Using Minimal Source Trees for On-Demand Routing in Ad Hoc Networks

Soumya Roy, J.J.Garcia-Luna-Aceves

Abstract—**The on-demand routing protocols that have been proposed to date use either path information (e.g., DSR) or distance information (e.g., AODV). We present SOAR, an on-demand link-state protocol based on partial link-state information in which a wireless router communicates to its neighbors the link states of only those links in its source tree that belong to the paths it chooses to advertise for reaching destinations with which it has active flows. SOAR does not require periodic link-state advertisements when there are no link connectivity changes in the network. Simulation studies for several scenarios of node mobility and traffic flows reveal that SOAR performs more efficiently than DSR, which is one of the best performing on-demand routing approaches based on path information.**

Keywords—**Mobile Networks, Wireless Networks, On-Demand Routing, Ad-Hoc networks, Link-State Routing**

I. Introduction

Multihop packet radio networks (or ad-hoc networks) consist of mobile routers that interconnect attached hosts. These networks play an important role in relief scenarios and battlefields, where there is no base infrastructure. Communication between notebook or palmtop computers in conference scenarios can also be achieved using the ad-hoc networks. The topology of such networks is very dynamic because of host and router mobility, signal loss, interference, and power outages. The bandwidth available is also much less compared to wired networks.

To minimize the control overhead, on-demand routing protocols maintain paths to only those destinations to which data must be sent and the paths to such destinations need not be optimum (e.g., DSR [1], AODV[2], TORA [3], ROAM [4]). The basic differences among these protocols are how they communicate information to obtain paths to destinations, how they use and maintain the information, and the way in which data packets are routed. All on-demand routing protocols proposed to-date use flood search messages that either give sources the complete paths to destinations (e.g., DSR) or provide only the distances and next-hops to destinations and validate such distances with sequence numbers (e.g., AODV) or timestamps (e.g., TORA), or internodal coordination (e.g., ROAM). Interestingly, there have been no detailed studies of on-demand routing protocols based on link-state information. Jacquet et. al. [5] present a link state routing protocol for dense mobile ad-hoc networks called Optimized Link State Routing (OLSR). OLSR is a pro-active routing protocol where the routers exchange periodic routing messages and periodic HELLO messages with the neighbors. It uses a concept of multipoint relays (MPRs), which act as intermediate routers from source to destinations and works best in dense networks. Hu et. al. [6] have proposed caching schemes for DSR in which paths to destinations are stored in the form of links for higher efficiency and the links are removed from link caches either by time outs or by ROUTE ERROR messages.

Recently, a routing protocol based on partial topology information named STAR (source tree adaptive routing [7]) was proposed in which wireless routers communicate to their neighbors their source trees, i.e., the state of links in the preferred paths to all destinations. Although STAR has been shown to be as efficient as such on-demand routing protocols as DSR [7], it requires each node to keep routing information for all network destinations, which may be undesirable in very large ad-hoc networks or networks in which battery life of nodes is at a premium.

This paper presents the source-tree on-demand adaptive routing protocol (SOAR), which is an on-demand routing protocol based on link-state information. Section II presents a detailed description of SOAR, in which wireless routers exchange minimal source trees, consisting of the state of the links that are in the paths used by the routers to reach active destinations. Active destinations are those for which the wireless router is a source of data packets, a relay, or a possible relay. Minimal source trees can be updated incrementally or atomically, and updates to source trees are validated using sequence numbers. A wireless router uses its outgoing links and the minimal source trees received from its neighbors to compute its own source tree using a local path selection algorithm. Our approach of caching path information in the form of links is similar to Hu and Johnson's [6]. However, while SOAR communicates a link only once in a source tree and validates each link with a sequence number, the approach in [6] specifies complete paths to destinations from which links are then extracted. Section III proves that, within a finite amount of time after the occurrence of the last topology change in the network, SOAR stops transmitting updates and routers have paths to active destinations that do not involve any loop. Section IV presents a comparative performance study of SOAR and DSR, which has been shown to require fewer control packets than other on-demand routing protocols (AODV and TORA) [1], [8]. The simulation results show that SOAR requires much fewer update packets than DSR, while providing similar average delays and packet delivery rates. Section V presents our conclusions.

II. SOAR Description

A. Overview

To describe SOAR, the topology of the network is modeled as a directed graph G = (V,E), where V is the set of nodes and E is the set of edges connecting the nodes. Each node has a unique identifier, by which routing protocols and other applications can identify it. Routers are assumed to operate correctly and information is assumed to be stored without errors.

Each link has a cost associated with it and it becomes infinite if the link fails. SOAR does not depend on a neighbor protocol

The authors are with Computer Engineering, University of California,Santa Cruz. E-mail: soumya,jj@cse.ucsc.edu

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

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2001		**00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Using Minimal Source Trees for On-Demand Routing in Ad Hoc Networks	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

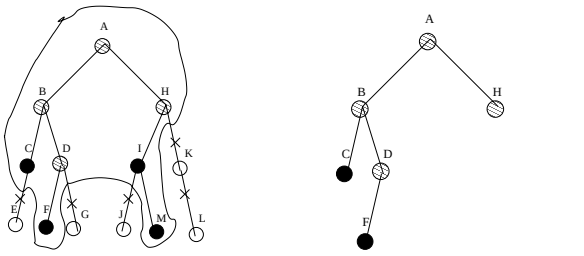
15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**10**	

for monitoring link connectivity with neighbors. SOAR declares a link as up when it receives a control packet from a new neighbor. It is assumed that either a link-level protocol can inform SOAR when data packets cannot be sent along a particular link, or SOAR can make that determination after a few transmissions to a neighbor. Control packets are sent unreliably and there may be packet losses due to changes in link connectivity, interference and signal loss. SOAR has been implemented on top of UDP and IP and has access to all data packets from the network layer as well as from the upper layers.

SOAR finds paths to destinations in an on-demand basis. When a router is asked to forward a data packet it forwards it to the next hop specified in the routing table if the next hop to the destination is known. Otherwise, the router sends a $query$ to its neighbors asking for the link-state information needed to produce a complete path to the destination. If the neighbors do not have a path, $queries$ are flooded to the entire network. Nodes send $replies$ in response to $queries$ if they have complete paths to the requested destinations. $Updates$ are exchanged when paths need to be updated to prevent loops or incorrect packet forwarding due to link connectivity changes.

All $updates$ are limited broadcast packets that travel one hop only and contain link states that belong to the *minimal* source trees used by routers to reach destinations. Fig.1 shows the *minimal* source tree advertised by a router (A) to its neighbors. In the example, router A knows about the links to nodes B,C,....,M and it has active flows with destinations C, F, I and M (shown as black nodes). The source tree reported by router A to its neighbors is a subset of the source tree it maintains. In the example of Fig.1, router A does not report links $I-J$, $H-K$, $K-L$, $D-G$ and $C-E$ (shown with cross-marks), and it advertises all the other links, shown within the curved boundary. SOAR uses hop-by-hop packet forwarding and a data packet specifies the path traversed, rather than the path to be traversed.



(a) Minimal source tree reported by A (b) Minimal exclusionary tree for A used by H

Fig. 1.

In SOAR, each router maintains a sequence number for each destination known by the router. A router increments its own sequence number when any of its adjacent links go down or is brought up. All the outgoing links of the router are identified by the same sequence number. If the partial topology table at a router contains links with the same head node but different sequence numbers, then the router keeps the links with the highest sequence number and removes the links with lower sequence numbers.

B. Information Maintained in SOAR

A router maintains a partial topology table T_i, a source tree ST_i, a routing table RT_i and the *minimal* tree ST_i^x, reported by each neighbor $x \epsilon N_i$, where N_i is the list of neighbors of i. A router also keeps a query table, a data buffer and a destination table (D_i) containing the highest node sequence number it has heard for each destination.

The routing table RT_i contains entries for those destinations that are reachable according to the information available at router i. Each entry in the routing table consists of the destination ID, the next hop for the destination and the cost of the path to the destination.

ST_i is the source tree used by router i to reach any destination, known to the router while ST_i^x is the *minimal* source tree of neighbor x advertised to router i. Though a router maintains a source tree, it determines its *minimal* source tree and reports that tree to its neighbors. The topology table T_i at router i specifies the links reported by its neighbors corresponding to paths that do not include router i in them. The topology table is obtained by the aggregation of all the *minimal exclusionary* trees reported by the router's neighbors. The *minimal exclusionary* tree corresponding to neighbor x at router i consists of the subset of ST_i^x with all branches rooted at i deleted. Fig. 1 shows an example of a *minimal* tree reported by A to H, and the corresponding *minimal exclusionary* tree used by H to compute its topology table.

Each link in i's topology table is identified by a tuple (u, v, l, s) where u is the head, v is the tail, l is the cost and s is the sequence number of the link. Each $node$ has a sequence number, and every time there is a link up or link down, the node increments its sequence number and it changes the value of s of each outgoing link to this new sequence number. When router i learns the same link (u, v) from several neighbors, it trusts the entry with the highest sequence number.

The data buffer is a queue that holds data packets waiting for routes to be discovered. The query table tracks the queries sent for each destination. For each destination, the query table logs the time when the last maximum-hop query was sent, the last time a zero-hop query was sent, and the last time a query was received.

C. Information Exchanged in SOAR

SOAR exchanges three types of control packets: $query$, $reply$ and $update$. $Query$ packets are sent when a node does not have a route to a destination for which it has a data packet to send. $Reply$ packets are sent by a node in response to $queries$ if it has a path to the destination queried. A node forwards a $query$ to its neighbors if it does not have a route to the destination for which it receives a $query$ from a neighbor. $Update$ packets are generated if next hop changes or the distance increases for any active destination after the reception of a control packet or after a link-connectivity change. The information transfered in control packets between nodes running SOAR is the *minimal* source tree. We denote by *important* nodes those nodes for which the router acts as a relay or a sender of data packets or those nodes which the router uses as a relay for data delivery. The *important* nodes of a router i are determined by doing a path traversal

through ST_i, in an order similar to the post-order walk in a binary tree. It should be noted that each node computes its own shortest path tree, but reports to its neighbors the *minimal* source tree containing links that are used to reach its *important* nodes. However, a router can choose to report the entire source tree to its neighbors instead of reporting *minimal* source trees.

Control packets are broadcast and are sent unreliably. Because the loss of control packets can lead to wrong path information and loop formations, the path traced by a particular data packet is kept in its header. When a node receives a data packet to forward, it reads the path traversed by the packet in the packet header and checks whether forwarding it to the successor, specified in the routing table leads to a loop. If it detects that the packet can go in a loop, it sends out an $update$ and determines if any of its neighbors has an alternate path to the destination that does not have any of the nodes specified in the path traversed by the packet. A router also sends an $update$ if it receives a data packet for forwarding and it does not have a route to the destination. This is to ensure that its neighbors having an outdated view of its *minimal* source tree is updated.

D. Operation of SOAR

Arrival of Data Packets: When a data packet arrives from the application layer and the router has a valid path to the destination, it immediately forwards the packet. Otherwise, it initiates a route discovery process by sending a non-propagating $query$ and keeps the data packet in its *data buffer*. If the data packet arrives from the network, and the router does not have a valid path to the destination, or finds that the packet can go in a loop if forwarded as indicated by the routing table, the packet is discarded and an $update$ is sent to all neighbors. To prevent an $update$ to be sent for each data packet received from a burst of data packets with no next hop or headed for a potential loop, a *minimum update time* is enforced in the transmission of consecutive updates. This time spacing of updates is maintained only for those $update$s generated in response to information obtained through data packets.

While forwarding a data packet, if the router finds that the next hop neighbor in the path to the destination is no longer a neighbor, it removes the entries corresponding to that nonexistent neighbor from its database. It then recomputes its routing table and tries to find an alternate path to the destination. If there is none, the packet is discarded if it came from the network; otherwise, it is kept in the data buffer while a route discovery process is started.

Two kinds of $queries$ are sent: non-propagating $queries$ which are meant for neighbors only and propagating $queries$ which travel up to a number of hops equal to the value specified int the *MAX_HOPS* field of $query$ packet. This is to prevent unnecessary flooding when the neighbors have a path to the required destination. Two path discovery processes are separated by $query\_send\_timeout$ seconds. Non propagating $queries$ are sent at the start of the path discovery process. If none of the neighbors send $reply$, propagating $queries$ are sent. If these $queries$ do not yield any response, then the route discovery process is restarted by sending a non-propagating $query$. Each time a response is not obtained during a a route discovery cycle the value of $query\_send\_timeout$ is doubled till a pre-defined number of attempts have been made, after which it is kept constant.

Arrival of Control Packets: All control packets are limited broadcast packets, but the src and dst are included to determine how to forward the control packets. $Queries$ have src set to the source of the $query$ and dst set to the destination being queried. Any node who sends a $reply$ interchanges the src and dst field, as if the dst replies to src, though some intermediate node may reply. For $update$s, the dst field is set to *BDCAST_ADDR*. When a router receives the first control packet from a node that is not in the neighbor list, it assumes the presence of a new neighbor within its range.

A $query$ for a particular destination is forwarded by a receiver if it does not have a path to *dst*, and if the $query$ has not traversed the maximum number of hops specified in that $query$ and if the difference between the present time and the time when the query for *dst* was last received is greater than $query\_receive\_timeout$. The last condition is imposed to limit the number of $queries$ in the network sent for a particular destination and originated from different sources. After receiving a $query$, a router marks *src* as *important*, so that it can maintain the correct path to the *src*, which it needs while propagating back the $replies$. While forwarding $replies$, the src, which was the dst in $query$, is marked as *important*. A node sends a $reply$ when it has a path to the destination queried. Because SOAR does not maintain up-to-date paths to all destinations, it may happen that the path advertised in a $reply$ is wrong and the sender or relay may realize that this is the case only after data packets start flowing along the path. Because that can add to loss of data packets, a node sends a $reply$ about a destination if it determines that the links in the path to the destination form part of its *minimal* source tree. A node forwards a $reply$ packet, if it has a path to the *dst* of the $reply$, has a new route to the *src* of the packet (this is to prevent multiple replies), and it is a node in the path from the source to the destination (this prevents sending replies to that part of the network, where this $reply$ is not asked for). If the node is not required to forward any $reply$, $update$s are sent if the distance to any *important* destination increases.

Each control packet contains several link-state-updates (LSUs) and each LSU is a tuple, $(u, v, l, v.seq\_no)$, where $v.seq\_no$ indicates the sequence number of the tail (v) of the link (u, v) [1]. The sequence number of the sender is kept in the SOAR header as a separate field. Every time a control packet is received, the sequence number for each known node j is updated to the highest sequence number heard for j; l refers to the cost of the link. The neighbor's *minimal* source tree and the partial topology table are updated using the information in valid LSUs in the control packet. A path selection algorithm (Dijkstra's SPF, Bellman-Ford) can be run on the partial topology table to determine the source tree and modify the routing table. An $update$ is sent if, according to the source tree, there is an increase of distance to any *important* destination or there is a change in the next hop. In addition, if some new destinations are obtained, packets waiting in the data buffer for the path to that destination are sent.

Path selection algorithms like Bellman-Ford or Dijkstra's SPF compute the shortest path in a graph from a source s to any other

[1] Link (u, v) implies the directed link from u to v while Link (v, u) implies the directed link from v to u.

destination. Due to changing network conditions, a situation may arise where the links in the shortest path to a particular destination t through a neighbor n may not have been learnt from n itself. In such a scenario, data packets if forwarded along n may be lost and $updates$ generated by n may not improve the condition. To remedy this situation, the path selection algorithm needs to be modified to ensure that the links in the anticipated path from s to t through n have been advertised by n, itself. This new class of path selection algorithms, unlike conventional shortest path algorithms that remember only the shortest path in each iteration, remembers all the paths that have been encountered while visiting a node and the information about the entire set of paths is passed to all its adjacent nodes for their individual computations. A link (x, y) is added to the source tree of node s if and only if there is at least one neighbor of s whose *minimal exclusionary tree* includes (x, y). The branch of the source tree of s leading to (x, y) constitutes the shortest path to node x in all the *minimal exclusionary* trees of the neighbors of s.

A router maintains a distance table D_i where each entry is of the form (j, $last\_heard_j$, seq_no_j) and $last\_heard_j$ refers to the time when the router has last seen a packet for destination j. When the difference between the present time and $last\_heard_j$ is greater than *refreshing_time*, the router is not interested in reporting routes for j, unless: (a) j is used as a relay for any other node k, and (b) the difference between present time and $last\_heard_k$ is less than *refreshing_time*. By the term "marking a node important" we mean that $last\_heard_j$ is updated to the $present\_time$.

To ensure that routers use up-to-date link state information to construct their source trees, a router sends update for *important* nodes in its source tree when their associated sequence number must be updated locally or at a given neighbor. When a router receives an LSU that increases the sequence number of any of its *important* nodes, it sends an $update$ to all its neighbors to propagate the updated sequence number for such *important* nodes. After several of these inter-nodal communications, the establishment of the same sequence number for each *important* node is referred to as the *synchronization* of the node sequence number. Before advertising the *minimal* source tree to its neighbors, a router ensures that none of the advertised links to its neighbors has a sequence number lower than the sequence number advertised for the head of those links.

There are two simple ways in which roll-over of sequence numbers can be supported in SOAR. In one approach, an aging field is used in addition to the sequence number of an LSU. The largest possible sequence number is sent with a zero age and each node is forced to delete the link from its tables and propagate such an LSU; furthermore after establishing a new link with a new neighbor, a node sends to its neighbor the last sequence number for the neighbor, so that the neighbor can start using a sequence number larger than such a value.

Another approach consists of using a timestamp together with the sequence number. The timestamp is maintained externally to the algorithm, and eliminates the need for resetting the sequence number, because the timestamp increases monotonically. For simplicity, in the rest of this description, we assume that the second scheme is used but omit the treatment of sequence number reuse in the proofs of correctness.

When a node receives the new *minimal* source tree from its neighbor, it updates the neighbor's entries in the database and its partial topology table. This process is illustrated using Fig. 2. For simplicity, assume that all the nodes have packets for every other node and so every other node is *important* for each node. Also assume that the network has converged to the same sequence number for each node, as indicated in Fig. 2. Here we will show how the partial topology table at a gets modified after link (b, c) fails. When link (b, c) goes down, b increments its sequence number to 35. The path to c breaks at b and so it sends an $update$, reporting its new *minimal* source tree. Node a receives the $update$ and modifies the entries of b. No $update$ has yet reached from f and so the *minimal exclusionary* tree of f at a remains unchanged. The links deleted from the old *minimal* source tree of b at a are (b, c), (c, d), (c, e). Links (c, d) and (c, e) do not appear in the *minimal* source tree of f; These links are deleted from the partial topology table at a. The node sequence number reported by b for itself is 35 and the node does not advertise link (b, c). Because every node must be using the shortest path to any destination, the only reason b has stopped using the link (b, c), is that (b, c) has failed or increased in cost or b does not use it, because there is an alternate lower cost path. Link (b, c) advertised by f has sequence number $34 < 35$. So as indicated in Fig.2, node a marks link (b, c) to be of infinite cost and having a sequence number of 35. The reason for setting the cost to infinity is to stop using the link as the neighbor has already stopped using it. This technique helps to inform routers that a link is no longer used for data delivery, without the explicit notification of the deletion of the link from the source tree.

At each event, SOAR needs multiple search of the database, the computational complexity of which has been greatly reduced by using hash tables. The most computationally intensive operations turn out to be the implementation of path selection algorithm (for Bellman-Ford, it is O($N.E$)) and the function that ensures that links advertised for the same head node have same sequence number (O(Nn)), when N is the number of nodes in the graph and n is the number of neighbors and E is the number of edges.

Buffer Timer: The buffer timer is set whenever there is a packet in buffer. When it times out, the packets waiting in the buffer are checked to see whether a new query has to be made for any destination. If there is one, $query$ is sent and the buffer timer is reset. If there is no packet in the buffer, the buffer timer is not started to prevent unnecessary interrupts.

Update Timer: The value of this Timer is referred to as *update timeout* in Table III. On reception of a control packet, an $update$ may become necessary for achieving synchronization of sequence numbers of $important$ nodes. The node waits for *update timeout* to allow for some time for $update$s to arrive from other neighbors, which can have data for the required synchronization.

III. Correctness of SOAR

This section addresses the correctness of SOAR. To simplify the proof, we assume the set of destinations for which a router is a source of packets is static, that the link layer can inform SOAR about link failure within a finite time after the link fails, and that control packets are exchanged reliably. We also assume

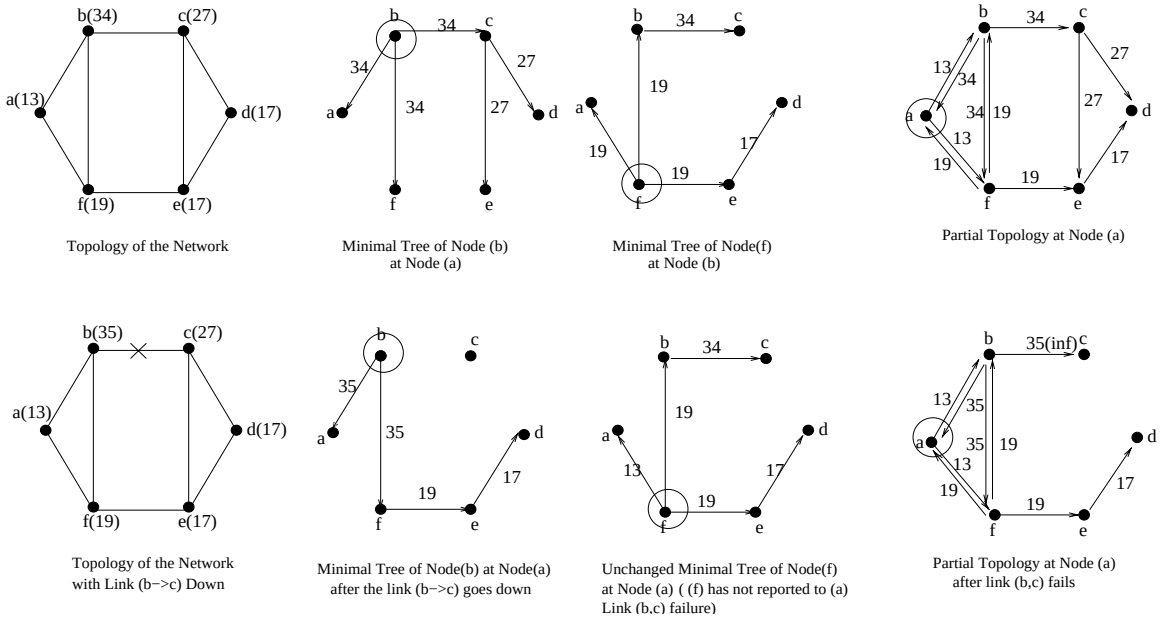


Fig. 2. Example showing how Link State Information is exchanged and updated in SOAR

that SOAR achieves correct reset of sequence numbers.

Theorem 1: Following a link cost change, there can only be a finite number of $updates$ generated for that change.

Proof: Suppose that a link cost change has occurred at t_0^+ and that there is no further change in the status of that link after that time. A node increments its sequence number upon undergoing a link cost change and assigns that value to all its outgoing links. If an outgoing link of a node decreases in cost, the node will not generate an $update$, because it does not lead to any increase in cost for any destination. If the head of the link does not generate any $update$, the theorem is proven.

Let us now prove the theorem for the case in which an $update$ is generated by the head node due to a link cost increase. After t_0^+, $updates$ can first be generated for (u, v) by the head u of a link (u, v), advertising a higher sequence number if the next hop changes or the path cost to any *important* destination increases due to a cost increase or failure of link (u, v). A node that receives an $update$ processes it within a finite time and sends an $update$ if next hop changes, the path to any *important* destination increases or if it needs to achieve *synchronization*, otherwise, no $update$ is sent. Equivalently, we can say that a node sends an $update$ only if it experiences an increase in the sequence numbers of some nodes.

If we can prove that a node can produce at most one $update$ for each link cost increase, the theorem is proved. This is because we have a network consisting of finite nodes and every node produces at most one $update$ for a link cost change and so we will have a finite number of $update$ messages for each link cost increase. This can be shown by contradiction.

Let us assume that a node x sends a second $update$ for the cost change of link (i, j), which can only be possible if it did not have the highest sequence number for node i before the event that caused the $update$ and after it has sent the $update$ it has the highest sequence number for i. Because x has already obtained the highest sequence number for node i, because it has generated the first $update$ for i, the second $update$ can only be sent if i has increased its sequence number after t_0^+, which implies that there is another link cost change after t_0^+. However, this contradicts the assumption made for the proof. So for one link cost change, any node can produce at most a single update. ■

Theorem 2: Within a finite time after the failure of a link (i, j) in the network at time t_0^+, all routers using link (i, j) at time t_0 stop assuming that link (i, j) exists and start finding alternate paths without link (i, j).

Proof: Suppose that there is a link failure at time t_0^+ and there is no link change after t_0^+. Let (i, j) be the link that fails. We have to prove that all routers using link (i, j) for data transfer at time t_0 start looking for alternate paths without link (i, j) within a finite time after t_0^+.

The link layer informs SOAR within a finite time that a link has gone down. If router i is not using link (i, j) to reach j at t_0, then none of the routers would be using link(i, j) at t_0; Therefore, no $update$ is produced at i as link (i, j) fails. Suppose router i is using link (i, j) for some active destination k and link (i, j) fails. Then router i will have a data packet to transfer over link (i, j) within a finite time after link failure, and SOAR will then find that the link has gone down and the router will increase its sequence number, because there is a link connectivity change.

If k is an active destination and if any router is using link (i, j) to reach k, then it generates $update$s when the cost of the path to k increases. Accordingly a node, that is n hops away from the head of the link (i, j): (a) processes the $update$, because it contains new sequence number for i (or processes link level information about link failure if $n = 0$); (b) stops using link (i, j); and (c) either sends an $update$ if the path to k increases on receiving the link failure information or remains silent if its next hop to the destination k remains same. There can be no alternate path to k of lower distance after receiving the link failure information.

Accordingly, an $update$ containing link failure information propagates up the tree rooted at i (Fig. 3) until either:

- It reaches a node X where no $update$ is required because the node experiences no path cost increase or next hop change, in which case no node upstream of X knows about the failure of link (i, j) and continues to use X as the next hop for k. Since X is not using link (i, j) the upstream node which still uses X, as next hop, effectively does not use (i, j).
- It reaches a node that is not using link (i, j) to reach destinations with which it has active flows.
- It reaches a node which has already updated the highest sequence number for i.

In all the above cases no further $update$s would be sent.

Because the link layer can detect a link failure within a finite time and a node can process a link failure information from an $update$ or link-layer indication within a finite time and the tree rooted at i is finite, it follows that within a finite time, all routers using link (i, j) at t_0 will not loose any more data packet thinking that link (i, j) exists. Using the same method, it can

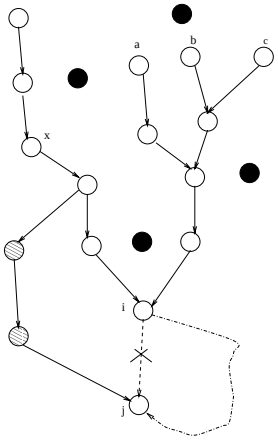


Fig. 3. Link failure information propagation in SOAR: White nodes have active flows with j (here $k = j$). Black Nodes do not care about paths for j.

be shown that if the link cost increases and the head of the link prefers to use different paths, all relevant nodes would also try to determine alternate paths within a finite time. ■

Lemma 1: If at time t_0 a node chooses a path for destination k, and the path is incorrect then data packets for k will stop traversing along that path within a finite time.

Proof: Suppose that node s chooses a path for k and within a finite time starts forwarding packets along that path. Because each node, that is in the path from s to k receives a packet for k, it marks k as *important*. When data packets are forwarded along the path, they will either reach a node (s_n), at a distance of n hops from s which has either a correct path to k or has no correct path to k. If the first condition is satisfied then the Theorem is proved because no correction of the path is necessary. For the second condition the problem at s_n becomes similar to the problem at s, which implies a recursion. Because the network is finite, after the data packet has traversed a finite number of hops, a node s_n that is n hops away from s, either selects a node s_∞ with no path to the destination or reaches a node s_{n+1} whose next hop for k is an already visited node s_i $(i <= n)$. In the first case s_∞ sends an $update$ advertising a higher sequence number for the head node of the link, which s_n still thinks exists. Node s_n processes the $update$ within a finite time, and finds that its original path is incorrect and in turn sends an $update$, advertising a higher sequence number for the head node of the link, which s_{n-1} still thinks exists. Accordingly s will rectify its path and the data stops flowing along the path.

We now show why data packets for a certain destination k will not go in a loop for an infinite time. Let $C_x(y)$ be the cost of the path to k at node x using link (y, k). Let us assume that a, b, c, d, e, f and g are involved in a loop (Fig.4). Because the downstream nodes always have a lower distance to the destinations, we have $C_a(x1) > C_b(x2) > ..... > C_f(x6)$. Now let us assume that g chooses a as the next hop and uses link $(x7, k)$. So $C_f(x6) > C_g(x7) > C_a(x7)$, which implies that $C_a(x1) > C_a(x7)$, that is, a has selected a path of higher distance, in which case a is supposed to send an $update$. Using a similar argument we can show that g's path also increases, in which case it sends an $update$ and the loop will break by backward propagation.

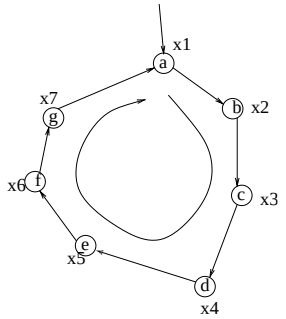


Fig. 4. Figure to depict how SOAR does not form permanent loops

If the $updates$ are unreliable, then it may have happened that the $update$ of a did not reach g. In that case a loop can persist. However SOAR sends $update$ if the packet is found to traverse in a loop, in which case g would send an $update$ and this breaks the loop. ■

Lemma 2: If a node does not have a path to k and has a data packet for k, it obtains a correct path to k, if there exists any, within a finite time after sending $query$.

Proof: A router s initiates a route discovery process by sending a non-propagating $query$. If none of the neighbors has any path, a propagating $query$ is sent which traverses multiple hops. When a router receives a $query$ for k, it marks s as *important* and hence reports the path to s in its control packets. [2] Hence, every node that forwards the $query$ knows how to forward the $reply$ back to s. Because the network is of finite size and is connected, at least one node (i.e. the head of the node reaching k) should be able to send a $reply$. Accordingly a $reply$ will be sent for this $query$ within a finite time by a node that has a path to k, and k is *important* to that node or has k as the end node in its outgoing link. All the intermediate nodes who have forwarded the $query$ know the path to s, and the $reply$ propagates back to s within a finite time. When the $reply$ propagates back to the sender, all the routers on the way marks k as *important*, and hence the $replies$ contain the path to k.

[2] A node stops considering s being *important*, if it has not received any packet for s for *refreshing_time* after marking it *important*.

For simplicity of the proof, we assumed that a $reply$ can only be sent by a node for k, if it has a path to k and k is an *important* node or has k as the end node in its outgoing link. The proof is still valid if a node sends in its $reply$ an old path. This will make the data packet flow along the wrong path for some time, but due to Lemma 1, the error would be detected within a finite time and in the worst case another $query$ has to be sent. ■

Theorem 3: If a path to a node breaks due to a link failure and there exists an alternate path, SOAR finds that path within a finite time.

Proof: From Theorem 2, we have that all nodes would stop using a failed link, except those whose downstream neighbors have not experienced a path change. Therefore all these downstream nodes have either selected a new path to k or do not have a path to k. By Lemma 1 any node that needs to send data along some new path, can detect within a finite time whether that path is correct. In the worst case when there is no alternate path, a route discovery is initiated and we know from Lemma 2 that, within a finite time, a $reply$ must come if there exists at least a single path. ■

Theorem 4: If a node becomes disconnected at time t_0^+, every node that considered that node to be *important* at $t_0 < t_0^+$, will have no path to it within a finite time.

Proof: Each node failure can be assumed to be equivalent to multiple link failures. Therefore using Theorem 2 and Lemma 1, we can say that after a node failure every node wishing to reach the failed node will have no path. ■

Although the above theorems assume reliable transmissions of updates, SOAR works properly for the case in which the control packets are sent unreliably. This is because data packets can reach a node who either detects a loop or finds no path to the destination. In both cases an $update$ must be generated and the network can recover within a finite time.

IV. Performance Evaluation

We ran a number of simulation experiments on a 20-node network under varying host mobility and network traffic to test the average performance of SOAR with respect to DSR. Both DSR and SOAR 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.[3] The stack uses IP as the network protocol. The routing protocols directly use UDP to transfer packets. The path selection algorithm coded in the simulation is simply a shortest path algorithm which leads to more control traffic in SOAR. The link layer implements a medium access protocol very similar to the IEEE 802.11 standard [9] 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 [10]. There are two differences in our DSR implementation as compared to the implementation used in [11]. First, we do not use the $promiscuous$ mode in DSR or SOAR. Besides introducing security problems, this feature 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. Second, the routing protocol in our stack does not have access to the MAC and link queues. Accordingly, packets, once scheduled over a link cannot be rescheduled if the link fails. Because both SOAR and DSR would benefit equally from such features, our comparative analysis is still valid.

[3]We thank NOKIA Wireless Routers for providing CPT.

A. Mobility Pattern and Traffic Flows

We have used the "random waypoint" model [11]. In this model, each node is at a random point at the start of the simulation and after *pause time* seconds selects a random destination and moves to that destination at 20 m/s for a period of time uniformly distributed between 5 and 11 seconds. Upon reaching the destination, the node pauses again for *pause time* seconds, chooses another destination, and proceeds there. We used the speed of 20m/s as it has been used in simulations in previous work [11], [12]. 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 $p > 0$. We use widely varying pause times: 0, 15, 30, 45, 60, 120, 300, 600 and 900 seconds. High mobility scenarios are tested with higher granularity than the low mobility scenarios with the basic aim of finding how the routing protocols impart extra overhead under rapidly changing network conditions than under almost static network conditions.

We have 20 nodes moving over a flat space of dimensions (5.7miles X 7.7miles) and initially randomly distributed with a density of approximately 0.3 node per square mile. During the simulations most of the routes consist of 2-4 hops, with each node having an average connectivity to about 30% of the total nodes. We have tested scenarios with the number of traffic flows as 4, 16, and 32. In the simulation with 4 flows, we have 4 sources with one destination each, while in the simulations greater than 4 flows, we have 8 sources with each source having 2 and 4 destinations. The varying number of flows are used as an attempt to capture most of the realistic scenarios for ad-hoc networks. It has been shown that depending on the scenarios, the number of flows in the network can widely vary[12]. Each flow is a peer-to-peer constant bit rate (CBR) flow and the data packet size is kept constant at 64 bytes. The flows start randomly from 20 to 250 seconds and each flow continues for 200 seconds and after the termination of the flow, within 1 sec, the source randomly chooses another destination and starts another flow, which again lasts for another 200 seconds. Hence throughout the simulation, at any point of time after all flows have started, the number of flows remains constant. In previous studies ([11], [8]), the flows start during the initial part of the simulation and stay throughout the simulation, which almost divides the entire simulation time into two separate phases : path discovery during the initial stage and path maintenance at the later stages. In order to simulate most realistic scenarios where flows can start and end randomly, we have used the traffic model mentioned above. The total load on the network is kept constant at 31 data packets/second. We have kept the load small with the aim of not creating congestion with our data packets, as our idea is to test how routing protocols react to changes in the network topology while delivering packets to their destinations. When the number of flows increases, the data rate of the flows decreases to achieve constant workload on the network.

B. Metrics used

In comparing the two protocols, we use the following performance metrics:

- *Packet delivery ratio*: The ratio between the number of packets sent out by the sender application and the number of packets correctly received by the corresponding peer application.
- *Control Packet Overhead*: The total number of control packets sent out during the simulation. Each broadcast packet is counted as a single packet. Low control packet overhead is desirable in low-bandwidth wireless environments.
- *Average Hop Count*: The average number of hops the data packet took from the sender to the receiver during one run of simulation. Shorter hop count implies that the routing protocol is using shorter paths to the destinations, thereby utilizing more efficiently the network resources.
- *Average end-to-end Delay*: The end-to-end delay implies the delay a packet suffers between leaving the sender application and arriving at the receiver application. This includes delays caused by route discovery latency at SOAR, delay due to waiting at IP and MAC layers and propagation delays.

C. Results

TABLE I
LINK CONNECTIVITY CHANGES DURING 900 SECS OF SIMULATION FOR A 20 NODE NETWORK

Pause Time	Connectivity Changes
0	695
15	257
30	170
45	140
60	126
120	102
300	80
600	72
900	72

Table I shows the number of link connectivity changes that occur during different host mobility patterns. Every time a link goes up or down, it is treated as one link connectivity change. So all the changes (72) that happen during *pause time* 900 secs is due to formation of initial topology while any other changes in link connectivity that occur for lower values of *pause time*, is due to host mobility.

TABLE II
CONSTANTS USED IN DSR SIMULATION

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

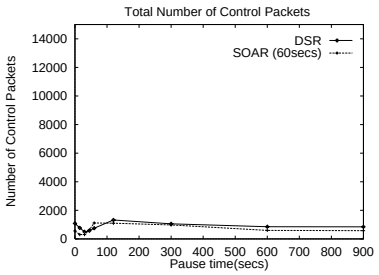
Tables II and III show the constants used for DSR and SOAR during the simulation. For SOAR, the value of *minimum update time* is chosen such that a sufficient amount of time is given for the network to recover from the wrong information without introducing more $update$ packets. This value should not be kept so high that during loss of $update$ packets, recovery takes a long time. The *update timeout* has been kept to a smaller value than *minimum update time*, such that a node sends an $update$ quickly before *minimum update time* expires. The *refreshing_time* value has been chosen to make a trade off between overhead of flood-search messages and maintaining up-to-date paths to all destinations. This value has been found to be most suitable for this scenario but in general, this value may not be the best under all circumstances. The values of other constants are chosen to match those used in the literature ([11], [12]).

TABLE III
CONSTANTS USED IN SOAR SIMULATION

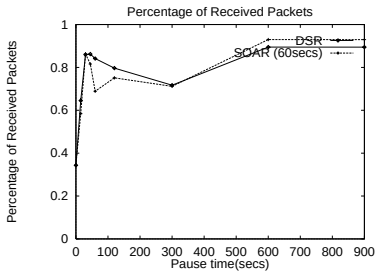
query send timeout (exponentially backed off) (ms)	500
Zero query send timeout (ms)	30
Time to hold packets awaiting routes (s)	30
Max number of pending packets	50
query receive timeout (s)	4.5
Update Timeout (s)	2
Minimum Update Time (s)	3
MAX_HOPS	17
refreshing_time (s)	60

Figures 5, 6, 7 give a comparative performance of SOAR and DSR under three scenarios, where the number of flows is 4, 16 and 32, respectively. We see that the highest number of packets are delivered as the networks become less mobile. This is expected, because all packets meant for a neighbor, are dropped after link failures and link failures occur less frequently when the nodes are less mobile. A considerable performance improvement can be achieved if the MAC layer, while communicating with SOAR, can reschedule packets along some alternate links. In our simulations we found that large number of packets got dropped at the routing layer when the network was getting partitioned, due to the unavailability of routes to destinations. We also see that there is an increase in the number of routing packets for both SOAR and DSR when the number of flows increases (Figs. 5.a, 6.a, 7.a). This is expected in on-demand routing protocols, because the number of routes that a node is required to maintain increases with the number of flows.

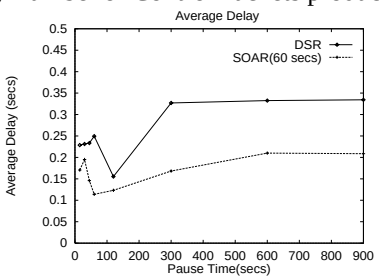
We observe from Figure 5 that the number of control packets exchanged in SOAR is almost similar to DSR when the number of flows is very small (4), compared to the number of nodes in the network. However as the number of flows increases, SOAR scales better than DSR (Fig.6, 7). This is because each node in DSR is required to communicate with more nodes when the number of flows increases, and, unlike DSR, SOAR utilizes the redundancy in the *minimal* source trees exchanged to reduce the number of flood search messages. Flood search messages are expensive as the entire network is flooded for routes in many situations and each $query$ can produce multiple replies. As SOAR



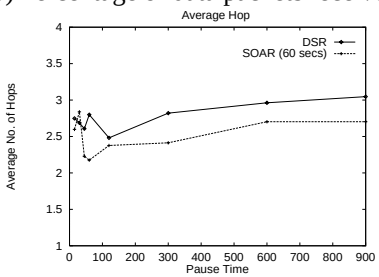
(a) Number of Control Packets produced



(b) Percentage of data packets received

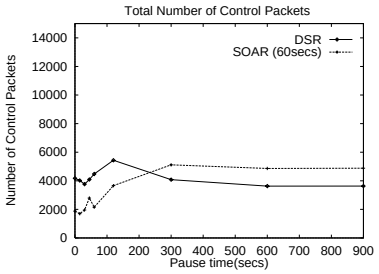


(c) Average Delay experienced by data packets

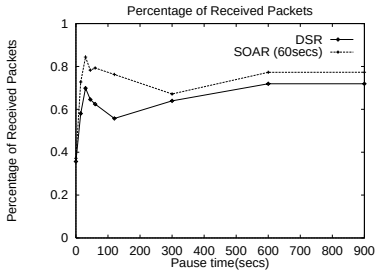


(d) Average Number of Hops traversed

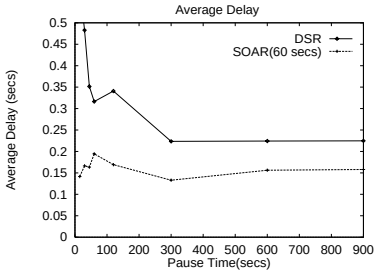
Fig. 5. Simulation results for the 20node Network with 4 flows



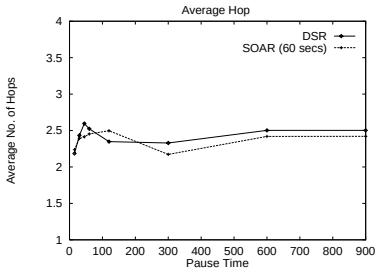
(a) Number of Control Packets produced



(b) Percentage of data packets received



(c) Average Delay experienced by data packets



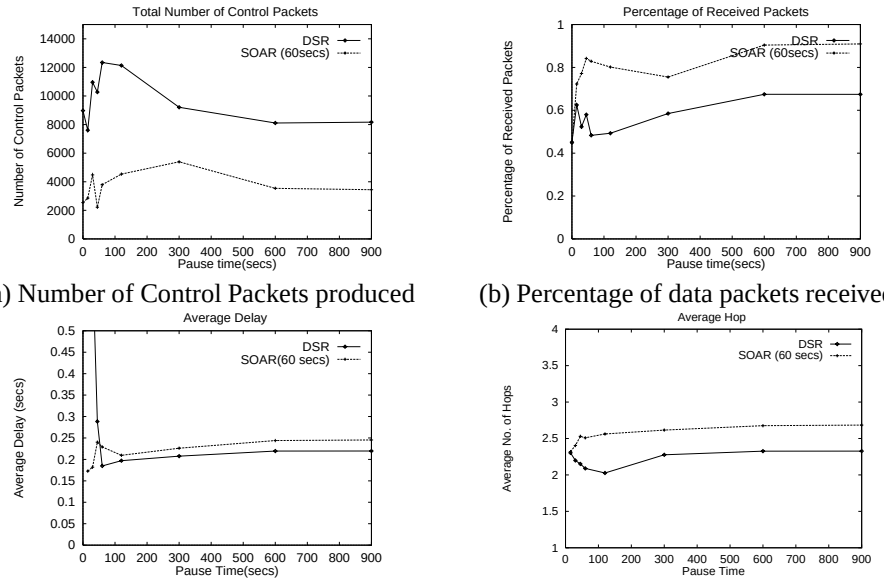
(d) Average Number of Hops traversed

Fig. 6. Simulation results for the 20node Network with 16 flows

exchanges control packets of bigger size, total byte usage has been found to be 2-3 times more in SOAR compared to DSR. (A reduction in the size of control packets of SOAR can be achieved by representing the advertised minimal source trees in the form of a list of paths). However, the cost for gaining access to the channel is constant with MAC protocols ([13],[14]) similar to IEEE802.11 and looking at byte overhead is not realistic. If the MAC layer allowed for transmission of reliable updates with no retransmission overhead, ([15], [16]), then only incremental changes to the *minimal* source tree can be exchanged, thereby reducing the control packet sizes of SOAR.

We observe from Figures 5.b, 6.b, and 7.b that the percentage of received packets is almost the same when the number of flows is 4. However when the number of flows increases, SOAR delivers many more data packets than DSR. One of the reasons for this difference is that DSR drops more packets due to the unavailability of buffer space. This is because, unlike SOAR, when the number of flows is high, DSR sends more $queries$ while more data packets sit in the buffer waiting for their routes to be discovered.

The average number of hops traversed in SOAR (Figs 5.d, 6.d, 7.d) is less than or equal to DSR in most of the situations. Part of the reason for the differences is SOAR, while transferring information about some path cost increase, can indicate shortening of distance for certain other nodes, which can belong to the same branch as the node whose distance has increased (as it happens in DSR) but also in some other branches of the tree. In [1] a method has been suggested to ensure the use of shortest paths in



(a) Number of Control Packets produced (b) Percentage of data packets received

(c) Average Delay experienced by data packets (d) Average Number of Hops traversed

Fig. 7. Simulation results for the 20node Network with 32 flows

DSR, but that needs the router to use *promiscuous* modes.

The average delay experienced by the data packets is higher for DSR than in SOAR when the number of flows is 4 and 16. This is because DSR waits more in the data buffer while the paths are discovered. However when the number of flows is 32, DSR delivers less data packets than SOAR, and as those data packets are mainly for nearer destinations, delay suffered by data packets in SOAR is higher.

V. Conclusions

We have presented SOAR, the first link state on-demand routing protocol that is suitable for ad-hoc networks. The simulation experiments we carried out show that SOAR incurs much less overhead than DSR under all scenarios, ranging from high mobility to low mobility. Given that DSR has been shown to require less control traffic than AODV and other protocols, we conjecture that SOAR is one of the most bandwidth-efficient routing protocols for ad hoc networks. SOAR achieves this by communicating to its neighbors the link states of only those links that belong to the paths it chooses to advertise for reaching destinations with which it has active flows, by allowing paths to deviate from optimal routes, and by sending updates only when the path increases, while not creating permanent loops.

References

[1] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad-Hoc Wireless Networks," *Mobile Computing, Kluwer Academic publishers*, 1996.

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

[3] 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.

[4] J. Raju and J.J. Garcia-Luna-Aceves, "ROAM: A New Approach to On-Demand Loop-Free Multipath Routing," in *Proc. IEEE IC3N*, Boston, Massachusetts, October 1999.

[5] P. Jacquet, P. Muhlethaler, and A. Qayyum, "Optimized Link State Routing Protocol," in *draft-ietf-manet-olsr-03.txt*, ftp://ftp.isi.edu/internet-drafts/draft-ietf-manet-olsr-03.txt, November 2000.

[6] Y. Hu and D. Johnson, "Caching Strategies in On-Demand Routing Protocols for Wireless Ad Hoc Networks ," in *ACM MOBICOM, 2000*, Boston, Massachusetts, August 2000.

[7] J.J.Garcia Luna-Aceves and M. Spohn, "Source-Tree Routing in Wireless Networks," in *Proc. of IEEE ICNP99*, November 1999.

[8] 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.

[9] 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.

[10] K. Fall and K. 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.

[11] 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.

[12] P. Johansson, T.Larsson, N. Hedman, B. Mielczarek, and M.Dagermark, "Scenario Based Performance Analysis of Routing Protocols for Mobile Ad-Hoc Networks," in *Proc. ACM Mobicom'99*, Seattle, Washington, August 1999.

[13] 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.

[14] 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.

[15] Z. Tang and J.J. Garcia-Luna-Aceves, "Collision-Avoidance Transmission Scheduling for Ad-Hoc Networks," in *Proc. IEEE ICC'2000*, June 2000.

[16] C. Zhu and S. Corson, "A Five-Phase Reservation Protocol (FPRP) for Mobile Ad Hoc Networks," in *Proc. IEEE Infocom, 98*, March,1998.