Internet Engineering Task Force
Internet-Draft
Intended status: Experimental
Expires: 20 April 2026

R. Bless
Karlsruhe Institute of Technology (KIT)

17 October 2025

Kademlia-directed ID-based Routing Architecture (KIRA)
draft-bless-rtgwg-kira-04

Abstract

This document describes the Kademlia-directed ID-based Routing Architecture KIRA. KIRA is a scalable zero-touch distributed routing solution that is tailored to control planes. It prioritizes scalable and resilient connectivity over route efficiency (stretched paths are acceptable vs. routing protocol overhead). KIRA's self-assigned topological independent IDs can be embedded into IPv6 addresses. Combined with further self-organization mechanisms from Kademlia, KIRA achieves a zero-touch solution that provides scalable IPv6 connectivity without requiring any manual configuration. For example, it can connect hundreds of thousands of routers and devices in a single network without requiring any form of hierarchy (like areas). It works well in various topologies and is loop-free even during convergence. This self-contained solution, and especially the independence from any manual configuration, make it suitable as resilient base for all management and control tasks, allowing to recover from the most complex failure scenarios. The architecture consists of the ID-based network layer routing protocol Rツイ/Kad in its Routing Tier (using source routing) and a PathID-based Forwarding Tier (using PathIDs as labels for paths). KIRA寰洰 tightly integrated add-on services (e.g., name resolution as well as fast and efficient topology discovery) provide a perfect basis for autonomic network management solutions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 20 April 2026.

Table of Contents

1. Introduction

KIRA is a scalable zero-touch distributed routing solution that is tailored to control planes. In contrast to commonly used routing protocols like OSPF, ISIS, BGP etc., it prioritizes resilient connectivity over route efficiency. It scales to hundreds of thousands of nodes in a single network, uses ID-based addresses, is zero-touch (i.e., it requires no manual configuration for and after deployment) and works well in various network topologies. Moreover, it offers a flexible memory/stretch trade-off per node, shows fast recovery from link or node failures, and is loop-free, even during convergence. Additionally, it includes a built-in Distributed Hash Table (DHT) as an add-on service that can be used for simple name service registration and resolution, thereby helping to realize autonomic network management, control, and zero-touch deployments.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Overview of KIRA

KIRA's main objective is to provide self-organized robust control plane (CP) connectivity on top of a link-layer topology. The CP is typically used to configure, monitor, manage, and control networked resources (switches, routers, end-systems). The goal is to never lose control over the resources as long as there exist paths leading to them. KIRA is structured into a two-tier architecture consisting of a Routing Tier and a Forwarding Tier (see Figure 1). KIRA runs the zero-touch, distributed, highly scalable, ID-based routing protocol Rツイ/Kad in the Routing Tier to find viable paths to destinations. The core concept of Rツイ/Kad is that it discovers paths in the underlying topology by using an ID-based overlay routing scheme (based on Kademlia) combined with source routing between overlay hops. KIRA nodes employ this information to construct fast path forwarding tables in the Forwarding Tier for CP data traffic (e.g., packets from control plane applications). While source routing is robust (since it does not require converged routes) it can cause significant overhead, especially for small payloads. The Forwarding Tier avoids this overhead for CP data traffic by using PathIDs as forwarding state instead of the source routes and a scheme that is similar to label switching. However, existing forwarding support for IPv6 in hardware can be used, so KIRA does not require specialized new forwarding mechanisms.

Add-on modules can make use of KIRA's routing information and available mechanisms, i.e., they are tightly coupled but offer optional support. Examples are a fully-distributed name resolution service (based on KIRA's built-in partial DHT functionality) and an efficient topology discovery service (KeLLy, [KeLLy-2023]).

Rツイ/Kad employs a flat ID-based addressing scheme to easily support zero-touch operation, self-organization as well as mobility and multi-homing. ID-based routing (sometimes also denoted as name-independent routing or routing with flat identifiers) has the advantage of providing stable addresses (called NodeIDs) to upper layers. Thus, in case (virtual) resources are moved within the topology, any control connection to them stays alive. In contrast to other ID-based addressing approaches, KIRA is a genuine ID-based approach, because it does not use topological addresses at all and thus does not require any additional identifier-locator mapping (increased risk of non-consistency) and associated protocols (additional overhead and convergence time).

As just motivated, Rツイ/Kad uses topologically independent NodeIDs, generated by the KIRA nodes themselves, so address assignment is performed in a distributed manner by each node autonomously. Typically, NodeIDs are taken from a 112 bit address space, but

depending on the network size, smaller NodeIDs are possible. KIRA uses IPv6 packets [RFC8200] for its messages and CP data packets, because NodeIDs can be easily embedded into an IPv6 address (e.g., 16 bit prefix + 112 bit NodeID) and existing hardware-based and software-based forwarding mechanisms can be leveraged. Mainly very basic IPv6 features like link-local addresses, the packet format, fragmentation, and neighbor discovery are used, e.g., it does not require any address configuration features or router discovery.

The _Forwarding Tier_ is able to forward IPv6 packets, so that (control) applications can use IPv6 and all corresponding transport protocols above. Rツイ/Kad messages use source routing based on NodeIDs whereas traffic in the Forwarding Tier uses NodeIDs and PathIDs for its forwarding decision. PathIDs are conceptually a hash from a sequence of NodeIDs that build a path segment. PathIDs are unique (with high probability) for a path segment. To carry PathIDs in addition to the final destination NodeID, the original IPv6 packet becomes encapsulated, e.g., using a segment routing header that contains the PathID for the path segment that the packet should traverse next. Intermediate nodes simply exchange the PathID at each hop with the PathID of the remaining path segment (similar to label switching), or they strip the outer header when the next node is the end of a path segment. PathIDs are precomputed in a 2-hop vicinity of a KIRA node and are installed by Rツイ/Kad signaling in some intermediate nodes on demand for paths longer than five hops. To forward CP packets KIRA nodes only need to perform lookups in their NodeID forwarding table and/or PathID forwarding table, perform encapsulation or decapsulation and rewrite PathIDs.

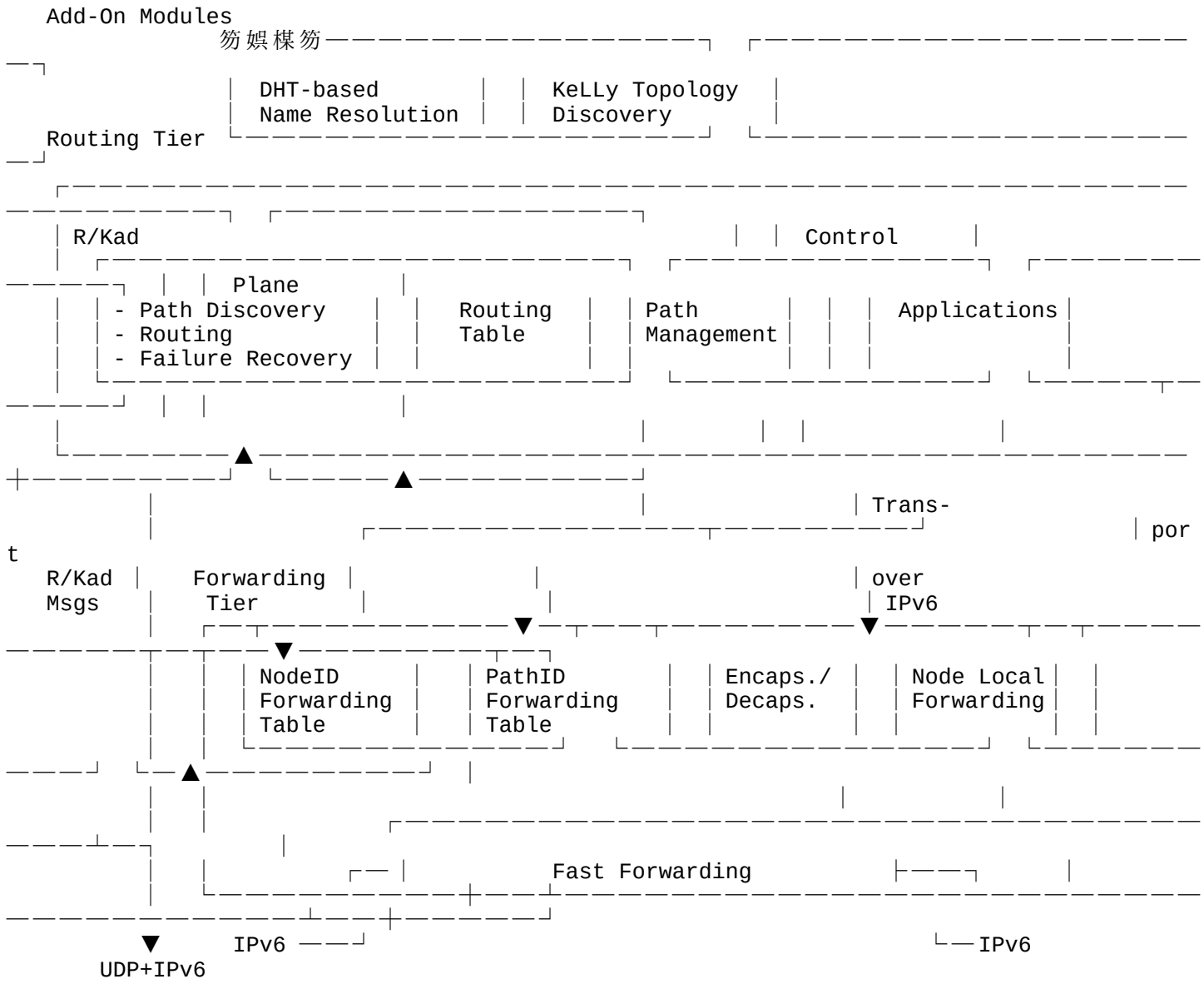

Figure 1: KIRA-Architecture

CP applications (e.g., SDN controllers, Kubernetes Cluster Controllers, Virtual Infrastructure Manager, traditional OAM applications and so on) simply use NodeIDs as addresses for the resources/devices they want to control or for other controllers they want to exchange state with. Therefore, CP applications can transparently use the connectivity established by KIRA. Since NodeIDs are randomly generated, KIRA provides a simple built-in key/value store (Distributed Hash Table DHT) that can be used as name service. KIRA nodes and services can dynamically register their NodeID under a certain well-known name and other KIRA nodes can lookup their corresponding NodeIDs. The DHT functionality will be specified as a separate KIRA module and corresponding application interfaces are out-of-scope for this specification.

KIRA nodes possess relatively small routing tables, that grow with O(log(n)), where n is the number of KIRA nodes in the network (see [KIRA-Networking-2022] for evaluation results). The advantage of small routing tables is scalability, but comes at the cost of path stretch. That is, packets to destinations that are not kept in the routing table of a node take a longer path than the shortest possible path, because they are using the ID-based overlay routing strategy. However, KIRA nodes will learn the shortest paths to all 'contacts' in their routing tables and it is a node local decision how large the routing table can be. For example, a controller node may add all KIRA nodes that it controls as contacts to its routing table. Because KIRA uses source routing in R/Kad and PathID-based forwarding in its forwarding tier, it can easily support multi-path routing and keep backup paths for fast fail-over reactions.

KIRA uses a mixture of reactive and periodic mechanisms to cope with link and node failures. Error messages that indicate failed links usually trigger routing updates and a path rediscovery procedure. However, routing updates are not flooded to all KIRA nodes, so some nodes may still have obsolete path information. These inconsistencies will be detected either when using the obsolete path to a contact (triggering an error message from the node before the broken link) or by a maintenance procedure that is carried out periodically. These periodic maintenance procedures test the validity of the currently known paths and may also trigger a rediscovery procedure to find alternative paths.

Moreover, KIRA also possesses a specific end-system mode, where KIRA nodes are part of the KIRA network, but they are not exchanging routing information and are not forwarding packets for other KIRA nodes (see Section 3.12).

Finally, KIRA also supports a domain concept. A KIRA node may be member of one or multiple domains. Unless configured otherwise, a KIRA node is member of the global domain with DomainID=0 by default. KIRA nodes keep their NodeID for all domains, the only difference is that routes are guaranteed to run inside a domain D in case source and destination nodes are both members of this domain D. This allows for using domains for administrative purposes (e.g., all KIRA nodes inside the same Autonomous System could be part of the same domain) or to use domains to build clusters of KIRA nodes that are grouped by closeness in the underlying network topology.

3. Protocol Operation

This section gives on overview of the main concepts with respect to the R/Kad protocol operation. First, KIRA's ID-wise addressing concept is introduced, then the routing table structure is presented. After that, several procedures are described, beginning with node startup, vicinity discovery, and the join procedure to populate the routing table, followed by path discovery, overhearing, and rediscovery mechanisms.

3.1. Addressing, NodeIDs and the XOR Metric

Every KIRA instance uses a single NodeID as its address. The NodeID is taken from a larger unstructured address space [0..2^(B-1)] (typically B=112). KIRA uses the XOR (logical exclusive or) metric in this address space to define the distance between two NodeIDs X and Y (see also [Kademlia2002]). The distance function d(X,Y)= X XOR Y is interpreted as integer and fulfills all properties of a metric (d(X,Y)>=0, and d(X,Y)=0 <=> X=Y; d(X,Y)=d(Y,X); as well as the triangle inequality d(X,Y)<=d(X,Z)+d(Z,Y)). This distance function largely corresponds to a prefix bit distance metric d_p(X,Y) = B-lcp(X,Y), where lcp(X,Y) denotes the length of the longest common prefix in bits. The XOR metric is finer than the d_p(X,Y) metric though, because when there are X,Y, and Z with d_p(X,Y)=d_p(Y,Z)=d_p(X,Z), the XOR metric can uniquely determine whether Y or Z are closer to X. More generally speaking, for a given distance d(X,Y)=d there exists exactly one Y so that d(X,Y)=d. This property is important since it allows to unambiguously determine which NodeID is ID-wise closer to a given other NodeID and it provides the basis for KIRA's loop-freedom. Note that the ID space with this metric is not cyclic (i.e., a node with a very small NodeID is not close to a node with a very large NodeID).

The XOR metric defines an overlay structure across all KIRA nodes in the ID space: KIRA nodes establish logical connections with their ID-wise closest overlay neighbors (which are typically different from the underlay neighbors) with respect to the XOR metric as distance metric, i.e., the ID-wise closer neighbors have a smaller distance according to XOR. KIRA uses this metric to determine the next KIRA node that a KIRA message is forwarded to in order to reach a certain destination NodeID. R/Kad messages are forwarded by using source routing between overlay hops.

In addition to NodeIDs, KIRA can use any ID from the ID space as destination address. Typically, names of objects can be hashed to result in a key value, which is called Resource ID. In this case, the ID-wise closest KIRA node will be found as responsible node for storing a value (or a referral) for this key. This makes it possible to provide an integrated DHT for name-to-NodeID registration and lookup.

3.2. NodeID Creation

NodeIDs are randomly generated and are taken from a 112 bit address space. Reserved NodeIDs listed in Section 7.3 MUST NOT be used as NodeID of a KIRA node. Future versions of this specification will detail an algorithm to create self-certifying NodeIDs as using certain hash functions from a public key. NodeIDs are unique with high probability, e.g., for 10^12 nodes the probability for uniqueness is at least 1-9.6310^−11. However, in case two nodes possess the same NodeID, protocol mechanisms can be used to detect this situation (will be specified in a later version of this document) and the conflict can be resolved by letting both sides generate new NodeIDs. Depending on a KIRA node’s capabilities, NodeIDs (together with other protocol parameters) may be stored in non-volatile memory so that nodes keep their NodeID even after restart. Other KIRA nodes may choose to generate a new NodeID on every restart.

3.3. Routing Table

The entries in the routing table (RT) are called ’contacts’. Contact data contains the NodeID of the contact as well a set of discovered paths that lead to this contact (besides other node state and attributes). A path to a contact is stored as path vector that contains a complete sequence of NodeIDs, which can be traversed to reach the contact. Except for contacts in its routing table, a KIRA node does not know paths to other destinations, but they can be discovered by using a recursive overlay routing strategy: a KIRA node source routes a packet to the contact (using the known path) that is the ID-wise closest to the destination ID according to its routing table. The next overlay hop performs the same action until the destination node is reached. After the initial discovery phase, only Underlay Neighbors (ULNs) and some contacts from the 3-hop underlay neighborhood are stored in the routing table. Underlay Neighbors are nodes attached to the links of the KIRA node (i.e., neighbors in the sense of [RFC8200] that are directly reachable via link layer and the Internet-layer or higher-layer tunnels).

R/Kad's efficiency and flexibility are closely related to its routing table. It is structured as tree of k-buckets as in [Kademlia2002]. A k-bucket in the routing table contains a list of (at most) k contacts in distance between 2^i and 2^(i+1) (i.e., the bucket's range, where 0<=i<112) from this node. Usually, k>=20 is constant and the same for all buckets and nodes, but it can also be varied per node (k=40 is RECOMMENDED as default for R/Kad). Buckets at deeper levels share more prefix bits with the node's own ID, however, buckets for small values of i are generally empty as no appropriate nodes exist in this address space. Thus, the highest bucket (depth 1) contains contacts from half of the ID space whose highest NodeID bit differs from the node's ID, whereas the deepest buckets contain all nodes that are ID-wise closest to the node (i.e., the ID-wise closest overlay neighbors).

If KIRA node X learns a new contact Y, it puts it into the corresponding k-bucket b_l in case it still has capacity left. The bucket index l is determined by calculating the common prefix length between X and Y (number of high-order zero bits of d(X,Y)). If the bucket contains k entries already, it is split into two new buckets (and the contained entries moved to them accordingly) in case X falls into the bucket's range. Otherwise, a selection algorithm determines whether the new contact should replace an existing entry in this bucket. In our case we use Proximity Neighbor Selection (PNS) in all but the deepest two buckets so that contacts with shorter path lengths are preferred. In case path lengths are equal, nodes with a higher degree (see NodeDegree in Section 3.3) SHOULD be preferred as this results in shorter paths (following a powerlaw topology strategy). However, other contact selection strategies are possible even on a per-node basis, but the desired effect may not be visible unless all nodes follow the same strategy. Setting and changing strategies even during runtime inside a Domain is left for future work.

An additional mechanism for path selection improves path diversity and prevents route flapping: in case an alternative path of equal length has been discovered for an already known contact, this path replaces the previous path only if the hash sum of the path elements is closer to the node's own NodeID. Due to the uniqueness property of the XOR metric, the path selection will always unambiguously converge to a unique path.

Underlay neighbors are kept in special buckets that have no capacity limit, i.e., they will never be preempted. In general, routing also works without this ULN buckets extension of [Kademlia2002], but the resulting stretch will be slightly higher.

X identifies its closest known contact in its RT by locating the k-bucket that corresponds to the longest matching prefix of destination Z with its own NodeID X by using d(X,Z). It then selects a contact with the shortest path vector from within the bucket; this is called Proximity Routing (PR). The XOR metric is used to uniquely select the ID-wise closest contact if all paths have equal length or if destination Z falls into the node's deepest bucket.

Each contact typically contains the following information:

* _NodeID_ of the contact.

* _State_: the state that the contact is considered to be in. It is one of _undefined_, _valid_, _rediscovering_, _invalid_, _dead_.

* A set of path vectors, i.e., a complete sequence of NodeIDs that lead to the contact when traversed. There is an _active_ path (used as default), a _proposed_ path and possibly several backup or alternative paths. Paths possess also timestamps for points in time when they were last validated and when they were refreshed (using PathSetupRsp or ProbeRsp, see also Section 3.10).

* _Sequence Number_: a sequence number that is determined by the contact and that changes when the underlay connectivity state changes at the contact, i.e., with newly detected or failed underlay neighbors.

* _LastSeen_: time of last direct contact

* _Sync ULN Sequence Number_: a sequence number used for state synchronization with ULNs.

* _UnderlayNeighbor_: a boolean value indicating that this contact is an underlay neighbor

* _NodeDegree_: the contact's number of underlay neighbors.

* _Rediscovery State_: temporary state keeping track of the rediscovery process.

3.4. Node Startup and Vicinity Discovery

KIRA nodes generate their NodeID first (see Section 3.2). After that they start an initial discovery phase to explore their underlay vicinity. After that, a join procedure and continuous discovery are periodically repeated. To let the node stay connected to its overlay and to improve the quality of discovered routes.

In its initial discovery phase, a KIRA node discovers its underlay adjacencies, i.e., its underlay neighbors (ULNs) that can be directly reached via link-local communication on one of their network interfaces. KIRA nodes periodically send ULNHello messages to a well-known link local multicast address ALL-KIRA-NODES, and receiving nodes may reply with a ULNDiscoveryReq to set up an adjacency. This ULNDiscoveryReq MUST be answered by a ULNDiscoveryRsp to establish an adjacency. The protocol exchange ensures that bidirectional communication is possible between directly adjacent nodes. ULNs may be put into the _ULNTable_ either after receiving a ULNDiscoveryReq as answer to a transmitted ULNHello or after receiving a ULNDiscoveryRsp as answer to a transmitted ULNDiscoveryReq. The ULNTable contains a mapping from NodeID to the link local unicast address of the ULN. This address is either taken from the source address of the ULNDiscoveryReq or the ULNDiscoveryRsp respectively.

KIRA nodes also discover all nodes in their 3-hop underlay neighborhood to populate their routing table, but the 2-hop vicinity is fully stored in a local graph structure ("vicinity graph"). The latter is used to precompute PathIDs for the 2-hop vicinity. ULNDiscoveryReq and ULNDiscoveryRsp messages contain a list of underlay neighbors so that all ULNs of ULNs will be learned, i.e., the 2-hop vicinity. However, possibly not all nodes in the 2-hop vicinity will be stored in the RT, therefore the vicinity graph structure is used to keep track of the 2-hop vicinity (new nodes and links). Especially, the state sequence numbers of all the nodes should be stored in the vicinity graph as well. All nodes in the 2-hop vicinity are queried for their ULNs by using a QueryRouteReq. QueryRouteReq/QueryRouteRsp messages are used to get ULNs or routing table information from nodes in the vicinity. Depending on their NodeID and other criteria (e.g., NodeDegree), nodes from the 3-hop vicinity will be stored as contacts into the routing table. The vicinity graph structure is also used to determine whether a new QueryRouteReq must be initiated according to the state-seq-num of the nodes in the vicinity. The vicinity graph can also be used to quickly calculate alternative paths, e.g., when direct links to ULNs break.

A KIRA node continues to populate its routing table by sending FindNodeReq messages to certain nodes (especially those in the own ID-wise neighborhood) and also to randomly chosen destinations. The "Random Probing" procedure uses a randomly chosen ID from the NodeID space as destination for a FindNodeReq, however, a KIRA node does not need to exist for the chosen ID. The FindNodeReq will simply end at the KIRA node that is ID-wise closest to the destination ID. FindNodeRsp messages return a set of contacts that are ID-wise closest to the destination NodeID from the viewpoint of the responding KIRA node. The returned information is analyzed whether it can improve the local routing table, e.g., new and 'better' contacts or 'better' paths to already known contacts.

3.5. Join Procedure

As result of the vicinity discovery, all ULNs of X and some nodes within its 3-hop radius will populate X' s RT. However, in order to get network connectivity and to contribute to connectivity, the node needs to find its ID-wise closest overlay neighbors and make itself known to them. Thus, to join the network KIRA node X simply "searches" for the k closest nodes to its own NodeID: X sends a FindNodeReq for its own NodeID X and the closest neighbor replies with FindNodeRsp. Join requests can be detected by seeing that the source NodeID (src-node-id) and destination ID (dest-id) are identical. In this case the closest neighbors may know X, but they will exclude X from the result list and also answer the FindNodeReq instead of forwarding it to X.

This is repeated with a limited exponential backoff in order to detect or heal any network partitioning. In case node X finds itself in situations where it needs to respond with "Dead End" Error messages to FindNodeReqs, it SHOULD reset the backoff timer, because it may be a hint for network partitioning or other inconsistencies. In order to let a joining node X quickly learn all existing ID-wise closest overlay neighbors, X SHOULD send a QueryRouteReq to every newly learned contact that enters X' s currently deepest k-bucket. The queried contact replies with a QueryRouteRsp returning its RT entries for the k closest contacts to node X. The so returned contacts will very likely also fall into X' s deepest bucket, possibly leading to a further split of its deepest bucket. Therefore, X will quickly populate its set of ID-wise closest overlay neighbors, which are needed for consistent overlay connectivity.

3.6. Path Discovery

Consider the example topology in Figure 2. Assume node X needs to send a message to node Z. In case Z is a known contact of X, a path vector is stored already in the routing table that can be used for strict source routing in order to reach Z. Otherwise, a path to Z must be discovered using ID-based overlay routing. R/Kad uses a recursive version of Kademlia (hence its name Routing with Recursive Kademlia R/Kad). The Path Discovery procedure uses a request/response message pair, FindNodeReq/FindNodeRsp. In this example, assume that X identifies its contact Y (learned from ULN A) as next (ID-wise closest) overlay hop toward Z. In order to discover a path to Z, X creates a FindNodeReq message that contains destination NodeID Z and source route r=X, A, Y using path vector A of contact Y. The FindNodeReq is forwarded along r (strict source route). Message forwarding between overlay neighbors requires source routing, because the path in the underlay may lead via nodes that are (ID-wise) further away from the destination (e.g., Y routes via A, Q, M to Z in Figure 2): using the ID-based overlay routing scheme on a hop-by-hop basis (i.e., between directly adjacent nodes in the underlay) would inevitably lead to forwarding loops in most cases.

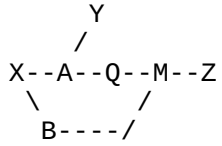

Figure 2: An example topology. Letters resemble NodeIDs. Letters closer in the alphabet have smaller distance in ID space.

When the FindNodeReq arrives at Y, the same procedure is repeated (since it is a recursive variant of Kademlia). Node Y tries to find a contact closer to Z than Y itself. If this contact exists, source route r is appended by the corresponding path vector and the FindNodeReq is forwarded to this contact. In the given example of Figure 2), we assume that Y knows Z as its contact with path vector A, Q, M. It extends source route r of the FindNodeReq by A, Q, M, Z and forwards it to A as next hop in the source route. If routing information has been converged, this ID-based routing scheme guarantees progress in the ID space [Kademlia2002] during forwarding and eventually finds node Z.

The destination node Z responds with a FindNodeRsp message along the reversed source path with any cycles removed (Z, M, Q, A, X). Due to XOR's symmetry, the responding node Z also learns the new contact X as neighbor. The FindNodeRsp returned to X not only provides a path to Z, but also a list of k closest contacts to Z together with their path vectors. This list is used to improve X's routing table.

In case that the FindNodeReq arrives at Y and it cannot find a contact closer to Z, the FindNodeReq is terminated at Y and a response is sent back depending on the "ExactFlag" Section 4.4.1 in the FindNodeReq. If the ExactFlag was not set, Y sends a FindNodeRsp back to originator X that contains an RT excerpt of Y's (at most) k closest contacts to Z in a so called RTable object. This enables finding the responsible node for a destination ID Z if used as object key. The latter allows for so called key-based routing that is used to realize DHTs. If exact was set, X assumed that a node with ID Z must exist, but the current node is the ID-wise closest node to Z and does not know Z as contact. Consequently, the node cannot forward the FindNodeReq closer to Z and returns a "Dead End" Error message (which may happen occasionally during convergence). In case source route r contains a broken link or unreachable node, a "Segment Failure" Error message will be sent back to X along the reversed source route (with any cycles removed).

3.7. Overhearing of R/Kad Messages

KIRA nodes use overhearing mechanisms for R/Kad messages. This is an important mechanism for KIRA to learn new contacts and better or improved paths.

Nodes that forward R/Kad messages SHOULD use the contained source route to improve their own routing information: they may learn new contacts or shortcut routes to known contacts. However, only the so far traversed path is considered as it can be assumed that all traversed links worked recently. Additionally, NotViaList information (see Section 4.4.2.3) is used to invalidate contacts that have an active path vector containing a link from the NotViaList.

The source routing path of incoming requests and responses is also considered for improving the RT. Some messages like FindNodeRsp, QueryRouteRsp or UpdateRouteReq contain RTable objects that are evaluated likewise. However, in contrast to the source route information that can be considered as being most recent information, RTable information may be less recent (and also less trustworthy) and needs to be validated first before it can be used. Therefore, in case a path from the RTable is considered being of interest (improvement over an existing path), it needs to be checked by sending a ProbeReq message and successfully receiving the corresponding ProbeRsp. See Section 3.9.3 for more details.

Bypassing QueryRouteRsp messages contain RTable objects (as requested by QueryRouteReq) and SHOULD be inspected for interesting contacts and paths.

3.8. Periodic Path Probing

Periodic Path Probing aims at reliably detecting any RT inconsistencies (e.g., seemingly valid contacts with paths that contain recently failed links). Each node periodically checks the path validity for all of its contacts by sending a ProbeReq message to them. ID-wise closest neighbors are probed more often than other contacts and those recently contacted (2s) are not probed. In case a path has a link or node failure, the ProbeReq will elicit a "Segment Failure" Error message from an intermediate node along the broken path, notifying about the failed link. The contact’ s state will be set to _invalid_ and a rediscovery process is scheduled (see Section 3.9.1).

3.9. Dynamics: Recovery from Failures

In order to improve R/Kad’ s robustness against link or node failures we introduce a recovery procedure that notifies about failures and actively tries to find alternative paths that route around the failure. This procedure is highly robust and achieves a fast convergence. R/Kad nodes detect link and node failures of ULNs by link layer notifications, missing ULNHello or ULNDiscoveryRsp messages as well as "Segment Failure" errors anytime during forwarding along source routes. To recover from such failures, R/Kad’ s recovery procedure uses the following mechanisms:

* Notify own nearest overlay neighbors about failed links or unreachable nodes ("bad news") by sending UpdateRouteReqs via a non-impacted underlay link.

* Rediscover a feasible alternative route to the affected node using FindNodeReqs. These carry NotViaList information about failed links that must not be considered for routing the FindNodeReq. Rediscovery is not performed for nodes that lost their only link, which can be deduced by the node' s degree information that is conveyed in R/Kad messages.

* Per contact _state sequence numbers_ avoid using obsolete information for path rediscovery. Additionally, an _aging_ mechanism is used to avoid dissemination of obsolete routing information. It uses time periods to assess the currentness of the related path.

* Overhearing of NotViaList information and UpdateRouteReqs about failed links during forwarding R/Kad messages informs nodes about failed links, which initiates a path rediscovery. Overhearing is also used to update obsolete path information.

* When an alternative path has been found for a prior affected contact or a link comes back up again, an UpdateRouteReq is sent to own ID-wise closest overlay neighbors for disseminating the "good news".

The ID-based overlay routing scheme is used for rediscovery of a route, because NodeIDs are randomly distributed all over the underlying topology. Therefore, a rediscovery uses different paths that are likely not affected by the failure. However, if overlay nodes still have obsolete routing information, i.e., they would normally route via the failed link, they can detect the need to update their routes as well by seeing the more current NotViaList information.

3.9.1. Path Rediscovery

A node X that detects its ULN B (cf. Figure 2) or the corresponding link X, B has failed, reacts as follows (unless isolated by that failure):

1. Set the state of the corresponding contact to _invalid_ (in Figure 2 contact B). Invalid contacts will temporarily not be considered for routing.

2. Set the state of all contacts whose paths contain the failed link X, B to invalid (in Figure 2 contact Z with B, M, Z becomes invalid).

3. Send UpdateRouteReq messages indicating the failure to four of its ID-wise nearest neighbors (e.g., Y and Z in Figure 2) via non-affected contacts. The UpdateRouteReq will also carry a NotViaList that contains X, B.

4. Trigger a rediscovery process (described below) for B (sets state to rediscovery) and for other invalid contacts.

5. If the rediscovery process is successful for a contact, its state is set to _valid_ and UpdateRouteReq messages are sent to notify ID-wise closest neighbors about the change.

Since UpdateRouteReqs have notification character only, they do not create any responses (even no error messages if dropped). The rediscovery process simply sends a FindNodeReq for all invalid contacts (all invalid contacts will be ignored in finding the next hop). This FindNodeReq for rediscovery (also denoted as rediscovery message) contains a set ExactFlag and the failed link X, B as additional NotViaList information. It is sent to X' s currently known ID-wise closest neighbors of the invalid contact (e.g., A in the example), which will then try to forward the FindNodeReq further toward the failed contact. The NotViaList information avoids that nodes use obsolete routing information when forwarding the rediscovery message, i.e., paths that contain the failed link will not be used for forwarding. Node A may not have heard yet about the broken link and thus will invalidate contact B if its prior preferred path is via X, B. In order to ensure that only current NotViaList information is considered, every link contained in the NotViaList is also accompanied by a related age value T, specifying in milliseconds how long ago the sender heard about the failed link. In case a FindNodeRsp is returned by B, a valid path has been discovered and the contact' s state is set to valid (triggering subsequent UpdateRouteReqs with the new path as mentioned before).

Nodes receiving UpdateRouteReqs or FindNodeReqs containing the failed link also set their corresponding affected contacts to invalid and trigger a rediscovery process of the routes (like Z in the example). The actual rediscovery messages are sent after different randomly chosen waiting times from an interval [0.5t_p, 1.5t_p]. The mean value t_p is set as follows: for invalidated ULNs 100ms, affected ID-wise near contacts (in the deepest buckets) 500ms, for contacts affected by the failure of a link to a ULN 1s and for all other affected contacts 2s. Rediscovery messages are sent simultaneously to two different (overlay) neighbors of the affected contact at a time, until k neighbors have been tried unsuccessfully to rediscover a path to the currently invalid contact. In the latter case, a new round of rediscovery attempts will be initiated with exponential backoff until a certain limit of retry rounds (default: 6) have been

made without any success, after which the contact will be deleted. Although there is no guarantee that a viable alternative route can be found, our simulation results show that connectivity is very quickly restored after a failure even in drastic failure scenarios (i.e., where a larger part of links, such as 15%, fail simultaneously and randomly).

Node B at the other end of the failed link X, B also tries to rediscover X and thus sends an UpdateRouteReq to its ID-wise closest overlay neighbors (e.g., A). Thereby, it may inform A as well as X about a new alternative route via M.

3.9.2. Fast Vicinity Alternatives

In case direct links to underlay neighbors fail or links to nodes in the vicinity, the vicinity graph can be used to find alternative paths. Many real-world topologies possess triangle like structures so that former direct underlay neighbors can be indirectly reached via another intact direct underlay neighbor. A path that is found by this method, needs to be validated first by a ProbeReq/ProbeRsp message exchange as discussed in section Section 3.9.3. It is RECOMMENDED that this fast procedure is carried out before a rediscovery of the faulty path starts. If the path validation is successful, a rediscovery can be avoided.

The advantage of this procedure is to provide an alternative route quickly without having to perform a full Path Rediscovery via the overlay.

3.9.3. Ensuring Routing Information Validity

R/Kad uses state sequence numbers and aging to prevent obsolete routing information from spreading or settling. Messages carry routing information in an RTable object that contains a list of contacts n_j, and for each contact n_j the corresponding path vector p_j leading from the reporting node to the contact, its state sequence number s_j and the age T_j of this information. The currentness of contact information can always be assessed by s_j. However, s_j alone does not suffice to assess the currentness of the associated path to this contact as intermediate links may have been failed/repaired. Therefore, each reported path, as well as NotVia links, carry an associated age value T_j 0 that corresponds to the time period when the path information was updated last at the originating node. This avoids spreading and wrongfully accepting obsolete routing information.

The age for underlay links of a node is always 0ms, because related information is always current at this node. A node simply sets a "last modification" timestamp t_j for the contact n_j to t_j := t_now − T_j and reports n_j' s age as t_now-t_j in messages with RTable objects (e.g., UpdateRouteReq, FindNodeRsp, QueryRouteRsp). A contact' s timestamp t_j is also updated by messages that allow to infer that the traversed path is current, e.g., incoming ProbeReq, ProbeRsp , FindNodeRsp, QueryRouteReq, and QueryRouteRsp messages. A path is updated only if the contact' s state sequence number is larger than the prior known sequence number for this contact, or, in case of equal sequence numbers, the received path information must be more recent when comparing their age values. Since age values are relative, they can be compared even if they stem from different nodes, i.e., synchronized clocks are not required.

Finally, path information that originates from Source Route Objects is considered to be validated, because the path was traversed by the corresponding message recently. Path information taken from RTable Objects or that was improved with local RT information is considered to be not validated. A not validated path should not overwrite an active valid path, because it may nevertheless include broken links that the node is not aware of yet. Therefore, a not validated path that is considered to be "better" (e.g., "shorter") than the current active path is stored as proposed path needs to be validated by probing. A ProbeReq for the proposed path should be scheduled. Newly arriving not validated path information for the same contact should be compared against this proposed path. In case the path probing of the proposed path is successful, the source route of the ProbeRsp will automatically update the active valid path and the proposed path should be dismissed (unless newer than the ProbeReq, but then another path probing is scheduled).

The previously described mechanisms cannot guarantee notification of all affected nodes about link failures in their path vectors. In order to reliably detect such inconsistencies, each node periodically probes the paths to all its contacts as described in Section 3.8. Nevertheless, if a node tries to use an obsolete path with a failed link, a viable path will be rediscovered immediately after receipt of the Error message from the node before the broken link.

3.10. Fast Forwarding of CP Traffic

A potential drawback of R/Kad is its use of source routing to forward between two overlay hops. Handling a (potentially long) list of source routing hops is currently not as efficiently realized as regular destination-based routing. Moreover, source routing increases per-packet overhead. To forward data packets more efficiently, the Forwarding Tier (see Figure 1) leverages an approach

similar to label switching, whereas the Routing Tier still uses source routing for R/Kad messages to remove cycles, detect shortcuts, and so on. Every source routing path (that consists of NodeIDs) to a contact is represented by up to two PathIDs that correspond to _path segments_. A PathID is a hash value of all NodeIDs along the corresponding path segment, e.g., PathID(A, Q, M, Z)=H(A|Q|M|Z). It serves as unique label for the path segment. The uniqueness is an important distinction from common label switching approaches where nodes assign labels of node local scope. It enables KIRA nodes to distributedly compute a set of PathIDs in advance. This avoids explicit path setup signaling for PathID installation in many cases. Only for paths longer than 5 hops PathID mappings have to be installed in some intermediate nodes. Another feature of PathIDs is their automatic aggregation toward a sink, i.e., paths that merge in a certain node and use the same residual path to a destination use the same PathID. The Forwarding Tier uses SRv6 encapsulation [RFC8754] to carry PathIDs in addition to source and destination NodeIDs (other encapsulation methods, e.g., IPv6-in-IPv6 or GRE are possible, see Section 5.2.1).

In detail, KIRA implements fast forwarding as follows:

* All paths of length 2 for a node's full 2-hop vicinity are discovered (as described in Section 3.4) and are then used for the precomputation of incoming and outgoing PathIDs, i.e., irrespective of their actual use. Longer paths to contacts are split into two path segments as follows: paths of lengths 4 and 5 hops have a second segment of 2 hops length and a first segment of length 2 or 3 hops respectively. Paths of length L6 hops have a second segment of 3 hops and a first segment of variable length L-3.

* The node creates forwarding table entries in the form of Incoming PathID -> (Outgoing PathID, Next Hop). The source route for the incoming PathID includes the own NodeID, whereas it is stripped off for computing the outgoing PathID. When forwarding to the last node of a path segment, the outgoing PathID is omitted.

* A node that wants to send a data packet (see example in Figure 3), sets the outgoing PathIDs of the source route path segments as destination addresses of the outer encapsulation headers (X uses H(A|Q|M|Z) as first segment and H(Z|C|E|T) as second segment in Figure 3) and sends it to its ULN. Source routes of less than 4 hops in length require only one PathID, the other two PathIDs. The source address of the outer header is set to the sender's NodeID so that errors can be reported back. The destination address of the inner most header is the destination NodeID.

* A node that receives a packet containing an incoming PathID tries to match it in its forwarding table. If it finds an entry, it rewrites the PathID with an outgoing PathID or removes the outermost PathID header in case the path segment ends at the next node. Including the own NodeID into the incoming PathID has the advantage of being more resilient against misrouted packets. If no entry is found, a corresponding Error is sent back, indicating a temporary inconsistency.

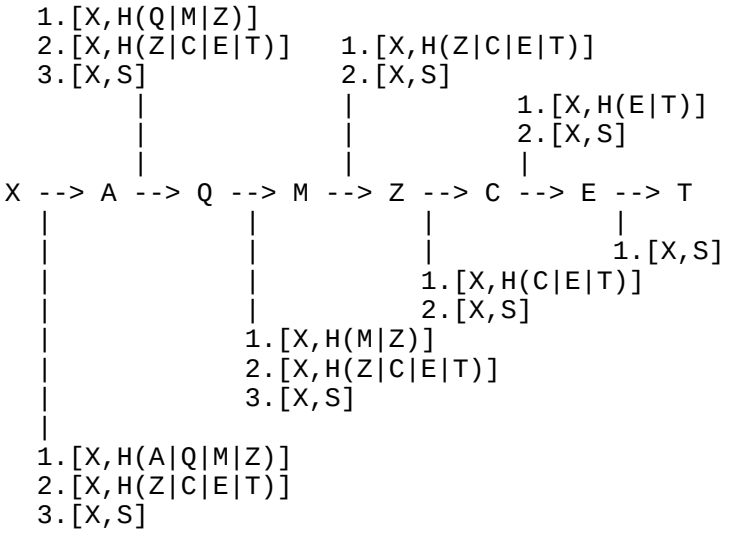

Figure 3: Example for a path of seven hops between X and contact T. X wants to send a packet to NodeID S and uses the path to its closest known contact T . The annotations above and below the arrows indicate up to three different packet headers with [source, destination] pairs, where 1. indicates the topmost header. X uses a PathID H(A|Q|M|Z) for the first segment and H(Z|C|E|T) for the second segment. Only node A must install a mapping H(A|Q|M|Z)-> H(Q|M|Z) and node Z must install mapping H(Z|C|E|T)-> H(C|E|T), all other mappings are precomputed.

Each node computes all PathIDs for its 2-hop vicinity to avoid path setup signaling, because it allows all nodes to assume that PathIDs exist for all source paths of length 3 hops. PathID precomputation for the full 2-hop vicinity provides a good trade-off between the number of a priori computed PathIDs and required path setup signaling. Intermediate nodes along a source route may not have computed the necessary PathIDs for others. Nodes explicitly setup paths via PathSetupReq only for paths >=5 hops. In the example of Figure 3, only nodes A and Z must install additional forwarding states when receiving a PathSetupReq, because all other nodes have

precomputed the PathIDs already. The PathSetupReq is answered with a PathSetupRsp by the node that marks the beginning of the second path segment. The forwarding states are implemented by soft states: contact probing also refreshes any required PathIDs in the intermediate nodes and so called "external" entries (i.e., those that are neither locally used nor precomputed) are deleted after three refresh intervals have passed without any refresh. Soft states are necessary, because paths are aggregated toward the same destination and usage relations are complex so it is nearly impossible to get a tracking completely right. For PathIDs that are used locally and not precomputed, a counter is kept that reflects the number of contacts using this PathID (note that different contacts may use the same first path segment).

The routing information from the Routing Tier is used by the Forwarding Tier to generate two forwarding tables inside each node: one based on the calculated PathIDs and one based on NodeIDs (generated from RT). One can employ common longest prefix matching for both tables. For NodeIDs the matching prefix length corresponds to the bucket depth. Thus, required prefix length is typically much shorter than the full length of the NodeIDs. The PathID forwarding table size comprises at least all stored contacts, but it is usually larger due to the number of precomputed and external entries.

3.11. Fast Next Hop Detour

The procedure described in Section 3.9.2 allows R/Kad to quickly come up with an alternative path without having to perform a full Path Rediscovery

It is possible to provide a fast fail-over in the data path by adding an additional PathID in case a next hop node (ULN) fails. A backup path for a ULN can be precalculated using the 2-hop vicinity (see Section 3.9.2). In case an incoming packet contains a PathID which uses a next hop that is invalid, an additional PathID (Next Hop Detour PathID) can be added "on top" to provide a two-hop detour for the next hop. Figure 4 provides an example: Node A receives a packet with PathID H(A|Q|M|Z) and knows that the link A -> Q failed. Therefore, Node A adds a PathID H(G|Q) before the outgoing PathID H(Q|M|Z), The new next hop G along the detour will remove the Next Hop Detour PathID H(G|Q) and forward the packet to the original next hop Q with the original PathID H(Q|M|Z). So the next PathID continues forwarding along the original path at the original next hop Q. So the overhead for the additional Next Hop Detour PathID exists for one hop only.

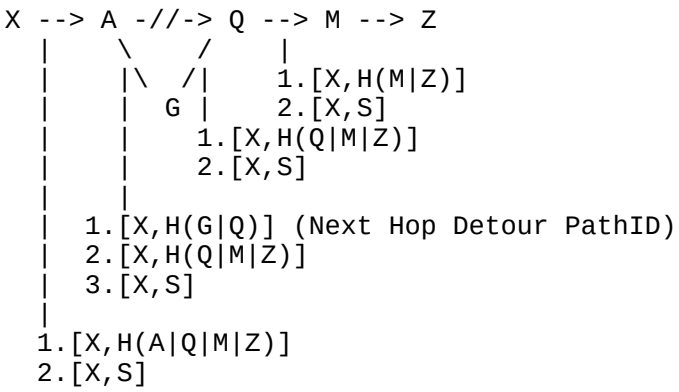

Figure 4: Example for fast next hop detour: the link between A and Q failed, so A inserts the fast next hop detour PathID H(G|Q) as first PathID, which is discarded by G when forwarded to Q.

The scheme is restricted to failed next hops only and merely considers detour paths that are two hops long. Forwarding loops are not possible in this case. Even if the link between G and Q has also failed, G may likewise use a fast next hop detour if one is available. However, the detour will always either end at Q or the packet will be dropped in case a detour is not possible.

3.12. End-system Mode

End-systems are only source or sink of data packets and do not forward them on behalf of others. We assume that end-systems are directly connected to at least one router, i.e., they can be multi-homed. Usually, the number of end-systems is vastly higher than the number of routers. Although it is straightforward to use the usual IPv6 subnetting approach with KIRA, one would lose the benefits of the ID-based approach w.r.t. mobility and multi-homing: as soon as an end-system changes its point of attachment, it would get a different address that belongs to the subnet of its new access router. Therefore, a special end-system extension of KIRA is provided that aims to reduce the control message overhead for end-systems even further while keeping the other benefits of KIRA's ID-based routing. Note that KIRA nodes can also limit their involvement into routing and forwarding by other means, e.g., by limiting the routing table size, independently of other nodes.

The extensions are quite minimal: some messages get an additional EndSystemFlag so that receiving nodes can identify the messages' origins as end-systems and treat them accordingly. The EndSystemFlag can be seen as a marker that changes the handling of end-system nodes in routing tables etc. Nodes marked as end-system are not used to

forward traffic to other nodes. This concerns both types of traffic: CP traffic as well as R/Kad control messages. Furthermore, end-systems are not reported to other nodes when routing information is disseminated in RTable objects. This means that they are not reported among routers, i.e., dissemination of routing information suppresses end-system related information.

End-systems can always initiate and send R/Kad messages due to the use of source routing. However, they can only be found and reached if their NodeID is known to their ID-wise closest neighbors that are routers. Therefore, end-systems ‘‘register’’ themselves at these routers by performing a join procedure (see Section 3.5) where the FindNodeReq contains a set EndSystemFlag. In case a message is destined for an end-system, ID-based routing will forward it to the ID-wise closest router that knows the end-system as contact due to its registration. In order to make sure that routers store all end-systems in their ID-wise neighborhood as contacts, an additional end-system only bucket is used for storing end-system contacts. This also eases to look up and return routing contacts while ignoring end-system contacts. The sum of router contacts and end-system contacts is considered when checking whether the bucket capacity k is reached and the bucket should be split further. Furthermore, router contacts are preferred in full buckets, i.e., they preempt end-system contacts. However, in networks with a high share of end-systems deepest buckets may consist of end-systems only. In this case these contacts cannot be preempted for a new end-system contact as they would not be reachable anymore by other nodes. Therefore, buckets that contain end-systems only, are not limited in their size. The expected number of end-systems a router is responsible for can be roughly estimated by 1/(1-p_es) where p_es is the share of end-systems w.r.t. all nodes, i.e., p_es=n_es/n_all with n_es being the number of end-systems and n_all being the number of all nodes. Note that this is independent of the network size n_all and for a share of 99% end-systems, each router needs to hold 100 end-systems in average only in its routing table.

End-systems MUST repeat the join procedure periodically in order to detect any change of the responsible routing node or path to it, since the routing nodes will not actively probe end-systems for connectivity or try to rediscover routes to them. Similarly, end-systems need to probe their neighbors regularly to detect broken paths.

4. Protocol Specification

This section defines the message syntax and node behavior for the R/Kad protocol.

4.1. Protocol Message Transport

R/Kad messages use the IPv6 packet format and are sent between KIRA nodes by using link-local addresses of the respective interfaces as source address and corresponding unicast or multicast addresses as destination address.

4.2. Protocol Encoding

An R/Kad message MUST be sent in the body of an IPv6 UDP datagram, with network-layer hop count set to 1, destined to the well-known KIRA multicast address or to an IPv6 link-local unicast address. Both the source and destination UDP port are set to a well-known port number. An R/Kad packet MUST be silently ignored unless its source port and destination is the well-known R/Kad port (to be allocated by IANA, use 19219 for experimentation). It MAY be silently ignored if its destination address is a global IPv6 address.

R/Kad messages consist of a common header and an optional sequence of type-length-value (TLV) encoded protocol objects. A single R/Kad message is limited in its size by the maximum length of an IPv6 payload minus the UDP header size of 8 bytes, because IPv6 fragmentation can be used between two R/Kad nodes. Larger message payloads can be transferred by using R/Kad fragmentation.

R/Kad messages use CBOR encoding [RFC8949] for the individual message fields. The lengths in the message specifications do not reflect the size after CBOR encoding on the wire. However, the final message after CBOR encoding must fit into the UDP payload (fragmentation of larger messages will be defined in later versions).

4.3. Protocol Message Notation

The protocol notation uses the Concise Data Definition Language (CDDL) defined in [RFC8610][RFC9682].

4.4. R/Kad Message Format

The overall message format consists of a common header that MUST be present in every message and a sequence of optional protocol objects that immediately follow the common header.

```
; R/Kad Message Format
r2kad-message = [
  header: common-header,        ; Common message header
  objects: [* protocol-object]  ; Array of protocol objects
]
```

Figure 5: Basic R/Kad Message Format

4.4.1. Common Message Header

The common message header has a fixed set of fields and a fixed size before CBOR encoding (the actual length on the wire may vary due to CBOR encoding). It is present in every R/Kad message.

```
; Common Message Header for R/Kad messages
common-header = [
  version: uint .size 1,          ; Version (8 bits)
  msg-type: uint .size 1,         ; Message Type (8 bits)
  flags: bstr .bits msgflags,     ; Flags (8 bits)
  msg-length: uint .size 2,       ; Message Length in bytes (16 bits)
  dest-id: nodeID-type,           ; Destination ID (112 bits)
  src-node-id: nodeID-type,       ; Source NodeID (112 bits)
  domain-id: bstr .size 8,        ; DomainID (64 bits)
  msg-id: bstr .size 8,           ; MessageID (64 bits)
  state-seq-num: uint .size 4,    ; State Sequence Number (32 bits)
  src-node-degree: uint .size 2   ; Source Node Degree (16 bits)
]

nodeID-type = bstr .size 14
```

Figure 6: Common Header Format

version: version is set to 0 for this specification.

msg-type: This field indicates the message type. Requests are odd numbers, Responses or Indications are even numbers.

```
msg-type = &amp;(
  ULNHello          : 0x01,
  ULNDiscoveryReq   : 0x03,
  ULNDiscoveryRsp   : 0x04,
  FindNodeReq       : 0x09,
  FindNodeRsp       : 0x0a,
  QueryRouteReq     : 0x0b,
  QueryRouteRsp     : 0x0c,
  UpdateRouteReq    : 0x11,
  ProbeReq          : 0x21,
  ProbeRsp          : 0x22,
  Error             : 0x70,
  PathSetupReq      : 0x81,
  PathSetupRsp      : 0x82,
  PathTearDownReq   : 0x83
)
```

```
flags:
   msgflags = &amp;amp;(
     ExactFlag      : 0,
     EndSystemFlag  : 1,
     Reserved       : 2..13,
     DiagnosticFlag : 14,
     Reserved       : 15
   )
```

   * _ExactFlag_ indicates whether the dest-id is a NodeID and is assumed to exist. If set to 1, the NodeID should exist, if set to 0, the node with the closest NodeID will process the request.

   * The _EndSystemFlag_ indicates that the originating source node is an end-system that does not perform routing or forwarding. See also Section 3.12.

   * _DiagnosticFlag_ triggers explicit Error Messages instead of dropping messages silently. This flag serves mainly debugging purposes. Verbose Error Messages may be used for amplification attacks. Therefore, a node MAY ignore this flag in case a sender sets it too often.

msg-length: This field specifies the length of the R/Kad Message in bytes including the Common Message Header.

dest-id: The NodeID of the final destination or a Resource ID for a lookup to find the responsible node for this ID. The special value "AllNodes NodeID" (see Section 7.3) is only allowed to be used ULNHello messages.

source-node-id: The NodeID of the originating node that created the message. Intermediate and receiving nodes do not modify the source-node-id.

domain-id: The DomainID 0x0 is the global domain of all KIRA nodes. By default every KIRA node is part of this domain. Other domains will have to be configured by an administrator or are constructed by a distributed cluster algorithm. Messages with unknown DomainIDs MUST be ignored. The DomainID is a selector for the corresponding RT, i.e., every domain maintains conceptually its own RT.

msg-id: The msg-id is chosen randomly for each request message and the responding node MUST copy it to the corresponding response message. It serves to uniquely map responses to related requests. In case a response message is received without a corresponding

msg-id of the open request, it SHOULD be silently discarded (unless the DiagnosticFlag is set, in this case an Error message with MessageIDUnknown SHOULD be sent back to the src-node-id); the error may be logged depending on local policy.

state-seq-num: This is the local state sequence number of the source node that originated the message. Within the node the sequence number is a global variable that changes with each underlay neighbor state change. That means, each time a new underlay neighbor is discovered or dismissed, the sequence number will be increased. 0 is an invalid sequence number and all state sequence numbers should start initially at 1. The sequence number space is only monotonically increasing, i.e., comparisons should be done without modulo wrap-around arithmetic. The value 0xffffffff signals a sequence number reset, i.e., a node receiving a 0xffffffff MUST initiate a resynchronization with this node by sending a ULNDiscoveryReq (for underlay neighbors), QueryRouteReq or ProbeReq to the corresponding node. Direct contact to a node (e.g., by receiving a ProbeRsp) will override and update its state sequence number, whereas indirectly heard state sequence numbers will not be accepted if they violate the monotonically increasing condition.

src-node-degree: This number describes the number of active KIRA interfaces where the KIRA instance sends ULNHello messages out and has discovered other KIRA nodes. Nodes that have more than 65535 interfaces simply use 65535 as maximum number. The value 0 is not allowed, since at least one interface must be present to sent this message.

4.4.2. Protocol Objects

Every Protocol Object starts with a common object header and has a specific content.

```
protocol-object = [
  common-object-header,
  [ ? object-contents ]
]
```

Figure 7: Protocol Object Format

4.4.2.1. common-object-header

The common-object-header contains the object type and the length of the following object. The object-length excludes the common-object-header, so a length of 0 indicates that no further object content follows.

```
common-object-header = [
  object-type : uint .size 1,
  object-length : uint .size 2
]

object-type = &amp;amp;(
   source-route-object-type        : 0x01,
   notvialist-object-type          : 0x02,
   contactlist-object-type         : 0x03,
   rtable-request-type-object-type : 0x04,
   rtable-object-type              : 0x05,
   rtable-update-info-object-type  : 0x06
)
```

Figure 8: Common Header Format

4.4.2.2. source-route-object

This object contains a source route in form of a list of NodeIDs and an index that points to the current NodeID when receiving and to the next NodeID when sending a message. The first NodeID (at index 0) is the src-node-id of the originating node.

```
source-route-object = [
  common-object-header,
  index : uint 0..1023,
  route : [+ nodeID-type]
]
```

Figure 9

In case index points behind the end of the list of present NodeIDs, a parameter problem error message MAY be sent back to the previous hop (see also Section 4.4.3.1 for node actions in erroneous situations). Typically, when forwarding an R/Kad message, the index pointer is advanced to the next entry in the route. In case the last node of the list received this object and the final destination has not been reached, it will append a path to the existing list that leads to the next overlay hop that is closer to the dest-id.

4.4.2.3. notvialist-object

```
notvialist-object = [
  common-object-header,
  failed-link-list: [+ link-list-type]
]

link-list-type = [
  src-node: nodeID-type,
  dst-node: nodeID-type,
  age-info: age-info-type
]

age-info-type = uint .size 4 ; age in ms
```

A failed-link-list is a sequence of NodeID pairs (src-node,dst-node) plus the age-info value. The latter specifies the age of this information in milliseconds. The maximum age that can be represented is large enough, because periodic updates usually refresh the corresponding information.

4.4.2.4. contactlist-object

```
contactlist-object = [
  common-object-header,
  contact-list: [+ contact-entry-type]
]

contact-entry-type = [
  contact-ID: nodeID-type,
  state-seq-num: uint .size 4,
  age-info: age-info-type,
  node-degree: uint .size 2
]
```

The list of contacts contains for each entry a NodeID, the corresponding known state-seq-num, an age-info that specifies how old the contact info is (time since last updated) and the known node-degree.

4.4.2.5. rtable-request-type-object

```
rtable-request-type-object = [
  common-object-header,
  rtable-request: rtable-request-type,
  radius: uint .size 1
]

rtable-request-type = &amp;(
  None                    : 0x00,
  ContactsOnly            : 0x01,
  OverlayNeighbors        : 0x02,
  OverlayNeighborsSource  : 0x03,
  ULNVicinity             : 0x04
)
```

The following rtable-request-type values can be used: _None_ will not return any Routing Table information. This is useful in case a FindNodeRsp should only report the source route back. _ContactsOnly_ reports only the ID-wise closest contacts to the dest-id of the FindNodeReq without their paths. _OverlayNeighbors_ reports the ID-wise closest contacts to the dest-id of the FindNodeReq including their path vectors. _OverlayNeighborsSource_ reports the ID-wise closest contacts to the source NodeID of the FindNodeReq. _ULNVicinity_ requests the underlay neighbors within the given radius. Radius specifies the number of entries to be returned and SHOULD be set to the bucket size k by default. A value of 0xff means to return the full routing table (for any request type other than _None_).

4.4.2.6. rtable-object

```
rtable-object = [
  common-object-header,
  rtable-length: uint .size 2,
  rtable-entries: [+ rtable-entry-type]
]

rtable-entry-type = [
  contact-ID: nodeID-type,
  path: path-vector-type,
  state-seq-num: uint .size 4,
  age-info: age-info-type,
  node-degree: uint .size 2
  [? node-attributes],
  [? path-attributes],
  [? link-attributes]
]

path-vector-type = [
  path-length: uint .size 2,
  path-vector: [* nodeID-type]
]
```

The rtable-entries contains a list of routing table entries. The list is preceded by rtable-length that provides the number of the following entries. For each entry the NodeID of the contact is given (contact-ID), the path from the reporting node to the contact-ID as sequence of NodeIDs as well as the corresponding known state-seq-num, an age-info that specifies how old the contact info is (time since last updated) and the known node-degree. Optional attributes for the node (node-attributes), the path (path-attributes) or individual links (link-attributes) along the path may follow. The path-vector-type is basically a counter that specifies the number of the immediately following node IDs.

4.4.2.6.1. rtable-update-info-object

```
rtable-update-info-object = [
  common-object-header,
  rtable-length: uint .size 2,
  rtable-update-entries : [+ rtable-update-entry-type]
]

rtable-update-entry-type = [
  contact-ID: nodeID-type,
  path: path-vector-type,
  state-seq-num: uint .size 4,
  age-info: age-info-type,
  node-degree: uint .size 2,
  route-update-action: route-update-action-type,
  [? node-attributes],
  [? path-attributes],
  [? link-attributes]
]

route-update-action-type = &amp;(
            announce    : 0x00,
            withdraw    : 0x01,
            change      : 0x02,
            unreachable : 0x03
)
```

The rtable-update-info object is similar to the rtable-object (Section 4.4.2.6) and contains a list of routing table entries with an associated route-update-action. Value "announce" means that the corresponding contact is a new entry in the routing table. Value "withdraw" means that the contact has been deleted from the routing table. Value "change" means that the path to the contact has been changed. Value "unreachable" means that the contact is not reachable.

Note: further objects will be detailed in future versions of this draft

4.4.3. R/Kad Messages

This section describes the R/Kad messages and what KIRA nodes do when sending or receiving those messages.

4.4.3.1. General Node Behavior

There are some actions that are performed in the same way for all messages.

* Most R/Kad control messages are sent with a rate limit to avoid overloading other nodes with high bursts of messages.

* In case of having problems to process a message (e.g., due to malformed messages, unknown objects, and so on) the message SHOULD be silently discarded. If the DiagnosticFlag is set, a corresponding Error message SHOULD be sent back to the src-node-id. The error may be logged locally on the node depending on its policy. An Error message MUST never be sent back for in incoming Error message.

* In order to avoid undesired synchronization effects, many actions are performed after a randomized waiting time. This specification uses the notion RandTime(T_p) for clarifying that the actual waiting time T_a is chosen randomly at every use from the Interval [0.5*T_p,1.5*T_p], so the mean waiting time is T_p.

* Response messages MUST copy the msg-id of the related request message into their header. In case a response message is received without a corresponding msg-id of the corresponding request, it SHOULD be silently discarded (unless the DiagnosticFlag is set, in this case an Error message with MessageIDUnknown SHOULD be sent back to the src-node-id); the error MAY be logged depending on local policy.

4.4.3.2. ULNHello Message

ULNHello messages are periodically sent (randomized with RandTime()) to each interface to indicate presence of a KIRA node. Its format is shown in Figure 10.

```
ULNHello = [
  header: common-header
]
```

Figure 10: ULNHello Message

4.4.3.2.1. Sending a ULNHello

The sender uses the Undefined NodeID as dest-id and its own NodeID as src-node-id. Other nodes that want to establish an adjacency SHOULD respond with a ULNDiscoveryReq message after a randomized waiting time and if the trigger condition is met. The trigger condition is calculated as follows: use the lowest 32 bit of the other NodeID and the own NodeID and calculate delta = otherNodeID mod 2^32 - ownNodeID mod 2^32. If (delta < 0x80000000 and delta != 0) or ((delta == 0 or delta == 0x80000000) and ownNodeId < otherNodeId) then trigger a ULNDiscoveryReq, otherwise wait for a ULNDiscoveryReq. This

heuristic avoids initiating the ULN Discovery handshake twice in case the ULNHello messages from both sides overlap time-wise.

At node startup or when a new link comes up RandTime(ULNHelloMinInterval) is used per link. The sending interval for subsequent ULNHello messages is doubled up to ULNHelloMaxInterval. The sending interval is reset to ULNHelloMinInterval for a link that comes up after it failed. Default values for fixed network links (e.g., Ethernet) are: ULNHelloMinInterval = 200ms, ULNHelloMaxInterval = 30s. It is assumed that these types of link layers can indicate a loss of the link layer connections. This is not true for most wireless link layers, therefore suggested default values for wirless links (e.g., WiFi or cellular networks) are: ULNHelloMinInterval = 100ms, ULNHelloMaxInterval = 1s.

4.4.3.2.2. Receiving a ULNHello

On receipt of a ULNHello Message the receiving node should check whether the originating source node is already known as contact. If it is already member of a ULN bucket and contained in the ULNTable, the LastSeen timestamp is updated and the provided sequence number state-seq-num is checked against the stored Sync ULN Sequence Number of the contact. If the provided sequence number is newer, a ULNDiscoveryReq to the source node SHOULD be scheduled to discover recent changes in the underlay neighborhood of the source node (and therefore the receiver’s 2-hop vicinity). If the source node is not known as contact yet, either a ULNDiscoveryReq will be triggered locally or will be triggered by the source node after the node’s own ULNHello was received. In the latter case (i.e., when the trigger condition above is not met) it should be checked, when the next ULNHello will be sent: if this period is longer than 5*ULNHelloMinInterval, a ULNDiscoveryReq SHOULD be sent immediately to avoid longer ULN Discovery periods.

4.4.3.3. ULNDiscovery Request Message (ULNDiscoveryReq)

A ULNDiscoveryReq is either sent as response to a ULNHello message or sent to test liveliness and bidirectional connectivity of an already known ULN or to resynchronize state with a ULN. Its format is shown in Figure 11.

```
ULNDiscovery-Request-Message = [
  header: common-header,
  uln-list : [* contactlist-object ]
]
```

Figure 11: ULNDiscovery Request Message

4.4.3.3.1. Sending a ULNDiscoveryReq

The sending node fills its currently known ULNs into the uln-list on first contact or each time its state sequence number has changed. It expects a ULNDiscoveryRsp as immediate reply and should set a timer as maximum waiting period (ULNDiscoveryRspInitialMaxWaitTime, default 200ms). After the corresponding timeout, the ULNDiscoveryReq should be repeated. The timeout for an answer should be doubled for each retry. The contact should be considered dead after two unsuccessful retry attempts.

4.4.3.3.2. Receiving a ULNDiscoveryReq

On receipt of a ULNDiscoveryReq, the receiving node MUST reply with a ULNDiscoveryRsp. If the contact was not known yet, it is put into the ULNTable and into the corresponding ULN bucket. In case the contact was already known, but not as direct underlay neighbor, the contact should be moved to the ULN bucket. This can be the case if the contact has been learned being an underlay neighbor of another ULN. The UnderlayNeighbor flag of the contact must be set to true.

4.4.3.4. ULNDiscovery Response Message (ULNDiscoveryRsp)

A ULNDiscoveryRsp is sent as answer to a previous ULNDiscoveryReq. Its format is shown in Figure 12.

```
ULNDiscovery-Response-Message = [
  header: common-header,
  uln-list : [* contactlist-object ]
]
```

Figure 12: ULNDiscovery Response Message

4.4.3.4.1. Sending a ULNDiscoveryRsp

The sending node MUST copy the msg-id from the ULNDiscoveryReq and fills its currently known ULNs into the uln-list on first contact or each time its state sequence number has changed.

4.4.3.4.2. Receiving a ULNDiscoveryRsp

Conceptually, on receipt of a ULNDiscoveryRsp bidirectional connectivity to the underlay neighbor has been demonstrated. The state of the contact should be changed to "valid", LastSeen, Sync ULN Sequence Number and Sequence Number should be updated accordingly. The ULNList should be parsed for new contacts or contact updates with respect to newer state-seq-num. In case of a new contact or its newer state-seq-num a QueryRouteReq SHOULD be sent to the

corresponding contact. The latter should request a ULNVicinity (rtable-request-type) of Radius 1 to learn the 2-hop underlay vicinity.

4.4.3.5. FindNode Request Message (FindNodeReq)

FindNodeReq messages are used (together with the corresponding FindNodeRsp) to find routes to nodes (Path Discovery, see Section 3.6), to improve the own routing table, or to find responsible nodes for a given key. Its message format is shown in Figure 13.

```
FindNode-Request-Message = [
  header: common-header,
  rtable-request: rtable-request-type-object,
  source-route: source-route-object,
  notvia: [? notvialist-object]
]
```

Figure 13: FindNode Request Message

4.4.3.5.1. Sending a FindNodeReq

Sending a FindNodeReq for an existing NodeID X (e.g., communicated by other nodes as part of their rtable-object) MUST set the ExactFlag in the flags field of the common-header. The rtable-request is typically set to OverlayNeighbors (depending on the purpose of the FindNodeReq). The dest-id is set to the NodeID X. In case X is a known contact, a source route to X is known and filled into the source-route. In case X is not a known contact, the routing table is used to look up the ID-wise closest known valid contact Y (using proximity routing, see Section 3.3) and the node fills in the corresponding path into the source-route object. The own NodeID is used as first element of the source-route and the contact's NodeID Y is the last element of the source-route (the path vector is the sequence of NodeIDs in between both entries). The index of the source-route object MUST be set to 1. In case of known failed links (esp. during Path Rediscovery, Section 3.9.1), the notvia object is also filled correspondingly.

The sender of the FindNodeReq looks up the underlay address of the next hop in the source route (using the ULNTable) and sends the message to the underlay neighbor. For the Join procedure (Section 3.5), the dest-id is set to the source-node-id of the sending node.

Depending on the context, the FindNodeReq may be repeated in case the corresponding FindNodeRsp is not received within 500ms. Further retries should double the timeout. A failure should be assumed after 2 unsuccessful retries.

4.4.3.5.2. Receiving a FindNodeReq

A node that receives a FindNodeReq checks at first whether it has the NodeID at the current index of the source route. If the NodeID at the current index is different from the node's NodeID, the message has been misrouted. This is a severe error and SHOULD be logged at least locally. In case of a set DiagnosticFlag an Error to the previous node.

An node that is an intermediate node on the source route should evaluate the notvia object first and invalidate any affected contacts (depending on the corresponding age values). Then the so far traversed path (in reverse direction) should be analyzed for interesting contacts or paths. Since the message just traveled along this path, one can assume that the path information is current and thus validated.

If the receiving node is neither the last node in the source route nor the destination node according to the dest-id, the FindNodeReq message is forwarded along the source route: the index is incremented by one and a lookup in the ULNTable is performed for this next hop NodeID. In case the next hop is missing in the ULNTable (e.g., failed link) there are two more options possible. The first option is used in case the next hop is known as contact and there is a valid route leading to it. The current source route is then adapted by inserting the detour path to the former next hop and continuing to forward to the new next hop. The second option is used when the dest-id is a known contact with a valid route. In this case the rest of the source route is replaced by the known path to the destination NodeID. If both options are not successful, an Error message of type SegmentFailure is sent back to the source along the reversed source route. The NodeID of the failed next hop node and the dest-id are provided as additional parameters in the SegmentFailure Error message.

If the receiving node (e.g., having NodeID Y) is the last node in the source route, it should look up the ID-wise closest contact (e.g., lets say NodeID Z) in its routing table (in case there are multiple equivalent choices, proximity routing is used as described in Section 3.3). If the XOR distance $d(Y,Z) < d(Y,X)$, the next contact Z leads closer to the destination ID X and the corresponding source route to Z is appended to the existing source route. In case Z is not closer to destination ID X than the current node Y, the behavior

depends on the ExactFlag: if the ExactFlag was set, an Error message is sent back indicating a RouteFailureDeadEnd, since there is no known overlay path leading to X. If the ExactFlag is not set, a FindNodeRsp message is sent back, containing the requested information according to the rtable-request.

If the NodeID of the recipient corresponds to dest-id, the rest of the source route should be ignored and a corresponding FindNodeRsp should be sent.

4.4.3.6. FindNode Response Message (FindNodeRsp)

A FindNodeRsp message is sent as response to a FindNodeReq either if the node with the dest-id has been reached or the FindNodeReq cannot be forwarded ID-wise closer to the dest-id when the ExactFlag is set. Its message format is shown in Figure 14.

```
FindNode-Response-Message = [
  header: common-header,
  source-route: source-route-object,
  notvia: [? notvialist-object],
  rtable: [? rtable-object]
]
```

Figure 14: FindNode Response Message

4.4.3.6.1. Sending a FindNodeRsp

The source route contained in the FindNodeReq is reversed and used as new source-route, any cycles MUST be removed. Since all links of this possibly shortened path haven been traversed by the FindNodeReq, the probability to have a working source route back to the source of the FindNodeReq is very high. The src-node-id is set to the NodeID of the responding node, the dest-id is set to the last NodeID of the just generated source-route. The notvia list from the FindNodeReq should be copied into the FindNodeRsp. The rtable object contains the requested information according to the rtable-request of the FindNodeReq. In addition to the requested contacts, additional gratuitous contacts SHOULD be provided as follows: two randomly chosen contacts from every bucket that are not already contained in the list of request contacts.

4.4.3.6.2. Receiving a FindNodeRsp

The recipient of a FindNodeRsp Message processes the notvia list first if present, then analyzes the source-route for new contacts or path information. The receiving node processes the rtable object by analyzing each given contact and related path vector information in order to improve its own routing table. Received path vectors should be inspected for further improvement with the node's valid paths. For example, if node X learns path A, Q, M to Z, it can shorten the path by using the path vector B to contact M, resulting in an improved path of B, M to Z. However, this path needs to be validated by probing this proposed path before it can be used as active and valid path.

4.4.3.7. QueryRoute Request Message (QueryRouteReq)

A QueryRouteReq message is used to receive requested routing information from other nodes. This is especially used to discover the 3-hop underlay neighborhood as described in Section 3.4. In this case a QueryRouteReq using rtable-request ULNVicinity is sent to 2-hop underlay neighbors. However, QueryRouteReq may be sent to any contact for improving a node's own RT. QueryRouteReq messages require a known source route to the destination, i.e., in contrast to FindNodeReq messages they are not forwarded closer to the dest-id. The message format is shown in Figure 15.

```
QueryRoute-Request-Message = [
  header: common-header,
  rtable-request: rtable-request-type-object,
  source-route: source-route-object,
  notvia: [? notvialist-object]
]
```

Figure 15: QueryRoute Request Message

4.4.3.7.1. Sending a QueryRouteReq

The sender of a QueryRouteReq message MUST set the ExactFlag, use the NodeID as dest-id and fill in the source route accordingly. QueryRouteReq messages can also be sent to nodes that are not known as contacts (i.e., not part of the RT), however, a valid path must be known to the destination node that can be used as source route. The rtable-request should be set according to the sending node's need. A notvia list is optionally provided.

4.4.3.7.2. Receiving a QueryRouteReq

The receiver of a QueryRouteReq message MUST inspect the source route for contact or route improvements and check whether it is the destination of the QueryRouteReq. In this case a QueryRouteRsp message MUST be sent back. Otherwise the QueryRouteReq MUST be forwarded to the next hop along the source route.

4.4.3.8. QueryRoute Response Message (QueryRouteRsp)

The QueryRouteRsp is a response to a QueryRouteReq and provides RT information of the sending node. The message format is shown in Figure 16.

```
QueryRoute-Response-Message = [
  header: common-header,
  source-route: source-route-object,
  notvia: [? notvialist-object],
  rtable: [? rtable-object]
]
```

Figure 16: QueryRoute Response Message

4.4.3.8.1. Sending a QueryRouteRsp

The sender MUST respond to a received QueryRouteReq with a corresponding QueryRouteRsp. The rtable object is filled according to the rtable-request of the QueryRouteReq. Gratuitous contacts SHOULD be added, too.

4.4.3.8.2. Receiving a QueryRouteRsp

The processing of a QueryRouteRsp is analogous to processing a FindNodeRsp.

4.4.3.9. UpdateRoute Request Message (UpdateRouteReq)

An UpdateRouteReq message provides information about changed routing information, e.g., new or removed contacts as well as changed paths. As the UpdateRouteReq message has informational character only, it is transmitted unreliably, i.e., its receipt is not confirmed. Consequently, there is no corresponding UpdateRouteRsp message. The message format is shown in Figure 17.

```
UpdateRoute-Request-Message = [
  header: common-header,
  source-route: source-route-object,
  notvia: [? notvialist-object],
  rtable-update: rtable-update-info-object
]
```

Figure 17: UpdateRoute Request Message

4.4.3.9.1. Sending an UpdateRouteReq

An UpdateRouteReq is sent for newly learned contacts (Announce), removed (WithDraw) or failed direct underlay contacts (Unreachable) as well as for changed paths (Change). UpdateRouteReq messages are scheduled when the local RT of a node is changed. This is useful to avoid sending too frequent update messages while the network is still converging. A path to a contact may possibly change multiple times in a short time frame or the modified contact may be preempted by another contact later. Therefore, consistency has to be checked at the time when the UpdateRouteReq should be sent, e.g., if a path change occurred first, but afterwards the contact was deleted, then a different type of UpdateRouteReq needs to be sent.

An UpdateRouteReq is sent to the four ID-wise closest contacts of the sending node. These destination contacts do not have to be valid contacts (i.e., possess a currently known valid active path) as UpdateRouteReq messages are forwarded via the overlay routing as close to the destination as possible. It is useful to create a list of RT updates per destination contact so that multiple updates can be aggregated into the same UpdateRouteReq message. Furthermore, path changes may occur several times during convergence and only the latest change should be reported. Moreover, updates are sent depending on their criticality: urgent updates are sent for Unreachable contacts after RandTime(250ms), other updates are sent after RandTime(500ms).

4.4.3.9.2. Receiving an UpdateRouteReq

A node that receives an UpdateRouteReq (it is at the end of the current source route) and that is not the final destination (its NodeID is equals to the dest-id), tries to forward the UpdateRouteReq closer toward the dest-id. If this is not possible, forwarding stops at this node (without triggering any Error message, e.g., RouteFailureDeadEnd due to this fact; other errors during message processing may trigger Error messages if the DiagnosticFlag is set). Due to overhearing (see Section 3.7) the rtable-update object is processed nevertheless. A node that is the final destination of the UpdateRouteReq simply processes the information provided in the

rtable-update object. No response will be created.

4.4.3.10. Probe Request Message (ProbeReq)

ProbeReq messages are sent to test liveliness of a path and contact. There are two processes that trigger sending of ProbeReq messages: Periodic Path Probing (Section 3.8) and checking validity of proposed paths (Section 3.9.3). Periodic Path Probing also refreshes any related PathIDs for the particular path in the Forwarding Tier as this information will otherwise be deleted after some time without refreshes (see Section 3.10). Therefore, ProbeReq messages need to travel the _complete_ path until the dest-id (unless this is impossible due to a failure and an Error message is sent back). ProbeReq messages SHOULD never be sent to ULNs if their direct attached link is working as there are periodic ULNHello messages used to check the connectivity.

ProbeReq messages are strictly following the specified source route, i.e., rerouting them is not allowed. A broken link will be reported by an Error message indicating a SegmentFailure. If the path is intact, a ProbeRsp will be sent back along the reverse source route by the contact. The ProbeReq messages are sent via the Routing Tier, i.e., they are not testing the actual data path in the Forwarding Tier. The message format of the ProbeReq is shown in Figure 18.

```
Probe-Request-Message = [
  header: common-header,
  source-route: source-route-object
]
```

Figure 18: Probe Request Message

4.4.3.10.1. Sending a ProbeReq

The sender uses the contact’s NodeID as dest-id, sets its own NodeID as src-node-id and fills in the source route. The ExactFlag MUST be set and it is assumed that the dest-id is a NodeID of an existing contact. The last node of the source route must correspond to the dest-id as ProbeReq message are not forwarded via overlay routing, but via strict source routing. Therefore, the source route must be complete, so that it leads to the dest-id. Retransmitting the ProbeReq in case of a missing matching ProbeRsp is not necessary for Periodic Path Probing as this will automatically send another ProbeReq message after a while.

4.4.3.10.2. Receiving a ProbeReq

An intermediate node (i.e., neither the first nor the last node) along the source route MUST check whether the corresponding PathID information is present in the Forwarding Tier and either install it or update its timestamp if already present. Then the ProbeReq MUST be forwarded to the next hop in the source route if any is left. If is not possible due to a failed link to the next hop or the next hop node failed, a SegmentFailure Error message MUST be sent back to the origin of the ProbeReq (along the reversed source route that was already traversed). This error reporting must not be suppressed as path liveliness check and connectivity detection of KIRA depends on it. An intermediate node MUST NOT send back a ProbeRsp. In case the node is the last node in the source route, but the dest-id is not the NodeID of the node, the source route is wrong or incomplete. In this case an Error message RouteFailureWrongPath MAY be sent back if the Diagnostic Flag is set.

The destination node of a ProbeReq updates its local contacts and paths according to the source route. It sends back a ProbeRsp message along the reversed source route.

4.4.3.11. Probe Response Message (ProbeRsp)

A ProbeRsp message MUST be sent as response to an incoming ProbeReq if the KIRA NodeID is the dest-id of the ProbeReq. The message format is shown in Figure 19.

```
Probe-Response-Message = [
  header: common-header,
  source-route: source-route-object
]
```

Figure 19: Probe Response Message

4.4.3.11.1. Sending a ProbeRsp

The node that was the destination of the ProbeReq sends back a ProbeRsp message along the reversed source route and sets the dest-id to the src-node-id of the ProbeReq.

4.4.3.11.2. Receiving a ProbeRsp

The node that originally sent the corresponding ProbeReq updates the contact state information accordingly (State Sequence Number, LastSeen, Validation and LastRefresh timestamps of the corresponding path).

4.4.3.12. Error Message (Error)

Error messages are sent back to indicate problems and for diagnostic purposes. A conservative reaction to errors during message processing is typically to drop the erroneous message silently and not sent back any feedback. This also reduces possibilities for potential amplification attacks. However, some Error messages must be sent for proper protocol operation, e.g., returning a SegmentFailure is essential for detecting broken paths. The DiagnosticFlag in the Common Header solicits the transmission of Error messages for debugging purposes. However, it is at a node's discretion whether or how often this request is fulfilled (e.g., it can apply rate limits). The format of an Error message is shown in Figure 20.

```
Error-Message = [
  header: common-header,
  source-route: source-route-object,
  error: error-type,
  origin-msg-id: bstr .size 8,
  additional-error-info: bstr
]

error-type = &amp;amp;(
  NoError               = 0x00,
  NodeUnreachable       = 0x01,
  MalformedMessage      = 0x02,
  ParameterProblem      = 0x03,
  HopLimitExceeded      = 0x04,
  SegmentFailure        = 0x05,
  PathIDUnknown         = 0x06,
  MessageIDUnknown      = 0x07,
  RouteFailureDeadEnd   = 0x0a,
  RouteFailureWrongHop  = 0x0b,
  RouteFailureWrongPath = 0x0c
)
```

Figure 20: Error Message

4.4.3.12.1. Sending an Error message

During processing protocol messages various error conditions may occur that can produce an Error message. The Error Type must be set according to the error condition that is given in this specification. The origin-msg-id is set to the MessageID of the message that caused the problem. Optional addition error information may be given, too. (TBD in future versions of this specification).

4.4.3.12.2. Receiving an Error message

Depending on the type of the Error message, the receiving node may trigger certain protocol actions, e.g., a starting a Path Rediscovery after receiving a SegmentFailure or silently discard the Error message after optionally generating a log message (possibly rate limited) with the information given in the Error message. An error occurring during processing the Error message MUST never result in sending an Error message back.

4.4.3.13. Path Setup Request Message (PathSetupReq)

A PathSetupReq is sent to install required PathID mappings in intermediate nodes (see Section 3.10). This is only necessary for paths to contacts that are at least 6 hops long. Its format is shown in Figure 21.

```
Path-Setup-Request-Message = [
  header: common-header,
  source-route: source-route-object
]
```

Figure 21: Path Setup Request Message

4.4.3.13.1. Sending a PathSetupReq

The originating node simply constructs the source route according to the path of the contact in question. The dest-id is set to the NodeID of the contact, the src-node-id is set to the NodeID of the sender. Typically, a PathSetupReq is scheduled when a new contact is put into a bucket. When the time has come to send the PathSetupReq a check should ensure that the current path of the contact still requires the PathSetupReq (meanwhile a shorter path may have been learned or the contact may have been preempted).

4.4.3.13.2. Receiving a PathSetupReq

A node that receives a PathSetupReq (also intermediate nodes) needs to check whether it needs to install PathIDs. PathIDs that are installed by a PathSetupReq get an "external entry" flag, indicating that other nodes require this path. If the necessary PathID is installed already, the "external entry" flag is set and a PathSetupRsp MUST be sent back. Otherwise the corresponding PathID mapping (incoming PathID to outgoing PathID and next hop) is installed and the PathSetupReq is forwarded to the next node. If there are only 3 hops left to the destination, a PathSetupRsp MUST be sent back, because of path precomputation in the 2-hop vicinity the next hop must have precomputed the PathID.

4.4.3.14. Path Setup Response Message (PathSetupRsp)

A PathSetupRsp message confirms that the necessary PathIDs are installed for the source route. The message format is shown in Figure 22.

```
Path-Setup-Response-Message = [
  header: common-header,
  source-route: source-route-object
]
```

Figure 22: Path Setup Response Message

4.4.3.14.1. Sending a PathSetupRsp

A PathSetupRsp is sent back as response to a PathSetupReq if the necessary PathIDs are already present in the node or have been installed and PathIDs do not have to be installed in further nodes along the path.

4.4.3.14.2. Receiving a PathSetupRsp

The node that receives a PathSetupRsp and that is not the dest-id of the PathSetupRsp simply forwards the PathSetupRsp forward to the dest-id.

4.4.3.15. Path TearDown Request Message (PathTearDownReq)

The PathTearDownReq initiates deletion of the PathID state information along the specified source route. This mechanism basically provides means a of optimization therefore a response message is not required. The format of the PathTearDownReq is shown in Figure 23.

```
Path-Setup-TearDown-Message = [
  header: common-header,
  source-route: source-route-object
]
```

Figure 23: Path TearDown Request Message

4.4.3.15.1. Sending a PathTearDownReq

A PathTearDownReq SHOULD be initiated if a contact is removed that had a path that required a PathSetupReq.

4.4.3.15.2. Receiving a PathTearDownReq

An (intermediate) node that receives a PathTearDownReq needs to check whether the corresponding PathIDs needs to be deleted. Normally, this clears an "external entry" flag of the corresponding entry. The flag will be set again in case another node requires this path by the next periodical refresh. This depends also on whether the PathID is required by local contacts or precomputed. Precomputed PathIDs will never be deleted by a PathTearDownReq, locally required PathIDs can be deleted if no contact uses this path anymore and it is neither precomputed nor possesses a foreign flag. Similarly to the PathSetupReq, a PathTearDownReq will not be forwarded if there are only 3 hops left to the destination.

5. Forwarding Tier Functionality

As described in Section 3.10 PathIDs are used to replace the source routes that are used in R/Kad messages in order to reduce the per packet overhead for the data packets (these are packets sent between applications in the control plane). This section describes encapsulation methods for the data packets and the required functionality that is required to forward the packets. Encapsulation is required, because the original data packet requires the end-to-end NodeIDs as source and destination IPv6 addresses.

KIRA uses up to two PathIDs per packet: one PathID per path segment. PathIDs are not required if packets are forwarded between directly adjacent ULNs. In this case the data packet simply uses a normal IPv6 header with the source NodeID and destination NodeID as IPv6 addresses. In case Fast Next Hop Detour (see Section 3.11) is used an additional PathID is added for only one hop.

PathIDs MUST use a different 16-bit IPv6 prefix than NodeIDs so that forwarding rules can clearly distinguish between NodeIDs and PathIDs.

5.1. Node Requirements

Intermediate KIRA nodes MUST be able to

* perform a longest prefix match or full match of IPv6 addresses
* support regular IPv6 routing/forwarding tables
* replace/overwrite a destination IPv6 address (for PathID swapping)
* decapsulate packets (remove outer headers)

* handle the encapsulation headers, e.g., a Segment Routing Header [RFC8754]

Sending KIRA nodes MUST be able to create IPv6 packets with encapsulation. If Fast Next Hop Detour (see Section 3.11) is supported, at most three PathIDs are required (i.e., up to three encapsulation headers depending on the encapsulation method). Otherwise two PathIDs are sufficient. This influences the MTU (Maximum Transmission Unit).

5.2. Encapsulation Formats

There are three proposed encapsulation formats as candidates for further discussion. They are: SRv6 [RFC8754], IPv6 in IPv6 [RFC2473], and GRE [RFC7676].

5.2.1. SRv6 Encapsulation

SRv6 would be the option with the lowest overhead among the listed options. The Segment Routing Header (SRH) [RFC8754] is an IPv6 extension header that would add an overhead of at most 24 bytes (without any additional SRH TLV objects) without Fast Next Hop Detour and 40 bytes with the possibility of using a Fast Next Hop Detour. The overall overhead with the outer IPv6 header is then adding up to 64 bytes (or 80 bytes respectively). In case the SRH is not needed, the outer IPv6 header adds 40 bytes of overhead. KIRA interfaces should consider the reduced MTU size of 64 bytes (or 80 bytes respectively).

Figure 24 shows a coarse layout of an outer IPv6 header with an SRH as extension header as well as the payload of the outer packet that encapsulates the end-to-end IPv6 packet. The SRH SHOULD be left out if there is only a single path segment (also allowed by [RFC8754]). The SRH therefore is only necessary if two path segments are required (that may be the case starting with paths of 4 hops in length). Furthermore, the SRH SHOULD be a Reduced SRH as specified in Section 4.1.1 of [RFC8754]. However, a reduced SRH cannot be used if SRH TLVs are needed.

The inner packet contains an IPv6 header that uses the source NodeID (Src-NodeID) of the originating KIRA node and the destination NodeID (Dst-NodeID) of the final destination of the packet. The outer IPv6 header carries the NodeID of the most recent overlay hop that created the encapsulation as IPv6 source address. That means Var-NodeID is initially the same as the Src-NodeID, but is then changed at the next overlay hop.

The destination address of the outer header is initially the PathID of the first path segment, but it is replaced in-situ with a new value at every intermediate node that forwards this packet, until both path segments have been traversed. At the last node of the first segment, the outer header is dropped and a new header will be prepended for the subsequent path segments to the following next overlay hop (if any).

At the end of the first path segment, the first element of the segment list of the SRH is copied to the destination IPv6 address of the outer header just as an ordinary second segment of any SRv6 would.

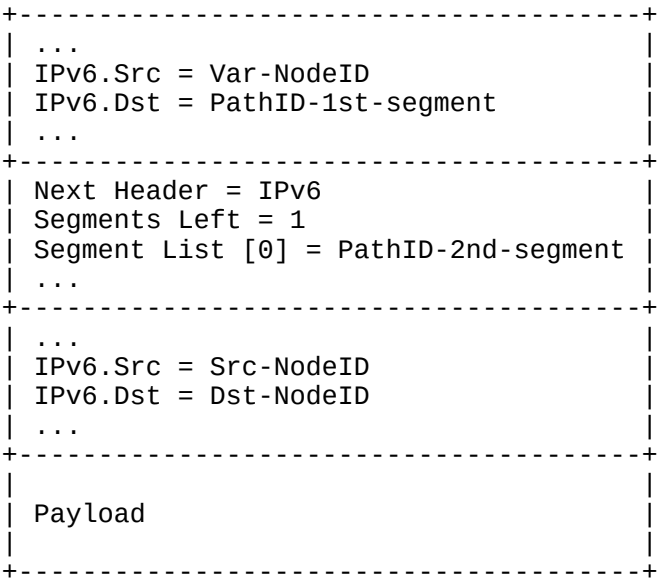

Figure 24: Reduced SRH encapsulation for forwarding KIRA data packets in the forwarding tier

5.2.2. IPv6-in-IPv6 Encapsulation

The IPv6-in-IPv6 simply uses an IPv6 header for every path segment. The outermost headers (up to three if Fast Next Hop Detour is used) contain the NodeID of the most recent overlay hop that created the encapsulation as IPv6 source address and the PathID for the corresponding path segment as IPv6 destination address.

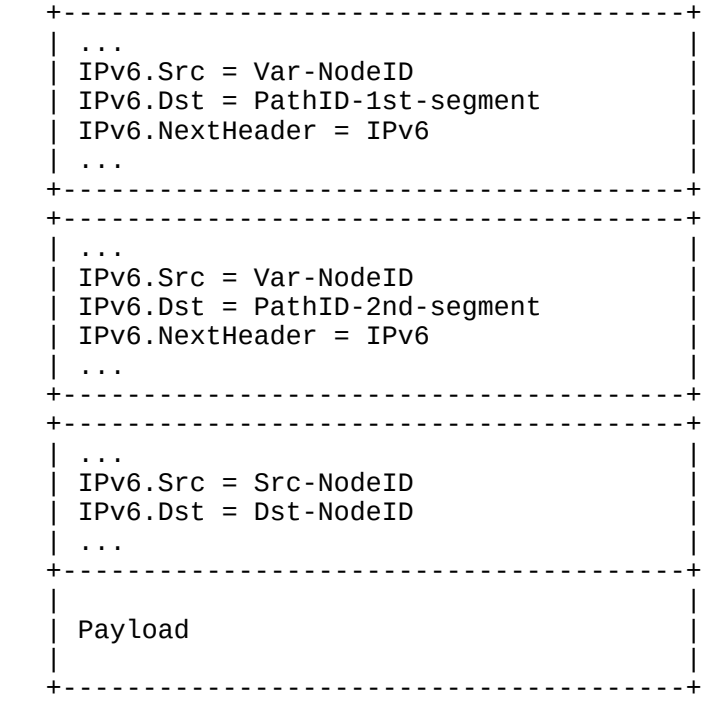

Figure 25: IPv6-in-IPv6 encapsulation for forwarding KIRA data packets in the forwarding tier

5.2.3. GRE Encapsulation

GRE encapsulation [RFC7676] is less efficient than the previously described encapsulations, but it is usually widely supported.

For GRE encapsulation the delivery header is IPv6 with a PathID as destination address and the payload is also IPv6. In case of a second path segment present, the payload may contain another GRE packet with the PathID of the second segment as destination of the delivery header. In case a Fast Next Hop Detour is used, an additional GRE header may be added at the top.

5.3. DomainID Integration

A DomainID acts as selector for the forwarding/routing tables while forwarding KIRA packets. How DomainIDs can be integrated into the forwarding tier is for further study. One possibility is to use an SRH TLV that defines the DomainID. However, the DomainID needs also to be visible for the innermost header that contains NodeIDs.

6. Hash Function

KIRA uses hash functions in various contexts. The used hash function is SHAKE256 with 128bit length output.

7. Protocol Parameters

7.1. Reserved Prefixes

The 16-bit prefixes for KIRA NodeIDs and PathIDs are taken from the ULA domain for experimentation.

KIRA NodeIDs: fc11::/16

KIRA PathIDs: fcaa::/16

7.2. Reserved Ports

KIRA port (to be allocated by IANA).

7.3. Reserved NodeIDs

Reserved NodeIDs:

```
Undefined NodeID = 0x00..00 (all zeros)
AllNodes  NodeID = 0xff..ff (all ones)
```

7.4. Timer Default Values

```
ULNHelloMinInterval = 200ms (fixed), 100ms (wireless)
ULNHelloMaxInterval = 30s (fixed), 1s (wireless)
ULNDiscoveryRspInitialMaxWaitTime = 200ms
UrgentUpdateHoldTime = 200ms
NormalUpdateHoldTime = 500ms
```

8. IANA Considerations

This memo currently includes no request to IANA yet. This may change in the future.

9. Security Considerations

There are various attacks that need to be considered. Future versions of this draft will have more detailed security considerations. However, cryptographic methods can be used to secure the integrity of routing information. NodeIDs can be generated as self-certifying identifiers, e.g., as hash from a public key, so that nodes can actually prove that they possess the corresponding private key, by adding a signature.

First of all, KIRA is assumed to be used in a trusted domain and that corresponding IPv6 address filtering is active at the domain boundaries. For bootstrapping, the KIRA routing daemon entity can install corresponding filtering rules at start and adapt them while building its adjacencies with authorized ULNs only (corresponding mechanism need to be specified in future versions of this document then).

Routing information is only trusted if it taken from source routes. It is assumed that routing messages have actually successfully traveled the source routing path up to this node and that this path has been working recently. Routing information received from other nodes (e.g., in RTable objects from FindNodeRsp, QueryRouteRsp, and UpdateRouteReq) is not trusted and only used for routing after path validation as described in Section 3.9.3. This increases the robustness and reliability of routing information used by KIRA: if an adversary node propagates wrong path information it will not be disseminated further. Adding proofs of transit for routing messages is possible in case certificates are used. However, this would be decoupled from a proof of transit in the data path.

One approach to achieve several security goals is to use a combination with Secure Zero Touch Provisioning (SZTP) [RFC8572]. This could be supported by an onboarding mode for KIRA nodes that only provides initial connectivity to perform the node’s onboarding procedure. This mode would use the EndSystemFlag first, so no messages would be routed via the node that tries to perform the onboarding. SZTP could provide NodeIDs derived from certificates and a KIRA node would restart and rejoin with the new NodeID. An advantage of using SZTP with KIRA is that the Bootstrap Server can be distributed, replicated, and located basically anywhere in the infrastructure by using KIRA’s built-in DHT. Similarly, bootstrapping information can be registered and found via the DHT. However, details how to combine KIRA with SZTP are left for future work.

10. References

10.1. Normative References

[RFC2473] Conta, A. and S. Deering, "Generic Packet Tunneling in IPv6 Specification", RFC 2473, DOI 10.17487/RFC2473, December 1998, <https://www.rfc-editor.org/info/rfc2473>.

[RFC7676] Pignataro, C., Bonica, R., and S. Krishnan, "IPv6 Support for Generic Routing Encapsulation (GRE)", RFC 7676, DOI 10.17487/RFC7676, October 2015, <https://www.rfc-editor.org/info/rfc7676>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC8572] Watsen, K., Farrer, I., and M. Abrahamsson, "Secure Zero Touch Provisioning (SZTP)", RFC 8572, DOI 10.17487/RFC8572, April 2019, <https://www.rfc-editor.org/info/rfc8572>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC8754] Filsfils, C., Ed., Dukes, D., Ed., Previdi, S., Leddy, J., Matsushima, S., and D. Voyer, "IPv6 Segment Routing Header (SRH)", RFC 8754, DOI 10.17487/RFC8754, March 2020, <https://www.rfc-editor.org/info/rfc8754>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9682] Bormann, C., "Updates to the Concise Data Definition Language (CDDL) Grammar", RFC 9682, DOI 10.17487/RFC9682, November 2024, <https://www.rfc-editor.org/info/rfc9682>.

10.2. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[Kademlia2002] Maymounkov, P. and D. Mazires, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric", In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), pages 53-65, Peer-to-Peer Systems, Springer Berlin Heidelberg, ISBN 978-3-540-45748-0, 2002.

[KIRA-Networking-2022] Bless, R., Zitterbart, M., Despotovic, Z., and A. Hecker, "KIRA: Distributed Scalable ID-based Routing with Fast Forwarding", 2022 IFIP Networking Conference (IFIP Networking), Catania, Italy, June 2022, <https://doi.org/10.23919/IFIPNetworking55013.2022.9829816>.

[KeLLy-2023] Seehofer, P., Bless, R., Mahrt, H., and M. Zitterbart, "Scalable and Efficient Link Layer Topology Discovery for Autonomic Networks", 19th International Conference on Network and Service Management (CNSM)Peer-to-Peer Systems, 30th OctNov 2nd, Niagara Falls, Canada, October 2023, <https://doi.org/10.23919/CNSM59352.2023.10327800>.

Acknowledgements

KIRA has been developed as joint work with Zoran Despotovic, Artur Hecker, and Martina Zitterbart. Hendrik Mahrt and Paul Seehofer are still contributing to KIRA's evolution.

Part of this work has been supported by the German Federal Ministry of Education and Research (BMBF) in the project "Open6GHub" (grant number 16KISK010).

Changes

changes in -04:

* added section about Fast Next Hop Detour

* expanded Security Considerations section a bit

* smaller fixes and clarifications

changes in -03:

* added a description of the endsystem mode and GRE encapsulation

* added a recommendation for default ULNHelloMinInterval, ULNHelloMaxInterval values of wireless interfaces

* various clarifications, fixed typos, added some more BCP14 keywords

changes in -02:

* used CDDL for message specifications

* fixed state-seq-num to be 32 bits

* more consistent wording, e.g., using underlay neighbor instead of physical neighbor

* added section about Fast Vicinity Alternatives

* added section about Forwarding Tier Functionality

* added a hint to SZTP in the security considerations section

changes in -01:

* few additions to the overview section

* detailed contact data

* changed physical neighbor to underlay neighbor and PN to ULN as an underlay neighbor can also be linked via L2 tunnel using IP etc.

* specified initial description of messages and their processing when sending or receiving

* removed PathTeardownRsp as it is not required

* minor edits (wording)

Author’s Address

Roland Bless
Karlsruhe Institute of Technology (KIT)
Kaiserstr. 12
76131 Karlsruhe
Germany

Phone: +4915201601400
Email: roland.bless@kit.edu
URI: https://tm.kit.edu/~bless