A Distributed Adaptive Cache Update Algorithm for the Dynamic Source Routing Protocol*

Xin Yu and Zvi M. Kedem
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012

Abstract

On-demand routing protocols use route caches to make routing decisions. Due to frequent topology changes, cached routes easily become stale. To address the cache staleness issue in DSR (the Dynamic Source Routing protocol), prior work mainly used heuristics with ad hoc parameters to predict the lifetime of a link or a route. However, heuristics cannot accurately predict timeouts because topology changes are unpredictable. In this paper, we present a novel distributed cache update algorithm to make route caches adapt quickly to topology changes without using ad hoc parameters. We define a new cache structure called a *cache table* to maintain the information necessary for cache updates. When a node detects a link failure, our algorithm proactively notifies all reachable nodes that have cached the broken link in a distributed manner. We compare our algorithm with DSR with path caches and with *Link-MaxLife* through detailed simulations. We show that our algorithm significantly outperforms DSR with path caches and with *Link-MaxLife*.

1 Introduction

In mobile ad hoc networks, nodes move arbitrarily, cooperating to forward packets to enable communication between nodes not within wireless transmission range. Frequent topology changes present the fundamental challenge to routing protocols. Routing protocols for ad hoc networks can be classified into two main types: proactive and reactive (on-demand). Proactive protocols attempt to maintain up-to-date routing information to all nodes by periodically disseminating topology updates throughout the network. On-demand protocols attempt to discover a route to a destination only when a node originates a packet. Several routing protocols use on-demand mechanisms, including AODV [14], DSR [7, 8], LAR [9], TORA [13], ZRP [3]. In this paper, we focus on DSR, which operates fully on-demand.

On-demand routing protocols use route caches to avoid the overhead and the latency of initiating a route discovery for each data packet. However, due to mobility, cached routes easily become stale. Using stale routes causes packet losses, increases packet delivery latency due to expensive link failure detections, and increases routing overhead. When responding to route requests from caches is used, stale routes will be quickly propagated to other nodes, aggravating the situation. Stale routes also seriously degrade TCP performance [4].

To address the cache staleness issue, prior work [5, 10, 12] proposed to use adaptive timeout mechanisms. Such mechanisms use heuristics with ad hoc parameters to predict the timeout of a link or a route. However, predetermined choice of ad hoc parameters for certain scenarios may not work well for others. The effectiveness of heuristics is limited by unpredictable topology changes. If timeout is set too short, valid links or routes will be removed; subsequent route discoveries introduce significant overhead. If timeout is set too long, stale routes will stay in caches. In addition, DSR uses a small cache size. Small caches with FIFO help evict stale routes but also remove valid ones. No single cache size provides the best performance for all mobility scenarios [5].

In this paper, we investigate how to make route caches adapt quickly to topology changes without using ad hoc mechanisms. When a node detects a link failure, our goal is to notify all reachable nodes whose caches contain the broken link to update their caches. To achieve this goal, we define a new cache structure called a cache table and present a distributed cache update algorithm. In a cache table, a node not only stores routes but also maintains the information necessary for cache updates. Each node maintains two types of information for each route: (1) how well the routing information is synchronized among nodes on the route, and (2) which neighbor node has learned which links through a ROUTE REPLY. Thus, for each cached link, a node knows which neighbor nodes have cached that link. When a link failure is detected, the algorithm notifies the neighborhood nodes that have that link in their caches. When a node receives a notification, the algorithm notifies selected neighbors. Therefore, the broken link information will be quickly propagated to all the reachable nodes that have the broken link in their caches.

We compare our algorithm with DSR with path caches and with *Link-MaxLife* [5], an adaptive timeout mechanism for link caches. We do not use promiscuous mode, which is an optimization for DSR [8]. Through detailed simulations, we show that our algorithm significantly outperforms DSR with path caches, improving packet delivery ratio by up to 13% for

*NYU Computer Science Department Technical Report TR2003-842, July 2003. Last revised: December 20, 2004.

50 node and 34% for 100 node scenarios, reducing packet delivery latency by up to 30% for 50 node and 25% for 100 node scenarios, and achieving 50% reduction in normalized routing overhead for 100 node scenarios. Compared with *Link-MaxLife*, our algorithm improves packet delivery ratio by up to 35%.

The rest of this paper is organized as follows. In Section 2 we give an overview of the DSR. In Section 3 we present the definition of a cache table and the distributed cache update algorithm. We present an evaluation of the algorithm in Section 4 and discuss related work in Section 5. Finally, in Section 6 we present our conclusions.

2 DSR: Dynamic Source Routing

DSR consists of two on-demand mechanisms: *Route Discovery* and *Route Maintenance*. When a source wants to send packets to a destination to which it does not have a route, it initiates a *Route Discovery* by broadcasting a ROUTE REQUEST. A node receiving a ROUTE REQUEST checks whether it has a route to the destination in its cache. If it has, it sends a ROUTE REPLY to the source including a source route, the concatenation of the source route in the ROUTE REQUEST and the cached route. Otherwise, it adds its address to the source route in the packet and rebroadcasts the ROUTE REQUEST. When the ROUTE REQUEST reaches the destination, the destination sends a ROUTE REPLY containing the source route to the source. When forwarding a ROUTE REPLY, a node stores the route starting from itself to the destination. Upon receiving the ROUTE REPLY, the source caches the source route.

In Route Maintenance, a node forwarding a packet is responsible for confirming that the packet has been received by the next hop. If no acknowledgement is received after the maximum number of retransmissions, this node will send a ROUTE ERROR to the source, indicating the broken link. Each node receiving a ROUTE ERROR removes from its cache the routes containing the broken link.

Besides Route Maintenance, DSR uses two mechanisms to remove stale routes. First, a source node piggybacks the last known broken link information on the next ROUTE REQUEST (called GRATUITOUS ROUTE ERROR) to clean more nodes. Second, it relies on heuristics: a small cache size with FIFO replacement policy for path caches, and adaptive timeout mechanisms for link caches [5], where the timeout of a link is predicted based on observed link usages and breakages.

Among the optimizations proposed in [11], we will not use GRATUITOUS ROUTE REPLIES and tapping. Both of them rely on *promiscuous mode*, which disables the network interface's address filtering function and thus causes the routing protocol to receive all packets overheard by the interface.

3 The Distributed Cache Update Algorithm

In this section, we first discuss the adverse effects of stale routes. We then present an overview of our caching strategy. After describing the definition of a cache table, we use examples to explain two algorithms used for maintaining the information for cache updates. Finally, we present the cache update algorithm with examples and a concise description.

3.1 The Impact of Stale Routes

The adverse effects of stale routes can be summarized as:

- Causing packet losses, increasing packet delivery latency and routing overhead. These effects will become more significant as mobility, traffic load, or network size increases, because more routes will become stale and/or stale routes will affect more traffic sources.
- Degrading TCP performance. Since TCP cannot distinguish between packet losses caused by route failures from those caused by congestion, it will falsely invoke congestion control mechanisms, resulting in the reduction in throughput.
- Wasting the energy of source nodes and intermediate nodes. If stale routes are not removed quickly, TCP will retransmit lost packets still using stale routes.

We refer to a route in a node's cache as ***pre-active***, if it has not been used; ***active***, if it is being used; ***post-active***, if it was used before, but now is not. It is not necessary to detect whether a route is active or post-active, but these terms help understand the cache staleness issue. It is easy to detect a stale route if it is active, but stale pre-active and post-active routes will not be detected until they are used. Therefore, they are the major sources of cache staleness.

3.2 Overview

Fast cache updating is important for reducing the adverse effects of stale routes. It is also necessary to constrain cache update notifications to the nodes that have cached a broken link in order to avoid the overhead of notifying other nodes. Thus, our goal is this: when a node detects a link failure, all reachable nodes whose caches contain the broken link will be notified about the link failure.

To achieve this goal, we make use of the information obtained from route discoveries and data transmission. We define a cache table to gather and maintain the information necessary for cache updates. Each node maintains two types of information for each route in its cache table: how well the routing information is synchronized among nodes on a route, and which neighbor node outside a route has learned which link of the route. By keeping such local information, a node knows which neighbor node needs to be notified about a broken link. A node receiving a cache update notification uses the local information kept in its cache table to determine and notify the neighbor nodes that have cached the broken link. Thus, a broken link information will be quickly propagated to all reachable nodes whose caches contain that link.

3.3 The Definition of a Cache Table

A cache table has no capacity limit and thus its size changes as needed. Each entry of a cache table contains four fields:

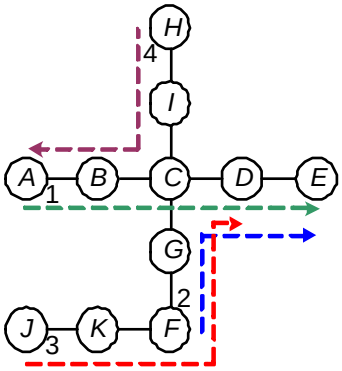

Figure 1: An Example of a Network with Four Flows

- ***Route***: It is a route a node learns. A node first stores the links from itself to a destination from a ROUTE REPLY and later completes the sub-route stored before by adding upstream links from the first data packet. If no *route* in the table is a sub-route of the source route, it stores the complete source route from the data packet.
- ***SourceDestination***: It is the source and destination pair.
- ***DataPackets***: It records whether a node has forwarded 0, 1, or 2 data packets using the *route*. This field indicates to what extent the routing information is synchronized among nodes on the route. It is 0 when the node stores downstream links from a ROUTE REPLY; it is incremented to 1 when the node forwards the first data packet; and it is incremented to 2 when the node forwards the second data packet.
- ***ReplyRecord***: When the node informs a neighbor of a sub-route through a ROUTE REPLY, it records the neighbor and the links used as an entry. If some entry contains a broken link, a node knows which neighbor it needs to notify about the link failure. This field has no capacity limit and thus its size changes as needed.

3.4 Information Collection and Maintenance for Cache Updates

During route discoveries and data transmission, we use two algorithms to collect and maintain the information necessary for cache updates: *addRoute* and *findRoute*.

We use a network shown in Fig. 1 in our examples. We will use S-D for SourceDestination, dP for DataPackets, and replyRec for ReplyRecord in the headers of tables describing the content of caches. Initially, there are no flows and all nodes' caches are empty.

Node *A* initiates a route discovery to *E* and a ROUTE REPLY is sent from *E* to *A*. Upon receiving a ROUTE REPLY, each intermediate node creates a new entry in its cache (*addRoute*: 6–11 in the Appendix). For instance, node *C* creates an entry consisting of four fields: (1) a route containing downstream links: *CDE*; (2) the source and destination pair: *AE*; (3) the number of data packets it received from the source node *A*: 0; (4) which neighbor will learn which route: *B* will learn *CDE*. This is described as:

C :

Route	S-D	dP	replyRec
CDE	*A E*	0	*B* ← *CDE*

When *A* receives the ROUTE REPLY, it adds to its cache (*addRoute*: 1–5):

A :

Route	S-D	dP
ABCDE	*A E*	0

When node *A* uses this route to send the first data packet, its entry is updated as (*findRoute*: 9–10):

A :

Route	S-D	dP
ABCDE	*A E*	1

Each node receiving the data packet updates its cache entry. For instance, node *C* increments *DataPackets* to 1, replaces *CDE* by *ABCDE* (*addRoute*: 20–24), and removes the entry in the *ReplyRecord* field (*addRoute*: 25–27) because the complete route indicates *A* and *B* have cached all links of the route. Thus, *C*'s cache is:

C :

Route	S-D	dP
ABCDE	*A E*	1

When *E* receives the first data packet, it creates a new entry (*addRoute*: 19) and its cache is the same as that of *C*.

When *C* receives the second data packet, it increments dP to 2 (*addRoute*: 14–17).

Now, assume that *C* receives a ROUTE REQUEST from *G* with source *F* and destination *D*. Before sending a ROUTE REPLY to *G*, *C* will extend its cacheEntry to (*findRoute*: 1–8):

C :

Route	S-D	dP	replyRec
ABCDE	*A E*	2	*G* ← *CDE*

Node *G* creates a cacheEntry(*addRoute*: 6–11) before sending a ROUTE REPLY to *F*:

G :

Route	S-D	dP	replyRec
GCDE	*F E*	0	*F* ← *GCDE*

When *F* gets the ROUTE REPLY, it inserts into its cache:

F :

Route	S-D	dP
FGCDE	*F E*	0

Now, assume *C* receives a ROUTE REQUEST from *I* with source *H* and destination *A*. *C* extends its cache entry to (*findRoute*: 1–8):

C :

Route	S-D	dP	replyRec	replyRec
ABCDE	*A E*	2	*G* ← *CDE*	*I* ← *CBA*

If a node does not cache a source route and no sub-route can be completed, it creates a new entry and add the source route to its cache (*addRoute: 28*), since the node knows that all the upstream nodes have stored the downstream links. For example, assume flow 2 starts. When it reaches *D*, *D* inserts the second entry into its cache:

	Route	S-D	dP
D :	$FGCDE$	$F\ E$	1

Whenever a route is completed or a new source route is added, a node always checks the *ReplyRecord* to see whether the concatenation of two fields in an entry is a sub-route of the source route (*addRoute: 25–27*). If so, it removes that entry.

Later, assume that after transmitting at least two packets, F receives a ROUTE REQUEST from K with source J and destination D. Before transmitting ROUTE REPLY to K, F extends its cache entry:

	Route	S-D	dP	replyRec
F :	$FGCDE$	$F\ E$	2	$K \leftarrow FGCD$

Each of J and K will save in its cache the route from itself to D.

3.5 The Distributed Adaptive Cache Update Algorithm

In this section, we first show several examples for the cache update algorithm. We then present a concise description of the algorithm. We show the pseudo code of the algorithm in the appendix.

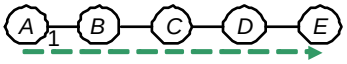

Figure 2: Scenario 1

3.5.1 Examples

Scenario 1 Here we focus on *DataPackets* and *ReplyRecord* in a simple case, as shown in Fig. 2. Assume that A has initiated a route discovery to E. Before any data packet from flow 1 with route $r_1 = ABCDE$ reaches C, C discovers that the link CD is broken. In its cache it finds:

	Route	S-D	dP	replyRec
C :	CDE	$A\ E$	0	$B \leftarrow CDE$

Since *DataPackets* is 0, C knows that CDE is a pre-active route, and therefore no downstream nodes need to be notified, since they did not cache the broken link when forwarding a ROUTE REPLY. Node C needs to check whether some neighbor nodes have cached the broken link (*cacheUpdate: 44–48*). It notifies its neighbor B and removes the entry from its cache. B notifies A and cleans its cache.

For another case, assume that A started transmissions for flow 1. While attempting to transmit a data packet, C detects that CD is broken. C's cache is:

	Route	S-D	dP
C :	$ABCDE$	$A\ E$	d

Since this packet is a data packet, d is 1 or 2. Thus, upstream nodes need to be notified about the broken link. C adds its upstream neighbor B to a list consisting of nodes to be notified (*cacheUpdate: 7–10*). If $d = 1$ and the route being examined is the same as the source route in the packet, which means that the packet is the first data packet, then no downstream node needs to be notified.

If $d = 2$, then the packet is at least the second data packet arriving at C. (If $d = 1$ and the route being examined is different from the source route in the packet, then the route being examined has been synchronized by its first data packet. We handle this case the same as the case where $d = 2$.) Therefore, at least one data packet has reached D and E and thus they have cached the route $ABCDE$. C searches in its cache for a shortest route to reach one of the downstream nodes (*cacheUpdate:11–18*). Assume that it finds a route to E. So C notifies E. As the table entry does not contain any *ReplyRecord*, no neighbor has learned a sub-route from C. C then removes the table entry. E in turn notifies D (*cacheUpdate: 39–40*). If no route to either D or E is found, they will not be notified at this time.

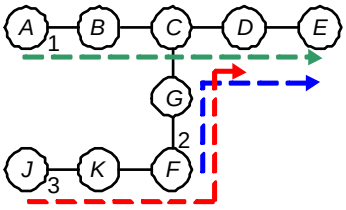

Figure 3: Scenario 2

Scenario 2 Here we focus on how *ReplyRecord* are handled in a slightly complex case, as shown in Fig. 3. As before, first A discovers $r_1 = ABCDE$, then F discovers $r_2 = FGCDE$ for flow 2 after receiving the ROUTE REPLY sent by C. Finally, J discovers $r_3 = JFKD$ for flow 3 after receiving the ROUTE REPLY sent by F. r_1 is an active route and both r_2 and r_3 are pre-active routes. When transmitting packets for flow 1, C discovers that CD is broken.

	Route	S-D	dP	replyRec
C :	$ABCDE$	$A\ E$	2	$G \leftarrow CDE$
G :	$GCDE$	$F\ E$	0	$F \leftarrow GCDE$
F :	$FGCDE$	$F\ E$	0	$K \leftarrow FGCD$
K :	$KFGCD$	$J\ D$	0	$J \leftarrow KFGCD$
J :	$JKFGCD$	$J\ D$	0	

C handles the upstream and the downstream nodes of r_1 the same as the second case in scenario 1. It also finds that it has notified a neighbor G of a route containing CD. So C notifies G that CD is broken and deletes r_1 from its cache (*cacheUpdate: 44–48*). After G examines its cache, it notifies F and cleans its cache. F notifies K and cleans its cache. K notifies J and cleans its cache. Finally, J cleans its cache.

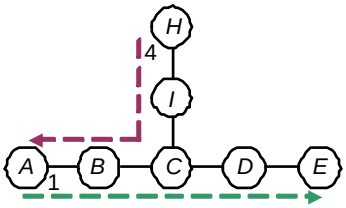

Figure 4: Scenario 3

Scenario 3 Here we focus on how to handle a broken link through which two (or more) flows in opposite directions are flowing, as shown in Fig. 4. Assume that *only* flows 1 and 4 have started. Here $r_1 = ABCDE$ and $r_4 = HICBA$. While transmitting a packet for flow 4, *C* detects that *CB* is broken. The cache of C is:

	Route	S-D	dP
C :	*ABCDE*	*A E*	2
C :	*HICBA*	*H A*	2

Nodes that need to be notified about the broken link are: *H*, *I* (upstream in r_4); *A*, *B* (upstream in r_1 and downstream in r_4); and *D*, *E* (downstream in r_1). Node *C* attempts to find a shortest path to reach *A* or *B*; it also notifies *D* and *I* about the broken link (*cacheUpdate:19–28*).

A node receiving a notification determines a list of neighbor nodes it is responsible for notifying. For example, node *I* knows that it needs to notify its upstream node *H* (*cacheUpdate:31-32*), and *D* knows that it needs to notify its downstream node, here *E* (*cacheUpdate:33–34*).

3.5.2 Concise Description

A node learns a broken link either by detecting it itself or through a cache update notification from another node. In either case, the cache update algorithm is started reactively to examine each entry of the cache table. For each route containing a broken link, the algorithm determines a set of neighborhood nodes it needs to notify about the broken link, its upstream and/or downstream neighbors as well as other neighbors outside the route. Finally, the algorithm produces a *notification list*, which includes nodes to be notified about the broken link and routes to reach those nodes. A node sends cache update notifications through ROUTE ERRORS.

In each route containing a broken link, the link is in the *forward direction* if the flow using that route crosses the link in the same direction as the flow that has detected the breakage; otherwise it is in the *backward direction*. For these two types of links, the operation of the algorithm is symmetric.

When a node detects a link failure, it examines each entry of its cache. If a route contains the broken link in the *forward* direction (*cacheUpdate: 8–18*), then the algorithm does the following steps:

- If *DataPackets* is 2 or 1, then upstream nodes (if any) need to be notified, but only the upstream neighbor is added to the notification list.
- If *DataPackets* is 2, or 1 and the route being examined is different from the source route in the packet, then downstream nodes need to be notified. The algorithm tries to find a shortest path to reach one of the downstream nodes. For both cases, the first data packet has traversed that route and thus all downstream nodes have cached the broken link.
- If *DataPackets* is 0, there is no upstream node and no downstream nodes need to be notified.

If a route contains the broken link in the *backward direction* (*cacheUpdate: 19–28*), which implies that the current node is the first downstream node, then its downstream neighbor is added to the list. Also, the algorithm tries to find a shortest path to reach one of the upstream nodes.

If a node learns a broken link through a notification from another node, it knows which node it needs to notify about the link failure based on its position in a route containing the broken link (*cacheUpdate: 29–43*).

If any entry in the *ReplyRecord* field contains a broken link, then the node adds the neighbor that got the sub-route containing the link to the list. This field needs to be checked whether dataPacket is 0, 1, or 2 and whether the link is in the forward or backward direction.

In this way, all reachable nodes that have cached the broken link will learn the link failure and update their caches. If a node cannot find a route to reach any downstream node, these nodes will learn the broken link when the first data flow detects the broken link in a reverse direction. Then nodes that got routes containing the broken link from these downstream nodes will also know the link failure and update their caches. Thus, proactive cache updating is triggered fully on-demand.

3.6 Some Implementation Decisions

In order to reduce the duplicate error notifications to a node, we attach a *reference list* to each ROUTE ERROR. The node detecting a link failure becomes the root node. It initializes the reference list to be its notification list; each child only sends cache update notifications to nodes not on this list and updates this list by adding nodes on its own notification list.

When using the cache update algorithm, we also use a small list of broken links like a negative cache to prevent a node from being re-polluted by the in-flight stale routes. Its size is set to 5 and timeout is set to 2 s for all scenarios used in simulations. This component is not part of the algorithm, which does not use any ad hoc parameter. This list can be replaced by a non-ad-hoc technique proposed by Hu and Johnson [6].

4 Performance Evaluation

4.1 Evaluation Methodology

We used detailed simulations to evaluate the performance of our algorithm. We compared DSR with our algorithm (called DSR-Adaptive) with DSR with path caches and with *Link-Maxlife* [5], which was shown to outperform other adaptive timeout algorithms. We evaluated the three caching strategies under non-promiscuous mode. We did not use GRATUITOUS ROUTE REPLIES and tapping since they rely on promiscuous mode. For DSR-Adaptive, we did not use GRATUITOUS ROUTE ERROR, since we wanted to evaluate this algorithm as the only mechanism to remove stale routes.

We used *ns-2* [2] network simulator together with Monarch Project's wireless and mobile extensions [1, 16]. The mobility model is *random waypoint model* [1] in a rectangular field. In this model, a node starts in a random position, picks a random destination, moves to it at a randomly chosen speed, and

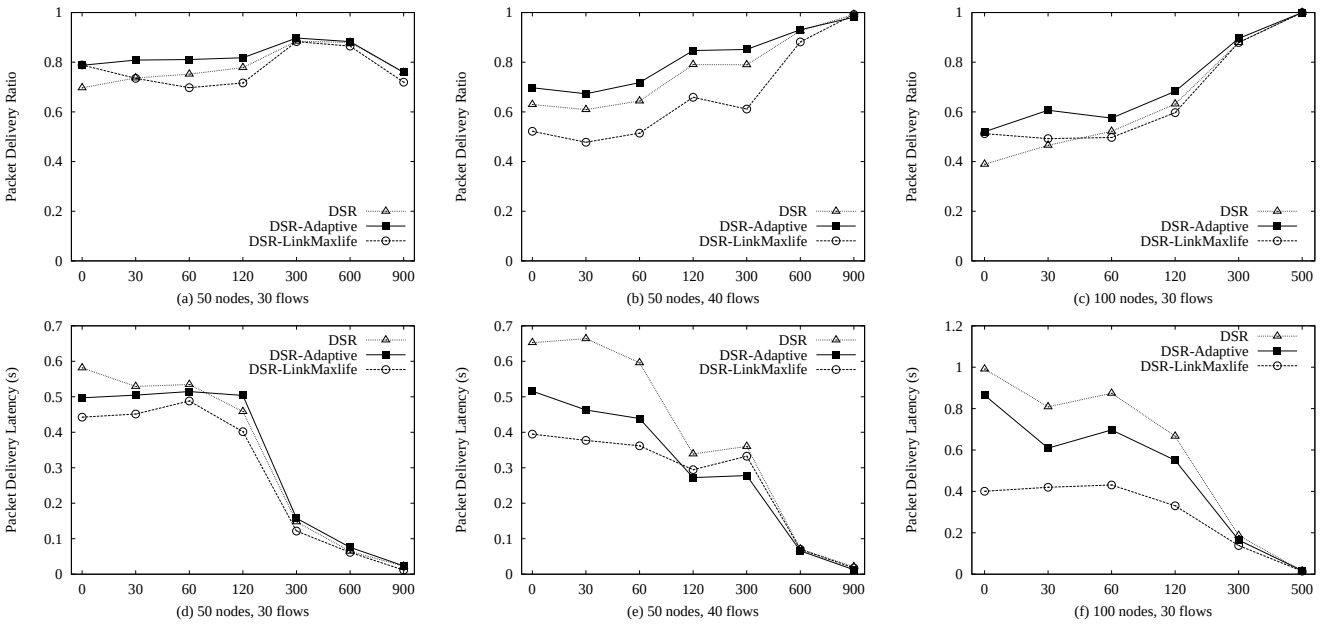

Figure 5: Packet Delivery Ratio and Packet Delivery Latency vs. Mobility (Pause Time (s))

pauses for a specified pause time. The speed was randomly chosen from 10 ± 1 m/s. Three field configurations were used:

- 1500m $\times$ 1000m field with 50 nodes
- 1500m $\times$ 500m field with 50 nodes
- 2200m $\times$ 600m field with 100 nodes

We used the first two configurations to evaluate how our algorithm performs under different node densities, and used the third one to see how it scales with network size. The pause times for the first two configurations are: 0, 30, 60, 120, 300, 600, and 900 s; and 0, 30, 60, 120, 300, 500 s for the third one. The simulation ran for 900 s for the first two and 500s for the third one as done in [12]. The communication model is constant bit rate (CBR) traffic with four packets per second and packet size of 64 bytes in order to factor out the effect of congestion. The traffic loads for the three field configurations are 30 flows, 40 flows, and 30 flows. Each data point represents an average of 10 runs different randomly generated scenarios. We use 50n-30f for 50 node and 30 flows scenarios, etc.

We used four metrics:

- ***Packet Delivery Ratio***: the fraction of data packets sent by the source that are received by the destination.
- ***Packet Delivery Latency***: the average time taken by a data packet to travel from the source to the destination.
- ***Normalized Routing Overhead***: the ratio of the total number of routing packets transmitted to the total number of data packets received. For the cache update algorithm, routing packets include ROUTE ERRORS used for cache updates.
- ***Good Cache Replies Received***: the percentage of ROUTE REPLIES without broken links received by the source that originated from caches.

4.2 Simulation Results

4.2.1 Packet Delivery Ratio

Packet delivery ratio is an important measure of the performance of a routing protocol. Figs. 5 (a)–(c) show the results for this metric.

DSR-Adaptive outperforms DSR for all mobility scenarios, obtaining improvement of 13% for 50n-30f, 11% for 50n-40f, and 34% for 100n-30f at pause time 0 s. Such significant improvement demonstrates that our algorithm quickly removes stale routes. Since DSR with path caches has delayed awareness of mobility, more packets are dropped at intermediate nodes due to stale routes. As we can see, the improvement increases as mobility increases, showing the efficient adaptation of our algorithm to topology changes. The improvement is much higher for 100 node scenarios than for 50 node ones. Fast cache updating is important for large networks, because more nodes will cache stale routes as network size increases. These results show that proactive cache updating is more efficient than FIFO in invalidating stale information.

DSR-Adaptive also outperforms *Link-Maxlife*, obtaining the maximum improvement of 16% for 50n-30f, 35% for 50n-40f, and 20% for 100n-30f. For 50n-40f, the highest traffic load and higher node density scenarios, DSR-Adaptive significantly outperforms *Link-Maxlife*, achieving the highest improvement. This is because as traffic load or node density increases, the adverse effects of stale routes become more significant: as traffic load increases, more traffic sources will use stale routes, resulting in more packet losses; as node density

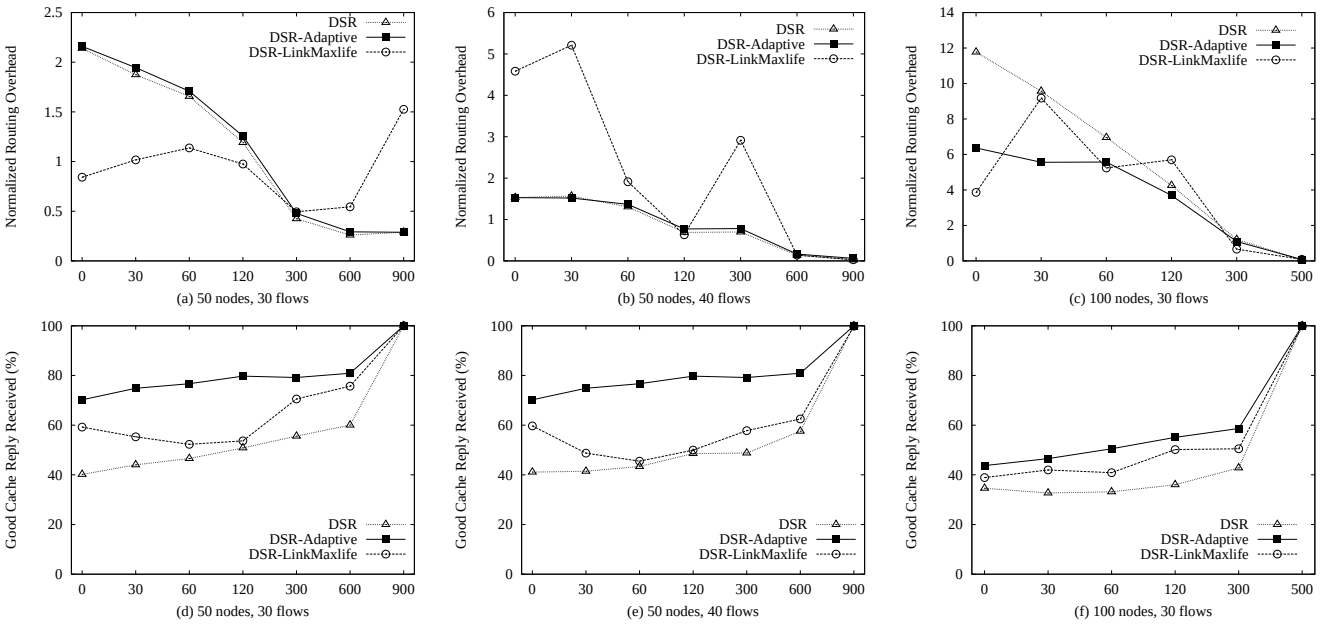

Figure 6: Normalized Routing Overhead and Good Cache Replies Received vs. Mobility (Pause Time (s))

increases, more nodes will cache stale routes. These results show that *Link-Maxlife* cannot accurately predict timeouts.

Link-Maxlife performs better than DSR for high mobility scenarios, since it is able to aggressively expire links when links break more frequently under high mobility. This is consistent with the statistics of good cache replies received, as shown in Fig. 6 (d) and (f), where the cache performance of the former is better than that of the latter under high mobility. However, *Link-Maxlife* performs worse for other mobility scenarios, especially under high traffic load. This is because when mobility is not high, path caches with FIFO remove stale routes faster than predicting timeouts, since high traffic load speeds up cache turnover.

For DSR and *Link-Maxlife*, there is an inconsistency between good cache replies received and packet delivery ratio at pause time 0 s for 50n-40f and when mobility is not high for other scenarios. We attribute the following reason to this observation. Although more cache replies received are good in *Link-Maxlife*, FIFO replacement policy has evicted more stale routes when cached routes are picked up to be used.

4.2.2 Packet Delivery Latency

Figs. 6 (d)–(f) show the results for packet delivery latency. Compared with DSR, DSR-Adaptive achieves the reduction in latency by up to 15% at pause time 0 s for 50n-30f, up to 30% at pause time 30 s for 50n-40f, and up to 25% at pause time 30 s for 100n-30f. Such significant reduction in latency results from DSR-Adaptive's fast cache updating. Since detecting link failures through several retransmissions is the dominant factor of the delay experienced by a packet, removing stale routes earlier from the caches of reachable nodes reduces link failure detections by multiple data flows, and thus reduces the overall packet delivery latency. In addition, when a data packet is salvaged and routed differently, the new route is generally longer than the original one [5], causing significant increase in latency.

As we expected, the reduction in latency compared with DSR increases as mobility, network size, or traffic load increases, a desirable adaptive property of our algorithm. As network characteristics become more challenging, the advantages of proactive cache updating become more significant.

DSR-Adaptive performs worse than *Link-Maxlife* in this metric. Since *Link-Maxlife* store more routing information in the topology graph, it performs much fewer route discoveries than DSR-Adaptive, resulting in lower latency.

4.2.3 Normalized Routing Overhead

Figs. 6 (a)–(c) show the results for the normalized routing overhead. DSR-Adaptive has almost the same overhead as DSR for 50n-30f and 50n-40f. Although proactive cache updating introduces overhead for cache updates, it reduces ROUTE ERRORS due to stale routes. DSR-Adaptive has a slightly higher packet overhead than DSR, but has almost the same the normalized routing overhead as DSR due to increased packet delivery ratio. DSR-Adaptive provides such benefit through efficient adaptation to mobility, whereas over-aggressive caching strategies introduce overhead for route rediscoveries and over-conservative caching strategies increase overhead caused by the use of stale routes.

As network size increases, DSR-Adaptive achieves a large reduction in overhead by about 50% of DSR. We observe from simulation data that DSR initiates more route discoveries than DSR-Adaptive, resulting in higher overhead. The higher number of route discoveries is due to the small cache size used in

DSR, which is not large enough to hold all useful routes as network size increases.

Link-Maxlife has an unstable curve for this metric for both 50 and 100 node networks. For example, for 50n-30f, as shown in Fig. 6 (a), *Link-Maxlife* has significantly lower overhead than the other two under high mobility due to fewer route discoveries, but has much higher overhead under static scenarios. It is because this mechanism sometimes over-aggressively expires links that are still valid. Simulation data shows that *Link-Maxlife* initiates a large number of route discoveries for one scenario at pause time 900 s. This is also the reason for other high overhead cases in 50n-40f and 100n-30f.

4.2.4 Good Cache Replies Received

The percentage of good cache replies received is an important metric for evaluating cache performance, which has been used in previous studies [11, 12]. As shown in Fig. 6 (d)–(f), DSR-Adaptive achieves very significant improvement in cache correctness compared with both DSR and *Link-Maxlife*. For example, compared with DSR, it obtains the improvement of 51% for 50n-30f at pause time 0 s, 75% for 50n-40f, and 51% for 100n-30f at pause time 60 s. Compared with *Link-Maxlife*, it provides the maximum improvement of 48% for 50n-30f at pause time 120 s, 67% for 50n-40f at pause time 60 s, and 25% for 100n-30f at pause time 60 s. To the best of our knowledge, these are the most significant results with respect to improving cache correctness of DSR. The worse cache performance of *Link-Maxlife* further shows that heuristics with ad hoc parameters cannot accurately predict timeouts.

5 Related Work

DSR's cache performance was first analyzed by Maltz et al. [11]. Hollan and Vaidya observed [4] that stale routes seriously affect TCP performance. Perkins et al. [15] pointed out the impact of stale routes on DSR's performance.

Hu and Johnson [5] studied design choices for cache structure, cache capacity, and cache timeout. They proposed the link cache structure and several adaptive link timeout algorithms. Marina and Das [12] proposed wider error notification and timer-based route expiry. Under wider error notification, a node receiving a ROUTE ERROR will rebroadcast it, but only if there is a cached route containing the broken link *and* that route was used before to transmit data packets by that node. Thus broken links which are frequently propagated in ROUTE REPLIES and cached by nodes for future use will not be removed. Moreover, some nodes that have cached broken links may not receive notifications, since broadcast is unreliable. Under timer-based expiry, the average timeout is assigned to all the routes based on some broken routes. While this works well when routes break uniformly, mobility may not be uniform in time or space. Lou and Fang [10] proposed an adaptive link timeout mechanism in which the lifetime of a link is adjusted based on the real link lifetime statistics. Hu and Johnson [6] proposed to use epoch numbers for preventing a node from re-learning a stale route that it previously found out was broken.

6 Conclusions

We have presented a novel solution to the cache staleness issue of DSR. We define a cache table to collect and maintain the information necessary for cache updates. When a node detects a link failure, our distributed cache update algorithm proactively notifies all the reachable nodes whose caches contain the broken link to update their caches, using local information kept by each node and relying on cooperative update propagation. The algorithm does not use ad hoc parameters, thus making route caches fully adaptive to topology changes.

Through detailed simulations, we show that the algorithm significantly improves packet delivery ratio and reduces packet delivery latency compared with DSR with path caches. It also considerably outperforms *Link-Maxlife* in packet delivery ratio. Our results lead to the following conclusions:

- Due to unpredictable topology changes, heuristics cannot accurately predict timeouts. Proactive cache updating is more efficient than adaptive timeout mechanisms in invalidating stale routing information.
- The effectiveness of predetermined choices of ad hoc parameters in caching strategies depends on scenarios. It is important to make route caches dynamically adapt to changing network characteristics.

Acknowledgements

We thank the developers of *ns-2*, David Johnson and his Monarch group for making available their wireless extensions to *ns-2*, and Yih-Chun Hu and David Johnson for making their link cache code available. This work was supported in part by DARPA under grant N66001-01-1-8929.

References

[1] J. Broch, D. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proc. 4th ACM MobiCom*, pp. 85–97, 1998.

[2] K. Fall and K. Varadhan, Eds. *ns* notes and documentation. The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, 1997.

[3] Z. Haas, M. Pearlman, and P. Samar. The Zone Routing Protocol (ZRP) for ad hoc networks, IETF Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-manet-zone-zrp-04.txt, July 2002.

[4] G. Holland and N. Vaidya. Analysis of TCP performance over mobile ad hoc networks. In *Proc. 5th ACM MobiCom*, pp. 219–230, 1999.

[5] Y.-C. Hu and D. Johnson. Caching strategies in on-demand routing protocols for wireless ad hoc networks. In *Proc. 6th ACM MobiCom*, pp. 231–242, 2000.

An extended version of this paper appears in Proceedings of IEEE INFOCOM 2005.

[6] Y.-C. Hu and D. Johnson. Ensuring cache freshness in on-demand ad hoc network routing protocols. In *Proc. 2nd POMC*, pp. 25–30, 2002.

[7] D. Johnson and D. Maltz. *Dynamic Source Routing in ad hoc wireless networks*, Chapter 5, pp. 153–181. Kluwer Academic Publishers, 1996.

[8] D. Johnson, D. Maltz, Y.-C. Hu, and J. Jetcheva. The Dynamic Source Routing for mobile ad hoc networks, IETF Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-07.txt, February 2002.

[9] Y.-B. Ko and N. Vaidya. Location-Aided Routing (LAR) in mobile ad hoc networks. *Wireless Networks*, 6(4):307–321, 2000.

[10] W. Lou and Y. Fang. Predictive caching strategy for on-demand routing protocols in wireless ad hoc networks. *Wireless Networks*, 8(6):671–679, 2002.

[11] D. Maltz, J. Brooch, J. Jetcheva, and D. Johnson. The effects of on-demand behavior in routing protocols for multi-hop wireless ad hoc networks. *IEEE J. on Selected Areas in Communication*, 17(8):1439–1453, 1999.

[12] M. Marina and S. Das. Performance of routing caching strategies in Dynamic Source Routing. In *Proc. 2nd WNMC* (in conjunction with ICDCS), pp. 425–432, 2001.

[13] V. Park and M. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proc. 16th IEEE INFOCOM*, pp. 1405–1413, 1997.

[14] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proc. 2nd WMCSA*, pp. 90–100, 1999.

[15] C. Perkins, E. Royer, S. Das, and M. Marina. Performance comparison of two on-demand routing protocols for ad hoc networks. *IEEE Personal Communications*, 8(1):16–28, 2001.

[16] The Monarch Project. Rice monarch project: Mobile networking architectures. Project page: http://www.monarch.cs.rice.edu/ .

7 Appendix: Pseudo Code

Procedures:

```
int Index(PATH route, ID id):
compute the index of node id in route route;
PATH subPath(PATH route, ID startID,ID endID):
compute from route a subpath starting with index being startID and ending with index endID;
boolean replyPairExist(vector <ReplyPair*>, replyRecord, ReplyPair reply_pair):
if reply_pair exists in replyRecord return true , else return false;
cacheEntry getFromCacheTable(PATH route):
If some entry e in cacheTable satisfies e.route = route return e, else return null;
boolean isFirstNode(PATH route, ID id):
if id = route[0] (first node) then return true, else return false;
boolean isLastNode(PATH route, ID id):
if id = route[route.length − 1] (last node) then return true, else return false;
```

Algorithm: *findRoute*

```
Input:  ID dest, PACKET p, boolean respond_to_RREQ, boolean used_for_salvaging
Output:  PATH route
 1 e_0 := ∅;
 2 for each entry e ∈ cacheTable do
 3   if dest ∈ e.route then
 4     temp := subPath(e.route, Index(e.route, netID), Index(e.route, dest))
 5     if route = ∅ or |temp| < |route| then route := temp; e_0 := e
 6 if e_0 = ∅ then exit;
 7 if respond_to_RREQ then reply_pair := (p.srcRoute[p.srcRoute.length − 1], route)
 8   if not replyPairExist(e_0.replyRecord, reply_pair) then e_0.replyRecord := e_0.replyRecord ∪ {reply_pair}
 9 elseif not used_for_salvaging then
10   if route = e_0.route and e_0.DP ≠ 2 then e_0.DP := e_0.DP + 1
11   else
12     cacheTable := cacheTable ∪ {(route, (netID, dest), 1, ∅)}
```

Algorithm: *addRoute*

```
Input:  PACKET p
 1 if p is a RREP packet then
 2   if netID = p.dest then
 3     e := getFromCacheTable(p.srcRoute);
 4     if e = null then
 5       cacheTable := cacheTable ∪ {(p.srcRoute, (p.srcRoute[0], p.srcRoute[p.srcRoute.length − 1]), 0, ∅)}
 6   else
 7     newRoute := subPath(p.srcRoute, Index(p.srcRoute, netID), Index(p.srcRoute, p.srcRoute[p.srcRoute.length − 1]));
 8     reply_pair := (p.srcRoute[Index(p.srcRoute, netID) − 1], newRoute);
 9     e := getFromCacheTable(newRoute);
10     if e = null then
11       cacheTable := cacheTable ∪ {(newRoute, (p.srcRoute[0], p.srcRoute[p.srcRoute.length − 1]), 0, reply_pair)}
12     else
13       if not replyPairExist(e.replyRecord, reply_pair) then e.replyRecord := e.replyRecord ∪ {reply_pair}
14 elseif p is a DATA packet then
15   e := getFromCacheTable(p.srcRoute);
16   if e ≠ null then
17     if e.DP = 1 then e.DP := 2
18   else
19     if netID = p.dest then cacheTable := cacheTable ∪ {(p.srcRoute, (p.src, p.dest), 1, ∅)}
20     else
21       for each entry e ∈ cacheTable do
22         if e.srcDest.src = p.src and e.srcDest.dest = p.dest and p.src = p.route[0] then
23           temp := subPath(p.srcRoute, Index(p.srcRoute, netID), Index(p.srcRoute, p.srcRoute[p.srcRoute.length − 1]));
24           if temp = e.route and e.DP = 0 then e.route := p.srcRoute; e.DP := 1; is_completed := TRUE
25       for each entry r ∈ e.replyRecord do
26           temp := subPath(p.srcRoute, Index(p.srcRoute, netID) − 1, Index(p.srcRoute, p.srcRoute[p.srcRoute.length − 1]));
27           if (r.nodeNotified||r.subrouteSent) = temp then e.replyRecord := e.replyRecord \ {r};
28       if not is_completed and p.src = p.route[0] then cacheTable := cacheTable ∪ {(p.srcRoute, (p.src, p.dest), 1, ∅)}
```

Algorithm: *cacheUpdate*

```
Input:  ID from, ID to, PACKET p, boolean detect_by_me, boolean continue_to_notify
/* If p is a ROUTE ERROR and p.src = from and netID = tellID, then continue_to_notify is set TRUE. */
Output:  vector <NotifyEntry*> notifyList
 1 for each entry e ∈ cacheTable do
 2  if link (from,to) ∈ e.route then has_broken_link := TRUE; direction := forward
 3  elseif link (to,from) ∈ e.route then has_broken_link := TRUE; direction := backward
 4  else has_broken_link := FALSE;
 5  if has_broken_link then
 6   position := Index(e.route,from);
 7   if detect_by_me then
 8    if direction = forward then
 9     if (e.DP = 1 or e.DP = 2) and (not isFirstNode(e.route,netID)) then
10      notifyList := notifyList ∪ {(e.route[position − 1],(netID||e.route[position − 1]))}
11     if e.DP = 2 or (e.DP = 1 and (not (p is a DATA packet and (p.srcRoute = e.route))) )then
12     routeToUse = ∅;
13      for each node n ∈ {e.route[position + 1],...,e.route[e.route.length − 1]} do
14       Try to find a shortest route in cacheTable from netID to n
15       if such route is found then
16        foundRoute := the found route;
17        if routeToUse = ∅ or |foundRoute| < |routeToUse| then routeToUse := foundRoute; tellID := n
18      if routeToUse ≠ ∅ then notifyList := notifyList ∪ {(tellID,routeToUse)}
19    elseif direction = backward then
20     if not isLastNode(e.route,netID) then
21      notifyList := notifyList ∪ {(e.route[position + 1],(netID||e.route[position + 1]))}
22     routeToUse = ∅;
23     for each node n ∈ {e.route[position − 1],...,e.route[0]} do
24      Try to find a shortest route in cacheTable from netID to n
25      if such route is found then
26       foundRoute := the found route;
27       if routeToUse = ∅ or |foundRoute| < |routeToUse| thenrouteToUse := foundRoute; tellID := n
28     if routeToUse ≠ ∅ then notifyList := notifyList ∪ {(tellID,routeToUse)}
29   else /* The current node receives a notification from another node.  */
30    index := Index(e.route,netID);
31    if direction = forward and index < position and (not isFirstNode(e.route,netID)) then
32     notifyList := notifyList ∪ {(e.route[index − 1],(netID||e.route[index − 1]))}
33    if direction = backward and index > position and (not isLastNode(e.route,netID)) then
34     notifyList := notifyList ∪ {(e.route[index + 1],(netID||e.route[index + 1]))}
35    if (e.DP = 1 or e.DP = 2) and ((direction = forward and index > position) or
       (direction = backward and index < position)) then
36     if continue_to_notify then
37      if (direction = forward and netID = to and (not isLastNode(e.route,netID))) or
         (direction = backward and isFirstNode(e.route,netID) and (not netID = to)) then
38       notifyList := notifyList ∪ {(e.route[index + 1],(netID||e.route[index + 1]))}
39      if (direction = forward and isLastNode(e.route,netID) and (not netID = to)) or
         (direction = backward and netID = to and (not isFirstNode(e.route,netID))) then
40       notifyList := notifyList ∪ {(e.route[index − 1],(netID||e.route[index − 1]))}
41     if not (netID = to or (direction = forward and isLastNode(e.route,netID)) or
         (direction = backward and isFirstNode(e.route,netID))) then
42      notifyList := notifyList ∪ {(e.route[index + 1],(netID||e.route[index + 1]))};
43      notifyList := notifyList ∪ {(e.route[index − 1],(netID||e.route[index − 1]))}
44   for each entry r ∈ e.replyRecord do
45    if link (from,to) ∈ e.replyRecord.subrouteSent or link (to,from) ∈ e.replyRecord.subrouteSent then
46     tellID := e.replyRecord.nodeNotified;
47     notifyList := notifyList ∪ {(tellID,(netID||tellID))};
48     e.replyRecord := e.replyRecord \ {r};
49   cacheTable := cacheTable \ {e};
50  else /* The route in the table entry does not contain a broken link.*/
51   for each entry r ∈ e.replyRecord do
52    if r.nodeNotified = to and netID = from then /* A broken link is detected by a ROUTE REPLY.*/
53     e.replyRecord := e.replyRecord \ {r};
54 for each entry n ∈ notifyList do
55  if (p is a ROUTE ERROR and n.tellID = p.src) or
     (n.routeToUse is a sub-route of another entry's routeToUse) or
     (entry m ∈ notifyList and n.tellID = m.tellID and |n.routeToUse| ≥ |m.routeToUse|) then
56   notifyList := notifyList \ {n};
57 return notifyList;
```