RIPPLE: Loop-Free Multi-Path Routing with Minimum Blocking during Convergence

J.J. Garica-Luna-Aceves
Computer Science and Engineering Department, University of California, Santa Cruz, CA, USA
jj@soe.ucsc.edu

***Abstract*—The Routing Information Protocol with Probing for Looplessness and Efficiency (RIPPLE) is introduced for loop-free multi-path routing. Each router maintains the distance and the hop-count (called the hop-count reference) along its preferred path to each destination. Routers are allowed to select neighbors as next hops to destinations as long as they satisfy an ordering condition based on the values of hop-count references. If needed, routers use probes to find valid routes provided by routers with the same hop-count references as those stated in probes. RIPPLE is shown to be loop-free, which allows it to converge to shortest paths within a finite time even when nodes fail or the network partitions. RIPPLE is also shown to be near-optimal in terms of the time routers take to attain new loop-free routes to destinations.**

I. Introduction

Eliminating routing-table loops while converging to optimal routes is much needed, because network resources are not wasted in the forwarding of data traffic that cannot reach their intended destinations.

As Section II points out, the methods used today to attain shortest paths and eliminate or detect routing loops can be classified into three types of mechanisms: using destination sequence numbers, establishing multi-hop router coordination, and sharing of either path information or complete-topology information. No new methods for loop-free routing have been reported for many years. However, existing distributed routing methods either result in temporary routing loops or in routers not having valid paths while distributed algorithms converge.

Section III presents the *Routing Information Protocol with Probing for Looplessness and Efficiency* (**RIPPLE**), which provides multiple loop-free routes to destinations without incurring long delays converging to shortest-path routes. RIPPLE replaces the destination sequence numbers used in routing protocols like the Destination Sequenced Distance Vector (DSDV) [3] with *hop references*, which are the number of hops along preferred paths to destinations. RIPPLE also replaces the diffusing computations used in the Diffusing Update Algorithm (DUAL) [1] with probes that carry requested hop references, so that they can be resolved with the first response that satisfies the requested hop reference. Section IV sows that RIPPLE is loop-free and converges to optimal routes within a finite time.

Section V shows that the speed with which RIPPLE converges to valid loop-free paths is inherently much faster than all prior methods for distributed shortest-path routing. Section VI provides our conclusions.

II. Motivation for RIPPLE

The reader is referred to [1], [2] for summaries of prior work on shortest-path routing methods. The Distributed Bellman-Ford (DBF) algorithm is arguably the simplest distributed approach for shortest-path routing, and has been used in many routing protocols, including the original ARPANET routing protocol. However, DBF suffers from the non-convergence problem usually called the counting-to-infinity problem. In practice, routing protocols based on DBF are forced to stop when a predefined maximum-distance value is reached. However, this does not guarantee convergence to shortest distances.

As a result of the non-convergence problem of DBF, many routing protocols were developed based on the dissemination of partial or complete topology information or the use of complete path information in routing updates . These routing protocols do not guarantee acyclic routing (avoiding routing-table loops at every instant), but guarantee convergence by detecting and breaking loops within a finite time.

All prior work on shortest-path routing protocols that *prevent* routing-table loops and try to ensure convergence are based using destination-based sequence numbers, and using multi-hop router coordination. The signaling used in routing protocols to attain shortest paths while preventing looping may result in multiple routers having no valid routing-table entries for one or more destinations even though physical paths exist to those destinations. We refer to this problem as **blocking**, which may persist for long periods of time, until all routers make the proper updates to their routing state according to the protocol.

Several shortest-path routing protocols (e.g., DSDV [3]) have used destination sequence numbers to eliminate the convergence problems of DBF and try to ensure acyclic operation. This approach has also been applied to the design of multi-path routing protocols. Using sequence numbers to provide acyclic operation is appealing, because it appears to be simple. However, the basic approach used in DSDV and similar routing protocols incurs considerable blocking after a link failure, node failure, or link-weight increase that affects any routing-table entry. This is the case because the destination itself must issue a new sequence number for itself to the rest of the network.

A number of shortest-path routing approaches have been developed that provide loop-free routes at every instant by requiring routers to coordinate the updating of routing tables on a multi-hop basis. The most popular of these schemes is the Diffusing Update Algorithm (DUAL) [1] and is the basis of Cisco's EIGRP (see RFC 7868).

This type of routing protocols convergence incur long convergence delays resulting from pre-update coordination that is unavoidable, because a router cannot determine from a single reply from a neighbor whether a reported distance corresponds to an acyclic path. As a result, a router that coordinates with its neighbors to update a routing-table entry must wait until all its neighbors attest that all possible paths that included the router itself have been eliminated or updated with new distances. Blocking in this type of routing protocols occurs after link failures or node failures when routers lose their successors to a destination and are forced to coordinate the update of their routing tables with other routers.

III. RIPPLE

We use the following terminology to describe distributed shortest-path routing algorithms in this section and RIPPLE in the next section: N is the set of network nodes (routers and destinations), and E is the set of links in a network. The set of nodes that are immediate neighbor routers of router k is denoted by N^k. A node in N is denoted by a lower-case letter, and a link between nodes n and m in N is denoted by (n, m), and the weight of the link from router i to router j is denoted by $l(i, j)$ and $l(i, j) \in \mathbb{R}^+$. The distance assumed for a destination for which no path is known is denoted by δ_∞, and the distance from destination d to itself is $\delta_0 = 0$.

A. Information Maintained

For each destination, a router maintains a list of neighbors that can serve as its successors (next hops) along paths to the destination. The nth path from router k to destination d is denoted by $P_d^k(n)$, the distance and successor (i.e., next hop) along that path are denoted by $\delta_d^k(n)$ and $s_d^k(n)$, respectively. The shortest distance reported by router k for destination d is denoted by δ_d^k.

Each router k knows its own identifier (k), its initialization status (σ^k), and maintains the following three tables.

Link-Weight Table (LWT^k): This table lists an entry for each link to a known neighbor router $n \in N^k$. The entry for link (k, n) in LWT^k states the weight $l(k, n)$ of the link and a lifetime LT_n^k for the neighbor entry of router n. The maximum lifetime of a neighbor entry is a constant LT defined for the network.

Neighbor Table (NT^k): This table lists the shortest distance and hop reference reported by each neighbor for each destination. The entry in NT^k for destination d at router k is denoted by $NT^k(d)$ and for each neighbor $p \in N^k$ states the distance δ_{dp}^k and the hop reference h_{dp}^k reported by neighbor p. If a neighbor q has not reported any distance for d to router k, then router k assumes that $\delta_{dq}^k = h_{dq}^k = \delta_\infty$.

Routing Table (RT^k): This table lists an entry for each destination. The entry in RT^k for destination d is denoted by $RT^k(d)$ and states: The distance (δ_d^k); the hop reference (h_d^k); the minimum value of a requested reference (μ_d^k) known to router k for destination d; the preferred successor (s_d^k); and the set of successors (S_d^k).

If router k has no successor for destination d, then $s_d^k = 0$. The value of μ_d^k is initialized to equal h_d^k and it is updated as needed to reflect the minimum value of a requested reference contained in a probe forwarded by the router.

B. Information Shared Periodically

Routers exchange messages reliably and periodically among one another to update their routing information. A routing message from router k is denoted by M^k and contains its identifier k and a list of two types of entries, which are updates and probes. We assume that a router has a pre-defined neighbor set, which is the case in wired networks. Minor changes to the signaling would be needed if unreliable message transmissions must be used, destinations were of interest to only a subset of routers, or RIPPLE operated in an ad-hoc wireless network.

RIPPLE Probe: Probes are used by routers to coordinate the updating of their routing tables. A probe from router k for destination d is denoted by $P(d, \delta_d^k, h_d^k, r_d^k, n_d^k)$, where d is the destination identifier, δ_d^k is the current distance to destination d, h_d^k is the hop reference to destination d, r_d^k is the requested reference that a router must have as its own hop reference in order to originate a response, and n_d^k is the intended recipient of the probe. A probe intended for all neighbors states $n_d^k = 0$

RIPPLE Update: Updates are used to respond to probes or inform neighbors of new distances and hop references. An update from router k for destination d is denoted by $U(d, \delta_d^k, h_d^k, r_d^k)$, which specifies the same elements of a probe. The value of r_d^k in an update sent in response to a probe is set to the requested reference in the probe being answered, or the requested reference in a response being relayed. Alternatively, the value of r_d^k in an update that is not a response to a probe is equal to h_d^k.

Router k maintains a timer UT^k to ensure that it sends a routing message soon after it updates its routing table or decides to forward or respond to a query, and sends routing messages often enough to inform its neighbors of its presence. If a router k needs to send a routing message with updates, it does so after a minimum amount of time t_{min} has elapsed from the time it sent its prior routing message.

In the absence of changes in its routing table, router k sends a message with a "hello" update $U(k, \delta_k^k = \delta_0, h_k^k = 0, r_k^k = 0)$ to update the lifetime entries maintained for itself by its neighbors no later than t_{max} seconds from the time it sent its last message. The timer t_{max} is shorter than a maximum lifetime LT. Router k sets UT^k equal to t_{max} after sending a routing message, and sets UT^k equal to t_{min} after preparing updates or queries to be sent in response to an input event.

C. Initialization

Router k is initialized after an initialization delay elapses, that is long enough to ensure that, in the event that router k is restarting after a failure, all neighbor routers have processed previous routing messages from router k and also determined that it is not operational. The steps taken during initialization at router k involve setting $\sigma^k = T$; $\delta_k^k = \delta_0$, $h_k^k = 0$, $\mu_k^k = 0$, $s_k^k = k$, and $S_k^k = \{k\}$. In addition, the following values are initialized for each $q \in N^k$: $\delta_q^k = h_q^k = \mu_q^k = \delta_\infty$, $s_q^k = 0$, $S_q^k = \emptyset$, $\delta_{qq}^k = h_{qq}^k = \delta_\infty$, $\delta_{kq}^k = h_{kq}^k = \delta_\infty$, and $\delta_{qq}^k = h_{qq}^k = \delta_\infty$. Router k then sends a routing message with a "hello" update $U(k, \delta_k^k = \delta_0, h_k^k = 0, r_k^k = 0)$ to announce its presence to its neighbors.

D. Handling Link Failures and Link-Weight Changes

Link failures can be detected in multiple ways, including keeping track of data packets not being acknowledged by a neighbor router. Router k assumes that a neighbor router q from which no signaling has been received for LT seconds can be declared to have failed. Accordingly, router k sets the distances and hop reference from router q to every destination to δ_∞. Router k updates LWT^k when the weight $l(k, q)$ of an adjacent outgoing link (k, q) changes.

E. Handling Routing Messages

Router k processes routing messages only if it has been initialized, which means that $\sigma^k = T$. After initialization, if router k receives a routing message from a neighbor q, it first updates the lifetime of the entry for that neighbor (LT_q^k).

Router k detects that a new neighbor q is present when it receives a "hello" update from q (i.e., $U(q, \delta_0, 0, 0)$) and its local state for q has $\delta_{qq}^k = \delta_{kq}^k = \delta_\infty$, which indicates that no messages were being received over link (k, q). In this case, router k immediately sends a routing message with an update $U(d, \delta_d^k, h_d^k, r_d^k = h_d^k)$ for each destination d for which $\delta_d^k < \delta_\infty$. If router k receives an update or a probe from $q \in N^k$ then it updates its neighbor table for destination d (i.e., $NT^k(d)$) with $\delta_{dq}^k \leftarrow \delta_d^q$ and $h_{dq}^k \leftarrow h_d^q$ before proceeding to update $RT^k(d)$.

F. Updating Routing State

After router k updates LWT^k and NT^k, it updates RT^k in two phases as shown in Algorithm 1. The way in which router k updates RT^k depends on whether or not the router has at least one neighbor that satisfied the following RIPPLE ordering condition ($\mathcal{ROC}$):

$$\mathcal{ROC} : q \in S_d^k \ \Big(\ \big[\ h_{dq}^k < h_d^k \big] \vee \big[(h_{dq}^k = h_d^k) \wedge (\delta_{dq}^k < \delta_d^k) \ \ \big] \ \Big) \quad (1)$$

$\mathcal{ROC}$ is similar to the condition used in DSDV [3], with the key differences being that hop references substitute sequence numbers and the condition is applied only to those neighbors in the successor set. $\mathcal{ROC}(q) = T$ in Algorithm 1 is used to denote the fact that neighbor q satisfies $\mathcal{ROC}$.

Phase 1 of Algorithm 1 is meant to eliminate current successors that, as a result of an input event, have larger hop references than the hop reference at router k and hence do not satisfy $\mathcal{ROC}$. However, a neighbor already in the successor set S_d^k could not change its own successors to include k with $\mathcal{ROC}$ being satisfied; therefore, router k does not need to consider the distances reported by neighbors in the successor set to determine which neighbors should remain in S_d^k. Those neighbors whose hop references are larger than the hop reference of router k are delete from S_d^k, and the distance is updated accordingly. Furthermore, the hop reference is not updated if S_d^k becomes empty, so that its value is remembered for Phase 2.

During Phase 2, those neighbors outside of S_d^k that satisfy $\mathcal{ROC}$ are brought into S_d^k, and the distance and hop reference are updated accordingly. This phase allows all neighbors that satisfy $\mathcal{ROC}$ to be considered taking into account the distance and hop-reference increases caused by successors in S_d^k. The hop reference is not updated if S_d^k is empty at the end of this phase, so that its value can be used in probes.

Algorithm 1 Routing Table Update ($RT^k(d)$)

```
INPUT: $N^k, S_d^k, LWT^k, NT^k(d), RT^k(d)$
PHASE 1:
for each $q \in S_d^k$ do
    if $(h_{dq}^k > h_d^k)$ then $S_d^k \leftarrow S_d^k - \{q\}$;
end for
if $(S_d^k = \emptyset)$ then
    $h_d^k \leftarrow h_d^k$ (i.e., hop reference is not updated); $\delta_d^k \leftarrow \delta_\infty$; $s_d^k \leftarrow 0$
else
    $\delta_d^k \leftarrow Min\{\delta_{dn}^k + l(k,n) \mid n \in S_d^k\}$;
    $s_d^k \leftarrow Min\{q \in S_d^k \mid \delta_{dq}^k + l(k,q) = \delta_d^k\}$; $h_d^k \leftarrow h_{ds_d^k}^k + 1$
end if
PHASE 2:
for each $q \in N^k - S_d^k$ do
    if $(\mathcal{ROC}(q) = T)$ then $S_d^k \leftarrow S_d^k \cup \{q\}$
end for
if $(S_d^k = \emptyset)$ then
    $h_d^k \leftarrow h_d^k$ (i.e., hop reference is not updated); $\delta_d^k \leftarrow \delta_\infty$; $s_d^k \leftarrow 0$
else
    $\delta_d^k \leftarrow Min\{\delta_{dn}^k + l(k,n) \mid n \in S_d^k\}$;
    $s_d^k \leftarrow Min\{q \in S_d^k \mid \delta_{dq}^k + l(k,q) = \delta_d^k\}$; $h_d^k \leftarrow h_{ds_d^k}^k + 1$
end if
return
```

Sending Updates or Probes: After updating RT^k, router k sends an update or a probe depending on the input event and whether router k is ordered or blocked.

Router k is said to be **ordered**, or to be in the ordered state, if the following **ordered condition** is true:

$$\Big[\exists q \in N^k (\ \ \mathcal{ROC}(q) = T \ \ ) \Big] \vee \Big[\forall q \in N^k (\ h_{dq}^k = \delta_\infty \ ) \Big] \quad (2)$$

Eq. (2) simply states that an ordered router has at least one neighbor that satisfies $\mathcal{ROC}$ or it knows that none of its neighbors have paths to the destination. Else, router k is said to be **blocked**, or to be in the blocked state.

If router k becomes blocked as a result of an input event other than a probe, then it originates probe $P(d, \delta_d^k, h_d^k, r_d^k, n_d^k = 0)$ stating a requested reference that equals the smaller value between its own hop reference and a requested reference received in a prior probe.

Only blocked routers need to send probes to all neighbors, and ripples of probes are forwarded by ordered routers towards the destination. This reduces the signaling overhead induced by probes. A router forwards a probe to all its neighbors if it becomes blocked or remains blocked and the probe it received stated a smaller requested reference than the value of μ_d^k. Router k forwards a probe from neighbor q only to its preferred successor s_d^k if the probe states $n_d^q = k$ or $n_d^q = 0$, i.e., router k is asked to help, and it remains ordered after receiving a probe but cannot send a response. The preferred successor is used for selective forwarding because the intent of the forwarded probe is to determine whether the shortest path preferred by router k remains a loop-free shortest path.

A router may send an update as a result of a link-weight change, an update, or a probe, provided that the router becomes or remains ordered. This causes ripples of updates to percolate among routers that continue to be ordered with respect to their successors.

Router k originates a response to a probe as an update if its hop reference is smaller than or equal to the requested reference in the probe. The router may also send an update in response to a probe if itself and none of its neighbors have finite distances, which means that the router is not blocked but there are not available paths. This causes ripples of updates sent back towards the origin of the probe. A router sending an update to relay a response states its own distance and hop reference, and states the same value of the requested hop reference in the update it received as a response to a probe.

If an update does not constitute a response to a probe, the values of h^k_d and r^k_d are the same in the update. Accordingly, ripples of updates can serve to respond to probes only if the origins of the updates have hop references whose values equal the requested references on the probes. Router k updates the value of μ^k_d to equal the smallest known value of requested references received or created by the router.

A router that sends a probe in RIPPLE can trust the first response that satisfies the requested reference stated in its probe. This is possible because a router remembers its own hop reference that needs to be satisfied for a response to its probe to be valid.

G. Examples of RIPPLE Operation

Figure 3 illustrates the fast convergence of RIPPLE with a five-node network example The distance, hop reference, and successor to destination d are indicated next to each router. The value of the minimum requested reference stored at each router for destination d is omitted, given that is not used in the example. Successors to destinations are indicated by arrowheads. An update sent by router k regarding destination d is denoted by $U[\delta^k_d, h^k_d, r^k_d]$.

Figure 3(a) shows the routing state of the routers when link (c, d) fails. Figure 3(b) shows that router c simply sends an update after the link failure because its distance increases to 6 but neighbor e satisfies $\mathcal{ROC}$ with $h^c_d = h^c_{de} = 1$ and $\delta^c_d = 2 > 1 = \delta^c_{de}$. As Figure 3(c) shows, the update from router c results in $\mathcal{ROC} = T$ at router b because $h^b_{dc} = h^b_d = 2$ during Phase 1 of the routing-table update process, and $c \in S^b_d$ during Phase 2 once router b updates $h^b_d = 3$. Accordingly, router b sends update $U[d, \delta^b_d = 7, h^b_d = 3, r^b_d = 3]$ Figure 3(d) shows that the same outcome occurs at router a when it processes the update from router b and sends update $U[d, \delta^c_d = 8, h^b_d = 4, r^b_d = 4]$, which reflects its shortest distance through router b, and also keeps router e as a successor. The small ripple of updates that occur in RIPPLE without blocking while routers update their distances to reach optimum values contrasts with the large waves of signaling messages and blocking in DSDV and DUAL. In this example, routers converge to valid routes just as fast as with the topology-broadcast method. Furthermore, fewer signaling messages are needed, and some routers have multiple loop-free routes to destinations.

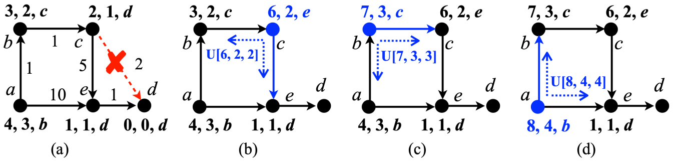

Figure 1: Loop-freedom without blocking in RIPPLE

Figure 4 illustrates the loop-free and fast convergence of RIPPLE after node failures or network partitions. The distance, hop reference, successor, and minimum requested reference for destination d are indicated next to each router. Successors to destinations are indicated by arrowheads. An update sent by router k regarding destination d is denoted by $U[\delta^k_d, h^k_d, r^k_d]$, and a probe is denoted by $P[\delta^k_d, h^k_d, r^k_d, n^k_d]$.

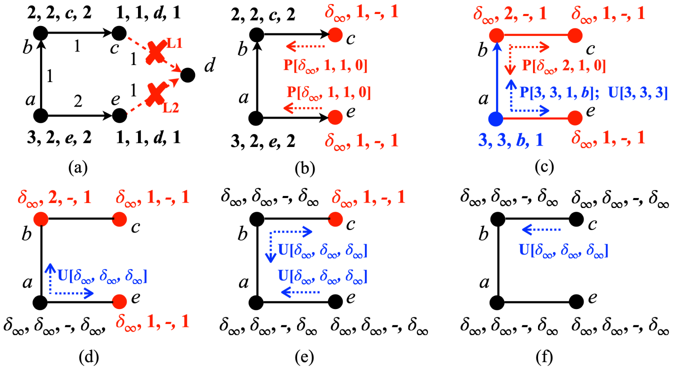
2, 2, c, 2
1, 1, d, 1
b
1
c
1
L1
d
1
a
2
e
1
L2
3, 2, e, 2
1, 1, d, 1
(a)
2, 2, c, 2
δ_∞, 1, -, 1
b
c
P[δ_∞, 1, 1, 0]
a
P[δ_∞, 1, 1, 0]
e
3, 2, e, 2
δ_∞, 1, -, 1
(b)
δ_∞, 2, -, 1
δ_∞, 1, -, 1
b
c
P[δ_∞, 2, 1, 0]
a
P[3, 3, 1, b]; U[3, 3, 3]
e
3, 3, b, 1
δ_∞, 1, -, 1
(c)
δ_∞, 2, -, 1
δ_∞, 1, -, 1
b
c
a
U[δ_∞, δ_∞, δ_∞]
e
δ_∞, δ_∞, -, δ_∞,
δ_∞, 1, -, 1
(d)
δ_∞, δ_∞, -, δ_∞
δ_∞, 1, -, 1
b
c
U[δ_∞, δ_∞, δ_∞]
a
U[δ_∞, δ_∞, δ_∞]
e
δ_∞, δ_∞, -, δ_∞
δ_∞, δ_∞, -, δ_∞
(e)
δ_∞, δ_∞, -, δ_∞
δ_∞, δ_∞, -, δ_∞
b
c
U[δ_∞, δ_∞, δ_∞]
a
e
δ_∞, δ_∞, -, δ_∞
δ_∞, δ_∞, -, δ_∞
(f)

Figure 2: Loop-freedom in RIPPLE after a node failure or network partition

Figure 4(a) shows the state of routers when links (c, d) and (e, d) fail and destination d is unreachable. Figure 4(b) shows that routers c and e become blocked and send probes to their remaining neighbors stating a requested reference of 1. As Figure 4(c) shows, router b becomes blocked but router a perceives router b as satisfying $\mathcal{ROC}$. Accordingly, router a is not blocked, forwards a probe to that neighbor only, and also sends an update to all its neighbors with its new distance and hop reference, and a requested reference equal to its own hop reference. However, the update from router a states a requested reference of 3 and hence cannot serve as a response to the probe that routers b and e sent stating a requested reference of 1. This prevents any looping.

As Figures 4(d) to 4(f) show, starting with router a, all routers receive probes or updates stating distances equal to δ_∞, which makes then stop being blocked and stay at δ_∞ for their distances, hop references, and requested references.

The number of steps needed for all routers to reach a distance of δ_∞ in this example is the same as the number of steps that would be required with the topology-broadcasting method.

However, no loops are created, and only two additional steps are spent in RIPPLE by routers going from the blocked to ordered state.

IV. RIPPLE CORRECTNESS

The following five theorems prove that RIPPLE is loop-free. The sketch of a proof that it converges to shortest paths within a finite time follows this result. As needed, the value of a variable β at time t is denoted by $\beta(t)$.

Theorem 1: A path in which $\mathcal{ROC}$ is satisfied at every router along the path cannot be a loop

Proof: Assume that $\mathcal{ROC}$ is true at every router along a path L. For the sake of contradiction, assume that L is a routing-table loop that excludes destination d at time t and let $L = \{v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_h \rightarrow v_{h+1}\}$, where $v_{h+1} = v_1$. Each router $v_i \in L$ informs its neighbors of its distance to d at a time denoted by t_i, where $t_i < t$, and its neighbors in L use that value at a subsequent time to determine whether $\mathcal{ROC}$ is satisfied. The time when router $v_i \in L$ makes router $v_{i+1} \in L$ a next hop to d is denoted by t_i^+ and $t_i^+ \leq t$, which implies that $s_d^{v_i}(t) = s_d^{v_i}(t_i^+)$, $\delta_d^{v_i}(t) = \delta_d^{v_i}(t_i^+)$, and $h_d^{v_i}(t) = h_d^{v_i}(t_i^+)$ for all $v_i \in L$.

The fact that $\mathcal{ROC}$ must be satisfied at each router $v_i \in L$ implies that, for all $v_i \in L$:

$$h_d^{v_i}(t_i^+) = h_d^{v_i}(t) \geq h_{dv_{i+1}}^{v_i}(t) = h_{dv_{i+1}}^{v_i}(t_i^+) = h_d^{v_{i+1}}(t_{i+1}) = h_d^{v_{i+1}}(t) \tag{3}$$

Eq. (3) implies that, for L to be a loop, it must be true that $h_d^{v_i}(t) = h_d^{v_{i+1}}(t)$ for every $v_i \in L$. Accordingly, for $\mathcal{ROC}$ to be true at every router along a path L, it must be true that $\delta_{dv_{i+1}}^{v_i}(t) < \delta_d^{v_i}(t)$. However, this is a contradiction, because it implies that $\delta_d^{v_i}(t) < \delta_d^{v_i}(t)$ for all $v_i \in L$ if L is a loop. Therefore, the theorem is true. ■

Theorem 2: No routing-table loop can be created in RIPPLE when routers transition from ordered to blocked state.

Proof: The proof is immediate from the definition of how RIPPLE operates, because a router that is blocked does not have a successor. ■

Theorem 3: RIPPLE is loop-free for any destination d.

Proof: If $\mathcal{ROC}$ is always satisfied at every router, then it follows from Theorem 1 that no routing-table loops can form. It also follows from Theorem 2 that no routing loop can occur when routers in a path become blocked. Thus, the proof needs to show that no routing loop can be created when a router transitions from blocked to ordered state.

For a router k to become ordered once it is blocked, it must receive an update or a response such that $\mathcal{ROC}$ is satisfied, and a router $n \in N^k$ can send an update or a response to router k only if it is ordered itself. The path from n to d either consists of routers that are ordered, or consists of both blocked and ordered routers. In the first case, it follows from Theorem 1 that router k cannot create a loop by setting $n = s_d^k$ because then the path from n to d is loop-free and extending that path with link (k, n) cannot create a loop. In the second case, the path from n to d is the concatenation of subpaths, each consisting of one or more routers that are all ordered or are all blockede, and it follows from Theorems 1 and 2 that such subpaths are loop-free and hence extending the path from n to d with link (k, n) cannot create a loop. ■

Theorem 4: RIPPLE converges to shortest routes for all reachable destinations and converges to δ_∞ for all unreachable destinations within a finite time after network changes stop occurring in a finite network.

Sketch of Proof: The proof uses the fact that RIPPLE is loop-free at every instant (Theorem 3) to show that routers receive valid shortest distances from their neighbors to reachable destinations, because updates, probes and responses must propagate over loop-free paths. By the same token, loop-freedom is used to show that each router converges to δ_∞ because there can be no simple path starting from a destination neighbor to any router in a connected component that does not include the destination. ■

V. COMPARING RIPPLE WITH AN OPTIMUM APPROACH

Typically, protocol complexity (e.g., the worst-case number of steps and messages needed for convergence) is used for this purpose. Unfortunately, complexity alone does not provide insight on the blocking or looping that occurs while a protocol is converging, and does not indicate how the performance of a protocol compares with the best performance possible.

Accordingly, we define two new performance metrics based on how the performance of a routing approach deviates from a notional optimal routing method (**NORM**) that incurs the smallest number of steps and messages to converge without looping after a single link change or node change. We call these metrics blocking optimality (**BO**) and communication optimality (**CO**).

BO is the additional number of steps after a link change during which looping or blocking persists for a given destination compared to a NORM, and **CO** is the additional number of messages sent compared to a NORM. These two metrics are strong indicators of the signaling overhead and how fast a routing protocol restores valid routing state after input events.

In the following, H denotes the network diameter in number of hops, A is the largest degree of a router, WP_d^k is the weight of path P_d^k, N is the number of routers in the network, and T is a timer delay that by design must be much longer than the number of steps needed for an update to traverse the network, i.e., $T = \Omega(H)$. For simplicity, we also assume the routing methods operate in a synchronous manner.

A. NORM

We assume that routers executing NORM somehow can differentiate a link failure from a node failure. This functionality does not exist in a practical routing method, of course, and is used to derive a lower bound on the number of steps during which routers are blocked. We also assume that the signaling in NORM is such that, in the worst case, any router with a path that includes a router that changes its distance to a destination or a link that changes its status or weight receives the corresponding routing information in H steps independently of the weights of the links and without incurring routing loops. This may be not be possible to attain in a practical routing method and is used also to derive a lower bound on the number of steps during which routers are blocked.

Figure 5 shows a generic topology in which link costs are such that the shortest path from router n_{N-1} to destination d involves all other routers in the network. The number of hops to destination d at each router along the shortest path from the router to d is indicated in parenthesis next to the router. Destination d has only two neighbors, namely routers n_1 and n_{N-1}. The weight of link (n_{N-1}, d) is assumed to be $l(n_{N-1}, d) > WP_d^{n_{N-1}}$ and makes path $P_d^{n_{N-1}}$ include all routers in the network. A link in dashed lines (e.g., the link between n_w and n_{N-v}) indicates the possibility of having such a link, and exists only when stated in an example.

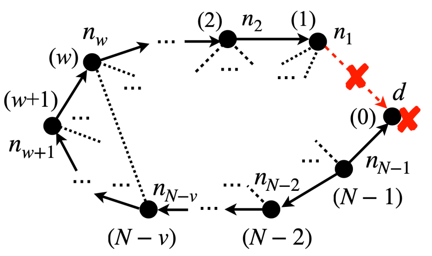

Figure 3: Sample topology for comparisons

Case 1: **A link-weight increase that affects one or more paths to destination d:** A NORM does not incur any blocking after a link-weight increase. Routers somehow know to report larger distances in updates and eventually report shortest paths when they are attained through the propagation of updates.

Case 2: **A link failure affecting all routes to destination d:** Assume that no dashed links exist and link (n_1, d) fails. According to a NORM, a wave of updates reporting δ_∞ propagates to all routers including router n_{N-1} taking up to H steps, and router n_{N-1} then issues an update stating $\delta_d^{n_{N-1}} = l(n_{N-1}, d)$ and a wave of updates taking up to H steps percolates to all other routers allowing them to correct their routes to destination d. Accordingly, a NORM would take $O(2H)$ steps and $O(2H \times A)$ messages to have no router blocked after the failure of link (n_1, d).

Case 3: **A destination failure or network partition:** A NORM would take $\Theta(H)$ steps and $\Theta(H \times A)$ messages for all routers to declare a destination to be unreachable after it fails or becomes unreachable after a network partition. By assumption, no routing loops would be formed during any of the steps needed for convergence.

B. RIPPLE

Case 1: Routers do not experience blocking in RIPPLE after any link-weight increase, because a neighbor in the successor set of a router satisfies $\mathcal{ROC}$ after the processing of any input event. Accordingly, RIPPLE has $BO = \Theta(1)$ and $CO = \Theta(1)$ in this case.

In the worst case, for routers to obtain correct distances after link (n_1, d) changes its weight, probes must propagate along the reverse shortest paths from router n_1 to router n_{N-1} and then responses propagate back from router n_{N-1} all the way to router n_1. Accordingly, RIPPLE takes $O(2N)$ steps and $O(2N \times A)$ messages to converge in this case.

Case 2: In the network of Figure 5, router n_1 sends a probe $P[d, \delta_\infty, 1, 1, 0]$ that causes probes to propagate all the way to n_{N-1} taking $N-1$ steps. Router n_{N-1} then sends response $U[d, \delta_d^{n_{N-1}}, 2, 1]$ with $\delta_d^{n_{N-1}} = l(n_{N-1}, d)$. This causes updates to be sent back all the way to n_1 taking an additional $N-1$ steps, and then router n_1 issues an update reflecting its new path going through n_2. Accordingly, RIPPLE takes $O(2N)$ steps and $O(2N \times A)$ messages to converge, and hence $BO = O(2(N-H))$ and $CO = \Theta(2(N-H)A)$ in this case.

Case 3: After the failure of destination d or the concurrent failure of links (n_1, d) and (n_{N-1}, d), it takes $O(N)$ steps and $O(N \times A)$ messages for all routers to set their distances to destination d equal to δ_∞. This is the case because all routers must receive probes from their next hops stating distances equal to δ_∞ and a requested reference equal to 1, the routers must forward those probes, and routers that only have neighbors reporting distances of δ_∞ must become passive silently with a distance and reference distance equal to δ_∞. Therefore, RIPPLE has $BO = O(N-H)$ and $CO = O((N-H)A)$ in this case.

VI. Conclusions

RIPPLE is a new shortest-path routing protocol that is simple and attains near-optimum convergence speed and loop-free multi-path routing using only distance information.

RIPPLE substitutes destination sequence numbers with hop-count references to define a new ordering condition that does not incur the extensive blocking required in protocols based on destination sequence numbers. RIPPLE substitutes diffusing computations [1], which require a router to receive replies from all its neighbors to satisfy a query, with probes that can be answered by the first valid response.

RIPPLE was proven to be correct, and to attain near-optimal performance after changes in the status or weights of links. Furthermore, in terms of blocking, RIPPLE is inherently more efficient than routing protocols intended to be loop-free and based on either the use of destination sequence numbers or multi-hop router coordination. This is the case because RIPPLE does not require sequence numbers to be reset throughput a network for distances to be trusted, and it does not force routers to wait for replies from all their neighbors before they can trust new distances.

References

[1] J.J. Garcia-Luna-Aceves, "Loop-Free Routing Using Diffusing Computations," *IEEE/ACM Trans. Networking*, 1993.

[2] J.J. Garcia-Luna-Aceves, "THORP: Choosing Ordered Neighbors To Attain Efficient Loop-Free Minimum-Hop Routing," *Proc. IEEE LANMAN '22*, July 2022.

[3] C. E. Perkins and P. Bhagwat, "Routing over Multihop Wireless Network of Mobile Computers," *Proc. ACM SIGCOMM '94*, 1994.