Fast Dissemination of Link States Using Bounded Sequence Numbers with no Periodic Updates or Age Fields

Jochen Behrens J.J. Garcia-Luna-Aceves

Baskin Center for
Computer Science and Computer Engineering
University of California
Santa Cruz, California 95064
jochen, jj@cse.ucsc.edu

Abstract

Routing protocols based on the distribution of link-state information rely on sequence numbers to validate information that a router receives. A fundamental problem is to bound the sequence-number space. We propose a new sequence-number reset algorithm that needs neither periodic retransmissions nor age fields. It is based on a recursive query-response procedure and is designed to handle resource failures during operation. This new algorithm is applicable to routing protocols based on both flooding and selective distribution of link-state information. The correctness of the algorithm is verified in the context of selective dissemination of topology information, and its complexity analyzed. Because the reset algorithm does not use any aging, the distribution of new link-state information or the purging of old information is always done in a time proportional to the time it takes to traverse the network.

1 Introduction

Disseminating link-state (topology) information reliably is essential to many internet routing protocols proposed or implemented to date. This dissemination can take the form of broadcast, in which every network node (router) maintains the same topology map [3], or selective distribution, in which each node maintains only the subset of the topology map it needs to perform correct routing [7]. In a very large internet, network resources must be aggregated into clusters or areas to reduce the amount of information each node needs to store and process; however, because the focus of this paper is on the basic algorithm used for disseminating topology information in a network a flat network organization is assumed.

Broadcast of link states can be accomplished by flooding or building a spanning tree over which link states are distributed [8]. This paper focuses on flooding because of its simplicity and popularity. Examples of standard internet routing protocols based on the flooding of link states are OSPF [10], IS-IS [9] and NLSP [11]. In addition, the interdomain policy routing (IDPR) architecture [6] and the Nimrod architecture for scalable internet routing [4] are both based on flooding. These protocols and architectures use the same basic approach for the flooding of topology information, which we simply call *intelligent flooding protocol* or IFP in the rest of this paper (e.g., see [12, 13]).

According to IFP, each network router ascertains the state of its outgoing links and reports this in what we will call a *link-state update* (LSU); for simplicity, we assume that an LSU reports the state of only one outgoing link adjacent to a router, which we call the source of the LSU. The basic problem then becomes one of broadcasting the most recent LSUs of each source to every router in the network. Once this is accomplished, each router has a topology map from which it can compute the desired paths to destinations. To flood LSUs, IFP uses sequence numbers to validate the most recent LSU; a router accepts a new LSU only if it has a higher sequence number than the one stored.

Because the sequence-number space available in a routing protocol is finite, IFP must operate with finite sequence numbers. To accomplish this, a linear sequence-number space is used together with an age field, and large enough that the maximum sequence number should be reached only in very rare circumstances. Each LSU specifies a sequence number and an age. The source sends a new LSU with a higher sequence number after either detecting a change in the state of an adjacent link, or after reaching a maximum time with no state changes in adjacent links. Each LSU sent by the source specifies the current sequence number and the maximum age for that LSU (in the order of an hour in today's protocols). No more LSUs from the same source are accepted when the sequence number reaches its maximum

This work was supported in part by the Advanced Research Projects Agency (ARPA) under contract F19628-96-C-0038

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1997		**00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Fast Dissemination of Link States Using Bounded Sequence Numbers with no Periodic Updates or Age Fields	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**8**	

value, until the LSU is erased due to aging. Aging means that every router that accepts an LSU decrements its age by at least one and also decrements the age while the LSU sits in memory. It must be ensured that nodes age LSUs at a similar pace, and LSUs must be sent reliably between neighbors.

The *link-vector algorithm* (LVA) introduced in [7] is based on the selective distribution of topology information, rather than on flooding. The purpose of this algorithm is to allow a router to maintain only the link-state information it needs to reach a destination, rather than the entire topology map. Each router maintains a subset of the topology map corresponding to its adjacent links and the links that its neighbor routers have reported as being used in their paths to destinations. The router uses this information to compute its own paths to destinations, and reports to its neighbors the states of only those links used in the chosen paths. In addition, the router tells its neighbors which links it no longer uses to reach destinations. A basic assumption for the correct operation of the LVA implementation introduced in [7] is that routers can determine whether an update contains up-to-date information using the same update validation scheme used for LSUs in IFP.

The inherent limitation with the above method of validating LSUs is that the age field must be very long to avoid situations in which, due to aging, routers loose LSUs that are still valid. Furthermore, because every LSU must expire in a finite time, the source of each LSU must retransmit new incarnations periodically in the absence of link-state changes. In practice, aging of sequence numbers introduces additional communication overhead. Furthermore, after resource failures that isolate any portion of the network from a source of LSUs, old LSU information can be erased only after reaching its maximum age. We propose a new algorithm that achieves fast dissemination of up-to-date link-state information without periodic updates or age fields. This algorithm is based on a finite and linear sequence-number space and *diffusing computations* [5].

Our algorithm can be applied to standard internet routing protocols based on flooding, eliminates the need for periodic flooding of LSUs, and can dramatically reduce the amount of time in which obsolete LSUs can be erased after resource failures or new LSUs can be copied throughout the network after resource recoveries. The latency of our algorithm is bounded only by the time it takes for an LSU to traverse the network, rather than by a global timer, which is the case in all previous reset schemes used or proposed to date (e.g., see [13, 1]).

The following section states the goals for our reset mechanism. Section 3 describes the new reset algorithm in the context of flooding as well as selective diffusing of topology information. Section 4 verifies that the new reset algorithm works correctly within the context of selective dissemination of topology information. We chose to address correctness in the context of selective dissemination because, as we will show, selective dissemination represents a generalization of flooding. Section 5 analyses the complexity of the selective dissemination algorithm. Section 6 summarizes the applicability of our results.

2 Objectives of The Reset Algorithm

The objective of the reset algorithm is threefold:

- When a source of LSUs must wrap around the sequence number it uses, all the routers affected by the LSU are forced to synchronize with the source in such a way that all other sequence numbers from the same source are purged and all routers affected by the LSU reset its sequence number.
- After a malfunction routers can force either the source or another router to provide a correct sequence number for any given link.
- After a resource failure, routers with no physical path to a source erase the LSU from that source within a finite time proportional to the time it takes to traverse their connected components.

A reset algorithm for IFP with goals similar to the above three has been sketched in [1]. According to this algorithm, whenever the sequence number of an LSU reaches its upper bound at some router, this router makes a reset request. When a request reaches a router other than the source, that router resets its sequence number to 0 and forwards the request. When the source of the LSU receives the request, it sets its sequence number to 1 and broadcasts its most recent LSU. This type of reset has two problems: erroneous LSU information has to propagate all the way to the source before it can be erased [3]. Other than having a global timer, there is no provision for erasing an obsolete LSU after the failure of the source of the LSU or the partition of the network. The following section outlines how our reset algorithm supports the three goals stated above.

3 Description of Reset Algorithm

The reset algorithm can be applied to both the replication of the same LSU at every router, or the selective dissemination of LSUs. Replicating LSUs at every router is a special case of selective dissemination of LSUs, and there are only two important simplifications: The first is that, because every router must receive every LSU, there is no need for a router to request its neighbors to delete any LSU. The other simplification is that a router does not have to decide which LSU to propagate depending on the link constituency of its paths to destinations; the router simply propagates each valid LSU. A network is modeled as an undirected graph

$G = (V, E)$, where V is the set of nodes (routers) and E is the set of edges (links). Links are bidirectional with a positive cost assigned in each direction. An underlying protocol assures that every node detects changes in link states within a finite amount of time. All changes are processed one at a time in the order in which they are detected.

Assume that a protocol is used for the dissemination of link-state information and the maintenance of topology and routing tables. This protocol, be it based on flooding or selective dissemination of link states, must use certain message formats to exchange link states among adjacent routers (LSUs). We assume that an LSU specifies who originates it, a sequence number, the state of the link, and an add or delete instruction in the case of selective dissemination. Sequence numbers are assumed to be drawn from a finite and linear sequence-number space. In the same way that some routing protocols based on topology broadcast do (e.g., OSPF), we assume that LSUs are exchanged reliably between neighbors. When a router sends an LSU in a message, it waits for acknowledgments from all its neighbors, and retransmits the message with the LSU to a neighbor if it does not receive an ack after a timeout. Connectivity with a neighbor is assumed lost after a number of unsuccessful message transmissions.

In normal operation nodes execute the flooding or selective dissemination protocol by exchanging LSUs as summarized in Section 1 for IFP and LVA. In this case, a node that originates an LSU and sends it to its neighbors only needs to receive acknowledgments from them stating that they have received the LSU.

There are three cases in which a node must ensure that all the nodes that need to know about the state of a given link receive the new information for the link and adapt the correct sequence number. These cases are the following:

- The node needs to reset the sequence number for one of its outgoing links.
- The node detects the failure of one of its adjacent links.
- The node detects that it has no physical path to the head of a remote link (i.e., the source of LSUs concerning that link).

This is accomplished by means of two additional types of update message entries: *queries* and *replies*. Both are reliably transmitted between neighbors by means of message acknowledgment and retransmissions, as are the LSUs. Queries have the same fields as an LSU. Replies do not need to transmit link information, but may carry a tag signaling the possibility of an error. Based on these queries and replies, our reset algorithm operates in a manner very similar to Dijkstra and Scholten's algorithm [5].

A node that needs to reliably distribute information about a link through the network and detect the termination of this, sends queries instead of LSUs to all its neighbors and then waits until it receives a reply from each neighbor. A reply signals that a neighbor and all nodes connected through that neighbor that need to process the query have done so. A node is said to be in *active* mode (or state) when it is waiting for replies; otherwise, it is *passive*. A passive node receiving a query for a given link follows the same pattern, it forwards the query to all its neighbors, waits for their replies, and, upon reception of the last reply, sends a reply to its predecessor in the diffusing computation, i.e., to the node from which it received the query that caused its transition to active state. If an active node receives another query, then it simply sends a reply back to the neighbor that sent the query.

An update message may contain queries and replies, as well as plain LSUs from the underlying routing protocol. When an update message is received, the node first processes all the replies, then the LSUs, and at last the queries that are included. The replies must be processed first so that updates can be buffered if the respective reply is in the same packet.

3.1 Reset for Flooding

In the case of flooding, the complete topology information needs to be replicated at every node. Figures 1 and 2 give a formal specification of the reset mechanism for flooding.

A passive node processes LSUs according to the rules for intelligent flooding. If an active node receives an LSU, it must check whether it already received a reply from the sender of the LSU. If this is the case, then the update must be buffered because it contains more recent information than the query did. There must be a separate buffer for each neighbor, but only the latest LSU must be kept. In addition, the buffer is flushed when a query is received subsequently over the same link.

When a passive node receives a reply, this reply is simply discarded. An active node receiving a reply checks if this is the last reply that it expects; if this is the case, the node goes into passive state and sends a reply to its predecessor

TT_x Topology table at node x, entries (i, j, l^i_j, sn, r, d), where
- (i, j) Link from node i to node j
- l^i_j Length of link (i, j)
- sn Sequence number of link
- d Status for diffusing computations: active/passive, set of replies received, source, predecessor

ST_x Shortest path tree at node x.
N_x Set of neighbors at node x.
t Sequence number at node.
t_j Last sequence number of neighbor j.

Messages are (ordered) sets of updates of the form
$(i, j, l^i_j, sn, type, ds)$ for link (i, j) with cost l^i_j, where
- sn Sequence number
- $type$ Type of update: update, query, reply
- ds Source of diffusing computation (if applicable)

Figure 1: Notation for Pseudo-Code

```
procedure process_packet (x, n, packet)
  begin
    process_replies (x, n, packet)
    process_updates (x, n, packet)
    for all q ∈ packet – – query q
      process_query (x, q)
    end for
    assemble_and_send_new_packets (x)
  end process_packet

procedure process_updates (x, message)
  begin
    for all m = (i, j, l^i_j, sn, update) do
      if TT_x(i, j).d = active then
        if reply from message.source received then
          buffer m
        else
          discard m
        end if
      else – – passive
        if (i, j) ∈ TT_x then
          if TT_x(i, j).sn < m.sn then
            if m.l^i_j < ∞ then
              TT_x(i, j) = m
            else
              TT_x(i, j) = ∅
            end if
          else if TT_x(i, j).sn > m.sn then
            send (message.source, (TT_x(i, j), update) )
          end if
        end if
      end if
    end for
  end process_updates

procedure set_entry (TT_x, i, j, status, source, l, sn)
  begin
    TT_x(i, j).d = status
    TT_x(i, j).d.source = source
    TT_x(i, j).l^i_j = l
    TT_x(i, j).sn = sn
  end set_entry

procedure link_change
  begin
    if TT_x(x, y).d = active then
      buffer update {(x, y, l^x_y, sn, update)}
    else
      process_updates (x, {(x, y, l^x_y, sn, update)})
    end if
    assemble_and_send_new_packets (x)
    sn = sn + 1
  end link_up

procedure process_query (x, q)
– – query q = (i, j, l^i_j, sn, type, ds)
  begin
    if TT_x(i, j).d = passive then
      if x = i then
        send (n, (i, j, reply))
        send (n, (TT_x(i, j), update)
      else if q.ds = i or i ∉ ST_x then
        set_entry (TT_x, i, j, active, i, q.l^i_j, 0)
        TT_x(i, j).d.predecessor = n
        for all k ∈ N_x do
          send (k, New_query)
        end for
      else
        send (n, (i, j, reply))
      end if
    else
      if q.ds = i then
        if TT_x(i, j).d.source = i and TT_x(i, j).l^i_j = q.l^i_j then
          send (n, (i, j, reply))
        else if TT_x(i, j).d.source ≠ i then
          set_entry (TT_x, i, j, active, i, q.l^i_j, q.sn)
          TT_x(i, j).d.predecessor = n
          for all k ∈ N_x do
            send (k, New_query)
          end for
          updated = true
        else
          if x = i then
            send (n, (i, j, reply))
            send (n, (i, j, l^i_j, sn, update))
          else
            TT_x(i, j).d = active, tagged
            send (n, (i, j, reply))
          end if
        end if
      else
        if TT_x(i, j).d.source = i then
          discard query
          send (n, (i, j, reply))
        else
          process update part
          send (n, (i, j, reply))
        end if
      end if
    end if
  end process_query

procedure link_failure (x, y)
  begin
    N_x = N_x – {y}
    message = {(x, y, ∞, t, update)}
    process_updates (x, message)
    assemble_and_send_new_packets (x)
    sn = sn + 1
  end link_failure

procedure process_replies (x, n, packet)
  begin
    for all rep ∈ packet do – – rep = (i, j, t)
      if TT_x(i, j).d = active then
        TT_x(i, j).d.received = TT_x(i, j).d.received ∪ n
        if t = true then
          TT_x(i, j).d.tag = true
        end if
        if TT_x(i, j).d.received = N_x then – – all replies received
          if x = i then
            if TT_x(i, j).d.tag = true then
              for all k ∈ N_x do
                send (k, (TT_x(i, j), update)
              end for
            end if
          else
            new_reply = (i, j, TT_x(i, j).d.tag)
            send (TT_x(i, j).d.predecessor, new_reply)
          end if
          TT_x(i, j).d = passive
          if there are buffered updates for (i, j) then
            for all m = (i, j, l^i_j, sn, update) in buffer do
              if m.sn > TT_x(i, j).sn then
                TT_x(i, j) = m
                for all k ∈ N_x do
                  send (k, m)
                end for
              end if
            end for
          end if
        end if
      else
        discard rep
      end if
    end for
  end process_replies

procedure assemble_and_send_new_packets (x)
  begin
    for all (i, j) ∈ TT_x do
      if TT_x(i, j).d = passive and i unreachable then
      – – i unreachable is the same as i ∉ ST_x
        TT_x(i, j).d = active
        TT_x(i, j).d.source = x
        u_message = u_message ∪ (TT_x(i, j), query, x)
      end if
    end for
    for all k ∈ N_x do
      message = u_message ∪ buffered information for k
      send (k, message)
    end for
  end assemble_and_send_new_packets

procedure link_up (x)
  begin
    N_x = N_x ∪ {y}
    process_updates (x, {(x, y, l^x_y, sn, update)})
    assemble_and_send_new_packets (x)
    sn = sn + 1
  end link_up
```

Figure 2: Specification of Reset Algorithm for Flooding

in the diffusing computation (unless it is the source of the diffusing computation, in which case the diffusing computation is terminated). It then processes buffered LSUs. In the case that the state of the link changed since the node became active (i.e., the buffered LSUs contained more recent information), LSUs are sent to all neighbors. If the reply carried an error tag, all subsequent replies for this diffusing computation that the node sends also carry such a tag.

The core of the algorithm is the way in which queries are handled. When the node that receives a query is in passive state, it generally accepts the query, goes into active state, and sends queries to its neighbors. However, there are two exceptions to this rule. First, if the source of the diffusing computation is not the head of the link and the receiving node has a path to the link reported in the query, the node simply sends a reply. This prevents a diffusing computation originated by a node other than the head of the link from propagating to parts of the network where a physical path to the head of the link is known. Second, if the head of the link reported in the query receives it, the node sends a reply. If the content of the query it receives is different from the current link information, the head of the link also sends an LSU with a higher sequence number; this ensures that the correct information about the link will be known throughout the network.

Figure 3 illustrates the normal action for a diffusing computation concerning link (i, j). First, i sends queries to all its neighbors (Fig. 3(a)). These nodes go into active state and forward the query to its neighbors (Fig. 3(b)). Since neighbor w also received a query from i, it is active and immediately sends a reply to x after receiving the query from this node (and vice versa). y and z also forward the query to their neighbors (Fig. 3(c)). After nodes y and z also receive replies from each other (Fig. 3(d)), they return to passive state and send replies to their predecessors in the diffusing computation, x and w, respectively. After x re-

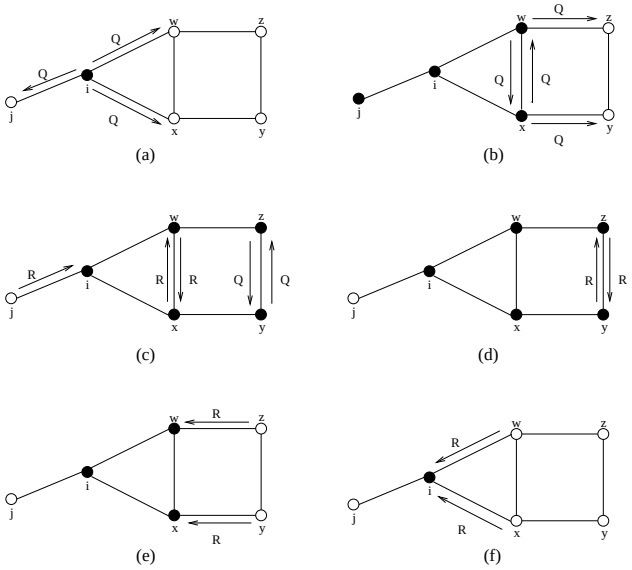

Figure 3: Normal action of reset algorithm. Filled circles denote active nodes

ceives the reply from y (Fig. 3(e)), it returns into passive state and sends a reply to its predecessor in the diffusing computation, node i, as does w after receiving the reply from z (Fig. 3(f)). When node i receives the last reply, it also returns into passive state and the diffusing computation terminates.

In active state, the normal action taken by a node after receiving a query is to send a reply. However, if the node is active in a diffusing computation that was started at a node other than the head of the link, but the origin of the new query received is the head of the link reported in the query, then the new computation takes over. That is, the node becomes active in the computation started by the head of the link, and sends out new queries to all its neighbors. These queries ensure that the new diffusing computation also takes over at all other nodes that were active in the old diffusing computation.

The other exception to the normal processing of queries occurs when a node detects an erroneous situation when it is active in a diffusing computation originated by the head of a link and receives a second query originated by the same node for the same link, but such that the query contains different link-state information. This situation can only occur after a component of the network, in which an old diffusing computation has not terminated, is reconnected to another component. This situation is illustrated in Figure 4, where node z is shown to receive two different queries. Because node z cannot decide which of the two diffusing computations is more recent, the situation must be corrected by the head of the link. Therefore, the node sends a reply that

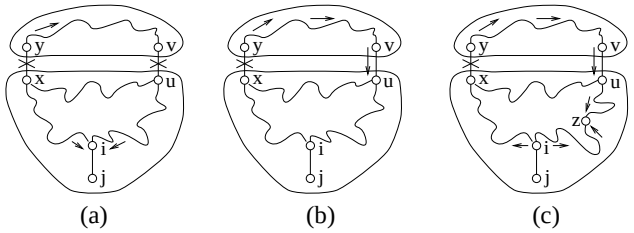

Figure 4: Detection of erroneous condition at node z

has an error tag, and tags its active state, meaning that all subsequent replies sent for the computation will be tagged as well. The propagation of the error tags ensures that the head of the link will be notified of the erroneous situation, unless there is no physical path to the head of the link; in that case the diffusing computations both terminate and the information about the link will be erased in this part of the network.

When a link fails, the head of that link updates its topology table and sends a query for this link to its neighbors. If the node is active in a diffusing computation concerning another link and is still waiting for a reply to come over the failed link, then the node assumes that the reply has been received, and that this reply was tagged; this helps prevent deadlocks.

When the cost of a link changes or a new link is established, the head of that link initiates the flooding of an LSU for that link if the node is passive. However, if the head of the link is already active for the link when a change of cost or reestablishment of the link is detected, then the link must wait to distribute the LSU upon termination of the diffusing computation.

As described above, tags in replies are needed to signal an erroneous situation to the head of a link, who then sends LSUs with higher sequence number to its neighbors. Figure 5 (a) illustrates the propagation of tagged replies back to node i, the source of the diffusing computation. This example assumes that x received the query from w earlier than the query from i and that y and z are waiting for replies from each other when link (y, z) fails. After receiving the tagged reply from w and the reply from j, the head of the link (node i) sends LSUs with a new sequence number to

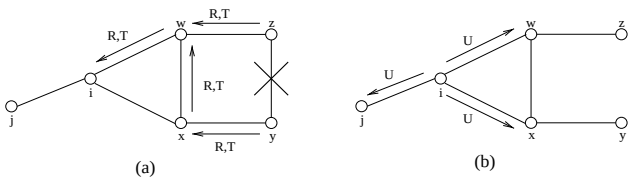

Figure 5: Propagation of tagged replies for diffusing computation to reset sequence number of (i, j) after link failure

its neighbors (Figure 5 (b)).

Unfortunately, the tagging mechanism may require extra communication in cases where it is not needed. In particular, whenever a link with outstanding replies fails, the reply is assumed to be tagged, causing an unnecessary new LSU to be generated by the source (as in Figure 5). It is, however, possible to use some other means to signal the erroneous situation to the head of the link. For example, a different diffusing computation could be used to make sure that the information gets to the head of the link. This would increase the worst case complexity, but reduce the complexity in the more likely case of a link failure. Moreover, the correctness of the basic algorithm would not be affected if the new algorithm assures delivery of the needed information to the head of the link.

3.2 Reset for Selective Dissemination

To use the reset algorithm described in the previous section with LVA, some minor modifications need to be made. Since in LVA not all nodes need to store information about a given link, nodes that do not have information about a link (i.e. the link is neither in the topology table nor in the list of deleted links) need not participate in a diffusing computation concerning that link. Therefore, a node without information about the link (obviously, such a node is passive) simply sends a reply to the sender of the query, if the link-state information in the query does not cause the node to store the link.

In addition to sending a reply, an active node that receives a query needs to update the list of reporting nodes kept in LVA. A node that receives the last reply for a query must check for changes in the state of the link since it became active. With LVA, such changes can be caused by the buffered LSUs as well as by other information acquired during the active period; a change of the state of the link here includes more recent information as well as a switch from used to not used or vice versa. If any such change occurred, the appropriate *add* or *delete* update must be sent.

3.3 Fast Deletion of Old Information

An important feature of our reset algorithm is the fast deletion of old information. This is important, because reconnecting previously disconnected parts of the network can lead to significant overhead. For example, assume that some part of a network is disconnected. With aging, it takes a long time for the information about links in the other network component to be flushed. Therefore, if the sequence number of a link has been reset using premature aging and the network is reconstituted, it is possible for older information with a higher sequence number to pollute the network.

Figure 6 illustrates the above problem. In Figure 6 (a), an example topology is shown where a sequence number of 20 is known for link (i, j) throughout the network when link (x, y) fails. Figure 6 (b) shows the situation after node i initiated a diffusing computation to reset the sequence number of (i, j) to 0. In Figure 6 (c), the link between nodes x and y is reestablished before the old information with sequence number 20 expired in the disconnected component. As seen in Figure 6 (d), the obsolete information can now be propagated in the other part of the network as well. Although this situation will eventually be noted and corrected by the head of the link (node i), it may result in temporary routing loops.

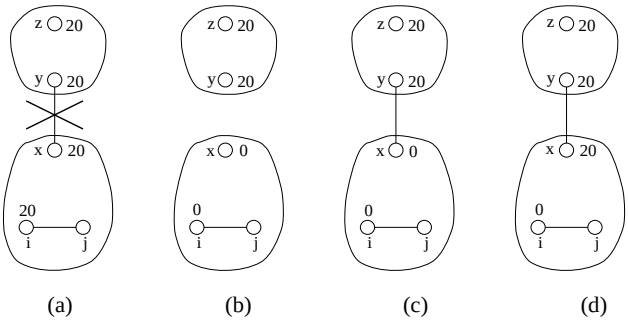

Figure 6: Example of polluting a network with old information.

On the other hand, if our reset mechanism is used rather than aging, the obsolete information in the component that was disconnected from the head of the link will very likely be erased by the time network reconststution occurs, because such erasure will occur in a matter of a few minutes. This also reduces the probability of temporary loops.

4 Correctness of Reset Algorithm

In this section we present an outline for the proof of correctness of the reset scheme for the case of selective dissemination of link-state information, which is a generalization of flooding. A subset of the same proof applies to the broadcast of link states.

Message transmissions over an operational link are made reliable (i.e., messages are received without error and in the order in which they are sent) by means of a correct retransmission strategy between any two nodes across a link. With this assumption, the proof of correctness can simply assume, without loss of generality, that LSUs, queries, and replies are always sent reliably over an operational link. We also assume that the routers perform LVA error free. The reset algorithm is correct if, after a finite sequence of topology changes, any diffusing computation started at some node for a given link, terminates within a finite amount of time, and upon termination, the network has *consistent information* about the link.

Consistent information here has the following meaning:

- If the diffusing computation was initiated by the head of the link, then all nodes that have any notion of

the link have the same information about it, and only nodes that use the link or whose neighbor uses the link have a notion of the link.

- If the diffusing computation was initiated by a node other than the head of the link, then the information about this link has been erased from those nodes that cannot reach the link.

The first step in proving correctness is to show that the reset algorithm terminates. Under the conditions stated above, we have the following:

Theorem 1 *Any diffusing computation for a given link (i, j) terminates within a finite amount of time.*

Proof: There are two possible scenarios for a diffusing computation not to terminate:

1. Deadlock could occur.
2. An infinite amount of queries could be generated.

To show termination of the algorithm, we need to show that neither of these scenarios can occur.

The proof that there can be no deadlock is by contradiction. Consider first a network with a static topology. Assume that there is some node x at which deadlock occurs. This implies that x does not receive a reply from at least one of its neighbors, say y. Node y must have some notion of (i, j) and y cannot be in active state when it receives the query from x, otherwise y would have sent a reply immediately after it received and processed the query from x. Hence, y becomes active with x's query and it must wait for a reply from a neighbor other than x, because x must send a reply to y. Following this line of argument, there must be an infinite number of nodes waiting for replies from nodes other than the node from which the query was received. This is not possible because the network is finite. Therefore, x cannot be in a deadlock situation.

Note that deadlock cannot occur even in a dynamic topology. This is the case because, when a link adjacent to x fails (or is established), then x simply assumes that a reply has been received over that link.

The proof that only a finite number of queries can be generated is also by contradiction. Assume that the diffusing computation does not terminate. Since there are only finitely many nodes, there must be a node x that produces an infinite number of queries. However, x cannot be the head of the link, which produces exactly one query. Therefore, node x must receive a query, send its own queries to all its neighbors, and send a reply infinitely often. Hence, at least one of its neighbors must do the same. Furthermore, w.l.o.g., this node must receive its first query earlier than x. In other words, there must be either a cycle of nodes which alternately go into active and passive state, or an inductive argument shows that there must be an infinite number of nodes in the stated situation. The first case can only occur if two parts of the network become disconnected and reconnect after the diffusing computation has terminated in one component but not in the other. Since there is only a finite sequence of changes in the network, this cannot go on indefinitely. Obviously, the second possibility contradicts the assumption of a finite network.

This concludes the proof of Theorem 1. **q.e.d.**

The algorithm works correctly if, after termination in a connected component, all the nodes in that component that have any information about a given link have the same sequence-number (and the same link information) for that link. In addition, the information must be consistent as required by the underlying routing protocol, i.e., it must be up to date and, for LVA, correctly reflect whether it is used by the neighbor nodes.

This means that, in any part of the network that has no connection to a given link, the information about that link must be completely removed. In the part of the network that uses the link, the nodes that do have information about the link (this set is determined by the basic algorithm) have the latest sequence number for the link reported by the head of the link and the other nodes have erased any information about the link from their topology table.

With the assumptions stated above, the following theorem applies.

Theorem 2 *Upon termination of a diffusing computation for a given link (i, j), the information about the link is consistent throughout the network, i.e., any node in the network has the correct information.*

The proof of this theorem is based on a series of lemmas [2] showing that

- Any node with information about the link will receive a query.
- After termination, all nodes have consistent information about (i, j).
- All information about (i, j) will be removed in a disconnected component.
- A diffusing computation started in a seemingly disconnected part of the network does not produce inconsistent link information.
- No permanent inconsistency is caused by the temporary disconnection and reconnection of the network while a diffusing computation is going on.

5 Complexity

5.1 Communication Complexity

For a single diffusing computation, the number of messages generated is $O(|E|)$. In the worst case, two queries are sent over each link, one in each direction. Note that there is exactly one reply sent for each query, which does not change the order of magnitude for the communication complexity.

The source of the diffusing computation obviously sends exactly one query over each outgoing link, because it must receive replies from all its neighbors before it can send a second query, which is the condition for the computation to terminate. Now consider an arbitrary node other than the source of the diffusing computation. When this node receives a query, it either sends a reply to the sender, or it sends exactly one query over each outgoing link. Before it can send more queries, it must first receive replies from all its neighbors and then receive a new query. Hence, it must become active more than once for the same diffusing computation. This is not possible in the connected part of a network.

If IFP is used as the underlying protocol, the number of queries can easily be restricted to one per link by not sending a query to the predecessor in the diffusing computation. With LVA, this extra query is used to update the set of reporting nodes at the predecessor node. On the other hand, with IFP, the worst case always occurs because the whole network is flooded with the information, while LVA produces fewer messages in the average case [7].

5.2 Time and Storage Complexity

In the connected component of the network, the worst-case time complexity is $O(x)$, where x is the number of affected routers. With IFP, x obviously is the number of nodes in the network, while with LVA it can be significantly less. The queries will travel to all nodes that do require the information, before the replies are sent back on the same paths and with the same time complexity. In the worst case, all affected routers lie along a single path, causing $O(x)$ communication steps.

Computational complexity at routers is determined by complexity of underlying protocol, for each query, only constant work is added.

The extra storage required while a node is passive is constant, only an extra tag indicating the state is needed. While a node is active, in addition to some extra state information, $O(|N_x|)$ storage (where $|N_x|$ is the set of neighbors of node x) is required to keep track of the received replies at node x, and to buffer LSUs.

6 Conclusions

We presented a new algorithm to reset sequence numbers in routing protocols based on link state information. This reset algorithm, which is based on a recursive query-response process, makes it possible to use a bounded sequence-number space without a need for periodic retransmissions or aging. Thus, its time complexity is determined entirely by the time it takes to traverse the network, and it does not rely on any global timers.

The reset algorithm can be used with routing protocols based on flooding as well as selective dissemination of link-state information to speed up their convergence. For instance, using our reset algorithm in OSPF, even when resources fail, all link-state information will be distributed in time proportional to the time needed to traverse the network, which should take in the order of minutes at the most. A version of intelligent flooding based on this reset mechanism was introduced.

We have shown that the reset algorithm leads to a correct routing protocol when applied to the selective dissemination of link-state information, which is a generalization of flooding.

References

[1] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded. In *Proceedings of IEEE INFOCOM '94 Conference*, volume 2, pages 776–783, Toronto, Ont., Canada, 12-16 June 1994.

[2] J. Behrens. *Distributed Routing for Very Large Networks based on Link Vectors*. Ph.D. thesis, University of California, Santa Cruz, May 1997.

[3] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1992.

[4] I. Castineyra, J. N. Chiappa, and M. Steenstrup. The Nimrod Routing Architecture. Internet Draft, January 1995.

[5] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[6] D. Estrin, M. Steenstrup, and G. Tsudik. A protocol for route establishment and packet forwarding across multidomain internets. *IEEE/ACM Transactions on Networking*, 1(1):56–70, February 1993.

[7] J.J. Garcia-Luna-Aceves and J. Behrens. Distributed, scalable routing based on vectors of link states. *IEEE Journal on Selected Areas in Communications*, 13(8):1383–95, October 1995.

[8] P.A. Humblet and S.R. Soloway. Topology broadcast algorithms. *Computer Networks and ISDN Systems*, 16(3):179–186, January 1989.

[9] International Standards Organization. *Intra-Domain IS-IS Routing Protocol*, ISO/IEC JTC1/SC6 WG2 N323, September 1989.

[10] J. Moy. OSPF Version 2. RFC 1583, Network Working Group, March 1994.

[11] Novell. *NetWare Link Services Protocol (NLSP) Specification, Revision 1.0*. Novell, Inc., San Jose, CA 95131, 2nd edition, February 1994.

[12] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7(6):395–405, 1983.

[13] R. Perlman, G. Varghese, and A. Lauck. Reliable Broadcast of Information in a Wide Area Network. US Patent 5,085,428, February 1992.