Loop-Free Link-State Routing

Pierre Fransson Lenka Carr-Motyčková

{pierre, lenka}@sm.luth.se

Division of Computer Science and Networking

Department of Computer Science and Electrical Engineering

Luleå University of Technology

SE - 971 87 Luleå

Sweden

KEYWORDS – *Loop-free shortest path routing, distributed algorithms, analysis of algorithms, telecommunications and Networking*

ABSTRACT

Improving the robustness of today's intra-domain link-state networks is of increasing importance. This is because it is becoming commonplace for such networks to carry telephony, on-demand video and other kinds of real-time traffic. Since user requirements on real-time traffic are exacting, it is essential that the network is able to respond rapidly to network errors.

A way of reaching fast reaction times in response to network errors is to precompute fail-over paths that can be utilized immediately when an error is detected. The fail-over path is thereafter in use until the network has converged, handling traffic that would otherwise have been lost due to the error. During convergence (when new paths are calculated) there is a caveat that needs to be avoided - the formation of temporary loops. Temporary loops can form naturally during convergence in link-state networks and can unfortunately prevent traffic from reaching a fail-over path. Traffic can therefore be lost even though a fail-over path is in place.

We present an algorithm that can be used in conjunction with link-state routing, to ensure that temporary loops do not form during convergence. The algorithm improves on previous loop-free algorithms for link-state routing by reducing the number of state transitions necessary before a router can update its routing table to a new, guaranteed to exist, path for a wide variety of topological configurations. Because all new paths are guaranteed to exist at the time of a routing table update, the algorithm is also more robust against loss of data than the recently proposed loop-free algorithm by Francoise and Bonaventure.

I. INTRODUCTION

Link-state routing protocols (e.g., OSPF [1] and IS-IS [2]) are today widely deployed for use within large intra-domain networks on the Internet. Their popularity most likely stem from being regarded as flexible and efficient, i.e., converging fast and not consuming too much networking resources [3], [1]. Given the profusion of networks utilizing link-state routing, it is apparent that their behavior has to be taken into

account when considering end-to-end traffic behavior on the Internet. This is especially true for real-time traffic, for which the Internet is becoming increasingly relied upon to transport.

Even though link-state routing has previously been regarded as quite efficient at recalculating routing paths when the network topology changes, this situation has of late changed. Newer applications (such as Voice-over-IP) have more stringent requirements on convergence speed, which cannot in general be met by "standard" link-state routing¹. Link-state networks require that information regarding a network topology change reaches every router in the network before the routing process is guaranteed to have successfully converged. The reaction time of the entire network therefore becomes a function of number of routers and subnets in the network and the topology of the network. Waiting for the entire network to converge before an error can be handled is problematic, since it does not scale well. As the network grows it becomes increasingly difficult to reach a constant reaction time of a few hundred milliseconds.

A viable alternative, capable of scaling to arbitrarily large networks, is instead to use fail-over routing in conjunction with link-state routing. Fail-over routing typically involves creating temporary local paths that can be used when a network error occurs. The fail-over paths are usually computed ahead of time to be utilized immediately when a network error is detected. They can be implemented either at the link layer, as in the case of MPLS-tunnels, or at the network layer, as in the case of [4]. The fail-over paths are only meant to be in operation during the convergence process, during which all routers in the network calculate new shortest paths and update their routing tables. Recalculating new shortest paths is necessary, since the fail-over path now presents a single point of failure, is quite likely a suboptimal path for many routers and may also be inadequately dimensioned in terms of bandwidth.

Unfortunately there exists a caveat that can negate the effect of fail-over routing. Even though link-state routing is guaranteed to be loop-free when the network has converged, temporary routing-loops can occur during the convergence phase. If such loops form, the traffic that is meant to be handled by a fail-over path can be affected. Traffic affected by a

¹Real-time applications typically require network faults to be repaired within a few hundred milliseconds.

routing loop is at the very least delayed but can also ultimately be lost. This suggests that in order for fail-over routing to function properly there needs to be a way of avoiding creating temporary loops while the network is converging.

This paper presents an algorithm that can be used in conjunction with link-state routing to prevent the creation of temporary loops during convergence. The algorithm is called the Loop-Free Link-State Update algorithm (LFLSU). The LFLSU-algorithm is based on message passing, using *query* and *reply*-messages between neighboring routers. The algorithm commences at a router as soon as a new link-state information has been received by the router either through a link-state message or a query message. The algorithm is proven to be loop-free during the convergence process and to be dead-lock-free.

The number of transitions caused by the forwarding of link-state messages can be used to compare how fast different loop-prevention algorithms enable routers to update their routing tables. The fewer transitions that are needed the faster the algorithm is. It is important that a loop-prevention algorithm is as fast as possible, since (as pointed out above) the fail-over path presents a single point of failure, may also be inadequately dimensioned and can therefore potentially cause loss of traffic. In this respect, the LFLSU-algorithm improves on previous loop-prevention algorithms [5], [6]. It reduces the number of state transitions necessary before a router can update its routing table to a new, guaranteed to exist, path for a wide variety of topological configurations. At worst the LFLSU-algorithm requires the same amount of transitions as previous algorithms before routers can update to a new path that is guaranteed to exist. Section II presents further differences between the LFLSU-algorithm and previous algorithms.

II. BACKGROUND

The occurrence of loops in routing algorithms is a fairly well studied area [7], [8], [9], [10], [11] that was active already in the 80's. However, the main focus has been on making distance-vector algorithms loop-free. This focus can likely be attributed to the popularity of distance-vector algorithms in the early days of the Internet [12, Ch. 4.6.1] and the fact that these algorithms suffer from the so-called count-to-infinity problem² [13]. An example of an improved version of a distance-vector routing algorithm that resulted from this research is the diffusing update algorithm [14], DUAL.

The work on loops in link-state routing protocols has not seen the same amount of attention. This is most likely due to that the loops are only temporary in nature, existing only during convergence. However, there has been some work done in the area, both in the past and in current research.

The work by Francois and Bonaventure in [6] assumes that an existing fail-over path is in place, handling the traffic on the paths that were affected by a network change. Since the fail-over path is already in place, it is reasoned that it does not

matter if routers update their routing tables in an order that allows the traffic on the affected paths to continue towards the network fault as long as routing loops are avoided. This allows for a simple and effective strategy of determining the order in which routers should update their routing tables. The algorithm ensures that a link-state message is propagated to the leave(s) of the induced sink-tree (with the failed link/node as root), before any nodes are allowed to update their routing tables. Since downstream routers are not allowed to update their routing tables before upstream routers are, this prevents any loops from forming. Approximately, this is ensured by making each router R , that receives a link-state message, query its neighbors as to whether any of them have it (i.e., router R) on the path to the failed link/node. If there exists any such neighbors they will reply with an affirmation, informing the router of which its upstream neighbors are in the induced sink-tree. The router must thereafter wait for each such upstream neighbor for a notification that they have updated their routing tables. After this it is safe for the node to update its routing table. Router R thereafter in turn notifies any downstream neighbors that it might have had. The stated major benefits of this approach is that it has low message and bit complexity, requires little computational power and does not force incremental updates to the routing tables.

The paper by Garcia-Luna-Aceves [5] describes a unified approach to loop-free routing for both distance-vector and link-state routing. The work is based on *diffusing computations* [15] and even though it both covers distance-vector and loop-free routing, focus is mainly on the distance-vector part. The algorithm by Garcia-Luna-Aceves and the algorithm by Francois and Bonaventure are actually quite similar in their basic structure; even though Garcia-Luna-Aceves does not explicitly state the existence of a fail-over path. The fundamental core of both algorithms is that they prevent loops from being formed by enforcing that downstream routers have to wait for upstream routers before they can update their routing tables (as described above). Also, both algorithms do not allow/require routers to perform incremental updates, for the algorithm by Garcia-Luna-Aceves this only applies to the Link-state case (not the distance-vector case). However, there exists some notable differences between the algorithms. The algorithm by Garcia-Luna-Aceves requires that a feasibility condition (FC) [5] has to be met before routers can update their routing tables. If the FC is met a router can safely update its routing table without waiting for upstream routers. The FC is therefore basically an optimization which allows the algorithm by Garcia-Luna-Aceves to be faster than the algorithm by Francois and Bonaventure for several topological configurations. Also, the work by Garcia-Luna-Aceves does not mention how entire router failures could be handled which both the algorithms by Francois and Bonaventure and the LFLSU-algorithm handles.

Comparing the LFLSU-algorithm to the algorithms by Garcia-Luna-Aceves and Francois and Bonaventure the following can be noted. In analogue to the work by Francois and Bonaventure, the LFLSU-algorithm also requires that there

²The count-to-infinity problem causes semi-persistent routing loops to form during routing convergence.

exists a fail-over path that handles traffic affected by a network change. However, as stated in the Introduction, the LFLSU-algorithm reduces the number of state transitions necessary before a router can update its routing table to a new, guaranteed to exist, path for a wide variety of topological configurations. The reason for this is that the LFLSU-algorithm does not enforce the same ordering of routing table updates that the other algorithms do, i.e., making downstream routers wait for upstream routers in the induced sink-tree. The LFLSU-algorithm instead directly computes the neighbor(s) for which it is dependent upon to be able to update its routing table safely, see Section III-C regarding details, allowing the LFLSU-algorithm to update its routing table as soon as it has received a reply from those specific neighbors. This approach is more powerful than the FC in the algorithm by Garcia-Luna-Aceves, since it not only identifies the paths found with the FC but also identifies paths that can not be found with the FC.

Additionally, the LFLSU-algorithm only allows routers to update their routing tables when a new path is guaranteed to exist. For such paths the LFLSU-algorithm is always faster or as fast as previous algorithms. However the algorithms by Garcia-Luna-Aceves and Francois and Bonaventure can be faster in certain cases when an update is made to a new path that is not guaranteed to exist yet. As shown in [5] and [6] this does not lead to the formation of temporary routing loops. However, this behavior may under certain circumstances (when an entire router fails) lead to loss of traffic, see Section V for a detailed description. Since the LFLSU-algorithm does not allow such updates it can be said to be more robust.

Another difference compared to the previous algorithms is that the LFLSU-algorithm can be run in two modes: incremental or non-incremental, thereby making it more flexible. In incremental mode the algorithm enables routers to make incremental updates to their routing tables, i.e., updating a subset of its routing entries. In non-incremental mode the algorithm requires that routers update their entire routing table. In both modes the above stated properties hold true compared to previous algorithms. However, the incremental mode is generally faster than the non-incremental while the incremental mode has a higher message complexity than the non-incremental mode. The advantage of supporting incremental updates is that even if it is not possible to redirect all traffic that is handled by the fail-over path at a given point in time, it may be possible to redirect some of the traffic. This becomes important in the case where the fail-over path is not adequately dimensioned, since the removal of some of the traffic reduces to load on the fail-over path and may therefore ultimately stop data from being lost. However, it should be noted that in the case where there are many paths that need to be updated, incremental updates could create greater strain on the routers in the network. Determining which mode is appropriate would be up to the network management. Herein we only present the incremental version of the algorithm. Modifying the algorithm to non-incremental mode is straightforward. The only necessary modification is to require that each router must wait until it can reply to a preceding query in its entirety.

In non-incremental mode the message complexities of the LFLSU-algorithm and the algorithm by Francois and Bonaventure are equal ($O(|E|)$), while the bit complexity of the LFLSU-algorithm is larger. For the incremental mode both the message and bit complexities are larger for the LFLSU-algorithm. Section IV presents a detailed comparison of the message and bit complexities involved. The comparison is only made against the algorithm by Francois and Boneventure since it has the best bit complexity.

III. LOOP-FREE LINK-STATE UPDATE ALGORITHM

This section presents the Loop-Free Link-State (LFLSU) algorithm, that prevents formation of temporary loops in link-state routing algorithms.

In order to elucidate the presentation of the algorithm, terminology from graph theory is used. Real network entities, such as routers, point-to-point links and shared media networks are therefore described in graph terms. Routers are represented as nodes and links between routers are represented as directed edges.

A. Definitions

1) *General Graph Definitions:* The pair $G = (N, E)$ is a directed graph, where N is the finite set of **nodes** in the graph and E is the set of directed **edges** in the graph. A directed edge is represented by an ordered pair (u, v) , where $u \in N \wedge v \in N$.

If (u, v) is an edge in graph G , then node v is said to be **adjacent to** node u , denoted by $u \rightarrow v$. The node u is defined as a **neighbor** of v and vice versa, if either u is adjacent to v or/and v is adjacent to u . The **out-degree** of a node is the number of edges leaving it and the **in-degree** of a node is the number of edges entering it. The **degree** of a node is simply defined as the sum of the out-degree and the in-degree.

A **path** in graph $G = (N, E)$ from node u to u' is a sequence of nodes $\langle v_0, v_1, v_2, \dots, v_k \rangle$ such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$. The node u' is said to be reachable from u if there is a path p from u to u' , denoted by $u \xrightarrow{p} u'$. The **weight** of an edge is a function $w : E \rightarrow \mathbf{R}$ that maps all edges in E to real-valued weights. The **weight** of a path, $d(p)$, $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$ is the sum of the weights of the edges in the path. A **shortest path** from u to v is defined as any path with the **shortest-path weight** $\delta(u, v)$ defined by:

$$\delta(u, v) = \begin{cases} \min\{d(p) : u \xrightarrow{p} v\} & \text{if } \exists u \xrightarrow{p} v, \\ \infty & \text{otherwise.} \end{cases}$$

$NextHop_u(d)$ is defined to be the second node on the shortest path p from node u to destination d , i.e., if $p = \langle v_0, v_1, v_2, \dots, v_k \rangle : u = v_0 \wedge d = v_k$ or if $p = \langle v_0, v_1 \rangle : u = v_0 \wedge d = v_1$ then $NextHop_u(d) = v_1$.

2) *Definitions Regarding Trees:* A rooted directed tree $T(r) = (N_r, E_r)$, $N_r \subseteq N \wedge E_r \subseteq E$ of graph $G = (N, E)$, is directed tree in which one of the nodes, r , is distinguished from the others. Node r has an in-degree of zero and all other nodes in the tree have an in-degree of one. The distinguished

node, r , is called the **root** of the tree. Node u is an **ancestor** of node v , in a rooted tree $T(r)$, if it is on the unique path from node r to v . If u is an ancestor of v , then v is a **descendant** of u ; every node in a rooted directed tree is both an ancestor and descendant of itself. If $u \neq v$ and u is an ancestor of v then u is said to be a **proper ancestor** of v and v is a **proper descendant** of u .

A **subtree** $T(r', r) = (N_{r'}, E_{r'})$ of the rooted tree $T(r)$ (with root r), is itself a rooted tree (with root r'), where $r' \in N_r$. Furthermore, $v \in N_{r'}$ iff v is a descendant of r' in $T(r)$ and $(u, v) \in E_{r'}$ iff $u \in N_{r'} \wedge v \in N_{r'}$ and $(u, v) \in E_r$. An **adjacent subtree** $T_{adj}(r', r)$ of the rooted tree $T(r)$, is a subtree of $T(r)$ where r' is adjacent to r in $T(r)$.

A **shortest path tree** $SPT(s)$ is defined as a rooted tree with root/source s , where the paths from s to all other nodes in $SPT(s)$ are minimum weight paths in G . The shortest path tree also contains the weight of each such minimum weight path. Also, it is assumed herein that every shortest path in the graph G can be determined uniquely and consistently for all nodes, even when equal cost shortest paths exist to a destination. In the case of equal cost shortest paths, it is possible to make paths unique by preferring the path containing the lowest node identifier.

Throughout the remainder of this paper the general case link-state routing system will be viewed as a distributed system, modeled as a *transition system* [16] with asynchronous message passing. It is also assumed that exactly one process is run at each node in the graph $G = (N, E)$.

3) Definitions Regarding Networking Events and Messages:

A networking event (*Evnt*) is taken to mean any of the following: a node or edge failure, an addition of a node or an edge, or an edge cost change. Each networking event is uniquely distinguished by the identifier I

All messages are defined as tuples. There are three types of messages that are part of the algorithm: link-state messages: $\langle \mathbf{LS}, \text{Evnt}, I \rangle$, query messages: $\langle \mathbf{Query}, \text{aff}Q_{uv}, I \rangle$ and reply messages: $\langle \mathbf{Reply}, \text{lpfree}, I \rangle$. The first value of the tuple is a constant that identify the type of message. The last value of each message tuple is an identifier I , which uniquely identifies to which networking event each query and reply message is related to. The significance of the other message values are described in Section III-D.

B. Formation of Temporary Loops in Link-State Routing Algorithms

Link-State algorithms operate by disseminating link-state messages containing new local topology information³ to all other nodes in the graph. A commonly used algorithm to disseminate link-state messages in real networks is to employ *flooding* [1]. After a node has received link-state messages from all other nodes, it assembles a global topology map from the set of all link-state messages. The global topology map is then used by the node to locally compute a shortest path tree

for all destinations⁴ after which the routing table of the node is updated.

When a networking event occurs (e.g., an edge or node fails), the nodes which are directly connected to the edge or node will generate new link-state messages that are then disseminated to all other nodes. From the moment the networking event occurs until all nodes have received the link-state messages, calculated new shortest path trees and updated their routing tables, the algorithm is said to be *converging*. After this the algorithm has *converged*.

It is easy to realize that a link-state algorithm that has converged does not contain any routing loops. However, during convergence, temporary routing loops can form. The reason is that since the link-state messages usually do not reach all nodes simultaneously, all nodes will not have access to the same global topology map during convergence. Leaving the nodes with inconsistent views of the global topology. Once a node u has received a link-state message it may update a shortest path to a destination d so that that the path traverses node v , that has not yet received the link-state message. Node v may in this case have a shortest path to destination d that traverses node u , thereby creating a temporary loop.

The problem of temporary loops in link-state algorithms is not only a theoretical problem, but also one which has caught the attention of the Routing Area Work Group within the Internet Engineering Task Force (IETF). The problem has been dubbed microloops and there exists a draft [17] that addresses the issue of such micro loops (temporary loops).

C. Detection of Networking Events

This section presents the types of networking events that the LFLSU-algorithm can handle and a method to determine the set of *affected destination nodes* (hereafter referred to as *affected nodes*). The set of affected nodes is defined as all the destinations to which a node has a new shortest path to, after it has received a link-state message and recalculated its shortest path tree.

The types of networking events that can be handled by the LFLSU-algorithm can be divided into two main categories, edge (uni- and bidirectional) events and node events. The edge events include: edge failures, edge cost decreases, edge cost increases and edge additions. The node events include: node failures and node additions. The networking events described above all affect different portions of the original shortest path tree (hereafter denoted $SPT_{old}(s)$) for any node s in the graph. For instance, in the case of a failed edge (u, v) the entire subtree $T(v, u)$ of $SPT_{old}(s)$ is affected, while for a cost increase only a subset of $T(v, u)$ may be affected.

To determine the set of affected nodes that will have new shortest paths due to a networking event, it simply suffices to retain the original shortest path tree $SPT_{old}(s)$ while computing the new shortest path tree $SPT_{new}(s)$. The set of affected nodes can then be calculated locally in $O(|N| \lg |N|)$ time, by performing a depth first search on $SPT_{old}(s)$ and at

³i.e., about the edges/nodes that are directly connected neighbors

⁴Dijkstra's Shortest Path First algorithm [15] is often used for this purpose

the same time performing the same traversal in $SPT_{new}(s)$. If an edge transition (x, y) performed in $SPT_{old}(s)$ is not possible in $SPT_{new}(s)$ then it is possible to conclude that all the nodes in the subtree $T(y, x)$ of $SPT_{old}(s)$ are affected. The complexity comes from locally sorting both shortest path trees (using the node identifiers) to be able to efficiently traverse both trees at the same time.

A crucial practical concern in the case of networking events regarding node failures, is the ability to detect that it is the neighboring node that has failed and not the edge connecting to the neighbor. While it is obvious that this distinction may not always be possible, the cases that are distinguishable have attracted enough interest to be addressed by the networking community. A working group within the Internet Engineering Task Force (IETF) are investigating the problem. They propose a protocol [18] capable of detecting errors of bidirectional edges between nodes and, when possible, errors that affect the nodes themselves.

D. Loop-Free Link-State Update (LFLSU) Algorithm

The basic idea behind the LFLSU-algorithm is to refrain from using newly calculated shortest paths until it is certain that they are safe to use. This is accomplished by controlling the order in which nodes are allowed to make use of new calculated shortest paths, i.e., update their routing tables. The algorithm ensures that when a node u updates its routing table to a destination node d , using a new shortest path, the following condition is met: all nodes that are down-stream from u on the new shortest path to d have either updated their routing tables for destination d or do not need to update their routing tables for destination d . Until this condition can be met and the routing table is updated, all nodes continue to forward incoming traffic, destined for d , according to the old shortest path.

In order to ensure that calculated paths are loop-free, the LFLSU-algorithm relies on the underlying link-state algorithm to provide two mechanisms. First, all link-state messages must be reliably disseminated to all nodes in the network. In the case of OSPF this is handled using *reliable flooding* [1]. Second, there needs to be a way to reliably communicate with adjacent nodes. In an IP-network this could, for instance, be accomplished through TCP [19]. The LFLSU-algorithm also requires that all nodes have already calculated a shortest path tree, SPT_{old} , before the first networking event occurs.

The loop-free algorithm consists of three procedures: a *link-state procedure*, a *query procedure* and a *reply procedure*. The first time a node u discovers a new link-state message, $\langle \mathbf{LS}, Evnt, I \rangle$, it calls the link-state procedure with the link-state message as input. All messages contain the identifier I , which serves to ensure that query and reply messages are sent in response to the correct link-state message.

The first step taken by the link-state procedure at node u is to compute a new shortest path tree $SPT_{new}(u)$, based on the networking event $Evnt$, and compare it to the original shortest path tree $SPT_{old}(u)$ in order to determine the set of *affected destinations* (denoted Aff_u) to which there now

are new shortest paths. The link-state procedure can at this stage determine if there are any safe paths that it can use. Any destination d_1 that is not in Aff_u is safe to use since it has not been affected by the event; the routing table of u is therefore updated with these destinations. Also, any destination d_2 where $d_2 = NextHop_u(d_2)$ is also safe to use immediately even if it is in Aff_u , since the destination is reached directly. The routing table of u is therefore also updated with these destinations. Since the destinations belonging to the latter category are safe to use, they are directly removed from Aff_u . However, for all other destinations still in Aff_u the link-state procedure must send out query messages $\langle \mathbf{Query}, affQ_{un_i}, I \rangle$ to subset of its neighbors ($Neigh_u$ ⁵), inquiring as to whether the paths to the destinations are safe to use. Each query message from node u to neighbor n_i (where $i \in 1 \dots |Neigh_u|$) includes $affQ_{un_i}$ which is a subset of Aff_u . Destination $d \in textaffQ_{un_i}$ if $NextHop_u(d) = n_i$. Only one query-message per neighbor is sent.

The query procedure is run at node n_i when a query message, $\langle \mathbf{Query}, affQ_{un_i}, I \rangle$, is received. The query procedure first checks to see if there are any destinations in $affQ_{un_i}$ that are not in Aff_{n_i} . Any such nodes are guaranteed to be loop-free since they are not in the set of affected nodes of node n_i . Any such destinations are then stored in the set $lpfree_u$. In the case where $lpfree_u$ is non-empty, the query procedure sends a reply message, $\langle \mathbf{Reply}, lpfree_u, I \rangle$, to node u containing the destinations in $lpfree_u$. If there are destinations in $affQ_{un_i}$ that are also in Aff_{n_i} then the query procedure of node n can not send a reply for these destinations, since it cannot guarantee that they are loop-free. Instead it stores those destinations in an array of sets, $Waiting[u]$, indexed by the node that sent the query message. Storing the destinations allows node n_i to send a reply message at later stage when it has learned that a destination in $Waiting[u]$ is guaranteed to be loop-free.

The reply procedure is run at node u when a reply message, $\langle \mathbf{Reply}, lpfree_u, I \rangle$, is received. The reply message first removes the destinations in $lpfree_u$ from Aff_u and updates the routing table to these destinations. The reply procedure thereafter checks $Waiting[n_i]$ for each of its neighbors, n_i , to see if any of the neighbors have made a query for any of the destinations in $lpfree_u$. If so, node u sends a reply message, $\langle \mathbf{Reply}, lpfree_{n_i}, I \rangle$, to each of the those neighbors, where $lpfree_{n_i} = Waiting[n_i] \cap lpfree_u$.

When node u no longer has any destinations left in Aff_u and $Waiting[\cdot]$ only contains empty sets the algorithm terminates.

E. Pseudo code

As described above, the loop-free algorithm consists of three procedures: a link-state procedure, a query procedure and a reply procedure. The algorithm also employs four functions. The function $RecomputeSPT()$ computes the new shortest path tree based on the networking event $Evnt$. The function $FindAFF(SPT_{new}, SPT_{old}, n)$ determines the set of *affected destinations*, see section III-C, for each neighbor n that

⁵ $Neigh_u$ denotes the set of neighbors of u

is a subtree root in SPT_{new} (i.e., $T_{adj}(n, u) \subseteq SPT_{new}(u)$). If n is not a subtree root in SPT_{new} , $\text{FindAff}()$ returns the empty set. The function $\text{FindN}()$ finds the destinations that are affected but that have $d = \text{NextHop}_u(d)$. The function $\text{UpdateRoutingTable}()$ uses the set of affected nodes and the new shortest path tree to determine and update the routing table with safe destinations. As described above the path to destination d is safe if $d \notin \text{Aff}_u[\cdot]$ where $\text{Aff}_u[\cdot] = \bigcup_{x \in \text{Neigh}_u} (\text{Aff}_u[x])$.

Algorithm 1: Link-State Procedure (node u)

```

var  $Id_u$       : Integer
     $G$          : Tuple of two Sets,  $(N, E)$ 
     $SPT_{old}$     : Tuple of two Sets,  $(N', E')$ 
     $SPT_{new}$     : Tuple of two Sets,  $(N'', E'')$ 
     $\text{Aff}_u[\cdot]$    : Array of Sets of nodes
     $\text{Neigh}_u$    : Set of neighbors of  $u$ 
     $\text{Waiting}[\cdot]$ : Array of Sets of nodes
     $lpfree$     : Set of nodes
     $affQ$       : Set of nodes

Processing a  $\langle \text{LS}, \text{Evtnt}, I \rangle$  from neighbor  $v$ :
1  $Id_u \leftarrow I$ 
2  $SPT_{new} \leftarrow \text{RecomputeSPT}(G, \text{Evtnt})$ 
3 foreach  $n \in \text{Neigh}_u$  do
4    $\text{Aff}_u[n] \leftarrow \text{FindAff}(SPT_{new}, SPT_{old}, n)$ 
5    $\text{Aff}_u[n] \leftarrow \text{Aff}_u[n] \setminus \text{FindN}(SPT_{new}, \text{Aff}_u[n])$ 
6  $\text{UpdateRoutingTable}(\text{Aff}_u[\cdot], SPT_{new})$ 
7 foreach  $n \in \text{Neigh}_u$  do
8   if  $\text{Aff}_u[n] \neq \emptyset$  then
9      $affQ \leftarrow \text{Aff}_u[n]$ 
10    send  $\langle \text{Query}, affQ, I \rangle$  to  $n$ 

```

Algorithm 2: Query Procedure (node u)

```

Processing a  $\langle \text{Query}, affQ_{vu}, I \rangle$  from neighbor  $v$ :
1 if  $Id_u \neq I$  then
2   wait until  $\langle \text{LS}, \text{Evtnt}, I \rangle$  is received  $\wedge$  Processing  $\langle \text{LS}, \text{Evtnt}, I \rangle$ 
   is complete
3  $lpfree \leftarrow \emptyset$ 
4  $\text{Waiting}[v] \leftarrow affQ_{vu}$ 
5 forall  $dst \in \text{Waiting}[v] : dst \notin \text{Aff}_u[\cdot]$  do
6    $lpfree \leftarrow lpfree \cup dst$ 
7 if  $lpfree \neq \emptyset$  then
8   send  $\langle \text{Reply}, lpfree, I \rangle$  to  $v$ 
9    $\text{Waiting}[v] \leftarrow \text{Waiting}[v] \setminus lpfree$ 

```

Algorithm 3: Reply Procedure (node u)

```

Processing a  $\langle \text{Reply}, lpfree_{in}, I \rangle$  from neighbor  $v$ :
1  $\text{Aff}_u[v] \leftarrow \text{Aff}_u[v] \setminus lpfree_{in}$ 
2  $\text{UpdateRoutingTable}(\text{Aff}_u[\cdot], SPT_{new})$ 
3  $lpfree \leftarrow \emptyset$ 
4 foreach  $n \in \text{Neigh}_u$  do
5   if  $\text{Waiting}[n] \cap lpfree_{in} \neq \emptyset$  then
6      $lpfree \leftarrow \text{Waiting}[n] \cap lpfree_{in}$ 
7      $\text{Waiting}[n] \leftarrow \text{Waiting}[n] \setminus lpfree_{in}$ 
8   send  $\langle \text{Reply}, lpfree, I \rangle$  to  $n$ 

```

IV. ANALYSIS

The worst-case analysis for both the incremental and non-incremental modes of the LFLSU-algorithm is quite simple

Fig. 1. Star-like topology

to compute. In the worst case a networking event will cause every node in the graph to label every other node in the graph as affected. This case can arise in a “star”-like topology, see Figure 1, where initially every node in the network uses the central/spoke node of the graph to reach any other node in the graph. If the central node should fail at this stage, then all nodes in the network would have to recompute their shortest paths to every other node. Also, if for each node u we have the set of neighbors Neigh_u equal to set of subtree roots in $SPT_{new}(u)$ and each destination (excepting direct neighbors) belonging to the set Aff_u , then we have the case that exactly one query message will traverse each link in the network. For the incremental mode we have that the resulting worst-case query message complexity becomes $O(|E|)$. The worst-case bit complexity for the query messages is $O(\lg(|N|)|N|^2)$, since the sum of the nodes included in all the queries sent by any node can maximally be equal to $O(|N|)$ and the length of any identifier is $O(\lg(|N|))$. Regarding replies, the maximum number of reply messages that can be received by any node is of course $O(|N|)$ since that is maximum number of affected nodes. This gives a worse-case complexity of $O(|N|^2)$ on the number of reply messages sent. The worst-case bit complexity for the reply messages is equal to the bit complexity of the query messages, $O(\lg(|N|)|N|^2)$. Making the total overall message complexity $O(|N|^2 + |E|)$ and the total overall bit complexity becomes $O(\lg(|N|)|N|^2)$. For the non-incremental mode the message complexity of the reply messages improves from $O(|N|^2)$ to $O(|E|)$, since there is only one reply per query in this mode. The other complexities are the same as in the incremental mode.

The algorithm of Francois and Bonaventure has a message complexity of $O(|E|)$, which is equal to the non-incremental mode of the LFLSU-algorithm and better than the incremental mode of the LFLSU-algorithm. The bit complexity of the algorithm by Francois and Bonaventure is $O(\lg(|N|)|E|)$ which is better than both modes of the LFLSU-algorithm.

Finally, both the LFLSU-algorithm and the algorithm by Francois and Bonaventure share the same asymptotic upper bound on the local running time. The asymptotic running time for both algorithms comes from needing to compute a new shortest path tree.

V. ALGORITHM COMPARISON

The LFLSU-algorithm is more robust against loss of data than previous algorithms. The algorithm by Francois and Bonaventure can cause unwanted loss of data in the case where only some (but not all) data flows are protected against node failures by use of fail-over paths. Such network configurations/setup are possible due to economical restrictions.

Fig. 2. Example of where a protected data flow can experience losses

Maintaining fail-over paths implies that there has to be kept redundant bandwidth within the network, which would not normally be used and also therefore not normally be an income of revenue. If not all data flows are protected by fail-over paths, the algorithm by Francois and Bonaventure may in fact lead to loss of data even in the case of data flows that are originally protected by fail-over paths. The problem arises since the algorithm by Francois and Bonaventure only ensures that the update order is enforced within a subtree of the implied sink tree. Since there is no synchronization of the update order between different subtrees of the implied sink tree, it is possible for protected data flows to be diverted to a node b in another subtree which has not yet updated its routing table. In this case the node b may actually forward the data flow towards the failed node and if there does not exist a fail-over path at this point (e.g., due to economical restrictions), the data flow will experience losses. Figure 2 shows an example of where a protected data flow could experience losses in such a case. The figure shows the implied sink tree of node X which is assumed to have failed. Traffic flowing from subtree T_a to subtree T_d is assumed to be important and therefore protected by a fail-over path, shown as a thick dashed line. In the example node a updates its routing table before node b . After node a has updated its routing table it starts to forward data flows destined to node d via node b . The thin dashed lines indicate the new shortest path from node a to node d after node X has failed. However, since node b has not yet updated its routing table, the data flow is not passed on to node d but instead forwarded via node b_r to the failed node X . And since there is no fail-over path at that point, the data flow will experience losses. The LFLSU-algorithm does not suffer from this problem since it does not constrain the synchronization of the update order. Therefore whenever an update occurs, a protected data flow is guaranteed to reach its destination.

VI. CONCLUSIONS

As today's intra-domain networks are increasingly relied on for services that have traditionally been associated with the "telecom world", e.g., telephony, the demands on the robustness of computer networks is increasing. Given that telephony customers are used to services with high availability and networks that are capable of repairing errors within some hundreds of milliseconds, it is obvious that the task facing intra-domain computer networks is demanding. However, by

using fail-over routing it is possible for today's computer networks to achieve very rapid responses to networking events, e.g., link failures. Unfortunately, commonly used intra-domain routing protocols, e.g., OSPF and IS-IS, allow temporary loops to form during convergence. Such temporary loops can negate the benefit of fail-over routing, since traffic may be prevented from reaching the fail-over path. In a step to increase the robustness of link-state intra-domain networks, we present an algorithm that prevents temporary loops from forming during convergence. The algorithm is proved to produce correct results, be loop-free at every instant and to be free from dead-lock. The algorithm improves on previous loop-free algorithms for link-state routing by reducing the number of state transitions necessary before a router can update its routing table to a new, guaranteed to exist, path for a wide variety of topological configurations. The algorithm is also more robust and flexible than previous algorithms.

REFERENCES

- [1] Moy, J.T.: OSPF version 2. Request for Comments 2328, IETF (1998)
- [2] Oran, D.: OSI IS-IS Intra-domain Routing Protocol. Request for comments 1142, Internet Engineering Task Force (1990)
- [3] Moy, J.T.: OSPF Anatomy of an Internet Routing Protocol. Addison-Wesley (1998) ISBN 0-201-63472-4.
- [4] Wang, Z., Crowcroft, J.: Shortest Path First with Emergency Exits. In: Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols, Philadelphia, PA (1990) 166–176
- [5] Garcia-Luna-Aceves, J.: A Unified Approach to Loop-Free Routing Using Distance Vectors or Link States. In: SIGCOMM Computer Communications Review, Vol. 19, No. 4. (1989) 212–223
- [6] François, P., Bonaventure, O.: Avoiding Transient Loops During IGP Convergence in IP Networks. In: Proc. IEEE INFOCOM 2005., Miami (2005)
- [7] Merlin, P., Segall, A.: A Failsafe Distributed Routing Protocol. In: IEEE Transactions on Communications. Volume COM-27. (1979) 1280–1288
- [8] Jaffe, J., Moss, F.: A Responsive Routing Algorithm for Computer Networks. In: IEEE Transactions on Communications. Volume COM-30. (1982) 1758–1762
- [9] Garcia-Luna-Aceves, J.: A Distributed Loop-Free Shortest-Path Routing Algorithm. In: Proceedings of IEEE INFOCOM'88. (1988)
- [10] Chandy, K., Misra, J.: Distributed Computation on Graphs: Shortest path Algorithms. In: Communications of the ACM. Volume 25. (1982) 833–837
- [11] Murthy, S., Garcia-Luna-Aceves, J.: A loop-free algorithm based on predecessor information. In: Proc. IEEE International Conference on Computer Communications and Networks (ICCCN), San Francisco, California (1994)
- [12] Kurose, J.F., Keith, W.R.: Computer Networking: A Top-Down Approach Featuring the Internet, 3rd ed. 3rd edn. Pearson Education Inc. (2004) ISBN 0-321-26976-4.
- [13] Malkin, G.: RIP Version 2. Request for comments 2453, Internet Engineering Task Force (1998)
- [14] Garcia Luna Aceves, J.: Loop-Free Routing Using Diffusing Computations. In: IEEE/ACM Transactions on Networking, Vol. 1, No. 1. (1993)
- [15] Dijkstra, E.: A Note on Two Problems in Connection with Graphs. *Numerische Mathematik* **1** (1959) 261–271
- [16] Tel, G.: Introduction to Distributed Algorithms. 2nd edn. Cambridge University Press, Cambridge, CB2 2RU, UK (2000) ISBN 0-521-79483-8.
- [17] Zinin, A.: Analysis and minimization of microloops in link-state routing protocols. Internet Draft, Internet Engineering Task Force (2005) Work in Progress.
- [18] Katz, D., Ward, D.: BFD for IPv4 and IPv6 (Single Hop). Internet Draft, Internet Engineering Task Force (2005) Active Internet Draft.
- [19] Postel, J.: Transmission control protocol. Request for comments 793, Internet Engineering Task Force (1981)