (Example).
When the node encounters new candidate neighbors that offer higher positions in the DAG, it may incorporate them directly into its set of DAG parents. In this case the node may update its choice of most preferred parent, discarding a deeper node and possibly causing its own advertised depth to decrease. This case is `moving inwards along the DAG' and does not require any additional coordination for loop avoidance.
If the DAG parent set of the node becomes completely depleted, the node will have detached from the DAG, and will become the root of its own floating DAG (thus establishing the frozen sub-DAG), and then may reattach to the original DAG at a lower point if it is able.
When the node encounters candidate parents that are in a different DAG, and decides to leave the current DAG in order to join the different DAG, it may do so safely without regard to loop avoidance. However, it may not return immediately to the current DAG as such movement may trigger the creation of loops.
When a node, and perhaps a related frozen sub-DAG, jumps to a different DAG, the move is coordinated by a DAG Hop timer. The DAG Hop timer allows the nodes who will attach closer to the sink of the new DAG to `jump' first, and then drag dependent nodes behind them, thus endeavoring to efficiently coordinate the attachment of the frozen sub-DAG into the new DAG. A further illustration of this mechanism may be found in Section 3.4.3 (Merging DAGs).
Section 5 (Specification of Core Protocol) contains more detail on the processes and rules used for DAG discovery and maintenance.
Appendix B (Examples) provides additional examples of DAG discovery and maintenance.

### 3.3.2. Source Routing

TOC

A Source Routing mechanism for RPL is currently under investigation.

---

### 3.3.3. Destination Advertisement

TOC

As RPL constructs DAGs, nodes are able to learn a set of default routes in order to send traffic to the sink. However, this mechanism alone does is not sufficient to support P2MP traffic flowing outward along the DAG from the DAG root toward nodes. A Destination Advertisement mechanism is employed by RPL to build up routing state in support of these outward flows.

---

#### 3.3.3.1. Destination Advertisement Option (DAO)

TOC

A Destination Advertisement Option (DAO) is used to convey the Destination information inward along the DAG toward the DAG root. The information conveyed in the DAO includes the following:

* A lifetime and sequence counter to determine the freshness of the Destination Advertisement.

* Depth information used by nodes to determine how far away the destination (the source of the Destination Advertisement) is

* Prefix information to identify the destination, which may be a prefix, an individual host, or multicast listeners

* Reverse Route information to record the nodes visited (along the outward path) when the intermediate nodes along the path cannot support storing state for Hop-By-Hop routing.

---

#### 3.3.3.2. Destination Advertisement Operation

TOC

As the DAG is constructed and maintained, nodes will emit messages containing Destination Advertisement Options to a subset of their DAG Parents. The selection of this subset is according to an implementation specific policy.

Note that further details of the message and its triggers are still under investigation, including whether or not the DAO should be a IPv6 Hop-By-Hop option or a Neighbor Discovery option.
When a DAO reaches a node capable of storing routing state, the node extracts information from the DAO and updates its local database with a record of the DAO and who it was received from. When the node later propagates DAOs, it selects the best (least depth) information for each destination and conveys this information again in the form of DAOs to a subset of its own DAG parents. At this time the node may perform route aggregation if it is able, thus reducing the overall number of DAOs.
When a DAO reaches a node incapable of storing additional state, the node MUST append its own address to a Reverse Route Stack carried within the DAO. The node then passes the DAO on to one or more of its DAG parents without storing any additional state.
When a node that is capable of storing routing state encounters a DAO with a Reverse Route Stack that has been populated, the node knows that the DAO has traversed a region of nodes that did not record any routing state. The node is able to detach and store the Reverse Route State and associate it with the destination described by the DAO. Subsequently the node may use this information to construct a source route in order to bridge the region of nodes that are unable to support Hop-By-Hop routing to reach the destination.
In this way the Destination Advertisement mechanism is able to provision routing state in support of P2MP traffic flows outward along the DAG, and as according to the available resources in the LLN nodes. Further aggregations of DAOs by destinations are possible in order to support additional scalability.
A further example of the operation of the Destination Advertisement mechanism is available in Appendix B.6 (Destination Advertisement)

## 3.4. Other Considerations

### 3.4.1. DAG Depth and Loop Avoidance

When nodes select DAG Parents, they should select the most preferred parent according to their implementation specific objectives, using the cost metrics conveyed in the DIOs along the DAG in conjunction with the related objective functions as specified by the OCP.
Based on this selection, the metrics conveyed by the most preferred DAG parent, the nodes own metrics and configuration, and a related function defined by the objective code point, a node will be able to compute a value for its depth as a consequence of selecting a most preferred DAG parent.

It is important to note that the DAG Depth is not itself a metric, although its value is derived from and influenced by the use of metrics to select DAG parents and take up a position in the DAG. The computation of the DAG Depth MUST be done in such a way so as to maintain the following properties for any nodes M and N who are neighbors in the LLN:

> For a node N, and its most preferred parent M, DAGDepth(N) > DAGDepth(M) must hold. Further, all parents in the DAG parent set must be of a depth less than or equal to DAGDepth(M). (This mechanism serves to avoid loops in the case where an alternate parent is used- if no alternate parent is deeper than the preferred parent then loops are avoided. The risk of loops occurs when an alternate parent goes deeper, and traffic then makes backwards progress and comes back to the node again).
>
> If DAGDepth(M) < DAGDepth(N), then M is located in a more optimum position than N in the DAG with respect to the metrics and optimizations defined by the objective code point. Node M may safely be a DAG Parent for Node N without risk of creating a loop.
>
> If DAGDepth(M) == DAGDepth(N), then M and N are located positions of relatively the same optimality within the DAG. In some cases, Node M may be used as a successor by Node N, but with related chance of creating a loop that must be detected and broken by some other means.
>
> If DAGDepth(M) > DAGDepth(N), then node M is located in a less optimum position than N in the DAG with respect to the metrics and optimizations defined by the objective code point. Further, Node (M) may in fact be in Node (N)'s sub-DAG. There is no advantage to Node (N) selecting Node (M) as a DAG Parent, and such a selection may create a loop.

For example, the DAG Depth could be computed in such a way so as to closely track ETX when the objective function is to minimize ETX, or latency when the objective function is to minimize latency, or in a more complicated way as appropriate to the objective code point being used within the DAG.
The DAG depth is subsequently used to restrict the options a node has for movement within the DAG and to coordinate movements in order to avoid the creation of loops.
A node may safely move `up' in the DAG, causing its DAG depth to decrease and moving closer to the DAG root without risking the formation of a loop.
A node may not consider to move `down' the DAG, causing its DAG depth to increase and moving further from the DAG root. Such a move will entail moving to a less optimum position in the DAG in all cases, as defined by the objective code point. In the case where a node looses

connectivity to the DAG, it must first leave the DAG before it may then rejoin at a deeper point.
Any neighboring nodes of lesser or equal depth are eligible to be considered as DAG parents.

### 3.4.1.1. Example

TOC

```
            :                      :                      :
            :                      :                      :
           (A)                    (A)                    (A)
            |\                     |                      |
            | `-----.              |                      |
            |        \             |                      |
           (B)       (C)          (B)       (C)          (B)
                      |                      |             \
                      |                      |              `-----.
                      |                      |                     \
                     (D)                    (D)                    (C)
                                                                    |
                                                                    |
                                                                    |
                                                                   (D)

                [1]                    [2]                    [3]
```

**Figure 1: DAG Maintenance**

Consider the example depicted in Figure 1 (DAG Maintenance)-1. In this example, Node (A) is attached to a DAG at some depth d. Node (A) is a DAG Parent of Nodes (B) and (C). Node (C) is a DAG Parent of Node (D). There is also an undirected sibling link between Nodes (B) and (C).
In this example, Node (C) may safely forward to Node (A) without creating a loop. Node (C) may not safely forward to Node (D), contained within it's own sub-DAG, without creating a loop. Node (C) may forward to Node (B) in some cases, e.g. the link (C)->(A) is temporarily unavailable, but with some chance of creating a loop and requiring the intervention of additional mechanisms to detect and break the loop.
Consider the case where Node (C) hears a DIO from a Node (Z) at a lesser depth and superior position in the DAG than node (A). Node (C)

may safely undergo the process to evict node (A) from its DAG Parent set and attach directly to Node (Z) without creating a loop, because its depth will decrease.
Consider the case where the link (C)->(A) becomes nonviable, and node (C) must move to a deeper depth within the DAG:

- Node (C) must first detach from the DAG by removing Node (A) from its DAG Parent set, leaving an empty DAG Parent set. Node (C) becomes the root of its own floating DAG.

- Node (D), hearing a modified RA-DIO from Node (C), follows Node (C) into the floating DAG. This is depicted in Figure 1 (DAG Maintenance)-2. In general, any node with no other options in the sub-DAG of Node (C) will follow Node (C) into the floating DAG, maintaining the structure of the sub-DAG.

- Node (C) hears a RA-DIO from Node (B) and determines it is able to rejoin the grounded DAG by reattaching at a deeper depth to Node (B). Node (C) starts a DAG Hop timer to coordinate this move.

- The timer expires and Node (C) adds Node (B) to its DAG Parent set. Node (C) has now safely moved deeper within the grounded DAG without creating any loops. Node (D), and any other sub-DAG of Node (C), will hear the modified RA-DIO sourced from Node (C) and follow Node (C) in a coordinated manner to reattach to the grounded DAG. The final DAG is depicted in Figure 1 (DAG Maintenance)-3

---

### 3.4.2. DAG Parent Selection, Stability, and Greediness

If a node is greedy and attempts to move deeper in the DAG, beyond its most preferred parent, in order to increase the size of the DAG Parent set, then an instability can result. This is illustrated in Figure 2 (Greedy DAG Parent Selection).
Suppose a node is willing to receive and process a RA-DIOs from a node in its own sub-DAG, and in general a node deeper than it. In such cases a chance exists to create a feedback loop, wherein two or more nodes continue to try and move in the DAG in order to optimize against each other. In some cases this will result in an instability. It is for this reason that RPL recommends that a node MUST NOT receive and process RA-DIOs from deeper nodes. This rule creates an `event horizon', whereby a node cannot be influenced into an instability by the action of nodes that may be in its own sub-DAG.

---

#### 3.4.2.1. Example

TOC

```
        (A)                    (A)                    (A)
         |\                     |\                     |\
         | `-----.              | `-----.              | `-----.
         |        \             |        \             |        \
        (B)       (C)          (B)        \            |        (C)
                                 \        |            |        /
                                  `-----. |            | .-----`
                                         \|            |/
                                         (C)          (B)

             [1]                    [2]                    [3]
```

**Figure 2: Greedy DAG Parent Selection**

Consider the example depicted in Figure 2 (Greedy DAG Parent Selection). A DAG is depicted in 3 different configurations. A usable link between (B) and (C) exists in all 3 configurations. In Figure 2 (Greedy DAG Parent Selection)-1, Node (A) is a DAG Parent for Nodes (B) and (C), and (B)--(C) is a sibling link. In Figure 2 (Greedy DAG Parent Selection)-2, Node (A) is a DAG Parent for Nodes (B) and (C), and Node (B) is also a DAG Parent for Node (C). In Figure 2 (Greedy DAG Parent Selection)-3, Node (A) is a DAG Parent for Nodes (B) and (C), and Node (C) is also a DAG Parent for Node (B).
If a RPL node is too greedy, in that it attempts to optimize for an additional number of parents beyond its preferred parent, then an instability can result. Consider the DAG illustrated in Figure 2 (Greedy DAG Parent Selection)-1. In this example, Nodes (B) and (C) may most prefer Node (A) as a DAG Parent, but are operating under the greedy condition that will try to optimize for 2 parents.

*Let Figure 2 (Greedy DAG Parent Selection)-1 be the initial condition.

*Suppose Node (C) first is able to leave the DAG and rejoin at a lower depth, taking both Nodes (A) and (B) as DAG parents as depicted in Figure 2 (Greedy DAG Parent Selection)-2. Now Node (C) is deeper than both Nodes (A) and (B), and Node (C) is satisfied to have 2 DAG parents.

- *Suppose Node (B), in its greediness, is willing to receive and process a DIO from Node (C) (against the rules of RPL), and then Node (B) leaves the DAG and rejoins at a lower depth, taking both Nodes (A) and (C) as DAG Parents. Now Node (B) is deeper than both Nodes (A) and (C) and is satisfied with 2 DAG parents.

- *Then Node (C) will leave and rejoin deeper, to again get 2 parents

- *Then Node (B) will leave and rejoin deeper, to again get 2 parents

- *...

- *The process will repeat, and the DAG will oscillate between Figure 2 (Greedy DAG Parent Selection)-2 and Figure 2 (Greedy DAG Parent Selection)-3 until the nodes count to infinity and restart the cycle again.

- *This cycle can be averted through mechanisms in RPL:

  - -Nodes (B) and (C) stick at a depth sufficient to attach to their most preferred parent (A) and don't go for any deeper (worse) alternate parents (Nodes are not greedy)

  - -Nodes (B) and (C) don't process DIOs from nodes deeper than themselves (possibly in their own sub-DAGs)

### 3.4.3. Merging DAGs

TOC

The merging of DAGs is coordinated in a way such as to try and merge two DAGs cleanly, preserving as much DAG structure as possible, and in the process effecting a clean merge with minimal likelihood of forming transient loops

#### 3.4.3.1. Example

TOC

```
           :
           :
          (A)        (D)
           |          |
           |          |
           |          |
          (B)        (E)
           |          |
           |          |
           |          |
          (C)        (F)
```

**Figure 3: Merging DAGs**

---

Consider the example depicted in Figure 3 (Merging DAGs). Nodes (A), (B), and (C) are part of some larger grounded DAG, where Node (A) is at a depth of d, Node (B) at d+1, and Node (C) at d+2. The DAG comprised of Nodes (D), (E), and (F) is a floating DAG, with Node (D) as the DAG root. This floating DAG may have been formed, for example, in the absence of a grounded DAG or when Node (D) had to detach from a grounded DAG and (E) and (F) followed. All nodes are using compatible objective code points.
Nodes (D), (E), and (F) would rather join the grounded DAG if they are able than to remain in the floating DAG.
Next, let links (C)--(D) and (A)--(E) become viable. The following sequence of events may then occur in a typical case:

*Node (D) will receive and process a RA-DIO from Node (C) on link (C)--(D). Node (D) will consider Node (C) a candidate neighbor, will note that Node (C) is in a grounded DAG at depth d+2, and will begin the process to join the grounded DAG at depth d+3. Node (D) will start a DAG Hop timer, logically associated with the grounded DAG at Node (C), to coordinate the jump. The DAG Hop timer will have a duration proportional to d+2.

*Similarly, Node (E) will receive and process a RA-DIO from Node (A) on link (A)--(E). Node (E) will consider Node (A) a candidate neighbor, will note that Node (A) is in a grounded DAG at depth d, and will begin the process to join the grounded DAG at depth d+1. Node (E) will start a DAG Hop timer, logically associated with the grounded DAG at Node (A), to coordinate the jump. The DAG Hop timer will have a duration proportional to d.

*Node (F) takes no action, for Node (F) has observed nothing new to act on.

*Node (E)'s DAG Hop timer for the grounded DAG at Node (A) expires first. Node (E), upon the DAG Hop timer expiry, is removes Node (D), thus emptying the DAG parent set for the floating DAG and leaving the floating DAG. Node (E) then jumps to the grounded DAG by entering Node (A) into the set of DAG Parents for the grounded DAG. Node (E) is now in the grounded DAG at depth d+1. Node (E), by jumping into the grounded DAG, has created an inconsistency and will begin to emit RA-DIOs more frequently.

*Node (F) will receive and process a RA-DIO from Node (E). Node (F) will observe that Node (E) has changed its DAGID and will directly follow Node (E) into the grounded DAG. Node (F) is now a member of the grounded DAG at depth d+2. Note that any additional sub-DAG of Node (E) would continue to join into the grounded DAG in this coordinated manner.

*Node (D) will receive a RA-DIO from Node (E). Since Node (E) is now in a different DAG, Node (D) may process the RA-DIO from Node (E). Node (D) will observe that, via node (E), it could attach to the grounded DAG at depth d+2. Node (D) will start another DAG Hop timer, logically associated with the grounded DAG at Node (E), with a duration proportional to d+1. Node (D) now is running two DAG hop timers, one which was started with duration proportional to d+1 and one proportional to d+2.

*Generally, Node (D) will expire the timer associated with the jump to the grounded DAG at node (E) first. Node (D) may then jump to the grounded DAG by entering Node (E) into its DAG Parent set for the grounded DAG. Node (D) is now in the grounded DAG at depth d+2.

*In this way RPL has coordinated a merge between the grounded DAG and the floating DAG, such that the nodes within the two DAGs come together in a generally ordered manner, avoiding the formation of loops in the process.

---

### 3.4.4. Local and Temporary Routing Decision

Although implementation specific, it is worth noting that a node may decide to implement some local routing decision based on some metrics, as observed locally or reported in the DIO. For example, the routing may reflect a set of successors (next-hop), along with various aggregated metrics used to load balance the traffic according to some local policy. Such decisions are local and implementation specific. Routing stability is crucial in a LLN: in the presence of unstable links, the first option consists of removing the link from the DAG and

triggering a DAG recomputation across all of the nodes affected by the removed link. Such a naive approach could unavoidably lead to frequent and undesirable changes of the DAG, routing instability, and high-energy consumption. The alternative approach adopted by RPL relies on the ability to temporarily not use a link toward a successor marked as valid, with no change on the DAG structure. If the link is perceived as non-usable for some period of time (locally configurable), this triggers a DAG recomputation, through the DAG Discovery mechanism further detailed in Section 5.3 (DAG Discovery), after reporting the link failure. Note that this concept may be extended to take into account other link characteristics: for the sake of illustration, a node may decide to send a fixed number of packets to a particular successor (because of limited buffering capability of the successor) before starting to send traffic to another successor.
According to the local policy function, it is possible for the node to order the DAG parent set from `most preferred' to `least preferred'. By constructing such an ordered set, and by appending the set with siblings, the node is able to construct an ordered list of preferred next hops to assist in local and temporary routing decisions. The use of the ordered list by a forwarding engine is loosely constrained, and may take into account the dynamics of the LLN. Further, a forwarding engine implementation may decide to perform load balancing functions using hash-based mechanisms to avoid packet re-ordering. Note however, that specific details of a forwarding engine implementation are beyond the scope of this document.
These decisions may be local and/or temporary with the objective to maintain the DAG shape while preserving routing stability.

---

### 3.4.5. Scalability

TOC

As each node selects DAG Parents according to implementation specific objectives, RPL is able to dynamically partition an LLN network into different regions, each anchored by a DAG root. Multiple DAG roots may be deployed in accordance with an implementation specific policy designed to limit the size of a partition, e.g. for performance or other reasons.
A further example is illustrated in Appendix C (Additional Examples).

---

### 3.4.6. Maintenance of Routing Adjacency

TOC

In order to relieve the LLN of the overhead of periodic keepalives, RPL MAY employ an as-needed mechanism of NS/NA in order to verify routing adjacencies just prior to forwarding data. Pending the outcome of verifying the routing adjacency, the packet may either be forwarded or an alternate next-hop may be selected.

## 4. Constraint Based Routing in LLNs

TOC

This aim of this section is to make a clear distinction between routing metrics and constraints and define the term constraint based routing as used in this document.

### 4.1. Routing Metrics

TOC

Routing metrics are used by the routing protocol to compute the shortest path according to one of more defined metrics. IGPs such as IS-IS ([RFC5120] (Przygienda, T., Shen, N., and N. Sheth, “M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs),” February 2008.)) and OSPF ([RFC4915] (Psenak, P., Mirtorabi, S., Roy, A., Nguyen, L., and P. Pillay-Esnault, “Multi-Topology (MT) Routing in OSPF,” June 2007.)) compute the shortest path according to a Link State Data Base (LSDB) using link metrics configured by the network administrator. Such metrics can represent the link bandwidth (in which case the metric is usually inversely proportional to the bandwidth), delay, etc. Note that in some cases the metric is a polynomial function of several metrics defining different link characteristics. The resulting shortest path cost is equal to the sum (or multiplication) of the link metrics along the path: such metrics are said to be additive or multiplicative metrics.
Some routing protocols support more than one metric: in the vast majority of the cases, one metric is used per (sub)topology. Less often, a second metric may be used as a tie breaker in the presence of ECMP (Equal Cost Multiple Paths). The optimization of multiple metrics is known as an NP complete problem and is sometimes supported by some centralized path computation engine.
In the case of RPL, it is virtually impossible to define *the* metric, or even a composite, that will fit it all:

- Some information apply to path setup time, other apply to packet forwarding time.

- Some values are aggregated hop-by-hop, others are triggers from L2.

- Some properties are very stable, others vary rapidly.

- Some data are useful in a given scenario and useless in another.

*Some arguments are scalar, others statistical.

For that reason, the RPL protocol core is agnostic to the logic that handles metrics. A node will be configured with some external logic to use and prioritize certain metrics for a specific scenario. As new heterogeneous devices are installed to support the evolution of a network, or as networks form in a totally ad-hoc fashion, it will happen that nodes that are programmed with antagonistic logics and conflicting or orthogonal priorities end up participating in the same network. It is thus RECOMMENDED to use consistent parent selection policy, as per Objective Code Points (OCP), to ensure consistent optimized paths.
RPL is designed to survive and still operate, though in a somewhat degraded fashion, when confronted to such heterogeneity. The key design point is that each node is solely responsible for setting the vector of metrics that it sources in the DAG, derived in part from the metrics sourced from its preferred parent. As a result, the DAG is not broken if another node makes its decisions in as antagonistic fashion, though an end-to-end path might not fully achieve any of the optimizations that nodes along the way expect. The to-be-defined NULL OCP and related behaviors will further clarify this point.

## 4.2. Routing Constraints

TOC

A constraint is a link or a node characteristic that must be satisfied by the computed path (using boolean values or lower/upper bounds) and is by definition neither additive nor multiplicative. Examples of links constraints are "available bandwidth", "administrative values (e.g. link coloring)", "protected versus non-protected links", "link quality" whereas a node constraint can be the level of battery power, CPU processing power, etc.

## 4.3. Constraint Based Routing

TOC

The notion of constraint based routing consists of finding the shortest path according to some metrics satisfying a set of constraints. A technique consists of first filtering out all links and nodes that cannot satisfy the constraints (resulting in a sub-topology) and then computing the shortest path.

> Example 1:
>
> > Link Metric:     Bandwidth

> > Link Constraint: Blue
> >
> > Node Constraint: Mains-powered node
>
> Objective function 1:
>
> > "Find the shortest path (path with lowest cost where the path cost is the sum of all link costs (Bandwidth)) along the path such that all links are colored `Blue' and that only traverses Mains-powered nodes."
>
> Example 2:
>
> > Link Metric:     Delay
> >
> > Link Constraint: Bandwidth
>
> Objective function 2:
>
> > "Find the shortest path (path with lowest cost where the path cost is the sum of all link costs (Delay)) along the path such that all links provide at least X Bit/s of reservable bandwidth."

---

## 5. Specification of Core Protocol

TOC

---

### 5.1. DAG Information Option

TOC

The DAG Information Option carries a number of metrics and other information that allows a node to discover a DAG, select its DAG parents, and identify its siblings while employing loop avoidance strategies.

---

#### 5.1.1. DIO base option

TOC

The DAG Information Option is a container option, which might contain a number of suboptions. The base option regroups the minimum information set that is mandatory in all cases.

---

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |     Type      |     Length    |G|D| Reserved  |   Sequence    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    | DAGPreference |                BootTimeRandom                 |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |   NodePref.   |   DAGDepth    |           DAGDelay            |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    | DIOIntDoubl.  |  DIOIntMin.   |     DAGObjectiveCodePoint     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                          PathDigest                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                                                               |
    +                                                               +
    |                            DAGID                              |
    +                                                               +
    |                                                               |
    +                                                               +
    |                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |   sub-option(s)...
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Figure 4: DIO Base Option**

**Type:** 8-bit unsigned identifying the DIO base option. The value is to be assigned by the IANA.

**Length:** 8-bit unsigned integer set to 4 when there is no suboption. The length of the option (including the type and length fields and the suboptions) in units of 8 octets.

**Grounded (G):** The Grounded (G) flag is set when the DAG root is offering a default route.

**Destination Advertisement (D):** The Destination Advertisement (D) flag is set when the DAG root or another node in the successor chain decides to trigger the sending of Destination Advertisements in order to update routing state for the outward direction along the DAG, as further detailed in Section 5.4 (Establishing Routing State Outward Along the DAG). Note that the use and semantics of this flag are still under investigation.

**Reserved:** 6-bit unsigned integer set to 0 by the DAG root and left unchanged by nodes propagating the DIO.

**Sequence Number:**
8-bit unsigned integer set by the DAG root, incremented with each new DIO it sends on a link, and propagated with no change outwards along the DAG.

**DAGPreference:** 8-bit unsigned integer set by the DAG root to its preference and unchanged at propagation. Default is 0 (lowest preference). The DAG preference provides an administrative mechanism to engineer the self-organization of the LLN, for example indicating the most preferred LBR.

**BootTimeRandom:** A random value computed at boot time and recomputed in case of a duplication with another node. The concatenation of the NodePreference and the BootTimeRandom is a 32-bit extended preference that is used to resolve collisions. It is set by each node at propagation time.

**NodePreference:** The administrative preference of that LLN Node. Default is 0. 255 is the highest possible preference. Set by each LLN Node at propagation time. Forms a collision tiebreaker in combination with BootTimeRandom.

**DAGDepth:** 8-bit unsigned integer. The DAG depth of the DAG root is 0. The DAG Depth of a node attached to the DAG should be greater than depth of its deepest DAG parent, as computed by an implementation specific routine. All nodes in the DAG advertise their DAG depth in the DAG Information Options that they append to the RA messages over their LLN interfaces as part of the propagation process.

**DAGDelay:** 16-bit unsigned integer set by the DAG root indicating the delay before changing the DAG configuration, in TBD-units. A default value is TBD. It is expected to be an order of magnitude smaller than the RA-interval. It is also expected to be an order of magnitude longer than the typical propagation delay inside the LLN.

**DIOIntervalDoublings:** 8-bit unsigned integer. Used to configure the trickle timer governing when RA-DIO should be sent within the DAG. DIOIntervalDoublings is the number of times that the DIOIntervalMin is allowed to be doubled during the trickle timer operation, i.e. DIOIntervalMax = DIOIntervalMin * 2^(DIOIntervalDoublings).

**DIOIntervalMin:** 8-bit unsigned integer. Used to configure the trickle timer governing when RA-DIO should be sent within the DAG. The minimum configured interval for the RA-DIO trickle timer in units of ms is 2^DIOIntervalMin. For example, a DIOIntervalMin value of 16ms is expressed as 4.

**DAGObjectiveCodePoint:**
The DAG Objective Code Point is used to indicate the cost metrics, objective functions, and methods of computation and comparison for DAGDepth in use in the DAG. The DAG OCP is set by the DAG Root. (Note: this specification recommends that another document, e.g. [I-D.ietf-roll-routing-metrics] (Vasseur, J., Kim, M., Networks, D., and H. Chong, "Routing Metrics used for Path Calculation in Low Power and Lossy Networks," April 2010.), define Objective Code Points and recommend a registry to manage them)

**PathDigest:** 32-bit unsigned integer CRC, updated by each LLN Node. This is the result of a CRC-32c computation on a bit string obtained by appending the received value and the ordered set of DAG parents at the LLN Node. DAG roots use a 'previous value' of zeroes to initially set the PathDigest. Used to determine when something in the set of successor paths has changed.

**DAGID:** 128-bit unsigned integer which uniquely identify a DAG. This value is set by the DAG root. The global IPv6 address of the DAG root can be used.

The following values MUST NOT change during the propagation of the DIO outwards along the DAG: Type, Length, G, DAGPreference, DAGDelay and DAGID. All other fields of the DIO are updated at each hop of the propagation.

---

### 5.1.1.1. DIO suboptions

TOC

In addition to the minimum options presented in the base option, a number of suboptions are defined for the DIO:

---

#### 5.1.1.1.1. Format

TOC

---

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Subopt. Type | Subopt Length | Suboption Data...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Figure 5: DIO Suboption Generic Format**

**Suboption Type:** 8-bit identifier of the type of suboption. When processing a DIO containing a suboption for which the Suboption Type value is not recognized by the receiver, the receiver MUST silently ignore and skip over the suboption, correctly handling any remaining options in the message.

**Suboption Length:** 8-bit unsigned integer, representing the length in octets of the suboption, not including the suboption Type and Length fields.

**Suboption Data:** A variable length field that contains data specific to the option.

The following subsections specify the DIO suboptions which are currently defined for use in the DAG Information Option. Implementations MUST silently ignore any DIO suboptions options that they do not understand.
DIO suboptions may have alignment requirements. Following the convention in IPv6, these options are aligned in a packet such that multi-octet values within the Option Data field of each option fall on natural boundaries (i.e., fields of width n octets are placed at an integer multiple of n octets from the start of the header, for n = 1, 2, 4, or 8).

#### 5.1.1.1.2. Pad1

TOC

The Pad1 suboption does not have any alignment requirements. Its format is as follows:

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|   Type = 0    |
+-+-+-+-+-+-+-+-+
```

**Figure 6: Pad 1**

NOTE! the format of the Pad1 option is a special case - it has neither Option Length nor Option Data fields.

The Pad1 option is used to insert one octet of padding in the DIO to enable suboptions alignment. If more than one octet of padding is required, the PadN option, described next, should be used rather than multiple Pad1 options.

#### 5.1.1.1.3. PadN

TOC

The PadN option does not have any alignment requirements. Its format is as follows:

```
     0                   1
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - -
    |   Type = 1    | Subopt Length | Subopt Data
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - -
```

**Figure 7: Pad N**

The PadN option is used to insert two or more octets of padding in the DIO to enable suboptions alignment. For N (N > 1) octets of padding, the Option Length field contains the value N-2, and the Option Data consists of N-2 zero-valued octets. PadN Option data MUST be ignored by the receiver.

#### 5.1.1.1.4. DAG Metric Container

TOC

The DAG Metric Container suboption may be aligned as necessary to support its contents. Its format is as follows:

```
     0                   1
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - -
    |   Type = 2    | Container Len | DAG Metric Data
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - -
```

**Figure 8: DAG Metric Container**

The DAG Metric Container is used to report aggregated path metrics along the DAG. The DAG Metric Container may contain a number of discrete node, link, and aggregate path metrics as chosen by the implementer. The Container Length field contains the length in octets of the DAG Metric Data. The order, content, and coding of the DAG Metric Container data is as specified in [I-D.ietf-roll-routing-metrics] (Vasseur, J., Kim, M., Networks, D., and H. Chong, “Routing Metrics used for Path Calculation in Low Power and Lossy Networks,” April 2010.).
The processing and propagation of the DAG Metric Container is governed by implementation specific policy functions.

### 5.1.1.1.5. Destination Prefix

TOC

The Destination Prefix suboption has an alignment requirement of 4n+1. Its format is as follows:

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Type = 3    |     Length    | Prefix Length |Resvd|Prf|Resvd|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |             Destination Prefix (Variable Length)              |
 .                                                               .
 .                                                               .
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Figure 9: DAG Destination Prefix**

The Destination Prefix suboption is used when the DAG root needs to indicate that it offers connectivity to destination prefixes other than the default. This may be useful in cases where more than one LBR is operating within the LLN and offering connectivity to different administrative domains, e.g. a home network and a utility network. (Note that a grounded DIO offers the default route without any other qualification needed). In such cases, upon observing the Destination Prefixes offered by a particular DAG root, a node MAY decide to join multiple DAGs in support of a particular application. Note that

Destination Prefixes specified in this manner inherit the Router Lifetime of their parent RA.
The Length is coded as the length of the suboption in octets, excluding the Type and Length fields. The Prefix Length is an 8-bit unsigned integer that indicates the number of leading bits in the destination prefix. Prf is the Route Preference as in [RFC4191] (Draves, R. and D. Thaler, “Default Router Preferences and More-Specific Routes,” November 2005.). The Destination Prefix contains Prefix Length significant bits of the destination prefix. The remaining bits of the Destination Prefix, as required to complete the trailing octet, are set to 0.
In the event that a DAG root may need to specify that it offers connectivity to more than one destination, the Destination Prefix suboption may be repeated.

## 5.2. Neighbor Discovery

TOC

### 5.2.1. RA-DIO Reception

TOC

An node will come to discover its link layer neighbors by a combination of link layer mechanisms and by hearing the multicast RA messages from the neighbors. Through these mechanisms a node is able to construct a set of known neighbors.
When receiving and processing the RA-DIO messages from known neighbors, in addition to link layer states and characteristics, the node will come to determine that a neighbor is of particular interest. As the LLN node periodically observes the neighbor and determines its behavior to be reliable beyond a certain threshold, the node may select the neighbor to be part of the candidate neighbor set and begin to maintain a local confidence value with respect to the neighbor.
As RA-DIOs are received from candidate neighbors, the DIO information will be consulted to determine, for example:

1. Does the candidate neighbor offer a position in a different DAG, or a better position in the current DAG? Is the OCP of the candidate neighbor compatible with the goals of this node? Do the related path metrics pass the criteria of a implementation specific policy function such that the candidate neighbor is considered feasible? If so then consider the candidate neighbor as a candidate parent. The decision to move up the DAG is a policy decision and a node may choose not to move up the DAG if the path metric is not significantly better than the current one.

2. Does the candidate neighbor exist at the same depth in the current DAG as this node? Do the related path metrics pass the criteria of a implementation specific policy function such that the candidate neighbor is feasible? If so then consider the candidate neighbor as a DAG sibling.

3. Otherwise, ignore the candidate neighbor. Ignored neighbors may periodically be re-evaluated to see if their situation has improved.

The implementation SHOULD provide the ability to bound the size of the candidate neighbor set, and a scheme SHOULD be applied to add and/or evict neighbors from the candidate neighbor set as necessary so as not to exceed the bounds.
As candidate parents are identified, they may subsequently be promoted to DAG parents by following the rules of DAG Discovery as described below. When a node adds another node to its set of candidate parents, the node becomes attached to the DAG through the parent node.
In the DAG Discovery implementation, the most preferred parent should be used to restrict which other nodes may become DAG parents. All nodes in the DAG Parent set should be of a depth less than or equal to the most preferred DAG parent.

---

### 5.2.2.  RA-DIO Transmission

Each node maintains a timer that governs when to multicast RAs. This timer is implemented as a trickle timer operating over a variable interval. Trickle timers are further detailed in Section 5.2.3 (Trickle Timer for RA Transmission). The governing parameters for the timer should be configured consistently across the DAG, and are provided by the DAG root in the DIO. In addition to periodic RAs, each LLN node will respond to Router Solicitation messages according to [RFC4861] (Narten, T., Nordmark, E., Simpson, W., and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” September 2007.).

* When a node detects an inconsistency, it may reset the interval of the trickle timer to a minimum value, causing RAs to be emitted more frequently as part of a strategy to quickly correct the inconsistency. Such inconsistencies may be, for example, an update to a key parameter (e.g. sequence number) in the DIO or a point-to-point loop detected when a node located inwards along the DAG forwards traffic intended for the default destination. Inconsistencies are further detailed in Section 5.2.3.2 (Determination of Inconsistency).

* When a node enters a mode of consistent operation within a DAG, it may begin to open up the interval of the trickle timer towards

a maximum value, causing RAs to be emitted less frequently, thus reducing network maintenance overhead and saving energy consumption (which is of utmost importance for battery-operated nodes).

* When a node is initialized, it may choose to remain silent and not multicast any RAs until it has encountered and joined a DAG (perhaps initially probing for a nearby DAG with an RS). Alternately, it may choose to root its own floating DAG and begin multicasting RAs using a default trickle configuration. The second case may be advantageous if it is desired for independent nodes to begin aggregating into scattered floating DAGs in the absence of a grounded node, for example in support of LLN installation and commissioning.

Note that if multiple DAG roots are participating in the same DAG, i.e. offering DIOs with the same DAGID, then they must coordinate with each other to ensure that their DIOs are consistent when they emit RA-DIOs. In particular the Sequence number must be identical from each DAG root, regardless of which of the multiple DAG roots issues the DIO, and changes to the Sequence number should be issued at the same time. The specific mechanism of this coordination is beyond the scope of this specification.

---

### 5.2.3. Trickle Timer for RA Transmission

RPL treats the construction of a DAG as a consistency problem, and uses a trickle timer [Levis08] (Levis, P., Brewer, E., Culler, D., Gay, D., Madden, S., Patel, N., Polastre, J., Shenker, S., Szewczyk, R., and A. Woo, “The Emergence of a Networking Primitive in Wireless Sensor Networks,” July 2008.) to control the rate of control broadcasts. The operation of this timer is in support of the procedures further discussed in Section 5.3 (DAG Discovery)
For each DAG that a node is part of, the node must maintain a single trickle timer. The required state contains the following conceptual items:

**I:** The current length of the communication interval

**T:** A timer with a duration set to a random value in the range [I/2, I]

**C:** Redundancy Counter

**I_min:** The smallest communication interval in milliseconds. This value is learned from the DIO as (2^DIOIntervalMin)ms. The default value is DEFAULT_DIO_INTERVAL_MIN.

**I_doublings:**

> The number of times I_min should be doubled before maintaining a constant rate, i.e. I_max = I_min * 2^I_doublings. This value is learned from the DIO as DIOIntervalDoublings. The default value is DEFAULT_DIO_INTERVAL_DOUBLINGS.

### 5.2.3.1. Resetting the Trickle Timer

TOC

The trickle timer for a DAGID is reset by:

1. Setting I_min and I_doublings to the values learned from the RA-DIO.

2. Setting C to zero.

3. Setting I to I_min.

4. Setting T to a random value as described above.

5. Restarting the trickle timer to expire after a duration T

When an LLN learns about a DAG through a RA and makes the decision to join it, it initializes the state of the trickle timer by resetting the trickle timer and listening. Each time it hears an RA for this DAG, it increments C.
When the timer fires at time T, the node compares C to the redundancy constant, DEFAULT_DIO_REDUNDANCY_CONSTANT. If C is less than that value, the node generates a new RA and broadcasts it. When the communication interval I expires, the node doubles the interval I so long as it has previously doubled it fewer then I_doubling times, resets C, and chooses a new T value.

### 5.2.3.2. Determination of Inconsistency

TOC

The trickle timer is reset whenever an inconsistency is detected within the DAG, for example:

*The node joins a new DAGID

*The node moves within a DAGID

*The node receives a modified DIO from a DAG parent

- *A DAG parent forwards a packet intended for the default route, indicating an inconsistency and possible loop.
- *A metric communicated in the DIO is determined to be inconsistent, as according to a implementation specific path metric selection engine.
- *The depth of a DAG parent has changed.

---

### 5.3. DAG Discovery

TOC

DAG Discovery is a form of distance vector protocol for use in LLNs. DAG Discovery locates the nearest sink and forms a Directed Acyclic Graph towards that sink, by identifying a set of DAG parents. During this process DAG Discovery also identifies siblings, which may be used later to provide additional path diversity towards the DAG root. DAG Discovery enables nodes to implement different policies for selecting their DAG parents in the DAG by using implementation specific policy functions. DAG Discovery specifies a set of rules to be followed by all implementations in order to ensure interoperation. DAG Discovery also standardizes the format that is used to advertise the most common information that is used in order to select DAG parents.
One of these information, the DAG depth, is used by DAG Discovery to provide loop avoidance even if nodes implement different policies. The DAG Depth is computed as specified by the Objective Code Point in use by the DAG, demonstrating the properties described in Section 3.4.1 (DAG Depth and Loop Avoidance). The depth should be computed in such a way so as to provide a comparable basis with other nodes which may not use the same metric at all. (The to-be-defined NULL OCP and related behaviors will clarify this point).
In order to organize and maintain loopless structure, the DAG Discovery implementation in the nodes MUST obey to the following rules and definitions:

1. A node that does not have any DAG parents in a DAG is the root of its own floating DAG. It's depth is 1. A node will end up in that situation when it looses all of its current feasible parents, i.e. the set of DAG parents becomes depleted. In that case, the node SHOULD remember the DAGID and the sequence counter in the DIO of the lost parents for a period of time which covers multiple DIO.

2. A LLN Node that is attached to an infrastructure that does not support DIO, is the DAG root of its own grounded DAG. It's depth is 1.

3. A router sending a RA without DIO is considered a grounded infrastructure at depth 0. (For example, a router that is in communication with an LLN node but not running RPL such as a backbone router in communication with an LBR)

4. The DAG root exposes the DAG in the Router Advertisement DAG Information Option and nodes propagate the DIO outwards along the DAG with the RAs that they forward over their LLN links.

5. A node MAY move at any time, with no delay, within its DAG as long as such a move does not increase its own DAG depth, as per the depth calculation indicated by the OCP. If a node is required to move such that it cannot stay within the DAG without a depth increase, then it needs to first leave the DAG. In other words a A node that is already part of a DAG MAY move or follow a DAG parent at any time and with no delay in order to be closer, or stay as close, to the DAG root of its current DAG as it already is. But a node MUST NOT move outwards along the DAG that it is attached, except in the special case when choosing to follow the last DAG parent in the set of DAG parents. RAs received from other routers located higher in the same DAG may be considered as coming from candidate parents. RAs received from other routers located at the same depth in the same DAG may be considered as coming from siblings. Nodes MUST ignore RAs that are received from other routers located deeper within the same DAG.

6. A node may jump from its current DAG into any different DAG if it is preferred for reasons of connectivity, configured preference, free medium time, size, security, bandwidth, DAG depth, or whatever metrics the LLN cares to use. A node may jump at any time and to whatever depth it reaches in the new DAG, but it may have to wait for a DAG Hop timer to elapse in order to do so. This allows the new higher parts (closer to the sink) of the DAG to move first, thus allowing stepped DAG reconfigurations and limiting relative movements. A node SHOULD NOT join a previous DAG (identified by its DAGID) unless the sequence number in the DIO has incremented since the node left that DAG. A newer sequence number indicates that the candidate parents were not attached behind this node, as they kept getting subsequent DIOs with new sequence numbers from the same DAG. In the event that old sequence numbers (two or more behind the present value) are encountered they are considered stale and the corresponding parent SHOULD be removed from the set.

7. If a node has selected a new set of DAG parents but has not moved yet (because it is waiting for DAG Hop timer to elapse), the node is unstable and refrains from sending Router Advertisement - DAG Information Options.

8. If a node receives a Router Advertisement - DAG Information Option from one of its DAG parents, and if the parent contains a different DAGID, indicating that the parent has left the DAG, and if the node can remain in the current DAG through an alternate DAG parent, then the node should remove the DAG parent which has joined the new DAG from its DAG parent set and remain in the original DAG. If the node was the last DAG parent then the node SHOULD follow that parent.

9. When a node detects or causes a DAG inconsistency, as described in Section 5.2.3.2 (Determination of Inconsistency), then the node sends an unsolicited Router Advertisement message to its one-hop neighbors. The RA contains a DIO that propagates the new DAG information. Such an event will also cause the trickle timer governing the periodic RAs to be reset.

10. If a DAG parent increases its depth such that the node depth would have to change, and if the node does not wish to follow (e.g. it has alternate options), then the DAG parent should be evicted from the DAG parent set. If the DAG parent is the last in the DAG parent set, then the node may chose to follow it.

---

### 5.3.1. DAG Selection

The DAG selection is implementation and algorithm dependent. Nodes SHOULD prefer to join DAGs advertising OCPs compatible with their implementation specific objectives. In order to limit erratic movements, and all metrics being equal, nodes SHOULD keep their previous selection. Also, nodes SHOULD provide a means to filter out a candidate parent whose availability is detected as fluctuating, at least when more stable choices are available. Nodes MAY place the failed candidate parent in a Hold Down mode that ensures that the candidate parent will not be reused for a given period of time.
The known DAGs are associated with the candidate parents that advertise them and kept in a list by extending the Default Router List (DRL). DRL entries are extended to store the information received from the last DIO. The DRL MAY need to be modified in order to keep track of membership to multiple DAGs simultaneously. The DRL entries are managed by states and timers described in the next section.
When connection to a fixed network is not possible or preferable for security or other reasons, scattered DAGs MAY aggregate as much as possible into larger DAGs in order to allow connectivity within the LLN. How to balance these DAGs is implementation dependent, and MAY use a specific visitor-counter suboption in the DIO.

A node SHOULD verify that bidirectional connectivity and adequate link quality is available with a candidate neighbor before it considers that candidate as a DAG parent.

### 5.3.2. Administrative depth

TOC

When the DAG is formed under a common administration, or when a node performs a certain role within a community, it might be beneficial to associate a range of acceptable depth with that node. For instance, a node that has limited battery should be a leaf unless there is no other choice, and may then augment the depth computation specified by the OCP in order to expose an exaggerated depth.

### 5.3.3. DRL entries states and stability

TOC

Candidate parents in the DRL may or may not be usable for forwarding traffic inward along the DAG toward the root depending on runtime conditions. The following states are defined:

**Current** This candidate parent is in the set of DAG parents and may be used for forwarding traffic inward along the DAG.

**Held-Up** This parent can not be used until the DAG hop timer elapses.

**Held-Down** This candidate parent can not be used till hold down timer elapses. At the end of the hold-down period, the candidate is removed from the DRL, and may be reinserted if it appears again with a RA.

**Collision** This candidate parent can not be used till its next RA.

#### 5.3.3.1. Held-Up

TOC

This state is managed by the DAG Hop timer, it serves 2 purposes:

Delay the reattachment of a sub-DAG that has been forced to detach. This is not as safe as the use of the sequence, but still covers that when a sub-DAG has detached, the Router Advertisement - DAG Information Option that is initiated by the new DAG root has a

> chance to spread outward along the sub-DAG so that two different DAGs have formed.
>
> Limit Router Advertisement - DAG Information Option storms when two DAGs collide/merge. The idea is that between the nodes from DAG A that decide to move to DAG B, those that see the highest place (closer to the DAG root) in DAG B will move first and advertise their new locations before other nodes from DAG A actually move.

A new DAG is discovered upon a router advertisement message with or without a Router Advertisement - DAG Information Option. The node joins the DAG by selecting the source of the RA message as a DAG parent (and possible default gateway) and propagating the DIO accordingly.
When a new DAG is discovered, the candidate parent that advertises the new DAG is placed in a held up state for the duration of a DAG Hop timer. If the resulting new set of DAG parents is more preferable than the current one, or if the node is intending to maintain a membership in the new DAG in addition to its current DAG, the node expects to jump and becomes unstable.
A node that is unstable may discover other candidate parents from the same new DAG during the instability phase. It needs to start a new DAG Hop timer for all these. The first timer that elapses for a given new DAG clears them all for that DAG, allowing the node to jump to the highest position available in the new DAG.
The duration of the DAG Hop timer depends on the DAG Delay of the new DAG and on the depth of candidate parent that triggers it: (candidates depth + random) * candidate's DAG_delay (where 0 <= random < 1). It is randomized in order to limit collisions and synchronizations.

---

### 5.3.3.2. Held-Down

TOC

When a neighboring node is 'removed' from the Default Router List, it is actually held down for a hold down timer period, in order to prevent flapping. This happens when a node disappears (upon expiration timer). An node that is held down is not considered for the purpose of forwarding traffic inward along the DAG toward the root. When the hold down timer elapses, the node is removed from the DRL.

---

### 5.3.3.3. Collision

TOC

A race condition occurs if 2 nodes send RA-DIO at the same time and then attempt to join each other. This might happen, for example, between nodes which act as DAG root of their own DAGs. In order to detect the situation, LLN Nodes time stamp the sending of RA-DIO. Any

RA-DIO received within a short link-layer-dependent period introduces a risk. To resolve the collision, a 32bits extended preference is constructed from the DIO by concatenating the NodePreference with the BootTimeRandom.
A node that decides to add a candidate to its DAG parents will do so between (candidate depth) and (candidate depth + 1) times the candidate DAG Delay. But since a node is unstable as soon as it receives the RA-DIO from the desired candidate, it will restrain from sending a RA-DIO between the time it receives the RA and the time it actually jumps. So the crossing of RA may only happen during the propagation time between the candidate and the node, plus some internal queuing and processing time within each machine. It is expected that one DAG delay normally covers that interval, but ultimately it is up to the implementation and the configuration of the candidate parent to define the duration of risk window.
There is risk of a collision when a node receives an RA, for another candidate that is more preferable than the current candidate, within the risk window. In the face of a potential collision, the node with lowest extended preference processes the RA-DIO normally, while the router with the highest extended preference places the other in collision state, does not start the DAG hop timer, and does not become instable. It is expected that next RAs between the two will not cross anyway.

---

#### 5.3.3.4. Instability

TOC

A node is instable when it is prepared to shortly replace a set of DAG parents in order to jump to a different DAGID. This happens typically when the node has selected a more preferred candidate parent in a different DAG and has to wait for the DAG hop timer to elapse before adjusting the DAG parent set. Instability may also occur when the entire current DAG parent set is lost and the next best candidates are still held up. Instability is resolved when the DAG hop timer of all the candidate(s) causing instability elapse. Such candidates then change state to Current or Held- Down.
Instability is transient (in the order of DAG hop timers). When a node is unstable, it MUST NOT send RAs with DIO. This avoids loops when node A decides to attach to node B and node B decides to attach to node A. Unless RAs cross (see Collision section), a node receives DIO from stable candidate parents, which do not plan to attach to the node, so the node can safely attach to them.

---

TOC

## 5.4.  Establishing Routing State Outward Along the DAG

The Destination Advertisement mechanism supports the dissemination of routing state required to support traffic flows outward along the DAG, from the DAG root toward nodes.
Note that some aspects of the Destination Advertisement mechanism are still under investigation.
As a result of Destination Advertisement operation:

- *DAG Discovery establishes a DAG oriented toward a DAG root using extended Neighbor Discovery RS/RA flows, along which inward routes toward the DAG root are set up.

- *Destination Advertisement extends Neighbor Discovery in order to establish outward routes along the DAG, along paths containing DA parents. Such paths consist of:

  - -Hop-By-Hop routing state within islands of `stateful' nodes.

  - -Source Routing `bridges' across nodes who do not retain state.

Destinations disseminated with the Destination Advertisement mechanism may be prefixes, individual hosts, or multicast listeners. The mechanism supports nodes of varying capabilities as follows:

- *When nodes are capable of storing routing state, they may inspect Destination Advertisements and learn hop-by-hop routing state toward destinations. In this process they may also learn necessary piecewise source routes to traverse regions of the LLN that do not maintain routing state. They may perform route aggregation on known destinations before emitting Destination Advertisements.

- *When nodes are incapable of storing routing state, they may forward Destination Advertisements, recording the reverse route as the go in order to support the construction of piecewise source routes.

Nodes that are capable of storing routing state, and finally the DAG roots, are able to learn which destinations are contained in the sub-DAG below the node, and via which next-hop neighbors. The dissemination and installation of this routing state into nodes allows for Hop-By-Hop routing from the DAG root outwards along the DAG. The mechanism is further enhance by supporting the construction of source routes across stateless `gaps' in the DAG, where nodes are incapable of storing additional routing state. An adaptation of this mechanism allows for the implementation of loose-source or landmark (waypoint) routing.
The design choice behind this is not to synchronize the parent and children databases along the DAG, but instead to update them regularly to cover from the loss of packets. The rationale for that choice is

time variations in connectivity across unreliable links. If the topology can be expected to change frequently, synchronization might be an excessive goal in terms of exchanges and protocol complexity. The approach used here results in a simple protocol with no real peering. The Destination Advertisement mechanism hence provides for periodic updates of the derivative routing state, as cued by occasional RAs and other mechanisms.

---

### 5.4.1. Destination Advertisement Message Formats

TOC

---

#### 5.4.1.1. DAO Option

TOC

RPL extends Neighbor Discovery [RFC4861] (Narten, T., Nordmark, E., Simpson, W., and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” September 2007.) and RFC4191 [RFC4191] (Draves, R. and D. Thaler, “Default Router Preferences and More-Specific Routes,” November 2005.) to allow a node to include a Destination Advertisement option, which includes prefix information, in the Neighbor Advertisements (NAs). A prefix option is normally present in Router Advertisements (RAs) only, but the NA is augmented with this option in order to propagate destination information inwards along the DAG. The option is named the Destination Advertisement Option (DAO), and an NA containing this option may be referred to as a Destination Advertisement. The RPL use of Destination Advertisements allows the nodes in the DAG to build up routing state for nodes contained in the sub-DAG in support of traffic flowing outward along the DAG.

---

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     | Prefix Length |    RRCount    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         DAO Lifetime                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Route Tag                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   DAO Depth   |   Reserved    |         DAO Sequence          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Prefix (Variable Length)                   |
.                                                               .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Reverse Route Stack (Variable Length)            |
.                                                               .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Figure 10: Destination Advertisement Option (DAO)**

**Type:** 8-bit unsigned identifying the Destination Advertisement option. The value is to be assigned by the IANA.

**Length:** 8-bit unsigned integer. The length of the option (including the Type and Length fields) in units of 8 octets.

**Prefix Length:** Number of valid leading bits in the IPv6 Prefix.

**RRCount:** 8-bit unsigned integer. This counter is used to count the number of entries in the Reverse Route Stack. A value of `0' indicates that no Reverse Route Stack is present.

**DAO Lifetime:** 32-bit unsigned integer. The length of time in seconds (relative to the time the packet is sent) that the prefix is valid for route determination. A value of all one bits (0xFFFFFFFF) represents infinity. A value of all zero bits (0x00000000) indicates a loss of reachability.

**Route Tag:** 32-bit unsigned integer. The Route Tag may be used to give a priority to prefixes that should be stored. This may be useful in cases where intermediate nodes are capable of storing a limited amount of routing state. The further specification of this field and its use is under investigation.

**DAO Depth:** Set to 0 by the node that owns the prefix and first issues the DAO. Incremented by all LLN nodes that propagate the DAO.

**Reserved:** 8-bit unused field. It MUST be initialized to zero by the sender and MUST be ignored by the receiver.

**DAO Sequence:** Incremented by the node that owns the prefix for each new DAO for that prefix.

**Prefix:** Variable-length field containing an IPv6 address or a prefix of an IPv6 address. The Prefix Length field contains the number of valid leading bits in the prefix. The bits in the prefix after the prefix length (if any) are reserved and MUST be initialized to zero by the sender and ignored by the receiver.

**Reverse Route Stack:** Variable-length field containing a sequence of RRCount (possibly compressed) IPv6 addresses. A node who adds on to the Reverse Route Stack will append to the list and increment the RRCount.

---

### 5.4.2. Destination Advertisement Operation

---

#### 5.4.2.1. Overview

Note that some aspects of the Destination Advertisement mechanism are still under investigation
According to implementation specific policy, a subset or all of the feasible parents in the DAG may be selected to receive prefix information from the Destination Advertisement mechanism. This subset of DAG parents shall be designated the set of DA parents.
RPL takes advantage of the DAG structure and allows a node capable of storing sufficient routing state to autonomously discover the destinations below itself through the operation of the Destination Advertisement mechanism. This allows participating nodes to build up routing state to support traffic flowing outwards along the DAG. Destination Advertisement messages convey the necessary information to learn the destinations.

As Destination Advertisements for particular destinations move inwards along the DAG, a sequence counter is used to guarantee their freshness. The sequence counter is incremented by the source of the DAO (the node that owns the prefix), each time it issues a DAO for its prefix. Nodes who receive the DAO and, if scope allows, will be forwarding a DAO for the unmodified destination inwards along the DAG, will leave the sequence number unchanged. Intermediate nodes will check the sequence counter before processing a DAO, and if the DAO is unchanged (the sequence counter has not changed), then the DAO will be discarded without additional processing. Further, if the DAO appears to be out of synch (the sequence counter is 2 or more behind the present value) then the DAO state is considered to be stale and may be purged, and the DAO is discarded. A depth is also added for tracking purposes; the depth is incremented at each hop as the DAO is propagated up the DAG. Nodes who are storing routing state may use the depth to determine which possible next-hops for the destination are more optimal.

If Destination Advertisements are activated in the DIO as indicated by the `D' bit, the node sends unicast Destination Advertisements to its DA parents, and only accepts unicast Destination Advertisements from any nodes BUT those contained in the DA parent subset.

Every NA to a DA parent MAY contain one or more DAOs. Receiving a DAG Discovery RA-DIO with the `D' Destination Advertisement bit set from a DAG parent stimulates the sending of a delayed Destination Advertisement back, with the collection of all known prefixes (that is the prefixes learned via Destination Advertisements for nodes lower in the DAG, and any connected prefixes). A Destination Advertisement is also sent to a DAG parent once it has been added to the DA parent set after a movement, or when the list of advertised prefixes has changed. Destination Advertisements may also be scheduled for sending when the PathDigest of the DIO has changed, indicating that some aspect of the inwards paths along the DAG has been modified.

Destination Advertisements may advertise positive (prefix is present) or negative (removed) DAOs. A no-DAO is stimulated by the disappearance of a prefix below. This is discovered by timing out after a request (a RA-DIO) or by receiving a no-DAO. A no-DAO is a conveyed as a DAO with a DAO Lifetime of 0.

A node who is capable of recording the state information conveyed in a DAO will do so upon receiving and processing the DAO, thus building up routing state concerning destinations below it in the DAG. If a node capable of recording state information receives a DAO containing a Reverse Route Stack, then the node knows that the DAO has traversed one or more nodes that did not retain any routing state as it traversed the path from the DAO source to the node. The node may then extract the Reverse Route Stack and retain the included state in order to specify Source Routing instructions along the return path towards the destination. The node MUST set the RRCount back to zero and clear the Reverse Route Stack prior to passing the DAO information on.

A node who is unable to record the state information conveyed in the DAO will append the next-hop address to the Reverse Route Stack, increment the RRCount, and then pass the Destination Advertisement on without recording any additional state. In this way the Reverse Route Stack will come to contain a vector of next hops that must be traversed along the reverse path that the DAO has traveled. The vector will be ordered such that the node closest to the destination will appear first in the list. In such cases the node may choose to convey the Destination Advertisement to one or more DAG Parents in order of preference as guided by an implementation specific policy.

In hybrid cases, some nodes along the path a Destination Advertisement follows inward along the DAG may store state and some may not. The Destination Advertisement mechanism allows for the provisioning of routing state such that when a packet is traversing outwards along the DAG, some nodes may be able to directly forward to the next hop, and other nodes may be able to specify a piecewise source route in order to bridge spans of stateless nodes within the path on the way to the desired destination.

In the degenerate case, no node is able to store any routing state as Destination Advertisements pass by, and the DAG sink ends up with DAOs that contain a completely specified route back to the originating node in the form of the inverted Reverse Route Stack.

Information learned through Destination Advertisements can be redistributed in a routing protocol, MANET or IGP. But the MANET or the IGP SHOULD NOT be redistributed into Destination Advertisements. This creates a hierarchy of routing protocols where DA routes stand somewhere between connected and IGP routes.

The Destination Advertisement mechanism requires stateful nodes to maintain lists of known prefixes. A prefix entry contains the following abstract information:

* A reference to the ND entry that was created for the advertising neighbor.

* The IPv6 address and interface for the advertising neighbor.

* The logical equivalent of the full Destination Advertisement information (including the prefixes, depth, and Reverse Route Stack, if any).

* A 'reported' Boolean to keep track whether this prefix was reported already, and to which of the DA parents.

* A counter of retries to count how many RA-DIOs were sent on the interface to the advertising neighbor without reachability confirmation for the prefix.

Note that nodes may receive multiple information from different neighbors for a specific destination, as different paths through the DAG may be propagating information inwards along the DAG for the same destination. A node who is recording routing state will keep track of

the information from each neighbor independently, and when it comes time to propagate the DAO for a particular prefix to the DA parents, then the DAO information will be selected from among the advertising neighbors who offer the least depth to the destination.
The Destination Advertisement mechanism stores the prefix entries in one of 3 abstract lists; the Connected, the Reachable and the Unreachable lists.
The Connected list corresponds to the prefixes owned and managed by the local node.
The Reachable list contains prefixes for which the node keeps receiving DAOs, and for those prefixes which have not yet timed out.
The Unreachable list keeps track of prefixes which are no longer valid and in the process of being destroyed, in order to send no-DAOs to the DA parents.
The Destination Advertisement mechanism requires 2 timers; the DelayNA timer and the DestroyTimer.

* The DelayNA timer is armed upon a stimulation to send a Destination Advertisement (such as a DIO from a DA parent). When the timer is armed, all entries in the Reachable list as well as all entries for Connected list are set to not reported yet for that particular DA parent.

* The DelayNA timer has a duration that is DEF_NA_LATENCY divided by a multiple of the DAG depth. The intention is that nodes located deeper in the DAG should have a shorter DelayNA timer, allowing DAOs a chance to be reported from deeper in the DAG and potentially aggregated by sub-DAGs before propagating further inwards.

* The DestroyTimer is armed when at least one entry has exhausted its retries, which means that a number of RA-DIO were sent toward the reporting neighbor but that the entry was not confirmed with a DAO. When the destroy timer elapses, for all exhausted entries, the associated route is removed, and the entry is scheduled to be destroyed.

* The Destroy timer has a duration of min (MAX_DESTROY_INTERVAL, RA_INTERVAL).

---

#### 5.4.2.2. Unicast Destination Advertisement messages from child to parent

When sending a Destination Advertisement to a DA parent, a LLN Node includes the DAOs about not already reported prefix entries in the Reachable and Connected lists, as well as no-DAOs for all the entries

in the Unreachable list. Depending on its policy and ability to retain routing state, the receiving node SHOULD keep a record of the reported DAO. If the DAO offers the best route to the prefix as determined by policy and other prefix records, the node SHOULD install a route to the prefix in the DAO via the link local address of the reporting neighbor and it SHOULD further propagate the information, either as a DAO or by means of redistribution into a routing protocol.
The RA-DIO from the DAG root is used to synchronize the whole DAG, including the periodic reporting of Destination Advertisements back up the DAG. Its period is expected to vary, depending on the configuration of the trickle timer that governs the RAs.
When a node receives a RA-DIO over an LLN interface from a DA parent, the DelayNA is armed to force a full update.
When the node broadcasts a RA-DIO on an LLN interface, for all entries on that interface:

- If the entry is CONFIRMED, it goes PENDING with the retry count set to 0.

- If the entry is PENDING, the retry count is incremented. If it reaches a maximum threshold, the entry goes ELAPSED If at least one entry is ELAPSED at the end of the process: if the Destroy timer is not running then it is armed with a jitter.

Since the DelayNA has a duration that decreases with the depth, it is expected to receive all DAOs from all children before the timer elapses and the full update is sent to the DA parents.
Once the Destroy timer is elapsed, the prefix entry is scheduled to be destroyed and moved to the Unreachable list if there are any DA parents that need to be informed of the change in status for the prefix, otherwise the prefix entry is cleaned up right away. The prefix entry is removed from the Unreachable list when no more DA parents need to be informed. This condition may be satisfied when a no-DAO is sent to all current DA parents indicating the loss of the prefix, and noting that in some cases parents may have been removed from the set of DA parents.

---

#### 5.4.2.3.  Other events

TOC

Finally, the Destination Advertisement mechanism responds to a series of events, such as:

- Destination Advertisement operation stopped: All entries in the abstract lists are freed. All the routes learned from DAOs are destroyed.

*Interface going down: for all entries in the Reachable list on that interface, the associated route is removed, and the entry is scheduled to be destroyed.

*Loss of routing adjacency: When the routing adjacency for a neighbor is lost, as per the procedures described in Section 5.5 (Maintenance of Routing Adjacency), and if the associated entries are in the Reachable list, the associated routes are removed, and the entries are scheduled to be destroyed.

*Changes to DA parent set: All entries in the Reachable list are set to not 'reported' and DelayNA is armed.

---

#### 5.4.2.4. Aggregation of prefixes by a node

TOC

There may be number of cases where a aggregation may be shared within a platoon of nodes. In such a case, it is possible to use aggregation techniques with Destination Advertisements and improve scalability. For example, consider a platoon formed by firefighters and their commander. Specifically, the commander may be configured as the Destination Advertisement aggregator for a group prefix. At run time, the commander absorbs the individual DAO information received from the platoon members down its sub-DAG and only reports the aggregation up the DAG. This works fine when the whole platoon is attached within the commander's sub-DAG.
Other cases might occur for which additional support is required:

1. The commander is attached within the sub-DAG of one of its platoon members.

2. A platoon member is somewhere else within the DAG.

3. A platoon member is somewhere else in the LLN.

In all those cases, a node situated above the commander in the DAG but not above the platoon member will see the advertisements for the aggregation owned by the commander but not that of the individual platoon member prefix. So it will route all the packets for the platoon member towards the commander, but the commander will have no route to the individual platoon member and will fail to forward.
Additional protocols may be applied beyond the scope of this specification to dynamically elect/provision a commander and platoon in order to provide route summarization for a sub-DAG.

---

#### 5.4.2.5. Default Values

TOC

DEF_NA_LATENCY = To Be Determined
MAX_DESTROY_INTERVAL = To Be Determined

## 5.5. Maintenance of Routing Adjacency

TOC

The selection of successors, along the default paths inward along the DAG, or along the paths learned from Destination Advertisements outward along the DAG, leads to the formation of routing adjacencies that require maintenance.
In IGPs such as OSPF [RFC4915] (Psenak, P., Mirtorabi, S., Roy, A., Nguyen, L., and P. Pillay-Esnault, “Multi-Topology (MT) Routing in OSPF,” June 2007.) or IS-IS [RFC5120] (Przygienda, T., Shen, N., and N. Sheth, “M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs),” February 2008.), the maintenance of a routing adjacency involves the use of Keepalive mechanisms (Hellos) or other protocols such as BFD ([I-D.ietf-bfd-base] (Katz, D. and D. Ward, “Bidirectional Forwarding Detection,” February 2009.)) and MANET Neighborhood Discovery Protocol (NHDP [I-D.ietf-manet-nhdp] (Clausen, T., Dearlove, C., and J. Dean, “MANET Neighborhood Discovery Protocol (NHDP),” July 2009.)). Unfortunately, such an approach is not desirable in constrained environments such as LLN and would lead to excessive control traffic in light of the data traffic with a negative impact on both link loads and nodes resources. Overhead to maintain the routing adjacency should be minimized. Furthermore, it is not always possible to rely on the link or transport layer to provide information of the associated link state. The network layer needs to fall back on its own mechanism.
Thus RPL makes use of a different approach consisting of probing the neighbor using a Neighbor Solicitation message (see [RFC4861] (Narten, T., Nordmark, E., Simpson, W., and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” September 2007.)). The reception of a Neighbor Advertisement (NA) message with the "Solicited Flag" set is used to verify the validity of the routing adjacency. Such mechanism MAY be used prior to sending a data packet. This allows for detecting whether or not the routing adjacency is still valid, and should it not be the case, select another feasible successor to forward the packet.

## 5.6. Expectations of Link Layer Behavior

TOC

This specification does not rely on any particular features of a specific link layer technologies. It is anticipated that an implementer should be able to operate RPL over a variety of different low power wireless or PLC (Power Line Communication) link layer technologies. Implementers may find RFC 3819 (Karn, P., Bormann, C., Fairhurst, G., Grossman, D., Ludwig, R., Mahdavi, J., Montenegro, G., Touch, J., and L. Wood, “Advice for Internet Subnetwork Designers,” July 2004.) [RFC3819] a useful reference when designing a link layer interface between RPL and a particular link layer technology.

## 6. Protocol Extensions

TOC

## 7. Manageability Considerations

TOC

## 8. Security Considerations

TOC

## 9. IANA Considerations

TOC

### 9.1. DAG Information Option

TOC

IANA is requested to allocate a new Neighbor Discovery Option Type from the IPv6 Neighbor Discovery Option Formats Registry in order to represent the DAG Information Option as described in Section 5.1 (DAG Information Option)

### 9.2. Destination Advertisement Option

TOC

IANA is requested to allocate a new Neighbor Discovery Option Type from the IPv6 Neighbor Discovery Option Formats Registry in order to represent the Destination Advertisement Option as described in Section 5.4.1.1 (DAO Option)

## 10. Acknowledgements

TOC

The ROLL Design Team would like to acknowledge the review, feedback, and comments from Dominique Barthel, Yusuf Bashir, Mathilde Durvy, Manhar Goindi, Mukul Goyal, Richard Kelsey, Quentin Lampin, Philip Levis, Jerry Martocci, Alexandru Petrescu, and Don Sturek.
The ROLL Design Team would like to acknowledge the guidance and input provided by the ROLL Chairs, David Culler and JP Vasseur.
The ROLL Design Team would like to acknowledge prior contributions of Richard Kelsey, Robert Assimiti, Mischa Dohler, Julien Abeille, Ryuji Wakikawa, Teco Boot, Patrick Wetterwald, Bryan Mclaughlin, Carlos J. Bernardos, Thomas Watteyne, Zach Shelby, Dominique Barthel, Caroline Bontoux, Marco Molteni, Billy Moon, and Arsalan Tavakoli, which have provided useful design considerations to RPL.

## 11. Contributors

TOC

ROLL Design Team in alphabetical order:

Anders Brandt
Zensys, Inc.
Emdrupvej 26
Copenhagen, DK-2100
Denmark

Email: abr@zen-sys.com

Thomas Heide Clausen
LIX, Ecole Polytechnique, France

Phone: +33 6 6058 9349
EMail: T.Clausen@computer.org
URI:   http://www.ThomasClausen.org/

Stephen Dawson-Haggerty
UC Berkeley
Soda Hall, UC Berkeley
Berkeley, CA  94720
USA

Email: stevedh@cs.berkeley.edu

Jonathan W. Hui
Arch Rock Corporation
501 2nd St. Ste. 410
San Francisco, CA  94107
USA

Email: jhui@archrock.com

Kris Pister
Dust Networks
30695 Huntwood Ave.
Hayward,   94544
USA

Email: kpister@dustnetworks.com

Pascal Thubert
Cisco Systems
Village d'Entreprises Green Side
400, Avenue de Roumanille
Batiment T3

Biot - Sophia Antipolis  06410
FRANCE

Phone: +33 497 23 26 34
Email: pthubert@cisco.com

Tim Winter (editor)

wintert@acm.org

## 12. References

TOC

### 12.1. Normative References

TOC

| | |
|---|---|
| [RFC2119] | Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” BCP 14, RFC 2119, March 1997 (TXT, HTML, XML). |

## 12.2. Informative References

TOC

| | |
|---|---|
| [I-D.ietf-bfd-base] | Katz, D. and D. Ward, “Bidirectional Forwarding Detection,” draft-ietf-bfd-base-09 (work in progress), February 2009 (TXT). |
| [I-D.ietf-manet-nhdp] | Clausen, T., Dearlove, C., and J. Dean, “MANET Neighborhood Discovery Protocol (NHDP),” draft-ietf-manet-nhdp-10 (work in progress), July 2009 (TXT). |
| [I-D.ietf-roll-building-routing-reqs] | Martocci, J., Riou, N., Mil, P., and W. Vermeylen, “Building Automation Routing Requirements in Low Power and Lossy Networks,” draft-ietf-roll-building-routing-reqs-05 (work in progress), February 2009 (TXT). |
| [I-D.ietf-roll-home-routing-reqs] | Porcu, G., “Home Automation Routing Requirements in Low Power and Lossy Networks,” draft-ietf-roll-home-routing-reqs-06 (work in progress), November 2008 (TXT). |
| [I-D.ietf-roll-indus-routing-reqs] | Networks, D., Thubert, P., Dwars, S., and T. Phinney, “Industrial Routing Requirements in Low Power and Lossy Networks,” draft-ietf-roll-indus-routing-reqs-06 (work in progress), June 2009 (TXT). |
| [I-D.ietf-roll-routing-metrics] | Vasseur, J., Kim, M., Networks, D., and H. Chong, “Routing Metrics used for Path Calculation in Low Power and Lossy Networks,” draft-ietf-roll-routing-metrics-06 (work in progress), April 2010 (TXT). |
| [I-D.ietf-roll-terminology] | Vasseur, J., “Terminology in Low power And Lossy Networks,” draft-ietf-roll-terminology-03 (work in progress), March 2010 (TXT). |
| [I-D.tavakoli-hydro] | Tavakoli, A., Dawson-Haggerty, S., Hui, J., and D. Culler, “HYDRO: A Hybrid Routing Protocol for Lossy and Low Power Networks,” draft-tavakoli-hydro-01 (work in progress), March 2009 (TXT). |
| [I-D.thubert-roll-fundamentals] | Thubert, P., Watteyne, T., Shelby, Z., and D. Barthel, “LLN Routing Fundamentals,” draft-thubert-roll-fundamentals-01 (work in progress), April 2009 (TXT). |
| [I-D.tsao-roll-security-framework] | Tsao, T., Alexander, R., Daza, V., and A. Lozano, “A Security Framework for Routing over Low Power and Lossy Networks,” draft-tsao-roll-security-framework-02 (work in progress), March 2010 (TXT). |
| [Levis08] | Levis, P., Brewer, E., Culler, D., Gay, D., Madden, S., Patel, N., Polastre, J., Shenker, S., Szewczyk, R., and A. Woo, “The Emergence of a Networking Primitive in Wireless Sensor Networks,” |

| | |
|---|---|
| | Communications of the ACM, v.51 n.7, July 2008 (HTML). |
| [RFC3819] | Karn, P., Bormann, C., Fairhurst, G., Grossman, D., Ludwig, R., Mahdavi, J., Montenegro, G., Touch, J., and L. Wood, “Advice for Internet Subnetwork Designers,” BCP 89, RFC 3819, July 2004 (TXT). |
| [RFC4101] | Rescorla, E. and IAB, “Writing Protocol Models,” RFC 4101, June 2005 (TXT). |
| [RFC4191] | Draves, R. and D. Thaler, “Default Router Preferences and More-Specific Routes,” RFC 4191, November 2005 (TXT). |
| [RFC4461] | Yasukawa, S., “Signaling Requirements for Point-to-Multipoint Traffic-Engineered MPLS Label Switched Paths (LSPs),” RFC 4461, April 2006 (TXT). |
| [RFC4861] | Narten, T., Nordmark, E., Simpson, W., and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” RFC 4861, September 2007 (TXT). |
| [RFC4875] | Aggarwal, R., Papadimitriou, D., and S. Yasukawa, “Extensions to Resource Reservation Protocol - Traffic Engineering (RSVP-TE) for Point-to-Multipoint TE Label Switched Paths (LSPs),” RFC 4875, May 2007 (TXT). |
| [RFC4915] | Psenak, P., Mirtorabi, S., Roy, A., Nguyen, L., and P. Pillay-Esnault, “Multi-Topology (MT) Routing in OSPF,” RFC 4915, June 2007 (TXT). |
| [RFC5120] | Przygienda, T., Shen, N., and N. Sheth, “M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs),” RFC 5120, February 2008 (TXT). |
| [RFC5548] | Dohler, M., Watteyne, T., Winter, T., and D. Barthel, “Routing Requirements for Urban Low-Power and Lossy Networks,” RFC 5548, May 2009 (TXT). |

## Appendix A. Deferred Requirements

TOC

NOTE: RPL is still a work in progress. At this time there remain many unsatisfied application requirements, but these are to be addressed as RPL is further specified.

TOC

## Appendix B.  Examples

Consider the example LLN physical topology in Figure 11 (Example LLN Topology). In this example the links depicted are all usable L2 links. Suppose that all links are equally usable, and that the implementation specific policy function is simply to minimize hops. This LLN physical topology then yields the DAG depicted in Figure 12 (Example DAG), where the links depicted are the edges toward DAG parents. This topology includes one DAG, rooted by an LBR node (LBR) at depth 1. The LBR node will issue RAs containing DIO, as governed by a trickle timer. Nodes (11), (12), (13), have selected (LBR) as their only parent, attached to the DAG at depth 2, and periodically advertise RA-DIO multicasts. Node (22) has selected (11) and (12) in its DAG parent set, and advertises itself at depth 3. Node (22) thus has a set of DAG parents {(11), (12)} and siblings {((21), (23)}.

---

```
                                   (LBR)
                                   / | \
                              .---`  |  `----.
                             /       |        \
                          (11)------(12)------(13)
                           | \       | \       | \
                           |  `----. |  `----. |  `----.
                           |        \|        \|        \
                          (21)------(22)------(23)      (24)
                           |        /|        /|         |
                           |  .----` |  .----` |         |
                           | /       | /       |         |
                          (31)------(32)------(33)------(34)
                           |        /| \       | \       | \
                           |  .----` |  `----. |  `----. |  `----.
                           | /       |        \|        \|        \
                 .--------(41)      (42)      (43)------(44)------(45)
                /         /        /| \       | \
          .----`    .----`    .----` |  `----. |  `----.
         /         /         /       |        \|        \
      (51)------(52)------(53)------(54)------(55)------(56)
```

Note that the links depicted represent the usable L2 connectivity available in the LLN. For example, Node (31) can communicate directly with its neighbors, Nodes (21), (22), (32), and (41). Node (31) cannot communicate directly with any other nodes, e.g. (33), (23), (42). In this example these links offer bidirectional communication, and `bad' links are not depicted.

**Figure 11: Example LLN Topology**

```
                                       (LBR)
                                       / | \
                                  .---`  |  `----.
                                 /       |        \
                              (11)      (12)      (13)
                               | \       | \       | \
                               |  `----. |  `----. |  `----.
                               |        \|        \|        \
                              (21)      (22)      (23)      (24)
                               |        /|        /|         |
                               |  .----` |  .----` |         |
                               | /       | /       |         |
                              (31)      (32)      (33)      (34)
                               |        /| \       | \       | \
                               |  .----` |  `----. |  `----. |  `----.
                               | /       |        \|        \|        \
                     .--------(41)      (42)      (43)      (44)      (45)
                    /         /         /| \       | \
              .----`    .----`    .----` |  `----. |  `----.
             /         /         /       |        \|        \
          (51)      (52)      (53)      (54)      (55)      (56)
```

Note that the links depicted represent directed links in the DAG overlaid on top of the physical topology depicted in Figure 11 (Example LLN Topology). As such, the depicted edges represent the relationship between nodes and their DAG parents, wherein all depicted edges are directed and oriented `up' on the page toward the DAG root (LBR). The DAG provides the default routes within the LLN, and serves as the foundation on which RPL builds further routing structure, e.g. through the Destination Advertisement mechanism.

**Figure 12: Example DAG**

TOC

## B.1. Moving Down a DAG

Consider node (56) in the example of Figure 11 (Example LLN Topology). In the unmodified example, node (56) is at depth 6 with one DAG parent, {(43)}, and one sibling (55). Suppose, for example, that node (56) wished to expand its DAG parent set to contain node (55), as {(43), (55)}. Such a change would require node (56) to detach from the DAG, to defer reattachment until a loop avoidance algorithm has completed, and to then reattach to the DAG with {(43), (55)} as it's DAG parents. When node (56) detaches from the DAG, it is able to act as the root of its own floating DAG and establish its frozen sub-DAG (which is empty). Node (56) can then observe that Node (55) is still attached to the original DAG, that its sequence number is able to increment, and deduce that Node (55) is safely not behind Node (56). There is then little change for a loop, and Node (56) may safely reattach to the DAG, with parents {(43), (55)}. At reattachment time, node (56) would present itself with a depth deeper than that of its deepest DAG parent (node (55) at depth 6), depth 7.

## B.2. Link Removed

Consider the example of Figure 11 (Example LLN Topology) when link (13)-(24) goes down.

*Node (24) will detach and become the root of its own floating DAG

*Node (34) will learn that its DAG parent is now part of its own floating DAG, will consider that it can remain a part of the DAG rooted at node (LBR) via node (33), and will initiate procedures to detach from DAG (LBR) in order to re-attach at a lower depth.

*Node (45) will similarly make preparations to remain attached to the DAG rooted at (LBR) by detaching from Node (34) and re-attaching at a lower depth to node (44).

*Node (34) will complete re-attachment to Node (33) first, since it is able to attach closer to the root of the DAG.

*Node (45) will cancel plans to detach/reattach, keep node (34) as a DAG parent, and update its dependent depth accordingly.

*Node (45) may now anyway add node (44) to its set of DAG parents, as such an addition does not require any modification to its own depth.

- *Node (24) will observe that it may reattach to the DAG rooted at node (LBR) by selecting node (34) as its DAG parent, thus reversing the relationship that existed in the initial state.

## B.3. Link Added

TOC

Consider the example of Figure 11 (Example LLN Topology) when link (12)-(42) appears.

- *Node (42) will see a chance to get closer to the LBR by adding (12) to its set of DAG parents, {(32), (12)}
- *Node (42) may be content to leave its advertised depth at 5, reflecting a depth deeper than its deepest parent (32).
- *Node (42) may now choose to remain where it is, with two parents {(12), (32)}. Should there be a reason for Node (42) to evict Node (32) from its set of DAG parents, Node (42) would then advertise itself at depth 2, thus moving up the DAG. In this case, Node (53), (54), and (55) may similarly follow and advertise themselves at depth 3.

## B.4. Node Removed

TOC

Consider the example of Figure 11 (Example LLN Topology) when node (41) disappears.

- *Node (51) and (52) will now have empty DAG parent sets and be detached from the DAG rooted by (LBR), advertising themselves as the root of their own floating DAGs.
- *Node (52) would observe a chance to reattach to the DAG rooted at (LBR) by adding Node (53) to its set of DAG parents, after an appropriate delay to avoid creating loops. Node (52) will then advertise itself in the DAG rooted at (LBR) at depth 7.
- *Node (51) will then be able to reattach to the DAG rooted at (LBR) by adding Node (52) to its set of DAG parents and advertising itself at depth 8.

## B.5. New LBR Added

TOC

Consider the example of Figure 11 (Example LLN Topology) when a new LBR, (LBR2) appears, with connectivity (LBR2)-(52), (LBR2)-(53).

*Nodes (52) and Node (53) will see a chance to join a new DAG rooted at (LBR2) with a depth of 2. Node (52) and (53) may take this chance immediately, as there is no risk of forming loops when joining a DAG that has never before been encountered. Note that the nodes may choose to join the new DAG rooted at (LBR2) if and only if (LBR2) offers more optimum properties in line with the implementation specific local policy.

*Nodes (52) and (53) begin to send RA-DIO advertising themselves at depth 2 in the DAGID (LBR2).

*Nodes (51), (41), (42), and (54) may then choose to join the new DAG at depth 3, possibly to get closer to the DAG root. Note that in a more advanced case, these nodes also remain members of the DAG rooted at (LBR), for example in support of different constraints for different types of traffic.

*Node (55) may then join the new DAG at depth 4, possibly to get closer to the DAG root.

*The remaining nodes may choose to remain in their current positions within the DAG rooted at node (LBR), since there is no clear advantage to be gained by moving to DAG (LBR2).

---

## B.6. Destination Advertisement

TOC

Consider the example DAG depicted in Figure 12 (Example DAG). Suppose that Nodes (22) and (32) are unable to record routing state. Suppose that Node (42) is able to perform prefix aggregation on behalf of Nodes (53), (54), and (55).

*Node (53) would send a DAO to Node (42), indicating the availability of destination (53).

*Node (54) and Node (55) would similarly send DAOs to Node (42) indicating their own destinations.

*Node (42) would collect and store the routing state for destinations (53), (54), and (55).

* In this example, Node (42) may then be capable of representing destinations (42), (53), (54), and (55) in the aggregation (42').

* Node (42) sends a DAO advertising destination (42') to Node 32.

* Node (32) does not want to maintain any routing state, so it adds onto to the Reverse Route Stack in the DAO and passes it on to Node (22) as (42'):[(42)]. It may send a separate DAO to indicate destination (32).

* Node (22) does not want to maintain any routing state, so it adds on to the Reverse Route Stack in the DAO and passes it on to Node (12) as (42'):[(42), (32)]. It also relays the DAO containing destination (32) to Node 12 as (32):[(32)], and finally may send a DAO for itself indicating destination (22).

* Node (12) is capable to maintain routing state again, and receives the DAOs from Node (22). Node (12) then learns:

  - Destination (22) is available via Node (22)

  - Destination (32) is available via Node (22) and the piecewise source route to (32)

  - Destination (42') is available via Node (22) and the piecewise source route to (32), (42').

* Node (12) sends DAOs to (LBR), allowing (LBR) to learn routes to the destinations (12), (22), (32), and (42'). (42), (53), (54), and (55) are available via the aggregation (42'). It is not necessary for Node (12) to propagate the piecewise source routes to (LBR).

---

## Appendix C. Additional Examples

TOC

Consider the expanded example LLN physical topology in Figure 13 (Expanded LLN Topology). In this example an additional LBR is added. Suppose that all nodes are configured with an implementation specific policy function that aims to minimize the number of hops, and that both LBRs are configured to root different DAGIDs. We may now walk through the formation of the two DAGs.

---

```
                             (LBR)                    (LBR2)
                             / | \                    /    \
                        .---`  |  `----.             /      \
                       /       |        \            |      |
                    (11)------(12)------(13)      (14)      (15)
                     | \       | \       | \       |        /|
                     |  `----. |  `----. |  `----. |  .----` |
                     |        \|        \|        \| /       |
                    (21)------(22)------(23)      (24)      (25)
                     |        /|        /|         |        / /
                     |  .----` |  .----` |  .-----]|[------` /
                     | /       | /       | /       |        /
                    (31)------(32)------(33)------(34)-----`
                     |        /| \       | \       | \
                     |  .----` |  `----. |  `----. |  `----.
                     | /       |        \|        \|        \
           .--------(41)      (42)      (43)------(44)------(45)
          /         /         /| \       | \
    .----`    .----`    .----` |  `----. |  `----.
   /         /         /       |        \|        \
(51)------(52)------(53)------(54)------(55)------(56)
```

**Figure 13: Expanded LLN Topology**

```
                             (LBR)                    (LBR2)
                             / | \                    /    \
                        .---`  |  `----.             /      \
                       /       |        \            |      |
                    (11)      (12)      (13)      (14)      (15)



                    (21)      (22)      (23)      (24)      (25)



                    (31)      (32)      (33)      (34)



                    (41)      (42)      (43)      (44)      (45)



(51)      (52)      (53)      (54)      (55)      (56)
```

**Figure 14: DAG Construction Step 1**

---

---

```
                             (LBR)                    (LBR2)
                             / | \                    /    \
                        .---`  |  `----.             /      \
                       /       |        \            |      |
                    (11)      (12)      (13)      (14)      (15)
                     | \       | \       |         |        /|
                     |  `----. |  `----. |         |  .----` |
                     |        \|        \|         | /       |
                    (21)      (22)      (23)      (24)      (25)


                    (31)      (32)      (33)      (34)


                    (41)      (42)      (43)      (44)      (45)


(51)      (52)      (53)      (54)      (55)      (56)
```

**Figure 15: DAG Construction Step 2**

```
                             (LBR)                    (LBR2)
                             / | \                    /    \
                        .---`  |  `----.             /      \
                       /       |        \            |      |
                    (11)      (12)      (13)      (14)      (15)
                     | \       | \       |         |        /|
                     |  `----. |  `----. |         |  .----` |
                     |        \|        \|         | /       |
                    (21)      (22)      (23)      (24)      (25)
                     |        /|        /          |        / /
                     |  .----` |  .----`    .-----]|[------` /
                     | /       | /         /       |        /
                    (31)      (32)      (33)      (34)-----`



                    (41)      (42)      (43)      (44)      (45)



(51)      (52)      (53)      (54)      (55)      (56)
```

**Figure 16: DAG Construction Step 3**

```
                             (LBR)                    (LBR2)
                             / | \                    /    \
                        .---`  |  `----.             /      \
                       /       |        \            |      |
                    (11)      (12)      (13)      (14)      (15)
                     | \       | \       |         |        /|
                     |  `----. |  `----. |         |  .----` |
                     |        \|        \|         | /       |
                    (21)      (22)      (23)      (24)      (25)
                     |        /|        /          |        / /
                     |  .----` |  .----`    .-----]|[------` /
                     | /       | /         /       |        /
                    (31)      (32)      (33)      (34)-----`
                     |        /|         | \       | \
                     |  .----` |         |  `----. |  `----.
                     | /       |         |        \|        \
                    (41)      (42)      (43)      (44)      (45)



(51)      (52)      (53)      (54)      (55)      (56)
```

**Figure 17: DAG Construction Step 4**

```
                             (LBR)                    (LBR2)
                             / | \                    /    \
                        .---`  |  `----.             /      \
                       /       |        \            |      |
                    (11)      (12)      (13)      (14)      (15)
                     | \       | \       |         |        /|
                     |  `----. |  `----. |         |  .----` |
                     |        \|        \|         | /       |
                    (21)      (22)      (23)      (24)      (25)
                     |        /|        /          |        / /
                     |  .----` |  .----`    .-----]|[------` /
                     | /       | /         /       |        /
                    (31)      (32)      (33)      (34)-----`
                     |        /|         | \       | \
                     |  .----` |         |  `----. |  `----.
                     | /       |         |        \|        \
           .--------(41)      (42)      (43)      (44)      (45)
          /         /         /|         | \
    .----`    .----`    .----` |         |  `----.
   /         /         /       |         |        \
(51)      (52)      (53)      (54)      (55)      (56)
```

**Figure 18: DAG Construction Step 5**

## Authors' Addresses

TOC

| | |
|---|---|
| | Tim Winter (editor) |
| Email: | wintert@acm.org |
| | |
| | ROLL Design Team |
| | IETF ROLL WG |
| Email: | dtroll@external.cisco.com |