Pre Active Circulated Cache Updating using Dynamic Source Routing Protocol

Gayathri C
Department of Information Science and Engineering
East West Institute of Technology, Bangalore, Karnataka, India

Manjula G
Department of Information Science and Engineering
East West Institute of Technology, Bangalore, Karnataka, India

Abstract- **The TCP performance degrades significantly in mobile ad hoc networks due to the packet losses. Most of these packet losses result from the route failures due to network mobility. TCP assumes such losses occur because of congestion, thus invokes congestion control mechanisms such as decreasing congestion windows, raising timeout, etc, thus greatly reduce TCP throughput. However, after a link failure is detected, several packets will be dropped from the network interface queue. TCP will time out because of these packet losses, as well as for acknowledgement losses caused by route failures. There is no intimation information about the failure links to the node from its neighboring nodes; thereby the source node is unable to make the routing decisions at the time of data transfer. Prior work in dynamic source routing uses heuristics with ad hoc parameters to predict the lifetime of a link or a route. However, heuristics cannot accurately estimate timeouts because the topology changes are unpredictable. Prior researches have proposed to provide link failure feedback to TCP so that TCP can avoid responding to route failures as if congestion had occurred.**

A new cache structure called a cache table is defined to maintain the information necessary for cache updates. A distributed cache update algorithm that uses the local information kept by each node to notify all reachable nodes that have cached a broken link is proposed. The algorithm enables dynamic source routing to adapt quickly to topology changes and present a distributed cache update algorithm. Each node maintains in its cache table the information necessary for cache updates. The source node has the information regarding about the destination and the intermediate node links failure, so that it is useful from further packet loss and reduce the latency time while data transfer throughout the network. The newly proposed algorithm quickly removes stale routes irrespective of node mobility and traffic model.

Keywords – congestion window, stale route, distributed cache update, packet loss, cache table

I. INTRODUCTION

Routing protocols for ad hoc networks can be classified into two major types: *proactive* and *on-demand*. Proactive protocols attempt to maintain up-to-date routing information to all nodes by periodically disseminating topology updates throughout the network. In contrast, on demand protocols attempt to discover a route only when a route is needed. To reduce the overhead and the latency of initiating a route discovery for each packet, on-demand routing protocols use route Caches. Due to mobility, cached routes easily become stale. Using stale routes causes packet losses, and increases latency and overhead. In this paper, we investigate how to make on-demand routing Protocols adapt quickly to topology changes. This problem is important because such protocols use route caches to make routing decisions; it is challenging because topology changes are frequent.

To address the cache staleness issue in DSR (the Dynamic Source Routing protocol) prior work used adaptive timeout mechanisms. Such mechanisms use heuristics with ad hoc parameters to predict the lifetime of a link or a route. However, a predetermined choice of ad hoc parameters for certain scenarios may not work well for others, and scenarios in the real world are different from those used in simulations. Moreover, heuristics cannot accurately estimate timeouts because topology changes are unpredictable. As a result, either valid routes will be removed or stale routes will be kept in caches.

II. DESIGN

On-demand Route Maintenance results in delayed awareness of mobility, because a node is not notified when a cached route breaks until it uses the route to send packets. We classify a cached route into three types:

- ***Pre-active***, if a route has not been used;

- ***Active***, if a route is being used;
- ***Post-active***, if a route was used before but now is not.

When a node detects a link failure, our goal is to notify all reachable nodes that have cached that link to update their caches. To achieve this goal, the node detecting a link failure needs to know which nodes have cached the broken link and needs to notify such nodes efficiently. This goal is very challenging because of mobility and the fast propagation of routing information.

Our solution is to keep track of *topology propagation state* in a distributed manner.. In a cache table, a node not only stores routes but also maintain synchronization of nodes routing information and link state through a ROUTE REPLY. The two types of information are sufficient because *each* node knows for each cached link which neighbors have that link in their caches. Each entry in the cache table contains a field called *datapackets*. This field records whether a node has forwarded 0, 1, or 2 data packets. A node knows how well routing information is synchronized through the *first* data packet.

When forwarding a ROUTE REPLY, a node caches only the downstream links; thus, its downstream nodes did not cache the first downstream link through this ROUTE REPLY. When receiving the first data packet, the node knows that upstream nodes have cached all downstream links. The node adds the upstream links to the route consisting of the downstream links. Thus, when a downstream link is broken, the node knows which upstream node needs to be notified.

The node also sets *datapackets* to 1 before it forwards the first data packet to the next hop. If the node can successfully deliver this packet, it is highly likely that the downstream nodes will cache the first downstream link; otherwise, they will not cache the link through forwarding packets with this route. Thus, if *datapackets* in an entry is 1 and the route is the same as the source route in the packet encountering a link failure, downstream nodes did not cache the link. However, if *datapackets* is 1 and the route is different from the source route in the packet, downstream nodes cached the link when the first data packet traversed the route. If *datapackets* is 2, then downstream nodes also cached the link, whether the route is the same as the source route in the packet. Each entry in the cache table contains a field called *replyrecord*. This field records which neighbor learned which links through a ROUTE REPLY. Before forwarding a ROUTE REPLY, a node records the neighbor to which the ROUTE REPLY is sent and the downstream links as an entry. Thus, when an entry contains a broken link, the node will know which neighbor needs to be notified. The algorithm uses the information kept by each node to achieve distributed cache updating.

When a node detects a link failure while forwarding a packet, the algorithm checks the *datapackets* field of the cache entries containing the broken link:

(1) If it is 0, indicating that the node has not forwarded any data packet using the route, then no downstream nodes need to be notified because they did not cache the broken link.
(2) If it is 1 and the route being examined is the same as the source route in the packet, indicating that the packet is the first data packet, then no downstream nodes need to be notified but all upstream nodes do.
(3) If it is 1 and the route being examined is different from the source route in the packet, then both upstream and downstream nodes need to be notified, because the first data packet has traversed the route.
(4) If it is 2, then both upstream and downstream nodes need to be notified, because at least one data packet has traversed the route.

The algorithm notifies the closest upstream and/or downstream nodes and the neighbors that learned the broken link through ROUTE REPLIES. When a node receives a notification, the algorithm notifies selected neighbors: upstream and/or downstream neighbors, and other neighbors that have cached the broken link through ROUTE REPLIES. Thus, the broken link information will be quickly propagated to all reachable nodes that have that link in their caches.

- **Cache table structure**

We design a cache table that has no capacity limit. Without capacity limit allows DSR to store all discovered routes and thus reduces route discoveries. The cache size increases as new routes are discovered and decreases as stale routes are removed.
There are four fields in a cache table entry:

- **Route:** It stores the links starting from the current node to a destination or from a source to a destination.
- **Sourcedestination:** It is the source and destination pair.

- **Datapackets:** It records whether the current node has forwarded 0, 1, or 2 data packets. It is 0 initially, incremented to 1 when the node forwards the first data packet, and incremented to 2 when it forwards the second data packet.
- **Replyrecord:** This field may contain multiple entries and has no capacity limit.

A *replyrecord* entry has two fields: the neighbor to which a ROUTE REPLY is forwarded and the route starting from the current node to a destination. A *replyrecord* entry will be removed in two cases: when the second field contains a broken link, and when the concatenation of the two fields is a sub-route of the source route, which starts from the previous node in the source route to the destination of the data packet.

The overall logical structure of the project is divided into processing modules and a conceptual data structure is defined as Architectural Design.

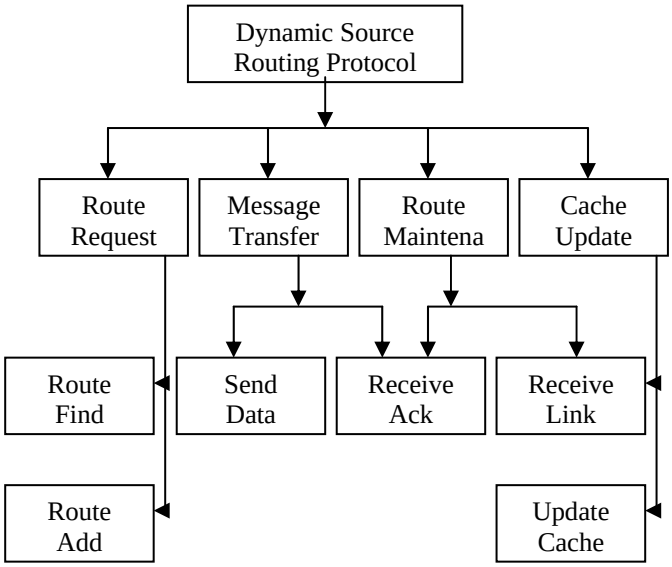

Figure 1. Architectural Design

III. IMPLEMENTATION

(A) Route Request - When a source node 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. The 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, which is the concatenation of the source route in the route request and the cached route. If the node does not have a cached route to the destination, it adds its address to the source route and rebroadcasts the route request. When the destination receives the route request, it sends a route reply containing the source route to the source. Each node forwarding a route reply stores the route starting from itself to the destination. When the source receives the route reply, it caches the source route.

(B) Message Transfer - The message transfer relates with that the sender node wants to send a message to the destination node after the path is selected and status of the destination node through is true. The receiver node receives the message completely and then it send the acknowledgement to the sender node through the router nodes where it is received the message.

(C) Route Maintenance - Route maintenance, the node forwarding a packet is responsible for confirming that the packet has been successfully received by the next hop. If no acknowledgement is received after the maximum number of retransmissions, the forwarding node sends a route error to the source, indicating the broken link. Each node forwarding the route error removes from its cache the routes containing the broken link.

(D) Cache Updating - When a node detects a link failure, our goal is to notify all reachable nodes that have cached that link to update their caches. To achieve this goal, the node detecting a link failure needs to know which nodes have cached the broken link and needs to notify such nodes efficiently.

IV. Conclusion

The algorithm enables DSR to adapt quickly to topology changes. We show that, under non-promiscuous mode, the algorithm outperforms DSR with path caches by up to 19% and DSR with *link-maxlife* by up to 41% in packet delivery ratio. It reduces normalized routing overhead by up to 35% for DSR with path caches. Under promiscuous mode, the algorithm improves packet delivery ratio by up to 7% for both caching strategies, and reduces delivery latency by up to 27% for DSR with path caches and 49% for DSR with *link-maxlife*.

The improvement demonstrates the benefits of the algorithm. Although the results were obtained under a certain type of mobility and traffic models, we believe that the results apply to other models, as the algorithm quickly removes stale routes no matter how nodes move and which traffic model is used. The central challenge to routing protocols is how to efficiently handle topology changes. Proactive protocols periodically exchange topology updates among all nodes, incurring significant overhead. On-demand protocols avoid such overhead but face the problem of cache updating. Our work combines the advantages of proactive and on-demand protocols: on-demand link failure detection and proactive cache updating. Our solution is applicable to other on-demand routing protocols. We conclude that proactive cache updating is key to the adaptation of on-demand routing protocols to mobility.

References

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

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

[3] 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.

[4] D. Johnson and D. Maltz. Dynamic Source Routing in ad hoc wireless networks. In *Mobile Computing*, T. Imielinski and H. Korth, Eds, Ch. 5, pp. 153–181, Kluwer, 1996.

[5] X. Yu and Z. Kedem. A distributed adaptive cache update algorithm for the Dynamic Source Routing protocol. In *Proc. 24th IEEE INFOCOM*, March 2005. (An earlier version appeared as NYU CS Technical Report TR2003-842, July 2003.)

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