Scenario-based Comparison of Source-Tracing and Dynamic Source Routing Protocols for Ad Hoc Networks *

Jyoti Raju
Computer Science Department
University of California
Santa Cruz, CA 95064
jyoti@cse.ucsc.edu

J.J. Garcia-Luna-Aceves
Computer Engineering Department
University of California
Santa Cruz, CA 95064
jj@cse.ucsc.edu

ABSTRACT

We present source tracing as a new viable approach to routing in ad hoc networks in which routers communicate the second-to-last hop and distance in preferred paths to destinations. We introduce a table-driven protocol (BEST) in which routers maintain routing information for all destinations, and an on-demand routing protocol (DST) in which routers maintain routing information for only those destinations to whom they need to forward data. Simulation experiments are used to compare these protocols with DSR, which has been shown to incur less control overhead that other on-demand routing protocols. The simulations show that DST requires far less control packets to achieve comparable or better average delays and percentage of packet delivered than DSR, and that BEST achieves comparable results to DSR while maintaining routing information for all destinations.

Keywords

On-demand routing, wireless routing, ad hoc networks.

1. INTRODUCTION

Ad hoc networks (or multi-hop packet-radio networks) consist of mobile routers interconnecting hosts. These networks are useful in tactical and commercial scenarios in which there is no base-station infrastructure present. The deployment of such routers is ad hoc and the topology of the network is very dynamic, because of host and router mobility, signal loss and interference, and power outages. Furthermore, the bandwidth available for the exchange of routing information in ad hoc networks is far lesser than the bandwidth available in a wired internet.

Routing for ad hoc networks can be classified into two main types: *table-driven* and *on-demand.* Table driven routing attempts to maintain consistent information about the path from each node to every other node in the network. The Destination-Sequenced Distance-Vector Routing (DSDV) protocol is a table driven algorithm that modifies the distributed Bellman-Ford routing algorithm to include timestamps that prevent loop-formation [15]. The Wireless Routing Protocol (WRP) is a distance vector routing protocol which belongs to the class of path-finding algorithms that exchange second-to-last hop (*predecessor*) to destinations in addition to distances to destinations [13]. This extra information helps remove the "counting-to-infinity" problem that most distance vector routing algorithms suffer from [1]. It also speeds up route convergence when a link failure occurs.

On-demand routing protocols have been designed to limit the amount of bandwidth consumed in maintaining up-to-date routes to all destinations in a network by maintaining routes to only those destinations to which the routers need to forward data traffic. The basic approach consists of allowing a router that does not know how to reach a destination to send a flood-search message to obtain the path information it needs. There are several recent examples of this approach (e.g., AODV [16], ABR [19], DSR [12], TORA [14], SSA [5], ZRP [11]) and the routing protocols differ on the specific mechanisms used to disseminate flood-search packets and their responses, cache the information heard from other nodes' searches, determine the cost of a link, and determine the existence of a neighbor. However, all the on-demand routing proposals to date use flood search messages that either: (a) give sources the entire paths to destinations, which are then used in source-routed data packets (e.g., DSR); or (b) provide only the distances and next hops to destinations, validating them with sequence numbers (e.g., AODV) or time stamps (e.g., TORA).

The Dynamic Source Tree (DSR) protocol [12] has been shown to outperform other on-demand routing protocols such as TORA and AODV [3, 17] from the standpoint of reducing the number of update packets needed to update routing tables, and therefore constitutes a good baseline for comparison. In DSR, the replies to flood search messages contain the entire route from source to destination, which are stored in route caches at the senders. One problem with source routing is that it results in long data-packet headers as the network size increases; in addition, source routing will not work with security schemes that encrypt headers.

In this paper, we introduce and analyze two efficient routing protocols for ad hoc networks using predecessor and distance information. The first protocol is DST (dynamic source tree) protocol, which constitutes a new approach for on-demand distance vector routing for ad hoc networks. Like other on-demand routing protocols, DST acquires routes to destinations only when traffic for those destinations exists and there is no known route to the destination. This implies

*This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under grant F30602-97-2-0338.

that route information is only maintained for destinations with which a router needs to communicate. It also implies thatthe routes used are not necessarily optimum; they only have to be valid and of a finite metric value. DST does not use source-routed packets or time stamps to validate distance updates. DST uses a source-tracing algorithm similar to the one advocated in prior table-driven routing protocols in which routers maintain routing information for all network destinations [13, 1]. To reduce the number of loops, the source-tracing algorithm allows for complete paths to be checked for loops before being added to the routing table. DST uses information about the length and predecessor of the shortest path to all known destinations to eliminate the counting to infinity problem of the distributed Bellman-Ford algorithm. A node running DST maintains shortest paths in its routing tables for all the destinations it knows. A node also maintains the routing tables reported incrementally by all of its known neighbors. A node uses the routing tables of known neighbors together with the link costs to known neighbors to generate its own routing table. A routing message broadcast by node i contains a vector of entries in which each entry corresponds to a route in the routing table; each entry contains a destination identifier j, the distance to the destination D_j^i and the predecessor to that destination p_j^i.

The second protocol, Bandwidth Efficient Source Tracing (BEST) protocol is also based on source-tracing and is an extension of WRP [13]. It uses unreliable updates and introduces a conservative approach to table-driven routing, i.e., routers send updates only under conditions where routing table loops are suspected.

There are three key contributions of this paper: (i) introducing source tracing for on-demand routing and table-driven routing as a viable approach for ad hoc networks; (ii) presenting the design of efficient protocols that do not use sequence numbers, internodal synchronization or complete paths to ensure that no permanent loops are formed; and (iii) examining the performance of DST, DSR and BEST in simulation scenarios that mimic real world scenarios and using these simulations to conclude that source-tracing can be the basis for a very efficient routing protocol that maintains routing information either on-demand or for all destinations.

Section 2 presents the network model used throughout the paper. Section 3 gives a detailed description of DST and presents an example of how it eliminates long-term looping. Section 4 gives a brief description of BEST. Section 5 uses simulations to compare the performance of DSR, DST and BEST using the same movement model used in [3, 4] to compare DSR with other on-demand and table-driven routing protocols.

2. NETWORK MODEL

To describe DST, a network is modelled as an undirected graph with V nodes and E links. Instead of having interface identifiers, a router has a single node identifier, which helps the routing and other application protocols identify it. It is assumed that a node has radio connectivity with multiple nodes using a single physical radio link. Accordingly, we map a physical broadcast link connecting a node and its multiple neighbors into point-to-point links between the node and its neighbors. Each link has a positive cost associated with it. If a link fails, its cost is set to infinity. A node failure is modelled as all links incident on the node being set to infinity.

For the purpose of routing-table updating, a node A considers another node B as its neighbor if A receives an update from neighbor B. Node B is no longer node A's neighbor when the medium access protocol at node A sends a signal to DST indicating that data packets can no longer be sent successfully to node B.

DST is designed to run on top of any wireless medium-access protocol. Routing messages are broadcast unreliably and the protocol assumes that routing packets may be lost due to changes in link connectivity, fading or jamming. Since DST only requires a MAC indication that data packets can no longer be sent to a neighbor, the need for a link-layer protocol for monitoring link connectivity with neighbors or transmitting reliable updates is eliminated, thus reducing control overhead. If such a layer can be provided with no extra MAC overhead, then DST can be made more proactive by identifying lost neighbors before data for them arrives, resulting in faster convergence and decreased data packet loss.

3. THE DST PROTOCOL

3.1 Routing Information Maintained in DST

A router in DST maintains a *routing table*, a *distance table*, a *data buffer* and a *query table*.

The set of known destinations is denoted bys N and N_i denotes the list of known neighbors.

The routing table at router i contains entries for all known destinations. Each entry consists of the destination identifier j, the successor to that destination s_j^i, the second-to-last-hop to the destination p_j^i, the distance to the destination D_j^i and a route tag tag_j^i. When the element tag_j^i is set to $correct$, it implies a loop-free finite value route. When it is set to $null$, it implies that the route still has to be checked and when it is set to $error$, an infinite metric route or a route with a potential loop is implied.

The distance table at router i is a matrix containing, for each $k \in N_i$ and each destination j, the distance value of the route from i to j through k, D_{jk}^i and the second-to-last hop p_{jk}^i on that route. D_{jk}^i is always set equal to $RD_j^k + l_k^i$, where RD_j^k is the distance reported by k to j in the last routing message and l_k^i is the link cost of link (i, k). The link cost may be set to one to support minimum-hop routing, or it may be set to some other link parameter like latency or bandwidth.

The data buffer is a queue that holds all the data packets waiting for routes to destinations. There are various approaches for buffer management. However, we chose to use the scheme used by most existing on-demand routing protocols. The buffer has a limited size and if it fills up, the packet at the head of the buffer is dropped to make room for the incoming data packet. Each data packet also has a time value, which is set to the time when the packet is put into the buffer. A packet that has been in the buffer for more

than $data_pkt_timeout$ seconds is dropped. The data buffer is checked periodically for any packets that may be sent or dropped.

The query table is used to prevent queries from being forwarded indefinitely. We use a scheme similar to the one used in DSR, which allows for two kinds of queries: (a) queries with a zero hop count, which are propagated to neighbors only; and (b) queries with maximum hop count, which are forwarded to a maximum distance of MAX_HOPS hops from the sender. For each destination j, the query table contains the last time a maximum hop query was sent qs_j^i, the last time a zero hop query was sent zqs_j^i, the hop count of the last query sent hqs_j^i, the last time a query was received qr_j^i. At the source of the flood search, two maximum hop count queries are always separated by $query_send_timeout$ seconds. A query is forwarded by a receiver only if the difference between the time it is received and qr_j^i is greater than $query_receive_timeout$, where $query_receive_timeout$ is slightly lesser than $query_send_timeout$. The reasoning for this can be explained using Fig. 1. In the figure, times $t1$ and $t3$ correspond to times when the querying is started at the source and $t3 - t1 \geq query_send_timeout$. Since it is possible for the queries to travel different paths, we can have a condition where the first flood took a longer time to reach the forwarding node than the second flood, i.e., $(t2 - t1) \geq (t4 - t3)$. If $query_receive_timeout$ were equal to $query_send_timeout$, the second flood will not be propagated. However, we require the $query_receive_timeout$ to be large enough to prevent propagation of queries from the same flood search. This is the first protocol to use only local clocks to separate flood searches and this is an important advantage over using sequence numbers, because this makes the protocol impervious to node failures.

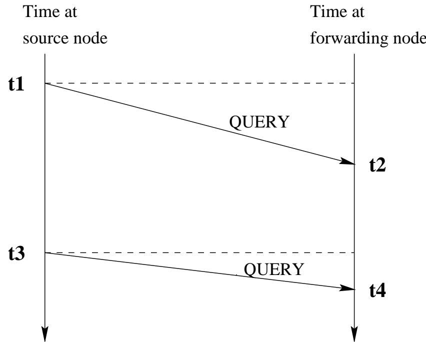


Figure 1: Querying timeline at source and forwarding nodes

3.2 Routing Information exchanged in DST

There are two types of control packets in DST - *queries* and *updates*. All control packets headers have the source of the packet ($pkt.src$), the destination of the packet ($pkt.dst$), the number of hops ($pkt.hops$) and an identifier $pkt.type$ that can be set to $QUERY$ or $UPDATE$. Each packet has a list of routing entries, where each entry specifies a destination j, a distance to the destination RD_j^i and a predecessor to the destination rp_j^i.

If the MAC layer allowed for transmission of reliable updates with no retransmission overhead (which is the case of wireless MAC protocols presented in [18, 20]), incremental routing updates would suffice to update routing tables among neighbors. In this paper, however, we assume a MAC protocol based on collision avoidance. In order to avoid collisions of data packets with other packets in the presence of hidden terminals , such protocols require nodes to defer for fixed periods of time after detecting carrier [7, 10]. Accordingly, sending larger control packets does not decrease throughput at the MAC layer, because the overhead ($RTS - CTS$ exchange) for the MAC protocol to acquire the channel does not depend on packet size. Therefore, in the rest of this paper, we assume that routers transmit their entire routing tables when they send control messages. Control packet size may affect the delay experienced by packets in the MAC layer. However, as our simulations show, this does not affect data packet delays because the number of control packets we generate is substantially low.

Data Packets in DST only need to have the source and destination in the header.

3.3 Creating Routes

When a network is brought up, each node (i) adds a route to itself into its routing table with a distance metric (D_i^i) of zero, the successor equal to itself (i) and the tag (tag_i^i) set to correct. To differentiate a route to itself from all other routes, a node sets the local host address (127.0.0.1) as the predecessor to itself.

When a data packet is sent by an upper layer to the forwarding layer, the forwarding layer checks to see if it has a correct path to the destination. If it does not, then the packet is queued in the buffer and the router starts a route discovery by broadcasting queries. Route discovery cycles are separated by $query_receive_timeout$ seconds. One zero hop query and one maximum hop query are sent in every cycle. A zero hop query allows the sender to query neighboring routing tables with one broadcast. If the zero hop query times out ((present time - zqs_j^i) $> zero_qry_send_timeout$), then an unlimited hop query (with $pkt.hops$ set to MAX_HOPS) is sent out. Consider the six-node network in Fig. 2.a where all link costs are of unit value and where node d broadcasts a query for destination a , with the $pkt.src$ set to d, $pkt.dst$ set to a, and $pkt.hops$ set to MAX_HOPS. The parenthesis next to each node in the example depicts the routing table entry (distance, predecessor) for destination a. The symbol lh stands for local host address (127.0.01). The query packet contains a list of all the routing table entries of the sender d. The entries are shown within the square brackets, each entry in the (destination, distance, predecessor) form. The entries are in a increasing-distance order, such that a node i receiving a query from an unknown neighbor k, adds the neighbor k to its distance tables on reading the first entry in the query and proceeds to consider all other entries as the distances reported by k.

Consider node e, where the query from d is received. To process the query, each entry (j, RD_j^d, rp_j^d) is read (procedure *Query* in Fig. 3). If the entry is for an unknown destination, then the destination is initialized ($D_j^i \rightarrow \infty$, $p_j^i, s_j^i \rightarrow NULL_ADDR$; $D_{jk}^i \rightarrow \infty$, $p_{jk}^i \rightarrow NULL_ADDR$

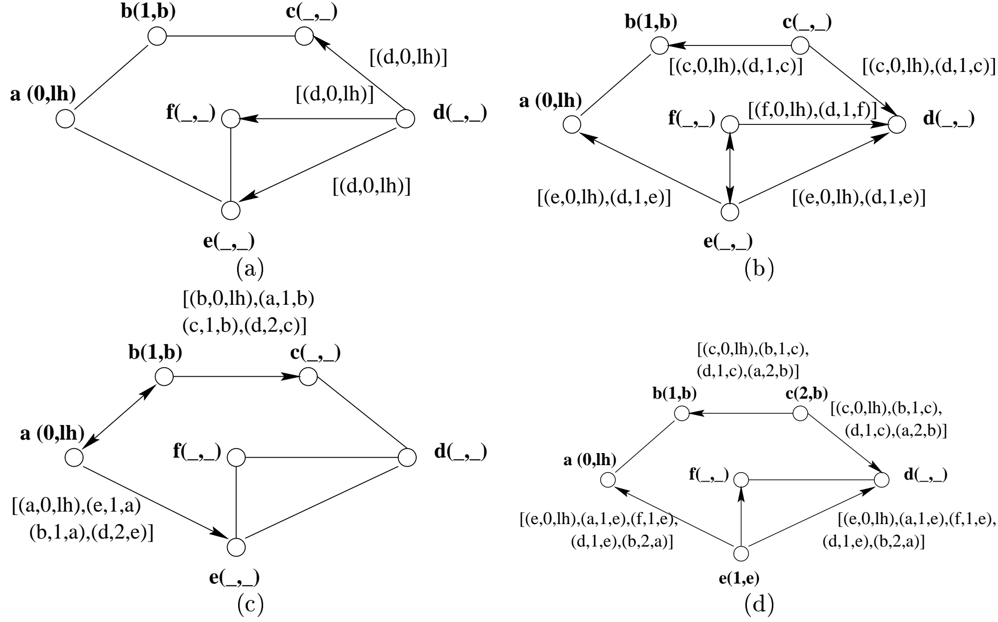


Figure 2: Example of the Query-Reply process in DST. Node d is searching for destination a. The parenthesis contains the distance and predecessor values for a.

$\forall k \in N_i$. Then, the distance table entry for neighbor d is updated in the procedure *DT_Update* (Fig. 3). Since the distance RD_d^d is equal to zero, d is marked as a neighbor. The procedure *DT_Update* (Fig. 3) also updates the value for j reported by other neighbors whose path contains d. This step helps prevent permanent loops by preemptively removing stale information.

Finally, procedure *RT_Update*(Fig. 3) is called to update routing table entries; this procedure iterates through each known destination and picks the neighbor k as a successor to destination j if both the following conditions are true

1. k offers the shortest distance to all nodes in the path from j to i.
2. the path from j to k does not contain i and does not contain any repeated nodes.

If either of the two conditions are not satisfied, then tag_j^i is set to *error*. Else, it is set to *correct* and neighbor k is designated the successor and the distance value to j is set to D_{jk}^i and the predecessor is set to p_{jk}^i.

After processing all entries and updating the routing table, node e checks to see if it has a route to a. Since there is no route, a query packet is broadcast with the same header fields as the processed query, besides *pkt.hops* which is decremented by one if all of the following conditions are met

1. Node does not have a route to *pkt.dst*
2. *pkt.hops* is greater than one
3. the time elapsed since the last query was received is greater than *query_receive_timeout*

The routing entries added to the forwarded query reflect the routing table entries of current node e. The packet is then broadcast to the limited broadcast address. In Fig. 2.b, nodes e, f and c broadcast queries.

In Fig. 2.c, we see that nodes e, f, and a do not send any more queries, because the time elapsed since the last query sent is shorter than *query_receive_timeout*. On the other hand, at nodes a and b, a finite and valid route to a is found and a reply update is sent. A reply update sent by a node i has a different structure than a regular update, which has *pkt.dst* set to the limited broadcast address and *pkt.src* set to i. The reply update sent by b has field *pkt.dst* set to the $pkt.src = d$ of the query and the field *pkt.src* set to the $pkt.dst = a$ of the query. All updates are broadcast to the limited broadcast address.

When node i receives an update, it checks the value of *pkt.dst*. If the value is other than the limited broadcast address, then the update being sent is a reply update, else it is a regular update. As shown in procedure *Update* of Fig. 3, the entries are processed in a manner similar to the entries of the query. A regular update is broadcast in response to a regular update, with *pkt.dst* set to the limited broadcast address and *pkt.src* set to i if any of the two following conditions is satisfied

1. Distance to a known destination increases.
2. A node loses the last finite route to a destination.

The reply update has different rules for propagation. In Fig 2.d , a reply update is rebroadcast by e with the original

$pkt.dst$ and $pkt.src$, because the following two conditions are met

1. A finite path to $pkt.dst = d$ exists.
2. Distance to $pkt.src = a$ changes from infinite to finite after processing the reply update.

Nodes a and b do not rebroadcast reply updates, because the second condition is not satisfied. Node d obtains a reply update from node e and sets its successor to node e after processing the entries in the query. Node d does not send any more reply updates. However, a regular update is sent if any of the two conditions for regular updates are satisfied.

Using the above procedure, DST allows a source to obtain multiple paths to a required destination. By forwarding a reply update only when the route to the required destination changes from infinite to finite, the number of updates is reduced at the expense of non-optimal routes. The same reasoning motivates not sending regular updates when a new destination is found or when a distance to a destination reduces. However, distance increases prompt updates because a loop can occur *only* when a node picks as successor a neighbor that has a distance greater than itself.

3.4 Maintaining Routes

DST does not poll neighbors constantly to figure out link connectivity changes, which avoids control overhead due to periodic update messages, but may result in sub-optimal routes and longer convergence time. A link to a neighbor is discovered only when a update or a query is received from that neighbor. On finding a new neighbor k the neighbor is added to the distance table. (procedure *Add_Nbr* (Fig. 3)) An infinite distance to all destinations through k is assumed, with the exception of node k itself and any destinations reported in the received routing message.

A failure of a link is detected when a lower level protocol sends an indication that a data packet can no longer be sent to a neighbor. The procedure *Rmv_Nbr* (Fig. 3) is called to remove the neighbor from the distance tables. Then, the procedure *RT_Update* is called to recompute distances to all destinations. Consider the six-node network in Fig. 4.a which is the same as that in Fig. 2 after the route discovery cycle started by node d for node a is done.

Consider Fig. 4.b, in which the link between a and e fails. Node e does not pick any of its neighbors f and d as successors because tracing the path in *RT_Update* allows node e to conclude that it lies in the paths of both f and d towards a. Thus, counting to infinity is avoided by the source tracing algorithm. Since there has been a distance increase, node e broadcasts an update. In Fig. 4.c, node d picks node c as its successor and changes its distance to 3 and predecessor to b. Node d sends out a regular update because it increased its distance. Node f also sends an update, which we do not show for the sake of brevity.

Let us assume that, due to some outside interference or fading, node c does not receive node d's update. Meanwhile, in Fig. 4.d, the link between c and b fails. Because node c's distance tables reflect a path through node d with predecessor e, node c increases its distance to 3 and changes its predecessor to e. Node c then sends an update. We consider two different sets of events that could happen. In Fig. 4.e, the update from b reaches d and d changes its distance to infinity and send out updates which cause e, f and c to reset their distance to a to infinity. In Fig. 4.f, we consider the case where the update from node c to node d is lost. This results in node d picking c as successor and node c picking d as its successor, which implies a 2-hop loop in the tables of c and d. To prevent such situations, we provide two conditions that prevent data packets from looping. A data packet is dropped and a regular update is sent if

A. The data packet is sent by a neighbor that is in the path from the present node to the destination of the data packet.

B. The path implied by the neighbor's distance table entry is different from the path implied in the routing table.

If node c receives a data packet from node d for destination j, it drops the data packet, because node d is in it's path to j and sends a regular update. Node d eventually receives and update from c and resets its tables. Thus, DST avoids long-term loops.

3.5 Packet Forwarding

The data packet header contains only the source and the destination of the data packet. When a data packet originated at a node arrives at its forwarding layer, the packet is buffered if there is no finite route to the destination. The node then starts the route discovery process. If a finite and correct route is found, then the packet is forwarded to the successor as specified by the routing table.

If a data packet is not originated at a node, then the data packet is only buffered if there is no entry in the routing table for $pkt.dst$. In this case, route discovery is started by the intermediate node. If there is a correct and finite route then the data packet is first checked for conditions A and B described in Section 3.4. If the two conditions are satisfied, the data packet is forwarded to the successor $s^i_{pkt.dst}$. If there is route with infinite distance, then the packet is dropped and a regular update is broadcast to all neighbors. Eventually, the source of the data will learn of the loss of routes and it will restart the route discovery process.

4. THE BEST PROTOCOL

BEST assumes the same network model introduced for DST in Section 2.

BEST is a table-driven routing protocol that reacts to changes in link states proactively. It uses a routing table and distance table with the same functionality as the tables introduced for DST.

BEST does not require a data buffer or a query table as it is a table-driven routing protocol; data packets are dropped if no path exists. The only type of packets used in BEST are updates which have functionality similar to the regular updates in DST, i.e., they are unreliably broadcast to the limited broadcast address and contain (distance,predecessor)

```
Procedure Recv_Ctl_Packet(pkt, nbr)
when node i receives a control packet from nbr
begin
   if (pkt.type = QRY)
      Query(pkt,nbr)
   else
      if (pkt.dst = BDCAST_ADDR)
         Update (pkt,nbr)
      else
         if(pkt.dst ∈ N and tag^i_{pkt.dst} = correct)
            Update (pkt,nbr)
   end else
end

Procedure Add_Nbr(k)
called when node i learns of new neighbor k
begin
   N_i ← N_i ∪ k
   for all (j ∈ N)
      D^i_{jk} ← ∞
      p^i_{jk} ← NULL_ADDR
   end for all
end

Procedure Rmv_Nbr(k)
called when node i learns of loss of neighbor k
begin
   N_i ← N_i − k
   for all (j ∈ N)
      tag^i_j ← null
   send ← FALSE
   RT_Update(send)
   if (send = TRUE)
      send update with source(i) and destination(BDCAST_ADDR)
end

Procedure DT_Update(k,j,RD^i_j,rp^i_j)
updating distance table entry
begin
   if (RD^i_j < ∞)
      D^i_{jk} ← RD^i_j + 1
   else D^i_{jk} ← ∞
   p^i_{jk} ← rp^i_j
   for all (b ∈ N_i)
      if k is in path from i to j via b
         D^i_{jb} ← D^i_{kb} + RD^i_j
   end for all
end

Procedure Query(pkt,nbr)
called for processing query
begin
   for each entry (j, RD^i_j, rp^i_j) in pkt
      if (j ∉ N)
         if (RD^i_j = ∞)
            continue
         else
            initialize j
            if (RD^i_j = 0)
               Add_Nbr(j)
         end else
      end if
      else
         if (RD^i_j = 0 and j ∉ N_i)
            Add_Nbr(j)
         end else
      DT_Update(pkt.src, j, RD^i_j, rp^i_j)
   end for each
   send ← FALSE
   RT_Update(send)
   if (tag^i_{pkt.dst} = correct)
      send update with source(pkt.dst) and destination(pkt.src)
   else
      if ( present time - qr^i_{pkt.dst} > query_receive_timeout)
         if (pkt.hops > 1)
            send query with destination(pkt.dst), hops (pkt.hops − 1))
            and source(pkt.src)
         if (pkt.hops ≥ 1)
            qr^i_{pkt.dst} ← present time
      end if
   end else
end

Procedure Update(pkt,nbr)
called for processing update
begin
   newpath ← FALSE
   if (pkt.dst ≠ BDCAST_ADDR)
      if (pkt.src ∉ N or tag^i_{pkt.src} ≠ correct)
         newpath ← TRUE
   for each entry (j, RD^i_j, rp^i_j) in pkt
      if (j ∉ N)
         if (RD^i_j = ∞)
            continue
         else
            initialize j
            if (RD^i_j = 0)
               Add_Nbr(j)
         end else
      end if
      else
         if (RD^i_j = 0 and j ∉ N_i)
            Add_Nbr(j)
      end else
      DT_Update(pkt.src, j, RD^i_j, rp^i_j)
   end for each
   send ← FALSE
   RT_Update(send)
   if (pkt.dst = BDCAST_ADDR)
      if (send = TRUE) then send update source(i) and
         and destination(BDCAST_ADDR)
   else
      if (pkt.dst = i)
         if (send = TRUE) then send update source(i) and
            and destination(BDCAST_ADDR)
      else
         if (newpath = TRUE and (pkt.src ∉ N or tag^i_{pkt.src} ≠ correct))
            newpath ← FALSE
         if (tag^i_{pkt.dst} = correct and newpath = TRUE
            and pkt.src is not in the path to pkt.dst)
            send update with source(pkt.src) and destination(pkt.dst)
         else
            if (send) then send update source(i) and
               and destination(BDCAST_ADDR)
      end else
   end else
end

Procedure RT_Update(send)
updating routing table entries
begin
   for all (j ∈ N)
      if (j = i)
         continue
      DTMin ← Min{D^i_{jb} ∀b ∈ N_i}
      if (D^i_{js^i_j} = DTMin) then ns ← s^i_j
      else ns ← b|{b ∈ N_i and D^i_{jb} = DTMin}
      x ← j
      loop ← FALSE
      for (m = 0; m < |N|; m + +)
         visited[m] ← NULL_ADDR
      num_visited ← 0
      while ((D^i_{x ns} = Min{D^i_{xb} ∀b ∈ N_i})
         and D^i_{x ns} < ∞ and tag^i_x ← null and loop = FALSE)
         m ← 0
         while (m < num_visited)
            if (visited[m] = x or x = i)
               loop ← TRUE
         end while
         x ← p^i_{x ns}
      end while
      if (loop = FALSE and (p^i_{x ns} = IP_LOCALHOST or tag^i_x = correct))
         tag^i_j ← correct
      else
         tag^i_j ← error
      if (tag^i_j = correct)
         if (D^i_j < DTMin) then send ← TRUE
         D^i_j ← DTMin
         s^i_j ← ns
         if (D^i_j = 1) then p^i_j ← i
         else p^i_j ← p^i_{j ns}
      end if
      else
         if (D^i_j ≠ ∞) then send ← TRUE
         p^i_j ← NULL_ADDR
         s^i_j ← NULL_ADDR
         D^i_j ← ∞
      end else
   end for all
end
```

Figure 3: Specification of selected procedures in DST

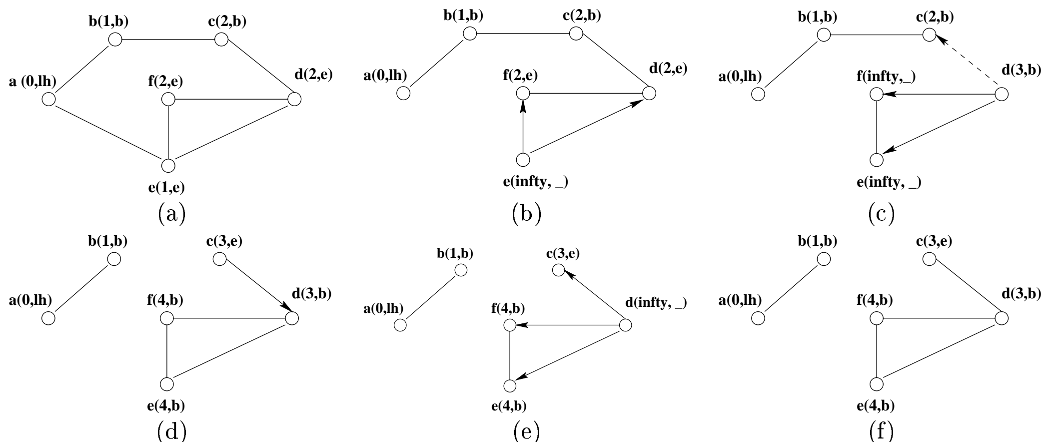


Figure 4: Maintaining routes in DST. The parenthesis contain the distance and predecessor values for destination a

tuples for all the destinations. BEST differs from WRP in allowing unreliable updates and in specifying different conditions to send updates. We focus our description of BEST on how updates are sent, because source tracing has been used extensively in the past in table-driven protocols [1, 13, 8]

The processing of an update in BEST is done in the same manner as in DST. When an update from neighbor k is received, the entries in the distance table corresponding to neighbor k are updated. The paths to each destination are then recomputed. BEST sends updates only if any of the following conditions have been met.

1. A node discovers a new destination with a finite and valid path to the destination.
2. A node loses the last path to a destination.
3. A node suffers a distance increase to a destination.

Two more conditions are added to prevent permanent looping due to unreliable broadcasts. These conditions are the same as conditions A and B in Section 3.4. When either of these conditions are satisfied, the data packets are dropped.

Permanent looping can occur when nodes are unaware of the latest changes in their neighbor's routing tables. The use of conditions A and B can be explained with the help of the example shown in Fig. 5.a. The node addresses are marked in bold font. Node j is the required destination. The path to j implied by traversing predecessors from j is marked in italics. Initially, all nodes have loop-free routes. The loss of links (i, j) and (m, j) and the loss of update packets from i and m can result in a loop shown in Fig. 5.b. When i gets a data packet from k, it finds that its distance table entry for k implies the path ij, while i's own path implies $ilmj$ which is different from ij. Therefore due to condition B, the data packet is dropped and a unicast routing update is sent resulting in k setting its path to kmj. Now, when k gets a data packet from m, it sends a unicast update to m because m is its successor on the path to j. This follows from condition A. When m gets the update, it detects a loop and resets its distance to infinity, thus breaking the loop.

The rules used in BEST to avoid permanent loops are much simpler than those introduced in STAR [9] which uses the link-state information in source trees, rather than distance and second-to-last-hop information to a destination in the tree.

5. PERFORMANCE EVALUATION

We ran simulations for two different experimental scenarios to compare DST's average performance against the performance of DSR and BEST. These simulations allowed us to change input parameters independently and check the protocol's sensitivity to these parameters. All three protocols are implemented in *CPT*, which is a C++ based toolkit that provides a wireless protocol stack and extensive features for accurately simulating the physical aspects of a wireless multi-hop network. The protocol stack in the simulator can be transferred with a minimal amount of changes to a real embedded wireless router. The stack uses IP as the network protocol. The routing protocols directly use UDP to transfer packets. The link layer implements the IEEE 802.11 standard [2] and the physical layer is based on a direct sequence spread spectrum radio with a link bandwidth of 1 Mbit/sec.

To run DSR in CPT, we ported the DSR code available in the *ns2* wireless release [6]. There are two differences in our DSR implementation as compared to the implementation used in [3]. Firstly, we do not use the *promiscuous* listening mode in DSR. We, however, implement the promiscuous learning of source routes from data packets. This follows the specification given in the Internet Draft of DSR. Our reason for not allowing promiscuous listening is that, besides introducing security problems, it cannot be supported in any IP stack where the routing protocol is in the application layer and the MAC protocol uses multiple channels to transmit data. The second difference in our implementation is that since the routing protocol in our stack does not have access to the MAC and link queues, we cannot reschedule packets that have already been scheduled over a link (for either DSR, DST or BEST). Tables 1 and 2 show the constants used in the implementation of DSR and DST, respectively.

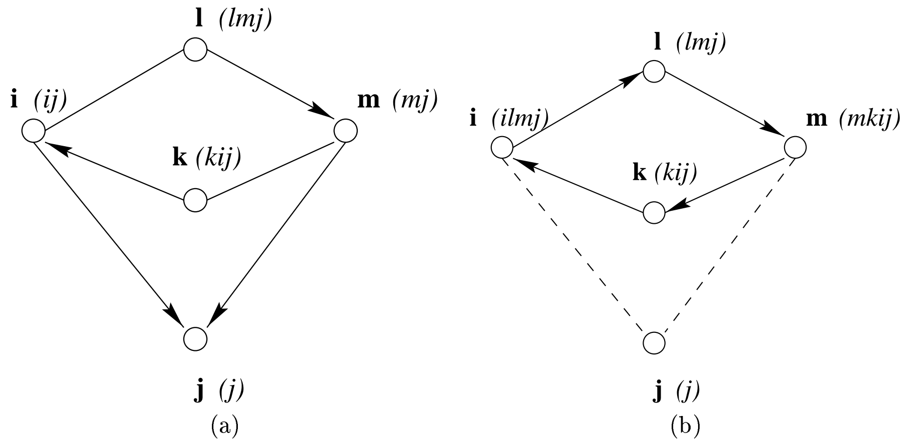


Figure 5: Creation of a permanent loop in BEST due to unreliable updates

Table 1: Constants used in DSR simulation

Time between ROUTE REQUESTS (exponentially backed off)	500(msecs)
Size of source route header carrying carrying n addresses	$4n$+4(bytes)
Timeout for Ring 0 search	30(msecs)
Time to hold packets awaiting routes	30 secs
Max number of pending packets	50

Table 2: Constants used in DST simulation

Query send timeout	5(secs)
Zero query send timeout	30(msecs)
Data packet timeout	30 secs
Max number of pending packets	50
Query receive timeout	4.5 (secs)
MAX_HOPS	17

5.1 Scenarios used in comparison

We compared DSR, DST and BEST using two types of scenarios. In both scenarios, we used the "random waypoint" model described in [3]. In this model, each node begins the simulation by remaining stationary for *pause time* seconds and then selects a random destination and moves to that destination at a speed of 20 m/s. Upon reaching the destination, the node pauses again for *pause time* seconds, selects another destination, and proceeds there as previously described, repeating this behavior for the duration of the simulation. We used the speed of 20m/s (72 km/hr), which is the speed of a vehicle, because it has been used in simulations in earlier papers [3, 4] and thus provides a basis for comparison with other protocols. All simulations are run for 900 seconds. In both scenarios, we used a 50 node ad hoc network, moving over a flat space of dimensions 7 X 6 miles (11.2 X 9.7 km) and initially randomly distributed with a density of approximately one node per square mile.

Two nodes can hear each other if the attenuation value of the link between them is such that packets can be exchanged with a probability p, where $p > 0$. Attenuation values are recalculated every time a node moves. Using our attenuation calculations, radios have a range of approximately 4 miles (135 db).

5.2 Metrics used

In comparing the protocols, we used the following metrics:

- *Packet delivery ratio*: The ratio between the number of packets received by an application and the number of packets sent out by the corresponding peer application at the sender.
- *Control Packet Overhead*: The total number of routing packets sent out during the simulation. Each broadcast packet/unicast packet is counted as a single packet.
- *Hop Count*: The number of hops a data packet took from the sender to the receiver.
- *End to End Delay*: The delay a packet suffers from leaving the sender application to arriving at the receiver application. Since dropped packets are not considered, this metric should be taken in context with the metric of packet delivery ratio.

Packet delivery ratio gives us an idea about the effect of routing policy on the throughput that a network can support. It also is a reflection of the correctness of a protocol.

Control packet overhead has an effect on the congestion seen in the network and also helps evaluate the efficiency of a protocol. Low control packet overhead is desirable in low-bandwidth environments and environments where battery power is an issue.

In ad hoc networks it is sometimes desirable to reduce the transmitting power to prevent collisions. This will result in packets taking more number of hops to reach destinations. However, if the power is kept constant, the distribution of the number of hops data packets travel through is a good measure of routing protocol efficiency.

Average end-to-end delay is not an adequate reflection of the delays suffered by data packets. A few data packets with high delays may skew results. Therefore, we plot the

cumulative distribution function of the delays. This plot gives us a clear understanding of the delays suffered by the bulk of the data packets. Delay also has an effect on the throughput seen by reliable transport protocols like TCP.

5.3 Performance results

5.3.1 Scenario 1

Scenario 1 mimics the behavior of an emergency network or a network set up for military purposes. Scenario 1 is almost identical to to the one presented in [3], barring any differences due to implementation of the MAC protocols.

We have 20 random data flows, where each flow is a peer-to-peer constant bit rate (CBR) flow with a randomly picked destination and the data packet size is kept constant at 64 bytes. Data flows were started at times uniformly distributed between 20 and 120 seconds and they go on till the end of the simulation at 900 seconds. We did 7 runs of the simulation where each run had different sets of source-destination pairs. The total load on the network is kept constant at 80 data packets per second (40.96 kbps) to reduce congestion. Our rationale for doing this is that increasing the packet rate of each data flow does not test the routing protocol. On the other hand, having flows with varying destinations does so. We also vary the pause times: 0, 30, 60, 120, 300, 600 and 900 seconds as done in [3].

Fig. 6.a shows the control packet overhead for varying pause times. An obvious result is that the control packet overhead for all the three protocols reduces as the pause time increases. BEST and DST are about 34 % better than DSR at pause time zero. At low rates of movement, DST is a clear winner with one third the control packet overhead of BEST and one tenth the control packet overhead of DSR. Clearly, the fact that the updates in DST contain the entire routing table, means that nodes running DST have a higher chance of knowing paths to destinations for whom no route discovery has been performed in the past. We are able to mimic the behavior of table-driven routing protocols in low topology change scenarios, in that we almost have information about the entire topology with very few flood searches.

As shown in Fig. 6.b, the percentage of data packets delivered is almost the same for DST and BEST. At lower pause times, DSR has the same packet delivery ratio as DST and BEST. However, as the pause time decreases, DSR suffers due to data packets getting dropped at the link layer, indicating that the routes provided in the source routes are not correct any more. At lower pause times, links get broken faster. Even though this results in higher control overhead, the routes obtained are relatively new. As mentioned earlier, we keep the load on the network constant. Since this load is divided among a large number of flows, we see very little congestion and therefore most packets get through at higher pause times during which the topology is close to static.

For Fig. 6.c we collated the hop count values for data packets during all pause times and plotted the hop distribution. All three protocols have almost the same number of one hop packets, indicating that the zero hop query is very effective in getting routes to neighbors. However, for the number of hops greater than one, we see that BEST performs the best. This is expected of a table driven routing protocol that tries to maintain valid routes at most times. DST's behavior is slightly worse than BEST. DSR on the other hand sends packets through longer routes. This is a direct consequence of the fact that after the initial query-reply process DSR pretty much uses the route it caches, without trying to better them.

Fig. 6.d shows the cumulative delay of all the protocols. The graphs shown are logarithmic in time to accommodate the wide variation. We see that BEST performs better than DSR or DST, with DST being very close. Almost all packets are sent within 4 seconds in BEST and within 8 seconds in DST. Some packets in DSR take almost 30 seconds. This is because a packet is allowed to stay in a buffer for a maximum of 30 seconds before it is dropped. These are packets that found the path just in time.

5.3.2 Scenario 2

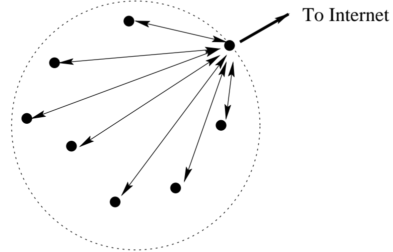
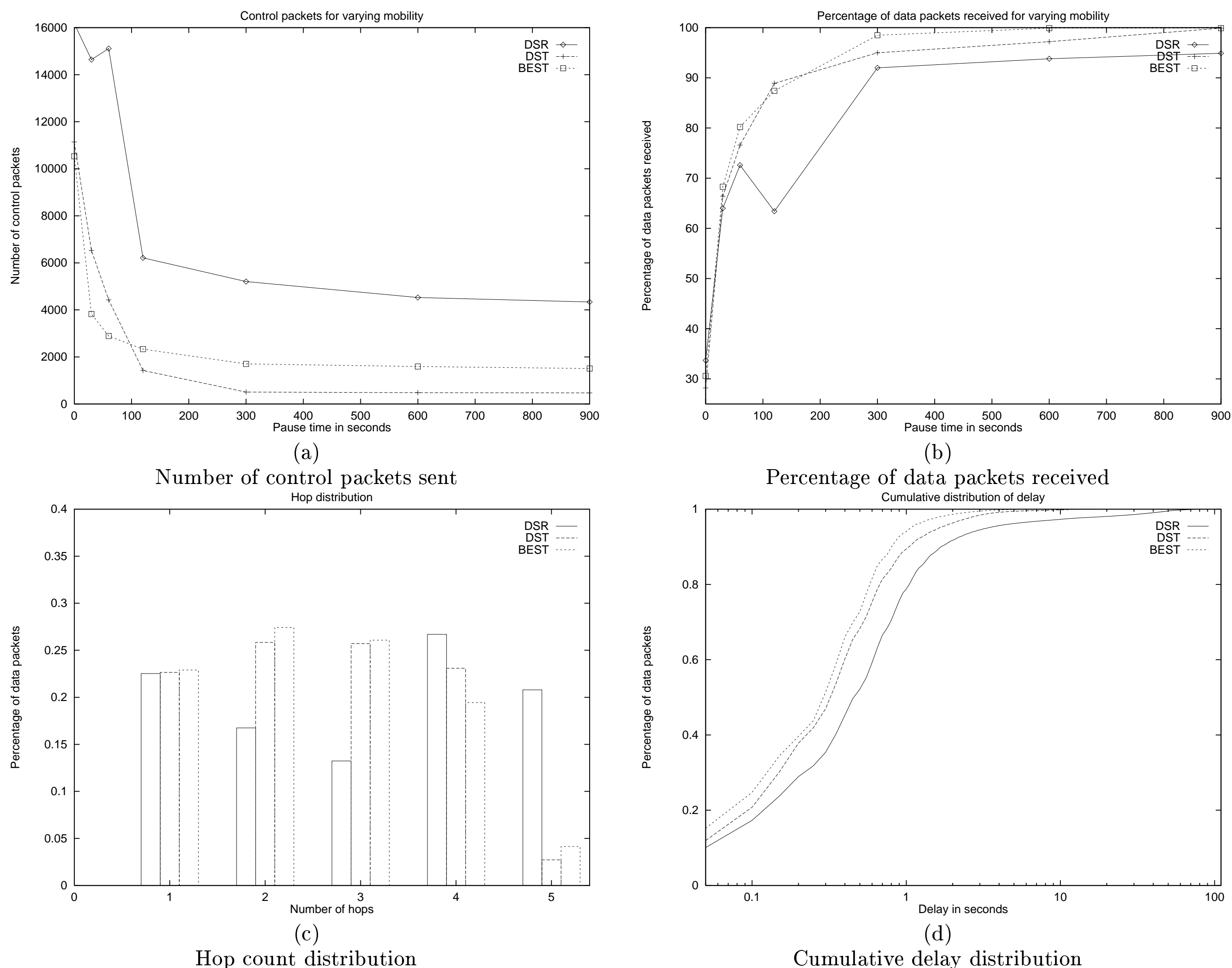


Figure 7: Scenario 2

Scenario 2 mimics the applications of ad hoc networks as wireless extensions to the Internet. In this case, one or two nodes act as points of attachment of the ad hoc network to the Internet. Accordingly, all Internet traffic travels to and from the attachment points as shown in Fig. 7. To model this situation, we pick one node as the point-of-attachment to the Internet for a simulation run of 900 seconds and we do five such runs and plot our results. During each run, the sender node first establishes a low rate connection (5.85 kbps) with the point-of-attachment. Immediately after the forward connection is established, the backward connection is started from the point-of-attachment to the sender. This connection has a higher rate of 40.96 kbps. Each pair of connections lasts for 300 seconds. In each epoch of 300 seconds, we start seven pairs at random times. This setup closely resembles number of nodes accessing the Web through the point-of-attachment. We run our simulations for two pause times, 0 (continuous movement) and 900 (no movement).

Fig. 8.a and Fig. 8.b show the results for the case of continuous movement. We see that BEST has almost double the control packet overhead of DST or DSR. The protocol is essentially reacting to the high rate of topology changes. The traffic does not seem to influence the behavior of BEST, because the same information needs to be maintained no matter what point-of-attachment is used. DSR and DST have almost the same behavior in terms of control overhead. DSR performs well in this traffic pattern, because with every flood search towards the point-of-attachment, the point-of-attachment learns the reverse path to the source from the source route accumulated in the queries. Another reason is that the fast changing topology forces out stale routes from



(a) Number of control packets sent

(b) Percentage of data packets received

(c) Hop count distribution

(d) Cumulative delay distribution

Figure 6: Results for 20 sources picking random destinations for peer-to-peer flow

DSR caches. This also results in DSR sending about 10 % more data packets than DST or BEST as shown in Fig 8.b.

Fig. 8.c and Fig. 8.d show us the results for the static case. This scenario is important because it resembles a static community network, e.g., households with wireless routers used to reach the Internet through an access point. In this case, BEST incurs about 3 times more control overhead than DST, whereas DSR incurs 14 times more control overhead than DST. DST performs this well because the entire network knows the path to the point of attachment with a single flood search. Since there are no topology changes, there is no need for another flood search. BEST also performs much better for a static network than for a dynamic one. No topology changes mean no table driven updates after the initial updates sent when the network comes up. The surprising result is the really bad behavior by DSR, most of which seems to be driven by increase in flood searches caused by old routes. A similar behavior is seen in terms of the ratio of data packets received. DST and BEST lose very few packets, while DSR seems to lose about 50% of them. As congestion due to control packets increases, we observe more and more data packets being dropped.

6. CONCLUSIONS

We presented source tracing as a new approach to achieve efficient routing in ad hoc networks using either on-demand routing or table-driven routing protocols. Simple rules were introduced in DST for the use of source tracing on demand, and simple rules were introduced in BEST for the efficient use of source tracing within the context of table-driven routing. The rules used in BEST are simpler than those introduced for STAR [9], which is the only other table-driven routing protocol that has been shown to be as efficient as on-demand routing protocols.

Simulations were used to compare DST and BEST with DSR, which is one of the most efficient on-demand routing protocols. The results showed that DST provides comparable average delays and packet delivery ratios while incurring far less control overhead than DSR or BEST. In our first scenario, which closely resembled an ad hoc scenario for a bat-

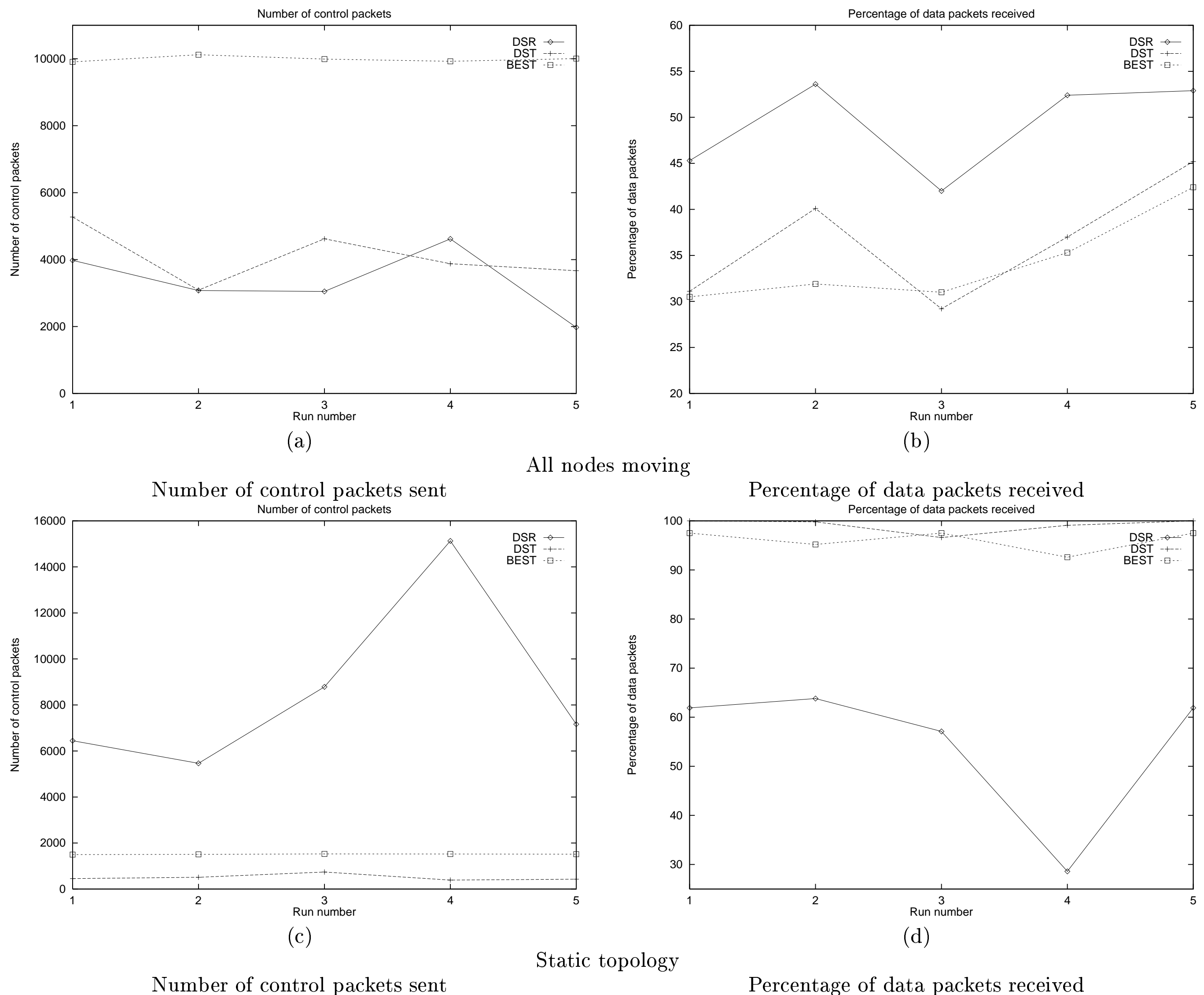


Figure 8: Results for single point of attachment

tlefield or an emergency situation, DST had about one-tenth the control overhead of DSR while delivering packets with the same efficiency as BEST, which is table-driven. BEST, has about one-third the control overhead of DST while having the best results for hop count and delay. For the second scenario, which is comparable to community networks accessing the Internet via wireless links, DSR had almost 14 times more overhead than DST, which suggests that DST is an ideal solution for static community networks. In static networks, the poor performance of DSR in terms of delay and throughput suggests that it needs a mechanism to flush out stale routes in static scenarios. In scenario 2, BEST incurs twice the overhead of DSR and DST when all the nodes are moving. This suggests that a table-driven routing protocol is a wrong choice for scenarios with many topology changes and only a few destinations. On the other hand, BEST delivers almost all the packets and has one fourth the control overhead of DSR for the static version of scenario 2, which implies that it may be used as a solution for community networks, though DST is a better option.

Given that BEST provided good results for application-oriented metrics like hop count and delays, which are of vital significance for QoS sensitive flows, it appears that an ideal routing protocol would have to use table-driven updates for certain sources and on-demand approach for others. This can be achieved with the proper combination of source tracing rules.

7. REFERENCES

[1] S.P.R. Kumar C. Cheng, R. Reley and J.J. Garcia-Luna-Aceves. A Loop-Free Extended Bellman-Ford Routing Protocol without Boumcing Effect. *ACM Computer Communications Review*, 19(4):224–236, 1989.

[2] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN Medium Access Control*

(MAC) and Physical Layer (PHY) Specifications. The Institute of Electrical and Electronics Engineers, 1997. IEEE Std 802.11.

[3] J. Broch et. al. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proc. ACM MOBICOM 98*, Dallas, TX, October 1998.

[4] Per Johansson et. al. Scenario Based Performance Analysis of Routing Protocols for Mobile Ad hoc Networks. In *Proc. ACM Mobicom'99*, Seattle, Washington, August 1999.

[5] R. Dube et. al. Signal Stability-Based Adaptive Routing (SSA) for Ad hoc Mobile Networks. *IEEE Pers. Commun.*, February 1997.

[6] Kevin Fall and Kannan Varadhan. *ns notes and documentation*. The VINT Project, UC Berkeley, LBL, USC/ISI and Xerox PARC, 1999. Available from http://www-mash.cs.berkeley.edu.

[7] C.L. Fullmer and J.J. Garcia-Luna-Aceves. Solutions to Hidden Terminal Problems in Wireless Networks. In *Proc. ACM SIGCOMM'97*, Cannes, France, September 1997.

[8] J.J. Garcia-Luna-Aceves and S. Murthy. A Path Finding Algorithm for Loop-Free Routing. *IEEE/ACM Trans. Networking*, February 1997.

[9] J.J. Garcia-Luna-Aceves and M. Spohn. Source-Tree Routing in Wireless Networks. In *Proc. IEEE ICNP 99, 7th International Conference on Network Protocols*, Toronto, Canada, 1999.

[10] J.J. Garcia-Luna-Aceves and A. Tzamaloukas. Reversing The Collision-Avoidance Handshake in Wireless Networks. In *Proc. ACM/IEEE Mobicom'99*, Seattle, Washington, August 1999.

[11] Z. Haas and M. Pearlman. The Performance of Query Control Schemes for the Zone Routing Protocol. In *Proc. ACM SIGCOMM '98*, Vancouver, British Columbia, August 1998.

[12] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad hoc Wireless Networks. *Mobile Computing*, 1994.

[13] S. Murthy and J.J Garcia-Luna-Aceves. An Efficient Routing Protocol for Wireless Networks. *ACM Mobile Networks and Applications Journal*, 1996.

[14] V. D. Park and M. S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proc. IEEE INFOCOM'97*, Kobe, Japan, April 1997.

[15] C. E. Perkins and P. Bhagwat. Highly Dynamic Distance-Sequenced Distance-Vector(DSDV) for mobile computers. *Computer Communication Review*, 24(4):234–244, October 1994.

[16] C. E. Perkins and E. M. Royer. Ad Hoc On-Demand Distance Vector Routing. In *Proc. of IEEE WMCSA'99*, New Orleans, LA, 1999.

[17] C. E. Perkins S. R. Das and E. M. Royer. Performance Comparison of Two On-Demand Routing Protocols for Ad hoc Networks. In *Proc. of IEEE Infocom 2000*, Tel Aviv, Israel, Mar 2000.

[18] Z. Tang and J.J. Garcia-Luna-Aceves. Hop-Reservation Multiple Access (HRMA) for Ad hoc Networks. In *Proc. IEEE INFOCOM'99*, March 1999.

[19] C.K. Toh. Associativity-Based Routing for Ad hoc Mobile Networks. *Wireless Personal Communications Journal, Special Issue on Mobile Networking and Computing Systems, Kluwer Academic Publishers*, 4(2):103–109, Mar. 1997.

[20] C. Zhu and S. Corson. A Five-Phase Reservation Protocol (FPRP) for Mobile Ad Hoc Networks. In *Proc. IEEE Infocom*, 98.