Mobility, Route Caching, and TCP Performance in Mobile Ad Hoc Networks

by

Xin Yu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2005

David B. Johnson

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

To my parents

Acknowledgements

I would like to thank Prof. Zvi Kedem for his guidance in the first year of my thesis work, especially for his help while I was preparing the first submission of my algorithm.

I gratefully acknowledge my advisor, Prof. David Johnson from Rice University, for his guidance, encouragement, and help. Since no professor here works on networks, I did not know whether the topic I chose can be a thesis topic. When I asked for help from Prof. Johnson, he readily agreed to be my thesis reader and helped me decide this topic; later he became my thesis advisor. Prof. Johnson gave me not only valuable technical advice but also good suggestions on improving my writing skills. He is a great advisor.

Abstract

In a mobile ad hoc network, mobile nodes communicate with each other through wireless links. Mobility causes frequent topology changes. This thesis addresses the fundamental challenges mobility presents to on-demand routing protocols and to TCP.

On-demand routing protocols use route caches to make routing decisions. Due to mobility, cached routes easily become stale. To address the cache staleness issue, prior work used adaptive timeout mechanisms. However, heuristics cannot accurately estimate timeouts because topology changes are unpredictable. I propose to proactively disseminate the broken link information to the nodes that have cached the link. I define a new cache structure called a cache table to maintain the information necessary for cache updates, and design a distributed cache update algorithm. This algorithm is the first work that proactively updates route caches in an adaptive manner. Simulation results show that proactive cache updating is more efficient than adaptive timeout mechanisms. I conclude that proactive cache updating is key to the adaptation of on-demand routing protocols to mobility.

TCP does not perform well in mobile ad hoc networks. Prior work provided link failure feedback to TCP so that it can avoid invoking congestion control mechanisms for packet losses caused by route failures. Simulation results show that my cache update algorithm significantly improves TCP throughput since it reduces the effect of mobility on TCP. TCP still suffers from frequent data and ACK losses. I propose to make

routing protocols aware of lost TCP packets and help reduce TCP timeouts. I design two mechanisms that exploit cross-layer information awareness: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). EPLN notifies TCP senders about lost data. BEAD retransmits ACKs at intermediate nodes or at TCP receivers. Simulation results show that the two mechanisms significantly improve TCP throughput. I conclude that cross-layer information awareness is key to making TCP efficient in the presence of mobility.

I also study the impact of caching strategies on the scalability of on-demand routing protocols with mobility. I show that making route caches adapt quickly and efficiently to topology changes is key to the scalability of on-demand routing protocols with mobility.

Contents

List of Figures

Chapter 1

Introduction

A mobile ad hoc network [32] is a collection of mobile nodes that communicate with each other through wireless links. There is no fixed network infrastructure; nodes cooperate to forward packets for other nodes not within wireless transmission range. Mobile ad hoc networks can be used in situations where an infrastructure does not exist or is inconvenient to use, thus enabling anytime and anywhere connectivity. Promising applications of ad hoc networks include emergency disaster relief, military communication, space exploration, and monitoring scenarios such as sensor networks.

Mobile ad hoc networks have several unique characteristics that differentiate them from traditional wired networks [8]: limited bandwidth, scarce energy since nodes typically rely on batteries for their power, and dynamic network topology. Since nodes may move arbitrarily, network topology changes frequently. Mobility is a crucial factor affecting the design of network protocols for ad hoc networks, including MAC (medium access control), routing, and transport protocols.

This thesis addresses the fundamental challenges mobility presents to on-demand routing protocols and to TCP (Transport Control Protocol) in ad hoc networks. The first problem I address is how to make on-demand routing protocols quickly adapt to

topology changes. The second problem I address is how to improve TCP performance in the presence of frequent packet losses. The third problem I address is how to improve the scalability of on-demand routing protocols with respect to mobility. In this chapter, I present the background, motivation, and contributions of my thesis work.

1.1 Routing Protocols for Mobile Ad Hoc Networks

Frequent topology changes present a fundamental challenge to routing protocols in ad hoc networks. Routing protocols for ad hoc networks can be classified into two major types: proactive and reactive (on-demand). Proactive routing protocols attempt to maintain up-to-date routing information to all nodes, regardless of the need for such routes. Examples of proactive protocols include DSDV (Destination-Sequenced Distance-Vector routing) [42], OLSR (Optimized Link State Routing) [28], and TBRPF (Topology Broadcast Based on Reverse-Path Forwarding routing) [5]. In contrast, on-demand routing protocols initiate a routing discovery only when a route is needed. Several routing protocols use on-demand mechanisms, including DSR (the Dynamic Source Routing protocol) [29, 30, 31], AODV (Ad-hoc On-Demand Distance Vector routing) [44, 43], TORA (Temporally-Ordered Routing Algorithm) [41], and LAR (Location-Aided Routing) [33].

Proactive routing protocols attempt to keep route tables up-to-date by periodically propagating topology updates throughout the network, thus incurring significant overhead. On-demand routing protocols avoid such overhead by adapting routing activities to traffic needs, thus efficiently utilizing network bandwidth and reducing power consumption. To reduce the overhead and latency of initiating a route discovery for each packet to be sent, on-demand routing protocols use route caches to store discovered routes for future use.

In my thesis, I focus on DSR [29, 30, 31], a well-known on-demand routing protocol. DSR consists of two on-demand mechanisms: Route Discovery and Route Maintenance. When a source node wants to send a packet to a destination to which it does not have a route, it initiates a Route Discovery by broadcasting a ROUTE REQUEST to its neighbors. Each ROUTE REQUEST contains a unique identifier generated by the node initiating the Route Discovery. A node receiving a ROUTE REQUEST discards the ROUTE REQUEST if it has recently seen a ROUTE REQUEST with the same unique identifier. Otherwise, the node checks whether it has a route to the destination in its cache. If so, it sends a ROUTE REPLY to the source node including a source route, which is the concatenation of the source route accumulated 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 in the ROUTE REQUEST and rebroadcasts the ROUTE REQUEST. When the destination node receives the ROUTE REQUEST, it sends to the source node a ROUTE REPLY containing the source route from the ROUTE REQUEST. When a node forwards a ROUTE REPLY, it caches the route starting from itself to the destination. When the source node receives the ROUTE REPLY, it 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, the forwarding node sends a ROUTE ERROR to the source node, indicating the broken link. When a node receives (or forwards) a ROUTE ERROR, it removes routes containing the broken link from its cache.

1.2 The Cache Staleness Issue in On-Demand Routing Protocols

Due to mobility, cached routes can easily become stale. In on-demand routing, a node is not notified when a cached route becomes stale until it uses the route to send packets. Using stale routes causes packet losses if packets cannot be salvaged by intermediate nodes, increases delivery latency due to expensive link failure detections, and increases routing overhead caused by ROUTE ERRORS. Since a node can respond to a ROUTE REQUEST with a cached route, stale routes can also be quickly propagated to the caches of other nodes. The first problem I address in this thesis is how to make route caches quickly adapt to topology changes. This problem is very important because on-demand routing protocols use route caches to make routing decisions. The problem is also very challenging because topology changes are frequent.

To address the cache staleness issue, prior work in DSR used adaptive timeout mechanisms [23, 39, 37]. 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. Thus, adaptive timeout mechanisms cannot make route caches quickly adapt to topology changes.

1.3 Overview of TCP

TCP is a reliable transport protocol that is widely used in the Internet. Before introducing the TCP performance issue in mobile ad hoc networks, I give an overview of TCP

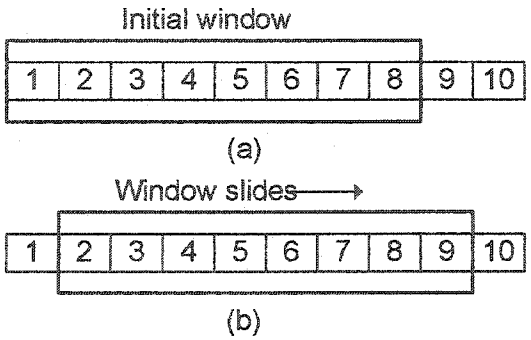


Figure 1.1: An Example of Sliding Window in TCP

including TCP congestion control mechanisms [27, 1].

TCP provides reliable end-to-end data transfer through a technique known as *positive acknowledgement with retransmission* [9]. This technique requires a TCP receiver to send an *acknowledgement* (ACK) to a TCP sender as it receives data. The sender starts a timer called the retransmission timer when it sends a packet. If the timer expires before the sender receives a corresponding ACK, the sender retransmits the packet.

If a sender sends each packet and waits for an ACK before sending another packet, potential throughput will be wasted. To make transmission efficient, TCP uses a technique known as a *sliding window*. The sliding window allows the sender to send multiple packets before waiting for an ACK. The window size is defined as the number of transmitted packets that can be unacknowledged at any given time [9]. For example, as shown in Figure 1.1, the window size is 8, and thus the sender is permitted to send 8 packets before waiting for an ACK. Once the sender receives an ACK for the first packet inside the window, it slides the window, enabling it to send the next packet. The maximum size of the sliding window is limited by the size of the receiver's advertised window, which indicates the amount of available buffer space at the receiver.

To avoid congestion, TCP maintains a second limit called the *congestion window*, which restricts the amount of data TCP can send. TCP must not send data with a se-

quence number higher than the sum of the highest acknowledged sequence number and the minimum of the congestion window size and the receiver's advertised window size.

TCP congestion control mechanisms consist of four algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery [1]. The slow start and congestion avoidance algorithms are used by a TCP sender to control the amount of data that can be present in the network at any time. The slow start algorithm is used at the beginning of a transfer or after retransmitting a lost packet. During slow start, the congestion window size is increased by one packet each time an ACK is received; thus, it grows exponentially. When the congestion window size reaches the slow-start threshold (*ssthresh*), TCP performs congestion avoidance. During congestion avoidance, the congestion window size is incremented by (*segsize* $\times$ *segsize*) / *cwnd* each time an ACK is received, where *segsize* is the packet (or segment) size and *cwnd* is the congestion window size, maintained in bytes. Thus, the congestion window size grows linearly and is increased by one packet each time a full window has been acknowledged. Congestion avoidance continues until congestion is detected.

TCP assumes that a packet loss is due to congestion. TCP detects a packet loss using a retransmission timeout (RTO) and duplicate ACKs. TCP uses cumulative ACKs, which acknowledge up through the last in-order packet received. A TCP receiver sends a duplicate ACK when an out-of-order packet arrives [1]. A duplicate ACK informs the TCP sender that a packet was received out-of-order and which sequence number is expected. A duplicate ACK can be caused a lost packet or by the re-ordering of packets by the network. TCP assumes that a packet is lost if it receives three duplicate ACKs, since it is unlikely that three duplicate ACKs are caused by the re-ordering of packets.

If the retransmission timer expires, TCP retransmits the lost packet, sets *ssthresh* to half of the congestion window size, reduces the congestion window size to one packet, and enters slow start. If a sender receives three consecutive duplicate ACKs, it invokes

the fast retransmit and fast recovery algorithms. The sender retransmits the packet that appears to be lost, and the fast recovery algorithm governs the transmission of new data until a non-duplicate ACK arrives. When the ACK that acknowledges the new data arrives, TCP sets *cwnd* to *ssthresh* and enters congestion avoidance.

1.4 TCP Performance in Mobile Ad Hoc Networks

Mobility presents a fundamental challenge to TCP [27, 1, 9], and TCP performance degrades significantly in mobile ad hoc networks [21, 13, 51]. It has been observed that route failures are the primary reason for most packet losses [13, 51]. Since TCP assumes that packet losses occur because of congestion, it will invoke congestion control mechanisms even for packet losses caused by route failures, resulting in the reduction in throughput. It is important to make TCP perform well in ad hoc networks, because of the universal use of TCP in the Internet and because connecting ad hoc networks to the Internet is a natural trend, which holds the promise of pervasive communication.

It is challenging to make TCP efficient in ad hoc networks. Several modifications to TCP [7, 21, 10, 36] have been proposed to address the problems caused by mobility. The major approach has been to provide link failure feedback to TCP so that TCP can avoid responding to route failures as if congestion had occurred. ELFN (Explicit Link Failure Notification) [21] is such a mechanism. With ELFN, when a node detects a link failure, it notifies the TCP sender about the link failure and the packet that encountered the failure. When receiving a notification, TCP freezes its retransmission timer and periodically sends a probing packet until it receives an ACK. TCP then restores congestion control mechanisms and continues as normal. ELFN was shown to outperform TCP [21].

TCP benefits from link failure feedback but is still affected by route failures. Holland and Vaidya [21] observed that TCP experiences repeated route failures due to the inabil-

ity of a TCP sender's routing protocol to quickly recognize and remove stale routes from its cache. This problem is complicated by allowing nodes to respond to ROUTE REQUESTS with cached routes, because they often respond with stale routes. They also showed that turning off replying from caches improves TCP performance for a network with a single TCP connection. However, this approach will degrade TCP performance in a network with multiple traffic sources due to increased routing overhead.

Prior work ignored an important fact: route failures are not equivalent to packet losses. Route failures do not imply that packets are lost, since packets can be salvaged by an intermediate node using a cached route. Many packets are dropped without being noticed: upon a link failure, existing routing protocols drop all packets with the same next hop from the network interface transmit queue. TCP will time out because of these losses. It will also time out for ACK losses caused by route failures. It is unclear how to make TCP efficient in the presence of frequent packet losses.

1.5 Thesis Contributions

This thesis makes four contributions. First, I addressed the cache staleness issue of on-demand routing protocols through a novel distributed cache update algorithm [58]. This algorithm is the first work that proactively updates route caches in an adaptive manner, in contrast to proactive protocols in which topology updates are periodically propagated to all nodes. I show that proactive cache updating is more efficient than adaptive timeout mechanisms. Second, I show that my cache update algorithm significantly improves TCP throughput without any modification to TCP [57]. This is the first work to demonstrate improved TCP performance through an efficient caching strategy. Third, I addressed the challenge mobility presents to TCP by exploiting cross-layer information awareness [55]. I proposed two mechanisms: early packet loss notification

(EPLN) and best-effort ACK delivery (BEAD). Both mechanisms significantly improve TCP throughput. Finally, I show that my cache update algorithm makes DSR scale significantly better with mobility [56]. This is the first work that studies the scalability of on-demand routing protocols with mobility. In the reminder of this section, I give an overview of my thesis work.

To make route caches quickly adapt to topology changes, I proposed to proactively disseminate the information about a broken link to the nodes that have that link in their route caches. This goal is very challenging because of mobility and fast propagation of routing information. To achieve this goal, I defined a new cache structure called a cache table and designed a distributed cache update algorithm. During route discoveries and data transmission, each node maintains in its cache table the information necessary for cache updates. When a link failure is detected, the algorithm notifies in a distributed manner all reachable nodes that have cached the link. The algorithm does not use any ad hoc parameters, thus making route caches fully adaptive to topology changes. I show that the algorithm outperforms DSR with path caches and with *Link-MaxLife*, an adaptive timeout mechanism for link caches [23]. I conclude that proactive cache updating is key to the adaptation of on-demand routing protocols to mobility.

Most attempts to improve TCP performance focused on transport layer mechanisms. I proposed a new approach to improve TCP performance at the network layer: reducing route failures by making route caches quickly adapt to topology changes. I investigated the impact of my cache update algorithm on TCP performance, without any modification to TCP. I show that this algorithm significantly improves TCP throughput and reduces normalized routing overhead. I conclude that it is important to make route caches reflect topology changes quickly so that the adverse effect of mobility on TCP is reduced.

To make TCP efficient in the presence of frequent packet losses, my solution is to exploit cross-layer information awareness. I proposed to make routing protocols *aware*

of lost TCP packets and help reduce TCP timeouts. To this end, I designed two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). EPLN seeks to notify TCP senders about lost data so that TCP retransmits lost packets earlier. For lost ACKs, BEAD attempts to retransmit ACKs either at intermediate nodes or at TCP receivers; therefore, TCP is unaware of lost ACKs. Both mechanisms extensively use cached routes, without initiating route discoveries at any intermediate node. The two mechanisms can be adapted to any routing protocol, as they address general problems that occur at the network layer. I evaluated TCP-ELFN enhanced with the two mechanisms using two caching strategies for DSR, path caches and my cache update algorithm. I show that TCP-ELFN with EPLN and BEAD significantly outperforms TCP-ELFN under both caching strategies. I conclude that cross-layer information awareness is key to making TCP efficient in the presence of mobility.

Scalability is an important design goal of routing protocols. Prior work in ad hoc network routing mainly focused on making routing protocols scale with network size [26, 50, 3, 35, 11]. It is essential for routing protocols to scale not only with network size but also with mobility. If a routing protocol cannot scale with mobility, its performance will degrade significantly as mobility increases for medium-scale networks. However, the scalability of on-demand routing protocols with mobility has not been studied. I studied the impact of caching strategies on the scalability of on-demand routing protocols with mobility in the context of DSR. I considered three caching strategies for DSR: path caches with FIFO, link caches with adaptive timeout mechanisms, and cache tables with my cache update algorithm. Simulation results show that the algorithm makes DSR scale significantly better with mobility. I conclude that making route caches adapt quickly and efficiently to topology changes is key to the scalability of on-demand routing protocols with mobility.

1.6 Thesis Outline

The organization of this thesis is as follows. In Chapter 2, I discuss related work on caching strategies in on-demand routing protocols and on TCP performance in ad hoc networks. In Chapter 3, I present the design and evaluation of my distributed adaptive cache update algorithm. In Chapter 4, I show that the cache update algorithm improves TCP throughput without modification to TCP. In Chapter 5, I show how mobility affects TCP and present two mechanisms to improve TCP throughput. In Chapter 6, I present my study on the scalability of on-demand routing protocols with mobility, and finally in Chapter 7, I present my conclusions.

Chapter 2

Related Work

In this chapter, I discuss related work on caching strategies in on-demand routing protocols and on TCP performance in ad hoc networks.

2.1 Caching Strategies in On-Demand Routing Protocols

Proactive protocols periodically disseminate topology updates in order to keep routing tables up-to-date. On-demand routing protocols use route caches to reduce the cost of route discoveries but face the problem of cache maintenance. Maltz *et al.* [38] were the first to study the cache performance of DSR. They found that the majority of ROUTE REPLIES are based on cached routes, and only 59% of ROUTE REPLIES carry correct routes. They also observed that even ROUTE REPLIES from the target are not 100% correct, since routes may break while a ROUTE REPLY is sent back to the source node.

Hu and Johnson [24] proposed a mechanism called epoch numbers to reduce cache staleness. This mechanism prevents a node from re-learning a stale link after having earlier heard that the link is broken. The mechanism does not rely on ad hoc mechanisms such as a short-lived negative cache; rather, it allows a node having heard of a broken

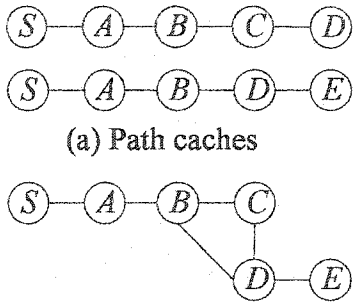


Figure 2.1: Path Cache and Link Cache Structure in DSR

link and a discovery of the same link to sequence the two events in order to determine which event occurred before the other. The mechanism did help reduce stale cache information; however, re-learning stale links is only one aspect of the cache staleness issue. How to quickly remove stale routes from route caches remains unaddressed.

Hu and Johnson [23] studied the design choices for cache structure, cache capacity, and cache timeout. The first cache structure proposed for DSR is a path cache [6], in which a node caches each path separately. Figure 2.1 (a) illustrates an example path cache for node *S*. In [23], the authors proposed a link cache structure and several adaptive timeout mechanisms for link caches. In a link cache, a node adds each link to a topology graph. Links obtained from different routes can form new routes, which may not be available in path caches. Figure 2.1 (b) illustrates an example link cache for node *S*. Adaptive timeout mechanisms estimate the lifetime of a link based on observed link usages and breakages. *Link-MaxLife* [23] was shown to outperform other adaptive timeout mechanisms. In *Link-MaxLife*, the timeout of a link is chosen according to a stability table in which a node records its perceived stability of each other node. A node chooses the shortest-length path that has the longest expected lifetime (the highest minimum timeout of any link in the path). When a link is used, the stability metric for both endpoints is incremented by the amount of time since the link was last used, multiplied

by some factor. When a link is observed to break, the stability metric for both endpoints is multiplicatively decreased by a different factor. When a link is added to the cache, it is given a lifetime equal to the stability of the less stable endpoint, but the lifetime is not allowed to be set to less than 1 s. The ad hoc parameters are set as follows: the additive increase factor is 4, the multiplicative decrease factor is 1.25, and the stability metric for each node is initialized to be 25 s.

Lou and Fang [37] proposed an adaptive timeout mechanism that adjusts the link lifetime based on the moving average of link lifetime statistics. Each link in a link cache is associated with three attributes: *born*, *lastUsed*, and *liveTo*. Attribute *born* indicates the time when a link is cached. Attribute *lastUsed* is the time when a link is last used to forward a packet. Attribute *liveTo* is the time when a link expires. The statistical lifetime data is collected whenever a link is removed. If a link is removed due to the reception of a ROUTE ERROR, lifetime l is calculated as the difference between the current time and the value of *born*. If a link is removed due to timeouts, lifetime l is calculated as the difference between the value of *lastUsed* and the value of *born*. Variable *LIFETIME* indicates the estimation of the link lifetime. It is assigned a static value initially and is adjusted dynamically using a moving average method whenever a lifetime l is collected: $LIFETIME = (1-\alpha) \times LIFETIME + \alpha \times l$, where α is set to 0.01.

Marina and Das [39] proposed wider error notification and timer-based route expiry. Wider error notification aims at increasing the speed and extent of ROUTE ERROR propagation. ROUTE ERRORS are transmitted as *broadcast* packets at the MAC layer. A node receiving a ROUTE ERROR will rebroadcast it, but only if the node has a cached route containing the broken link *and* that route was used to forward packets. Thus, stale routes propagated through ROUTE REPLIES and cached for future use will not be removed. Moreover, some nodes that have cached broken links may not receive notifications, since broadcast is unreliable. Timer-based expiry is a heuristic for adaptive

selection of timeouts based on the average route lifetime and the time between link breakages. An average route lifetime is obtained using the lifetime of all broken routes in the past. Timeout ΔT is defined as the maximum of $\alpha \times$ average route lifetime and the time since last link breakage, where α is set to 5 and the minimum ΔT is 1 s. The time since last link breakage is used to correct the ΔT estimates during periods when no routes break. Although this approach works well when routes break uniformly in time, mobility may not be uniform in time or space.

Adaptive timeout mechanisms have several limitations. First, heuristics cannot give accurate timeout estimates because topology changes are unpredictable. Second, ad hoc parameters used in such mechanisms work well for certain scenarios but may not work well for others. Finally, the effectiveness of such mechanisms relies on the timely acquisition of the link failure information; however, a node may not be able to know about a link failure as soon as a link breaks. In contrast, my cache update algorithm does not use ad hoc mechanisms or heuristics. The algorithm allows the nodes that have cached a broken link to know about the link failure at the earliest possible time.

AODV [44, 43] (the Ad hoc On-demand Distance Vector routing protocol) uses a mechanism called "the precursor list" for ROUTE ERROR reporting. For each route table entry, a node maintains a list of precursors that may be forwarding packets on this route. When a node forwards a ROUTE REPLY, it adds the last hop node (from which it received the ROUTE REQUEST) into the precursor list for the forward route entry, i.e., the entry for the destination. The node also adds the next hop towards the destination into the precursor list for the reverse route entry, i.e., the entry for the originator of the ROUTE REQUEST. These precursors will receive notifications from the node when the next hop link is detected as broken. Each time a route table entry is used, the lifetime of that route is updated to be the current time plus a fixed parameter. When a route table entry is expired, the precursor list associated with the entry will be removed. The

precursor list is designed with a similar goal as the *ReplyRecord* field in a cache table, which records the neighbor to which a ROUTE REPLY is forwarded and the route starting from the current node to the destination. The main difference is that there is no timeout for a *ReplyRecord* entry. Moreover, precursor lists keep track of only the nodes recently using some route; once a route table entry is expired, precursors that have not used or did not recently use that route will not be tracked. In contrast, my mechanism completely keeps track of topology propagation state in a distributed manner.

2.2 TCP Performance in Ad Hoc Networks

As described in Section 1.4, TCP does not perform well in ad hoc networks. Several studies have attempted to identify the factors that affect TCP performance. Gerla *et al.* [16] investigated the impact of the MAC protocol on TCP performance. Holland and Vaidya [22] studied the effect of routing and link layer mechanisms on TCP performance in a static ad hoc network. Fu *et al.* [14] studied the effect of the wireless channel on TCP throughput and loss and proposed two link layer techniques to improve TCP throughput. Xu *et al.* [53] studied the TCP fairness issue in ad hoc networks and proposed a neighborhood RED (Random Early Detection) [15] scheme to improve TCP unfairness. Fu *et al.* [13] observed that mobility has the most significant impact on TCP performance. TCP achieves only about 10% of a reference TCP's throughput. As mobility increases, the relative throughput drop ranges from almost 0% in static scenarios to 1000% in highly mobile scenarios where node speed is 20 m/s. In contrast, congestion and mild channel errors have less effect on TCP, with less than 10% performance drop compared with the reference TCP.

Anantharaman and Sivakumar [2] identified several problems at the MAC and the network layers and proposed a framework called ATRA to address these problems.

ATRA includes three mechanisms: Symmetric Route Pinning (SRP), Route Failure Prediction (RFP), and Proactive Route Errors (PRE). In a routing protocol, the route used to send data packets can be different from the route used to send ACKs. SRP aims to reduce route failures by ensuring that the ACK path is always the same as the data path. In RFP, a node predicts the occurrence of a link failure based on the progression of signal strength of packet receptions from the concerned neighbor. When a source receives a predicted route failure message, it initiates a route discovery but continues to use the current path. PRE aims at reducing the latency involved in the propagation of link failure information to the sources that use the broken link. In PRE, each node maintains a cache of TCP senders that have used a particular link in the past T seconds. When a link failure is detected, all sources that have used the link in the past T seconds are informed about the link failure. Thus, this approach reduces the latency involved in route failure propagation.

Sundaresan *et al.* [51] argued that a majority of the components of TCP are inappropriate for ad hoc networks. They developed a new transport protocol called ATP. ATP consists of the following mechanisms: rate based transmissions, quick-start during connection initiation and route switching, network supported congestion detection and control, no retransmission timeouts, decoupled congestion control and reliability, and coarse grained receiver feedback. ATP focuses on achieving effective congestion control and reliability.

Most prior work on TCP performance in ad hoc networks focused on transport layer mechanisms. The major approach has been to provide link failure feedback to TCP so as to prevent TCP from invoking congestion control mechanisms. Chandran *et al.* [7] proposed a feedback-based technique called TCP-Feedback. An intermediate node that detects a broken link sends a *route failure notification* (RFN) to the TCP sender. TCP freezes its state and resumes transmission only when it receives a *route reestablishment*

notification (RRN) from the intermediate node. This technique was not evaluated in mobile ad hoc networks. Holland and Vaidya [21] proposed *explicit link failure notification* (ELFN) to counter the effects of mobility. ELFN freezes TCP upon route failures and periodically sends a probing packet until a sender receives an ACK. The authors also observed that stale routes seriously degrade TCP throughput. ELFN was shown to outperform TCP in mobile ad hoc networks. However, the authors studied only scenarios with a single TCP connection. It is unclear how ELFN performs when traffic load is high. I will give more detailed discussion about ELFN in Chapter 5.

Monks *et al.* [40] studied ELFN in both static and dynamic networks and proposed hop-by-hop rate control-based mechanisms along with ELFN for congestion control. Dyer and Boppana [10] proposed a heuristic called *fixed RTO* to distinguish route failures from congestion. They assume that consecutive timeouts are an evidence of a route loss. Thus, a TCP sender retransmits the unacknowledged packet before the second RTO expires and the RTO is not doubled a second time. The RTO remains fixed until the retransmitted packet is acknowledged. Liu and Singh [36] introduced a thin layer between TCP and the network layer, which listens to the network feedback information provided by *explicit congestion notification* (ECN) and by *destination unreachable* message and puts TCP at the sender into the appropriate state. Wang and Zhang [52] proposed an approach to make TCP adapt to frequent route changes without relying on network feedback. This approach is based on TCP detecting out-of-order delivery events and inferring route changes from these events.

In summary, prior work mainly focused on making TCP aware of route failures. However, it is insufficient to notify TCP only about route failures. In contrast, my work focuses on issues at the network layer, the transport layer, and cross-layer. My work aims to make routing protocols in ad hoc networks efficiently handle both data and ACK losses so as to reduce TCP timeouts for mobility-induced losses.

Chapter 3

The Distributed Adaptive Cache Update Algorithm

On-demand routing protocols use route caches to make routing decisions. Due to mobility, cached routes easily become stale. To address the cache staleness issue in DSR, I propose to proactively disseminate the information about a broken link to the nodes that have that link in their caches. I define a new cache structure called a cache table and design a novel distributed cache update algorithm. Each node maintains in its cache table the information necessary for cache updates. Based on the information kept by each node, the algorithm notifies all reachable nodes that have cached a broken link in a distributed manner. Therefore, it enables route caches to adapt quickly to topology changes. In this chapter, I present the design and evaluation of my distributed cache update algorithm. Although this work is presented in the context of DSR, it can be adapted to other on-demand ad hoc network routing protocols as well.

3.1 Introduction

To address the cache staleness issue, prior work [23, 39, 37] in DSR used adaptive timeout mechanisms. Such mechanisms use heuristics with ad hoc parameters to estimate 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. Moreover, topology changes are unpredictable. If the timeout is set too short, valid routes will be removed; if the timeout is set too long, stale routes will be kept in caches. Thus, adaptive timeout mechanisms cannot make route caches adapt quickly to topology changes.

To evict stale routes faster, some implementations of DSR use a small cache size. However, as traffic load or network size increases, small caches will cause route rediscoveries, because more routes need to be stored, but small caches cannot hold all useful routes. If the cache size is set large, more stale routes will stay in caches because FIFO becomes less effective. It was shown [23] that path caches with unlimited size perform much worse than caches with limited size, due to the large amount of ROUTE ERRORS caused by the use of stale routes. No single cache size provides the best performance for all mobility scenarios [23]. Whether the cache size is small or large, fast cache updating is critical.

I propose to proactively disseminate the information about a broken link to the nodes that have that link in their caches. Proactive cache updating is key to making route caches quickly reflect topology changes. It is also important to constrain cache update notifications to the nodes that have cached a broken link to avoid unnecessary overhead. Thus, when a link failure is detected, my goal is to notify all reachable nodes that have cached the broken link to update their caches.

I define a new cache structure called a cache table to maintain the information necessary for cache updates. A cache table has no capacity limit, and thus its size dynamically changes as needed. Each node maintains in its cache table two types of information for

each route. The first type of information is how well routing information is synchronized among nodes on a route: whether a link has been cached in only upstream nodes, or in both upstream and downstream nodes, or neither. The second type of information is which neighbor has learned which links through a ROUTE REPLY. Thus, for each cached link, a node knows which neighbors have that link in their caches. Therefore, *topology propagation state*, the information necessary and sufficient to remove stale routes, is kept in a distributed manner.

I design a novel algorithm for distributed cache updating using the information kept by each node. When a link failure is detected, the algorithm notifies selected neighborhood nodes about the broken link: the closest upstream and/or downstream nodes in each route containing the broken link, and other neighbors that cached the link through ROUTE REPLIES. When a node receives a notification, it starts the algorithm to notify selected neighbors. Therefore, the information about a broken link will be quickly propagated to all reachable nodes that need to be notified.

My algorithm has the following desirable properties:

- Distributed: The algorithm uses only local information and communicates with neighborhood nodes; therefore, it is scalable with network size.
- Adaptive: The algorithm notifies only the nodes that have cached a broken link to update their caches; therefore, cache update overhead is low.
- Proactive on-demand: Proactive cache updating is triggered on-demand, without periodic behavior.
- Without ad hoc mechanisms: The algorithm does not use any ad hoc parameters, thus making route caches fully adaptive to topology changes.

Each node gathers the information about which node learns which link through forwarding packets, not through promiscuous mode, which is an optimization for DSR [38].

To handle situations where promiscuous mode is used, I combine my algorithm and the secondary cache used in DSR with path caches, without modification to the algorithm.

I evaluated the algorithm with and without promiscuous mode through detailed simulations. Although *Link-MaxLife* was not designed to operate without promiscuous mode, it is useful to evaluate it in this way to reveal more about its behavior. 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. 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 and 49% for *Link-MaxLife*.

My contributions are threefold. First, I addressed the cache updating issue of on-demand routing protocols. Second, I show that proactive cache updating is more efficient than adaptive timeout mechanisms. Finally, I conclude that proactive cache updating is key to the adaptation of on-demand routing protocols to mobility.

The organization of this chapter is as follows. Section 3.2 gives an overview of route caching in DSR. Section 3.3 describes my cache update algorithm and two algorithms used to maintain the information for cache updating. In Section 3.4, I present an evaluation of my algorithm, and in Section 3.5, I present conclusions.

3.2 Routing Caching in DSR

Two caching structures have been proposed for DSR: path caches [6] and link caches [23]. In a path cache, a node stores each route starting from itself to another node separately. In a link cache, a node adds a link to a topology graph, which represents the node's view of the network topology. Links obtained from different routes can form new routes. Thus, link caches provide more routing information than path caches.

As described in Chapter 1, DSR uses *source routes* in routing packets: each packet carries the complete path to its destination in the packet header. A node can learn a routes through forwarding ROUTE REPLIES and data packets, or by overhearing packets when promiscuous mode is used [38]. Although a ROUTE REQUEST also contains a source route accumulated so far, DSR does not cache such a route because ROUTE REQUESTS are broadcast packets and hence links discovered may not be bi-directional [31]. Due to the same reason, when forwarding a ROUTE REPLY, DSR caches only the links that have been confirmed by the MAC layer to be bi-directional [31], which are the downstream links starting from the node to a destination. When forwarding a data packet, a node caches the upstream links as a separate route. After initiating a Route Discovery, a source node may learn many routes returned either by intermediate nodes or by the destination; it will cache all those routes. Thus, DSR aggressively caches and uses routing information.

As described in Chapter 1, DSR uses on-demand Route Maintenance to detect whether a route has broken. If any link on a source route is detected as broken, the node detecting the link failure will send a ROUTE ERROR to the source. Thus, on-demand Route Maintenance removes stale routes only from the upstream nodes. Besides Route Maintenance, DSR uses two mechanisms to maintain its cache. First, a source piggybacks on the next ROUTE REQUEST the last broken link information, which is called a GRATUITOUS ROUTE ERROR. Although this optimization helps remove stale routes from more caches, GRATUITOUS ROUTE ERRORS are not able to reach all nodes whose caches contain the broken link, because some ROUTE REQUESTS will not be further propagated due to the use of responding to ROUTE REQUESTS with cached routes. Second, DSR uses heuristics: a small cache size with FIFO replacement for path caches and adaptive timeout mechanisms for link caches [23], where link timeouts are chosen based on observed link usages and breakages.

3.3 The Distributed Cache Update Algorithm

In this section, I first describe the cache staleness issue. Then I give the definition of a cache table and present two algorithms used to maintain the information for cache updates. Finally, I present my cache update algorithm in detail.

3.3.1 Problem Statement

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. I 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.

It is not necessary to detect whether a route is active or post-active, but these terms help clarify the cache staleness issue. Stale pre-active and post-active routes will not be detected until they are used. Due to the use of responding to ROUTE REQUESTS with cached routes, stale routes may be quickly propagated to the caches of other nodes. Thus, pre-active and post-active routes are an important source of cache staleness.

An example of the cache staleness issue is shown in Figure 3.1. Assume that route *ABCDE* is active, route *FGCDH* is post-active, and route *IGCDJ* is pre-active. Thus, node *C* has cached both the upstream and the downstream links for the active and post-active routes, but only the downstream links, *CDJ*, for the pre-active route. When forwarding a packet for the source *A*, node *C* detects that the link from node *C* to node *D* is broken. It removes stale routes from its cache and sends a ROUTE ERROR to node *A*. However, the downstream nodes, *D* and *E*, will not know about the broken link.

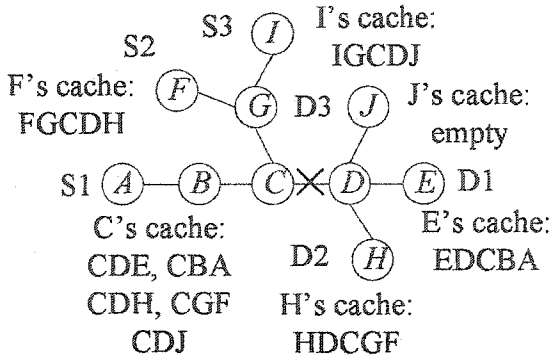


Figure 3.1: An Example of Routing Caching in DSR

Moreover, node *C* does not know that other nodes also have cached the broken link, including all the nodes on the post-active route, *F, G, D*, and *H*, and the upstream nodes on the pre-active route, *I* and *G*.

Stale routes have several adverse effects:

- Using stale routes causes packet losses if packets cannot be salvaged by intermediate nodes;
- Using stale routes increases delivery latency, since the MAC layer goes through multiple retransmissions before concluding a link failure;
- Using stale routes increases routing overhead, since the node detecting a link failure will send a ROUTE ERROR to the source;
- Using stale routes degrades TCP performance, since TCP will invoke congestion control mechanisms for packet losses caused by route failures.

3.3.2 Assumption

Promiscuous mode [38] disables the network interface's address filtering function and thus causes a protocol to receive all packets overheard by the interface. I did not consider

this mode when maintaining the information for cache updates, because it is impossible to know which neighbor overhears which link. In Section 3.3.10, I present an approach to handle promiscuous mode without any modification to the algorithm.

3.3.3 Overview

When a link failure is detected, my design goal is to disseminate the information about the broken link to all reachable nodes that have that link in 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.

My solution is to keep track of *topology propagation state* in a distributed manner. Topology propagation state means which node has cached which link. Each node maintains in its cache two types of information for each route: (1) whether a link has been cached in only upstream nodes, or both upstream and downstream nodes, or neither; and (2) which neighbor has learned which links through a ROUTE REPLY. A node gathers this information during route discoveries and data transmission, without introducing additional overhead. To remove stale routes, such information is 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.

Each entry in the cache table contains another 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 a *ReplyRecord* 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 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 they cached the link when the first data packet 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 information about a broken link will be quickly propagated to all reachable nodes that have cached that link.

3.3.4 The Definition of a Cache Table

A cache table has no capacity limit. Its size increases as new routes are discovered and decreases as stale routes are removed. There are four fields in a 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.

3.3.5 Information Collection and Maintenance

I use algorithms *addRoute* and *findRoute* to maintain the information necessary for cache updates. Algorithm *addRoute* is called when a node adds a route to its cache. Algorithm *findRoute* is called when a node tries to find a route to some destination.

Adding a Route

Algorithm *addRoute* is shown in Figure 3.2. A node adds a route either from a ROUTE REPLY or from a data packet. When receiving a ROUTE REPLY, a node attempts to add the route starting from itself to the destination (lines: 1–14). If the node is the source node (lines: 2–5), it stores the source route and sets *DataPackets* to 0, since the route has not been used. If the node is an intermediate node forwarding the ROUTE REPLY (lines: 7–14), it checks whether the route exists in its cache. If the route does not exist, the node creates an entry in which *DataPackets* is set to 0 and a *ReplyRecord* entry records which neighbor will learn the downstream links. If the route exists, the node adds a *ReplyRecord* entry if the entry does not exist in the corresponding table entry.

When receiving a data packet, a node checks whether the source route exists in its cache. If the route exists and *DataPackets* is 1 (lines: 16–18), the node sets *DataPackets* to 2, since the node is forwarding the second data packet. If the route does not exist and the node is the destination (lines: 19–21), it creates an entry and sets *DataPackets* to 1, since the destination has received the first data packet. If the route does not exist and the node is an intermediate node (lines: 24–34), it searches its cache for a route consisting of the downstream links of the source route. If such a route exists, the node adds the upstream links to the route to complete a full path, and sets *DataPackets* to 1, since it is forwarding the first data packet. The node also removes the *ReplyRecord* entry in which the concatenation of two fields is the route starting from the previous node to the destination of the packet. This is because the node has kept the information that the upstream nodes have cached the downstream links. The upstream nodes include the neighbor recorded in the *ReplyRecord* entry. If the node cannot complete a full path (lines: 35–36), it creates a cache table entry to store the source route and sets *DataPackets* to 1. For this case, the packet is the first packet from the source node that received a ROUTE REPLY sent by the current node or by another node that has a

```
Algorithm: addRoute
Input: PACKET p;
Variables:
ID first, last; ID next; ID netID; ID pre; boolean is_completed;
 1 if p is a ROUTE REPLY then
 2   if netID = p.dest then
 3     e := getFromCacheTable(p.srcRoute);
 4     if e = null then
 5       cacheTable := cacheTable ∪ {(p.srcRoute, (first, last), 0, ∅)}
 6   else
 7     newRoute := p.srcRoute.subPath(netID, last);
 8     reply_pair := (next, newRoute);
 9     e := getFromCacheTable(newRoute);
10     if e = null then
11       cacheTable := cacheTable ∪ {(newRoute, (first, last), 0, reply_pair)}
12     else
13       if reply_pair ∉ e.replyRecord then
14         e.replyRecord := e.replyRecord ∪ {reply_pair}
15 elseif p is a data packet then
16   e := getFromCacheTable(p.srcRoute);
17   if e ≠ null then
18     if e.DP = 1 then e.DP := 2
19   else
20     if netID = p.dest then
21       cacheTable := cacheTable ∪ {(p.srcRoute, (p.src, p.dest), 1, ∅)}
22     else
23       for each entry e ∈ cacheTable do
24         if e.srcDest.src = p.src and e.srcDest.dest = p.dest
           and p.src = p.route[0] then
25           temp := p.srcRoute.subPath(netID, last);
26           if temp = e.route and e.DP = 0 then
27             e.route := p.srcRoute; e.DP := 1; is_completed := TRUE
28         for each entry r ∈ e.replyRecord do
29           temp := p.srcRoute.subPath(pre, last);
30           if (r.nodeNotified || r.subrouteSent) = temp then
31             e.replyRecord := e.replyRecord \ {r};
32       if not is_completed and p.src = p.route[0] then
33         cacheTable := cacheTable ∪ {(p.srcRoute, (p.src, p.dest), 1, ∅)}
```

Figure 3.2: Pseudo Code for Algorithm *addRoute*

```
Algorithm: findRoute
Input:  ID dest, PACKET p, boolean respond_to_RREQ,
        boolean used_for_salvaging
Output: PATH route

 1 e0 := ∅;
 2 for each entry e ∈ cacheTable do
 3   if dest ∈ e.route then
 4     temp := e.route.subPath(netID, dest)
 5     if route = ∅ or |temp| < |route| then
 6       route := temp; e0 := e
 7 if e0 = ∅ then exit;
 8 if respond_to_RREQ then
 9   reply_pair := (p.srcRoute[p.srcRoute.length − 1], route)
10   if reply_pair ∉ e0.replyRecord then
11     e0.replyRecord := e0.replyRecord ∪ {reply_pair}
12 elseif not used_for_salvaging then
13   if route = e0.route and e0.DP ≠ 2 then
14     e0.DP := e0.DP + 1
15   else
16     cacheTable := cacheTable ∪ {(route, (netID, dest), 1, ∅)}
```

Figure 3.3: Pseudo Code for Algorithm *findRoute*

cached route to the destination, and the route consisting of the downstream links has been completed by another flow.

Finding a Route

Algorithm *findRoute* is shown in Figure 3.3. A node attempts to find a route either to respond to a ROUTE REQUEST or to send data packets. If a node finds a route to send a ROUTE REPLY, it adds an entry to the *ReplyRecord* field of the corresponding cache table entry, which includes the neighbor to which the ROUTE REPLY is forwarded and the found route. If a node is a source node and finds a route to send data packets, it increments *DataPackets* by 1 if it is not already set to 2, since the node is going to send the first or second data packet. If the found route is a sub-route of the route stored, the

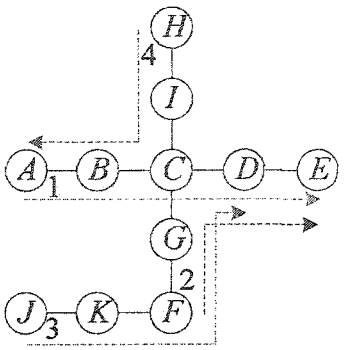


Figure 3.4: An Example Network

node creates a new cache table entry to store the found route and sets *DataPackets* to 1. If a node finds a route to salvage a data packet or forwards a data packet that was salvaged, it does not change the content of its cache table, because the synchronization information will be maintained when the first data packet traverses the original route.

Examples

I use the network shown in Figure 3.4 for examples. Initially, there are no data flows and all caches are empty. I use S-D for *SourceDestination* and DP for *DataPackets* in the tables describing the content of caches.

Node *A* initiates a route discovery to node *E*, and *E* sends a ROUTE REPLY to *A*. Each node forwarding the ROUTE REPLY creates a table entry (*addRoute*: 6–11). For instance, node *C* creates an entry consisting of four fields: the route consisting of the downstream links, the source and destination pair, the number of data packets the node has forwarded using the route, and which neighbor will learn which links through the ROUTE REPLY.

C:

Route	S-D	DP	ReplyRecord
CDE	*A E*	0	*B* ← *CDE*

When node A receives the ROUTE REPLY, it creates a table entry (*addRoute*: 1–5):

	Route	S-D	DP
A:	*ABCDE*	*A E*	0

When node A uses this route to send the first data packet, it increments *DataPackets* to 1 (*findRoute*: 12–14). Each intermediate node receiving the first data packet updates its table entry (*addRoute*: 22–31). For instance, node C increments *DataPackets* to 1, adds the upstream links to *CDE*, and removes the *ReplyRecord* entry, as the complete route indicates that the upstream nodes, A and B, have cached the downstream links, *CDE*.

	Route	S-D	DP
C:	*ABCDE*	*A E*	1

When node E receives the first data packet, it creates an entry (*addRoute*: 19–21) and its cache is the same as that of node C. When a node on this route receives the second data packet, it increments *DataPackets* to 2 (*addRoute*: 15–18).

Assume that after transmitting at least two data packets for flow 1, node C receives a ROUTE REQUEST from node G with source F and destination E. Before sending a ROUTE REPLY to G, C adds a *ReplyRecord* entry to its cache (*findRoute*: 1–11):

	Route	S-D	DP	ReplyRecord
C:	*ABCDE*	*A E*	2	*G* ← *CDE*

Before sending a ROUTE REPLY to node F, node G creates a table entry (*addRoute*: 6–11):

	Route	S-D	DP	ReplyRecord
G:	*GCDE*	*F E*	0	*F* ← *GCDE*

When F receives the ROUTE REPLY, it creates an entry (*addRoute*: 1–5):

	Route	S-D	DP
F:	*FGCDE*	*F E*	0

When C receives a ROUTE REQUEST from I with source H and destination A, it adds the second *ReplyRecord* entry to its cache (*findRoute*: 1–11):

	Route	S-D	DP	ReplyRecord	ReplyRecord
C:	*ABCDE*	*A E*	2	*G* ← *CDE*	*I* ← *CBA*

As described in Section 3.3.5, a node creates an entry to store a source route if a route consisting of the downstream links in the source route does not exist in its cache (*addRoute*: 32–33). Assume that flow 2 starts. When it reaches node D, D adds the second entry to its cache, as the sub-route *CDE* has been completed by flow 1. When receiving the first data packet, node D knows that its upstream nodes have cached the downstream link, *DE*.

	Route	S-D	DP
D:	*ABCDE*	*A E*	2
D:	*FGCDE*	*F E*	1

When F receives a ROUTE REQUEST from node K with source J and destination D, it extends its cache entry (*findRoute*: 1–11):

	Route	S-D	DP	ReplyRecord
F :	*FGCDE*	*F E*	2	*K* ← *FGCD*

Summary

The first data packet serves as a "synchronization signal," indicating that the upstream nodes have cached the downstream links. By storing a full path, a node keeps the association between the upstream and the downstream nodes, so that it knows which upstream

nodes need to be notified if a downstream link is broken. Based on the information in the *DataPackets* field, a node determines whether downstream nodes need to be notified about a broken link. Based on the information in the *ReplyRecord* field, a node knows which neighbors have cached a broken link through ROUTE REPLIES. Each node gathers the local information about which neighbors have cached which link, without the need to know all the nodes in the network that have cached a particular link. Thus, topology propagation state is kept in a distributed manner.

3.3.6 The Distributed Cache Update Algorithm

In this section, I present my cache update algorithm. I define a broken link as a *forward* or *backward* link. A link is a *forward* link if the flow using the route crosses the link in the same direction as the flow detecting the link failure; otherwise, it is a *backward* link. For these two types of links, the operation of the algorithm is symmetric.

Detailed Description

The algorithm starts either when a node detects a link failure or when it receives a notification. In either case, the algorithm generates a *notification list*, which is a list of neighborhood nodes that need to be notified. Each entry in this list includes a node and a cached route to reach that node. A notification will be sent as a ROUTE ERROR. The algorithm is shown in Figure 3.5, Figure 3.6, and Figure 3.7.

When a node detects a link failure, the algorithm checks the cache table. If a route contains a *forward* link, the algorithm does the following steps (lines: 11–25):

1. If *DataPackets* is 0, indicating that the route is pre-active, then no downstream node needs to be notified, because the downstream nodes did not cache the link when forwarding a ROUTE REPLY. For example, in Figure 3.4, before C forwards a ROUTE REPLY to B, it caches route CDE, sets *DataPackets* to 0, and creates a *ReplyRecord* entry

```
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
 3     has_broken_link := TRUE;
 4     direction := forward
 5   elseif link (to,from) ∈ e.route then
 6     has_broken_link := TRUE;
 7     direction := backward
 8   else has_broken_link := FALSE;
 9   if has_broken_link then
10     position := Index(e.route,from);
11     if detect_by_me then
12       if direction = forward then
13         if (e.DP = 1 or e.DP = 2)
            and (not isFirstNode(e.route,netID)) then
14           notifyList := notifyList ∪ {(e.route[position − 1],
                 (netID||e.route[position − 1]))}
15         if e.DP = 2 or (e.DP = 1 and
            (not (p is a data packet
              and (p.srcRoute = e.route)))) then
16           routeToUse = ∅;
17           for each node n ∈ {e.route[position + 1],...,
                 e.route[e.route.length − 1]} do
18             Search for a shortest cached route to n;
19             if such a route is found then
20               foundRoute := the found route;
21               if routeToUse = ∅ or |foundRoute| < |routeToUse| then
22                 routeToUse := foundRoute;
23                 tellID := n
24           if routeToUse ≠ ∅ then
25             notifyList := notifyList ∪ {(tellID, routeToUse)}
```

Figure 3.5: Pseudo Code for the Distributed Adaptive Cache Update Algorithm (Part I)

```
Algorithm:  cacheUpdate (continued)
26        elseif direction = backward then
27          if not isLastNode(e.route, netID) then
28            notifyList := notifyList ∪ {(e.route[position + 1],
                    (netID||e.route[position + 1]))}
29          routeToUse = ∅;
30          for each node n ∈ {e.route[position − 1], ..., e.route[0]} do
31            Search for a shortest cached route to n;
32            if such a route is found then
33              foundRoute := the found route;
34              if routeToUse = ∅ or |foundRoute| < |routeToUse| then
35                routeToUse := foundRoute;
36                tellID := n
37          if routeToUse ≠ ∅ then
38            notifyList := notifyList ∪ {(tellID, routeToUse)}
39      else /* The node receives a notification.*/
40        index := Index(e.route, netID);
41        if direction = forward and index < position and
            (not isFirstNode(e.route, netID)) then
42          notifyList := notifyList ∪
                  {(e.route[index − 1], (netID||e.route[index − 1]))}
43        if direction = backward and index > position and
             (not isLastNode(e.route, netID)) then
44          notifyList := notifyList ∪
                  {(e.route[index + 1], (netID||e.route[index + 1]))}
45        if (e.DP = 1 or e.DP = 2) and
            ((direction = forward and index > position) or
            (direction = backward and index < position)) then
46          if continue_to_notify then
47            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
48              notifyList := notifyList ∪
                      {(e.route[index + 1], (netID||e.route[index + 1]))}
```

Figure 3.6: Pseudo Code for the Distributed Adaptive Cache Update Algorithm (Part II)

```
Algorithm: cacheUpdate (continued)
49              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
50                  notifyList := notifyList ∪
                            {(e.route[index − 1], (netID||e.route[index − 1]))}
51            if not (netID = to or (direction = forward and
                  isLastNode(e.route, netID)) or (direction = backward and
                  isFirstNode(e.route, netID))) then
52                notifyList := notifyList ∪
                          {(e.route[index + 1], (netID||e.route[index + 1]))};
53                notifyList := notifyList ∪
                          {(e.route[index − 1], (netID||e.route[index − 1]))}
54        for each entry r ∈ e.replyRecord do
55          if link (from, to) ∈ e.replyRecord.subrouteSent or
              link (to, from) ∈ e.replyRecord.subrouteSent then
56            tellID := e.replyRecord.nodeNotified;
57            notifyList := notifyList ∪ {(tellID, (netID||tellID))};
58            e.replyRecord := e.replyRecord \ {r};
59        cacheTable := cacheTable \ {e};
60      else /* The route does not contain the broken link. */
61        for each entry r ∈ e.replyRecord do
62          if r.nodeNotified = to and netID = from then
63            e.replyRecord := e.replyRecord \ {r};
64  for each entry n ∈ notifyList do
65    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 notifyList := notifyList \ {n};
66  return notifyList;
```

Figure 3.7: Pseudo Code for the Distributed Adaptive Cache Update Algorithm (Part III)

recording that *B* will learn route *CDE*. Assume that *C* detects through another flow that link *CD* is broken. Node *C* does not need to notify *D* and *E* because they did not cache link *CD* when forwarding the ROUTE REPLY.

2. If *DataPackets* is 1 or 2, then the upstream nodes need to be notified, because at least one data packet has reached the node and hence the upstream nodes have cached the broken link. The algorithm adds the upstream neighbor to the notification list.

3. If *DataPackets* is 2, or if *DataPackets* is 1 and the route being examined is different from the source route in the packet, then the downstream nodes need to be notified, because at least one data packet has traversed the route and hence the downstream nodes have cached the link. I show an example for the second case. As shown in Figure 3.1, node *C* detects that *CD* is broken when using route *ABCDE*. In the table entry with route *FGCDH*, *DataPackets* is 1 or 2 since the route is post-active. Node *C* needs to notify *D* and *H* because they cached the link when forwarding the first data packet. The algorithm searches the cache to find a shortest route to reach one of the downstream nodes. If it finds such a route, it adds that downstream node to the list. If *DataPackets* is 1 and the route being examined is the same as the source route in the packet, then no downstream node needs to be notified, because the first data packet cannot be delivered and hence downstream nodes did not cache the link through forwarding packets with this route.

If a route contains a *backward* link (lines: 26–38), which means the link to the previous hop in the route is broken, the algorithm adds the downstream neighbor to the list. Since the node has forwarded at least one data packet using the route, the downstream nodes have cached that link. The upstream nodes also need to be notified. The algorithm searches the cache to find a shortest route to reach one of the upstream nodes. If it finds such a route, it adds that upstream node to the notification list.

When a node detects a link failure, the algorithm does the above operation to add the closest upstream and/or downstream nodes to the list. When a node learns through

a notification that a link is broken, it is responsible for notifying its upstream and/or downstream neighbors. It determines the neighbors to be notified based on the position of the node in a route and whether the link is *forward* or *backward* (lines: 39–53):

1. If the link is a *forward* link, and the node is upstream to it but not the source node, then the algorithm adds the upstream neighbor to the notification list. If the link is a *backward* link, and the node is downstream to it but not the destination, then the algorithm adds the downstream neighbor to the notification list.

2. If the link is a *forward* link, and the node is downstream to it and receives a notification from the upstream endpoint of the broken link, then there are three cases: (1) If the node is the other endpoint of the link, then the algorithm adds its downstream neighbor to the notification list; (2) If the node is the destination, then the algorithm adds its upstream neighbor to the notification list; (3) Otherwise, the algorithm adds both the upstream and downstream neighbors to the notification list.

3. If the link is a *backward* link, and the node is upstream to it and receives a notification from the downstream endpoint of the broken link, then there are three cases: (1) If the node is the other endpoint of the link, then the algorithm adds its upstream neighbor to the notification list; (2) If the node is the source, then the algorithm adds its downstream neighbor to the notification list; (3) Otherwise, the algorithm adds both the upstream and downstream neighbors to the notification list.

After adding upstream and/or downstream neighbors to the list, the algorithm checks the *ReplyRecord* field. If an entry contains a broken link, the algorithm adds the neighbor that learned the link to the list (lines: 54–58). The algorithm then removes the cache entry containing the broken link (line: 59). If a node detects a link failure when attempting to send a ROUTE REPLY, the algorithm removes the corresponding *ReplyRecord* entry (lines: 61–63). Finally, the algorithm removes duplicate nodes from the list. Duplicate nodes may occur in the list when the node is on multiple routes containing a broken

link. The algorithm also removes the node that is the source node of a notification, since the algorithm adds both upstream and downstream neighbors to the list for the node that receives a notification from its upstream or downstream neighbor (lines: 51–53).

Each node receiving a notification notifies its selected neighbors about the broken link. Therefore, if a route contains a *forward* link, notifications will be propagated among the upstream nodes towards the source, and among the downstream nodes towards the destination and/or towards the downstream endpoint of the broken link. If a route contains a *backward* link, notifications will be propagated among the downstream nodes towards the destination, and among the upstream nodes towards the source and/or towards the upstream endpoint of the broken link. Notifications will also be propagated to the nodes that learned routes containing a broken link from ROUTE REPLIES.

Examples

Example 1 Figure 3.8 shows an example where only flow 1 starts. The focus of this example is the *DataPackets* field. Assume that node *A* initiates a route discovery to node *E* and receives a ROUTE REPLY with the source route, *ABCDE*. Before any data packet from flow 1 reaches node *C*, node *C* detects through another flow or a control packet that the link from node *C* to node *D* is broken.

	Route	S-D	DP	ReplyRecord
C :	*CDE*	*A E*	0	*B* ← *CDE*

Since *DataPackets* is 0, the algorithm knows that route *CDE* is a pre-active route, and therefore no downstream nodes need to be notified. The algorithm then checks the *ReplyRecord* field for possible neighbors that have learned the broken link through ROUTE REPLIES (*cacheUpdate*: 54–58). It removes the *ReplyRecord* entry and notifies the neighbor *B*. Node *B* also cached a pre-active route, *BCDE*. According to the

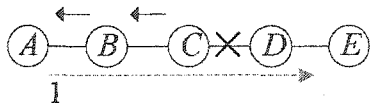


Figure 3.8: Example 1 (Case 1 and Case 2)

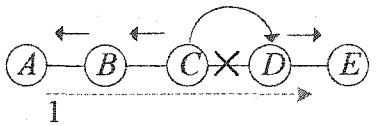


Figure 3.9: Example 1 (Case 3)

ReplyRecord field in node B's cache, node B removes the stale route and notifies node A (*cacheUpdate*: 54–58). Finally, node A removes the stale route from its cache.

Assume that node C detects that the link from node C to node D is broken while attempting to transmit a data packet for flow 1. Node C's cache is:

C :

Route	S-D	DP
ABCDE	*A E*	*d*

Here $d = 1$ or $d = 2$. For either case, upstream nodes need to be notified. The algorithm adds only the upstream neighbor B to the notification list (*cacheUpdate*: 9–14). The algorithm then determines whether it needs to notify the downstream nodes (*cacheUpdate*: 15–25). If $d = 1$ and the route being examined is the same as the source route in the packet, indicting that the packet is the first data packet, then downstream nodes do not need to be notified. Thus, node C notifies only node B (*cacheUpdate*: 39–42). When B receives the notification, the algorithm checks the cache entries containing the broken link (*cacheUpdate*: 1–10). The algorithm determines which upstream and/or downstream neighbors need to be notified based on the position of the node in the route and whether the link is a *forward* or *backward* link. Since the broken link is a *for-*

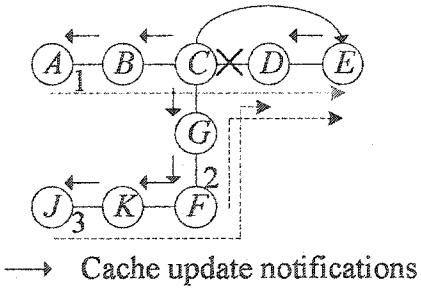


Figure 3.10: Example 2

ward link and node B is upstream to it, the algorithm notifies its upstream neighbor A (*cacheUpdate*: 39–42). This is the second case, which is also shown in Figure 3.8.

If $d = 1$ and the route being examined is different from the source route in the packet, then downstream nodes need to be notified, since one data packet carrying the broken link has traversed that route. If $d = 2$, indicting that node C is forwarding *at least* the second data packet, then downstream nodes also need to be notified. For either case, the algorithm attempts to find a shortest route to reach one of the downstream nodes (*cacheUpdate*: 11–25). Assume that the algorithm finds a route to node D, so it adds D to the list. The algorithm then checks the *ReplyRecord* field (*cacheUpdate*: 54–59). As the cache table entry does not contain any *ReplyRecord* entry, no other neighbors need to be notified. Finally, the algorithm removes the cache table entry and sends notifications to node B and node D, as shown in Figure 3.9. When B receives a notification, it starts the algorithm to notify its upstream neighbor A (*cacheUpdate*: 39–42). When node D receives a notification, it notifies its downstream neighbor E (*cacheUpdate*: 45–48).

Example 2 Figure 3.10 shows another example. The focus of this example is the *ReplyRecord* field. Assume that node A discovers route r_1, *ABCDE*; node F discovers route r_2, *FGCDE*; and node J discovers route r_3, *JKFGCD*. Also assume that route r_1 is active, and both route r_2 and route r_3 are pre-active. When transmitting a packet using route r_1, node C detects that the link from node C to node D is broken.

	Route	S-D	DP	ReplyRecord
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	

For route r_1, node C needs to notify its upstream neighbor B and the closest reachable downstream node, since *DataPackets* is 2. Assume that node C does not have a cached route to reach node D but finds a route to reach node E, so the algorithm adds E to its notification list (*cacheUpdate*: 9–25). The algorithm then checks the *ReplyRecord* field and finds that node G learned a route containing the broken link (*cacheUpdate*: 54–58). Thus, the algorithm adds G to the list and sends notifications to B, E, and G. When B receives a notification, it starts the algorithm to notify node A. When node E receives a notification, it starts the algorithm. The algorithm adds node D to the list, since the broken link is a *forward* link and node E is the destination (*cacheUpdate*: 49–50). When node G receives a notification, it starts the algorithm to notify node F according to the *ReplyRecord* field in its cache. Similarly, node F notifies node K, and node K notifies node J. Thus, stale pre-active routes will be quickly removed.

Example 3 This example shows how the algorithm handles a *backward* link. As shown in Figure 3.11, assume that route r_1 is post-active, and route r_4, *HICBA*, is active. While transmitting a packet for flow 4, node C detects that the link from node C to node B is broken.

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

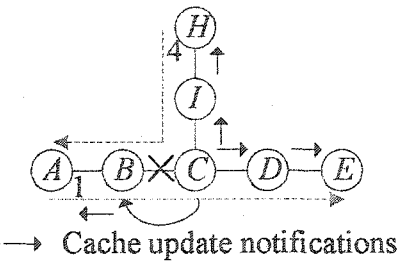


Figure 3.11: Example 3

For route r_4, node C needs to notify its upstream neighbor I and the closest reachable downstream node. For route r_1, the link from node C to node B is a *backward* link, and thus node C needs to notify its downstream neighbor D and the closest reachable upstream node (*cacheUpdate*: 26–38). Assume that the algorithm finds a route to reach node B, so it sends notifications to I, D, and B. Node I notifies node H as the broken link is a *forward* link and node I is upstream to the link. When node D receives a notification, it starts the algorithm to notify node E, since the broken link is a *backward* link and node D is downstream to the link (*cacheUpdate*: 43–44). When node B receives a notification, it starts the algorithm. For route r_4, the algorithm adds the downstream neighbor A to the list (*cacheUpdate*: 45–48), since the broken link is a *forward* link and node B is downstream to the link. For route r_1, the algorithm adds the upstream neighbor A to the list, since the broken link is a *backward* link and node B is upstream to the link. Finally, the algorithm sends a notification to node A.

Example 4 For the *forward* link on route r_1 in Figure 3.10, notifications are propagated among the upstream nodes towards the source, and among the downstream nodes towards the downstream endpoint of the broken link. For the *backward* link on route r_1 in Figure 3.11, notifications are propagated among the downstream nodes towards the destination, and among the upstream nodes towards the source. This example shows a situation in which notifications will be propagated in two directions either among down-

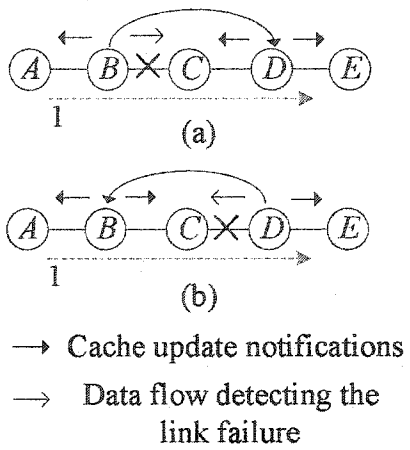


Figure 3.12: Example 4

stream nodes for a *forward* link, or among upstream nodes for a *backward* link.

In Figure 3.12 (a), the link from node B to node C is a *forward* link. Assume that *DataPackets* is 2 in node B's cache, and node B finds a shortest path to reach node D, so it sends notifications to node A and node D. When node D receives a notification, it starts the algorithm. The algorithm adds both the upstream neighbor C and the downstream neighbor E to the list (*cacheUpdate*: 51–53).

In Figure 3.12 (b), the link from node D to node C is detected as broken by another flow and thus is a *backward* link. Assume that node D finds a shortest path to reach node B and sends a notification to node B. When node B receives a notification, it starts the algorithm. The algorithm adds both the upstream neighbor A and the downstream neighbor C to the list (*cacheUpdate*: 51–53).

3.3.7 Correctness

In this section, I prove the correctness of my algorithm.

Definition Let node u be the node that detects a link failure. Node v is *reachable* from node u if node u has a cached route to node v, or node u has a cached route to some

intermediate node and that intermediate node has a cached route to node v.

Theorem 3.1 *When a node detects a link failure, the algorithm notifies all reachable nodes that have cached the link about the link failure.*

Proof In a network with many nodes, suppose that node i on a route with n nodes detects that link $(i, i+1)$ is broken. This route is shown in Figure 3.13. Let R be a set of nodes that have cached the broken link and are reachable from node i. Nodes in R must learn the link either from data packets or from ROUTE REPLIES. If a node in R learned the link from a data packet, then the node has cached a complete source route, which is either active or post-active. If a node in R learned the link from a ROUTE REPLY, then the node has cached the route starting from itself to a destination, which is pre-active. Thus, all nodes in R must be on pre-active, active, or post-active routes.

Node i may have cached multiple routes containing the broken link. These routes contain either a *forward* link or a *backward* link: a link is either $(i, i+1)$ or $(i+1, i)$ on a route. For the current route, the broken link is a *forward* link and *DataPackets* is either 1 or 2. If *DataPackets* is 1, the algorithm notifies only node $i-1$ $(i \neq 1)$. If *DataPackets* is 2, the algorithm notifies both node $i-1$ $(i \neq 1)$ and the closest reachable downstream node, say node j, where $i+1 \leq j \leq n$. For other routes containing a *forward* link, the *DataPackets* is either 1 or 2 since at least one data packet has traversed the route. For either case, the algorithm notifies both the upstream neighbor and the closest reachable downstream node. For routes containing a *backward* link, the algorithm notifies the downstream neighbor and the closest reachable upstream node. According to the *ReplyRecord* field, the algorithm also notifies the neighbors that learned the link through ROUTE REPLIES. Thus, all neighbors of node i that have cached the broken link are notified about the link failure.

Each node receiving a notification starts the algorithm to notify the upstream and/or downstream neighbors for each route containing the broken link. For the route shown

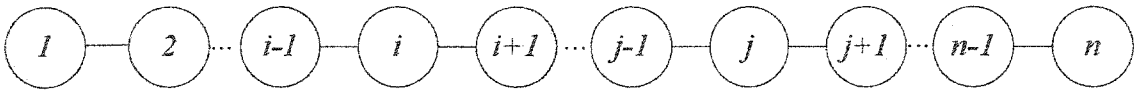


Figure 3.13: A route with n nodes

in Figure 3.13, node $i-1$ notifies its upstream neighbor $i-2$ ($i \neq 2$). For node j, there are three cases: (1) If $j = i+1$, the algorithm notifies its downstream neighbor $j+1$; (2) If $i+1 < j < n$, the algorithm notifies both its upstream neighbor $j-1$ and downstream neighbor $j+1$; (3) If $j = n$, the algorithm notifies its upstream neighbor $n-1$. When node $i-2$ receives a notification, the algorithm notifies its upstream neighbor. Thus, all upstream nodes are notified. When node $j+1$ receives a notification, the algorithm notifies its downstream neighbor $j+2$ ($j+2 \leq n$). When node $j-1$ receives the notification, the algorithm notifies its upstream neighbor $j-2$ ($j-2 \geq i+1$). Thus, all downstream nodes are notified. Similarly, each node on other routes that receives a notification notifies its upstream and/or downstream neighbors. Thus, all nodes in R that are either on active or on post-active routes are notified. If a node receiving a notification sent a ROUTE REPLY containing the broken link, the algorithm notifies the neighbor that learned the link. Thus, all nodes in R that are on pre-active routes are notified.

Now I show when the algorithm terminates. For a route with a *forward* link, the algorithm terminates at the source node, the downstream endpoint of the broken link and/or the destination. For a route with a *backward* link, the algorithm terminates at the destination, the upstream endpoint of the broken link and/or the source node. The algorithm also terminates at a source node that has cached a stale pre-active route.

Some nodes that have cached a broken link may not be reachable in four cases. First, a node detecting a link failure may not have a cached route to any downstream node. Second, a notification may encounter a broken link, and the node detecting the link failure may not have a cached route to salvage the notification. Third, some nodes that have cached a broken link may not be reachable due to a node failure. If a node crashes

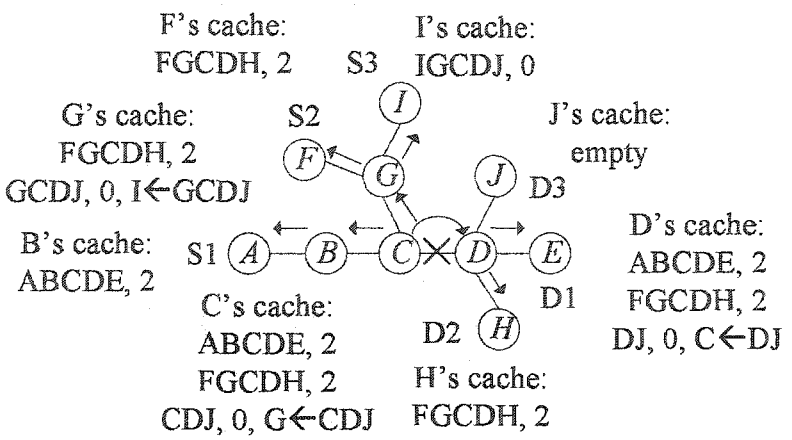


Figure 3.14: Distributed Cache Updating for the Example shown in Figure 3.1

and reboots, the information in its cache table will be lost; therefore, it will not be able to notify its neighbors that may have cached a broken link. Finally, if the cache size is set with some limit, for example because the node has only a very small amount of memory, it is possible that not all nodes on a route have the route in their caches. As a result, a notification cannot be propagated to some nodes that have cached a broken link. For all cases, the algorithm will notify those nodes that previously cannot be reached when a flow detects the link failure through normal on-demand Route Maintenance. Thus, node failures or a limited cache size do not affect the correctness of the algorithm. ∎

3.3.8 Algorithm Summary

The algorithm has achieved the design goal: it notifies all reachable nodes that have cached a broken link. For the example shown in Figure 3.1, in which route *ABCDE* is active, route *FGCDH* is post-active, and route *IGCDJ* is pre-active, the algorithm notifies all nodes that need to be notified (assume that node *C* has a cached route to reach node *D*), as shown in Figure 3.14.

My algorithm enables DSR to quickly remove stale routes, thus reducing packet losses, delivery latency, and routing overhead. These benefits will become more significant as mobility, traffic load, or network size increases. As mobility increases, routes break more frequently. As traffic load increases, stale routes affect more traffic sources: without proactive cache updating, each flow has to detect a broken route on-demand. As network size increases, more nodes will cache stale routes. Thus, proactive cache updating provides more advantages under more challenging network characteristics. Since the algorithm informs only the nodes that have cached a broken link, cache update overhead is low. Since the algorithm does not use any ad hoc parameters, it makes DSR fully adaptive to topology changes.

3.3.9 Implementation Decisions

I use two optimizations for my algorithm. First, to reduce duplicate notifications to a node, I attach a *reference list* to each notification. The node detecting a link failure is the root, initializing the list to be its notification list. Each child notifies only the nodes not in the list and updates the list by adding the nodes in its notification list. The graph will be close to a tree. Second, I piggyback a notification on the data packet that encounters a broken link if that packet can be salvaged.

When using the algorithm, I also use a small list of broken links, which is similar to the negative cache proposed in prior work [30], to prevent a node's cache from being re-polluted by in-flight stale routes. This component is not a part of the algorithm. The size of the list is 5 and the timeout is set to 2 s. This list can be replaced by a non-ad-hoc technique proposed by Hu and Johnson [24].

3.3.10 Working with Promiscuous Mode

To handle situations where promiscuous mode is used, I combine my algorithm and the secondary cache used in DSR with path caches, without any modification to the algorithm. This section presents this implementation.

When using promiscuous mode, DSR with path caches uses a secondary cache to store the routes a node overhears. If a route in the secondary cache is used to respond to a ROUTE REQUEST or to send packets, it will be added to a primary cache. DSR does not distinguish whether the route contained in a ROUTE REPLY is an overhead route, and stores overhead routes in ROUTE REPLIES in the primary cache.

I use the secondary cache to store both the routes a node overhears and the overheard routes learned from ROUTE REPLIES. The second type of overheard routes is not added to the cache table because the algorithm does not keep track of which neighbor overhears which link, but completely maintains topology propagation state in cache tables. If an intermediate node sends a ROUTE REPLY using an overhead route, it marks a flag in the packet, so that the node forwarding the ROUTE REPLY stores the downstream links of the source route in its secondary cache.

An overheard route in a secondary cache will be added to a cache table when a source node uses the route to send data packets. Each node on the route will add the route to its cache table. The algorithm begins to track which node caches which link of the route. An overheard route in the secondary cache will also be evicted by FIFO replacement or through overhearing ROUTE ERRORS.

3.4 Performance Evaluation

3.4.1 Evaluation Methodology

I compared my algorithm called DSR-Update with DSR with path caches and with *Link-MaxLife* under both promiscuous and non-promiscuous mode. When promiscuous mode (also called tapping) was not used, I did not use GRATUITOUS ROUTE REPLIES since it relies on this mode. For DSR-Update under non-promiscuous mode, I did not use GRATUITOUS ROUTE ERRORS, since I wanted to use the algorithm as the only mechanism to remove stale routes. When promiscuous mode was used, I used all optimizations for the three caching strategies.

I used the *ns-2* [12] network simulator with the Monarch Project's wireless extensions [6, 47]. The network interface is modelled after Lucent's WaveLAN, which provides a 2 Mbps transmission rate and a nominal transmission range of 250 m. The network interface uses IEEE 802.11 Distributed Coordination Function (DCF) MAC protocol [25]. The mobility model is *random waypoint model* [6]. In this model, a node starts in a random position, picks a random destination, moves to it at a randomly chosen speed, and pauses for a specified pause time. Node speed was randomly chosen from 10 ± 1 m/s. I used two field configurations: a 1500 m $\times$ 500 m field with 50 nodes and a 2200 m $\times$ 600 m field with 100 nodes. I used CBR traffic with four packets per second and packet size of 64 bytes to factor out the effect of congestion [6]. I used 20 and 40 flows for 50-node scenarios and 20 flows for 100-node scenarios. Simulations ran for 900 s of simulated time. Each data point in the graphs represents an average of 10 runs of randomly generated scenarios. I will use labels such as "50n-20f" for 50 node and 20 flow scenarios, etc.

I used four metrics: (1) *Packet Delivery Ratio*: the ratio of the number of data packets received by the destination to the number of data packets sent by the source; (2) *Packet*

Delivery Latency: the delay from when a packet is sent by the source until it is received by the destination; (3) *Percentage of Good Replies Sent from Caches*: the percentage of ROUTE REPLIES sent by intermediate nodes that do not contain broken links; (4) *Packet Overhead*: the total number of routing packets transmitted; and (5) *Normalized Routing Overhead*: the ratio of the number of routing packets transmitted to the number of data packets received. For DSR-Update, packet overhead and normalized routing overhead include ROUTE ERRORS used for cache updating.

3.4.2 Simulation Results

Packet Delivery Ratio

Figure 3.15 shows packet delivery ratio and Figure 3.16 shows the percentage of good ROUTE REPLIES sent from caches. Without promiscuous mode, DSR-Update outperforms DSR by up to 19% and *Link-MaxLife* by up to 41%. The improvement demonstrates that proactive cache updating is more efficient than FIFO and predicting timeouts. Moreover, the improvement increases as mobility, traffic load, or network size increases. As mobility increases, more routes become stale; therefore, the advantages of fast cache updating become more significant. As traffic load increases, proactive cache updating reduces packet losses from more traffic sources. Proactive cache updating is important for large networks because as network size increases, more nodes will cache stale routes.

Link-MaxLife performs better than DSR with path caches under high mobility and low traffic load, because it expires links aggressively when links break more frequently. It performs worse than DSR-Update, especially for high traffic load and large networks because of worse cache performance. For example, for 50n-40f at pause time 0 s, the percentage of good ROUTE REPLIES sent from caches is 68% for DSR-Update and 51%

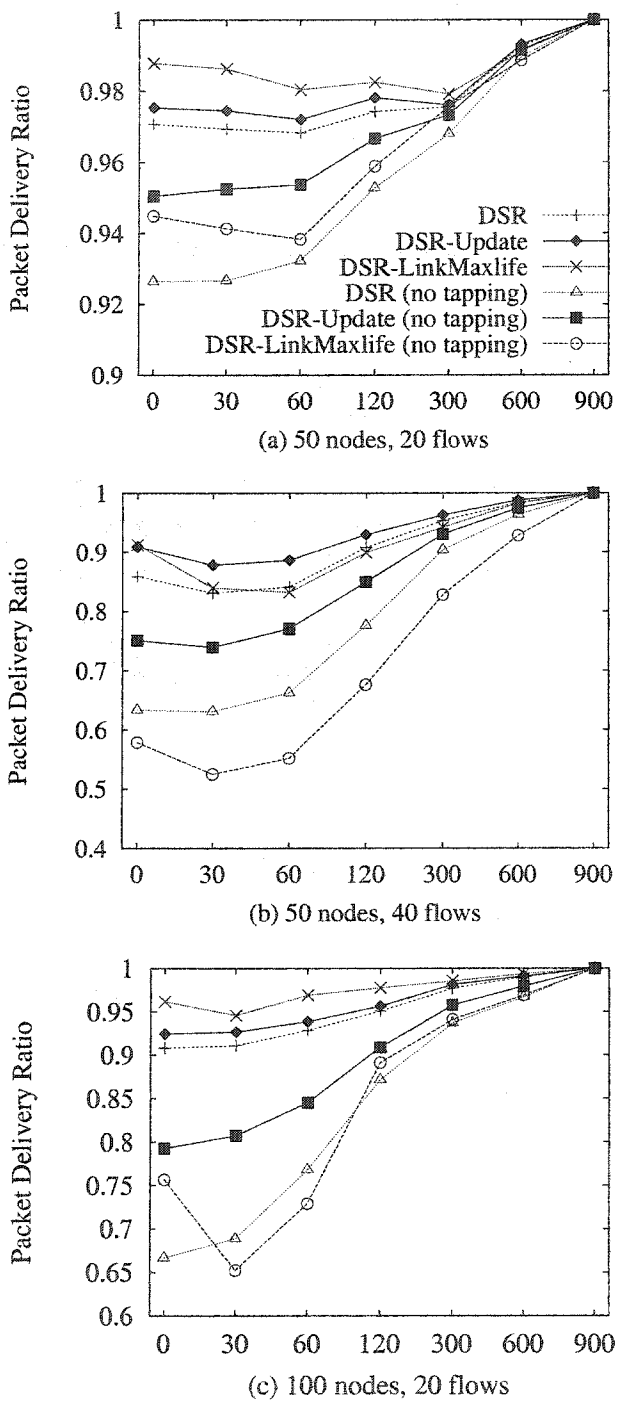


Figure 3.15: Packet Delivery Ratio vs. Mobility (Pause Time (s))

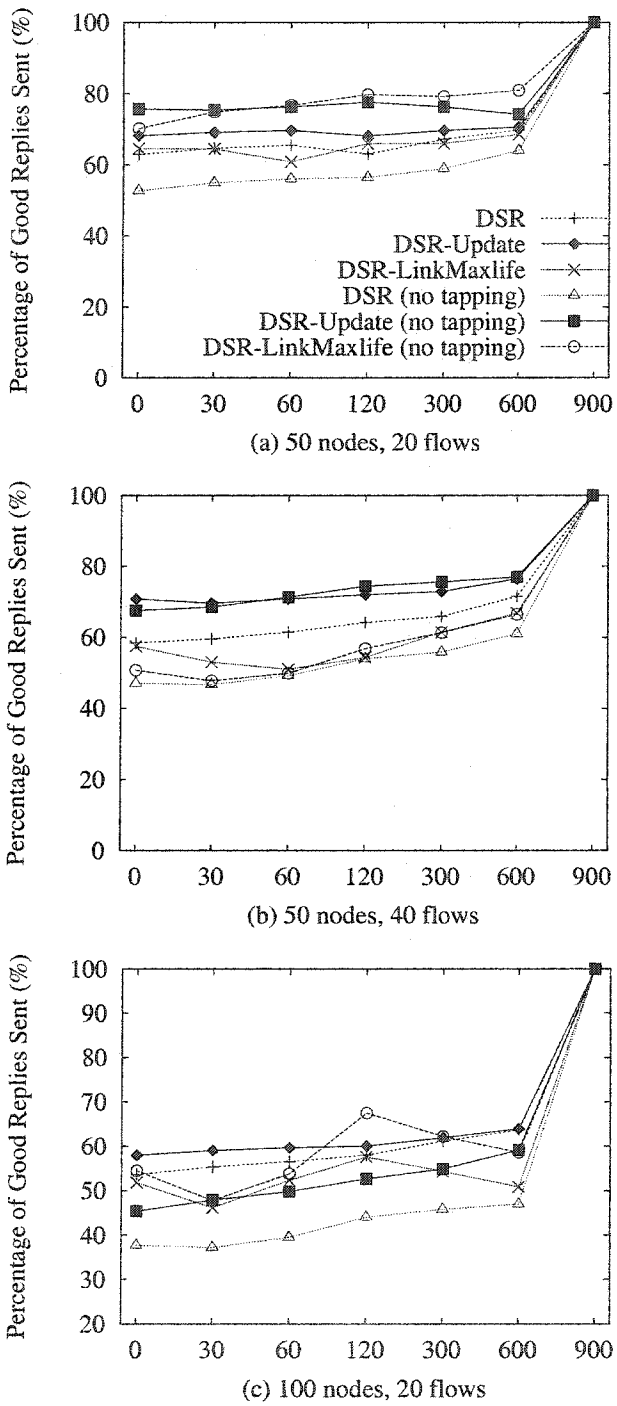


Figure 3.16: Percentage of Good Replies Sent from Caches vs. Mobility (Pause Time (s))

for *Link-MaxLife*. Under promiscuous mode, *Link-MaxLife* performs better than DSR-Update under low traffic load, since it caches more overheard routes in the topology graph, but performs worse than DSR-Update under high traffic load.

Compared with DSR under promiscuous mode, DSR-Update performs better but does not give as much improvement as it does under non-promiscuous mode. DSR-Update stores overheard routes learned from ROUTE REPLIES in a secondary cache, whereas DSR stores such routes in a primary cache. Thus, DSR benefits more from promiscuous mode than DSR-Update by storing more overheard routes. But DSR-Update achieves the improvement of 7% under high traffic load.

Since packet delivery ratio is affected by both cache performance and promiscuous mode, the results without promiscuous mode allow the effect of different caching strategies on this metric to be observed.

Packet Delivery Latency

Figure 3.17 and Figure 3.18 show packet delivery latency. Without promiscuous mode, DSR-Update reduces latency by up to 54% of DSR. Since detecting a link failure is the dominant factor of latency, the reduction demonstrates the effectiveness of the algorithm. The reduction increases as mobility, traffic load, or network size increases because quick removing stale routes reduces link failure detections by multiple flows.

DSR-Update has lower latency than *Link-MaxLife* in most cases of 100 node scenarios. Although link caches help reduce latency due to fewer route discoveries, *Link-MaxLife* has higher overall latency because packets are salvaged multiple times due to stale links. Under promiscuous mode, the reduction becomes more significant. For example, for 100n-20f, the maximum reduction is 49%. At pause time 30 s, where the maximum reduction is achieved, the percentage of good ROUTE REPLIES sent from caches is 46% for *Link-MaxLife* and 59% for DSR-Update.

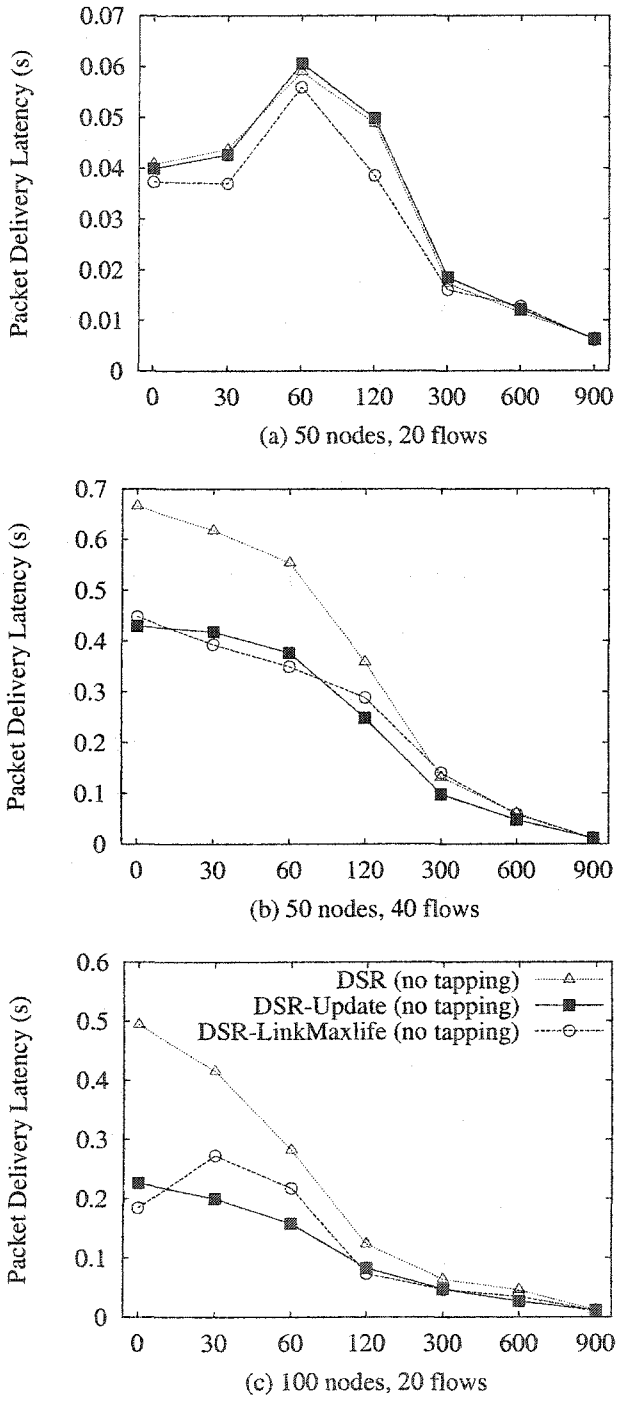


Figure 3.17: Packet Delivery Latency vs. Mobility (Pause Time (s))

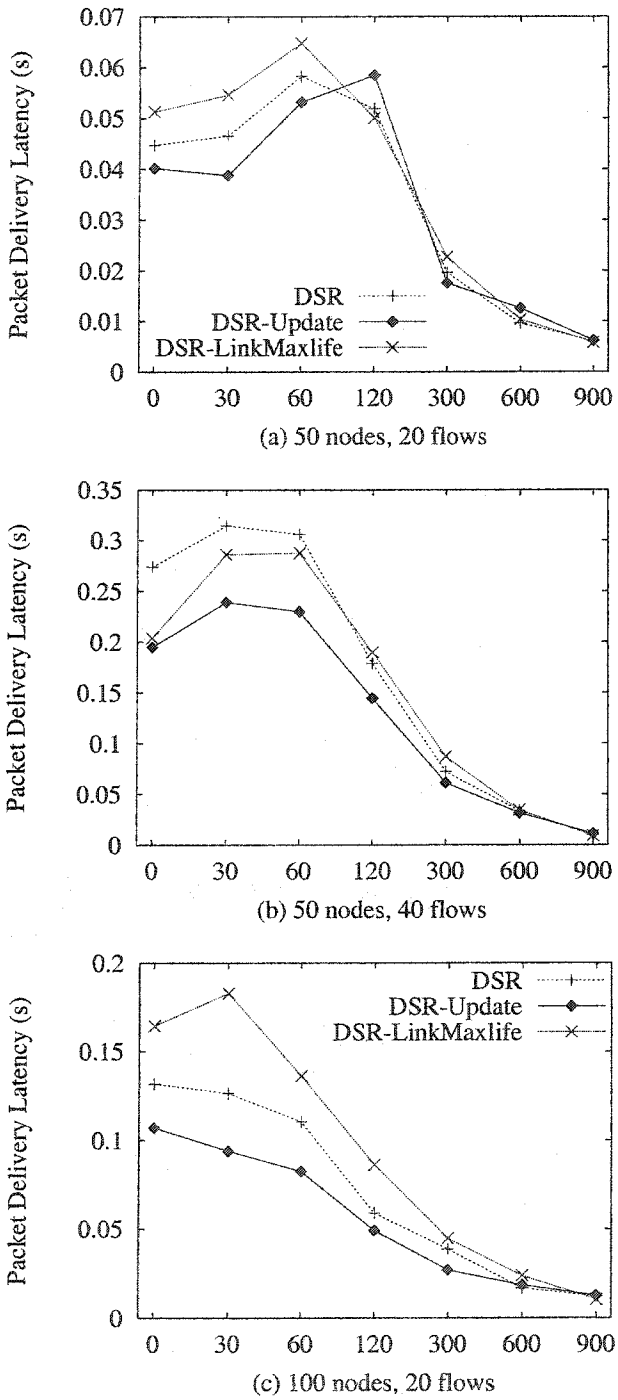


Figure 3.18: Packet Delivery Latency vs. Mobility (Pause Time (s))

Compared with DSR under promiscuous mode, DSR-Update reduces latency up to 27% for 100n-20f. The higher latency in DSR also results from worse cac performance. For example, for 50n-40f at pause time 0 s, the percentage of good ROU REPLIES sent from caches is 71% for DSR-Update and 58% for DSR.

Overhead

Although the algorithm introduces cache update overhead, it reduces ROUTE ERRO caused by stale routes. Overall, under non-promiscuous mode, DSR-Update uses mu fewer ROUTE ERRORS to maintain caches than both DSR and *Link-MaxLife*, since t latter two protocols also rely on GRATUITOUS ROUTE ERRORS to remove stale rout Under promiscuous mode, DSR-Update uses slightly more ROUTE ERRORS than DS for 50 node scenarios and a similar number of ROUTE ERRORS for 100 node scenari

Figure 3.19 and Figure 3.20 show packet overhead and normalized routing overhe Without promiscuous mode, DSR-Update has slightly higher packet overhead than DS for 50-node scenarios, but achieves a large reduction for 100-node scenarios. The hi overhead in DSR with path caches results from a large number of route discoveri caused by the small cache size. Under promiscuous mode, there is not much differen in overhead, since DSR uses a secondary cache to store more routes. DSR-Update h similar normalized routing overhead as DSR under promiscuous mode. Under no promiscuous mode, it achieves a reduction by up to 35% for DSR under high traffic lo and large networks.

Analysis of the Cache Size

The cache table size dynamically changes as needed. I define the average cache size the average over the size sampled when a route is added or when at least one route deleted. Under non-promiscuous mode, the average cache size at pause time 0 s is 10 f

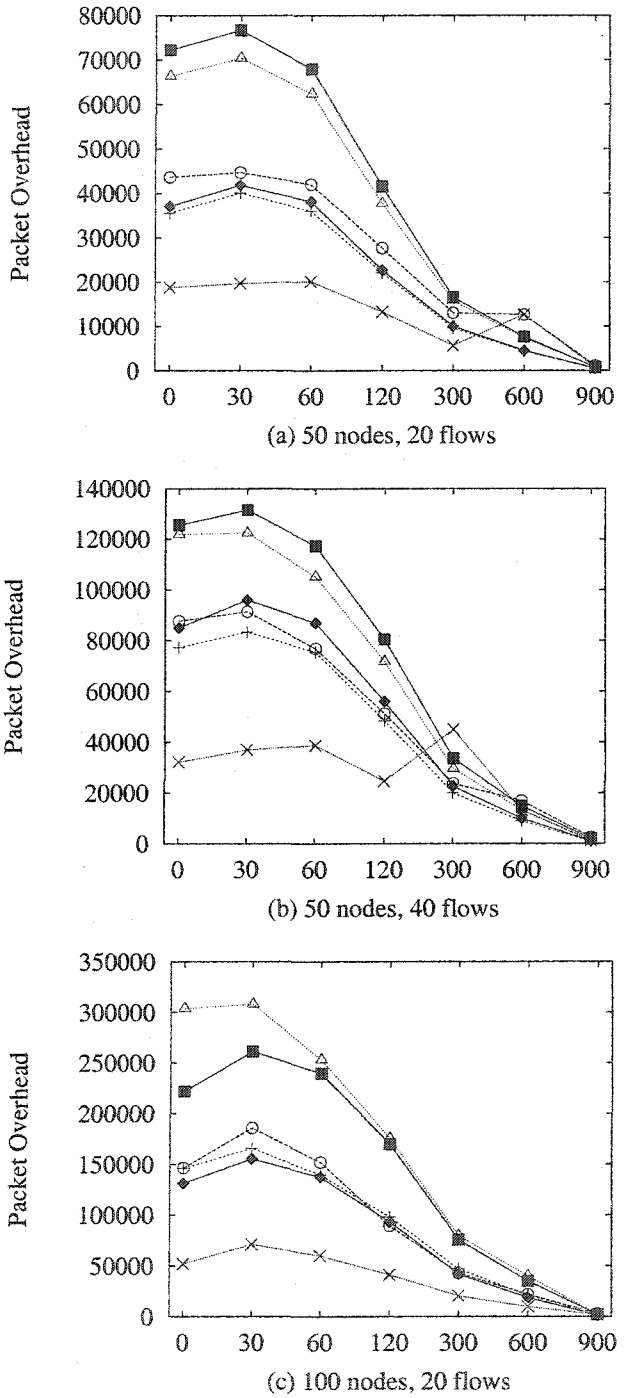


Figure 3.19: Packet Overhead vs. Mobility (Pause Time (s))

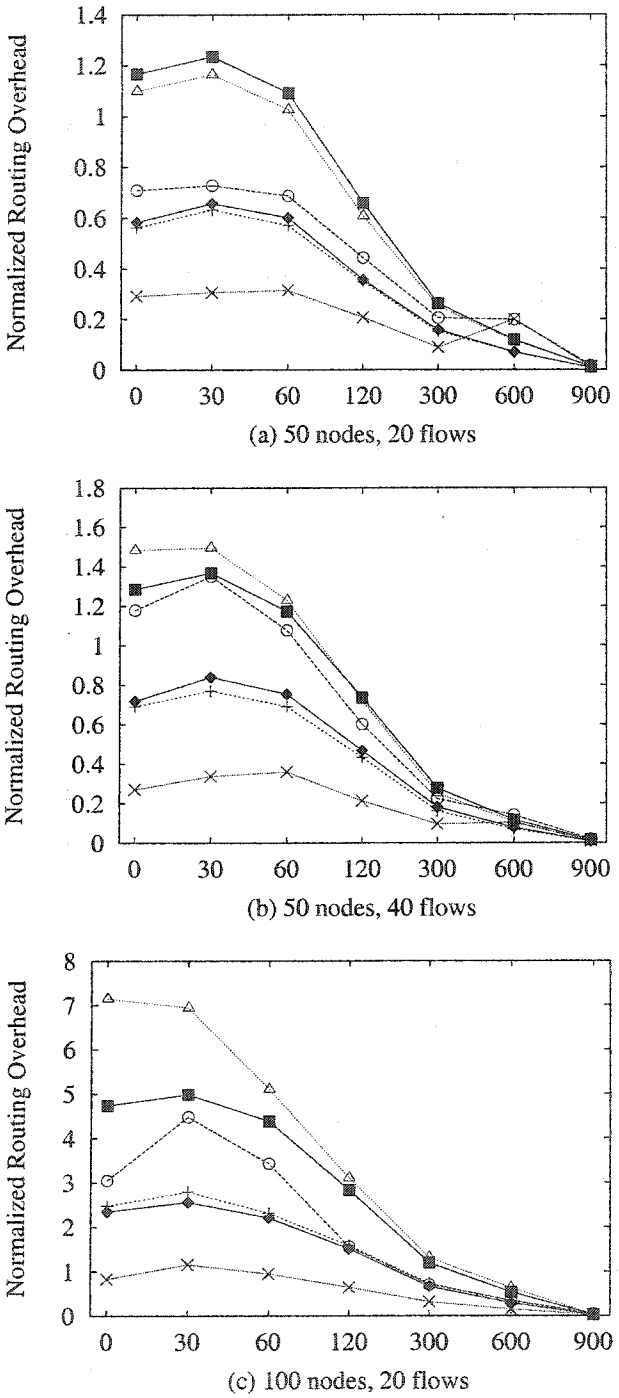


Figure 3.20: Normalized Routing Overhead vs. Mobility (Pause Time (s))

50n-20f, 20 for 50n-40f, and 18 for 100n-20f. Under promiscuous mode, the average cache size at pause time 0 s is 8 for 50n-20f, 15 for 50n-40f, and 12 for 100n-20f. Thus, the cache size increases as traffic load or network size increases. It also increases as mobility increases because more route discoveries take place. I also measured the maximum cache size. Under non-promiscuous mode, the maximum cache size at pause time 0 s is 68 for 50n-20f, 80 for 50n-40f, and 112 for 100n-20f. Under promiscuous mode, the maximum cache size at pause time 0 s is 50 for 50n-20f, 96 for 50n-40f, and 86 for 100n-20f. The maximum cache size also decreases as mobility decreases.

3.5 Conclusions

In this chapter, I have presented the first work that proactively updates route caches in an adaptive manner. I have defined a new cache structure called a cache table to maintain the information necessary for cache updates. Based on the local information kept by each node, my cache update algorithm disseminates the broken link information to all reachable nodes that have cached the link in a distributed manner. Therefore, my algorithm enables DSR to adapt quickly to topology changes.

I show that, under non-promiscuous mode, the algorithm outperforms DSR with path caches by up to 19% and *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 latency by up to 27% for DSR with path caches and 49% for *Link-MaxLife*. The improvement demonstrates the benefits of the algorithm.

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. Simulation results show that proactive cache updating is more efficient than adaptive timeout mechanisms. This work combines the advantages of proactive and on-demand protocols: on-demand link failure detection and proactive cache updating. My solution is applicable to other on-demand routing protocols. I conclude that proactive cache updating is key to the adaptation of on-demand routing protocols to mobility.

Chapter 4

Reducing the Effect of Mobility on TCP by Proactive Cache Updating

As discussed in Chapter 1, TCP performance is adversely affected by frequent route failures. Most recent attempts to improve TCP performance focus on transport layer mechanisms. Several modifications to TCP were proposed to prevent TCP from invoking congestion control mechanisms for packet losses caused by route failures. Chapter 3 presented a distributed cache update algorithm for DSR. In this chapter, I investigate the impact of the algorithm on TCP performance, without any modification to TCP.

4.1 Introduction

On-demand routing protocols use route caches to reduce the cost of route discoveries. Stale routes present a serious challenge to TCP [21, 10]. Stale routes cause route failures and packet losses. For such losses, TCP will invoke congestion control mechanisms, resulting in the reduction in throughput. If stale routes are not removed quickly, TCP may retransmit lost packets still using stale routes, resulting in repeated timeouts. Although

several caching strategies [23, 39, 37, 24] have been proposed to improve cache correctness of DSR, their impact on TCP performance has not been studied.

Although route failures are inherent due to mobility, not all of them are unavoidable. Route failures can be classified into two cases. In the first case, a route is valid when the source node uses it, but some link breaks during the transmission of a packet; such route failures are unavoidable. In the second case, a route has broken when a source node uses it; such route failures can be reduced by making route caches up-to-date.

The closer route caches track topology changes, the fewer route failures there will be. My distributed cache update algorithm makes route caches in DSR adapt quickly to topology changes. When a link failure is detected, the algorithm proactively notifies all reachable nodes that have cached a broken link. Proactive cache updating reduces route failures and consequent packet losses. Fast cache updating also prevents stale routes from being propagated to the caches of other nodes.

I investigate the impact of the algorithm on TCP performance by comparing DSR with the algorithm to DSR with path caches. Simulation results show that the algorithm significantly improves TCP throughput. For example, the algorithm improves throughput by 134% for 50 node scenarios at node mean speed of 15 m/s and by 193% for 100 node scenarios at node mean speed of 20 m/s. Moreover, the improvement increases as mobility, traffic load, and network size increase. The algorithm also significantly reduces normalized routing overhead. For example, it reduces overhead by up to 27% for 50-node and up to 89% for 100-node scenarios.

4.2 Mobility, Route Caches, and TCP

In this section, I discuss the effect of mobility on TCP and the effect of proactive cache updating on TCP.

4.2.1 The Effect of Mobility on TCP

The effect of mobility on TCP can be either *direct* or *indirect*. The former means that a route breaks although it was valid when the source node selected it, resulting in unavoidable route failures. The latter means that TCP is indirectly affected by mobility through stale routes. It was reported [51] that more than 80% of packet losses are due to route failures; as a result, TCP spends more than 50% time in slow-start.

The effect of stale routes on TCP is summarized as follows:

- *Data and ACK losses.* Each route failure will result in up to a congestion window number of data losses. Moreover, stale routes will cause ACK losses in the reverse path, and therefore TCP has to wait for timeouts, since TCP relies on the arrival of an ACK to trigger further transmission.

- *Increased latency.* Link failure detection through several retransmissions increases packet delivery latency if packets can be salvaged by intermediate nodes. Since only the source node is notified about a link failure, other TCP senders that have cached the broken link will have to detect the link failure themselves.

- *Increased MAC layer contentions.* ROUTE ERRORS caused by the use of stale routes can interfere with the transmission of data packets and ACKs.

TCP is very sensitive to *timeliness*: even a small delay in the transmission of data packets or the receipt of ACKs will easily result in timeouts at a TCP sender. Therefore, it is important to make route caches up-to-date as much as possible.

4.2.2 The Effect of Proactive Cache Updating on TCP

Through proactive link failure feedback, TCP senders and receivers will learn the information about a broken link at the earliest possible time, so that they use other cached

routes or initiate route discoveries earlier. As a result, packet losses and the latency due to link failure detection are reduced. The benefits of timely awareness of topology changes become significant when there are multiple TCP connections from multiple sources. I will validate these claims in the next section.

4.3 Performance Evaluation

4.3.1 Simulation Environment

I evaluated the performance of TCP with DSR augmented with DSR-Update and compared it with DSR with path caches through detailed simulations. I did not use promiscuous mode and thus did not use two optimizations, GRATUITOUS ROUTE REPLY and tapping, which rely on this mode. I used all other standard optimizations for DSR with path caches, but did not use GRATUITOUS ROUTE ERROR for DSR-Update.

I used the *ns-2* [12] network simulator together with the Monarch Project's wireless and mobile extensions [6, 47]. The mobility model is *random waypoint model* [6] in a rectangular field. Pause time 0 s was used for all scenarios. Node speed was randomly chosen from the interval $v \pm 0.1v$ for mean speeds v of 5 m/s, 10 m/s, 15 m/s and 20 m/s. Two field configurations were used: a 1500m $\times$ 1000m field with 50 nodes and a 2200m $\times$ 600m field with 100 nodes.

I used 10 CBR connections with 4 packets per second and packet size of 512 bytes as background traffic. After a warm-up time of 100 s, one or more TCP connections were established. FTP is the application over TCP. The FTP file transfer ran 900 s for 50-node and 500 s for 100-node scenarios. I used TCP-Reno with a packet size 1460 bytes. The maximum size of both the congestion window and the receiver's advertised window is 8. Each data point on the graph is the average of 10 runs of randomly generated scenarios.

Four metrics were used in the evaluation:

- *TCP Throughput*: the total size of TCP packets received by the TCP receiver divided by the duration of the TCP connection. For multiple TCP connections, this metric refers to the aggregate throughput.

- *Normalized Routing Overhead*: the ratio of the total number of routing packets transmitted (both sent and forwarded) to the total number of data packets received including both TCP packets and CBR packets. Routing packets include ROUTE ERRORS used for cache updates.

- *Cache Hit Ratio*: the ratio of the number of cache lookups in which routes are found to the total number of cache lookups.

- *Percentage of Valid Cache Hits*: the percentage of the total number of cache hits that result in valid routes.

4.3.2 Simulation Results

TCP Throughput

TCP throughput is the best TCP measure of the effectiveness of a caching strategy, as using stale routes can cause TCP repeated timeouts. The results for TCP throughput are shown in Figure 4.1 and Figure 4.2. For 50-node scenarios, the maximum improvement achieved by DSR-Update is 134% for one connection, 42% for 5 connections, and 22% for 10 connections. For 100-node scenarios, the maximum improvement is 86% for one connection, 174% for 5 connections, and 193% for 10 connections.

Such improvement demonstrates that DSR-Update quickly removes stale routes. Recall that DSR evicts stale routes through Route Maintenance, GRATUITOUS ROUTE ERRORS, and FIFO replacement policy. Because Route Maintenance operates fully on-demand, a TCP sender will not know about a broken link until a data packet from its

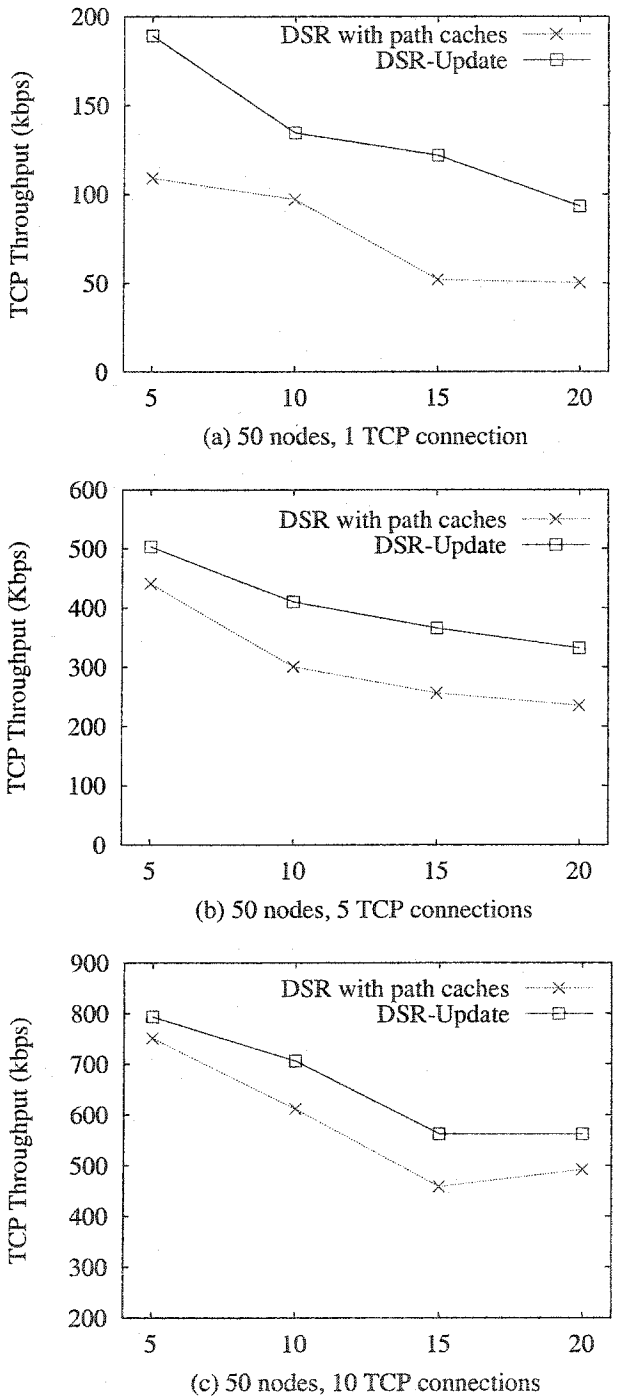


Figure 4.1: TCP Throughput vs. Mobility (Mean Speed (m/s)) for 50 Node Scenarios

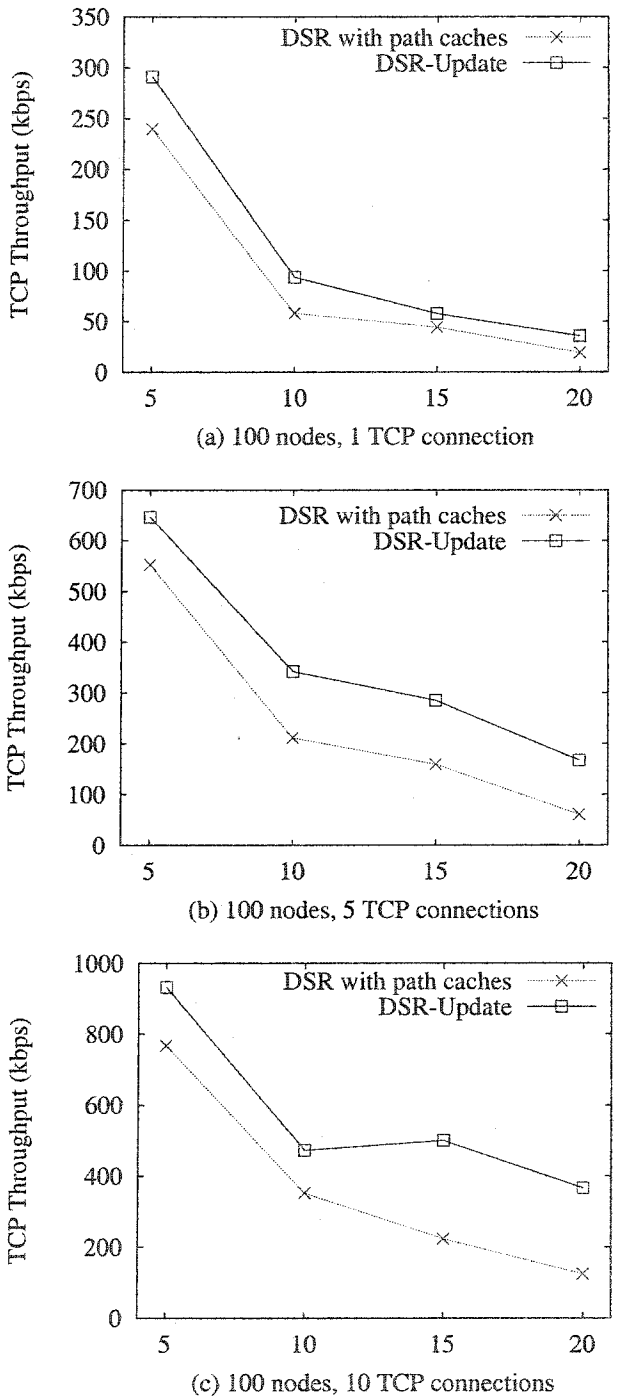


Figure 4.2: TCP Throughput vs. Mobility (Mean Speed (m/s)) for 100 Node Scenarios

connection encounters the link failure. Thus, the routing protocol at the TCP sender has delayed awareness of topology changes. Although GRATUITOUS ROUTE ERRORS help remove stale routes from the caches of additional nodes, a source node will not initiate a route discovery until no other route to the destination is available in its cache. FIFO also cannot quickly remove stale routes as it has little control of evicting which route at what time. As a result, TCP suffers from frequent route failures.

I make the following key observations:

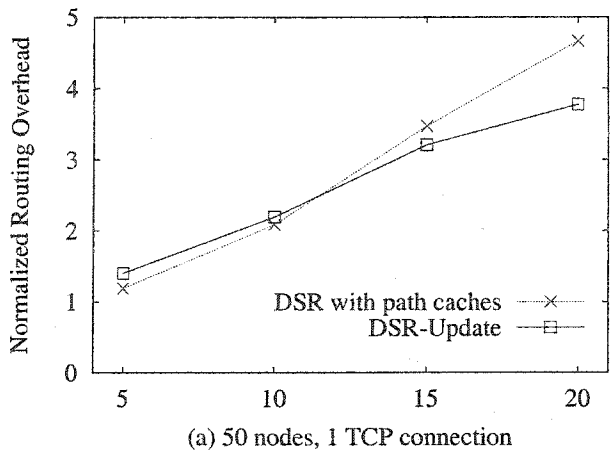
- The improvement increases as mobility increases. This result is more significant for 100-node scenarios than for 50-node ones. For example, for 10-connection scenarios shown in Figure 4.2 (c), TCP throughput is improved by 21% at 5 m/s, 34% at 10 m/s, 124% at 15 m/s, and 194% at 20 m/s. As mobility increases, more routes become stale; therefore, the advantages of quick and efficient cache updating becomes more significant.

- The improvement increases as the number of TCP connections increases. For example, for 100-node scenarios at mean speed of 15 m/s, shown in Figure 4.2, the improvement increases from 30% for one connection to 124% for 10 connections.

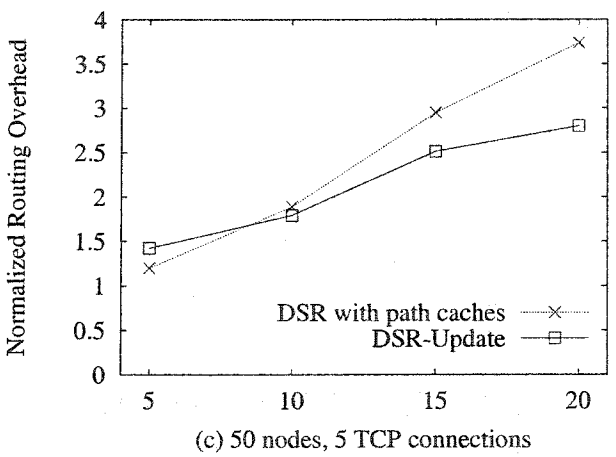
- The improvement increases as network size increases. As network size increases from 50 to 100 nodes, higher improvement is achieved. For example, for 5-connection scenarios at mean speed 20 m/s, TCP with DSR-Update obtains 41% improvement for 50-node and 175% for 100-node scenarios. This result demonstrates the benefits of proactive cache updating for large networks.

Normalized Routing Overhead

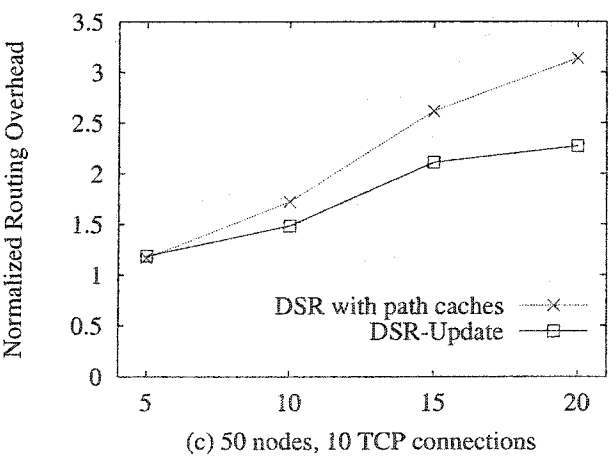
The results for normalized routing overhead are shown in Figure 4.3 and Figure 4.4. For 50-node scenarios, TCP with DSR-Update has higher overhead than TCP with DSR



(a) 50 nodes, 1 TCP connection

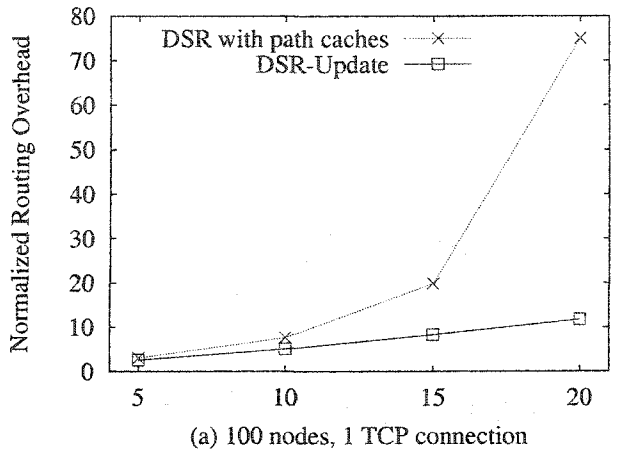


(c) 50 nodes, 5 TCP connections

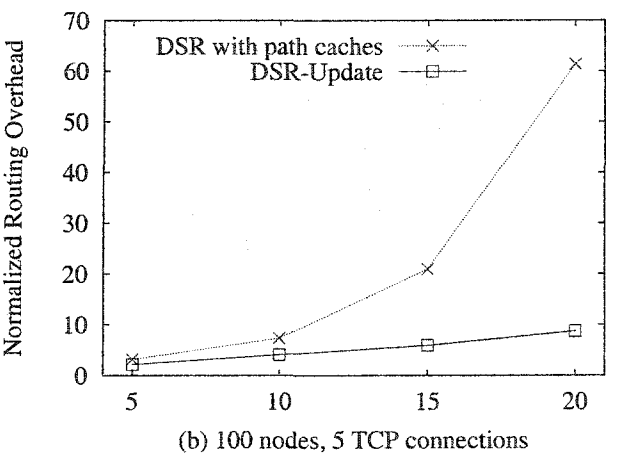


(c) 50 nodes, 10 TCP connections

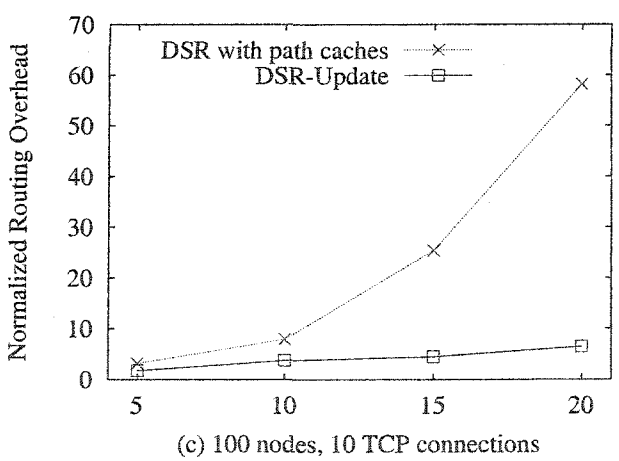
Figure 4.3: Normalized Routing Overhead vs. Mobility (Mean Speed (m/s)) for 50 Node Scenarios



(a) 100 nodes, 1 TCP connection



(b) 100 nodes, 5 TCP connections



(c) 100 nodes, 10 TCP connections

Figure 4.4: Normalized Routing Overhead vs. Mobility (Mean Speed (m/s)) for 100 Node Scenarios

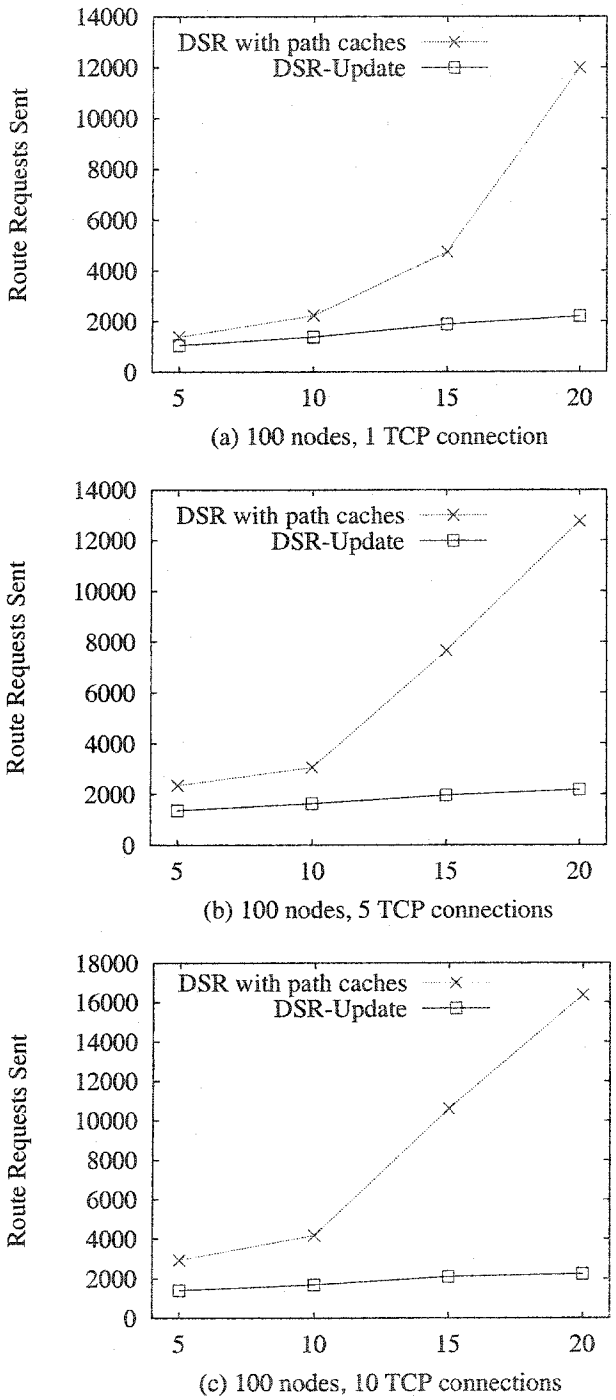


Figure 4.5: Route Requests Sent vs. Mobility (Mean Speed (m/s))

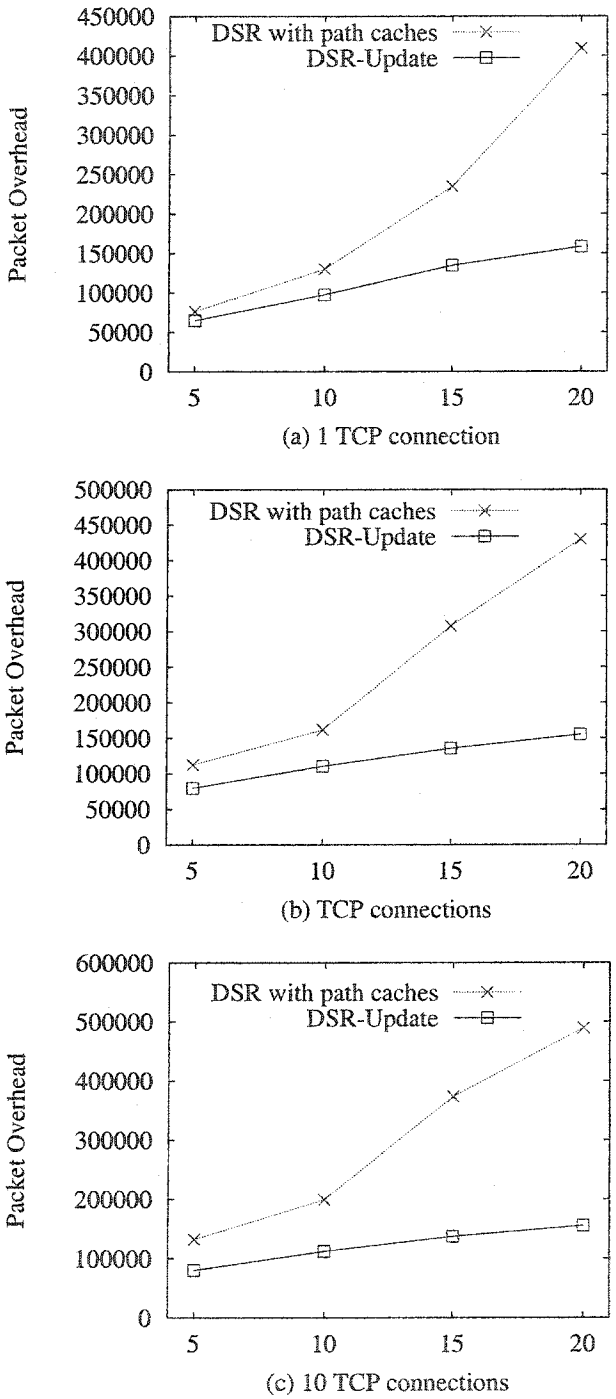


Figure 4.6: Packet Overhead vs. Mobility (Mean Speed (m/s))

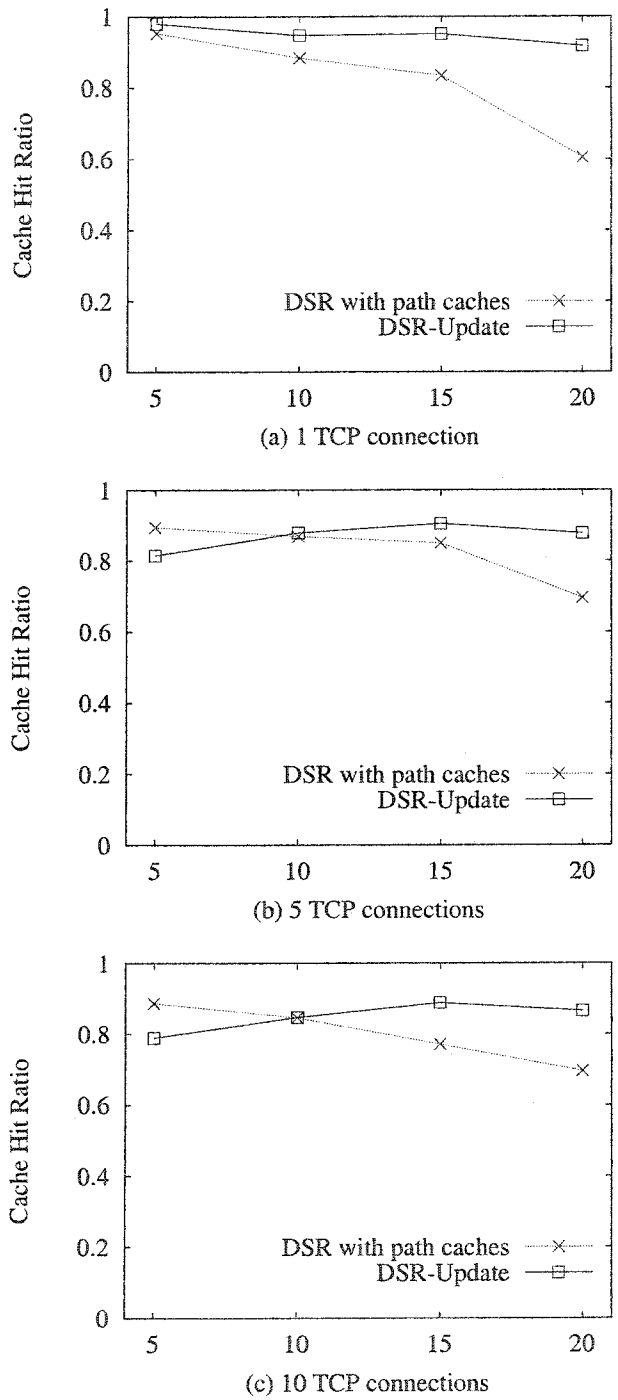


Figure 4.7: Cache Hit Ratio at both TCP senders and receivers vs. Mobility (Mean Speed (m/s))

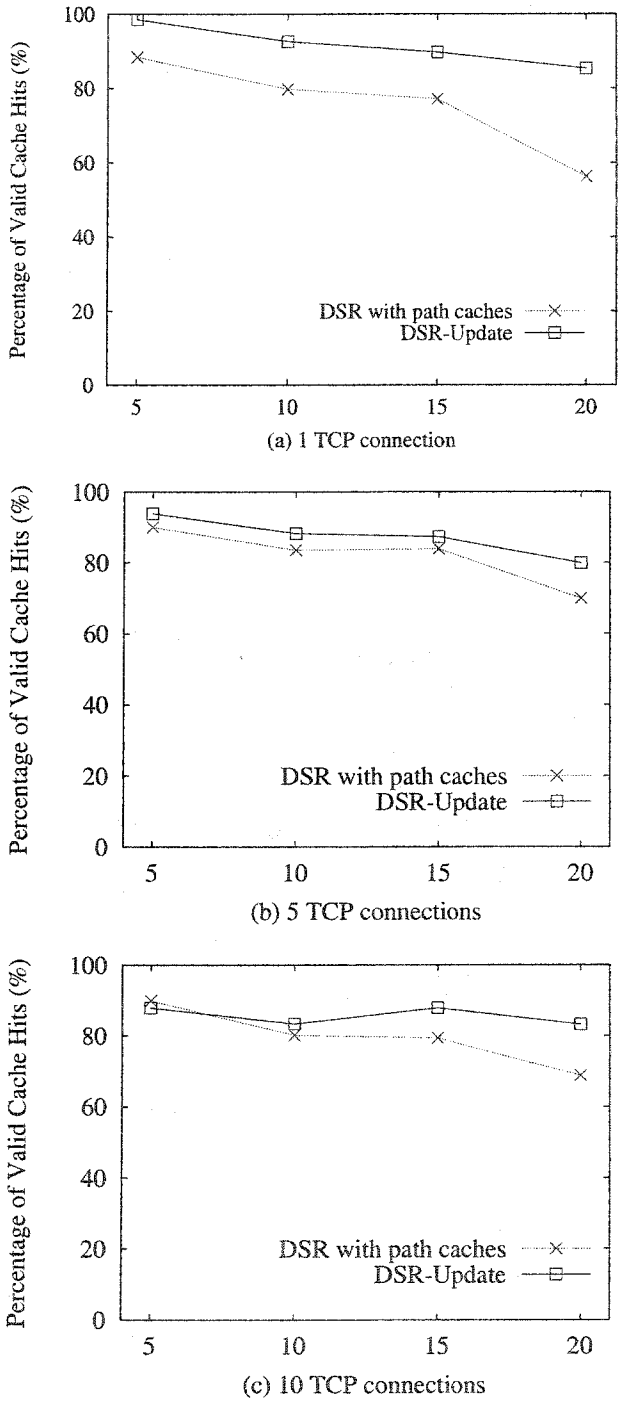


Figure 4.8: Percentage of Valid Cache Hits at both TCP senders and receivers vs. Mobility (Mean Speed (m/s))

under low mobility, but has lower overhead under high mobility. Although the algorithm introduces overhead due to cache update notifications, it reduces the number of ROUTE ERRORS caused by the use of stale routes. Another reason for the reduced overhead is that DSR initiates more route discoveries than DSR-Update because stale routes are not evicted quickly and valid routes are removed due to the small cache size.

The reduction in overhead also increases as the number of TCP connections increases. For example, for 50-node scenarios at mean speed of 20 m/s, the reduction is 19%, 25%, and 28% for 1, 5, and 10 connections respectively. Moreover, the reduction increases as network size increases, especially under high mobility. As shown in Figure 4.4, for 100-node scenarios at mean speed of 20 m/s, the maximum reduction is 84%, 86%, and 89% for 1, 5, and 10 connections respectively.

DSR with path caches has high normalized routing overhead for 100-node scenarios at mean speeds of 15 m/s and 20 m/s. As shown in Figure 4.5, DSR initiates a large number of route discoveries for these scenarios, resulting in the higher packet overhead shown in Figure 4.6. I attribute two reasons for this observation. The first reason is the fixed and small cache size. As network size increases, more routes will be discovered. As mobility increases, route failures become more frequent, and thus more route discoveries will take place and more routes need to be stored. However, a small cache is not large enough to hold all discovered routes. I confirm this analysis through cache hit ratio metric. As shown in Figure 4.7, there are more cache misses in DSR than in DSR-Update at mean speeds of 15 m/s and 20 m/s. The second reason is that stale routes are not removed quickly, and the FIFO policy evicts many valid routes. I verify this point through the percentage of valid cache hits metric. As shown in Figure 4.8, DSR-Update has better cache performance than DSR with path caches, providing higher improvement at mean speed of 20 m/s. Note that this metric reflects the percentage of valid routes among all cache hits at both TCP senders and receivers. DSR-Update allows

both senders and receivers to use more valid routes, contributing to the improvement in throughput.

4.4 Conclusions

Route failures due to mobility are the major factor degrading TCP performance in mobile ad hoc networks. I presented a new approach to improve TCP performance: reducing route failures by making route caches in on-demand routing protocols adapt quickly to topology changes. In this chapter, I investigated the impact of my distributed cache update algorithm on TCP performance. I evaluated TCP performance with DSR augmented with this algorithm and compared it with DSR with path caches through detailed simulations. I show that the algorithm improves TCP throughput and reduces normalized routing overhead. I conclude that it is important to make route caches quickly reflect topology changes so that the effect of mobility on TCP is reduced.

It is important for the network layer to be more mobility-aware. It is also necessary to modify TCP so as to prevent congestion control mechanisms from being falsely triggered, as done in recent work. However, it is insufficient to make route caches more up-to-date and to make TCP aware of route failures. In the next chapter, I will show why TCP enhanced with these two approaches still does not perform well.

Chapter 5

Improving TCP Performance in Mobile Ad Hoc Networks by Exploiting Cross-Layer Information Awareness

In this chapter, I first investigate how mobility affects TCP. I found that it is insufficient to notify TCP only about route failures. After a link failure is detected, several packets with the same next hop will be dropped from the network interface queue; TCP will time out because of these losses. It will also time out for ACK losses caused by route failures. To reduce TCP timeouts for mobility-induced losses, I present two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). EPLN seeks to notify TCP senders about lost data. For lost ACKs, BEAD attempts to retransmit ACKs either at intermediate nodes or at TCP receivers. Both mechanisms exploit cross-layer information awareness: the network layer is aware of lost TCP packets. I evaluated TCP-ELFN [21] enhanced with the two mechanisms using two caching strategies for DSR, path caches and my distributed cache update algorithm. I show that TCP-ELFN with EPLN and BEAD significantly outperforms TCP-ELFN under both caching strategies.

5.1 Introduction

TCP performance degrades significantly in mobile ad hoc networks [21, 13, 51]. In such networks, nodes move arbitrarily. Route failures due to mobility are the primary reason for most packet losses [13, 51]. Since TCP assumes that packet losses occur because of congestion, it will invoke congestion control mechanisms for packet losses caused by route failures, resulting in the reduction in throughput.

Several transport layer mechanisms [7, 21, 10, 36] have been proposed to address the problems caused by mobility. One of the promising approaches is to provide link failure feedback to TCP so that TCP can avoid responding to route failures as if congestion had occurred. ELFN (Explicit Link Failure Notification) [21] is such a mechanism. With ELFN, when a node detects a link failure, it will notify the TCP sender about the link failure and the packet that encountered the failure. When receiving a notification, TCP freezes its retransmission timer and periodically sends a probing packet until it receives an ACK. TCP then restores its retransmission timer and continues as normal. ELFN was shown to outperform TCP in mobile ad hoc networks.

TCP benefits from link failure feedback but is still affected by frequent route failures. Holland and Vaidya [21] observed that TCP experiences repeated route failures due to the inability of a TCP sender's routing protocol to quickly recognize and remove stale routes from its cache. This problem is complicated by allowing nodes to respond to route discovery requests with routes from their caches, because they often respond with stale routes. Holland and Vaidya showed that turning off replying from caches improves TCP performance for a network with a single TCP connection. However, this approach will degrade TCP performance when multiple traffic sources exist because of increased routing overhead. Thus, stale routes present a serious challenge to TCP.

To address the cache staleness issue in the context of DSR (the Dynamic Source Routing protocol) [30, 31], Chapter 3 presented a distributed cache update algorithm.

When a link failure is detected, the algorithm proactively notifies all reachable nodes that have cached that link to update their caches. Therefore, the algorithm enables the routing protocol at TCP senders and receivers to quickly remove stale routes from their caches. Chapter 4 showed that this algorithm improves TCP throughput, because it reduces route failures by making the network layer more mobility-aware.

In this chapter, I investigate how to make TCP perform well in the presence of frequent packet losses due to mobility. In contrast to prior work, my work focuses on issues at both the network layer and the transport layer, as well as the interactions between these two layers. I seek to answer two questions:

1. What should be the appropriate responses of TCP to frequent route failures and packet losses? For example, is it always good to freeze TCP when route failures occur? Or is it better to freeze TCP only when packets losses occur?

2. How can TCP be made efficient through approaches at the network layer and cross-layer?

To answer these questions, I first study how mobility affects TCP through simulation of ELFN. I make several observations. First, I find that, after congestion control mechanisms are restored, keeping TCP's state the same as it was when TCP was frozen improves throughput and reduces TCP timeouts compared with using default values. Second, I find that there is a trade-off between freezing TCP upon route failures and upon packet losses. Route failures do not imply packet losses because packets can be salvaged by an intermediate node using a cached route. If packets are salvaged, freezing TCP may decrease throughput because TCP can continue to send packets using other routes; however, if TCP is not frozen when packets are salvaged, it will time out if salvaged packets are dropped. I observe that these two choices result in similar through-

put, but freezing TCP upon route failures reduces TCP timeouts. Finally, I identify two problems at the network layer that affect the efficient operation of TCP:

- Unaware of lost data packets: Prior work mainly focused on making TCP aware of route failures. However, after a link failure is detected, a routing protocol will drop all the data packets with the same next hop in the network interface queue. TCP will time out because of these losses.

- Unaware of lost ACKs: Upon route failures, ACKs are also dropped silently. As a result, TCP senders have to wait for timeouts and retransmit unacknowledged packets. Waiting for timeouts not only degrades throughput but wastes bandwidth; retransmitting the packets that have been received wastes nodes' energy.

I propose to make routing protocols aware of lost data packets and ACKs and help reduce TCP timeouts for mobility-induced losses. Toward this end, I present two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD).

With EPLN, when a node detects a link failure, if it cannot salvage data packets and the packets have not been salvaged, it sends a notification to the TCP sender. The notification includes the sequence numbers of all dropped packets for that connection. For the lost packets that were salvaged by an intermediate node, the node sends a notification to the intermediate node, which attempts to send a notification to the TCP sender using a cached route. When the sender's routing protocol receives a notification, it notifies TCP about all lost packets. TCP disables its retransmission timer, records these lost packets, and retransmits the lost packet with the lowest sequence number. When an ACK arrives, TCP restores its retransmission timer and retransmits the remaining lost packets.

With BEAD, when a node detects a link failure, if it cannot salvage ACKs, it sends a notification about the lost ACKs to the TCP receiver if the ACKs have not been salvaged, or to the intermediate node that salvaged the ACKs. When forwarding a notification,

a node attempts to retransmit an ACK with the highest sequence number among lost ACKs to the sender using a cached route. If the intermediate node that salvaged the ACKs cannot retransmit an ACK, it sends a notification to the receiver. If none of the intermediate nodes is able to retransmit an ACK, the receiver's routing protocol retransmits an ACK with the highest sequence number if it has a route to the sender.

Since EPLN and BEAD extensively use cached routes, I evaluate the effectiveness of the mechanisms using different caching strategies. I incorporate EPLN and BEAD into DSR with path caches and into DSR with my distributed cache update algorithm. Through detailed simulations, I compare the performance of TCP-ELFN; TCP-ELFN with EPLN and BEAD; and TCP-ELFN with EPLN, BEAD, and my cache update algorithm. I show that, compared with TCP-ELFN, EPLN and BEAD significantly improve TCP throughput under both caching strategies. For example, for 100-node networks, TCP throughput is improved by up to 173% for DSR and up to 210% for DSR with my cache update algorithm, both at node mean speed of 20 m/s. Moreover, EPLN and BEAD considerably reduce TCP timeouts, by more than 33% for DSR and 44% for DSR with my cache update algorithm for 100-node networks. In addition, enhanced with my cache update algorithm, TCP-ELFN with EPLN and BEAD outperforms TCP-ELFN with EPLN, BEAD, and path caches by up to 43% in throughput.

The rest of this chapter is organized as follows. In Section 5.2, I study how mobility affects TCP. In Section 5.3, I describe EPLN and BEAD, and in Section 5.4, I present an evaluation of EPLN and BEAD. Finally, I present my conclusions in Section 5.5.

5.2 Mobility, TCP, and ELFN

In this section, I study how mobility affects TCP through simulation of ELFN in a network with 50 nodes and one TCP connection. I used the same simulation environment

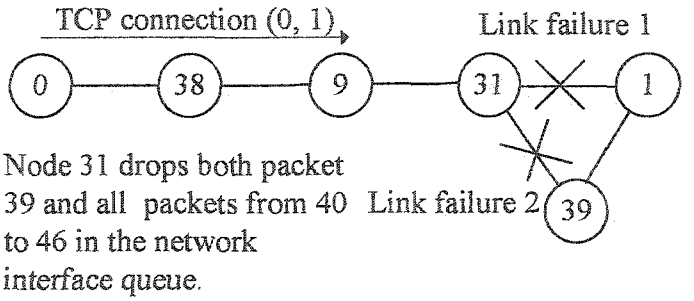


Figure 5.1: An Example of How Mobility Affects TCP

as the one described in Chapter 4. The node speed was randomly chosen from 10 ± 1 m/s. I explore three issues: (1) how to set the retransmission timeout (RTO) and the congestion window size after congestion control mechanisms are restored; (2) whether to freeze TCP upon route failures or upon packet losses; and (3) the network layer is unaware of lost data packets and ACKs.

5.2.1 How should RTO and cwnd be Set after Congestion Control Mechanisms are Restored?

Node 0 starts a TCP connection to node 1 at 100 s. At time 101.607677 s, node 31 detects that the link from node 31 to node 1 is broken when transmitting the packet with sequence number 39 using route 0–38–9–31–1, as shown in Figure 5.1. Node 31 salvages this packet and the packets from 40 to 46 in the network interface queue using route 31–39–1. It then sends a ROUTE ERROR to node 0, including the sender and the receiver addresses, ports, and sequence number 39. This is an ELFN message. However, the link from node 31 to node 39 is also broken, and therefore node 31 drops all packets. At time 101.621079 s, node 0 receives the ROUTE ERROR; the routing protocol sends an ICMP message to TCP. TCP disables its retransmission timer, starts a timer called a thaw timer with timeout 2 s, and sets the sequence number of the probing packet to 39.

After TCP is frozen, it does not send any packet until the thaw timer times out. Thus, at time 103.621079 s, node 0 sends the probing packet 39. At time 103.695992 s, node 0 receives the ACK for packet 39 and restores TCP's retransmission timer.

I consider two choices for setting RTO and congestion window size, cwnd. One choice is to use the default value 6 s for RTO and 2 for cwnd; the other choice is to keep TCP's state the same as it was when TCP was frozen. Both choices were discussed by Holland and Vaidya [21]. They observed that adjusting window size had little impact on throughput, but changing RTO resulted in more reduction in throughput. They suspected that the impact of RTO was most probably caused by the frequency at which routes break and the proclivity of the Address Resolution Protocol (ARP) [46] to silently drop packets. If a restored route breaks and results in a failed ARP query, the sender will likely time out. They concluded that, given the length of timeout, using the default RTO does not dramatically affect performance.

I attribute a different reason to the reduced throughput when default values are used. In this example, the sequence number of the next packet to be sent, 47, is larger than the highest sequence number of acknowledged packets, 39, plus the reset window size, 2. Therefore, TCP will not send any packet until an ACK arrives. However, the packets with sequence numbers from 40 to 46 were dropped, and thus no ACK will arrive. At time 109.695992 s, the retransmission timer expires and TCP retransmits packet 40. Here, reducing cwnd causes TCP to enter an idle state; if the packets already sent are lost, TCP has to wait for timeouts. Therefore, using default values for RTO and cwnd degrades TCP throughput. Since TCP relies on RTO to recover from an idle state, it is better to use a smaller RTO, such as the "old" value, which is 0.8 s in this example. I will present an evaluation of the two choices in Section 6.3.

5.2.2 When should TCP be Frozen?

In ELFN, TCP will be frozen either when a TCP sender initiates a route discovery, or when a TCP sender receives an ELFN message indicating a link failure. A TCP sender will receive a notice only when a data packet encounters a link failure for the first time. If the packet encountering a link failure was salvaged before the occurrence of this failure, then only the node salvaging the packet can receive a notice, since ELFN piggybacks a notice on the ROUTE ERROR sent by DSR. For example, in Figure 5.1, after the link from node 31 to node 39 is detected as broken, node 31 does not send a ROUTE ERROR to node 0, since it is the source node of route 31–39–1. Thus, node 0 will not know about the link failure.

ELFN does not distinguish packet losses from link failures; it freezes TCP even if packets are salvaged. At time 109.695992 s, TCP retransmits packet 40 using route 0–38–9–31–39–1. However, the link from node 0 to node 38 is detected as broken at time 109.728763 s. TCP is frozen because no route is available in node 0's cache. At time 109.738086 s, node 0 sends packet 40 using the discovered route 0–22–16–1. But this is also a stale route, since the link from node 22 to node 16 has broken. Node 22 salvages this packet using another route and sends a ROUTE ERROR to node 0. Because TCP has been frozen, no changes are made to TCP's state.

At time 109.924568 s, node 0 receives the ACK for packet 40. TCP's state is restored: RTO is set to 6 s and cwnd is set to 2. The sequence number of the next packet to be sent, 41, is less than the highest sequence number of acknowledged packets, 40, plus the reset window size, 2. Thus, TCP sends packets 41 and 42 using route 0–9–16–1, since TCP is in slow-start phase. However, the link from node 9 to node 16 is broken. Node 9 salvages the packets using another route and sends a ROUTE ERROR to node 0. TCP is frozen although the packets have been salvaged. TCP remains frozen till the ACK for packet 41 arrives.

As discussed in Section 5.1, if packets are salvaged, freezing TCP upon route failures may decrease TCP throughput. On the other hand, if TCP is not frozen when packets are salvaged, it will time out if the salvaged packets are dropped. I observed from simulations that freezing TCP upon route failures reduces TCP timeouts. Therefore, I believe that this is a conservative but reliable approach because route failures are frequent. Thus, I will use this option in simulations.

5.2.3 The network layer is unaware of Lost Data Packets and ACKs

As I have shown, ELFN notifies a TCP sender about a link failure only when a packet encounters a link failure for the first time; a notification does not indicate whether the packet is lost. Another problem exists at the network layer: upon route failures, a routing protocol silently drops all the packets with the same next hop in the network interface queue. Since TCP does not know about these losses, it has to time out. If an intermediate node notifies a TCP sender about packet losses, the TCP sender will retransmit lost packets earlier and thus avoid waiting for timeouts.

I discuss another example in which TCP times out because RTO and cwnd are set to default values and because data packets are dropped silently. At time 120.140313 s, node 16 attempts to transmit packet 409 using route 0–25–16–1 but detects that the link from node 16 to node 1 is broken. It salvages this packet using route 16–34–1 and sends a ROUTE ERROR to node 0. TCP is frozen and the sequence number of the probing packet is set to 409. At time 120.164540 s, node 0 receives the ACK for packet 408, and thus TCP's state is restored. The next packet to be sent is 412, larger than 408 plus the reset window size. Therefore, TCP enters an idle state, although node 0 has routes to reach node 1 *and* the window size before being reset allows TCP to send more packets. At time 120.239832 s, node 16 detects that the link from node 16 to node 34 is broken and drops packets 409 and 410. As a result, TCP times out at time 126.164540 s.

If TCP's state is kept the same as it was when TCP was frozen, TCP will be able to send packet 412. If this packet is delivered, a duplicate ACK with sequence number 408 will be returned because packet 409 was dropped. Three duplicate ACKs trigger TCP's fast retransmission; however, fast retransmission recovers only the first lost packet. TCP still will time out if there are multiple losses. Therefore, it is necessary to let TCP know about lost packets whether TCP's state is set to default values or not.

Upon route failures, ACKs are also dropped silently; therefore, TCP senders will time out and retransmit unacknowledged packets. Due to mobility, retransmitted data packets and ACKs could be salvaged multiple times until they reach their destinations, or they could be dropped, and thus TCP would time out and start another retransmission. Since TCP relies on ACKs to ensure reliability and to trigger further transmissions, it is important for a routing protocol to fast deliver ACKs.

5.3 Early Packet Loss Notification and Best-Effort ACK Delivery

It is important for the network layer to be aware of lost data packets and ACKs and to help reduce TCP timeouts for mobility-induced losses. To achieve this goal, I present two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). In this section, I describe the two mechanisms in detail with examples.

5.3.1 Overview

The key idea of EPLN and BEAD is that intermediate nodes notify TCP senders about lost data packets and retransmit ACKs for lost ACKs by extensively using cached routes. No route discovery is initiated at any intermediate node. It is simple to find a route by

initiating a route discovery, but such an approach is not efficient, because packet losses are frequent and route discoveries introduce significant overhead.

I consider three types of packets that encounter route failures: data packets, ACKs, and packet loss notifications. I summarize the operation of EPLN and BEAD as follows:

1. If data packets or ACKs are dropped and this is the first time they have encountered a link failure, then the current node sends a notification to the TCP sender for lost data packets or to the TCP receiver for lost ACKs, using the route obtained by reversing the source route.

2. If data packets are dropped after being salvaged by an intermediate node, then the current node notifies the intermediate node about lost packets. The intermediate node sends a notification to the sender if it has a cached route.

3. If ACKs are dropped after being salvaged by an intermediate node, then the current node notifies the intermediate node about lost ACKs. That node first attempts to retransmit an ACK with the highest sequence number among lost ACKs using a cached route; if it cannot, it sends a notification about lost ACKs to the receiver.

4. When forwarding a notification about lost ACKs, a node attempts to retransmit an ACK with the highest sequence number among lost ACKs to the sender using a cached route. If it can do so, it marks the notification to indicate that an ACK has been retransmitted. If none of the intermediate nodes is able to retransmit an ACK, the routing protocol at the receiver retransmits an ACK if it has a cached route to the sender.

5. If a notification is dropped due to a link failure, the node detecting the link failure notifies the node that is the source of the notification. That source node will send another notification to the sender or the receiver using a cached route.

Thus, the network layer tries its best to let TCP senders know about lost data packets and to retransmit ACKs for lost ACKs. The two feedback mechanisms are applicable to any routing protocol, as they address general problems that occur at the network layer.

Route caches play an important role in both EPLN and BEAD, due to the extensive use of cached routes. My prior work [58] has addressed the cache staleness issue; I will use my distributed cache update algorithm as one caching strategy in the evaluation of EPLN and BEAD. For DSR with path caches, EPLN and BEAD provide another benefit: quick detection and eviction of stale routes.

5.3.2 Packet Loss Notifications

I define a data structure called *drop_list* to record dropped packets. Before a node drops a data packet or an ACK, it records in its *drop_list* the following information about the packet: source address, source port, destination address, destination port, packet type (data or ACK), sequence number, and the source route used in routing the packet. A node uses the information in its *drop_list* to construct packet loss notifications.

I define another structure called *conn_info* to record in a notification the connection information about lost packets originating from the same connection. The information includes source address, source port, destination address, destination port, packet type, and the sequence numbers of lost packets. When possible, I piggyback the information about lost packets on a ROUTE ERROR sent by DSR; otherwise, a notification will be sent as a ROUTE ERROR. I extend the format of a ROUTE ERROR to include an optional field called *conn_list*, which contains one or more *conn_info* structures.

5.3.3 EPLN and BEAD

In this section, I describe EPLN and BEAD together and elaborate on each when necessary, since they have common operations at the node detecting a link failure and different

operations at the node forwarding or receiving a notification.

At the node detecting a link failure

When a node detects a link failure, it attempts to salvage the data packet or ACK encountering the broken link. If it cannot salvage the packet, it creates an entry in its *drop_list*, recording the information about the packet. It then checks the network interface queue for the packets that have the same next hop in their routes. For the data packets or ACKs to be dropped, the node records the information about the packets in the *drop_list*.

If the node is the TCP sender of the packet encountering the broken link, it sends an ICMP message to TCP including the sequence number of the packet. ELFN uses this operation to freeze TCP. The node then tries to find a route either from its cache or by initiating a route discovery. I focus on the operation at the network layer first and will describe the responses of TCP to an ICMP message later.

If the node is not the source node of the packet, it will send a ROUTE ERROR to the source node. This source node is a TCP sender, or a TCP receiver, or an intermediate node that salvaged the packet before this link failure. The node piggybacks on the ROUTE ERROR the information about the lost packets that have the same source node as the packet encountering the broken link. It creates one entry in the *conn_list* of the ROUTE ERROR for the lost packets originating from the same connection. The node then sends the notification using the route obtained by reversing the source route. For the lost packets that have different source nodes, the node sends one notification to each source. If the packet encountering the broken link is a data packet and is salvaged, the node also adds the TCP connection information to the ROUTE ERROR, since I choose to freeze TCP upon route failures as done in ELFN.

If the node is the source node of a lost packet but not the TCP sender or the TCP receiver, then it is the intermediate node that salvaged the packet. The node attempts

to send a notification to the TCP sender for lost data packets or to the TCP receiver for lost ACKs using a cached route. If no route can be found, the node will not process the information about lost packets. An extension to this approach is to record all nodes that salvaged a packet in the packet header, so that these nodes can relay a notification until it reaches the TCP sender or the TCP receiver.

At the node forwarding a notification

When a node forwards a notification about lost ACKs, it checks whether it can retransmit an ACK to the TCP sender using a cached route. If it finds a route, it sends an ACK with the highest sequence number among lost ACKs to the TCP sender. The node then marks a field called *ack_sent* as *true* in the corresponding entry of the *conn_list*, indicating that an ACK has been sent. Thus, other nodes forwarding the notification only attempt a retransmission for the entries in which *ack_sent* is *false*.

If a retransmitted ACK is dropped due to a link failure, a ROUTE ERROR is sent to the node that retransmitted the ACK. That node attempts to retransmit another ACK using a cached route. If the node does not have a cached route to reach the TCP sender, it sends a notification to the TCP receiver. If the notification to the TCP receiver encounters a link failure and is dropped, the node detecting the link failure sends a ROUTE ERROR to the source node of the notification, which will attempt to send another notification to the TCP receiver using a cached route. This is *Best-Effort ACK Delivery* (BEAD).

I used the nested ROUTE ERROR technique of DSR. With this technique, when a ROUTE ERROR encounters a link failure, the node detecting the broken link sends a ROUTE ERROR to the source of the previous ROUTE ERROR, including the information about both broken links. That source node will send another ROUTE ERROR to the previously intended destination. I modified this technique: if no cached route is available, an intermediate node does not initiate any route discovery in order to send a notification.

This is because my mechanisms generate more ROUTE ERRORS than DSR, and route discoveries introduce significant overhead.

At the node receiving a notification

The destination of a notification is a TCP sender, or a TCP receiver, or an intermediate node that salvaged either data packets or ACKs. As shown in Figure 5.2, for each entry in the *conn_list* of the notification, the node does the following steps:

1. If the node is a TCP sender, it sends an ICMP message to TCP for each sequence number, notifying TCP about each lost packet.

2. If the node is a TCP receiver and no ACK was retransmitted, the node checks whether it has a cached route to the TCP sender. If it has a cached route, it sends an ACK with the highest sequence number among lost ACKs to the TCP sender.

3. If the node is an intermediate node, it handles two cases: (1) If the lost packets are ACKs and no ACK was retransmitted, the node first checks whether it has a cached route to reach the TCP sender. If it has a cached route, it sends an ACK with the highest sequence number to the TCP sender; otherwise, if it has a route to the TCP receiver, it sends a notification to the TCP receiver. If the node has no cached route to reach either of them, it will not process the information about lost ACKs. (2) If the lost packets are data packets, the node checks whether it has a cached route to reach the TCP sender. If it has a cached route, it sends a notification to the TCP sender.

If the node is an intermediate node, it will send a notification either to a TCP sender or to a TCP receiver only when the notification received is not a nested ROUTE ERROR, indicated by a field called *num_route_error* shown in Figure 5.2. If the notification is a

```
Variables:
iph:IP header; tcph:TCP header; srh:Source Route header;
p:the current packet; new_p:a new packet;
deliver_to_dest:whether to send an ACK to the sender;
num_route_error:the number of route errors in p;
if has_conn_info then
    for each entry e in the conn_list do
      if e.src = net_id then
        p.iph.saddr := e.src; p.iph.sport := e.sport;
        p.iph.daddr := e.dst; p.iph.dport := e.dport
        if e.ptype = TCP then
          for each seq_no in e do
            new_p := p.copy(); new_p.tcph.seqno := seq_no;
            sendICMPtoTCP(new_p)
        if e.ptype = ACK and e.ack_sent = FALSE then
          new_p := p.copy(); new_p.tcph.seqno := max(seq_no ∈ e);
          new_p.srh.has_conn_info := FALSE;
          new_p.src := net_id; new_p.dest := p.iph.daddr
          if findRoute(new_p.dest, new_p.route) then
            sendOutPacketWithRoute(new_p);
      else
        deliver_to_dest := FALSE; new_p := p.copy()
        if e.ptype = ACK and e.ack_sent = FALSE then
          if findRoute(e.dst, new_p.route) then
            new_p.dest := e.dst; deliver_to_dest := TRUE
          elseif findRoute(e.src, new_p.route) then new_p.dest := e.src
        elseif e.ptype = TCP then
          if findRoute(e.src, new_p.route) then new_p.dest := e.src
        if new_p.route ≠ ∅ then new_p.src := net_id
          if e.ptype = ACK and deliver_to_dest = TRUE then
            new_p.tcph.seqno := max(seq_no ∈ e);
            new_p.iph.saddr := e.src; new_p.iph.sport := e.sport;
            new_p.iph.daddr := e.dst; new_p.iph.dport := e.dport;
            new_p.srh.has_conn_info := FALSE;
            sendOutPacketWithRoute(new_p)
          elseif num_route_error = 1 then
            new_p.srh.e := e; new_p.srh.has_conn_info := TRUE;
            sendOutPacketWithRoute(new_p);
```

Figure 5.2: Pseudo Code Executed at the Node Receiving a Packet Loss Notification

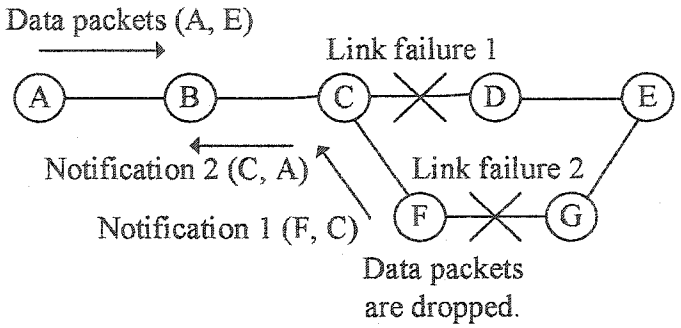


Figure 5.3: An Example of Early Packet Loss Notification

nested ROUTE ERROR, the information about lost packets is piggybacked on the ROUTE ERROR sent by DSR. I will show an example for this case in the next section.

Examples

An example of EPLN is shown in Figure 5.3. Node *A* starts a TCP connection to node *E* using route *A–B–C–D–E*. When node *C* detects that the link from node *C* to node *D* is broken, it salvages the packet using route *C–F–G–E*. The information about the TCP connection remains unchanged in the IP header of the packet, but node *C* becomes the source node of the new route in the source route header.

Then node *F* detects that the link from node *F* to node *G* is broken and finds that it cannot salvage the packet. Before dropping the packet, node *F* records the information about the packet in its *drop_list*. It then checks the network interface queue for data packets or ACKs that have the same next hop in their routes. For simplicity, assume that this packet is the only packet to be dropped. Otherwise, node *F* needs to send a notification to each TCP sender, TCP receiver, or intermediate node that salvaged the packets, but sends only one notification for the packets with the same source node. Node *F* piggybacks the packet loss information on the ROUTE ERROR sent to node *C*, the intermediate node that salvaged the packet. When receiving the notification, node *C*

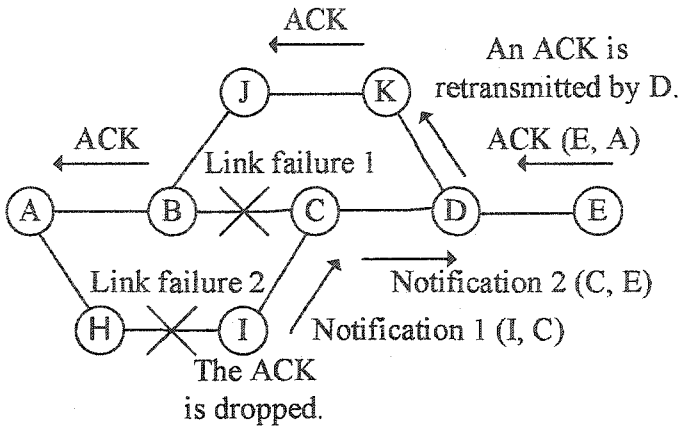


Figure 5.4: An Example of Best-Effort ACK Delivery

finds that it is not the TCP sender of the packet. It checks its cache and finds a route to the TCP sender; it then sends a notification to node *A*. If this notification encounters a link failure, for instance, the link from node *B* to node *A* is detected as broken, node *B* sends a ROUTE ERROR to node *C*, which is a nested ROUTE ERROR. Node *C* attempts to send another notification to node *A* using a cached route.

Next, I show an example of BEAD. As shown in Figure 5.4, node *E* sends an ACK to node A using route *E–D–C–B–A*. Node *C* detects that the link from node *C* to node *B* is broken and salvages the ACK using route *C–I–H–A*. Then node *I* detects that the link from node *I* to node *H* is broken. It piggybacks the information about the lost ACK on the ROUTE ERROR sent to node *C*, the intermediate node that salvaged the ACK. When node *C* receives the notification, it first checks whether it has a cached route to reach the TCP sender, node *A*. Assume that node *C* does not have such a route, so it sends a notification to the TCP receiver, node *E*. When node *D* forwards this notification, it checks whether it has a cached route to reach the TCP sender and finds that it has a route *D–K–J–B–A*. Thus, node *D* retransmits an ACK to node *A*. If node *D* does not have a cached route to reach node *A*, node *E* will attempt to retransmit an ACK to node *A* using

a cached route. If multiple ACKs from the same connection are dropped, an intermediate node or a TCP receiver retransmits an ACK with the highest sequence number among the lost ACKs recorded in a notification, without any delay.

At a TCP sender: cross-layer interactions

I have presented the operation designed at the network layer. In this section, I show how the transport layer makes use of the information provided by the network layer to achieve efficient adaptation to packet losses.

In BEAD, the routing protocol attempts to retransmit an ACK for lost ACKs, exploiting cross-layer information awareness, without cross-layer information exchange. Cross-layer interactions exist in EPLN at a TCP sender. I modified the operation of ELFN at a TCP sender to make TCP respond to packet losses. Instead of notifying TCP about the packet encountering a link failure, the network layer sends an ICMP message to TCP for each lost packet. An ICMP message includes the sequence number of a lost packet. However, it is insufficient to notify TCP only about a sequence number. Since I choose to freeze TCP upon route failures, the network layer sends an ICMP message to TCP even if the packet encountering a link failure is salvaged. If a packet is salvaged, TCP does not need to retransmit the packet; if a packet is dropped, TCP needs to retransmit the packet. Thus, an ICMP message also indicates whether a packet is lost.

As shown in Figure 5.5, when TCP receives an ICMP message, it does the following steps:

1. If the sequence number in the packet is less than or equal to the highest sequence number of acknowledged packets, or larger than the sequence number that the congestion window allows to send, then drop this packet.

2. If TCP is not frozen, then freeze TCP by disabling its retransmission timer. If the thaw timer is idle, then start the timer with timeout value 2 s and set *thaw_seqno*

```
TCP-Reno recv():
Variables:
tcph:TCP header; icmp:ICMP header; p:the current packet;
thaw_timer:thaw timer in ELFN; t_thaw:the probing interval;
thaw_seqno:sequence number of the probing packet;
lost_pkt: lost packets that have not been retransmitted;
num_lost_pkt:the number of packets in lost_pkt;
tcp_melt:whether TCP's state is restored or not;
if p.ptype = ICMP then
    if not (p.tcph.seqno > highest_ack_ and
        p.tcph.seqno <= highest_ack + window()) then
      free(p);
      return
    if not frozen() then
      freeze()
    if thaw_timer.status() = TIMER_IDLE then
      thaw_timer.resched(t_thaw);
      thaw_seqno := p.tcph.seqno
      if p.icmp.pkt_lost = TRUE then /*new*/
        output(thaw_seqno)
    elseif p.tcph.seqno < thaw_seqno then
      thaw_seqno := p.tcph.seqno
      if p.icmp.pkt_lost = TRUE then /*new/
        output(thaw_seqno)
    elseif p.tcph.seqno = thaw_seqno then /*new*/
      if p.icmp.pkt_lost = TRUE then
        output(thaw_seqno)
    elseif p.tcph.seqno > thaw_seqno then /*new*/
      if p.icmp.pkt_lost = TRUE and p.tcph.seqno ∉ lost_pkt then
        lost_pkt[num_lost_pkt] := p.tcph.seqno;
        num_lost_pkt := num_lost_pkt + 1
    free(p)
elseif frozen() then melt(); /* tcp_melt is set to TRUE.*/
if tcp_melt = TRUE then /*new*/
    if num_lost_pkt ≠ 0 then
      for each seqno ∈ lost_pkt do
        if seqno > last_ack then
          output(seqno); num_lost_pkt := 0; tcp_melt := FALSE
```

Figure 5.5: Pseudo Code Executed at TCP sender When Receiving an ICMP Message

to be the sequence number in the ICMP packet. If the original packet is lost, then retransmit the packet, rather than wait for the timeout of the thaw timer as done in ELFN.

3. If TCP is frozen and the sequence number is less than or equal to the *thaw_seqno*, then update the *thaw_seqno* to be the sequence number in the ICMP packet. If the original packet is lost, then retransmit the packet.

4. If TCP is frozen, the sequence number is larger than the *thaw_seqno*, and the original packet is lost, then record the lost packet in an array called *lost_pkt*, but do not retransmit it now. TCP was frozen by a previous ICMP packet either because of a link failure and the packet with the thaw_seqno was salvaged, or because of a lost packet and TCP has retransmitted that packet. In either case, a packet is on its way to the TCP receiver. Due to possible stale routes, it is better to wait for the arrival of an ACK.

When an ACK arrives, TCP restores congestion control mechanisms and retransmits other lost packets recorded in the *lost_pkt*. Thus, TCP adapts quickly to packet losses.

5.4 Performance Evaluation of EPLN and BEAD

5.4.1 Evaluation Methodology

I performed two sets of experiments. In the first set of experiments, I evaluated the effects of two choices for setting RTO and cwnd on TCP performance. One choice is to use the default value 6 s for RTO and 2 for cwnd; the other choice is to use the values used before TCP is frozen. In the second set of experiments, I evaluated the

effectiveness of EPLN and BEAD under two caching strategies for DSR: path caches and my distributed cache update algorithm, which I call DSR-Update. I used the basic operation of ELFN: freezing TCP upon route failures, sending a probing packet every time the thaw timer expires, and restoring TCP's state when an ACK arrives.

I compared the performance of TCP enhanced with three combinations of the mechanisms at the transport layer and the network layer: (1) TCP-ELFN with default RTO 6 s and cwnd 2, and DSR with path caches; (2)TCP-ELFN with RTO and cwnd set to the values computed before TCP is frozen, and DSR with path caches with EPLN and BEAD; (3) TCP-ELFN with RTO and cwnd set to the values computed before TCP is frozen, and DSR with EPLN, BEAD, and DSR-Update. In addition, I evaluated TCP performance for DSR under both promiscuous and non-promiscuous mode.

I studied the effects of traffic load on TCP enhanced with different mechanisms by investigating scenarios with 1, 5, and 10 TCP connections. I did not use higher traffic load in order to factor out the effect of congestion. I used node mean speed of 5 m/s, 10 m/s, 15 m/s, and 20 m/s. Node pause time was 0 s for all scenarios. Each simulation ran for 900 s of simulated time. Each data point represents an average of 10 runs of different randomly generated scenarios. The probing interval of ELFN was 2 s.

I used three metrics: (1) *TCP Throughput*: the amount of data transferred by TCP divided by the duration of the TCP connection. For multiple TCP connections, it refers to the aggregate throughput. (2) *Average Number of Slow-starts*: the average number of TCP slow-starts among all TCP connections. (3) *Packet Overhead*: the total number of routing packets transmitted, including ROUTE ERRORS used by EPLN and BEAD. For DSR-Update, this metric includes ROUTE ERRORS used for cache updates.

5.4.2 Two Choices for Setting RTO and cwnd

Figure 5.6 and Figure 5.8 show TCP throughput, and Figure 5.7 and Figure 5.9 show the average number of slow-starts for the first set of experiments. For the 50-node scenarios with one TCP connection, using "old" values for RTO and cwnd improves TCP throughput by up to 17% for DSR with path caches with promiscuous mode and up to 21% for DSR with path caches without promiscuous mode, compared with using default values. As discussed in Section 6.2, reducing congestion window size may cause TCP to stop sending packets, because the sequence number of the next packet to be sent could be larger than that congestion window allows to send. Thus, TCP has to rely on retransmission timeout to recover from an idle state *if the packets already sent or ACKs are lost*. The smaller the RTO is, the faster TCP resumes transmission if no ACK arrives.

Using "old" values for RTO and cwnd reduces the average number of slow-starts by up to 70% for DSR with path caches with and without promiscuous mode. This is because using "old" window size reduces the occurrences of TCP entering an idle state and hence reduces timeouts. DSR with promiscuous mode has fewer timeouts than DSR without promiscuous mode, since promiscuous mode allows DSR to cache more routes, which helps salvage packets.

As traffic load increases, the improvement in throughput decreases. As analyzed in Section 5.2.1, when using default values, TCP spends more time in an idle state only if data packets or ACKs are lost. If an ACK arrives soon, this choice has less impact on throughput. The validity of cached routes plays an important role. As traffic load increases, FIFO evicts stale routes faster; thus more data packets or ACKs can be delivered. Thus, the improvement is not as high as that for low traffic load scenarios.

For the 100-node scenarios with one TCP connection, using "old" values improves TCP throughput by up to 21% and 29% for DSR with path caches with and without

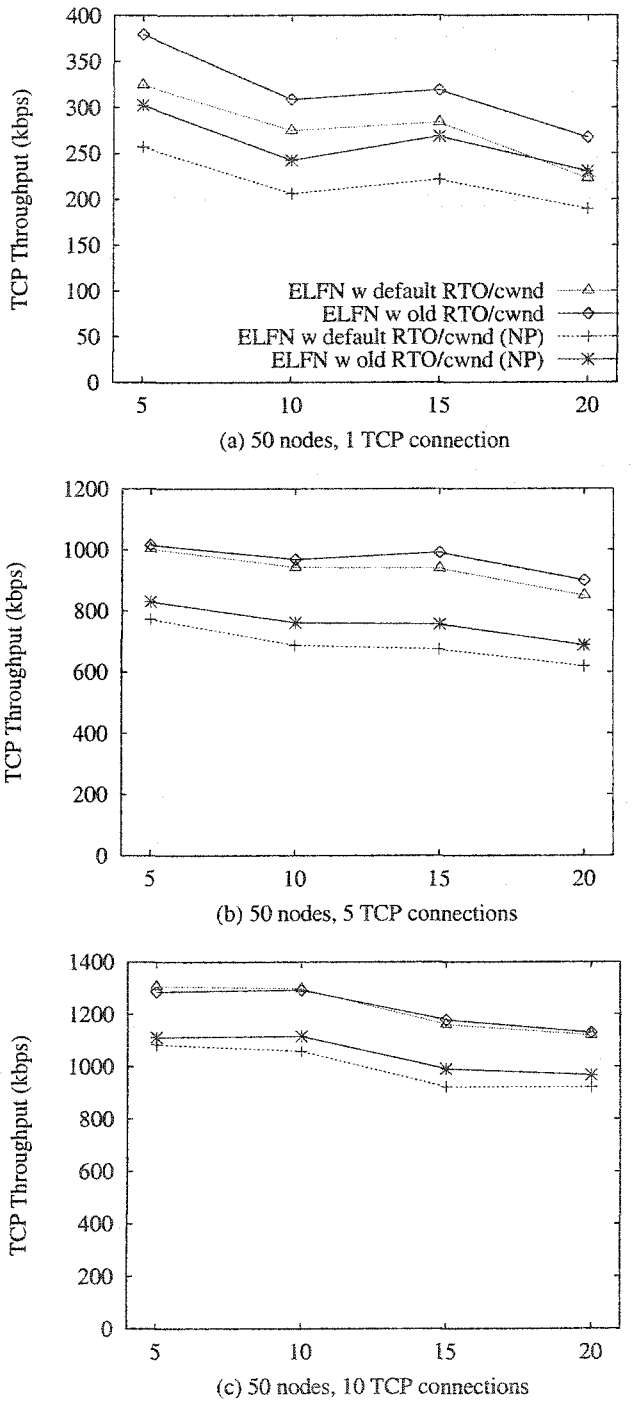


Figure 5.6: TCP Throughput vs. Mobility under Two Choices for Setting RTO and cwnd

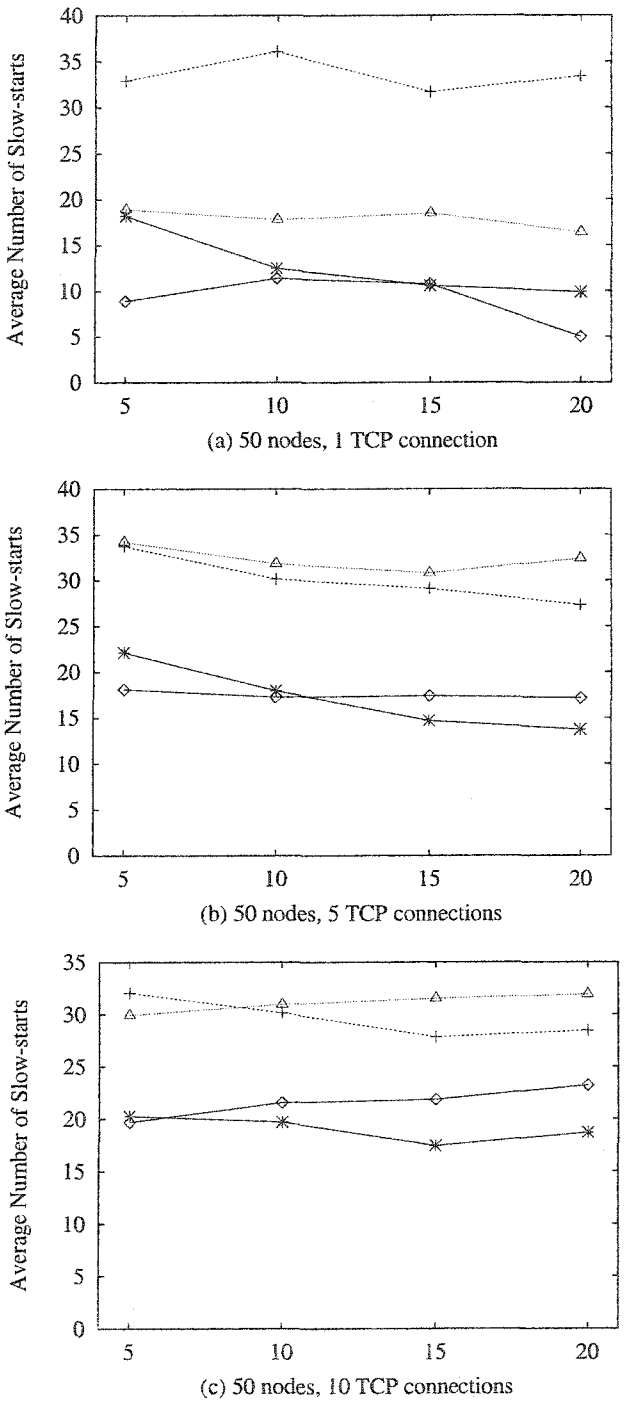


Figure 5.7: Average Number of Slow-starts vs. Mobility under Two Choices for Setting RTO and cwnd

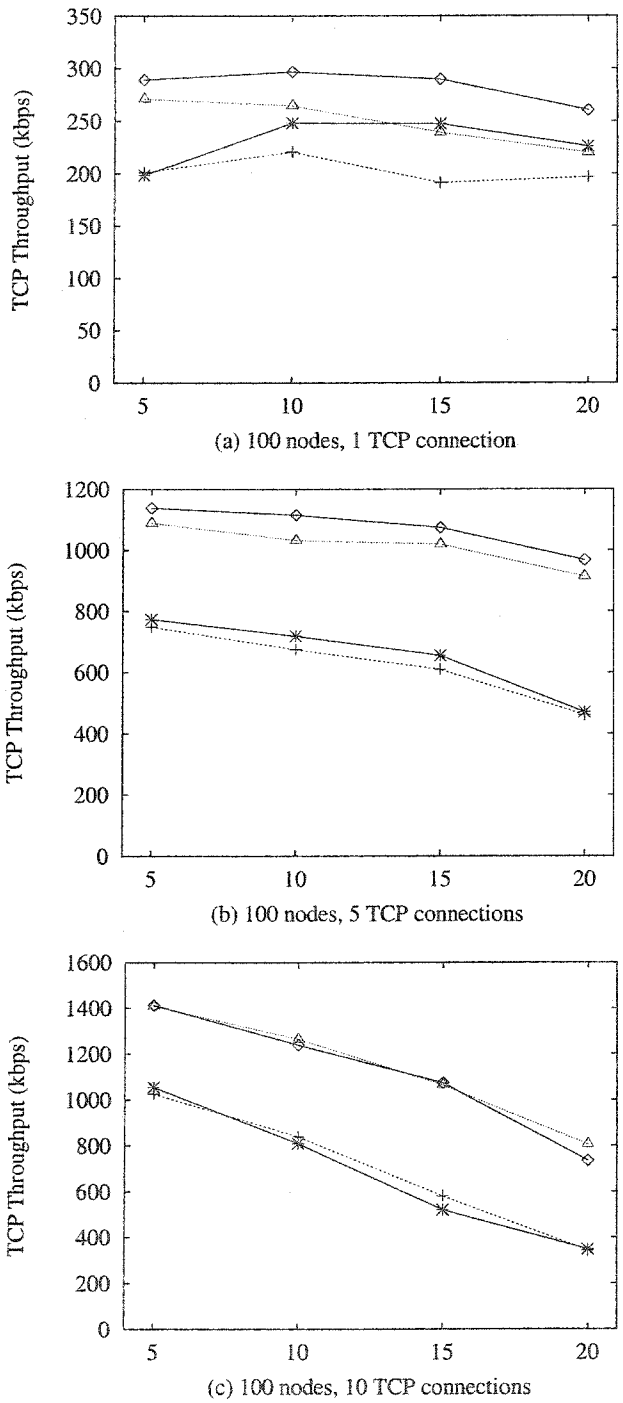


Figure 5.8: TCP Throughput vs. Mobility under Two Choices for Setting RTO and cwnd

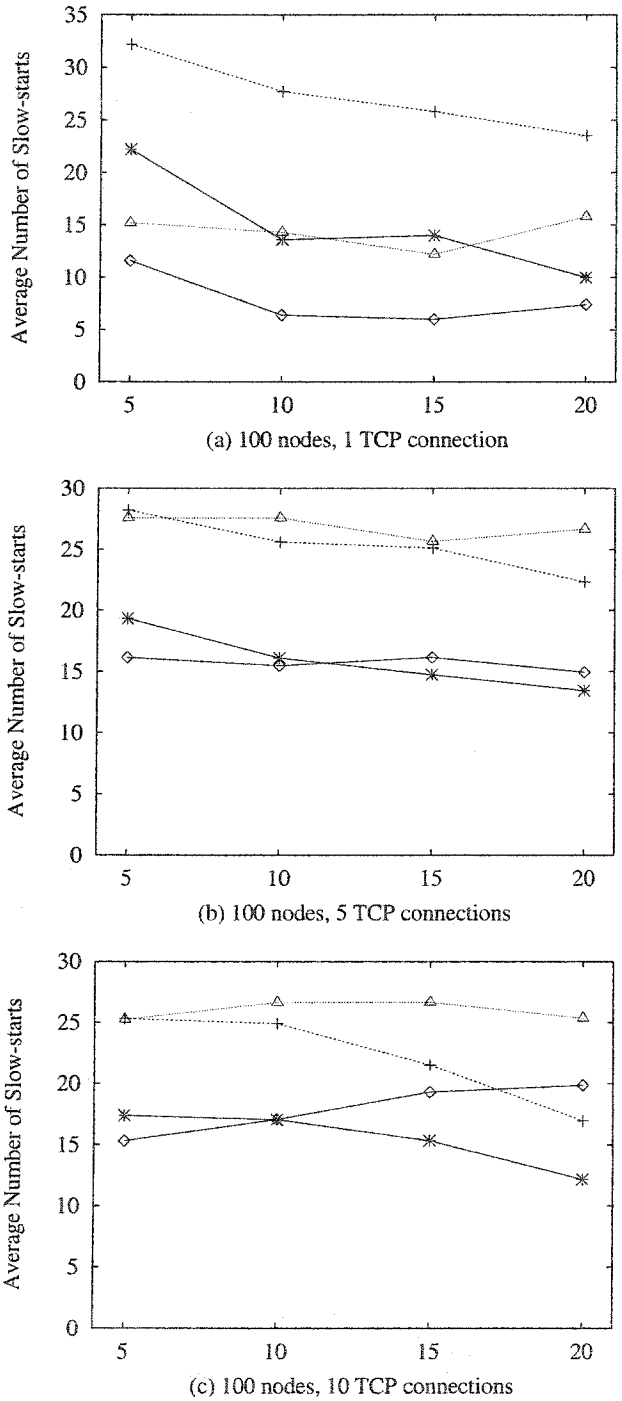


Figure 5.9: Average Number of Slow-starts vs. Mobility under Two Choices for Setting RTO and cwnd

promiscuous mode. Moreover, there is a large reduction in the average number of slow-starts. For the 10-connection scenarios, TCP throughput decreases slightly when using "old" RTO and cwnd values. I found that this choice causes more route discoveries than using default values because TCP is more aggressive to send packets. The overhead introduced by route discoveries results in more MAC contention, which somewhat offsets the improvement in throughput due to the fast recovery from the idle state. However, using "old" values reduces timeouts, whether for higher traffic load or larger network scenarios.

5.4.3 Evaluation Results of EPLN and BEAD

In this section, I present the results for the second set of experiments, in which I evaluated TCP performance for three combinations of mechanisms: TCP-ELFN, TCP-ELFN with EPLN and BEAD, TCP-ELFN with EPLN, BEAD, and DSR-Update.

TCP Throughput

Figure 5.10 and Figure 5.11 show TCP throughput. For the 50-node scenarios with 1 TCP connection, when used with DSR with path caches under promiscuous mode, EPLN and BEAD improve TCP throughput by up to 30% compared with TCP-ELFN. When used with DSR-Update, these two mechanisms improve throughput by 81% over TCP-ELFN at node mean speed of 20 m/s. Without promiscuous mode, EPLN and BEAD achieve the similar improvement. For example, TCP-ELFN with EPLN, BEAD, and DSR-Update outperforms TCP-ELFN by 63%; TCP-ELFN with EPLN and BEAD performs 70% better than TCP-ELFN. EPLN and BEAD also provide significant improvement over TCP-ELFN as traffic load increases. For example, for the 50-node scenarios with 5 TCP connections, without promiscuous mode, EPLN and BEAD improve TCP throughput by 24% with path caches and by 27% with DSR-Update.

For the 100-node scenarios, EPLN and BEAD achieve higher improvement. For example, under non-promiscuous mode, the maximum improvement is 173% for path caches and is 210% for DSR-Update, both at node mean speed of 20 m/s. Under promiscuous mode, the maximum improvement is 62%, the same for both caching strategies. Such improvement demonstrates the effectiveness of my mechanisms. The higher improvement in larger networks is due to this fact: as network size increases, nodes will cache more routes and thus will deliver more packet loss notifications and retransmit more ACKs for lost ACKs.

EPLN and BEAD with DSR-Update always outperform the two mechanisms with path caches under non-promiscuous mode. Due to on-demand Route Maintenance, a node is not notified when a cached route becomes stale until it uses that route to send packets. Thus DSR has delayed awareness of mobility. FIFO has little control of evicting which route at what time and therefore cannot quickly remove stale routes. In contrast, my cache update algorithm proactively notifies all reachable nodes that have cached a broken link to update their caches, thus enabling route caches to adapt fast to topology changes. Making caches more up-to-date not only reduces route failures and packet losses, but also allows EPLN and BEAD to use more valid routes, contributing to the higher improvement in throughput. For example, EPLN and BEAD with DSR-Update outperform the two mechanisms with path caches by up to 43% and 34% for the 50 and 100 node scenarios respectively.

Under promiscuous mode, EPLN and BEAD with DSR-Update outperform EPLN and BAED with path caches for the single TCP connection scenarios, and perform almost the same as the latter for the 5 and 10 TCP connection scenarios. I offer the following explanation for this observation. DSR with path caches caches the routes a node overhears in a secondary cache and the overheard routes learned from ROUTE REPLIES in a primary cache, whereas DSR-Update caches all overheard routes in a secondary

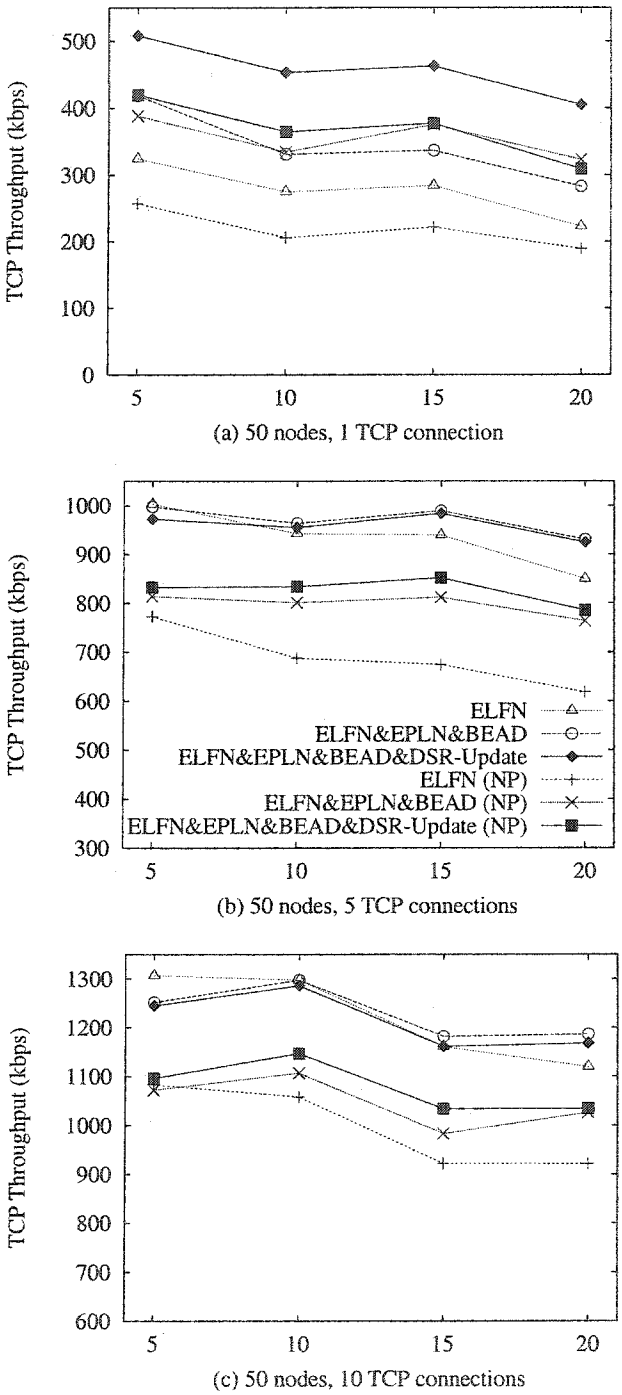


Figure 5.10: TCP Throughput vs. Mobility (mean speed (m/s))

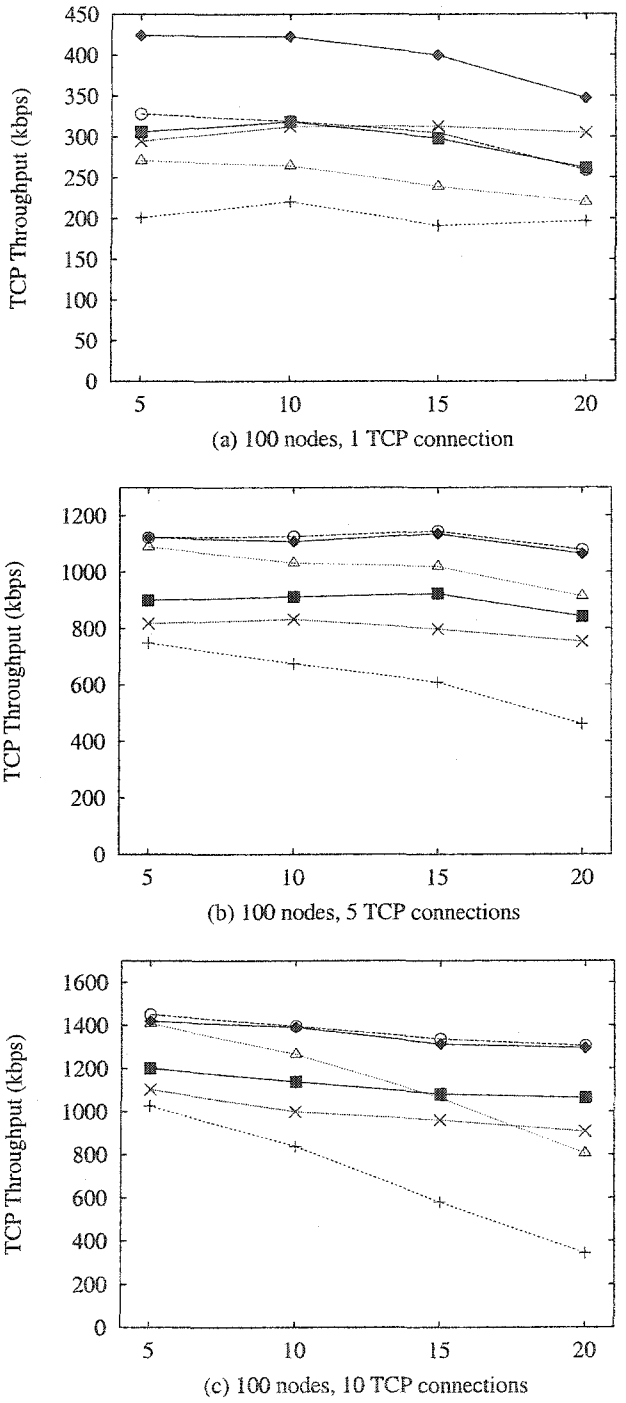


Figure 5.11: TCP Throughput vs. Mobility (mean speed (m/s))

cache. As traffic load increases, nodes will overhear more routes. EPLN and BEAD with path caches store more overheard routes and thus benefit more from this mode than EPLN and BEAD with DSR-Update.

Average Number of TCP Slow-starts

Figure 5.12 and Figure 5.13 show average number of TCP slow-starts. EPLN and BEAD reduce timeouts under both caching strategies and under both promiscuous and non-promiscuous mode. For example, for the 50-node scenarios with one connection, under promiscuous mode, EPLN and BEAD reduce timeouts by 60% with path caches and 90% with DSR-Update. Moreover, EPLN and BEAD with DSR-Update have less time-outs than EPLN and BEAD with path caches, giving reduction by 90% and 74% under promiscuous and non-promiscuous mode respectively.

When EPLN and BEAD use DSR-Update as a caching strategy, TCP performs slow-start and thus invokes congestion control mechanisms only twice during the 900 s simulation. These results not only show that EPLN and BEAD reduce TCP timeouts for mobility-induced losses, but also show that my cache update algorithm is very efficient in dealing with route failures.

As traffic load increases, EPLN and BEAD reduce timeouts by more than 35% with path caches and 40% with DSR-Update. As network size increases, EPLN and BEAD reduce timeouts by more than 33% with path caches and 44% with DSR-Update. EPLN actively delivers packet loss notifications and BEAD retransmits ACKs for lost ACKs in a best-effort way. Thus, TCP either starts retransmissions earlier or continues to advance congestion window without awareness of lost ACKs.

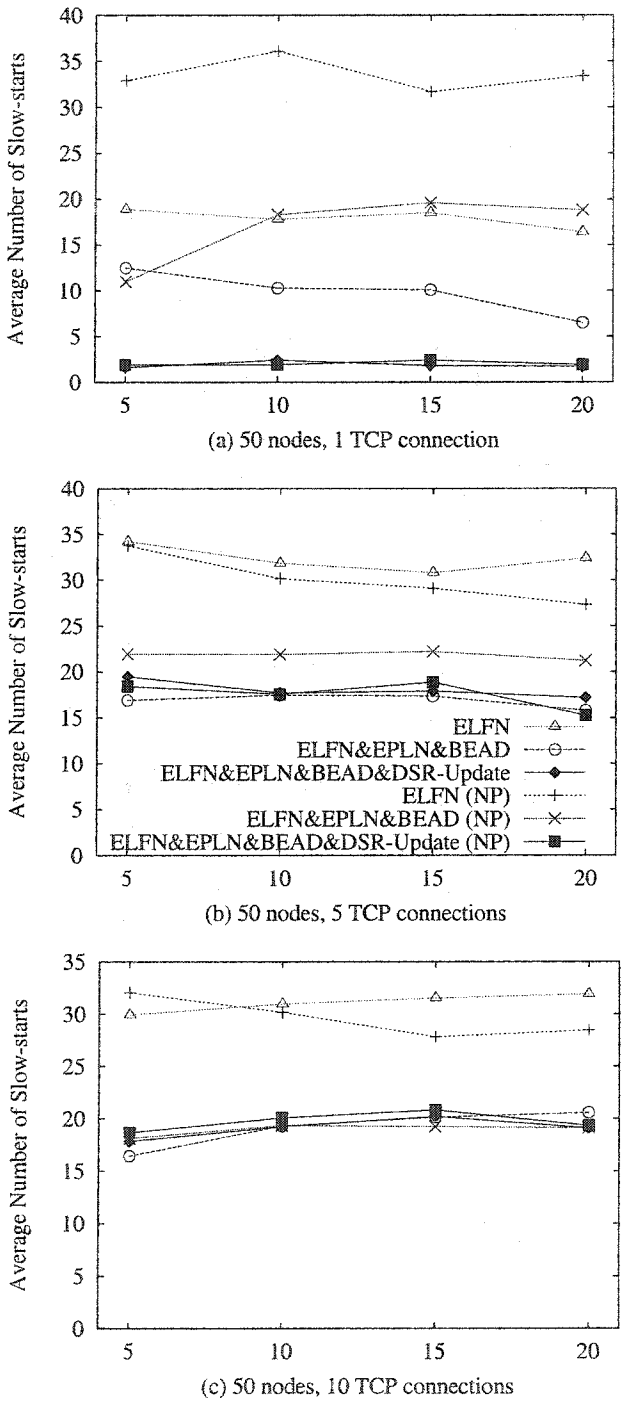


Figure 5.12: Average Number of Slow-starts vs. Mobility (mean speed (m/s))

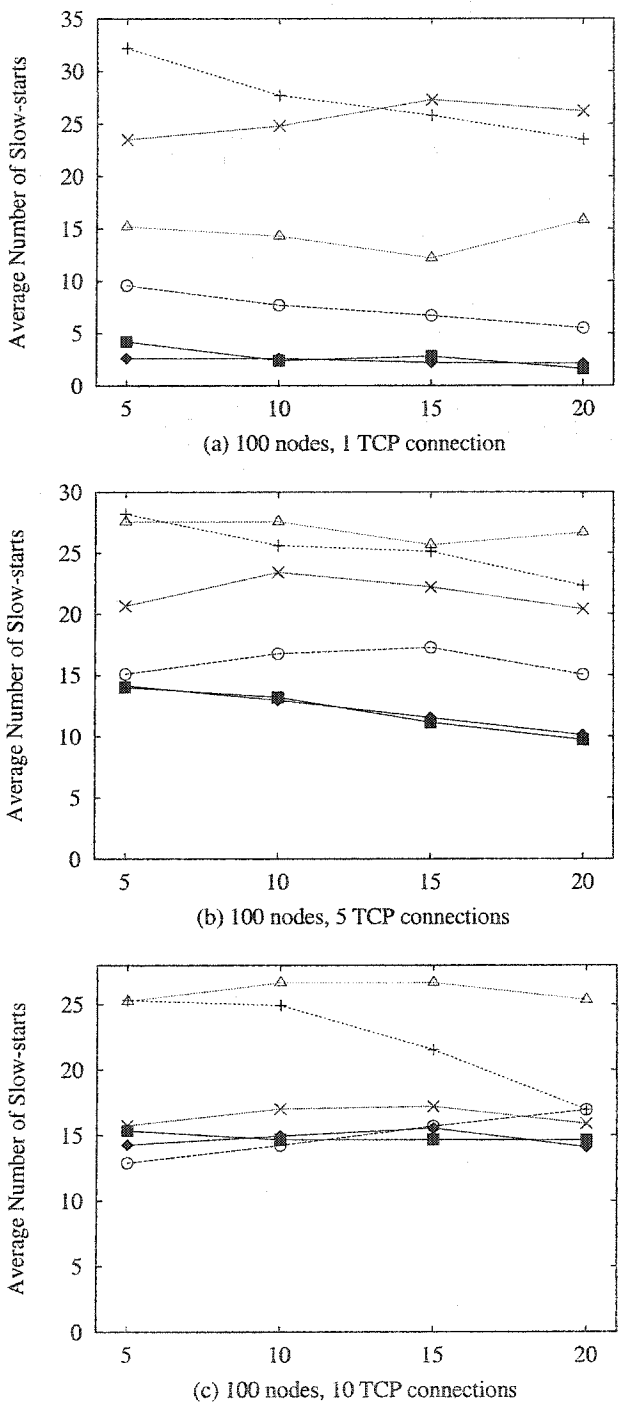


Figure 5.13: Average Number of Slow-starts vs. Mobility (mean speed (m/s))

Packet Overhead

Figure 5.14 and Figure 5.15 show packet overhead. For the 50-node scenarios with one connection, TCP-ELFN with EPLN and BEAD has a higher overhead than TCP-ELFN under both caching strategies due to packet loss notifications. As traffic load increases, the overhead of TCP-ELFN increases faster than that of TCP-ELFN with EPLN and BEAD and is higher than the latter for the 10 connections at node speed of 20 m/s. For these scenarios, DSR with path caches initiates more route discoveries than DSR with EPLN and BEAD. As mobility increases, routes break more frequently, and therefore more route discoveries take place; as traffic load increases, more routes need to be stored, and the path cache's FIFO replacement speeds up cache turnover. FIFO evicts many valid routes because of the small cache size.

When incorporated into DSR, EPLN and BEAD also use path caches, but DSR with EPLN and BEAD has lower overhead than DSR without them for high traffic load and high mobility scenarios. This is because EPLN and BEAD actively detect and evict stale routes due to the extensive use of cached routes.

I confirm this analysis through the results for the 100-node scenarios shown in Figure 5.15. For one TCP connection and without promiscuous mode, TCP-ELFN has a lower overhead than EPLN and BEAD with DSR-Update under low mobility, but has a higher overhead than the latter under high mobility. The higher overhead is due to the small cache size, which cannot hold all useful routes and thus results in more route discoveries, even under low traffic load. Under promiscuous mode, the overhead of TCP-ELFN decreases because DSR uses a secondary cache to store more routes, which helps reduce route discoveries. For the 100-node scenarios, the overhead of TCP-ELFN increases under high mobility. For 10 TCP connections, TCP-ELFN has a higher overhead than TCP-ELFN with EPLN and BEAD under both caching strategies.

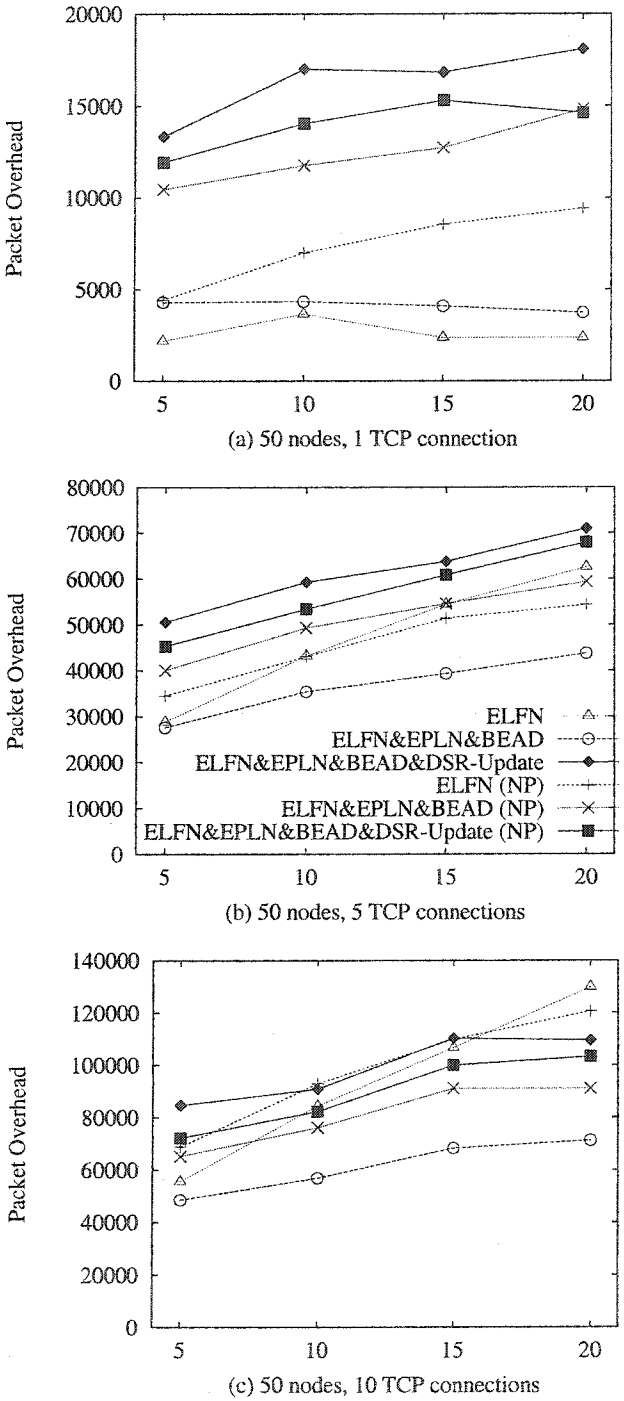


Figure 5.14: Packet Overhead vs. Mobility (mean speed (m/s))

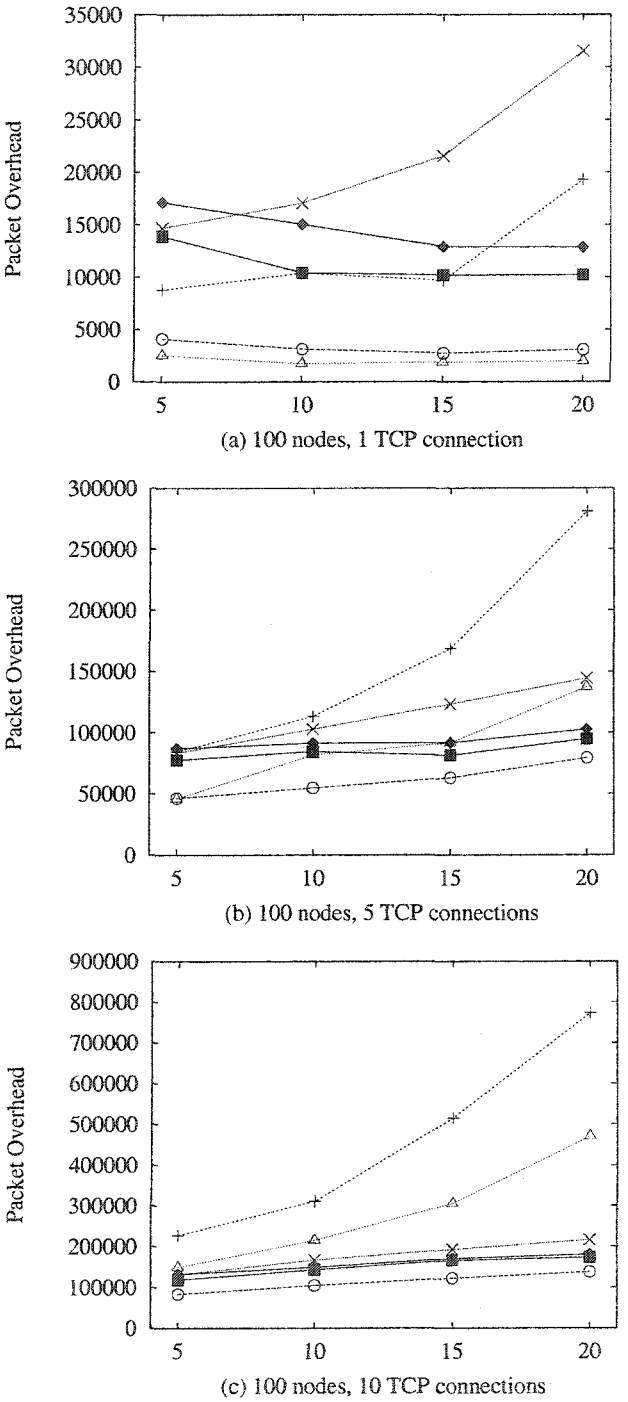


Figure 5.15: Packet Overhead vs. Mobility (mean speed (m/s))

DSR-Update dynamically adjusts its cache size as needed: the cache size increases as new routes are discovered and decreases as stale routes are removed. Thus, the cache size adapts to mobility, traffic load, and network size. As shown in Figure 5.15 (c), under non-promiscuous mode, EPLN and BEAD with DSR-Update have the lowest overhead. Under promiscuous mode, EPLN and BEAD with DSR-Update have a higher overhead than EPLN and BEAD with path caches because of cache update notifications. Under promiscuous mode, EPLN and BEAD obtain the maximum reduction in overhead for the 10 TCP connection scenarios: 71% with path caches and 62% with DSR-Update.

5.5 Conclusions

In this chapter, I presented a detailed study of how mobility affects TCP. I proposed to make routing protocols aware of lost TCP packets and help reduce TCP timeouts for mobility-induced losses. To achieve this goal, I presented two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD).

I made several observations through simulation of TCP-ELFN. First, I found that, when congestion control mechanisms are restored, keeping TCP's state the same as it was when TCP was frozen improves TCP throughput when traffic load is not high, and significantly reduces TCP timeouts compared with using default values. Second, I found that there is a trade-off between freezing TCP upon route failures and upon packet losses; freezing TCP upon route failures reduces TCP timeouts. Finally, I found that it is insufficient to notify TCP only about link failures, because many packets are dropped from the network interface queue without experiencing link failure detection. TCP will time out because of these losses. Upon route failures, ACKs are also dropped silently; therefore, TCP has to wait for timeouts.

EPLN and BEAD reduce TCP timeouts for mobility-induced losses by exploiting cross-layer information awareness. With EPLN, intermediate nodes seek to notify TCP senders about lost packets so that TCP can start retransmission earlier. With BEAD, intermediate nodes or TCP receivers retransmit ACKs for lost ACKs in a best-effort way. Both mechanisms extensively use cached routes, without initiating route discoveries at any intermediate node. The two feedback mechanisms are applicable to any routing protocol, as they address general problems that occur at the network layer.

I incorporated EPLN and BEAD into DSR with path caches and into DSR with my cache update algorithm. I show that, compared with TCP-ELFN, EPLN and BEAD significantly improve throughput and reduce timeouts. Moreover, EPLN and BEAD with my algorithm outperform EPLN and BEAD with path caches, because proactive cache updating is more efficient than FIFO in removing stale routes.

My results lead to the following conclusions:

- Cross-layer information awareness is key to making TCP efficient in the presence of mobility. It is necessary for the network layer to notify TCP senders about lost packets and to retransmit ACKs for lost ACKs, so that TCP reacts quickly to frequent packet losses and is unaware of lost ACKs.

- It is important to make route caches adapt fast to topology changes, because the validity of cached routes affects not only TCP performance but also the effectiveness of the mechanisms used to improve TCP performance, whether at the network layer or cross-layer.

Chapter 6

The Impact of Caching Strategies on the Scalability of On-Demand Routing Protocols

In this chapter, I investigate the impact of caching strategies on the scalability of on-demand routing protocols in the context of DSR. Prior work on the scalability of routing protocols mainly focused on making routing protocols scale with network size. The scalability of on-demand routing protocols with mobility has not been studied. I study the scalability of DSR with mobility, also examining varying levels of traffic load and network size. I consider three caching strategies proposed for DSR: path caches with FIFO, link caches with adaptive timeout mechanisms, and cache tables with my distributed cache update algorithm. Simulation results show that the distributed cache update algorithm makes DSR scale significantly better with mobility.

6.1 Introduction

Scalability is an important design goal of routing protocols. Prior work mainly focused on making protocols scale with network size [26, 50, 3, 35, 11]. Moreover, most work focused on the scalability of proactive routing protocols [26, 50, 49], and few work studied the scalability of on-demand routing protocols. Aron and Gupta [3] investigated the impact of error prevention and recovery on the scalability of DSR. Their analytical study shows that some local error recovery mechanism is needed to deal with route failures in order for DSR to scale with network size. Lee *et al.* [34] studied the scalability of AODV (Ad hoc On-demand Distance Vector routing protocol) [44] with network size.

Besides the studies of the scalability of routing protocols, there are two important theoretical results for the scalability of ad hoc networks. Gupta and Kumar [18] analyzed the asymptotic capacity of static ad hoc networks. They showed that as the number of nodes, n, increases, the maximum achievable per node throughput decreases approximately as $1/\sqrt{n}$. However, they did not consider the effect of mobility. Grossglauser and Tse [17] showed that per node throughput can increase dramatically when nodes are mobile. However, this result was obtained under several idealistic assumptions; one of the assumptions is that very long end-to-end delays are tolerable.

More recently, Santivanez *et al.* [49] proposed a theoretical framework for analyzing the scalability of routing protocols. They developed the first asymptotic expressions of total overhead that reflect the effect of traffic load and network size on a set of protocols, including plain flooding (no routing), proactive routing protocols (Standard Link State, or SLS), reactive routing protocols (DSR), hybrid routing protocols (Zone Routing Protocol [19, 20], or ZRP), hierarchical routing protocols (Hierarchical Link State [48], or HierLS), and limited dissemination (Hazy Sighted Link State [50], or HSLS). For SLS, HierLS, and HSLS, the expressions of total overhead also reflect the effect of the rate of topology changes. For DSR and ZRP, the authors only derived a

lower bound of total overhead, without considering the effect of mobility. Moreover, the result for DSR was obtained without the route cache option. Route caches are critical for on-demand routing protocols, because such protocols use them to make routing decisions, and not using route caches introduces significant overhead. It has been shown that caching strategies significantly affect the performance of DSR [38, 23, 39, 37, 58].

In this chapter, I study the impact of caching strategies on the scalability of on-demand routing protocols with mobility, also examining varying levels of traffic load and network size. It is essential for routing protocols to scale not only with network size but also with mobility. If a routing protocol cannot scale with mobility, its performance will degrade as mobility increases even for medium-scale networks. It is very challenging for routing protocols to scale with mobility because of frequent topology changes. I seek to answer two questions:

1. What factors limit the scalability of on-demand routing protocols with mobility?

2. How can such protocols be made to scale with mobility?

Route caches are the component that is most directly affected by mobility. Due to mobility, cached routes easily become stale. To address the cache staleness issue, prior work in DSR used heuristics with ad hoc parameters to predict the timeout of a link or a route [23, 39, 37]. I designed a distributed cache update algorithm, described in Chapter 3, to make route caches adapt quickly to topology changes without using ad hoc parameters. I have shown that the algorithm outperforms DSR with path caches [6] and with *Link-MaxLife* [23].

There are several definitions of scalability. Santivanez *et al.* [49] defined the scalability of a routing protocol as "the ability of a routing protocol to support the continuous increase of network parameters without degrading network performance." Network parameters refer to factors such as traffic load, network size, mobility rate. Arpacioglu *et*

al. [4] provided four definitions of scalability: absolute scalability, optimal scalability, relative scalability, and weak scalability. A protocol is termed *absolutely scalable* with respect to a given (environment, network parameter p, metric m) triple if the efficiency of the network does not vanish as the parameter tends to infinity. Metric m refers to factors such as 1/throughput, latency, overhead. A protocol is termed as *optimally scalable* with respect to a given triple if no other protocol is more scalable with respect to the same triple. This definition characterizes the best achievable scaling properties. Protocol A is termed to be *more scalable* than protocol B with respect to a given triple, if as p approaches infinity, the limit of $m(A)/m(B)$ is zero. If the limit is a positive constant, then protocol A and protocol B are termed *equally scalable*. Since a network parameter may not grow arbitrarily large in ad hoc networks, the authors also defined *weak scalability*. Protocol A is termed *more weakly scalable* than protocol B with respect to a given triple, if the growth rate of $m(A)$ is slower than $m(B)$ within a range of the network parameter.

According to the above definition of relative scalability, a protocol may be more scalable than another with respect to one metric, but less scalable with respect to another metric. Moreover, two protocols are termed as equally scalable if their throughput decreases or latency increases at the same rate, although one protocol may have higher throughput or lower latency than another. I believe that it is necessary to use both performance and overhead in scalability comparison, because the growth rate of overhead itself cannot reflect how well a protocol performs. Thus, I give a new definition of relative scalability, in which I use multiple metrics and the absolute value for throughput and latency. As traffic load, network size, or mobility increases, if protocol A not only has a slower overhead growth rate but also has better performance than protocol B, then A is more scalable than B. Performance refers to throughput and latency.

I perform the study in the context of DSR through extensive simulations. I consider three caching strategies: path caches with FIFO, link caches with *Link-MaxLife*, and

cache tables with my cache update algorithm. To evaluate the scalability of DSR with mobility, I perform two sets of experiments: fix node mean speed as 10 m/s and change pause time, the period during which a node remains static; and fix pause time as 0 s and change node mean speed. Most prior work chose node speed randomly between 0 m/s and 20 m/s. This choice results in node mean speed of 4 m/s due to the use of zero minimum speed [54]. I used more challenging mobility scenarios than those used in previous studies: node mean speed varies from 5 m/s to 20 m/s.

The results and findings can be summarized as follows:

- Fixed cache size for path caches limits the scalability of DSR with mobility. I also found that DSR with path caches performs worse under high traffic load and large networks. As traffic load, network size, or mobility increases, more routes need to be stored, but fixed cache size cannot hold all discovered routes and thus results in significant overhead due to route re-discoveries. Therefore, it is necessary to make the cache size adaptive.

- Adaptive timeout mechanisms expire valid links and keep stale ones because topology changes are unpredictable. Removing valid routes results in overhead due to route re-discoveries; using stale routes not only increases overhead but also degrades performance. Both problems limit the scalability of DSR with mobility.

- The distributed cache update algorithm makes DSR scale significantly better with mobility because of fast and efficient cache updating. I conclude that making route caches adapt quickly and efficiently to topology changes is key to the scalability of on-demand routing protocols with mobility.

The contributions of this work are threefold. First, I give a new definition of relative scalability. This work is the first that uses both performance and overhead in scalability

comparison. Second, I identify the factors that limit the scalability of on-demand routing protocols with mobility. Third, I show how to make such protocols scale with mobility.

The organization of this chapter is as follows. In Section 6.2, I analyze the impact of caching strategies on the scalability of DSR. In Section 6.3, I present simulation results. Finally, in Section 6.5, I present my conclusions.

6.2 The Impact of Caching Strategies on the Scalability of DSR

In this section, I first review the adverse effects of stale routes. I then analyze the impact of caching strategies on the scalability of DSR with mobility at different levels of traffic load and network sizes.

6.2.1 The Adverse Effects of Stale Routes

As discussed in Chapter 3, stale routes have three adverse effects:

- Causing packet losses if packets cannot be salvaged by intermediate nodes.
- Increasing latency, since the MAC layer goes through multiple retransmissions before concluding a link failure.
- Increasing routing overhead, since the node detecting a link failure needs to send a ROUTE ERROR to the source node.

As traffic load, network size, or mobility increases, these effects will become more significant. As mobility increases, more routes will become stale; as traffic load increases, stale routes will affect more traffic sources; as network size increases, more nodes will cache stale routes. Stale routes degrade performance and increase overhead;

therefore, they limit the scalability of on-demand routing protocols, since I compare the scalability of two protocols by both performance and overhead.

6.2.2 Path Caches with FIFO

Two factors in path caches limit the scalability of DSR. First, FIFO cannot quickly remove stale routes. Second, cache size is fixed and small. Small caches help remove stale routes, but also remove valid ones. As traffic load, network size, or mobility increases, a small cache will cause route re-discoveries, because more routes need to be stored but a small cache cannot hold all useful routes. If cache size is large and there is no fast cache updating mechanism, more stale routes will stay in caches. It was shown that path caches with unlimited size perform much worse than caches with limited size, due to the large amount of ROUTE ERRORS caused by the use of stale routes [23].

It is necessary to make cache size adaptive so as to scale with traffic load, network size, and mobility. My insight can be generalized: if an ad hoc parameter limits the scalability of a routing protocol, it is necessary to make the parameter adaptive. Predetermined choices 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. It is important to make routing protocols to be adaptive in the real world.

6.2.3 Adaptive Timeout Mechanisms

Two factors in adaptive timeout mechanisms limit the scalability of DSR with mobility. First, heuristics cannot accurately estimate timeouts because topology changes are unpredictable. Second, such mechanisms use ad hoc parameters. As a result, valid links will be removed if timeouts are set too short, and stale links will be kept in caches if timeouts are set too long. Removing valid links causes route re-discoveries; keeping stale links not only incurs overhead but also degrades performance.

I give two examples of *Link-MaxLife*. *Link-MaxLife* predicts the stability of nodes based on observed link usages and breakages, and computes the timeout of a link using stability of endpoints. Suppose that node A learns that the link from node B to node C) is broken. Node A decreases the stability metric for B and C, not knowing which of the two nodes is unstable. For any link with node B as an endpoint, if the stability metric of node B is less than that of the other endpoint, node A decreases the lifetime of that link to be the stability metric of node B. Similarly, node A decreases the lifetime of the links with node C as an endpoint. However, the breakage of a link does not imply that both endpoints are unstable. If either node B or node C is stable, node A will expire some valid links. On the other hand, the usage of a link does not imply that both endpoints are stable. For example, when a link was last used, two endpoints were close to each other; when the link is used, one endpoint is moving away from the other endpoint. The link is unstable, but *Link-MaxLife* will increase the stability metric for both endpoints and therefore produce wrong timeout estimates. As mobility increases, links break more frequently; wrong timeout estimates will have more adverse effect on scalability.

6.2.4 Cache Tables with Distributed Cache Updating

Making route caches quickly and accurately reflect topology changes is critical to the scalability of on-demand protocols with mobility. Any mechanism aiming to keep route caches up-to-date needs to operate efficiently, without incurring much overhead. Proactive protocols do not scale well because topology updates are periodically propagated throughout the network. Thus, the challenge to scalability is that as network parameters increase, a protocol should provide satisfactory performance while maintaining a slow growth rate of overhead.

My cache update algorithm addresses the problems of path caches and adaptive timeout mechanisms. First, a cache table has no capacity limit and thus allows DSR to store

all discovered routes; the cache size decreases as stale routes are removed. The adaptive cache size enables DSR to scale with dynamic network parameters. Second, the algorithm enables DSR to adapt quickly to topology changes, thus reducing packet losses, delivery latency, and routing overhead. These benefits will become more significant as traffic load, network size, or mobility increases.

The algorithm has the following desirable properties with respect to scalability. First, cache update overhead is low, since the algorithm notifies only the nodes that have cached a broken link. Second, the algorithm is distributed and thus is scalable with network size. Finally, the algorithm does not use any ad hoc parameters and thus is adaptive to topology changes.

6.3 Simulation Methodology

I evaluated DSR with three caching strategies under promiscuous and non-promiscuous mode. Promiscuous mode affects both performance and overhead, and thus it affects scalability. Although *Link-MaxLife* was not designed to operate without promiscuous mode, factoring out the effect of this mode allows me to accurately evaluate the impact of caching strategies on scalability. When DSR runs in non-promiscuous mode, I did not use GRATUITOUS ROUTE REPLY since it relies on this mode. For DSR with my cache update algorithm and without promiscuous mode, I did not use GRATUITOUS ROUTE ERROR. When promiscuous mode is used, I used all optimizations for the three caching strategies.

I used the *ns-2* [12] network simulator with the Monarch Project's wireless and mobile extensions [6, 47]. The network interface uses the IEEE 802.11 DCF MAC protocol [25]. The mobility model is the *random waypoint model* [6]. I used this model because it is the most widely used mobility model in ad hoc network simulations. I

chose node speed randomly from $v \pm 0.1v$, for different values of v. For path caches and *Link-MaxLife*, I used the parameters recommended by Johnson *et al.* [31].

I performed two sets of experiments to evaluate the scalability of DSR with mobility. In the first set of experiments, I chose node speed randomly from 10 ± 1 m/s and used pause times of 0, 30, 60, 120, 300, 600 and 900 s. I also evaluated the effect of traffic load on the performance of DSR with different caching strategies. I used CBR traffic with 4 packets per second and packet size of 64 bytes [6]. I used 20 and 30 flows and did not use higher traffic load in order to avoid network congestion. In the second set of experiments, I used pause time 0 s and node mean speeds of 5 m/s, 10 m/s, 15 m/s and 20 m/s. Varying node mean speed presents more challenging mobility scenarios than varying pause time. In this set of experiments, I also studied the effect of network size on the performance of DSR by using 50, 100, and 150 node networks. I do not provide results for larger networks because I was not able to get results for 200-node networks since trace files exceed the file size limit.

I used three field configurations: a 1500 m $\times$ 500 m field with 50 nodes, a 2200 m $\times$ 600 m field with 100 nodes [45], and a 2200 m $\times$ 1000 m field with 150 nodes. Each data point represents an average of 10 runs of randomly generated scenarios.

I used five metrics: (1) *Packet Delivery Ratio*: the ratio of the number of data packets received by the destination to the number of data packets sent by the source; (2) *Packet Delivery Latency*: the delay from when a packet is sent by the source until it is received by the destination; (3) *Percentage of Good Replies Sent from Caches*: the percentage of ROUTE REPLIES sent by intermediate nodes that do not contain broken links; (4) *Packet Overhead*: the total number of routing packets transmitted (both sent and forwarded); and (5) *Normalized Routing Overhead*: the ratio of the number of routing packets transmitted to the number of data packets received. For DSR-Update, packet overhead and normalized routing overhead include ROUTE ERRORS for cache updates.

6.4 Simulation Results

6.4.1 Varying Node Pause Time

Figure 6.1 shows packet delivery ratio. DSR-Update outperforms DSR under both promiscuous and non-promiscuous mode. DSR-Update also outperforms *Link-MaxLife* under non-promiscuous mode. As mobility increases, DSR-Update has a slower decrease in packet delivery ratio than DSR with path caches and *Link-MaxLife*, especially under non-promiscuous mode. This is because DSR-Update removes stale routes more quickly than FIFO and predicting timeouts. I verify this claim through cache performance. As shown in Figure 6.2, DSR-Update provides better cache correctness than DSR with path caches and *Link-MaxLife*. Under non-promiscuous mode, DSR with path caches has worse cache performance than *Link-MaxLife* because predicting timeouts is more effective than FIFO. However, under promiscuous mode, DSR with path caches has better cache performance, since *Link-MaxLife* keeps more overheard stale links in the topology graph.

DSR-Update achieves higher improvement for higher traffic load scenarios. For example, it provides the maximum of improvement, 57%, over DSR and *Link-MaxLife* for 100n-30f. As traffic load increases, stale routes adversely affect more traffic sources, since each flow has to detect link failures on-demand. DSR-Update prevents more traffic sources from using stale routes and thus reduces more packet losses.

Figure 6.3 and Figure 6.4 show packet delivery latency. DSR-Update has lower latency than DSR with path caches under both promiscuous and non-promiscuous mode. Moreover, the reduction in latency increases as traffic load and mobility increases, because fast cache updating reduces link failure detections by multiple flows.

DSR-Update has lower latency than *Link-MaxLife* for 100n-20f. Although link caches help reduce latency due to fewer route discoveries, *Link-MaxLife* has higher

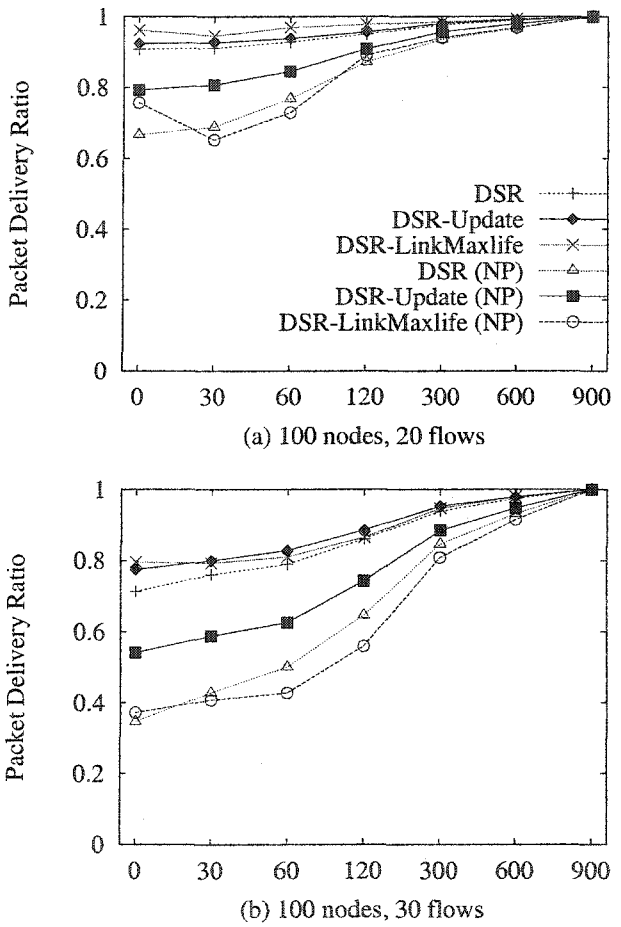


Figure 6.1: Packet Delivery Ratio vs. Mobility (Pause Time (s))

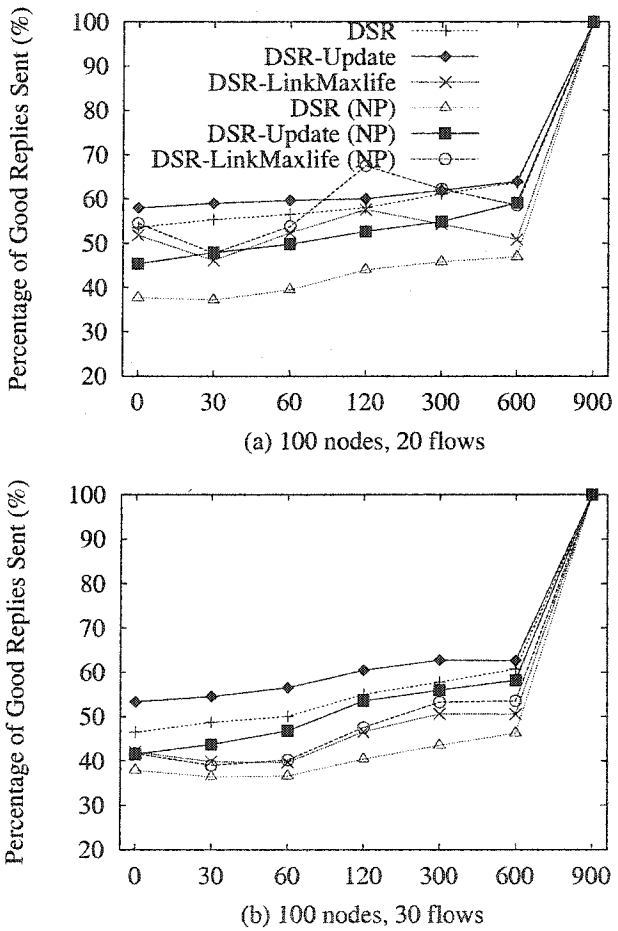


Figure 6.2: Percentage of Good Replies Sent from Caches vs. Mobility (Pause Time (s))

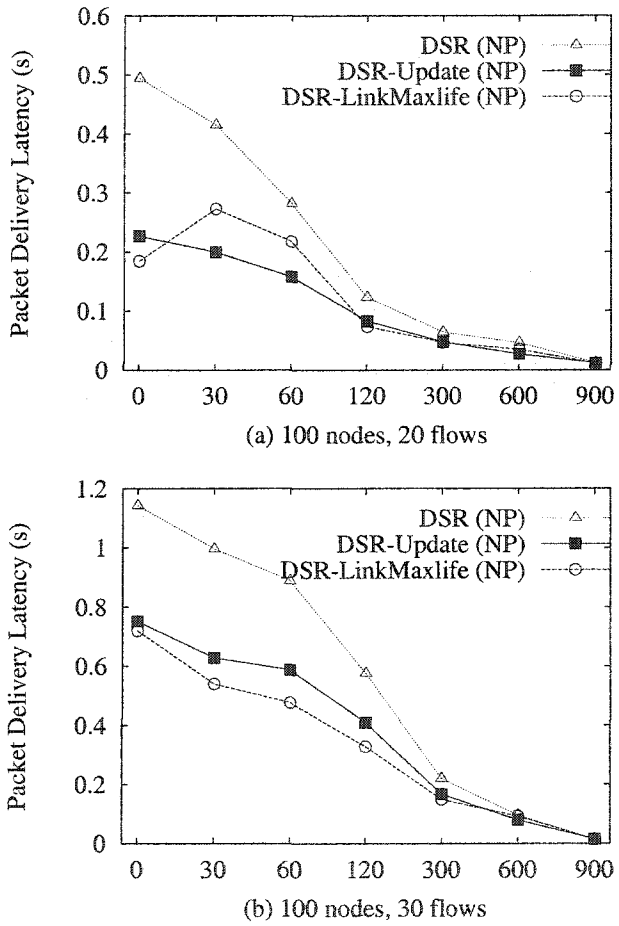


Figure 6.3: Packet Delivery Latency vs. Mobility (Pause Time (s))

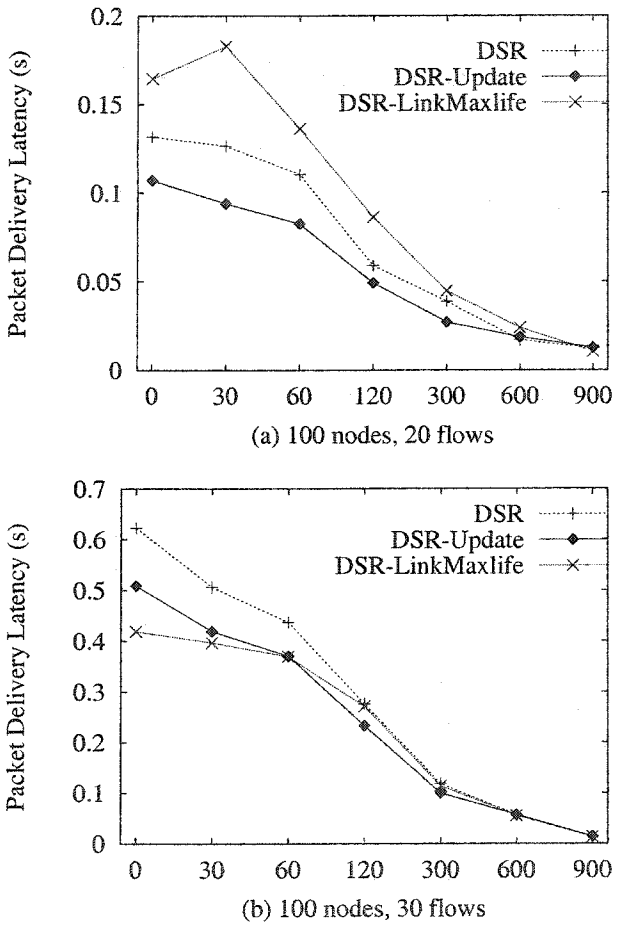


Figure 6.4: Packet Delivery Latency vs. Mobility (Pause Time (s))

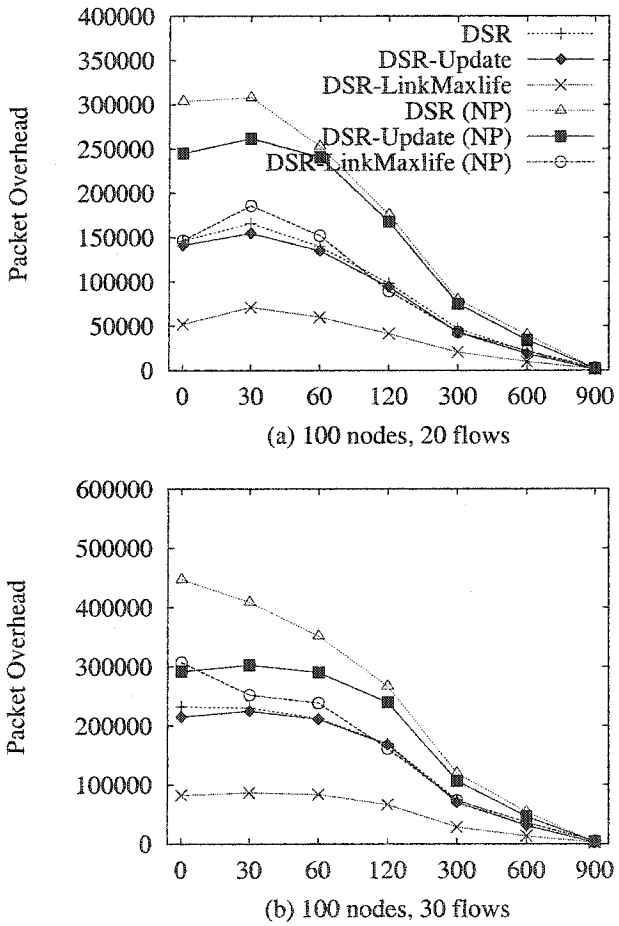


Figure 6.5: Packet Overhead vs. Mobility (Pause Time (s))

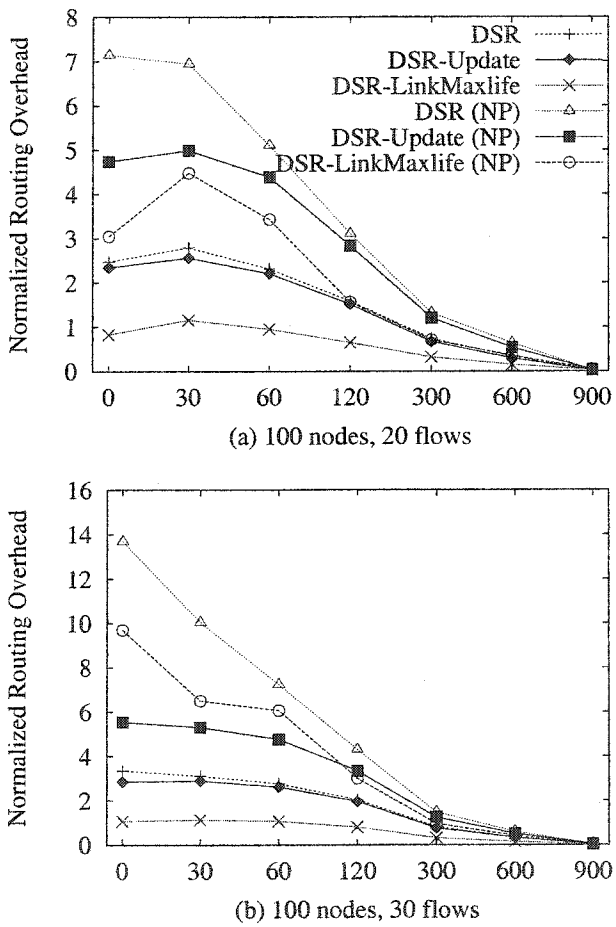


Figure 6.6: Normalized Routing Overhead vs. Mobility (Pause Time (s))

latency because packets are salvaged multiple times due to stale links. For 100n-30f, DSR-Update has higher latency than *Link-MaxLife*, since the latency due to route discoveries sometimes dominates.

Routing overhead is an important measure of scalability. Figure 6.5 shows packet overhead, and Figure 6.6 shows normalized routing overhead. As mobility increases, a scalable protocol should have a slow overhead growth rate. DSR-Update has a slower overhead growth rate than both DSR with path caches and *Link-Maxlife* under non-promiscuous mode. Under promiscuous mode, DSR-Update has a similar growth rate in overhead as DSR with path caches and *Link-Maxlife*. Moreover, DSR-Update has a lower overhead than DSR with path caches under non-promiscuous mode. For example, for 100n-30f at pause time 0 s, DSR-Update reduces packet overhead by up to 30% and normalized overhead by up to 60%. Such reduction is partly due to reduced ROUTE ERRORS caused by stale routes, but is mainly due to fewer route discoveries. DSR with path caches initiates a large number of route discoveries due to the small cache size. I will discuss more about this observation in the next section.

6.4.2 Varying Node Mean Speed

In this section, I present the results of the second set of experiments. In these experiments, I fixed node pause time as 0 s and changed node mean speed from 5 m/s to 20 m/s. Mobility rate here is much higher than that in previous studies.

Figure 6.7 shows packet delivery ratio. I first analyze the results for non-promiscuous mode. DSR-Update outperforms both DSR with path caches and *Link-MaxLife*, especially under high mobility. For example, DSR-Update provides 200% improvement for 100n-20f and 477% for 150n-20f, both at node mean speed of 20 m/s. Moreover, as mobility increases, the delivery ratio of DSR-Update decreases slowly, whereas the delivery ratio of DSR with path caches and *Link-MaxLife* degrades quickly.

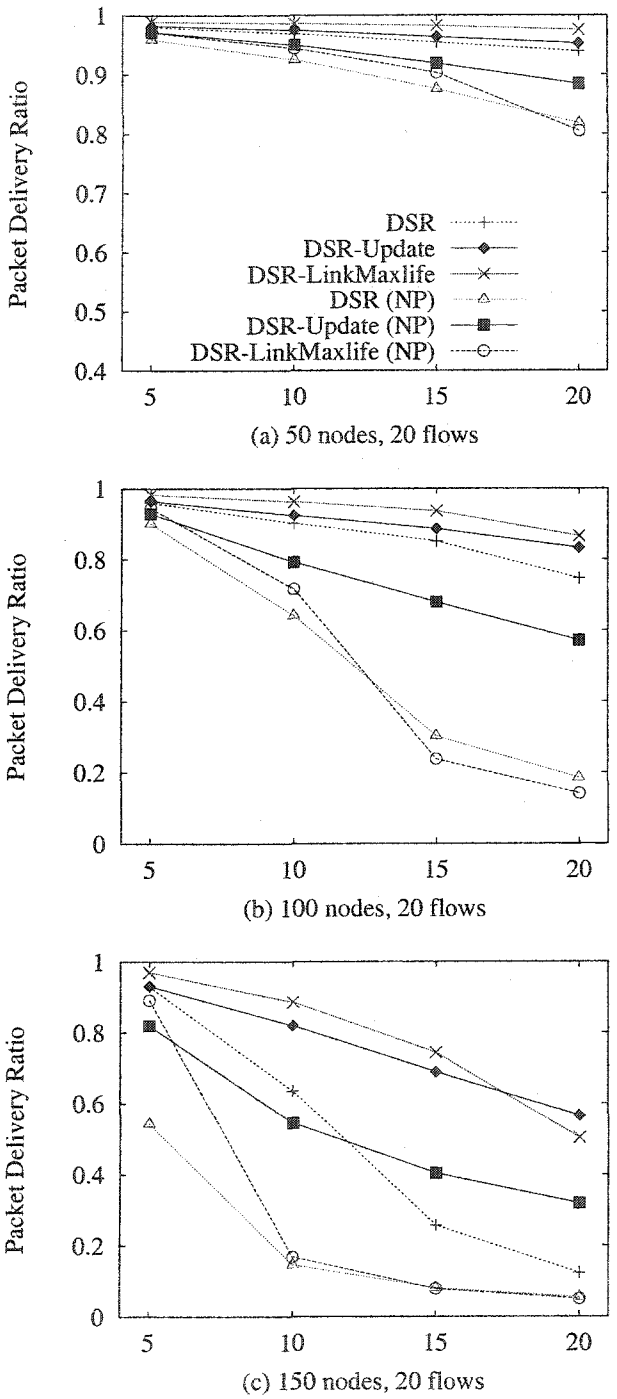


Figure 6.7: Packet Delivery Ratio vs. Mobility (Mean Speed (m/s))

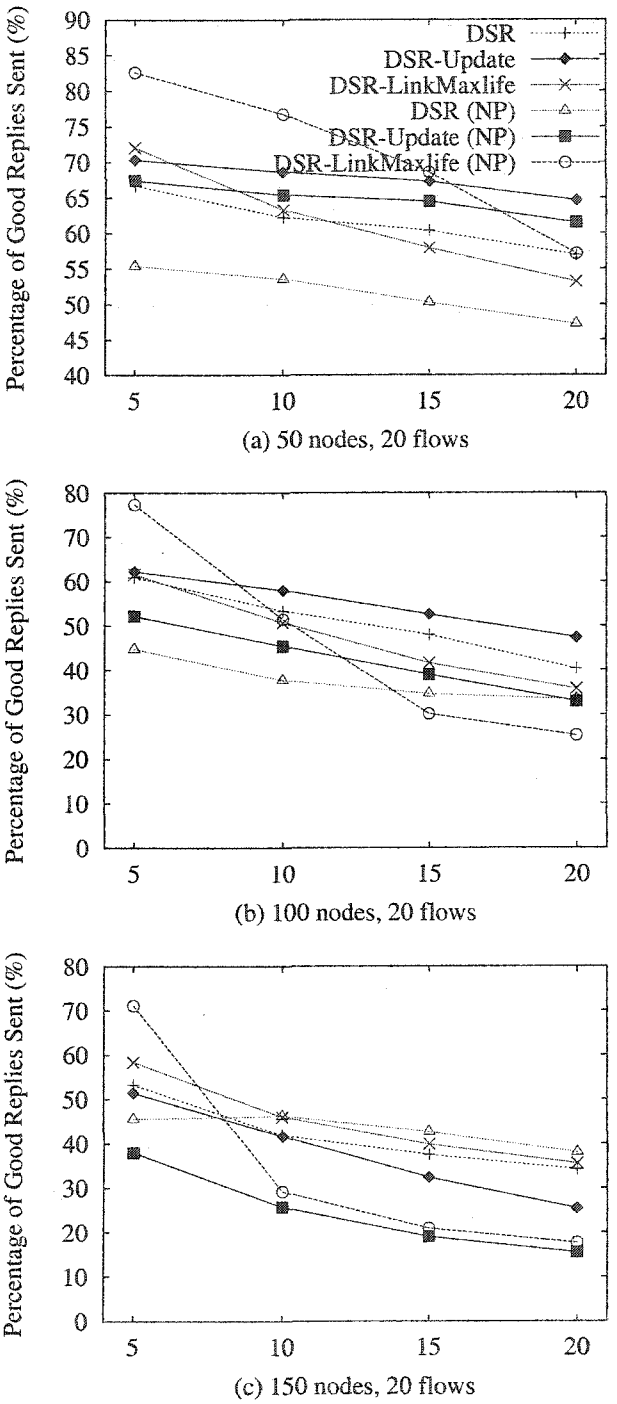


Figure 6.8: Percentage of Good Replies Sent from Caches vs. Mobility (Mean Speed (m/s))

Under promiscuous mode, DSR-Update outperforms DSR with path caches, and the improvement increases as network size and mobility increase. For 50 and 100 node networks, the improvement is not as high as that under non-promiscuous mode. DSR-Update stores overhead routes only in a secondary cache, whereas DSR with path caches stores such routes in both primary and secondary caches; therefore, DSR with path caches benefits more from promiscuous mode than DSR-Update. DSR-Update, however, provides higher improvement for 150 node networks, as shown Figure 6.7 (c). DSR with path cache's performance degrades quickly, because the small cache size cannot meet higher storage requirements and hence more route discoveries took place. DSR-Update performs worse than *Link-MaxLife* under low mobility but has slower decrease rate than *Link-Maxlife* under high mobility.

As mobility increases, the cache performance of DSR-Update degrades slowly. In contrast, the cache performance of *Link-MaxLife* degrades quickly under non-promiscuous mode because links break more frequently under high mobility, but *Link-MaxLife* cannot learn as much link failure information as it can under promiscuous mode. DSR-Update has better cache correctness than both DSR with path caches and *Link-MaxLife* for 50 and 100 node networks, as shown in Figure 6.8. For 150-node networks, DSR with path caches has better cache performance than DSR-Update and *Link-MaxLife*. This is because as network size increases, the small cache size causes faster cache turnover.

Figure 6.9 and Figure 6.10 show packet delivery latency under non-promiscuous and promiscuous modes. DSR-Update has lower latency than both DSR with path caches and *Link-MaxLife*. As shown in the first set of experiments, the reduction in latency increases as traffic load and mobility increase. Such advantages become more significant under more challenging mobility scenarios, such as at node mean speed of 20 m/s, and for larger networks, such as 150-node networks. Therefore, fast cache updating is important for the scalability of on-demand routing protocols.

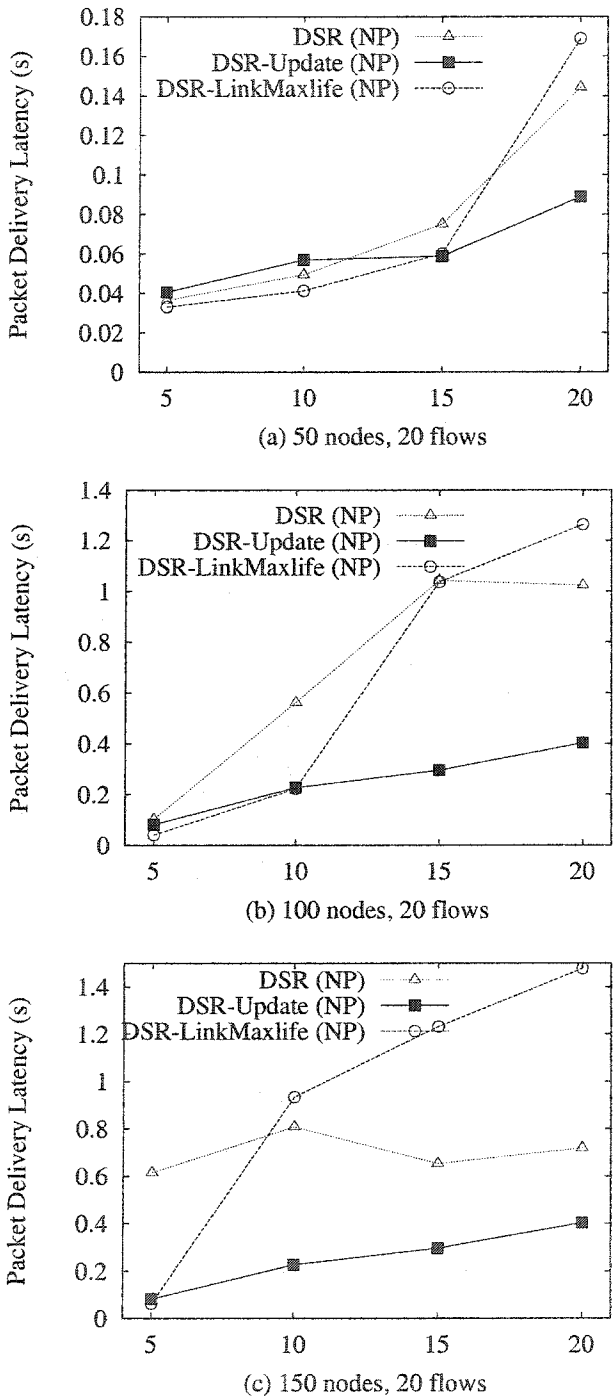


Figure 6.9: Packet Delivery Latency vs. Mobility (Mean Speed (m/s))

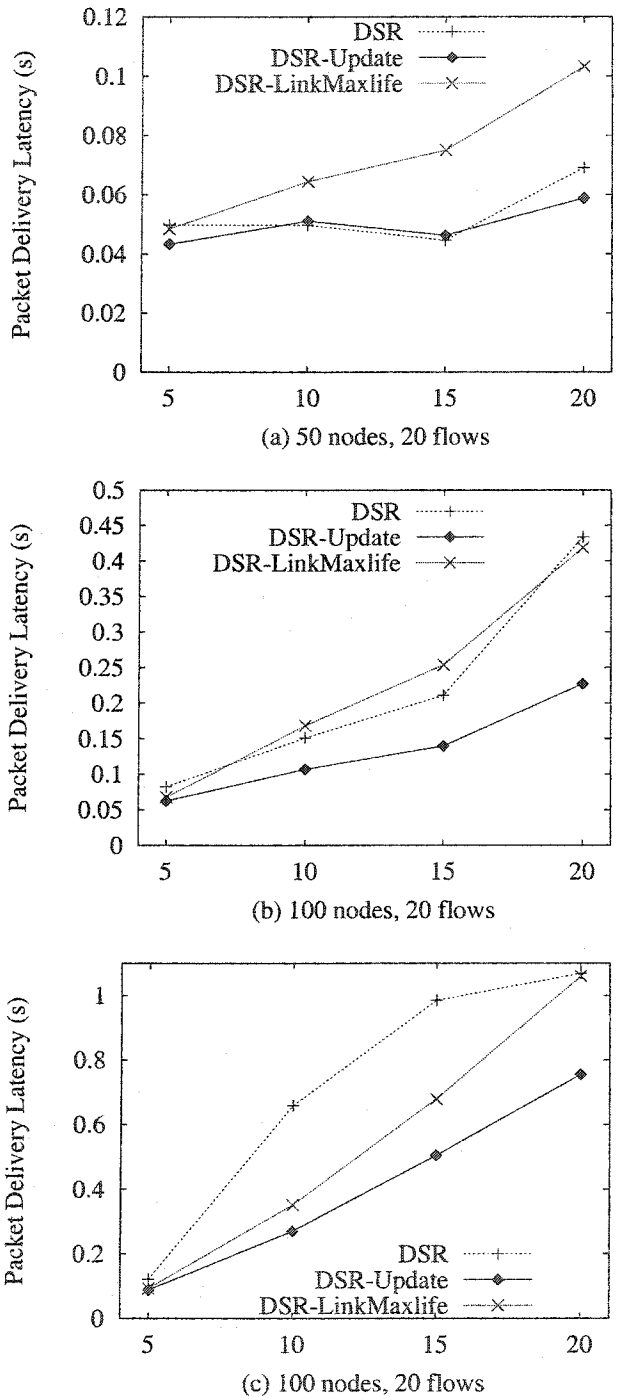


Figure 6.10: Packet Delivery Latency vs. Mobility (Mean Speed (m/s))

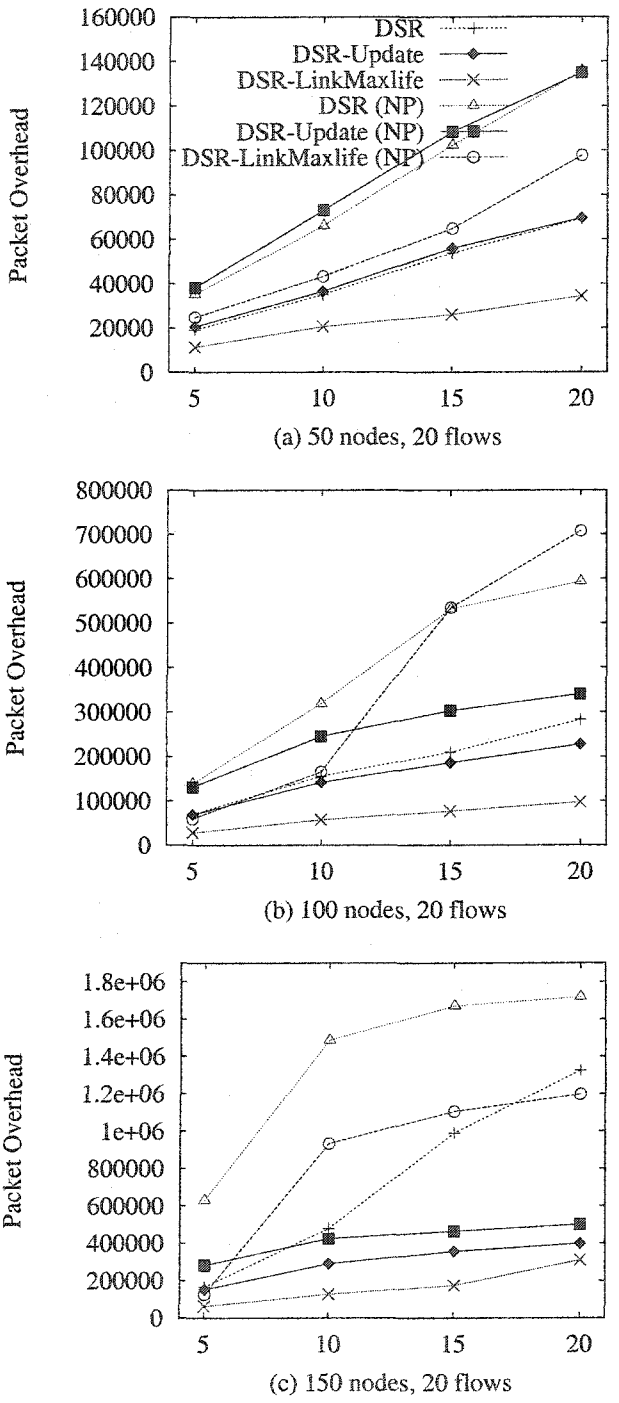


Figure 6.11: Packet Overhead vs. Mobility (Mean Speed (m/s))

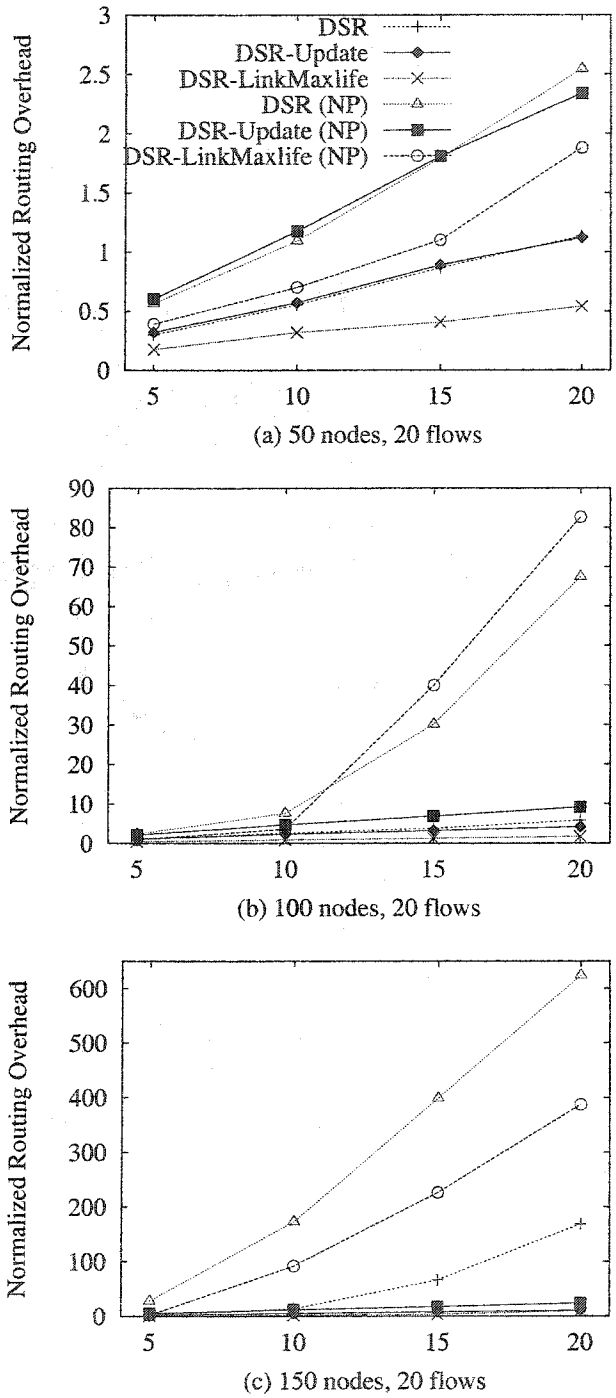


Figure 6.12: Normalized Routing Overhead vs. Mobility (Mean Speed (m/s))

Figure 6.11 shows packet overhead, and Figure 6.12 shows normalized routing overhead. For 50-node networks, DSR has a similar overhead as DSR-Update. For 100-node networks, DSR with path caches has a higher overhead than DSR-Update under promiscuous mode and higher overhead under non-promiscuous mode and high mobility. As network size increases to 150 nodes, DSR has a higher overhead under promiscuous mode. The overhead in DSR with path caches increases quickly as mobility and network size increase because of the small cache size. The primary cache of DSR has a capacity limit of 30 entries, and the secondary cache has a capacity limit of 34 entries; the secondary cache is used under promiscuous mode. These choices of cache size are suitable for 50-node networks, but unsuitable for 100-node or larger networks. As network size increases, more routes will be discovered; as mobility increases, routes break more frequently and thus more route discoveries take place. Small caches cannot hold all useful routes and thus cause a large number of route discoveries, as shown in Figure 6.13.

Link-MaxLife cannot accurately predict timeouts and thus keeps many stale links. The inaccurate prediction causes another problem: under non-promiscuous mode, *Link-MaxLife* expires many valid links under high mobility. Removing valid links results in overhead due to route re-discoveries, as shown in Figure 6.13.

In contrast, under non-promiscuous mode, DSR-Update has the lowest overhead under high mobility and for larger networks. Moreover, DSR-Update has the slowest overhead growth rate, because a cache table allows DSR to store all discovered routes, and the cache update algorithm quickly removes stale routes.

I measured the average cache size of DSR-Update, with the size sampled when a route is added or at least one route is deleted. The results are shown in Figure 6.14. The cache table size adapts well to traffic load, network size, and mobility. The average cache size under non-promiscuous mode for 100n-20f is almost the same as that under

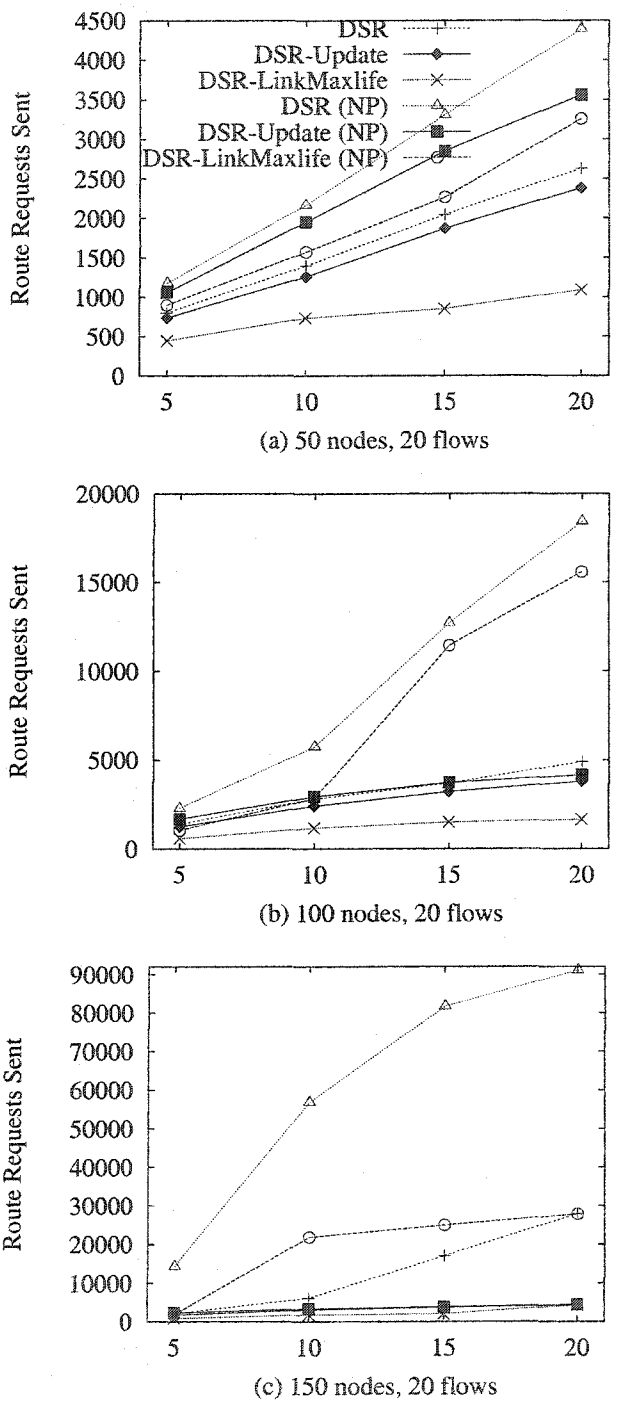


Figure 6.13: Route Requests Sent vs. Mobility (Mean Speed (m/s))

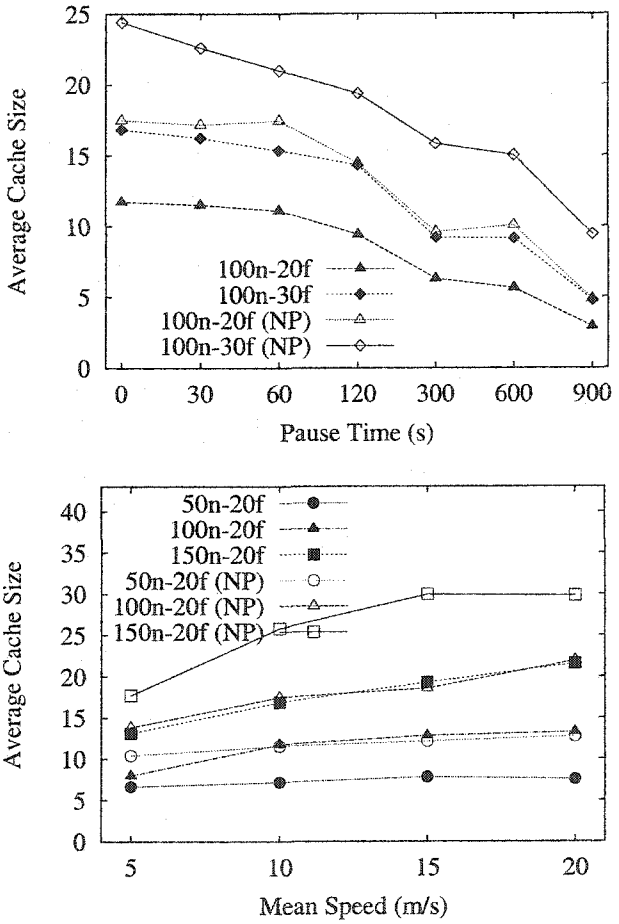


Figure 6.14: Average Cache Size of DSR-Update

promiscuous mode for 150n-20f; both are 20 under high mobility. These results further demonstrate that the capacity limit of 30 entries used in the primary cache is too small for 100-node or larger networks. The average cache size under non-promiscuous mode for 150n-20f is 30, showing that storage requirements increase quickly as network size increases. Therefore, it is necessary to make cache size adaptive for on-demand routing protocols to scale with dynamic network characteristics.

6.5 Conclusions

It is important for ad hoc network routing protocols to be scalable not only with network size but also with mobility. Prior work mainly focused on making routing protocols scale with network size. In this chapter, I studied the impact of caching strategies on the scalability of on-demand routing protocols with mobility, also examining varying levels of traffic load and network size. I performed the study in the context of DSR through extensive simulations. I considered three caching strategies proposed for DSR: path caches with FIFO, *Link-MaxLife*, and cache tables with my cache update algorithm.

My results and findings are summarized as follows:

- Fixed cache size limits the scalability of DSR with mobility. I also found that DSR performs worse under higher traffic load and larger networks. As traffic load, network size, or mobility increases, DSR initiates a large number of route discoveries because more routes need to be stored, but small caches cannot hold all useful routes. To be scalable, a routing protocol must maintain a slow growth rate of overhead as mobility increases. Therefore, it is necessary to make the cache size adaptive.

- Adaptive timeout mechanisms can expire valid links and keep stale ones due to unpredictable topology changes. Removing valid routes results in overhead due

to route re-discoveries; using stale routes not only increases overhead but also degrades performance. As mobility increases, links break more frequently; therefore, inaccurate timeout prediction will have more adverse effect on scalability.

- The distributed cache update algorithm makes DSR scale significantly better with mobility, because the cache table size adapts to mobility and the algorithm quickly removes stale routes.

All on-demand routing protocols use route caches. Route caches are the component that is most directly affected by mobility. As I have shown, stale routes limit the scalability of on-demand routing protocols with mobility. Mobility makes the scalability of on-demand routing protocols with network size and traffic load more difficult. Thus, it is important to make route caches quickly and accurately reflect topology changes. Moreover, a cache update mechanism should introduce as little overhead as possible. Therefore, I conclude that making route caches adapt quickly and efficiently to topology changes is key to the scalability of on-demand routing protocols with mobility.

Chapter 7

Conclusions

Mobility presents a fundamental challenge to routing and transport protocols in mobile ad hoc networks. In my thesis work, I have addressed the challenges mobility presents to on-demand routing protocols and to TCP. In this chapter, I summarize the contributions of my thesis and discuss future work.

7.1 Thesis Contributions

The first problem I addressed is the cache staleness issue of on-demand routing protocols. Due to mobility, cached routes easily become stale. Prior work addressing the cache staleness issue mainly used adaptive timeout mechanisms. However, heuristics with ad hoc parameters cannot accurately estimate timeouts because topology changes are unpredictable. To make route caches quickly adapt to topology changes, I proposed to proactively disseminate the broken link information to the nodes that have that link in their caches. I defined a new cache structure called a cache table and designed a distributed cache update algorithm. Each node maintains in its cache table the information necessary for cache updates. When a link failure is detected by some node, the

algorithm notifies in a distributed manner all reachable nodes that have cached that link. This algorithm is the first work that proactively updates route caches in on-demand ad hoc network routing in an adaptive manner. Moreover, the algorithm does not use any ad hoc parameters, thus making route caches fully adaptive to topology changes. I show that my algorithm outperforms DSR with path caches and with the *Link-MaxLife* link cache [23]. My solution is applicable to other on-demand routing protocols. I conclude that proactive cache updating is key to the adaptation of on-demand routing protocols to mobility.

The second problem I addressed is TCP performance in mobile ad hoc networks. TCP performance degrades significantly due to frequent route failures [21, 13, 51]. The first factor that adversely affects TCP is stale routes, which can cause TCP to experience repeated timeouts. Most attempts to improve TCP performance focused on transport layer mechanisms. I proposed a new approach to improve TCP performance at the network layer: reducing route failures by making route caches quickly adapt to topology changes. I investigated the impact of my cache update algorithm on TCP performance without any modification to TCP. I show that this algorithm significantly improves TCP throughput and reduces normalized routing overhead. I conclude that it is important to make route caches reflect topology changes quickly so that the adverse effect of mobility on TCP is reduced.

Making route caches more up-to-date reduces route failures; however, TCP still does not perform well because of frequent data and ACK losses. My solution is to exploit cross-layer information awareness. I proposed to make routing protocols aware of lost TCP packets and help reduce TCP timeouts for mobility-induced losses. To this end, I designed two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). EPLN seeks to notify TCP senders about packet losses so that TCP retransmits lost packets earlier. For lost ACKs, BEAD attempts to retransmit ACKs

either at intermediate nodes or at TCP receivers so that TCP is unaware of lost ACKs. Both mechanisms extensively use cached routes, without initiating route discoveries at any intermediate node. The two feedback mechanisms can be adapted to other routing protocols, as they address general problems that occur at the network layer. I evaluated TCP-ELFN enhanced with EPLN and BEAD using two caching strategies for DSR, path caches [6, 23] and my cache update algorithm. I show that TCP-ELFN with EPLN and BEAD significantly outperforms TCP-ELFN [21] alone under both caching strategies. I conclude that cross-layer information awareness is key to making TCP efficient in the presence of mobility.

The third problem I addressed is the scalability of on-demand routing protocols with respect to mobility. Scalability is an important design goal of routing protocols. Prior work mainly focused on making routing protocols scale with network size. It is essential for routing protocols to scale not only with network size but also with mobility. However, the issue of scalability with mobility has not been studied. I studied the impact of caching strategies on the scalability of on-demand routing protocols with mobility. I performed my study in the context of DSR through extensive simulations. I considered three caching strategies proposed for DSR: path caches with FIFO replacement policy, link caches with adaptive timeout mechanisms, and cache tables with my distributed cache update algorithm. Simulation results show that the cache update algorithm makes DSR scale significantly better with mobility. I conclude that making route caches adapt quickly and efficiently to topology changes is key to the scalability of on-demand routing protocols with mobility.

7.2 Future Work

Route caches in on-demand routing protocols are as important as routing tables in standard IP routing in the Internet. Due to mobility, it is much more difficult to keep route caches up-to-date than to maintain routing tables in the Internet. I have presented a novel approach that achieves fast, accurate, and efficient cache updating. Two directions can be explored in future work. First, optimizations can be incorporated into my cache update algorithm. For example, the downstream nodes that have cached a broken link sometimes may not be reachable. If there is a way to notify such nodes without initiating route discoveries, then protocol performance will be further improved. Second, my solution is applicable to other on-demand routing protocols. It is interesting to replace adaptive timeout mechanisms used in other on-demand routing protocols with distributed cache updating.

Two future directions can be explored with respect to improving TCP performance. First, the network layer is a critical layer in the protocol stack. My distributed cache update algorithm makes the network layer more mobility-aware. I also show that the network layer should be aware of the transport layer information, provide feedback about lost data, and provide reliability by retransmitting ACKs. More work to improve TCP performance can be done at the network layer, such as providing feedback about congestion. Second, a general solution to frequent packet losses is to send packet loss notifications and to retransmit ACKs for lost ACKs. It is interesting to apply my solution to other routing protocols.

Finally, it is challenging for routing protocols to be scalable with traffic load, network size, and mobility. My work contributes to the understanding of the scalability of on-demand routing protocols. My insight can be generalized and applied to other on-demand routing protocols. For example, if an ad hoc parameter limits the scalability of a routing protocol, then it is necessary to make the parameter adaptive.

Bibliography

[1] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581. `http://www.faqs.org/rfcs/rfc2581.html`, April 1999.

[2] V. Anantharaman and R. Sivakumar. A microscopic analysis of TCP performance over wireless ad-hoc networks. Presented in 2nd ACM SIGMETRICS (Poster Paper), 2002.

[3] I. Aron and S. Gupta. On the scalability of on-demand routing protocols for mobile ad hoc networks: an analytical study. *Journal of Interconnection Networks*, 2(1):5–29, 2001.

[4] O. Arpacioglu, T. Small, and Z. Hass. Notes on scalability of wireless ad hoc networks, IETF Internet Draft. `http://www.flarion.com/ans-research/Drafts/draft-irtf-ans-scalability-definition-01.txt/`, December 2003.

[5] B. Bellur and R. Ogier. A reliable, efficient topology broadcast protocol for dynamic networks. In *Proc. 18th IEEE INFOCOM*, 1999.

[6] 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.

[7] K. Chandran, S. Raghunathan, S. Venkatesan, and R. Prakash. A feedback based scheme for improving TCP performance in ad-hoc wireless networks. In *Proc. 18th IEEE ICDCS*, pp. 472–479, 1998.

[8] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): routing protocol performance issues and evaluation considerations, RFC 2501. `http://www.faqs.org/rfcs/rfc2501.html`, January 1999.

[9] D. Comer. *Internetworking with TCP/IP: principles, protocols, and architectures, Volume 1*. Prentice Hall, 2000.

[10] T. Dyer and R. Bopanna. A comparision of TCP performance over three routing protocols for mobile ad hoc networks. In *Proc. 2nd ACM MobiHoc*, pp. 56–66, 2001.

[11] J. Eriksson, M. Faloutsos, and S. Krishnamurthy. Scalable ad hoc routing: the case for dynamic addressing. In *Proc. 23rd IEEE INFOCOM*, 2004.

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

[13] Z. Fu, X. Meng, and S. Lu. How bad TCP can perform in mobile ad hoc networks. In *IEEE ISCC*, 2002.

[14] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on TCP throughput and loss. In *Proc. 22nd IEEE INFOCOM*, 2003.

[15] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. In *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[16] M. Gerla, K. Tang, and R. Bagrodia. TCP performance in wireless multi-hop networks. In *Proc. 2nd IEEE WMCSA*, 1999.

[17] M. Grossglauser and D. Tse. Mobility increases the capacity of wireless ad hoc networks. In *Proc. 20th IEEE INFOCOM*, 2001.

[18] P. Gupta and P. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, **46**(2):388–404, March 2000.

[19] Z. Haas and M. Pearlman. The performance of query control schemes for the Zone Routing Protocol. In *Proc. 9th ACM SIGCOMM*, 1998.

[20] Z. Haas, M. Pearlman, and P. Samar. The Zone Routing Protocol (ZRP) for ad hoc networks. IETF Internet Draft. `http://www.ietf.org/proceedings/02nov/I-D/draft-ietf-manet-zone-zrp-04.txt`, July 2002.

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

[22] G. Hollan and N. Vaidya. Impact of routing and link layers on TCP performance in mobile ad hoc networks. In *Proc. IEEE WCNC*, 1999.

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

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

[25] IEEE Computer Society LAN MAN Standards Committee. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, IEEE Std 802.11-1997. The IEEE, New York, New York, 1997.

[26] A. Iwata, C. Chiang, G. Pei, M. Gerla, and T. Chen. Scalable routing strategies for ad hoc wireless networks. *IEEE Journal on Selected Areas in Communications*, **17**(8):1369–1379, August 1999.

[27] V. Jacobson. Congestion avoidance and control. *Computer communication review*, **18**(4):314–329, August 1988.

[28] P. Jacquet, P. Muhlethaler, A. Qayyum, A. Lanouiti, L. Viennot, and T. Clausen. Optimized link state routing protocol (OLSR). IETF Internet Draft. `http://hipercom.inria.fr/olsr/draft-ietf-manet-olsr-10.txt`, May 2003.

[29] D. Johnson. Routing in ad hoc networks of mobile hosts. In *Proc. 1st IEEE WMCSA*, pp. 158–163, 1994.

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

[31] D. Johnson, D. Maltz, and Y.-C. Hu. The Dynamic Source Routing for mobile ad hoc networks, IETF Internet Draft. `http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt`, July 2004.

[32] J. Jubin and J. Tornuw. The DARPA packet radio network protocols. *Proceedings of IEEE*, **75**(1):21–32, 1987.

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

[34] S.-J. Lee, E. Royer, and C. Perkins. Scalability study of the ad hoc on-demand distance vector routing protocol. *International Journal of Network Management*, 13(2):97–114, 2003.

[35] J. Li, J. Jannotti, D. Couto, D. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proc. 6th ACM MobiCom*, 2000.

[36] J. Liu and S. Singh. ATCP: TCP for mobile ad hoc networks. *IEEE Journal on Selected Areas in Communication*, **19**(7):1300–1315, 2001.

[37] 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.

[38] 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 Journal on Selected Areas in Communication*, **17**(8):1439–1453, August 1999.

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

[40] J. Monks, P. Sinha, and V. Bharghavan. Limitations of TCP-ELFN for ad hoc networks. In *Proc. 5th Workshop on Mobile and Multimedia Communication*, 2000.

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

[42] C. Perkins and P. Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In *Proc. 5th ACM SIGCOMM*, **24**(4):234–244, 1994.

[43] C. Perkins, E. Royer, and S. Das. Ad hoc On-demand Distance Vector (AODV) routing. RFC 3561. `http://www.ietf.org/rfc/rfc3561.txt`, July 2003.

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

[45] C. Perkins, E. Royer, S. Das, and M. Marina. Performance comparison of two on-demand routing protocols for ad hoc networks. *IEEE Personal Communications Magazine special issue on Ad hoc Networking*, pp. 16–28, February 2001.

[46] D. Plummer. Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. RFC 826. `http://www.faqs.org/rfcs/rfc826.html`, November 1982.

[47] Monarch Project. Mobile networking architectures. `http://www.monarch.cs.rice.edu/`.

[48] S. Ramanathan and M. Steenstrup. Hierarchically-organized, multihop mobile networks for multimedia support. *ACM/Baltzer Mobile Networks and Applications*, 3(1): pp. 101–119, 1998.

[49] C. Santivanez, B. McDonald, I. Stavrakakis, and R. Ramanathan. On the scalability of ad hoc routing protocols. In *Proc. 21st IEEE INFOCOM*, 2002.

[50] C. Santivanez, R. Ramanathan, and I. Stavrakakis. Making link-state routing scale for ad hoc networks. In *Proc. 1st ACM MobiHoc*, 2001.

[51] K. Sundaresan, V. Anantharaman, H.-Y. Hsieh, and R. Sivakumar. ATP: A reliable transport protocol for ad-hoc networks. In *Proc. 4th ACM MobiHoc*, pp. 64–75, 2003.

[52] F. Wang and Y. Zhang. Improving TCP performance over mobile ad-hoc networks with out-of-order detection and response. In *Proc. 3rd ACM MobiHoc*, pp. 217–225, 2002.

[53] K. Xu, M. Gerla, L. Qi, and Y. Shu. Enhancing TCP fairness in ad hoc wireless networks using neighborhood RED. In *Proc. 9th ACM MobiCom*, pp. 16–28, 2003.

[54] J. Yoon, M. Liu, and B. Noble. Random waypoint considered harmful. In *Proc. 22nd IEEE INFOCOM*, 2003.

[55] X. Yu. Improving TCP performance over mobile ad hoc networks by exploiting cross-layer information awareness. In *Proc. 10th ACM MobiCom*, pp. 231-244, Sept. 2004.

[56] X. Yu and D. Johnson. The impact of caching strategies on the scalability of on-demand routing protocols. *Submitted for Publication.*

[57] X. Yu and Z. Kedem. Reducing the effect of mobility on TCP by making route caches quickly adapt to topology changes. In *Proc. 40th IEEE ICC*, June 2004.

[58] 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 appears as NYU Computer Science TR2003-842, July 2003.)