KIRA: Distributed Scalable ID-based Routing with Fast Forwarding

Roland Bless, Martina Zitterbart
Institute of Telematics, Karlsruhe Institute of Technology (KIT)
firstname.lastname@kit.edu

Zoran Despotovic, Artur Hecker
Huawei Research Center, Munich, Germany
firstname.lastname@huawei.com

***Abstract*—Emerging network infrastructures are increasingly softwarized, virtualized and, thus, flexible. They may even be viewed as a large, dynamic, and distributed elastic resource pool of network devices that can be flexibly configured and employed according to the needs of network services. Full control of such a resource pool requires resilient control plane connectivity. In this paper, we present KIRA, a two-tier routing architecture that provides self-organized, zero-touch, and extremely robust control plane connectivity. KIRA consists of the distributed, highly scalable, ID-based routing protocol R²/Kad that can run on top of any link layer. It is complemented by a forwarding tier with PathID-based fast forwarding for (control) data packets. KIRA shows excellent performance even in very large networks (evaluated with up to 200 000 nodes). R²/Kad allows for flexible memory/stretch tradeoff per node and finds shortest paths to certain destinations in most cases. R²/Kad converges loop-free and fast, even in very large networks with drastic failure scenarios.**

I. Introduction

Current and future network infrastructures tend to be more complex than ever: the sheer number and diversity of networked devices highly increased during the last couple of years and is envisioned to grow even further. This includes not only switches, routers, and end-systems like smartphones and servers, but also common devices in the context of Internet of Things (e.g., drones, mobile robots). Furthermore, the increased use of virtualized devices and softwarization enables higher flexibility in network infrastructures. For instance, the 6G ecosystem assumes that the future infrastructures used by mobile telco providers will be provisioned on-demand, mostly as virtual infrastructure allocated across administrative boundaries [1]. This increases infrastructure reach, e.g., a network service can be placed on a drone in a different administrative domain. Even more dynamic network infrastructures are envisioned, so that networks (or parts thereof) will appear and disappear, grow and shrink in much higher dynamics as seen today. Moreover, connectivity becomes denser, because redundant links have become cheaper.

On a more abstract level, various network services will run on top of a highly *dynamic* and distributed *elastic resource pool*, which is made of the above mentioned networked devices. This requires the ability to interconnect and reach all these resources in order to exert the necessary *resource control*, e.g., as part of Operations, Administration, and Maintenance (OAM) tasks. In this context, IETF's ANIMA working group envisions a unified and autonomous controllability of the resource pool [2]–[4]. However, a separate infrastructure for *out-of-band* OAM (requiring its own setup, configuration, and also its own OAM) has prohibitive costs and scaling limitations for such dynamic resource pools. Thus, a pervasive, highly reliable and resilient *in-band control plane (CP) connectivity* between the networked resources is required to control an elastic on-demand infrastructure [2]. Any CP message exchange (e.g., OAM, SDN/NFV/Cluster control) will run on top of this base CP connectivity. So before any CP messages can be sent, this connectivity must first be established, which can be achieved by a suitable routing protocol that interconnects all resources.

For seamless and continuous CP connectivity, such a routing protocol should have the following characteristics:

- *Massive Scalability* w.r.t. the number of nodes (100 000s of nodes). Routing table size and control message overhead are important given the high number of (virtualized) networked resources and increasingly denser meshed networks.
- *Self-organization and Supporting Dynamics* – the solution should run *"zero-touch"*, i.e., without any manual configuration or administration. Both are prohibitive for the projected network sizes. Connectivity must be (quickly) restored after failures in order to regain control over the resources.
- *Low Stretch* – used routes should be (close to) shortest paths.
- *Topological Versatility* – in contrast to recently upcoming topology-specific tailored versions of routing protocols [5]–[7], it should work on top of various topologies.

This paper presents *KIRA (Kademlia-directed ID-based Routing Architecture)* that provides self-organized and robust CP connectivity. Our contributions comprise:

- R^2/Kad, a distributed, highly scalable, ID-based routing protocol with configurable average stretch
- A scalable *dynamic routing* mechanism with fast convergence, which is loop-free, even during convergence
- A node individual adaptation mechanism, i.e., a node may reduce stretch by increasing memory for its routing table
- A *fast forwarding* scheme that eliminates source routes for CP data traffic, thereby reducing per-packet overhead

II. Overview of KIRA

KIRA's main objective is to provide self-organized robust CP connectivity, enabling in-band control communication between all resources. The latter is important as we consider elastic distributed resource control scenarios in which controllers are instantiated dynamically on existing resources. Moreover, their set of controlled resources may also change dynamically.

KIRA is structured into a two-tier architecture consisting of a *Routing* Tier and a *Forwarding* Tier (see Figure 1).

© IFIP, 2022. This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution. The definitive version was published in proceedings of Networking 2022.

KIRA runs the zero-touch, distributed, highly scalable, ID-based *routing protocol R^2/Kad* in the Routing Tier to find viable paths to destinations. Nodes employ this information to construct *fast path* forwarding tables in the Forwarding Tier for data packets (e.g., packets from resource control applications).

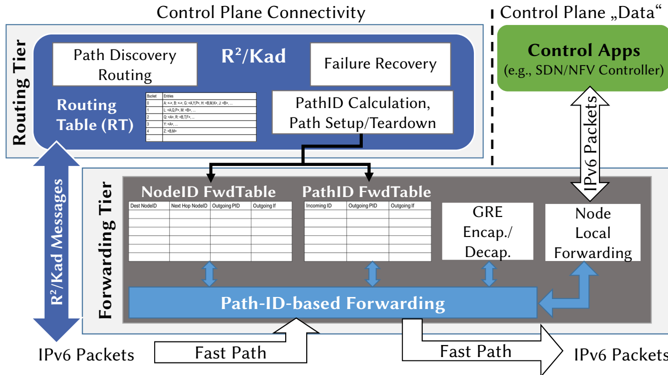


Figure 1. KIRA Architecture

R^2/Kad employs a flat ID-based addressing scheme to easily support self-organization and zero-touch as well as mobility and multi-homing. ID-based routing has the advantage of providing IDs as stable addresses to upper layers. Thus, in case (virtual) resources are moved within the topology, any control connection to them stays alive. KIRA is a *genuine* ID-based scheme, 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).

R^2/Kad messages are forwarded using source routing. In order to avoid the per-packet overhead of source routing for KIRA's *data* packets the Forwarding Tier employs *PathID-based forwarding* for the learned paths. It thus enables a more efficient and fast forwarding of data packets. A PathID denotes a certain path uniquely and is used as path label. It replaces the source routing path. Some PathIDs are distributedly computed a priori, others need to be setup in intermediate nodes by R^2/Kad on demand. KIRA's Routing Tier *does not depend* on the Forwarding Tier and also works without it as its routing messages bypass it (cf. fig. 1). R^2/Kad constructs the forwarding tables and includes the path setup procedure to install PathIDs along certain paths. Figure 1 also shows that R^2/Kad messages and data packets use IPv6 with IDs as addresses. The Forwarding Tier uses IPv6 GRE (Generic Routing Encapsulation) [8] for packets with PathIDs as destination.

III. $R^2/$KAD: BASIC OPERATION

Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ represents a network topology with node set $\mathcal{V}$ and set of edges $\mathcal{E}$. R^2/Kad uses topologically independent identifiers (NodeIDs) as node addresses. At startup, each node $v \in \mathcal{V}$ creates a random NodeID $n_v \in \mathcal{A}$ as its address (self-organized assignment). Address space $\mathcal{A}$ is usually large, e.g., 48–128 bits (in our implementation we chose 112 bits that make up an IPv6 address with a prepended 16-bit prefix).

We denote $\mathcal{G}$ as *underlay* where the nodes $v \in \mathcal{V}$ are connected by physical links from $\mathcal{E}$, whereas the same nodes form an *overlay* in $\mathcal{A}$ by logical connections. Figure 2 shows some nodes in an exemplary underlay $\mathcal{G}$ together with their overlay representations, e.g., node w has NodeID n_w=Y and node u has NodeID n_u=Z. In fig. 2 we assume letters closer in the alphabet have closer distance according to some metric $d(\cdot,\cdot)$. Although w and u are not neighbors in $\mathcal{G}$ they are overlay neighbors in $\mathcal{A}$, i.e., they have a logical connection.

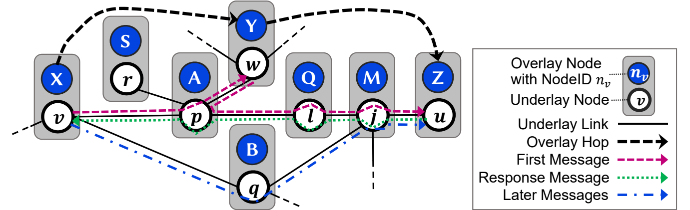


Figure 2. An exemplary topology. Letters resemble NodeIDs. Letters closer in the alphabet have smaller distance in ID space. First message uses overlay routing $\langle X, A, Y, A, Q, M, Z\rangle$, response message uses reversed path without cycles $\langle X, A, Q, M, Z\rangle$, later messages can use shortcut $\langle X, B, M, Z\rangle$.

The core concept of R^2/Kad is that it *discovers paths in the underlay* by using an ID-based overlay routing scheme (based on [9]) combined with source routing between overlay hops. Every node v manages a *routing table* (RT) (see section III-B) that contains all its known overlay neighbors called *contacts*. Each *contact* is identified by its NodeID n_j and has associated data that includes a *path vector* p_j. p_j represents the ordered sequence $\langle n_h, \ldots, n_l\rangle$ of all nodes between this node (n_v) and n_j (n_h is the next hop of n_v), corresponding to the currently best known path in the underlay leading from v to j. On startup (see section III-C), each node populates its RT with discovered *physical neighbors (PNs)* (link layer) and continues to discover nodes in its 3-hop vicinity. The overlay metric $d(\cdot,\cdot)$ determines which nodes will be put as contact into the RT. Further RT entries are added by various procedures, e.g., by incoming or overheard R^2/Kad messages during forwarding or active probing for paths to contacts (see section III-E).

A. Path Discovery and Routing

Assume node v with ID X needs to send a message to node u with ID Z. In case Z is a known contact of X, a path vector is stored already in RT 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^2/Kad uses a recursive version of Kademlia (Routing with Recursive Kademlia – R^2/Kad), which defines the (overlay) distance between two NodeIDs X and Y by the XOR metric $d(X,Y) = X \oplus Y$ [9].

The *Path Discovery* procedure is illustrated in fig. 2 and uses a request/response message pair, FINDNODEREQ/FINDNODERSP. In this example, assume that X identifies its contact Y (learned from PN 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* $r = \langle X, A, Y\rangle$ using path vector $p_w = \langle A\rangle$ of n_w=Y. The FINDNODEREQ is forwarded along

r (strict source route). Assume it eventually arrives at node Y which also looks up its currently known ID-wise closest contact to Z. The ID space is not cyclic, i.e., in fig. 2, X and Y are overlay neighbors as well as Y and Z, but A and Z not. Forwarding messages 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 $\langle A, Q, M\rangle$ to Z in fig. 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.

When the FINDNODEREQ arrives at Y, the same procedure is repeated (thus, it is a recursive variant of Kademlia). Node Y tries to find a contact n_o closer to Z than Y itself (i.e., $d(n_o, Z) < d(Y, Z)$). If n_o exists, source route r is appended by $\langle p_o, n_o\rangle$ and the FINDNODEREQ is forwarded to n_o. Otherwise, the FINDNODEREQ is terminated at this node and a response is sent back depending on the *"exact flag"* in the FINDNODEREQ. If *exact* 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 [10] that is used to realize distributed hash tables (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 the given example of fig. 2, we assume that Y knows Z as its contact with path vector $\langle A, Q, M\rangle$. It extends source route r of the FINDNODEREQ by $\langle A, Q, M, Z\rangle$ 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 [9] during forwarding and eventually finds node Z. 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.

The destination node responds with a FINDNODERSP message along the reversed source path with any cycles removed (see Response Message in fig. 2). 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.

The first message (e.g., initial FINDNODEREQ to Z) incurs a stretch that is largely correlated with the average number of traversed overlay hops (which grows with $\mathcal{O}(\log n)$), because each node knows the shortest paths to its contacts in most cases as shown in section VII. Stretch of response messages, e.g., FINDNODERSP is already reduced by eliminating any cycles. Subsequent messages (or "later messages") can do even better (cf. fig. 2): the originating node tries to shorten the reversed path of the response by applying shortcuts using its own RT information, thereby reducing path stretch further.

Note that R^2/Kad (and also KIRA) is *loop-free even during convergence* due to its ID-based routing with XOR's uniqueness ($\forall X, d \in \mathcal{A}\ \exists! Y \in \mathcal{A}\colon d(X, Y) = d$) and strict source routing between overlay hops: messages get dropped if there is no guaranteed progress.

B. Routing Table

R^2/Kad's efficiency and flexibility is closely related to its routing table. It is structured as tree of k-buckets (see fig. 3) as in [9]. A k-bucket in the RT contains a list of (at most) k contacts in distance between 2^i and 2^{i+1} (i.e., the bucket's range, where $0 \le i < 112$) from this node. Usually, $k \ge 20$ is constant and the same for all buckets and nodes, but it can also be varied per node. 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 contains contacts from half of the ID space whose highest NodeID bit differs from the node's ID, whereas the deepest bucket contains all nodes that are ID-wise closest to the node (i.e., the ID-wise closest overlay neighbors).

If 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)* so that contacts with shorter path lengths are preferred. Physical 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 PN buckets extension of [9], but the resulting stretch will be slightly higher.

X identifies its closest known contact in 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)*. If no prefix-wise progress can be made in this node, all paths have equal length, or it is the deepest bucket, the XOR metric is used to uniquely select the closest contact.

Consequently, the overall RT size grows only with $\mathcal{O}(l_{\mathcal{G}} \cdot \log n)$, where n is the number of existing nodes and $l_{\mathcal{G}}$ is the average path length of network $\mathcal{G}$: there are $\mathcal{O}(\log n)$ contacts with a path vector of length $l_{\mathcal{G}}$ in average. These small RT sizes lead to enormous advantages in dynamic scenarios (see section IV), because they cause less RT updates compared to larger RTs.

A remarkable property of R^2/Kad is that the paths for contacts get improved over time (see section III-E) and tend to be shortest paths regardless of the topology. That means the average routing table stretch $S_{RT} = \frac{1}{|RT|}\sum_{i \in RT} |\langle p_i, n_i\rangle| / |P_i^*|$

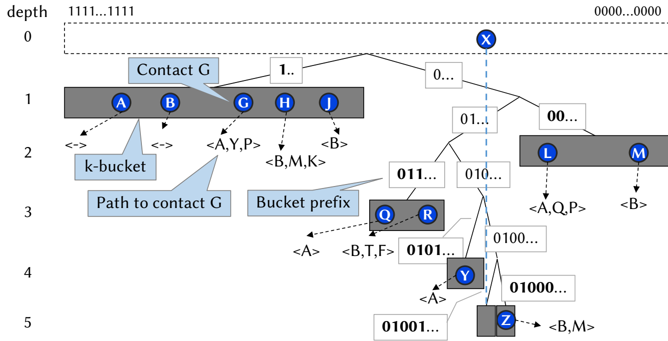


Figure 3. Sketch of the R^2/Kad routing table structure for node $n_v = X$ with ID 01001101 . . . 0: k-buckets hold contacts with path vectors. For clarity reasons, not all contacts of a bucket are shown.

is very close to 1 ($|RT|$: number of contacts, $|\langle p_i, n_i\rangle|$: current path length to i, $|P_i^*|$: length of the shortest path to i).

C. Startup and Vicinity Discovery

At startup, each node v generates its address n_v randomly. Afterwards, it explores and discovers its 3-hop vicinity, i.e., underlay nodes in physical distance ≤ 3 hops. Every node v maintains a local *state sequence number* s_v that is increased by one with every connectivity change in its set of directly attached *physical* neighbors. The discovery of PNs uses PNHELLO messages that are sent periodically on every physical link and contain the node's NodeID n_v and s_v. A node w may respond (e.g., in case of detecting v as new PN) with a PNDISCREQ, which also contains node w's current set of PNs. The PNDISCRSP from v contains s_v and its current set of PNs. PNDISCREQ and PNDISCRSP ensure bidirectional connectivity. They can be sent any time to check connectivity or to resynchronize the state, e.g., in case node v sees a newer s_w in w's PNHELLO or hears it indirectly from other nodes. During vicinity discovery, node v puts every PN into its RT. QUERYROUTEREQ/QUERYROUTERSP messages are used to get PNs or *RTable* objects from nodes in the vicinity.

D. Initial Join

As result of the vicinity discovery, all PNs of v and some nodes within its 3-hop radius will populate v'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 v simply "searches" for the k closest nodes to its own ID n_v: v sends a FINDNODEREQ for its own ID n_v and the closest neighbor replies with FINDNODERSP. This is repeated with a limited exponential backoff in order to detect or heal any network partitioning.

E. Routing Table Population and Improvement

Nodes may learn new contacts or new paths to already known contacts while forwarding messages, thereby refreshing several buckets. Normally, routing traffic can be triggered by KIRA application traffic, e.g., a control plane application that wants to initiate a control connection to a new destination. In addition, nodes may randomly probe for entries in their buckets to keep the overlay connected, heal partitions, etc. There are multiple mechanisms that may create new or update existing RT entries:

1) Random Probing: FINDNODEREQ messages may carry arbitrary destination IDs that do not correspond to NodeIDs of existing nodes. In this case, the ID-wise closest node will send back its k closest contacts. Each node periodically sends FINDNODEREQs to randomly chosen IDs with cleared *exact* flag (by default 2.5 messages/s). This will slowly populate or improve all buckets with existing contacts.

2) Path Overhearing: Nodes that forward routing messages may use the contained source route to improve their own routing information: they may learn shortcut routes to existing contacts or learn new existing contacts. However, only the so far traversed path is considered as it can be assumed that all traversed links worked recently.

3) Incoming Requests and Responses: 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.

4) Overhearing QUERYROUTERSP Messages: Bypassing QUERYROUTERSP messages contain *RTable* objects (as requested by QUERYROUTEREQ) and are inspected for interesting contacts and paths.

5) 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 FINDNODEREQ to them. ID-wise closest neighbors are probed more often than other contacts and those recently contacted ($\leq 2\,s$) are not probed. In case a path has a link or node failure, the FINDNODEREQ 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 IV).

IV. DYNAMICS: RECOVERY FROM FAILURES

In order to improve R^2/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 extremely robust and achieves a fast convergence. It retains R^2/Kad's utmost scalability by showing affordable overhead w.r.t. the amount of additionally required control messages (see section VII-D).

R^2/Kad nodes detect link and node failures of PNs by link layer notifications, missing PNHELLO or PNDISCRSP messages as well as *"Segment Failure"* errors anytime during forwarding along source routes (see section III-E5). To recover from such failures, R^2/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 physical link.

- Rediscover a feasible alternative route to the affected node using FINDNODEREQs. These carry *NotVia* information about failed links that must not be considered for routing.
- Per contact state sequence numbers s_j (see section III-C) 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 *NotVia* information and UPDATEROUTEREQs about failed links during forwarding 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 nearest 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 *NotVia* information.

A. Path Rediscovery

A node v that detects its PN q (cf. fig. 2) or the corresponding link (v, q) has failed, reacts as follows (unless isolated by that failure):

1) Set the state of the corresponding contact to *invalid* (in fig. 2 n_q=B). Invalid contacts will temporarily not be considered for routing.
2) Set the state of all contacts whose paths contain the failed link (n_v, n_q)=(X, B) to *invalid* (in fig. 2 n_u=Z with $\langle B, M, Z\rangle$ becomes invalid).
3) Send UPDATEROUTEREQ messages indicating the failure to four of *its* ID-wise nearest neighbors (e.g., Y and Z in fig. 2) via non-affected contacts (cf. also fig. 4).
4) Trigger a *rediscovery process* (described below) for n_q (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 nearest neighbors about the change.

Since UPDATEROUTEREQs have notification character only, they do not create any responses (even no errors 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 *exact* flag and the failed link (v, q) as additional *NotVia* information (n_v, n_q). It is sent to v's currently known nearest neighbors of the invalid contact (e.g., n_p=A in the example), which will then try to forward the FINDNODEREQ further toward the failed contact. The *NotVia* information avoids that nodes use obsolete routing information when forwarding the rediscovery message, i.e., paths that contain the failed link will not be

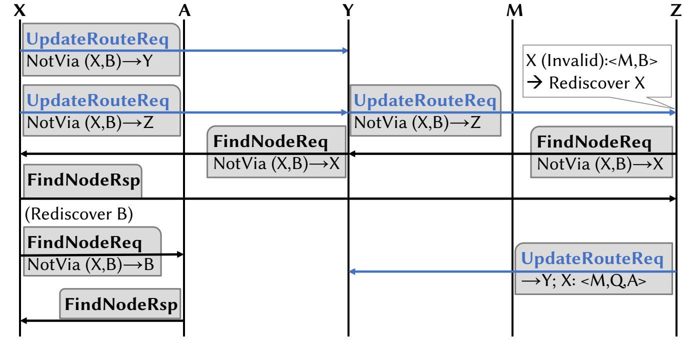


Figure 4. Exemplary excerpt of messages from n_v=X (see fig. 2) to inform about the failure of link (X, B) and to rediscover alternative paths. Node n_q=B will perform corresponding actions, which are not shown.

used for forwarding. Node A may not have heard yet about the broken link and thus will invalidate contact n_q=B if its prior preferred path is via $\langle X, B\rangle$. In order to ensure that only current *NotVia* information is considered, every link contained in *NotVia* is also accompanied by a related *age* value ΔT, specifying in ms how long ago the sender heard about the failed link. In case a FINDNODERSP is returned by n_q, 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 fig. 4). 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 affected ID-wise near contacts (in the deepest bucket) 500 ms, for PNs 1 s and for all others 2 s. Rediscovery messages are sent simultaneously to two different 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 results in section VII show that connectivity is very quickly restored after a failure even in drastic failure scenarios.

Node q at the other end of the failed link (v, q) also tries to rediscover v and thus sends an UPDATEROUTEREQ to its nearest overlay neighbors (e.g., n_p=A). Thereby, it may inform A as well as X about a new alternative route via M.

B. Ensuring Routing Information Validity

R^2/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* ΔT_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 ΔT_j 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. 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.

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 III-E5. Thanks to R^2/Kad's small routing table sizes, it is guaranteed that *all* inconsistencies are detected within at most 2 min in a 100 000 nodes network (k=20) with a probing rate of 2.5 messages/s per node.

V. Fast Forwarding

A potential drawback of R^2/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 fig. 1) leverages an approach similar to label switching. KIRA distributedly computes so-called *PathIDs* to avoid explicit path setup signaling in most cases. Every source routing path is hashed into a unique PathID (similar to [11]). IPv6 GRE [8] is used to carry the outgoing PathID in addition to source and destination NodeIDs. KIRA implements fast forwarding as follows:

- Each node computes *incoming* and *outgoing PathIDs* for each discovered path. Paths must be calculated for the full 2-hop vicinity and are then used for the a priori pre-computation of PathIDs. A PathID is a hash of *all* NodeIDs along a source path.
- The node inserts 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. The outgoing PathID is omitted for the last hop.
- A node that wants to send a data packet, sets the outgoing PathID of the source route as destination address of the outer encapsulation header and sends it to its PN. The source address of the outer header is set to the sender's 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 the outgoing PathID. 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.

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 $\leq$3 hops. PathID pre-computation for the full 2-hop vicinity provides a good trade-off between the number of a priori computed PathIDs and required path setup signaling. Routing information between nodes differ (in contrast to [11]), because they discover a different part of the topology and learn different contacts and paths due to cycle elimination and applied shortcuts. Therefore, intermediate nodes along a source route may not have computed the necessary PathIDs for others. Nodes explicitly setup paths via PathSetupReq only for paths $>$3 hops. The PathSetupReq can be answered earlier if the PathID for the rest of the path is already known.

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). We can employ common *longest prefix matching* for both tables. The 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 pre-computed and signaled entries (cf. section VII-E).

VI. Related Work

KIRA provides underlay connectivity by using source routing for path discovery based on Kademlia [9] overlay principles. Though overlay-based routing was proposed earlier [12]–[14], KIRA goes far beyond these approaches in several aspects. It provides fast forwarding of data packets without any source routing by employing PathIDs. Kademlia-based underlay connectivity and routing are optimized by a careful design of mechanisms, allowing to discover shortest paths for contacts in nearly all cases. Moreover, its design includes a highly robust and efficient mechanism to recover from failures. To best of our knowledge, KIRA is a novel concept that comprises many well-engineered mechanisms for a self-organized underlay connectivity even in very large networks and under high dynamic situations.

The original Kademlia scheme, in contrast, is completely situated in the overlay and does not provide any path rediscovery mechanism. UIP [14] also uses genuine ID-based addressing with a Kademlia-like overlay, but the efficiency of routes was not in focus (e.g., no PNS and PR used). Moreover, dynamics besides network split and network merge were not considered or evaluated. Virtual Ring Routing (VRR) [13] is an ID-based overlay-inspired routing protocol that sets up virtual links between overlay neighbors. Route efficiency is not considered, so routes may incur high stretch. Virtual links may cause lots of forwarding entries in intermediate nodes that have to forward lots of traffic. In contrast to KIRA's PathID scheme, paths setup by VRR are not aggregatable. VIRO [15] proposes a virtual ID-routing scheme based on an additional mapping layer that employs a Kademlia-like structure. Its virtual IDs

are topology dependent, so changes in the underlay require changes in the virtual ID space. The protocol requires address space construction, address assignment as well as a publish and query mechanism for address mappings.

DISCO [16] is an ID-based distributed compact routing scheme using landmarks, so traffic will concentrate around them. Our investigations of KIRA's routes showed no indication of traffic concentrations. DISCO possesses a worst case stretch guarantee, but its RT size scales with $\mathcal{O}(\sqrt{n \log n})$. It is not a genuine ID-based protocol since it uses topological addresses and two mapping systems for ID to locator resolution. Its reaction to link failure/repair have not been designed or evaluated in [16] and larger network changes (e.g., partitions) require re-election of landmarks that would change all addresses. The approach is more complex as it requires four interworking protocols. RPL [17] is used as routing protocol for the Autonomic Control Plane [4]. RPL, however, is based on a tree-like structure, requires manual configuration of roots and produces heavy traffic concentration near the root node as well as stretch [18]. More efficient routes come at the cost of additional entries and also additional route discovery overhead. Recent efforts try to mitigate the scaling issues of link-state protocols in denser data center topologies [5]–[7]. However, some of the solutions possess inherent scaling limitations, because they still use flooding and $\mathcal{O}(n)$ state, others (e.g., RIFT) are designed for specific topologies only.

VII. Evaluation

KIRA's behavior with varying network topologies and sizes was investigated by using the OMNeT++ [19] simulation framework. To achieve some variation, the processing time per message was randomly uniformly drawn from $[0, 500]\,\mu s$, whereas links have no speed limitation. As soon as a link comes up, a PNHello is sent to the other end and startup continues as described in section III-C. The initial join starts after 100 ms plus a random delay chosen from an interval of 250 ms, which is doubled every time until a limit of 300 s is reached. Simulation runs were performed multiple times ($\geq$ 10) with different seeds, but the deviation between the results was always so small (cf. error bars at 2σ in figs. 5 and 8) that mostly mean values are shown without deviation. This is expected as the change in the random distribution of NodeIDs across the topology has negligible impact on R^2/Kad.

A. Short Paths and Small Routing Tables

R^2/Kad discovers shortest paths to contacts almost always despite using small routing tables. We use the path stretch $S_{P_i} = |P_i|/|P_i^*|$ to assess the quality of the discovered path, P_i is the current discovered path, P_i^* the shortest path to node i in the underlay. The used topology was a Holme-Kim power-law graph [20] with parameter m=3 and clustering parameter 0.5. Many practical networks like the Internet are scale-free and challenging for dynamics [21]. The bottom plot in fig. 5 shows path stretch to randomly chosen destinations for first and later packets as well as average RT stretch (i.e., path stretch for contacts) in dependency of different k values and network sizes n. Stretch of response packets is between that of first and later packets. For k=40 path stretch of later packets is below 1.25 for sizes up to 10^4 nodes and below 1.5 even for 200 K nodes. Stretch for first, response, and later packets grows with $\mathcal{O}(\log n)$, caused by the average number of required overlay hops that increases in the same way with n. The results highlight the efficacy of eliminating cycles for response packets and applying shortcuts for later packets. Stretch of later packets is reduced by about 25% compated to first packets. Most importantly, the average RT stretch is ≈ 1 irrespective of n (and even from topology, see fig. 6).

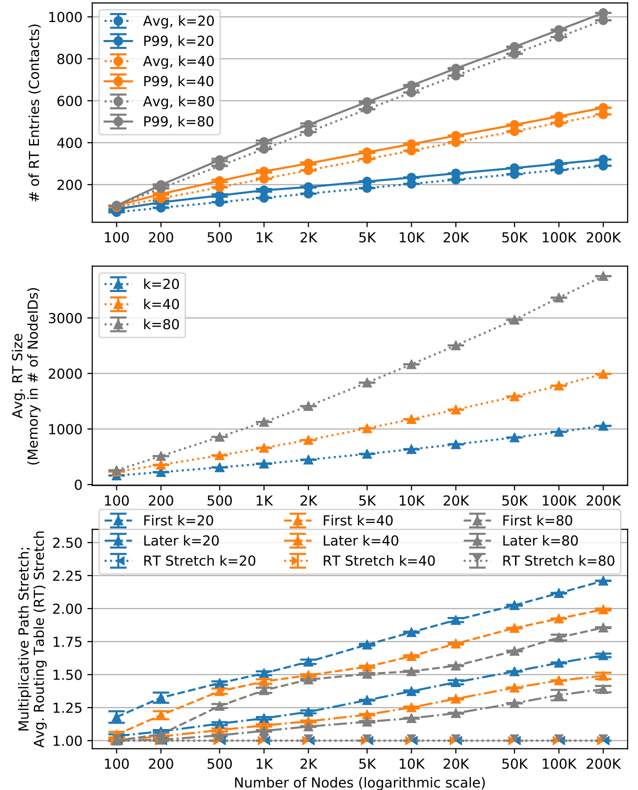


Figure 5. RT Size and Path Stretch for varied sizes n and values of k

Besides providing short paths, KIRA's small RT sizes are especially advantageous in very large networks. The number of RT entries scales with $\mathcal{O}(\log n)$ as shown in the topmost plot of fig. 5. For a network of 100 k nodes, 300 entries are needed, in case of 200 k nodes 320 RT entries (99 percentiles for k=20). Compared to a traditional link-state or distance vector routing approaches KIRA yields a reduction of three orders of magnitude. The memory requirements depicted in the middle plot of fig. 5 were calculated by multiplying the *average RT path length* by a node's average number of contacts. They also scale with $\mathcal{O}(l_{\mathcal{G}} \log n)$ as discussed in section III-B.

B. Tunable Efficiency – Trading RT Size for Path Stretch

With R^2/Kad, path stretch can be reduced by increasing the RT size. This can be done individually by each node – independently from any other node. This way, stronger nodes with more resources can increase the number of RT entries, e.g., by increasing k. Figure 5 shows the impact of RT size (by

varying k) on stretch. Clearly, with increasing k, the number of contacts increases and, consequently, the overall path stretch decreases. In this case, all network nodes used the same k.

C. Topological Versatility

Figure 6 shows that KIRA works definitely well in different topologies due to its structural pervasion by randomly distributing NodeIDs across the topology. The used topologies are (from left to right, number of nodes in parentheses): Internet (autonomous systems topology from 2021/12/21), Holme-Kim power-law [20], Watts Strogatz [22] (with different p values), Random (with average degree 8), Random Geometric (with average degree 8), 100×100 Grid, MixedFT-PL (a 400 node power-law topology connecting all cores nodes of four 20-ary Fat Trees [23]), 32-ary Fat Tree and the Kentucky Data Link topology from the Topology Zoo [24]. Watts Strogatz graphs can be varied by p values from regular structure (small p) via small world to random (p=1) graphs. The results for bucket size k=40 show that the average RT stretch is 1, i.e., shortest paths to contacts are learned in *all* different topologies. Stretch becomes larger in randomly connected topologies, because there is no underlying structure that R^2/Kad can make use of. R^2/Kad is able to work efficiently also in topologies with large diameters (e.g., 151 for p=0.003) and in a mixture of Fat Tree and power-law topologies (MixedFT-PL).

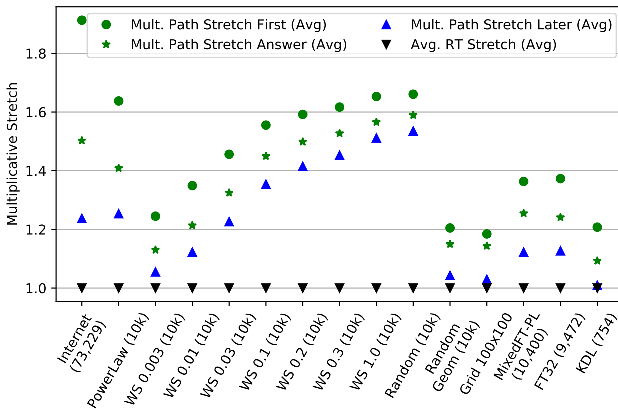
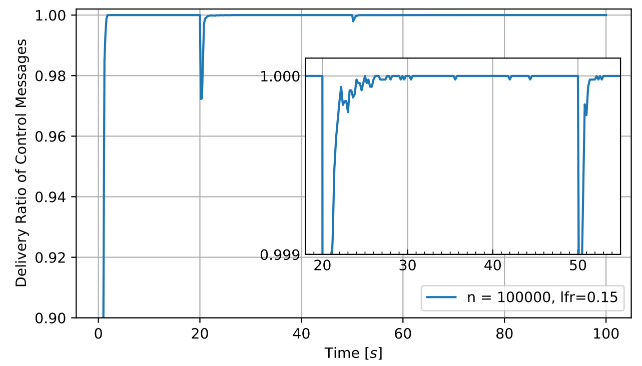


Figure 6. Mult. Path Stretch and RT Stretch for different topologies (k=40)

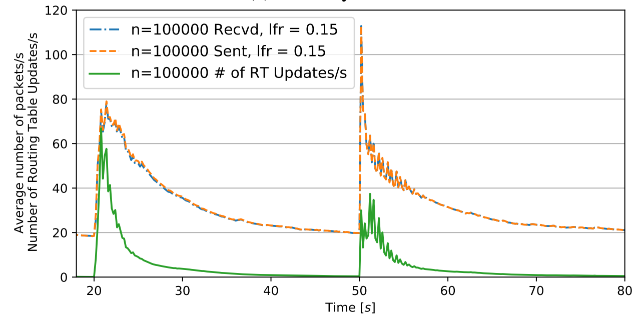
D. Dynamics – Recovery from Failures

KIRA's dynamics mechanism (see section IV) is extremely robust, shows fast convergence and overhead with scalable growth w.r.t. increasing n. The first scenario tests whether recovery from link failures is successful so that all (non-isolated) nodes can be reached. Every node sends 2.5 FIND-NODEREQs/s as reachability test packets randomly to other nodes (but not to isolated nodes, because they cannot be reached physically at all). In fig. 7 a *drastic failure scenario* is investigated: 15% of the links fail randomly and *simultaneously* at t=20 s and are re-enabled again at once at t=50 s.

Figure 7a shows that the delivery ratio drops during the link failure at t=20 s slightly (y-axis begins at 0.9), but it takes only roughly 6 s (cf. zoomed range) until nearly all test packets reach their destinations again, i.e., recovery is extremely fast.



(a) Delivery ratio



(b) Average rate of received and sent control messages as well as routing table updates/s per node

Figure 7. 15% randomly distributed links (41 226) fail at t=20 s and are re-enabled at t=30 s in a power-law topology of n = 100 000 nodes

When the failed links become available again after 30 s, the recovery period is even shorter. At first, some packets get dropped, because test packets are sent to previously isolated nodes again, but the reachability information has not reached all nodes yet. Figure 7b shows the average rate of sent and received control traffic per node as well as the number of RT updates. Since not all rediscovery messages are successfully delivered, there is a small difference between sent and received messages during the failure period. The rate of RT updates shows quick convergence in both cases. Link state routing would have sent at least 82 452 link state advertisements instantaneously and forwarded them to all nodes.

KIRA is also extremely scalable while coping with dynamics, because its message overhead per node grows only logarithmically with increasing network sizes. Figure 8 shows overhead in terms of the number of additionally sent and received messages for the recovery from link failures (upper plot) and during the period after the failed links have been restored (lower plot) respectively. The overhead is calculated by summing up all messages after the corresponding event that are above the regular periodic messages level. The link failure ratio (LFR) is varied as well as the size of the (power-law) topology. The overhead clearly increased with higher LFRs, but corresponds to logarithmic growth. Overhead may even decrease with increasing topology size due to overhearing messages with repaired/restored paths. These results confirm that KIRA is able provide fast recovery from (even severe) failure situations with scalable overhead growth.

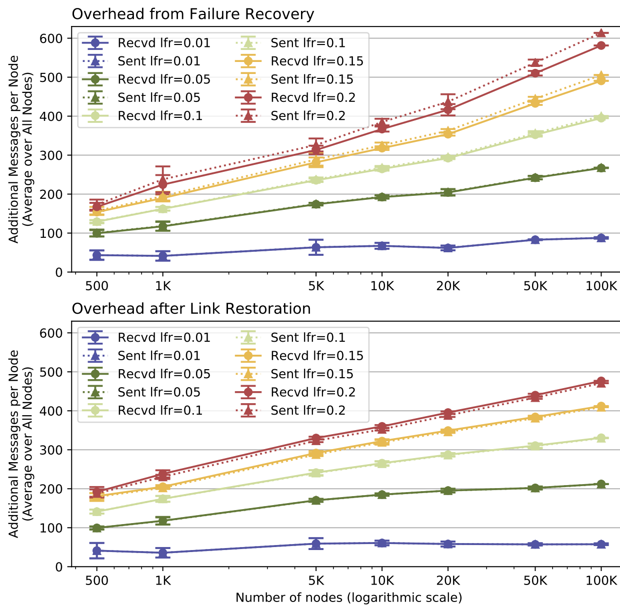


Figure 8. Overhead in total number of additionally sent/received messages per node relating to different link failure ratios (LFR) and topology sizes

E. Fast Forwarding

We briefly show that KIRA's forwarding table sizes remain scalable even though other other nodes create "foreign" entries by path setup requests. The left plot in fig. 9 shows different types of forwarding table entries for different power-law topology sizes (log scale). The number of foreign PathID entries increases with $\sqrt{n}$ only.

The right plot shows the cumulative distribution functions of the number of forwarding table entries for a 100 k nodes topology. One can see that the number of actually used pre-computed entries is often smaller than the overall pre-computed entries and in the range of the number of contacts. Implementations can install actually used pre-computed entries on demand, thereby saving precious forwarding table space.

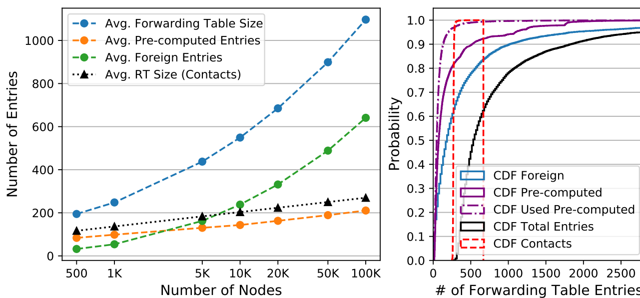


Figure 9. Left: Average Number of Different Forwarding Table Entry Types; Right: CDFs for 100 k nodes; all power-law topologies [20]

VIII. Conclusions and Outlook

KIRA is a novel architecture that comprises a combination of a distributed, zero-touch, highly scalable ID-based routing protocol R^2/Kad and PathID-based forwarding for data packets. It is well suited to provide robust control connectivity for a large resource pool. R^2/Kad's routing table size grows with $\mathcal{O}(l_{\mathcal{G}} \log n)$, where n is the number of existing nodes and $l_{\mathcal{G}}$ is the average path length. KIRA is loop-free, shows acceptable low stretch in various topologies, discovers shortest paths for contacts, and showed fast convergence even for drastic failure scenarios with scalable overhead growth in large (100k+) topologies. A unique feature is also its per-node tunable efficiency, i.e., nodes can increase their local routing table size, thereby reducing global stretch. KIRA's PathID-based forwarding scheme allows fast and more efficient forwarding for data packets. Due to space restrictions we cannot present extensions for end-system nodes or for improved path diversity. Further work investigates the introduction of link weights, multi-path routing, and support of mobile end-systems.

References

[1] C. J. Bernardos and M. A. Uusitalo, "European Vision for the 6G Network Ecosystem," Jun. 2021, 5GPPP, doi:10.5281/zenodo.5007671.

[2] T. Eckert (Ed.) and M. Behringer, "Using an Autonomic Control Plane for Stable Connectivity of Network Operations, Administration, and Maintenance (OAM)," RFC 8368, RFC Editor, May 2018.

[3] M. Behringer (Ed.) *et al.*, "A Reference Model for Autonomic Networking," RFC 8993, RFC Editor, May 2021.

[4] T. Eckert (Ed.), M. Behringer (Ed.), and S. Bjarnason, "An Autonomic Control Plane (ACP)," RFC 8994, RFC Editor, May 2021.

[5] R. White and M. Aelmans, "Recent Developments in Link State on Data-Center Fabrics," *Internet Prot. Journal*, vol. 23, no. 2, Sep. 2020.

[6] A. Przygienda *et al.*, "IS-IS Flood Reflection," Internet Draft draft-ietf-lsr-isis-flood-reflection-07, Dec. 2021, work in progress.

[7] T. Li *et al.*, "Dynamic Flooding on Dense Graphs," Internet Draft draft-ietf-lsr-dynamic-flooding-10, Dec. 2021, work in progress.

[8] C. Pignataro, R. Bonica, and S. Krishnan, "IPv6 Support for Generic Routing Encapsulation (GRE)," RFC 7676, RFC Editor, Oct. 2015.

[9] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[10] F. Dabek *et al.*, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Peer-to-Peer Systems II*. Springer, 2003, pp. 33–44.

[11] H. T. Kaur *et al.*, "BANANAS: An Evolutionary Framework for Explicit and Multipath Routing in the Internet," in *Proc. of the ACM SIGCOMM FDNA Workshop*. ACM, 2003.

[12] K. Kutzner, K. Cramer, and T. Fuhrmann, "Towards Autonomic Networking Using Overlay Routing Techniques," in *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*. Springer, 2005.

[13] M. Caesar *et al.*, "Virtual Ring Routing: Network Routing Inspired by DHTs," *SIGCOMM CCR*, vol. 36, no. 4, pp. 351–362, Oct. 2006.

[14] B. Ford, "Unmanaged Internet Protocol: Taming the Edge Network Management Crisis," *SIGCOMM CCR*, vol. 34, no. 1, Jan. 2004.

[15] S. Jain, Y. Chen, Z.-L. Zhang, and S. Jain, "VIRO: A scalable, robust and namespace independent virtual Id routing for future networks," in *2011 Proceedings IEEE INFOCOM*. IEEE, Apr. 2011, pp. 2381–2389.

[16] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy, "Scalable Routing on Flat Names," in *Proc. CoNEXT*. ACM, 2010.

[17] T. Winter (Ed.), P. Thubert (Ed.) *et al.*, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," RFC 6550, RFC Editor, Mar. 2012.

[18] J. Tripathi (Ed.), J. de Oliveira (Ed.), and J. Vasseur (Ed.), "Performance Evaluation of the Routing Protocol for Low-Power and Lossy Networks (RPL)," RFC 6687, RFC Editor, Oct. 2012.

[19] OpenSim Ltd., "OMNeT++," https://omnetpp.org, Feb. 2022.

[20] P. Holme and B. J. Kim, "Growing scale-free networks with tunable clustering," *Physical Review E*, vol. 65, no. 2, p. 026107, 2002.

[21] D. Krioukov, k. c. Claffy, K. Fall, and A. Brady, "On Compact Routing for the Internet," *SIGCOMM CCR*, vol. 37, no. 3, pp. 41–52, Jul. 2007.

[22] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, Jun. 1998.

[23] M. Al-Fares *et al.*, "A Scalable, Commodity Data Center Network Architecture," in *Proc. of SIGCOMM*. ACM, 2008.

[24] University of Adelaide, "The Internet Topology Zoo," Jan. 2022, http://www.topology-zoo.org/.