Scalable Link-State Internet Routing[*]

J. J. Garcia-Luna-Aceves and Marcelo Spohn
Computer Engineering Department
School of Engineering
University of California
Santa Cruz, CA 95064, USA
jj, spohn@cse.ucsc.edu

Abstract

We present and verify the Adaptive Link-State Protocol (ALP), a new link-state routing protocol that does not require the state of each link to be flooded to the entire internetwork, or to entire areas if hierarchical routing is used. A router in ALP disseminates link-state updates incrementally to its neighbors for only those links along paths used to reach destinations. Link-state updates are validated using time stamps and contain the same information used in other link-state protocols. For the case of neighbor routers connected through a broadcast medium, a designated router is distributedly elected for each link state reported over the medium, rather than requiring a designated router to report every topology change over the broadcast medium, like OSPF does. Simulation experiments illustrate that ALP is as efficient as the Distributed-Bellman Ford algorithm when distances to destinations do not increase and resources do not fail, and more efficient than traditional link-state protocols based on flooding after distances increase or resources fail. ALP also outperforms the link-vector algorithm (LVA), which is the only prior routing algorithm based on selective dissemination of link states.

1. Introduction

The majority of the work on routing protocols for internetworks has proceeded in two main directions: distance-vector protocols (e.g., BGP [12], IDRP [13], RIP [7], and EIGRP [1]) and link-state protocols (e.g., ISO IS-IS [8] and OSPF [9]). In a distance-vector protocol, routers exchange path information (e.g., distance or complete path to one or more destinations) and compute their preferred paths in a distributed manner. Several routing algorithms based on distance vectors have been proposed to eliminate the counting-to-infinity problem that prevents the Bellman-Ford algorithm from working efficiently in large networks (e.g., [5, 3, 11]) and a number of these algorithms have been shown to outperform the traditional approach used in implementing link-state routing. Most approaches to link-state routing are based on topology broadcast [4, 10]. Unfortunately, disseminating complete link-state information to all routers incurs excessive communication overhead. The link-vector algorithm (LVA) [6] was recently proposed to avoid the overhead of topology broadcast when using link-state information. In LVA, each router updates its neighbors with the state of each of the links it uses to reach a destination through one or more preferred paths, and also informs them of the links that it stops to reach destinations. Updates to or deletions of links are propagated incrementally, based on the distributed computation of preferred paths at routers, just like distance information propagates in a distance-vector algorithm.

Although efficient algorithms have been proposed based on both link-state and distance-vector information, link-state routing is more efficient than distance-vector routing when constraints are placed on the paths offered to destinations, which is the case for QoS routing offering paths with required delay, bandwidth, reliability, cost, or other parameters. The communication overhead

of a link-state protocol increases to the extent that more parameters have to be communicated for each link whose state is updated, i.e., the added overhead is at most linear with the number of link parameters. In contrast, the communication overhead of a distance-vector protocol grows with the number of combinations of values of the link parameters needed to define the quality of paths.

As the Internet continues to evolve to support QoS routing, obtaining more efficient approaches to link-state routing has become an important design and engineering problem. This paper presents a new link-state routing protocol for internetworks called ALP (adaptive link-state protocol). In ALP, a router sends updates to its neighbors regarding the links in its preferred paths to destinations. Each router decides which links to report to its neighbors based on its local computation of preferred paths. In contrast to LVA, a router does not ask its neighbors to delete links; instead, a router simply updates its neighbors with the most recent information about those links it decides to take out of its preferred paths. Link states are validated using time stamps and a router accepts only more recent link-state updates.

Link-state information is aged out at each router just like in traditional link-state protocols. Furthermore, when multiple routers are connected through a broadcast medium (e.g., a LAN), they elect distributedly a designated router for each link reported over the broadcast medium; this reduces the number of updates per link sent over a given network. Unlike OSPF, ALP does not require backbones and can be used with distributed hierarchical routing schemes proposed in the past for distance-vector routing. Because routers in ALP propagate link-state information selectively, it incurs less communication overhead than algorithms based on topology broadcast.

The following sections introduce the network model assumed throughout the rest of the paper, describe ALP, show that ALP converges to correct paths a finite time after the occurrence of an arbitrary sequence of link-cost or topological changes, calculate its complexity, and present simulation results comparing ALP's performance against the performance of an ideal topology-broadcast algorithm, the distributed Bellman-Ford algorithm (DBF), and LVA.

2. Network Model

In ALP, routers maintain a partial topology map of their network. In this paper we focus on flat topologies only, i.e., there is no aggregation of topology information into areas or clusters.

The topology of a network is modeled as a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of edges connecting the nodes. Each node has a unique identifier and represents a router with input and output queues of unlimited capacity updated according to a FIFO policy. For the purpose of routing-table updating, a Node A can consider another Node B to be adjacent (we call such a node a "neighbor") if there is link-level connectivity between A and B and A receives update messages from B reliably. Accordingly, we map a physical broadcast link connecting multiple nodes into multiple point-to-point bidirectional links defined for these nodes. A functional bidirectional link between two nodes is represented by a pair of edges, one in each direction and with a cost associated that can vary in time but is always positive.

An underlying protocol, which we call the neighbor protocol, assures that a router detects within a finite time the existence of a new neighbor, the loss of connectivity with a neighbor, and the reliable transmission of packets between neighbors. All messages, changes in the cost of a link, link failures, and new-neighbor notifications are processed one at a time within a finite time and in the order in which they are detected. Because of the neighbor protocol, ALP assumes that all messages transmitted over an operational link are received correctly and in the proper sequence within a finite time. Routers are assumed to operate correctly, and information is assumed to be stored without errors.

3. Operation of ALP

In ALP, each router reports to its neighbors the characteristics of every link it uses to reach a destination through a preferred path. The set of links used by a router in its preferred paths is called the *source graph* of the router. A router knows its adjacent links and the source graphs reported by its neighbors; the aggregation of a router's adjacent links and the source graphs reported by its neighbors constitute a partial *topology graph*. The links in the source graph and topology graph must be adjacent links or links reported by at least one neighbor. The router uses one or more local *route selection algorithms*, the topology graph to generate its own source graph, and a routing table specifying the successor, successors, or paths to each destination.

The basic update unit used in ALP to communicate changes to source graphs is the link-state update (LSU). An LSU reports the characteristics of a link; an update message contains one or more LSUs. For a link between router u and router or destination v, router u is called the *head node* of the link in the u to v direction. The head node of a link is the only router that can report changes in the parameters of that link.

The head of a link reports the state of the link in an LSU periodically or when the parameters of the link change. A router w other than the head of link (u, v) sends LSUs about the link when the link is in its source graph and it receives a more recent LSU for the link from any of its neighbors, when the link is added to its source graph, or when the link that has been removed from the source graph changes state. Therefore, the LSUs from a router specify the state of links that the router currently uses in its source graph and links that are removed from the source graph.

Each router ages the link-state information it receives and erases a link from its topology graph if the age of the link-state information reaches a maximum value.

Figure 1 illustrates the fact that routers in ALP have to maintain only partial topology information. For simplicity, this figure and the rest of the paper assume that a single parameter is used to characterize a link in one of its directions, which we will call the cost of the directed link. Furthermore, although any number and type of local route selection algorithms can be used in ALP, we describe ALP assuming that shortest paths are used for routing and that Dijkstra's shortest-path first is used locally at each router. Figure 1b through 1d show the selected topology according to ALP at the routers indicated with filled circles. Solid lines represent the links that are part of the source graph of the respective router, dashed lines represent links that are part of the router's topology graph but not of its source graph. Arrowheads on links indicate the direction of the link stored in the router's topology graph. Router x's source graph shown in Figure 1b is formed by the source graphs reported by its neighbors y and z, and the links for which router x is the head node (namely links (x, y) and (x, z)). From the figure, the savings in storage requirements are clear, even for the small example shown in the figure.

We have developed a routing protocol framework based on the API provided by gateD [14] for implementation of routing protocols (Figure 2). A Hello Protocol is used to detect the presence of new neighbors and the loss of connectivity to them; the Retransmission Protocol is responsible for delivering update messages correctly and in the proper sequence to neighbors, as long as the physical link and network interface are operational. Among other things, the gateD API manages the routing forwarding table
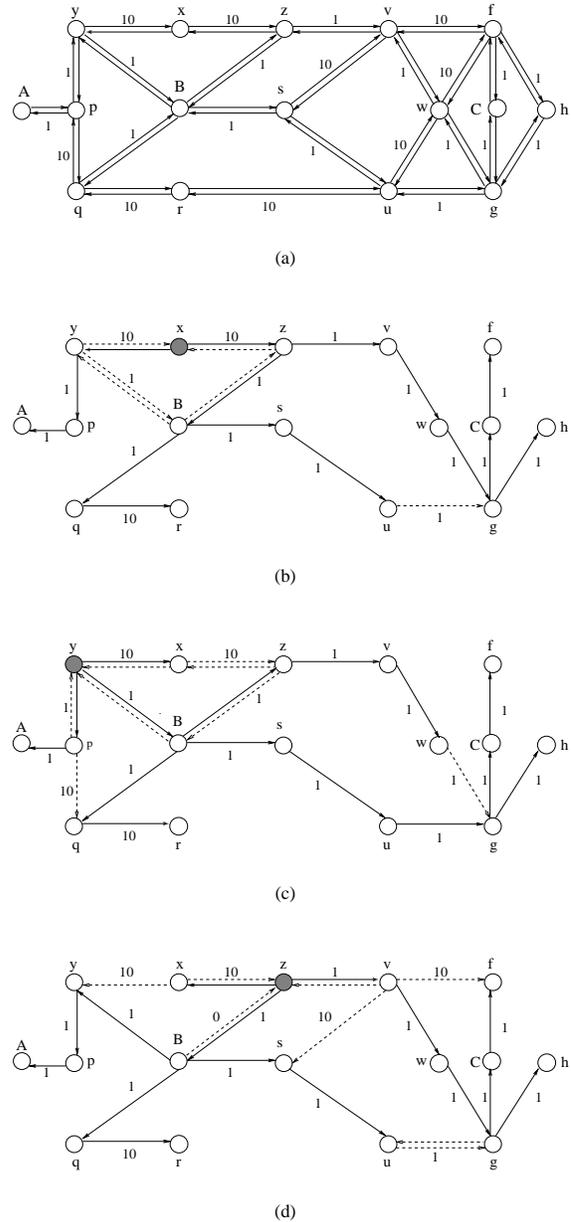

Figure 1: Topology as seen by routers indicated with filled circle. Solid lines indicate links in source graph; dashed lines indicate links in topology graph but not in source graph.

and relays indications to ALP reporting changes in the parameters of adjacent links, such as link cost changes, link failures, and link recoveries. In the rest of this paper we describe ALP assuming the services provided by the three underlying modules, which corresponds to the services provided by the neighbor protocol.

3.1. Information Stored and Exchanged

The LSU for a link (u, v) in an update message is a tuple (u, v, l, tod, sn, age) reporting the characteristics of the link, where l represents the cost of the link, tod and sn is the timestamp assigned to the LSU, and age corresponds to maximum age the LSU can achieve while in the topology graph.

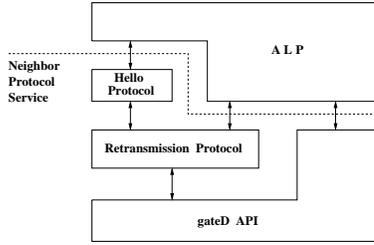A router i maintains a topology graph TG_i, a source graph

Figure 2: ALP protocol building modules

SG_i, a routing table, and the set of neighbors N_i. The record entry for link (u, v) in the topology graph of router i is denoted $TG_i(u, v)$ and is defined by the tuple $(u, v, l, tod, sn, age, F, l', tag, rn)$, and a parameter p in the tuple is denoted by $TG_i(u, v).p$.

$TG_i(u, v).F$ contains the set of network interfaces through which Node i has received up-to-date link-state information for (u, v), and $TG_i(u, v).F(f)$ holds the addresses of the neighbors who have reported up-to-date link-state information for (u, v) through interface f. In the example shown in Figure 1, router x's topology graph would have a record for link (s, u) indicating that y and z reported the same link. The link parameters l', tag, and rn are described in the next sections.

A vertex v in TG_i is denoted $TG_i(v)$ and contains a tuple $(d, pred)$ whose values are used on the computation of the source graph. $TG_i(v).d$ is the distance of the path $i \rightsquigarrow v$, and $TG_i(v).pred$ is v's predecessor in $i \rightsquigarrow v$.

The source graph SG_i is a subset of TG_i. The routing table contains record entries for destinations in SG_i, each entry consists of the destination address, the cost of the path to the destination, and the address of the next-hop towards the destination.

In our description, we refer to an LSU that has a cost infinity and the age field is greater than zero as a RESET, and refer to an LSU with an infinity link cost, a zero age, and a timestamp equal to the corresponding entry in the topology graph, as a DELETE. However DELETEs and RESETs are simply LSUs.

3.2. Validating Updates

Because of delays in the routers and links of an internetwork, update messages sent by a router may propagate at different speeds along different paths. Therefore, a given router may receive an LSU from a neighbor with stale link-state information, and a distributed termination-detection mechanism is necessary for a router to ascertain when a given LSU is valid and avoid the possibility of LSUs circulating forever. ALP implements the termination-detection mechanism used in several prior link-state protocols based on topology broadcast [9, 10], which consists in time stamps.

A router receiving an LSU accepts the LSU as valid if the received LSU has a larger timestamp than the timestamp of the LSU stored from the same source, or if there is no entry for the link in the topology graph. There is a special case in which a router other than the head of the link can change the cost of a link to infinity and report the new cost to the neighbors; this type of LSU will be considered valid under certain circumstances, as discussed in the next section. Each LSU sent by the same source specifies the current timestamp and the maximum age for that LSU (which is in the order of an hour). Every router that accepts an LSU decrements its age by at least one and also decrements the age while the LSU sits in memory. The timestamp of an LSU consists of two values: time-of-the-day (tod), and sequence number (sn). The tod value corresponds to the number of seconds elapsed from midnight (t) to the moment a timestamp must be assigned to the LSU. The head of a link that generates two or more LSUs with the same tod value uniquely identifies them by assigning different sns from a linear sequence-number space starting at 0.

The head of the link can reset sn whenever $\mid tod$ - tod of last LSU $\mid \geq 1$. The maximum age assigned to an LSU must not be larger than the maximum value of t, which is reseted every 24 hours. This leads to a *self-stabilizing* system because it makes the timestamp look like an "unbounded register model." Alternatively, a large linear sequence number space can be used, together with a reset mechanism for the sequence number to guard against malfunctions. We opt for the timestamp method in order to make our treatment of ALP simple.

3.3. Processing Input Events

An update message from a router k consists of a list of LSUs reporting incremental updates to its source graph and deletion of links from the topology graph not caused by aging; the procedure *Update* (Figure 3) is executed when a router i processes an update message. First, the topology graph is updated, then the source graph is updated, which may cause the router to update its routing table and to send its own update message.

An LSU for (u, v) updates the topology graph if its timestamp is larger than the timestamp maintained for the same link in the topology graph, or no entry for the link exists in the topology graph, or the entry in the topology graph has the cost set to infinity and the LSU has the same timestamp as the entry in the topology graph but the link cost is not infinity.

An LSU is considered outdated not only if it specifies a timestamp that is smaller than the one in the topology graph, but also if the timestamps are the same and the LSU carries a link cost infinity, while the entry in the topology graph has a cost different than infinity and the link is in the source graph. The reception of an outdated LSU causes the router to send an LSU with up-to-date information to the neighbor that originated the update message.

If the LSU is a valid RESET and there is an entry in the topology graph for the link, the LSU is forwarded to the neighbors.

A new source graph is computed and the routing table is updated if new link-state information is added to the topology graph or links are deleted from the topology graph. The shortest-path tree is generated by running Dijkstra SPF algorithm on the topology graph.

Rather than generating delete updates every time a link is removed from the source graph, as is the case in LVA, ALP reports to its neighbors the new value of the parameters of a link removed from the source graph if the cost of the link has increased. For those links that are removed from the source graph and that had not a cost increase, the node will only announce their removal when it learns that the cost of such links increased. The links stored in the topology graph have a tag that is used to keep track of those links that had the source graph removal announcement postponed, and gives the current state of the link in the state diagram of Figure 4. The state diagram shows the transition to a new state for a link $l = (u, v)$, given its current state and the type of input event received for the link. The tag of a link (u, v) for Node i is denoted $TT_i(u, v).tag$, and its possible values at time t are the following:

0 : Link (u, v) is not in the source graph at time t_i, where $t_{reset} \leq t_i \leq t$ and t_{reset} is the time the link last transitioned to State 0. The tag of a link is set to 0 when the link is inserted into the topology graph or when the cost of the link increases.

1 : Link (u, v) is in the source graph at time t.

2 : Link (u, v) is not in the source graph at time t, but it was in the source graph at time t_i, where $t_{reset} \leq t_i < t$. The removal of the link from the source graph had not been announced.

The transition to NIL in the state diagram corresponds to the deletion of the link from the topology graph. The description of possible input events summarized on the state diagram in Figure 4 are given in Figure 5.

In our current implementation of ALP, a link in the topology graph has just one *reporting neighbor*. This contrasts with

```
Update(k, msg)
{
    if ( msg ≠ ∅ )
        changed ← Update_Topology_Graph(k, msg);
    else
        changed ← TRUE;
    if ( changed = TRUE )
    {
        newSG ← Build_Shortest_Path_Tree();
        Process_Cost_Increase_State_1(newSG);
        Process_Cost_Increase_State_2(newSG);
        Process_Links_RemovedFrom_SG(newSG);
        event ← Update_Routing_Table(newSG);
        if ( event = NEW_LINK or
             event = PARAMETER_CHANGE or
             event = NEWSG_EMPTY )
        {
            Compare_Source_Graphs(SG_i, newSG);
        }
        SG_i ← newSG;
    }
    if ( k ≠ i )
        Send();
}
```

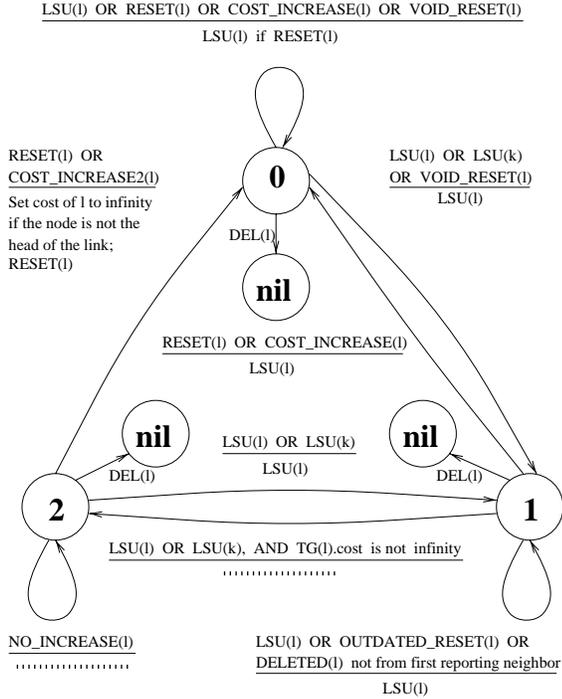Figure 3: Processing update message msg received from neighbor k

Figure 4: State diagram for a link l

LVA, which considers a reporting neighbor to be any neighbor that has reported an LSU with sequence number that matches the sequence number for the link in the topology graph. The reporting neighbor for a link (u, v) in the topology graph is denoted $TT_i(u, v).rn$, and consists of the address of the neighbor that last reported a valid LSU for the link if the state of (u, v) is 0, or the address of the neighbor that is in the shortest-path to u if (u, v) is in the source graph (i.e, the state of (u, v) is 1), or the address of the neighbor that was in the shortest-path to u at the time link (u, v) was removed from the source graph and transitioned to State 2. The reporting neighbor of a neighbor's adjacent link is the neighbor itself.

LSU(l) :	LSU for link l other than RESET(l), VOID_RESET(l), OUTDATED_RESET(l), COST_INCREASE(l), COST_INCREASE2(l), and DELETE(l).
RESET(l) :	LSU with cost infinity generated when link l fails or when l transitions from State 2 to State 0.
VOID_RESET(l) :	Timestamp of LSU is equal to the timestamp of the link in the topology graph, the cost of the LSU is not infinity, and the cost of the link in the topology graph is infinity.
OUTDATED_RESET(l) :	Timestamp of link in LSU is equal to the timestamp of the link in the topology graph, the cost of the LSU is infinity, the age of the LSU is greater than zero, and link l is in the source graph.
COST_INCREASE(l) :	Cost of link l has increased.
COST_INCREASE2(l) :	Cost of LSU is greater than $TT_i(l).l'$.
DELETE(l) :	LSU reporting link l was removed from the topology graph.
DEL(l) :	DELETE(l) or there is no reporting neighbor for link l.
NO_INCREASE(l) :	cost of link l in LSU is not greater than $TT_i(l).l'$

Figure 5: Input events of the state diagram

When a link (u, v) is removed from the source graph SG_i and transitions to State 2, Node i needs to store the current cost of the link in $TT_i(u, v).l'$, which is used for checking increases in the cost of the link while (u, v) is in State 2.

After computing the new source graph $newSG$ (Figure 3) the router generates link-state updates for those links whose cost has increased and were removed from the source graph, i.e., the links have transitioned from State 1 to State 0 (Figure 4). Then, the router generates RESETs for those links that had the cost increased while in State 2. If the router that transitions a link from State 2 to State 0 is not the head of the link, the cost of the link in the topology graph is set to infinity. Procedure *Update* then executes *Process_Links_RemovedFrom_SG* which sets $TT_i(u, v).l' \leftarrow TT_i(u, v).l$ and $TT_i(u, v).tag \leftarrow 2$ for each link (u, v) that was removed from the source graph and had not the cost increased. The router then compares the new source graph $newSG$ against the current source graph SG_i, and LSUs are created with the link-state information for links that are in $newSG$ but not in SG_i, or that are in both graphs but had their timestamp changed. After LSUs are generated, $newSG$ becomes the current source graph SG_i.

The procedure *Update* is run having as input an empty message when a link-state information has been erased from the topology graph due to aging.

If a link cost changes, then its head node is notified by an underlying protocol. The router then runs *Update* with the appropriate message as input; the LSU in the message gets a current timestamp. This holds for simple changes in link cost, as well as for a link failure. The same approach is used for a new link or a link that comes up after a failure. When a router establishes connectivity to a new neighbor, the router sends its complete source graph to the neighbor (much like a distance vector protocol sends its complete routing table).

When the link (i, k) to neighbor k fails, the topology graph is updated to erase the neighbor from the set $TT_i(i, k).F(f)$, and a RESET for (i, k) is transmitted to the neighbors. For each link (u, v) in the topology graph whose reporting neighbor is router k, router i generates a DELETE update for (u, v) and deletes the link from the topology graph. A router that receives a DELETE update from a node other than the reporting neighbor transmits to the sender of the DELETE an LSU with the current state of the link if the link is in the source graph. This guarantees that the tree of reporting neighbors for link $(u, v) \in E$, formed by the links $(i, TT_i(u, v).rn) \in E$, for each Node $i \in V$, is updated accordingly. Link-state information for failed links that have a reporting neighbor must be kept in the topology graph in order to validate incoming LSUs for the link.

Consider the topology in Figure 1 and assume that link (y, B) increases its cost dramatically (e.g., from 1 to 100). Node y processes the link-cost increase and generates a new source graph;

the update message sent by Node y to its neighbors specifies an LSU for the link (y, B) with the new cost and LSUs for links (p, q), (x, z), (z, B) and (w, g), which must now be used to reach all the nodes in the graph. Note that router y does not inform its neighbors that it removed links (B, z), (B, q) and (u, g) from its source graph, as would be the case in LVA [6], but makes the links to transition from State 1 to State 2 (see Figure 4). An important difference between ALP and LVA is that in ALP a router informs its neighbors of a link removed from its source graph only if it is removed because its cost increased or because there is no reporting neighbor for the link anymore (in which case the cost is set to infinity). LVA reports all deletions to the source graph, and all such deletions represent infinite link costs.

3.4. Electing Designated Routers

Because routers in ALP communicate partial topology information to their neighbors, defining a designated router as in OSPF to be in charge of sending topology information over a network connecting multiple neighbor routers cannot be applied. In ALP, a link is assigned a designated router if it needs to be reported by at least one router over a given broadcast medium (a LAN, a network, or a link).

The idea of assigning one designated router per link-state update consists of making only one router responsible for reporting the link-state update in a broadcast link. In this way, when an adjacency is formed with a new neighbor x through a broadcast link, x will receive only one copy of the link-state information for a link (u, v) from the routers that already have adjacencies in the link. To accomplish this, the topology graph entry $TG_i(u, v)$ maintains the set of interfaces through which Node i has received link-state updates for the link (u, v), as well as the list of neighbors attached to the interface that have reported the link-state update. $TG_i(u, v).F$ contains the set of interfaces, and $TG_i(u, v).F(f)$ is the list of neighbors with adjacencies through interface f that have reported a link-state update.

When Node i receives a valid LSU from neighbor x for link (u, v), the LSU for (u, v) is forwarded to all neighbors except x, i then stores neighbor's x's address in the list $TG_i(u, v).F(f)$

$$TG_i(u, v).F \leftarrow \emptyset;$$
$$TG_i(u, v).F \leftarrow TG_i(u, v).F \cup \{f\};$$
$$TG_i(u, v).F(f) \leftarrow \{ \text{address of } x \};$$

If the received LSU (u, v) is not valid, but its timestamp is equal to the one stored in the topology graph, Node i also stores neighbor's x's address in the list $TG_i(u, v).F(f)$, where f is the incoming interface

if ($f \notin TG_i(u, v).F$)
$\quad TG_i(u, v).F \leftarrow TG_i(u, v).F \cup \{f\};$
$TG_i(u, v).F(f) \leftarrow$ **SORT**$(TG_i(u, v).F(f) \cup \{ \text{address of } x \});$

When a router i reports an LSU (u, v) through interface f, it adds the address of f into $TG_i(u, v).F(f)$.

The list of neighbors $TG_i(u, v).F(f)$ is always sorted in ascending order of router addresses. The first router address in the list $(TG_i(u, v).F(f).0)$ is the one with the smallest address.

When a new adjacency is formed to a neighbor k through interface f, Node i will only report an LSU for link (u, v) to k if i is the head node of the link, or $f \in TG_i(u, v).F$ and $TG_i(u, v).F(f).0$ equals f's address. The procedure *DST_SET* shown in Figure 6 returns the set of interfaces through which a link-state update for link (u, v) can be announced, and *ANNOUNCE* returns TRUE if Node i can announce an LSU for (u, v) through the interface i has connectivity to neighbor k. For any given link used by a set of routers connected to a LAN, the router with the smallest ID is the only one allowed to send LSUs for the link over the LAN.

DST_SET(u, v)
{
$\quad F \leftarrow$ set of operational interfaces;

$\quad$**if** ($r \neq i$)
$\quad\quad$**for each** (interface $f \in TG_i(u, v).F$)
$\quad\quad\quad$**if** ($TG_i(u, v).F(f) \neq \emptyset$ and
$\quad\quad\quad\quad TG_i(u, v).F(f).0 \neq f.address$)
$\quad\quad\quad\quad F \leftarrow F - \{f\};$

$\quad$**return** F;
}

ANNOUNCE(k, u, v)
{
$\quad announce \leftarrow$ TRUE;
$\quad f \leftarrow$ interface attached to k;

$\quad$**if** ($r \neq i$ and $f \in TG_i(u, v).F$)
$\quad\quad$**if** ($TG_i(u, v).F(f) \neq \emptyset$ and
$\quad\quad\quad TG_i(u, v).F(f).0 \neq f.address$)
$\quad\quad\quad announce \leftarrow$ FALSE;

$\quad$**return** $announce$;
}

Figure 6: Procedures used to determine to which neighbors a link-state update can be announced

When i detects loss of connectivity to a neighbor x attached to a broadcast link through interface f, and x was the only router in $TG_i(u, v).F(f)$, router i will announce an LSU for (u, v) in the broadcast link if it has a path to destination v whose successor is not a neighbor in the broadcast link. This guarantees that new neighbors that have not received link-state information about (u, v) will get it as soon as i detects lack of connectivity to x.

4. ALP Correctness

In this section we show that routers executing ALP stop disseminating link-state updates and obtain shortest paths to destinations within a finite time after the cost of one or more links changes and there are no more changes afterwards.

For simplicity of exposition, we assume that all links are bidirectional point-to-point links and that shortest-path routing is implemented. Let t_0 be the time when the last of a finite number of link-cost changes occur, after which no more such changes occurs. The network $G = (V, E)$ in which ALP is executed has a finite number of nodes ($|V|$) and links ($|E|$), and every message exchanged between any two routers is received correctly within a finite time. According to ALP's operation, for each direction of a link in G, there is a router that detects any change in the cost of the link within a finite time.

The following theorems relay on the use of timestamps as described in Section 3.2; the same approach applies if an alternative update validation scheme based on resets is used. We also assume that all routers use the same type of tie-braking rules in computing shortest paths, e.g., if a shortest path to j is obtained through two different relays, routers choose the relay with the smallest identifier.

Lemma 1: *The dissemination of LSUs in ALP, other than DELETEs, stops a finite time after t_0.*

Proof: A router that detects a change in the cost of any outgoing link must update its topology graph, update its source graph as needed, and send an LSU if the link is added to or is updated in its source graph. Let l be the link that last experiences a cost change up to t_0, and let t_l be the time when the head of link l originates the last LSU of the sequence of LSUs originated as a result of the link-cost change occurring up to t_0. Any router that receives the LSU for link l originated at t_l must process the LSU within a finite time, and decides whether or not to forward the LSU based on its

updates to its source graph. A router can accept and propagate an LSU only once because each LSU has a timestamp; accordingly, given that G is finite, there can only be a finite chain of routers that can propagate the LSU for link l originated at t_l, and the same applies to any LSU originated from the finite number of link-cost changes that occur up to t_0. Therefore, ALP stops the dissemination of LSUs a finite time after t_0. $\square$

Lemma 2: *The dissemination of DELETEs in ALP stops a finite time after t_0.*

Proof: A router i that detects failure of the link to the reporting neighbor of a link l in the topology graph must delete l from the topology graph, update its source graph, and send a DELETE LSU for link l. Let the failed link be the link that last experiences a cost change up to t_0, and let t_l be the time when Node i originates the last LSU of the sequence of LSUs originated as a result of the link-cost changes occurring up to t_0. Any router that receives the DELETE for link l originated at t_l must process the DELETE within a finite time, and forwards the DELETE after deleting l from its topology graph if the sender of the DELETE was the first reporting neighbor of l. A router can accept and propagate a DELETE only once because the link is deleted from the topology graph when the DELETE is accepted for the first time, and a DELETE for link l is not propagated if link l is not in the topology graph of the router processing the DELETE. Given that G is finite, there can only be a finite chain of routers that propagate the DELETE for link l originated at t_l, and the same applies to any DELETE originated from the finite number of link-cost changes that occur up to t_0. Therefore, ALP stops the dissemination of DELETEs a finite time after t_0. $\square$

Theorem 1: *The dissemination of LSUs in ALP stops a finite time after t_0.*

Proof: The proof is immediate from Lemmas 1 and 2. $\square$

From Theorem 1, it must be true that there is a time t_s when no more LSUs are queued or in transit anywhere in the network.

Lemma 3: *A router with a tag value of 1 for link l at time t_s must be the head of the link or have at least one neighbor with a tag value of 2 or 1.*

Proof: The proof is obvious if the router is the head of the link. Assume that router i is not the head of link l and that all of its neighbors have tags equal to 0 at time t_s.

Because router i is not the head of link l and has link l in its source graph, it must have received an LSU reporting l from at least one neighbor k at some time $t' < t_s$, which required k to have link l in its source graph at that time, i.e., to have a tag value of 1 for l at time $t' < t_s$. By assumption, k has a tag equal to 0 for link l, which means that k must have transitioned its tag value from 1 or 2 to 0 before time t_s. According to ALP's operation, at the time of its transition, k must have sent an LSU reporting an increase in the cost of link l, and it may also have sent LSUs for links that k adds or updates in its source graph. Because by assumption no LSUs are queued at or in transit to router i at time t_s, i must have processed the LSU from k indicating the cost increase for l, as well as any LSUs needed to bring i topology graph consistent with k's source graph.

Because none of i's neighbors use link l in their shortest paths, because i has received the LSUs from k that exclude link l from being part of any shortest path from k, and because all routers use the same tie-braking rules for shortest paths, it follows that router i cannot use l in any of its shortest paths, because k does not. Accordingly, router i must transition to a tag value of 0 or 2 after processing the LSUs from k, and the Lemma is true. $\square$

Lemma 4: *A router with a tag value of 2 for link l at time t_s must be the head of the link or have at least one neighbor with a tag value of 2 or 1.*

Proof: The proof is obvious if i is the head of link l, because i may have shorter paths to the tail of the link than the link itself. Assume that router i is not the head of link l and that all its neighbors have tags equal to 0 for link l at time t_s.

Because router i is not the head of link l and has link l in its source graph, it must have received an LSU reporting l from at least one neighbor k at some time $t' < t_s$. Following the same line of argument used in the proof of Lemma 3, we can show that, at time t_s, router i must have processed the LSU from k indicating the cost increase for l, together with any LSUs needed to bring i topology graph consistent with k's source graph. According to ALP's operation, when router i has a tag value of 2 for link l and receives an LSU reporting a cost increase for l, then it must transition to a tag value of 0 and send an LSU; therefore, the theorem is true. $\square$

Theorem 2: *In a connected network, and in the absence of link failures, all routers have the most up-to-date link-states they need to compute shortest paths to all destinations within a finite time after t_s.*

Proof: The proof is by induction on the number of hops of a shortest path to a destination, and is basically a generalization of the proof for SPTA [2].

Consider the shortest path from router s_0 to a destination j at time t_s, and let h be the number of hops along such a path. For $h = 1$, the path from s_0 to j consists of one of the router's outgoing links. By assumption, an underlying neighbor protocol provides the correct parameter values of adjacent links within a finite time; therefore, the Theorem is true for $h = 1$, i.e., s_0 must have link (s_0, j) in its source graph, which means that its tag value for the link is 1 and it must have sent its neighbors an LSU for that link.

Assume that that any router with a path of n or fewer hops to j has the correct link-state information about all the links in the shortest path to j, and consider the case in which the path from s_0 to j at time t_s is $n + 1$ hops.

Router s_0 has a tag value of 1 for each link in the shortest path to j, because the path belongs to its source graph. For any such link l in the shortest path to j, it follows from Lemma 3 that the router has a neighbor that by time t_s has reported an LSU it can believe that specifies the up-to-date cost of l. Accordingly, the shortest path from s_0 to j must be through a neighbor s_1 with a tag value of 1 or 2 for link l, which means that s_1 must send the most up-to-date LSUs it receives for each link in its shortest path to j. The sub-path from s_1 to j has $h - 1$ hops and, by the inductive assumption we have made, such a path must be the true shortest path from s_1 to j by time t_s. Because all routers use the same tie-braking rules to choose shortest paths, this also means that that s_1 must have a tag value of 1 for each link in its shortest path to j.

Because it is also true that s_0 has the most recent link-state information about link (s_0, s_1), it follows that s_0 has the most recent information about all the links in its chosen path to j. The Theorem is therefore true, because the same argument applies to any chosen destination and router. $\square$

Theorem 3: *In a connected network, and in the absence of link failures, a tree of reporting neighbors for a link l will be formed within a finite time after t_s.*

Proof: For the neighbors of the head of the link l the root of the tree of reporting neighbors is the head of the link. The tree of reporting neighbors consist of routers whose value of the tag for link l can be 0, 1, or 2. Routers that have the tag set to 1 elect as the reporting neighbor for l the next hop in the shortest-path to the head of the link. Given that the source graph is computed within a finite time after t_s according to Theorem 2, the subtree of the tree of reporting neighbors that includes the source graph is computed a finite time after t_s. Whenever a valid link-state update for l is processed and l has a tag set to 0 or 2 after computing the source graph, the reporting neighbor is set to be the router which sent the update message. Together with Lemma 1, this implies that the Theorem is true. $\square$

Theorem 4: *All the routers of a connected network have the most up-to-date link-state information needed to compute shortest paths to all destinations.*

Proof: The result is immediate from Theorem 2 in the absence of link failures. Consider the case in which the only link that fails in the network by time t_0 is link (s, d). Call this time $t_f \leq t_0$. According to ALP's operation, router s sends an LSU reporting an infinite cost for (s, d) within a finite time after t_f; furthermore, every router receiving the LSU reporting the infinite cost of (s, d) must forward the LSU if the link exists in its topology graph, i.e., the LSU gets flooded to all routers in the network that had heard about the link, and this occurs within a finite time after t_0. It than follows that no router in the network can use link (s, d) for any shortest path within a finite time after t_0. DELETE updates will also be propagated by router s for all those links in the topology graph that had router d as the reporting neighbor, as described in the proof of Lemma 2. A router sends and LSU for a link l to the router that transmitted a DELETE update if l is in the source graph and the router is not the reporting neighbor of l. Accordingly, within a finite time after t_0 all routers must only use links of finite cost in their source graphs; together with Theorem 2, this implies that the Theorem is true. $\square$

Theorem 5: *If destination j becomes unreachable from a network component C at t_0; the topology graph of all routers in C includes no finite-length path to j.*

Proof: ALP's operation is such that, when a link fails, its head node reports an LSU with an infinite cost to its neighbors, and the state of a failed link is flooded through a connected component of the network together with DELETE updates for those links j that are part of the disconnected component to all those routers that knew about the link. Because a node failure equals the failure of all its adjacent links, it is true that no router in C can compute a finite-length path to j from its topology graph after a finite time after t_0. $\square$

Note that, if a connected component remains disconnected from a destination j all link-state information corresponding to links for which j is the head node is updated when the network components get connected.

The previous theorems show that ALP sends correct routing tables within a finite time after link costs change, without the need to replicate topology information at every router (like OSPF does) or use explicit delete updates to delete obsolete information every time the source graph of a router changes (like LVA does).

5. Performance

ALP has the same communication, storage, and time complexity than LVA. However, worst-case performance is not truly indicative of ALP's performance advantage over LVA. Because link-states are deleted from the topology graph of a router, rather than after receiving explicit delete updates from neighbors, ALP incurs less communication overhead than LVA. ALP also compares favorably against recent distance vectors based on "source tracing" [3] [11], or the diffusion of distances [5], which do solve the looping problems of RIP and RIP-2.

Compared to the diffusion of distances, ALP disseminates link-state information from only the source of an LSU out to those routers that need the link, while DUAL requires distances to be disseminated from the source of the update out to those routers whose path included the source of the update, followed by replies going back to the source. Hence, when such coordination occurs in DUAL, ALP incurs half the communication overhead.

Compared to source tracing algorithms, it is interesting to observe that in ALP a router notifies its routing tree to its neighbors by specifying each link in the tree, while in a source tracing algorithm the same tree is specified by reporting, for each node on the tree, the distance from the root of the tree to the node and the identification of the previous node on the tree. Clearly, there is an one-to-one mapping between the two representations, which means that the same routers will receive LSUs or distance-vectors updates reporting changes to the routing tree. In other words, the communication overhead is the same. Furthermore, in terms of communication overhead, it is not possible to attain a smaller overhead than sending updates (of links or distances) to

only those routers whose shortest paths are affected by a topology change, i.e., ALP and source-tracing algorithms make very efficient use of communication, and both amount to a more distributed implementation of Dijkstra's SPF algorithm than protocols using topology broadcast (e.g., OSPF), which *replicate* SPF runs at each router.

In terms of storage overhead, ALP has similar overhead than distance-vector protocols and link-state protocols for the case of shortest-path routing. ALP and other link-state protocols become more attractive than distance-vector protocols when providing multiple paths to the same destinations becomes necessary.

Because of the way in which ALP updates link-state information, ALP outperforms any topology broadcast protocol. Because ALP does not use "delete" updates we expect ALP to outperform LVA, specially when nodes fail or resources recover. Furthermore, because no counting-to-infinity occurs in ALP, ALP should outperform protocols based on the Bellman-Ford algorithm. To verify this, we ran a number of simulation experiments to compare its average performance against DBF, topology-broadcast (called LSA in prior literature), and LVA. We used the same topology and experiment reported in [6] in order to compare ALP against the best-performing published results for other approaches. The performance metrics consist of the number of steps and update messages that are required for each algorithm to converge (i.e., the algorithm stops sending messages), and the size of these updates. When a router receives an update message, it compares its local step counter with the sender's counter, takes the maximum and increments the count. Update messages are processed one at a time in the order in which they arrive. Like LVA and LSA, ALP uses Dijkstra's algorithm to compute the local shortest-path tree. The results presented are based on simulations for the DOE-ESNET topology [6] which was used in order to simply use published simulation results for the competing approaches. The graphs in Figure 7 show the results for every single link changing cost from 1 to 2; in Figures 8 and 9 for every link failing and recovering; as well as every node failing and recovering (Figures 10 and 11). All changes were performed one at a time, and the algorithms had time to converge before the next change occurred. The ordinate of Figures 7, 8, and 9 represent identifiers of the links, and the ordinate of Figures 10 and 11 represent the identifiers of the nodes that are altered in the simulation.

ALP, DBF, and LVA propagate updates to only those routers affected by single link-cost changes (Figure 7). In contrast, LSA shows almost constant behavior because the same link-state update must be sent to all routers; ALP is the most efficient of the four algorithms. Each update message contains one link-state update in LSA, and an average of 1.10 links in ALP; the average number of messages transmitted in ALP is 43.36, 48.67 in LVA, and 57.45 in DBF.

Figure 8 depicts DBF suffering from *counting to infinity* in some cases. There is a small difference in the average number of updates and synchronization steps required in ALP and LVA. The average size of an ALP message is 2.40.

When a failed link recovers, ALP is superior to all three algorithms. The average number of messages in LVA is 70% more than in ALP; LSA exhibits the same behavior as with link-cost changes, and in average more than three times the number of update messages generated by ALP. With an average of 5.85 steps, ALP is twice as fast as LSA, and 50% faster than LVA. Messages in LSA are no longer one-link long due to the packets containing complete topology information sent over the recovering link.

ALP also shows to have the best performance of the four algorithms for failing nodes. DBF always suffers from *counting to infinity*. ALP needs to send 23% fewer updates than LSA, and 80% less the amount experienced by LVA. For recovering nodes, ALP shows to be more efficient than LVA, DBF, and LSA, both in terms of the amount of information sent through the network and speed of convergence.

The simulation results show that ALP has better overall average performance than LVA, LSA, and DBF. ALP behaves better than DBF and LVA when link cost changes and is always faster and produces less overhead traffic than LVA and LSA when resources are added to the network, and behaves better than the

ideal LSA when links or routers fail. This is precisely the desired result, and indicates that ALP is desirable even if multiple constraints are not an issue.

6. Conclusions

We have presented ALP, which we believe is the first example of a link-state protocol that has been shown to be more efficient than topology broadcast and recent distance-vector routing approaches. ALP is currently running in a small testbed implemented with PCs running gateD, and the very same code was used in the reported simulation experiment. The size of ALP's executable code including the Hello Protocol and the Retransmission Protocol (Figure 2) is 96 Kbytes, compared to the 226 Kbytes of OSPF. A novel feature in ALP is the use of designated routers per link for each broadcast medium, rather than a designated router for the entire medium, and its ability to accommodate partitioned areas and multi-hop or partitioned IP subnets, which makes it adaptable to *ad hoc* networks. Simulations using the actual code for ALP corroborate the fact that ALP achieves the most efficient way of disseminating update information in a routing protocol compared to topology broadcast, the distributed Bellman-Ford algorithm, and LVA. ALP addresses the complexity of today's approach to link-state routing by making the computation of routing trees using link-states costs a distributed computation and establishes link-state routing as the more efficient approach for the Internet, in terms of communication overhead and the ability to support efficient types of paths to destinations, which well become more important as QoS support emerges in the Internet.

7. References

[1] R. Albrightson, J.J. Garcia-Luna-Aceves, and J. Boyle, "EIGRP–A Fast Routing Protocol Based on Distance Vectors" *Proc. Networld/Interop 94*, Las Vegas, Nevada, May 1994.

[2] D. Bertsekas and R. Gallager, *Data Networks*, Second Edition, Prentice-Hall, Inc., 1992.

[3] C. Cheng, et al, "A Loop-Free Extended Bellman-Ford Routing Protocol without Bouncing Effect," *Proc. ACM SIGCOMM 89*, Austin, TX, September 1989.

[4] E. Gafni, "Generalized Scheme for Topology-Update in Dynamic Networks," *Lecture Notes in Computer Science* (G. Goos and J. Hartmanis, Eds.), No. 312, pp. 187-196, 1987.

[5] J.J. Garcia-Luna-Aceves, "Loop-Free Routing Using Diffusing Computations," *IEEE/ACM Trans. Networking*, Vol. 1, No. 1, 1993.

[6] J.J. Garcia-Luna-Aceves and J. Behrens, "Distributed, scalable routing based on vectors of link states," *IEEE Journal on Selected Areas in Communications*, Vol. 13, No. 8, 1995.

[7] C. Hedrick, "Routing Information Protocol," RFC 1058, Network Information Center, SRI International, Menlo Park, CA, June 1988.

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

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

[10] R. Perlman, "Fault-Tolerant Broadcast of Routing Information," in *Computer Networks*, North-Holland, Vol. 7, pp. 395-405, 1983.

[11] B. Rajagopalan and M. Faiman, "A New Responsive Distributed Shortest-Path Routing Algorithm," *Proc. ACM SIGCOMM 89*, Austin, TX, September 1989.
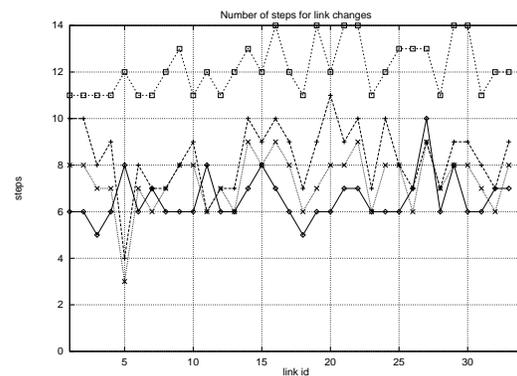
[12] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC 1654, July 1994.

[13] Y. Rekhter, "Inter-Domain Routing Protocol (IDRP)," *Internetworking: Research and Experience*, Wiley, Vol. 4, No. 2, June 1993, pp. 61-80.

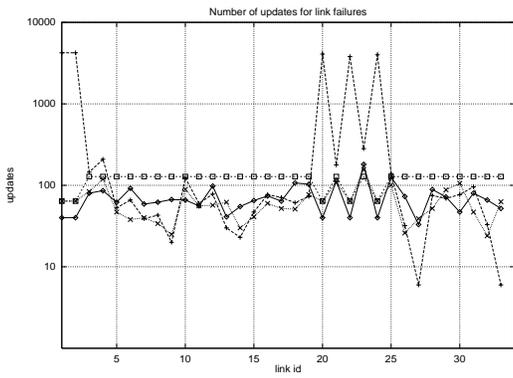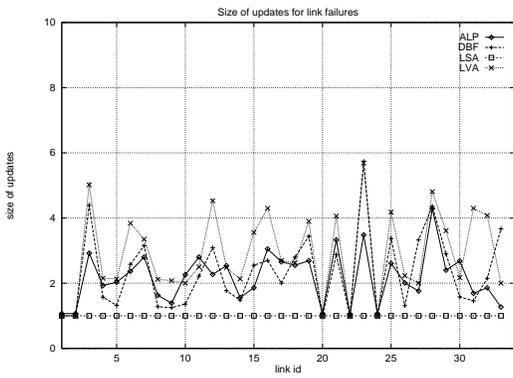[14] Merit GateD Consortium, http://www.gated.org
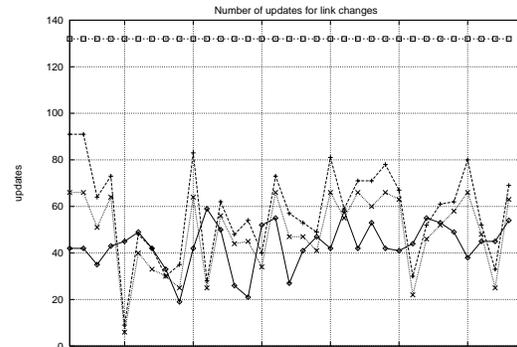
(a)
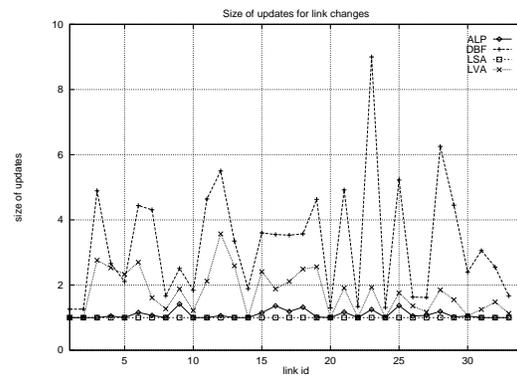
(b)

(c)
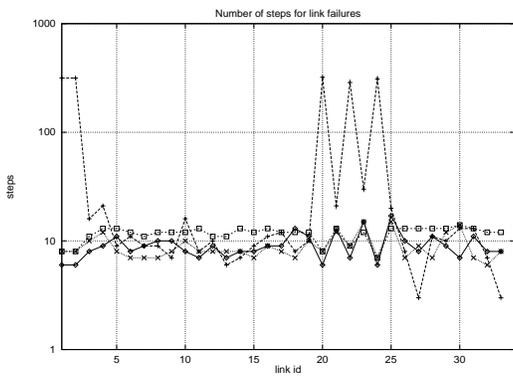
Figure 7: Links changing cost; (a) number of update messages, (b) average size of messages, and (c) number of steps for convergence.
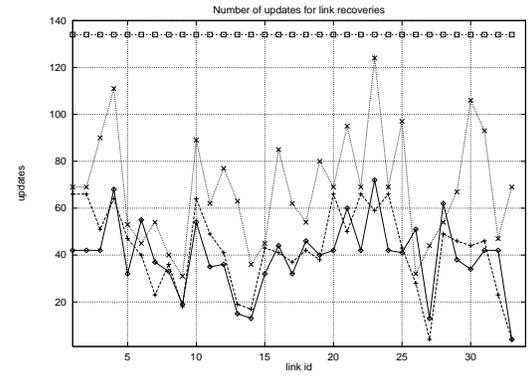
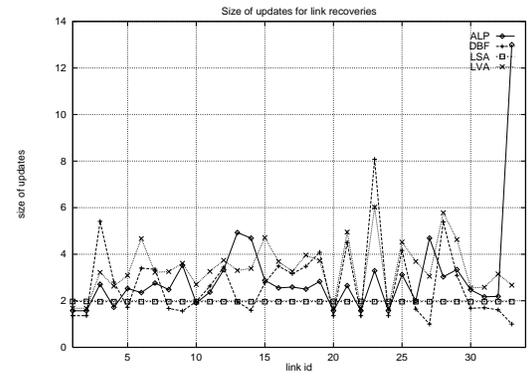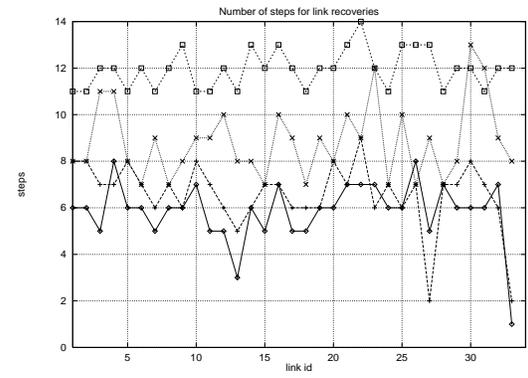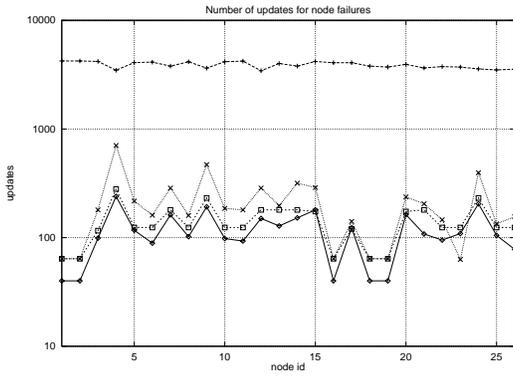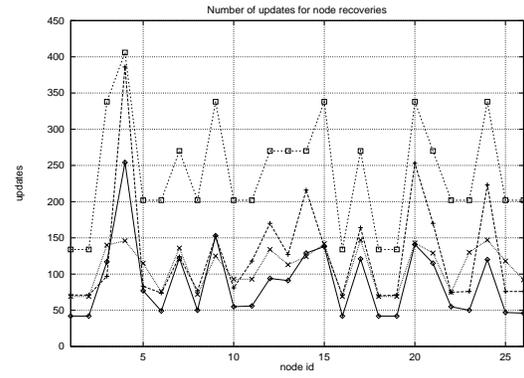Figure 8: Links failing; (a) number of update messages, (b) average size of messages, and (c) number of steps for convergence.
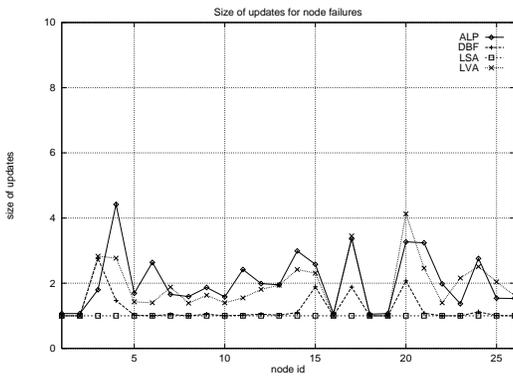

Figure 9: Links recovering after failure; (a) number of update messages, (b) average size of messages, and (c) number of steps for convergence.
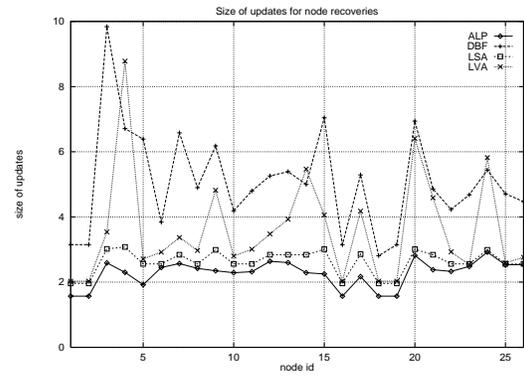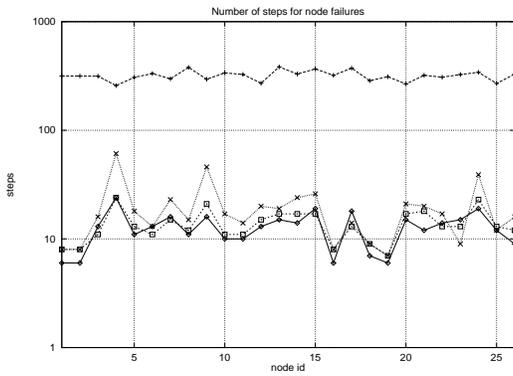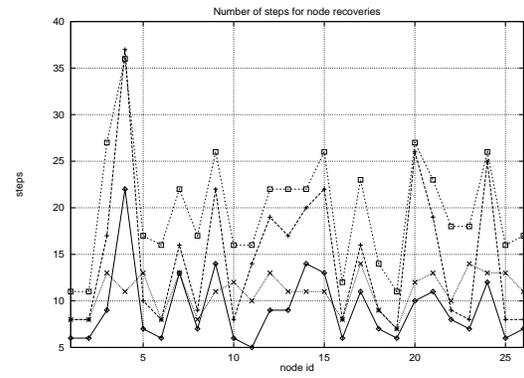
(a)

(b)

(c)

Figure 10: Nodes failing; (a) number of update messages, (b) average size of messages, and (c) number of steps for convergence.

(a)

(b)

(c)

Figure 11: Nodes recovering after failure; (a) number of update messages, (b) average size of messages, and (c) number of steps for convergence.