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

Xin Yu
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
xinyu@cs.nyu.edu

ABSTRACT

TCP performance degrades significantly in mobile ad hoc networks because most of packet losses occur as a result of route failures. Prior work proposed to provide link failure feedback to TCP so that TCP can avoid responding to route failures as if congestion had occurred. However, after a link failure is detected, several packets 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. In this paper, we propose to make routing protocols aware of lost data packets and ACKs and help reduce TCP timeouts for mobility-induced losses. Toward this end, we present two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). EPLN seeks to notify TCP senders about lost data packets. For lost ACKs, BEAD attempts to retransmit ACKs at either intermediate nodes or TCP receivers. Both mechanisms extensively use cached routes, without initiating route discoveries at any intermediate node. We evaluate TCP-ELFN enhanced with the two mechanisms using two caching strategies for DSR, path caches and a distributed cache update algorithm proposed in our prior work. We show that TCP-ELFN with EPLN and BEAD significantly outperforms TCP-ELFN under both caching strategies. We conclude that cross-layer information awareness is key to making TCP efficient in the presence of mobility.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

General Terms

Algorithms, Design, Performance

Keywords

Ad hoc networks, TCP, Mobility, Transport layer, Cross-layer, Early packet loss notification, Best-effort ACK delivery

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiCom'04, Sept. 26-Oct. 1, 2004, Philadelphia, Pennsylvania, USA.
Copyright 2004 ACM 1-58113-868-7/04/0009 ...$5.00.

1. INTRODUCTION

TCP performance degrades significantly in mobile ad hoc networks [9, 6, 18]. In such networks, nodes move arbitrarily, cooperating to forward packets to enable communication between nodes not within wireless transmission range. Route failures due to mobility are the primary reason for most of packet losses [6, 18]. 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 [3, 9, 4, 15] 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) [9] 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.

TCP benefits from link failure feedbacks but is still affected by frequent route failures. Holland and Vaidya [9] 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 responds with stale routes. Holland and Vaidya showed that turning off replying from caches improves TCP performance for a network with a single TCP connection. But 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) [13, 14], we proposed a distributed cache update algorithm [21]. When a node detects a link failure, our algorithm proactively notifies all reachable nodes that have cached that link about the link failure. Therefore, it enables the routing protocol at TCP senders and receivers to quickly remove stale routes from their caches. Proactive cache updating also prevents stale routes from being propagated to other nodes. We showed [22] that this algorithm significantly improves TCP throughput, because it reduces route failures by making the network layer more mobility-aware.

In this paper, we investigate how to make TCP perform well in the presence of frequent packet losses due to mobility. In contrast to prior work, we focus on issues at both the network layer and the transport layer, as well as the interactions between these two layers. We 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, we first study how mobility affects TCP through simulation of ELFN. We make several observations. First, we 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, we 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 intermediate nodes using cached routes. If packets are salvaged, freezing TCP may decrease throughput because TCP can continue to send packets using other routes; however, if we do not freeze TCP when packets are salvaged, TCP will time out if salvaged packets are dropped. We observe that these two choices result in similar throughput, but freezing TCP upon route failures reduces TCP timeouts. Finally, we 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 TCP throughput but wastes limited bandwidth; retransmitting the packets that have been received wastes nodes' energy.

We propose to make routing protocols aware of lost data packets and ACKs and help reduce TCP timeouts for mobility-induced losses. Toward this end, we 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 TCP 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 ACK either 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 TCP sender using a cached route. If the intermediate node that salvaged the ACKs cannot retransmit an ACK, it sends a notification to the TCP receiver. If none of the intermediate nodes is able to retransmit an ACK, the TCP receiver's routing protocol retransmits an ACK with the highest sequence number if it has a route to reach the TCP sender.

Since EPLN and BEAD extensively use cached routes, we evaluate the effectiveness of the mechanisms using different caching strategies. We incorporate EPLN and BEAD into DSR with path caches and into DSR with our distributed cache update algorithm. Through detailed simulations, we compare the performance of TCP-ELFN; TCP-ELFN with EPLN and BEAD; and TCP-ELFN with EPLN, BEAD, and our cache update algorithm. We 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 our 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 our cache update algorithm for 100-node networks. In addition, enhanced with our 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 paper is organized as follows. In Section 2, we present as background an overview of DSR, a summary of our distributed cache update algorithm, and a description of simulation environment. We study how mobility affects TCP in Section 3. In Section 4 we describe EPLN and BEAD, and in Section 5 we present an evaluation of EPLN and BEAD. We discuss related work in Section 6 and present our conclusions in Section 7.

2. BACKGROUND

2.1 Overview of DSR

We present a simplified description of DSR [13, 14]. DSR is composed of two entirely on-demand mechanisms: *Route Discovery* and *Route Maintenance*. When a source node wants to send packets to a destination, it first checks whether it has a route in its cache. If it does not have a route, it initiates a Route Discovery by broadcasting a ROUTE REQUEST. When receiving a ROUTE REQUEST, a node checks whether it has a route to the destination in its cache. If it has, it sends a ROUTE REPLY to the source, including a source route formed as the concatenation of the source route in the ROUTE REQUEST and the cached route. Otherwise, the node adds its address to the source route in the packet header and rebroadcasts the ROUTE REQUEST. When the ROUTE REQUEST reaches the destination, the destination sends a ROUTE REPLY containing the source route to the source. Each node forwarding the ROUTE REPLY caches the route starting from itself to the destination. When receiving the ROUTE REPLY, the source caches the source route.

In Route Maintenance, a node forwarding a packet is responsible for confirming that the packet has reached the next hop in the route. If no acknowledgement is received after the maximum number of retransmission attempts, this node assumes that the next hop is unreachable and sends a ROUTE ERROR to the source node, indicating the broken link. Each node receiving a ROUTE ERROR removes from its cache the routes containing the broken link.

Besides Route Maintenance, DSR uses two mechanisms to remove stale routes from caches. First, a source node piggybacks on the next ROUTE REQUEST the last known broken link information (called a GRATUITOUS ROUTE ERROR). Second, DSR uses heuristics: a small cache size with FIFO replacement policy for path caches, and adaptive timeout mechanisms for link caches [11].

2.2 The Distributed Cache Update Algorithm

Cached routes easily become stale due to mobility. In our prior work [21], we proposed to proactively disseminate the information about a broken link to the nodes that have that link in their caches. We defined a new cache structure, called a cache table, and pre-

sented a distributed cache update algorithm to make route caches in DSR adapt quickly to topology changes.

In a cache table, a node not only stores routes but also maintains the information necessary for cache updates. A node gathers two types of information for each route: (1) how well the routing information is synchronized among the nodes on the route, which means whether a link has been cached in only the upstream nodes, or both the upstream and the downstream nodes, or neither; and (2) which neighbor has learned which link of the route through a ROUTE REPLY. Thus, for each cached link, a node knows a set of neighborhood nodes that have that link in their caches.

The algorithm uses the local information kept by each node to achieve distributed cache updating. The algorithm is started either when a node detects a link failure or when a node receives a cache update notification. In either case, it notifies only the neighborhood nodes that have cached the broken link to update their caches. Therefore, the information about a broken link will be propagated to all reachable nodes that have that link in their caches.

The algorithm provides three benefits. First, it reduces packet losses because of the improved cache correctness. Second, it reduces packet delivery latency, since detecting broken links is the dominant factor of latency. Finally, it reduces ROUTE ERRORS caused by the use of stale routes. We showed that the algorithm outperforms DSR with path caches and with *Link-MaxLife* [11], an adaptive timeout mechanism for link caches.

2.3 Simulation Environment

In this section, we introduce simulation environment, since we will present a simulation study of how mobility affects TCP in the next section. We will give a detailed description of our evaluation methodology for the proposed mechanisms in Section 5.

We used *ns-2* [5] network simulator with Monarch Project's wireless and mobile extensions [2, 17]. The network interface is modelled after the Lucent's WaveLAN, which provides a 2Mbps transmission rate and a nominal transmission range of 250m; the network interface uses IEEE 802.11 DCF MAC protocol [12]. The mobility model is *random waypoint model* [2] in a rectangular field. In this model, a node starts at a random position, picks a random destination, moves to it at a randomly chosen speed, and pauses for a specified pause time. The node speed was randomly chosen from $v \pm 1$ m/s, where v is node mean speed. We used pause time 0 s for all simulations. The two field configurations we used were 1500m $\times$ 1000m field with 50 nodes and 2200m $\times$ 600m field with 100 nodes. We used TCP-Reno with the packet size of 1460 bytes. The maximum size of both congestion window and receiver's advertised window is 8. FTP is the application that we used over TCP.

3. MOBILITY, TCP, AND ELFN

In this section, we study how mobility affects TCP through simulation of ELFN in a network with 50 nodes and one TCP connection. The node speed was randomly chosen from 10 ± 1 m/s. We explore three issues: (1) how to set RTO and 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.

3.1 How to Set RTO and cwnd after Congestion Control Mechanisms are Restored?

Node 0 starts a TCP connection to node 1 at 100 s. At 101.607677 s, node 31 detects that link (31, 1) is broken when transmitting the packet with sequence number 39 using route 0–38–9–31–1, as shown in Fig. 1. Node 31 salvages this packet and the packets from 40 to 46 in the network interface queue using route 31–39–1. It

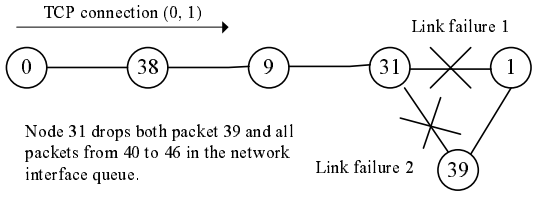

Figure 1: An Example of How Mobility Affects TCP

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, link (31, 39) is also broken, and therefore node 31 drops all packets. At 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 103.621079 s, node 0 sends the probing packet 39. At 103.695992 s, node 0 receives the ACK for packet 39 and restores TCP's retransmission timer.

We 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 [9]. 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 ARP's proclivity to silently drop packets. If a restored route breaks and results in a failed ARP lookup, the sender will likely time out. They concluded that, given the length of timeout, using default RTO does not dramatically affect performance.

We attribute a different reason to the reduced throughput when default values are used. In our 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 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. We will present an evaluation of the two choices in Section 5.

3.2 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 Fig. 1, after link (31, 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 109.695992 s, TCP re-

transmits packet 40 using route 0–38–9–31–39–1. However, this is a stale route, since link (0, 38) is detected as broken at 109.728763 s. TCP is frozen because no route is available in node 0's cache. At 109.738086 s, node 0 sends the packet using the discovered route 0–22–16–1. But this is also a stale route, since link (22, 16) has broken. Node 22 salvages this packet using another route and sends a ROUTE ERROR to node 0. No changes are made to TCP's state because TCP has been frozen.

At 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 the packets 41 and 42 using route 0–9–16–1. (When TCP is in the slow-start phase, it sends two packets for one ACK.) However, link (9, 16) is broken. Node 9 salvages the two 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 we discussed in Section 1, if packets are salvaged, freezing TCP upon route failures may decrease TCP throughput; if we do not freeze TCP when packets are salvaged, TCP will time out if the salvaged packets are dropped. We observed that freezing TCP upon route failures reduces TCP timeouts; therefore, we believe that this is a conservative but reliable approach because route failures are frequent. Thus, we will use this option in our simulations.

3.3 Unaware of Lost Data Packets and ACKs

As we showed, 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 TCP senders about packet losses, TCP senders will retransmit lost packets earlier and thus avoid waiting for timeouts.

We show another example in which TCP times out because RTO and cwnd are set to default values and because data packets are dropped silently. At 120.140313 s, node 16 attempts to transmit packet 409 using route 0–25–16–1 but detects that link (16, 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 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 120.239832 s, node 16 detects that link (16, 34) is broken and drops the packets 409 and 410. As a result, TCP times out at 126.164540 s.

If we keep TCP's state 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. Thus, 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.

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.

4. 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, we present two mechanisms: early packet loss notification (EPLN) and best-effort ACK delivery (BEAD). In this section, we describe the two mechanisms in detail with examples.

4.1 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.

We consider three types of packets that will encounter route failures: data packets, ACKs, and packet loss notifications. We summarize the operations of the two mechanisms as follows:

1. If data packets or ACKs are dropped and this is the first time they encounter 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 TCP 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 TCP 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 TCP 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 TCP receiver retransmits an ACK if it has a cached route.

5. If a notification packet 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 TCP sender or the TCP 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. Our prior work [21] has addressed the cache staleness issue; we will use our distributed cache update algorithm as one caching strategy in our evaluation of EPLN and BEAD. For DSR with path caches, our mechanisms provide another benefit: quick detection and eviction of stale routes.

4.2 Packet Loss Notifications

We 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), TCP sequence number, and the source route used in routing the packet. A node uses the information recorded in its *drop_list* to construct packet loss notifications.

We 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 (data or ACK), and the TCP sequence numbers of lost packets. When possible, we piggyback the information about lost packets on a ROUTE ERROR sent by DSR; otherwise, a notification will be sent as a ROUTE ERROR. We extend the format of a ROUTE ERROR to include an optional field called *conn_list*, which contains one or more *conn_info* structures.

4.3 EPLN and BEAD

In this section, we 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.

4.3.1 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 the *drop_list*, recording the information about the packet. The node 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. We focus on the operation at the network layer first and will describe the responses of TCP to an ICMP message in Section 4.3.5.

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 we 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.

4.3.2 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 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*.

We use 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. We modify 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 our mechanisms generate more ROUTE ERRORS than DSR and route discoveries introduce significant overhead.

4.3.3 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. 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 reach the TCP sender. If it has a 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 route to reach the TCP sender. If it has a 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 cannot reach either of them, it will not process the information about lost ACKs. (2) If lost packets are data packets, the node checks whether it has a route to reach the TCP sender. If it has a route, it sends a notification to the TCP sender.

If the node is an intermediate node, it will send a notification to either a TCP sender or a TCP receiver only when the notification received is not a nested ROUTE ERROR, indicated by a field called *num_route_error* shown in the pseudo code. If the notification is a nested ROUTE ERROR, we piggyback the information about lost packets on the ROUTE ERROR sent by DSR. We will show an example for this case in the next section.

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 TCP sender;

```
if has_conn_info then
  for all entry e ∈ 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 all seq_no ∈ e do
          new_p ⇐ p.copy()
          new_p.tcph.seqno ⇐ seq_no
          sendICMPtoTCP(new_p)
        end for
      end if
      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)
        end if
      end if
    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
        else if findRoute(e.src, new_p.route) then
          new_p.dest ⇐ e.src
        end if
      else if e.ptype = TCP then
        if findRoute(e.src, new_p.route) then
          new_p.dest ⇐ e.src
        end if
      end if
      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)
        else if num_route_error = 1 then
          new_p.srh.conn_list[0] ⇐ e
          new_p.srh.has_conn_info ⇐ TRUE
          sendOutPacketWithRoute(new_p)
        end if
      end if
    end if
  end for
end if
```

Algorithm 1: At the Node Receiving a Packet Loss Notification

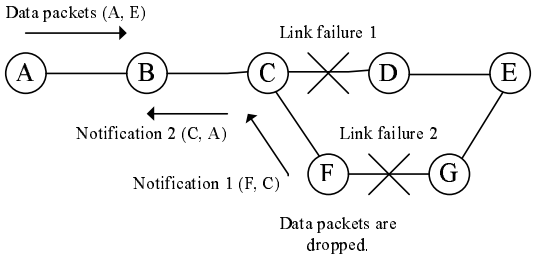

Figure 2: An Example of Early Packet Loss Notification

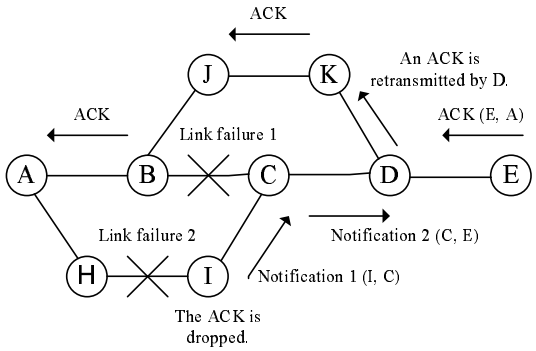

Figure 3: An Example of Best-Effort ACK Delivery

4.3.4 Examples

We show an example of EPLN in Fig. 2. Node A starts a TCP connection to node E using route A–B–C–D–E. When node C detects that link (C, 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 link (F, G) is broken and finds that it cannot salvage the packet. Before dropping the packet, F records the information about the packet in its *drop_list*. It then checks the network interface queue for the data packets or ACKs that have the same next hop in their routes. For simplicity, we assume that this packet is the only packet to be dropped. Otherwise, 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 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, link (B, A) is detected as broken, node B sends a ROUTE ERROR to C, which is a nested ROUTE ERROR. Node C attempts to send another notification to node A using cached routes.

Next, we show an example of BEAD. As shown in Fig. 3, node E sends an ACK to node A using route E–D–C–B–A. Node C detects that the link (C, B) is broken and salvages the ACK using route C–I–H–A. Then node I detects that link (I, 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 C receives the notification, it first checks whether it has a route to reach the TCP sender, node A. We assume that it does not have such a route. So node C sends a notification to the TCP receiver,

node E. When node D forwards this notification, it checks whether it has a route to reach the TCP sender and finds that it has a route D–K–J–B–A. Node D retransmits an ACK to A. If node D does not have a route to reach A, node E will attempt to retransmit an ACK to 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.

4.3.5 At a TCP sender: cross-layer interactions

We have presented the operation designed at the network layer. In this section, we 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. We modify 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 we choose to freeze TCP upon route failures, the network layer will send 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. Therefore, an ICMP message also contains the information indicating whether a packet is lost.

As shown in the pseudo code, when TCP receives an ICMP packet, 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* 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. In this case, 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 the remaining lost packets recorded in the *lost_pkt*. Thus, TCP adapts fast to packet losses.

TCP-Reno *recv*():
Variables:
tcph: TCP header; *icmp*: ICMP header;
p: the current packet;
thaw_timer: the thaw timer in ELFN;
t_thaw: the probing interval;
thaw_seqno: the sequence number of the probing packet;
lost_pkt: an array recording the 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
  end if
  if not frozen() then
    freeze()
  end if
  if thaw_timer.status() = TIMER_IDLE then
    thaw_timer.resched(t_thaw)
    thaw_seqno ⇐ p.tcph.seqno
    if p.icmp.pkt_lost = TRUE then
      output(thaw_seqno)
    end if
  else if p.tcph.seqno < thaw_seqno then
    thaw_seqno ⇐ p.tcph.seqno
    if p.icmp.pkt_lost = TRUE then
      output(thaw_seqno)
    end if
  else if p.tcph.seqno = thaw_seqno then
    if p.icmp.pkt_lost = TRUE then
      output(thaw_seqno)
    end if
  else if p.tcph.seqno > thaw_seqno then
    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
    end if
    free(p)
  end if
else if frozen() then
  melt()
end if
if tcp_melt = TRUE then
  if num_lost_pkt ≠ 0 then
    for all seqno ∈ lost_pkt do
      if seqno > last_ack_ then
        output(seqno)
      end if
    end for
    num_lost_pkt ⇐ 0
    tcp_melt ⇐ FALSE
  end if
end if
{/*followed by the operation executed by TCP when it receives an ACK.*/}
```

Algorithm 2: At a TCP Sender

5. PERFORMANCE EVALUATION

5.1 Evaluation Methodology

We performed two sets of experiments. In the first set of experiments, we 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 computed before TCP is frozen. In the second set of experiments, we evaluated the effectiveness of EPLN and BEAD under two caching strategies for DSR: path caches and our distributed cache update algorithm, which we call DSR-Update. We used the basic operation of ELFN: freezing TCP upon route failures, sending a probing packet every time a thaw_timer expires, and restoring TCP's state when an ACK arrives. As we described, we modified the operation of ELFN at a TCP sender to make TCP adapt to packet losses.

We 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; (2)TCP-ELFN with RTO and cwnd set to the values computed before TCP is frozen, and DSR 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, we evaluated TCP performance for DSR under both promiscuous and non-promiscuous mode. Promiscuous mode, also called tapping, is an optimization for DSR [14], which disables the network interface's address filtering function and thus causes the routing protocol to receive all packets overheard by the interface. This optimization allows the routing protocol to get all useful routing information in the packet header.

We studied the effects of traffic load on TCP enhanced with different mechanisms by investigating scenarios with 1, 5, and 10 TCP connections. We did not use higher traffic load in order to factor out the effect of congestion. We 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. Each data point represents an average of 10 runs of different randomly generated scenarios. The probing interval of ELFN was 2 s.

We used three metrics:

- *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.
- *Average Number of Slow-starts*: the average number of TCP slow-starts among all TCP connections.
- *Packet Overhead*: the total number of routing packets transmitted (both sent and forwarded), including ROUTE ERRORS used by EPLN and BEAD. For DSR-Update, this metric includes ROUTE ERRORS used for cache updates.

5.2 Two Choices for Setting RTO and cwnd

Fig. 4 shows TCP throughput and 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 promiscuous mode and up to 21% for DSR without promiscuous mode, compared with using default values. As we discussed in Section 3, 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 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 TCP throughput decreases. As we analyzed, when using default values, TCP will spend 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. As a result, 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 and without promiscuous mode. Moreover, there is a large reduction in the average number of slow-starts. For 10 connection scenarios, TCP throughput decreases slightly when using "old" RTO and cwnd values. We 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. But using "old" values significantly reduces TCP timeouts, whether for higher traffic load or larger network scenarios.

5.3 The Evaluation of EPLN and BEAD

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

5.3.1 TCP Throughput

Fig. 5 shows TCP throughput. For the 50-node scenarios with 1 TCP connection, when used with DSR 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 much 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 significant improvement demonstrates the effectiveness of our 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 pack-

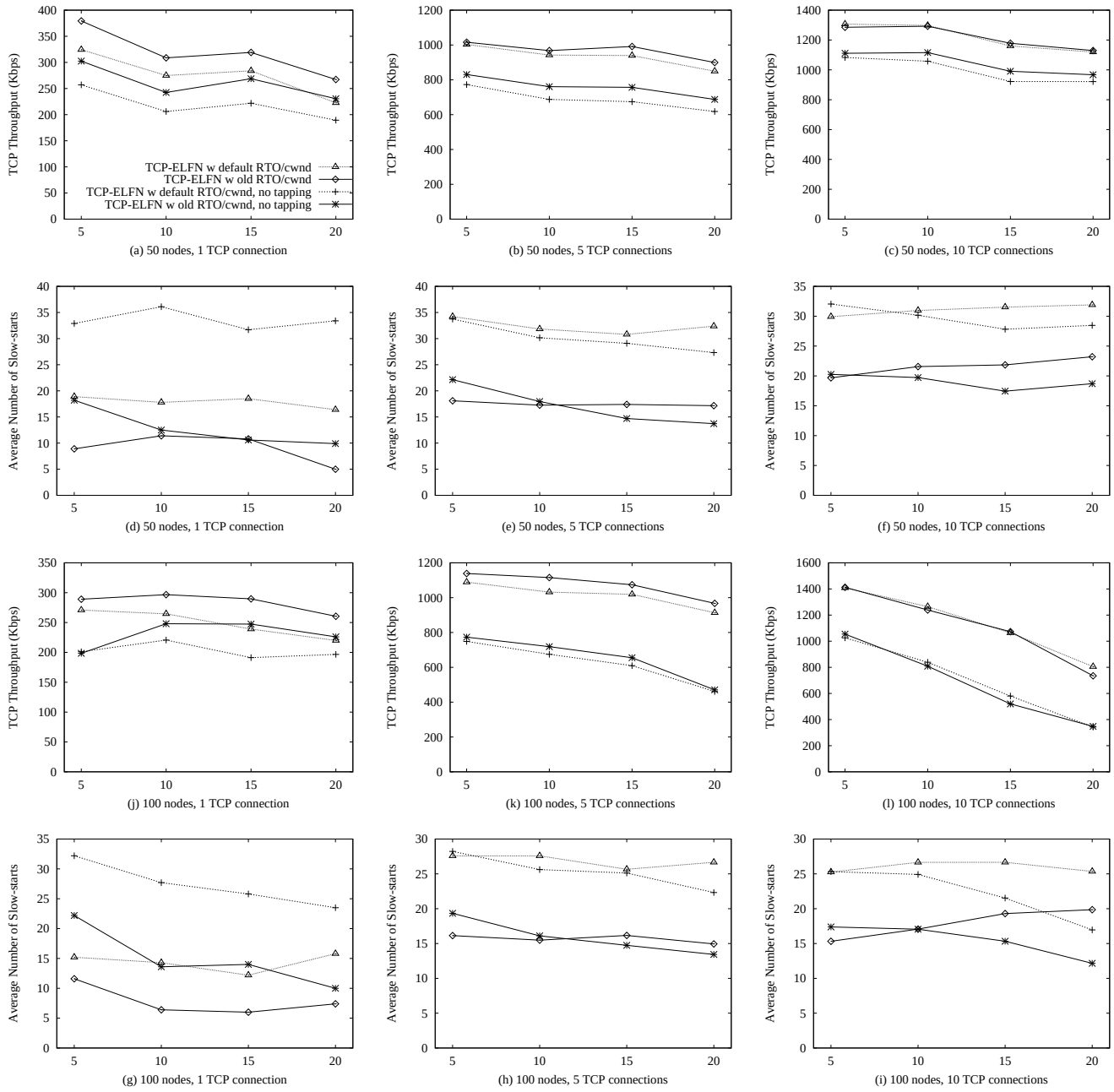

Figure 4: TCP Throughput and Average Number of Slows-starts vs. Mobility under Two Choices for Setting RTO and cwnd

ets. 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, our 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. We attribute the following reason to this observation. DSR 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 cache. A source node adds an overheard route to a cache table only when it is going to use that route. As traffic load increases, nodes will overhear more routes. As a result, EPLN and BEAD with path caches store more overheard routes and thus benefit more from promiscuous mode than EPLN and BEAD with DSR-Update.

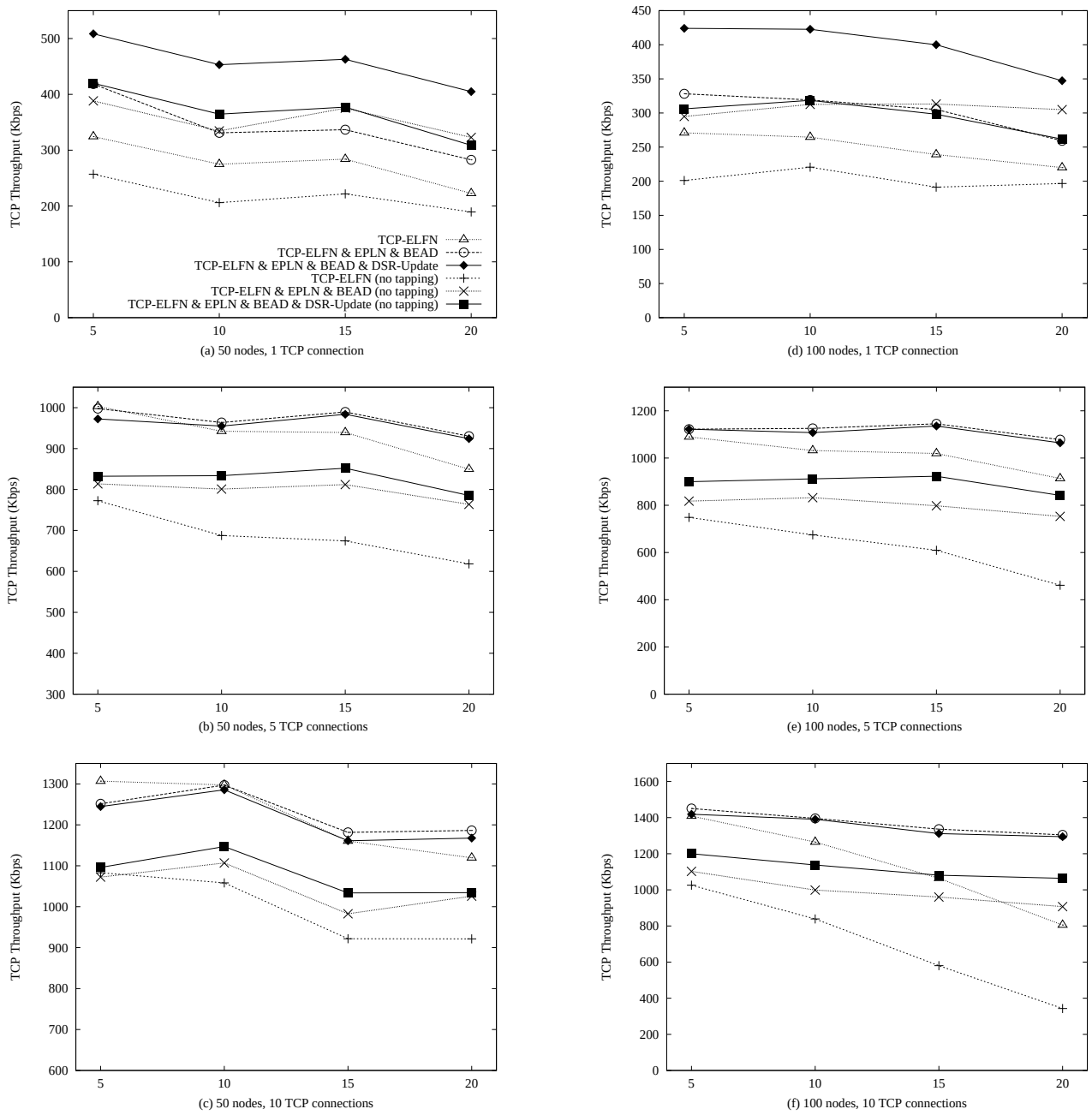

Figure 5: TCP Throughput vs. Mobility (mean speed (m/s))

5.3.2 Average Number of Slow-starts

Fig. 6 shows average number of slow-starts. EPLN and BEAD considerably reduce TCP timeouts under both caching strategies and under both promiscuous and non-promiscuous mode. For example, for the 50-node scenarios with one TCP connection, under promiscuous mode, EPLN and BEAD reduce TCP timeouts by 60% with path caches and 90% with DSR-Update compared with TCP-ELFN. Moreover, EPLN and BEAD with DSR-Update have less timeouts than EPLN and BEAD with path caches, giving the reduction by 90% and 74% under promiscuous and non-promiscuous mode respectively.

Note that TCP has only twice slow-starts when EPLN and BEAD use DSR-Update as a caching strategy, which means that TCP invokes congestion control mechanisms for packet losses caused by route failures only twice during a 900 s simulation. These results not only show that EPLN and BEAD significantly reduce TCP timeouts for mobility-induced losses, but also show that our cache update algorithm is very efficient in dealing with route failures.

As traffic load increases, EPLN and BEAD achieve the reduction in timeouts by more than 35% with path caches and 40% with DSR-Update. As network size increases, EPLN and BEAD obtain the reduction in 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.

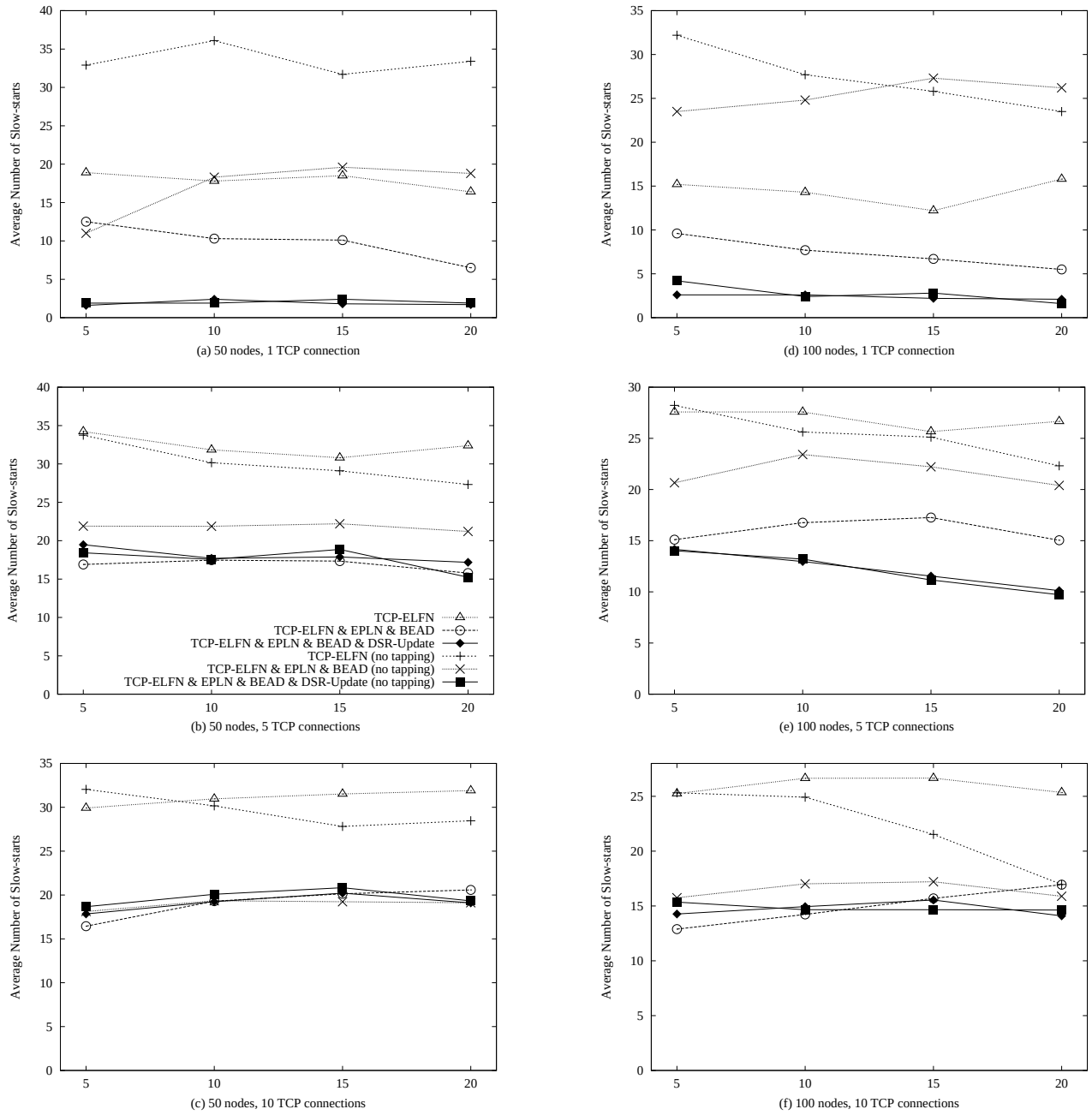

Figure 6: Average Number of Slow-starts vs. Mobility (mean speed (m/s))

5.3.3 Packet Overhead

Fig. 7 shows packet overhead. For the 50-node scenarios with one TCP connection, TCP-ELFN with EPLN and BEAD has higher overhead than TCP-ELFN under both caching strategies due to packet loss notifications. But 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 10 TCP connections at node speed of 20 m/s. For these scenarios, DSR initiates more route discoveries than DSR with EPLN and BEAD. The reason is this: as mobility increases, route failures become more frequent and thus more route discoveries take place; as traffic load increases, more routes need to be stored and FIFO 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 much 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.

We further confirm this analysis through the results for the 100-node scenarios, as shown in Figs. 7 (d)–(f). For one TCP connection and without promiscuous mode, TCP-ELFN has lower overhead than EPLN and BEAD with DSR-Update under low mobility, but has higher overhead than the latter under high mobility. This is also due to the small cache size used in DSR, 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-

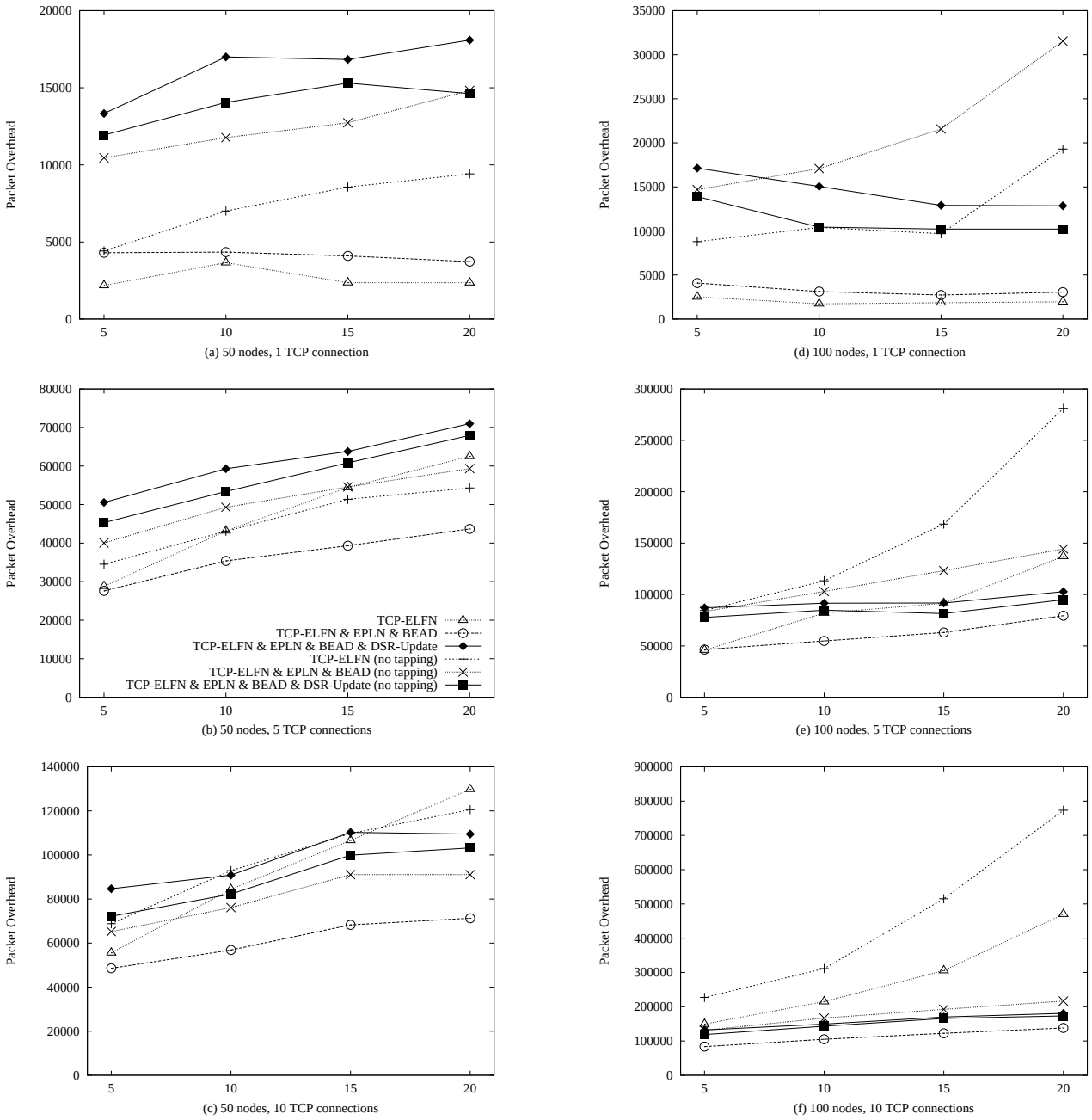

Figure 7: Packet Overhead vs. Mobility (mean speed (m/s))

ELFN decreases significantly, because DSR uses a secondary cache to store more routes, which helps reduce route discoveries. For the 100-node scenarios, TCP-ELFN also has a fast increase in overhead under high mobility. For 10 TCP connections, TCP-ELFN has significantly higher overhead than TCP-ELFN with EPLN and BEAD under both caching strategies.

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 Fig. 7 (f), under non-promiscuous mode, EPLN and BEAD with DSR-Update have the lowest overhead among the three. Under promiscuous mode, EPLN and BEAD with DSR-Update have a slightly 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.

6. RELATED WORK

TCP performance over ad hoc networks has been an active research area. Gerla *et al.* [8] investigated the impact of MAC protocol on TCP performance. Holland and Vaidya [10] studied the effect of routing and link layers on TCP performance in mobile ad hoc networks. Fu *et al.* [7] studied the effect of wireless channel on TCP throughput and loss and proposed two link layer techniques

to improve TCP throughput. Xu *et al.* [20] studied the TCP fairness issue in ad hoc networks and proposed a neighborhood RED (Randomly Early Detection) scheme to improve TCP unfairness.

Most of prior work focused on transport layer mechanisms. The major approach was to provide link failure feedback to TCP so as to prevent TCP from falsely invoking congestion control mechanisms. Chandran *et al.* [3] 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 [9] proposed to use *explicit link failure notification* (ELFN) to counter the effects of link failures due to mobility. ELFN freezes TCP upon route failures and periodically sends a probing packet until a TCP sender receives an ACK. ELFN was shown to outperform TCP in mobile ad hoc networks.

Monks *et al.* [16] 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 [4] evaluated TCP performance over three routing protocols and proposed a heuristic called *fixed RTO* to distinguish route failures from congestion. Liu and Singh [15] 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 into an appropriate state. Wang and Zhang [19] proposed to improve TCP performance by detecting and responding to out-of-order packet delivery events, which are the results of frequent route changes.

Fu *et al.* [6] observed that mobility has the most significant impact on TCP performance. TCP achieves only about 10% of a reference TCP's throughput; TCP throughput degrades by 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 [1] identified several problems at the MAC and the network layers and proposed a framework called ATRA. ATRA includes three mechanisms: reducing route failures by ensuring that ACK path is always the same as data path, predicting route failures before they occur based on the progression of signal strength of packet receptions from the concerned neighbor, and reducing the latency for route error propagation. Sundaresan *et al.* [18] studied the behavior of TCP over mobile ad hoc networks and argued that a majority of the components of TCP are inappropriate for such networks. They developed a new transport protocol called ATP to achieve effective congestion control and reliability.

Prior work that modified TCP to react to route failures ignored an important fact: route failures are not equivalent to packet losses. Route failures do not imply that packets are lost; many packets are dropped without being noticed. In contrast, we aim to make routing protocols efficiently deal with packet losses.

7. CONCLUSIONS

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

We made several observations through simulation of TCP-ELFN. First, we 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, we 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, we 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 detections. 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.

We incorporated EPLN and BEAD into DSR with path caches and into DSR with our distributed cache update algorithm [21]. We showed that, compared with TCP-ELFN, EPLN and BEAD not only significantly improve TCP throughput but also considerably reduce TCP timeouts. Moreover, EPLN and BEAD with our cache update algorithm outperform EPLN and BEAD with path caches, because proactive cache updating is more efficient than FIFO in removing stale routes.

Our 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.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We also thank David Johnson from Rice University for his helpful comments on the final version of this paper.

8. REFERENCES

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

[2] 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 *Proceedings of 4th ACM MobiCom*, pp. 85–97, 1998.

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

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

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

[6] Z. Fu, X. Meng, and S. Lu. How bad TCP can perform in mobile ad hoc networks. In *Proceedings of 7th IEEE ISCC*, 2002.

[7] 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 *Proceedings of 22nd IEEE INFOCOM*, 2003.

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

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

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

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

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

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

[14] D. Johnson, D. Maltz, 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

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

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

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

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

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

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

[21] X. Yu and Z. Kedem. A distributed adaptive cache update algorithm for the dynamic source routing protocol. NYU Computer Science Department Technical Report TR2003-842, July 2003.

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