DART: Dynamic Address RouTing for Scalable Ad Hoc and Mesh Networks

Jakob Eriksson
jeriksson@cs.ucr.edu
Michalis Faloutsos
michalis@cs.ucr.edu
Srikanth Krishnamurthy
krish@cs.ucr.edu
University of California, Riverside

***Abstract*— It is well known that the current ad hoc protocol suites do not scale to work efficiently in networks of more than a few hundred nodes. Most current ad hoc routing architectures use flat static addressing and thus, need to keep track of each node individually, creating a massive overhead problem as the network grows. Could dynamic addressing alleviate this problem? In this paper, we argue that the use of dynamic addressing can enable scalable routing in ad hoc networks. We provide an initial design of a routing layer based on dynamic addressing, and evaluate its performance. Each node has a unique permanent identifier and a transient routing address, which indicates its location in the network at any given time. The main challenge is dynamic address allocation in the face of node mobility. We propose mechanisms to implement dynamic addressing efficiently. Our initial evaluation suggests that dynamic addressing is a promising approach for achieving scalable routing in large ad hoc and mesh networks.** [1] [2]

I. Introduction

How large can an ad hoc network be? Scalability is a critical requirement if we want these networking technologies to reach their full potential. Ad hoc networking technology has advanced tremendously over the last ten years but it has yet to become a widely deployed technology. This is similar to the early stages of the Internet, where very few could predict its explosive growth. A difference is that in the Internet, scalability was, from the very beginning, a design constraint. Ad hoc networks research seems to have downplayed the importance of scalability. In fact, current ad hoc architectures do not scale well beyond a few hundred nodes.

The easy-to-use, self-organizing nature of ad hoc networks make them attractive to a diverse set of applications. Today, these are usually limited to smaller deployments, but if we can solve the scalability problem, and provide support for heterogeneous means of connectivity, including directional antennas, communication lasers, even satellites and wires, ad hoc and mesh-style networking is likely to see adoption in very large networks as well. Large-scale events such as disaster relief or rescue efforts are highly dependent on effective communication capabilities. Such efforts could benefit tremendously from the use of self-organizing networks to improve the communications and monitoring capabilities available. Other interesting candidate scenarios are community networks in dense residential areas, large scale, long-range networks in developing regions, and others, where no central administrator exists, or where administration would prove too costly. Already, non-military technology and applications seem to point towards future networks with: a) ad hoc pockets of connectivity [2], b) consumer-owned networks [3] [4] [5], and c) sensor-net technologies [6]. All of these applications will place increased scalability demands on self-organizing routing protocols.

The current routing protocols and architectures work well only up to a few hundred nodes. Most current research in ad hoc networks focus more on performance and power-consumption related issues in relatively small networks, and less on scalability. We believe the main reason behind the lack of scalability is that these protocols rely on flat and static addressing. With scalability as a partial goal, some efforts have been made in the direction of hierarchical routing and clustering [7] [8] [9]. These approaches do hold promise, but they do not seem to be actively pursued. It appears to us as if these protocols would work well in scenarios with group mobility [10], which is also a common assumption among cluster based routing protocols.

We examine whether dynamic addressing is a feasible way to achieve scalable ad hoc routing. By ”scalable” we mean thousands up to millions of nodes in an ad hoc or mesh network. With dynamic addressing, nodes change addresses as they move, so that their addresses have a topological meaning. Dynamic addressing simplifies routing but introduces two new problems: address allocation, and address lookup.

As a guideline, we identify a set of properties that a scalable and efficient solution must have:

- *Localization of overhead:* a local change should affect only the immediate neighborhood, thus limiting the overall overhead incurred due to the change.
- *Lightweight, decentralized protocols:* we would like to avoid concentrating responsibility at any individual node, and we want to keep the necessary state to be maintained at each node as small as possible.
- *Zero-configuration:* we want to completely remove the

[1]This work was supported by the NSF CAREER grant ANIR 9985195, DARPA award NMS N660001-00-1-8936, NSF grant IIS-0208950, TCS Inc., DIMI matching fund DIM00-10071, DARPA award FTN F30602-01-2-0535.

[2]This is an extended version of our earlier INFOCOM paper [1].

need for manual configuration beyond what can be done at the time of manufacture.
- *Minimal restrictions on hardware:* Omnidirectional link-layers do not scale to large networks. Localization technologies, such as GPS, may limit protocol applicability.

In this paper, we examine how dynamic addressing can be a building block toward a scalable ad hoc routing architecture. We present a complete design including address allocation, routing and address lookup mechanisms, and provide thorough evaluation results for the address allocation and routing components. Our earlier work [11] describes how address lookup can be efficiently handled in some more detail.

First, we develop a dynamic addressing scheme, which has the necessary properties mentioned above. Our scheme separates node identity from node address, and uses the address to indicate the node's current location in the network. Second, we study the performance of a new routing protocol, based on dynamic addressing, through analysis and simulations.

In more detail, our work leads to the following results.

- Our address allocation scheme uses the address space efficiently on topologies of randomly and uniformly distributed nodes, empirically resulting in average routing table sizes of less than $2\log_2 n$ where n is the number of nodes in the network.
- We compare our protocol to reactive protocols (AODV, DSR) and a proactive protocol (DSDV). Our results suggest that dynamic addressing and proactive routing together provide significant scalability advantages. Based on simulation results, even an unoptimized version of DART significantly outperforms other routing protocols based on static addresses, in large and actively used networks.

Our work in perspective. We describe a new approach to routing in ad hoc networks, and compare it to the current routing architectures. However, the goal is to show the potential of this approach and not to provide an optimized protocol. We believe that the *address equals identity* assumption used in current ad hoc routing protocols is most likely inherited from the wireline world, which is much more static and is explicitly managed by specialist system administrators[3]. Although much work remains to be done, we believe that the dynamic addressing approach is a viable strategy for scalable routing in ad hoc networks.

The rest of the paper is structured as follows. In section II we describe the related work, in section III we give a high-level overview of all aspects of our proposed routing protocol. In section IV, we describe the routing operation in some detail, and in section V we show how the current routing address of a node is found. In section VI we go into more detail on the specifics of our address allocation scheme. Section VII describes how routing tables are computed and maintained. Section VIII reports some of our simulation results, and section IX gives a brief analysis of the protocol and the relative overhead of reactive and proactive routing protocols. In section X we discuss optimizations and other issues, and section XI concludes the paper.

[3]Even in the wireline world, mobility has started to challenge this assumption, creating a need for workaround solutions such as mobile IP.

II. Related Work

In most common IP-based ad hoc routing protocols [12] [13] [14], addresses are used as pure identifiers. Without any structure in the address space, there are two choices: either keep routing entries for every node in the network, or resort to flooding route requests throughout the network upon connection setup. Neither of these alternatives scale well. Other protocols [15] [16] use geographic location information to assist in the routing, and thereby try to achieve scalability. However, this approach can be severely limiting as location information is not always available and can be misleading in, among others, non-planar networks. For a survey of ad hoc routing, see [17].

In the Zone Routing Protocol (ZRP) [18] and Fisheye State Routing (FSR) [19], nodes are treated differently depending on their distance from the destination. In FSR, link updates are propagated more slowly the further away they travel from their origin, with the motivation that changes far away are unlikely to affect local routing decisions. ZRP is a hybrid reactive/proactive protocol, where a technique called bordercasting is used to limit the damaging effects of global broadcasts.

Some work has been done on using clustering in ad hoc networks. In multilevel-clustering approaches such as Landmark [20], LANMAR [9], L+ [21], MMWN [7] and Hierarchical State Routing (HSR) [8], certain nodes are elected as cluster heads (also called Landmarks). These cluster heads in turn select higher level cluster heads, up to some desired level. A node's address is defined as a sequence of cluster head identifiers, one per level, allowing the size of routing tables to be logarithmic in the size of the network, but easily resulting in long hierarchical addresses. In HSR, for example, the hierarchical address is a sequence of MAC adresses, each of which is 6 bytes long.

A problem with having explicit cluster heads is that routing through cluster heads creates traffic bottlenecks. In Landmark, LANMAR and L+, this is partially solved by allowing nearby nodes route packets instead of the cluster head, if they know a route to the destination. All of the above schemes have explicit cluster heads, and all addresses are therefore relative to these, and are likely to have to change if a cluster head moves away. This reliance on cluster head nodes makes the above schemes best suited to scenarios involving group mobility, such as troop movements.

Area Routing, as described by Kleinrock and Kamoun in [22], is the method most similar to the one used in today's Internet. Here, nodes that are close to each other in the network topology have similar addresses, without any explicit hierarchy of nodes. Our work is, as far as we know, the first attempt to use this type of addressing in ad hoc and mesh networks.

Tribe [23] is similar to DART at a high level, in that it uses a two phase process for routing: first address lookup, and then routing to the address discovered. However, the tree-based routing strategy used in Tribe bears little or no resemblance to the area based approach in DART. Tree-based

routing may under many circumstances suffer from severe traffic concentration at nodes high up in the tree, and a high sensitivity to node failure.

NoGeo [24] embeds the network graph in a virtual 2-dimensional coordinate space, and uses geographical forwarding techniques for routing. The approach is interesting, in that it achieves the O(1) complexity of geographical routing, but does not require actual geographical coordinates. However, the scheme will only work on certain types of graphs (typically unit-disk like graphs). In addition, NoGeo has not been evaluated for networks with more than very low rates of mobility.

Another coordinate based routing scheme, GEM [25], embeds a sensor network graph in a polar coordinate system. Starting at the sink, nodes are assigned ranges based on the distance to the sink, and angles, such that greedy routing is possible. GEM does tree-based routing, resulting in a heavy concentration of traffic around the root node, and was designed for sensor networks. It has not not evaluated for mobile ad hoc or mesh networks.

Dynamic Address Routing in relation to peer-to-peer DHT's. We have received many inquiries as to the relationship between peer-to-peer distributed hashtables, such as Chord [26], and our work. First, let us point out that our proposed node lookup table is in fact a special purpose distributed hashtable, similar in many ways to what has already been done in peer-to-peer networks. To clearly demonstrate that Dynamic Address Routing is, with the exception of the node lookup table, only superficially related to peer-to-peer DHT's, we will now point out a few important differences.

First of all, in peer-to-peer DHT's, there is an assumption of any-to-any connectivity. That is, any node can reach any other node by using an underlying routing mesh. In our work, we are building the routing mesh and can only rely on immediate neighbors for communication. In essence, a node in a DHT can locate itself at any point in the key space and the DHT will still be consistent, although perhaps somewhat disadvantaged performance-wise. If a node in our routing protocol does not pick its address carefully, routing will not work, because there is no underlying routing layer there to save us.

Second, DHT's are an application layer overlay network, with the consequence that a single physical link could be traversed several times when routing a packet through the overlay. In our work, we work directly with the physical links, and every packet traverses any given link at most once.

Third, in a DHT, one expects to see packets delivered in at most $O(\log N)$ "virtual hops". In network layer routing, the number of hops depends almost entirely on the underlying topology, and thus such bounds cannot possibly be stated.

III. Overview and Definitions

In this section, we present our main ideas for dynamic address allocation and define various terms that we use. We also sketch a network architecture, which could utilize the new addressing scheme effectively. In fact, dynamic routing and addressing form the basis for a novel networking layer, which we describe in some detail in our earlier work [11].

In our approach, we separate the routing address and the identity of a node. The **routing address** of a node is dynamic and changes with node movement to reflect the node's location in the network topology. The **identifier** is a globally unique number that stays the same throughout the lifetime of the node. For ease of presentation, we can assume for now that each node has a single identifier[4].

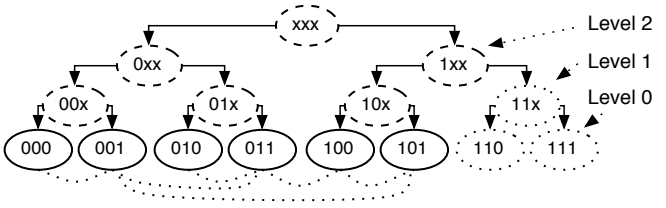

Fig. 1. The address tree of a 3-bit binary address space. Leaves represent actual addresses, whereas inner nodes represent groups of addresses with a common prefix.

We distinguish three major functions. First, **address allocation** maintains one routing address per network interface, in such a way that the address indicates the node's relative network location. Second, **routing** delivers packets from a node to a given routing address. Third, **node lookup** is a distributed lookup table mapping every node identifier to its current network address. We defer all details of the address allocation process to section VI.

Let us first describe how we want things to work from an operational point of view. When a node joins the network, it listens to the periodic routing updates of its neighboring nodes, and uses these to identify an unoccupied address. We will describe how this is done later. The joining node registers its unique identifier and the newly obtained address in the distributed node lookup table. Due to mobility, the address may subsequently be changed and then the lookup table needs to be updated. When a node wants to send packets to a node known only by its identifier, it will use the lookup table to find its current address. Once the destination address is known the routing function takes care of the communication. The routing function should make use of the topological meaning that our routing addresses possess.

We start by presenting two views of the network that we use to describe our approach: a) the address tree, and b) the network topology.

The Address Tree. In this abstraction, we visualize the network from the address space point of view. Addresses are l bit binary numbers, $a_{l-1}, \ldots, a_0$. The address space can be thought of as a binary **address tree** of $l+1$ levels, as shown in figure 1. The leaves of the address tree represent actual node addresses; each inner node represents an **address subtree**: a range of addresses with a common prefix. For presentation purposes, nodes are sorted in increasing address order, from left to right. We stress that the links in the tree do not correspond to physical links in the network topology. The actual physical links are represented by dotted lines connecting leaves in figure 1.

[4]We currently use IP addresses as identifiers. Thus, the transport and application layers do not need to change, and the routing address is only seen at the network layer. There exist situations where we may want to map a node to more than one identifier, for example in supporting multicasting [11].

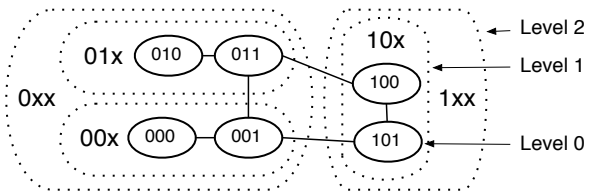

Fig. 2. A network topology with node addresses assigned. Dotted enclosures correspond to subtrees in the address tree.

The Network Topology. This view represents the connectivity between nodes. In figure 2, the network from figure 1 is presented as a set of nodes and the physical connections between them. Each solid line is an actual physical connection, wired or wireless, and the sets of nodes from each subtree of the address tree are enclosed with dotted lines.

Note that the set of nodes from any subtree in figure 1 induces a connected subgraph in the network topology in figure 2. This is not a coincidence, but a crucial property of our dynamic addressing approach. Intuitively, nodes that are close to each other in the address space should be relatively close in the network topology. More formally, we can state the following constraint.

Prefix Subgraph Constraint: The set of nodes that share a given address prefix form a connected subgraph in the network topology.

This constraint is fundamental to the scalability of our approach. Intuitively, this constraint helps us map the virtual hierarchy of the address space onto the network topology. The longer the shared address prefix between two nodes, the shorter the expected distance in the network topology.

Finally, let us define two new terms that will facilitate the discussion in the following sections.

A **Level-k subtree** of the address tree is defined by an address prefix of $(l-k)$ bits, as shown in figure 1. For example, a Level-0 subtree is a single address or one leaf node in the address tree. A Level-1 subtree has a $(l-1)$-bit prefix and can contain up to two leaf nodes. In figure 1, [0xx] is a Level-2 subtree containing addresses [000] through [011]. Note that every Level-k subtree consists of exactly two Level-$(k-1)$ subtrees.

We define the term **Level-k sibling** of a given address to be the sibling[5] of the Level-k subtree to which a given address belongs. By drawing entire sibling subtrees as triangles, we can create abstracted views of the address tree, as shown in figure 3. Here, we show the siblings of all levels for the address [100] as triangles: the Level-0 sibling is [101], Level-1 is [11x], and the Level-2 sibling is [0xx]. Note that *each address has exactly one Level-k sibling, and thus at most l siblings in total.*

Finally, we define the **identifier of a subtree** to be the min of the identifiers of all nodes that have addresses from that subtree. In cases where the prefix subgraph constraint is temporarily violated, two disconnected instances of the address subtree exist in the network. In this case, each instance is uniquely identified by the min of the subset of identifiers that belongs to its connected subgraph. As we describe in more detail in section VI, only the instance with the lowest identifier is a valid part of the network.

[5]We define siblings as subtrees, or leaves, that have the same immediate parent.

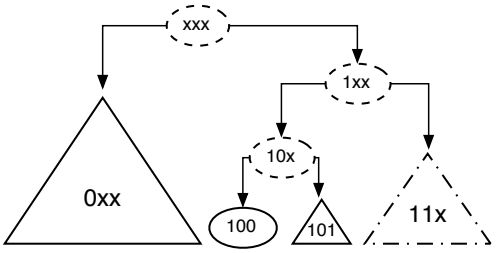

Fig. 3. Routing entries corresponding to figure 2. Node 100 has entries for subtrees 0xx, 11x (null entry) and 101.

A. Other important characteristics.

Our addressing and routing schemes have several attractive properties. First, they can work with omnidirectional and directional antennas as well as wires. Second, we do not need to assume the existence of central servers or any other infrastructure, nor do we need to assume any geographical location information, such as GPS coordinates. However, if infrastructure and wires exist, they can, and will, be used to improve the performance. Third, we make no assumptions about mobility patterns, although high mobility will certainly lead to increased overhead, and decreased throughput. Finally, since our approach was designed primarily for scalability, we do not need to limit the size of the network; most popular ad hoc routing protocols today implicitly impose network size restrictions.

IV. Routing

In this work, we use a form of proactive distance-vector routing, made scalable due to the hierarchical nature of the address space. Although we have chosen to use distance vector routing, we would like to point out that many of the advantages of dynamic addressing can be utilized by a link-state protocol as well.

Listing 1 Routing state kept by each node. neighbor_updates contains last-received routing update, and expiry time.

```
struct NodeState
{
    address_bit[ADDR_SIZE]
    /* routing table */
    nexthop[ADDR_SIZE]
    cost[ADDR_SIZE]
    id[ADDR_SIZE]
    route_log[ADDR_SIZE][ADDR_SIZE]
    neighbor_updates[]
}
```

Each node keeps some routing state, as specified in Listing 1. Routing state about a node's Level-i sibling is stored at position i in each of the respective arrays.

Intuitively, the routing state for a sibling contains the information necessary to maintain a route toward a node (any node) in that subtree. The $address$ field contains the current address of the node, and bit i of the address is referred to as $address[i]$, where $i = 0$ for the least significant bit of the address. Arrays $nexthop$ and $cost$ are self-explanatory. The id array contains the identifier of the subtree in question. As described earlier, the identifier of a subtree is equal to the lowest out of all the identifiers of the nodes that constitute that subtree.

Finally, $route\_log[i]$ contains the log of the current route to the sibling at level i, where bit b of log i is referenced by the syntax $route\_log[i][b]$. The use of route logs for loop avoidance is discussed further below. [6]

Packet forwarding under DART is a matter of looking up the next hop in the routing table. In our example shown in figures 1-3, node [100] has routing entries for sibling subtrees [0xx], [11x] and [101]. To route a packet to address [000], node [100] first determines the (sibling) subtree to which the destination address belongs ([0xx]). Practically, this is done by identifying the most significant bit that differs between the current node's address and the destination's address. In this case, the most significant differing bit is bit number 2. The node then looks up the entry with index two in the nexthop table, and then sends the packet there. In our example, this is the neighbor with address [011]. The process is repeated until the packet has reached the given destination address.

The hierarchical technique of only keeping track of sibling subtrees rather than complete addresses has three immediate benefits. One, the amount of routing state kept at each node is drastically reduced. Two, the size of the routing updates is similarly reduced. Three, it provides an efficient routing abstraction such that routing entries for distant nodes can remain valid despite local topology changes in the vicinity of these nodes.

A. Loop Avoidance

DART uses a novel scheme for detecting and avoiding routing loops, which leverages the hierarchical nature of the address space to improve scalability.

First, let us review the general concept of loop avoidance, to lay the foundation for the discussion of our loop avoidance scheme. In an abstract sense, routing loop avoidance is about remembering what nodes a route update has traversed, and making sure that these nodes do not accept route updates that they have already seen. As long as this requirement is satisfied, routing loops cannot occur.

A simple way of implementing this is to concatenate a list of all visited nodes in the routing update, and to have nodes check this list before accepting an update. However, this approach has a scalability problem, in that routing updates will quickly grow to unwieldy sizes.

Instead, DART makes use of the structured address space to create a new kind of loop avoidance scheme. In order to preserve scalability, we generalize the loop freedom rule above. For each subtree, once a routing entry has left the subtree, it is not allowed to re-enter. This effectively prevents loops, and can be implemented in a highly scalable manner: A bit array of $ADDR\_SIZE$ bits is kept together with the routing update. Bit k of the route log indicates whether the route update arrived at the current node via the level-k sibling. Loop-free operation of the protocol is ensured by blocking each routing entry from entering a level-k sibling if bit k in its route log is set to 1. For more details, see section VII.

V. Node Lookup

The missing link is: how do we find the current address of a node, if we know its identifier? We propose to use a distributed node lookup table, which maps each identifier to an address, similar to what we proposed in [11]. Here, we assume that all nodes take part in the lookup table, each storing a few[7] $<$*identifier, address*$>$ entries. However, this node lookup scheme is only one possibility among many, and more work is needed to determine the best lookup scheme to deploy.

For our proposed distributed lookup table, the question now becomes: which node stores a given $<$*identifier, address*$>$ entry? Let us call this node the **anchor node** of the identifier. The solution is simple yet elegant, and reminiscent of consistent hashing [27].

We use a globally, and a priori, known hash function that takes an identifier as argument and returns an address where the entry can be found. If there exists a node that occupies this address, then that node is the **anchor node**. If there is no node with that address, then the node with the least edit distance between its own address and the destination address, is the **anchor node**.

To route packets to an **anchor node**, we use a slightly modified routing algorithm: If no route can be found to a sibling subtree indicated by a bit in the address, that bit of the address is ignored, and the packet is routed to the subtree indicated by the next (less significant) bit. When the last bit has been processed, the packet has reached its destination. This method effectively finds the node with the address minimum edit distance to the address returned by the hash function.

For example, using figure 3 for reference, let's assume a node with identifier ID_1 has a current routing address of [010]. This node will periodically send an updated entry to the lookup table, namely $<ID_1, 010>$. To figure out where to send the entry, the node uses the hash function to calculate an address, like so: $hash(ID_1)$. If the returned address is [100], the packet will simply be routed to the node with that address. However, if the returned address was instead [111], the packet could not be routed to the node with address [111] because there is no such node. In such a situation, the packet gets automatically routed to the node with the most similar address, which in this case would be [101].

[6] We have deliberately left typing to the implementation. However, thinking of id as an array of large integers, say 32 or 48 bits, may be instructive. The type of the $cost$ array depends entirely on the cost metric used. In this work, we use a simple hop count metric.

[7] We expect to see on average $O(logN)$ entries per node assuming a balanced address tree and uniformly distributed identifiers.

A. Improved Scalability.

We would like to stress that all node lookup operations use unicast only: no broadcasting or flooding is required. This maintains the advantage of proactive and distance vector based protocols over on-demand protocols: the routing overhead is independent of how many connections are active. When compared with other distance vector protocols, our scheme provides improved scalability by drastically reducing the size of the routing tables, as we described earlier. In addition, updates due to a topology change are in most cases contained within a lower level subtree and do not affect distant nodes. This is efficient in terms of routing overhead. To further improve the performance of our node lookup operations, we envision using the locality optimization technique described in [11]. Here, each lookup entry is stored in several locations, at increasing distance from the node in question. By starting with a small, local lookup and gradually going to further away locations, we can avoid sending lookup requests across long distances to find a node that is nearby.

B. Coping with Temporary Route Failures

On occasion, due to link or node failure, a node will not have a completely accurate routing table. This could potentially lead to lookup packets, both updates and requests, terminating at the wrong node. The end result of this is that requests cannot be promptly served. In an effort to reduce the effect of such intermittent errors, a node can periodically check the lookup entries it stores, to see if a route to a more suitable host has been found. If this should be the case, the entry is forwarded in the direction of this more suitable host. Requests are handled in a similar manner: if the request could not be answered with an address, it is kept in a buffer awaiting either the arrival of the requested information, or the appearance of a route to a node which more closely matches the key requested.

This way, even if a request packet arrives at the **anchor node** before the update has reached it, the request will be buffered and served as soon as the update information is available.

C. Practical Considerations

Due to the possibility of network partitioning and node failure, it is necessary to have some sort of redundancy mechanism built-in. We have opted for a method of periodic refresh, where every node periodically sends its information to its **anchor node**. By doing so, the node ensures that if its anchor node should become unavailable, the lookup information will be available once again within one refresh period. Similarly, without a mechanism of expiry, outdated information may linger even after a node has left the network. Therefore, we set all lookup table entries to expire automatically after a period twice as long as the periodic refresh interval.

VI. Dynamic Address Allocation

To assess the feasibility of dynamic addressing, we develop a suite of protocols that implement such an approach. Our work effectively solves the main algorithmic problems, and forms a stable framework for further dynamic addressing research. Although the design has not yet been optimized for maximum throughput, its scalability properties and predictable performance show promise (see section VIII).

Listing 2 SelectAddress()

```
    neighbor ← BestNeighbor()
    for bit from InsertionPoint(neighbor) to 0 do
      address = neighbor.address
      address[bit] != neighbor.address[bit]
 5:   if ValidateAddress(address) == valid then
        return address
      else
        neighbor.ids[bit] = OCCUPIED;
      end if
10: end for
```

When a node joins an existing network, it uses the periodic routing updates of its neighbors to identify and select an unoccupied and legitimate address, as specified in Listing 2.

Listing 3 BestNeighbor()

```
    best ← neighbors.first();
    for neighbor in neighbors do
      if InsertionPoint(neighbor) < InsertionPoint(best)
      then
        best ← neighbor;
 5:   end if
    end for
    return best
```

It starts out by selecting which neighbor to get an address from. As illustrated in Listing 3, the neighbor with the highest-level insertion point is selected as the best neighbor.

The insertion point is defined as the highest level for which no routing entry exists in a given neighbor's routing table. However, the fact that a routing entry happens to be unoccupied in one neighbor's routing table does not guarantee that it represents a valid address choice. We discuss how the validity of an address is verified in the next subsection.

The new node picks an address out of a possibly large set of available addresses. In our current implementation, we make nodes pick an address in the largest unoccupied address block. For example, in figure 3, a joining node connecting to the node with address [100] will pick an address in the [11x] subtree. There are several ways to choose among the available addresses, and we have presented only one such method. However, it has turned out that this method of address selection works well in simulation trials.

Under steady-state, and discounting concurrency, the presented address selection technique leads to a legitimate address allocation: the joining node is by definition connected to neighbor it got its new address from, and the new address is taken from one of the neighbors' empty sibling subtrees, so the prefix subgraph constraint is satisfied. We will discuss concurrency and mobility issues below.

Let us see an example of address allocation in action. Figure 4 illustrates the address allocation procedure for a 3-bit address

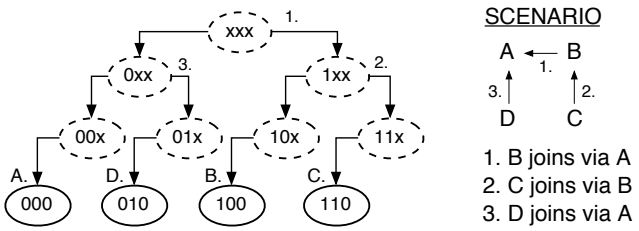

Fig. 4. Address tree for a small network topology. The numbers 1-3 show the order in which nodes were added to the network.

space. Node A starts out alone with address [000]. When node B joins the network, it observes that A has a null routing entry corresponding to the subtree [1xx], and picks the address [100]. Similarly when C joins the network by connecting to B, C picks the address [110]. Finally, when D joins via A, A's [1xx] routing entry is now occupied. However, the entry corresponding to sibling [01x] is still empty, and so, D takes the address [010].

A. Determining the Validity of an Address using Network and Subtree Identifiers

Node mobility, concurrency, and link instability all contribute to situations where the prefix subgraph constraint can be temporarily violated. To detect and address this condition, DART makes use of the unique identifier in each node.

At a high level, the goal is to detect the presence of the same address prefix in two disconnected parts of the network. The simplest case where this happens is when two networks, that were previously disconnected, are joined together by a new link.

In this scenario, it is highly likely that the prefix subgraph constraint is violated, since addresses in the two networks were previously chosen without knowledge of each other. Selecting new addresses is a well understood process, as described above. However, a technique for detecting the fact that the prefix subgraph constraint has been violated, and determining what to do to return to a valid address allocation, is required.

Listing 4 LevelId(level)

```
levelId ← self.identifier
for i from level to 0 do
  if self.id[i] < levelId then
    levelId ← self.id[i]
  end if
end for
```

To determine that the prefix subgraph constraint has been violated, we devise a way to compute a unique identifier for a connected set of nodes with a given address prefix. This is done by computing, for all the connected nodes with a given address prefix, the lowest node identifier among them. Listing 4 show's how this is done using information from the routing table. In situations where the prefix subgraph constraint is being violated, two or more routes will be announced to the same address prefix, but with different identifiers. When a node encounters such a situation, it simply ignores the route with the higher identifier. If a node encounters a route to its own subtree, but with a lower identifier than the one it has computed for its own subtree, it takes this to mean that it is violating the prefix subgraph constraint, and selects a new address.

Listing 5 ValidateAddress(address**)**

```
for nbr in self.neighbors do
  bit → DiffBit(nbr.address, self.address)
  if bit == -1 then
    // identical address: lower id prevails
5:  if nbr.identifier < self.identifier then
      return invalid
    end if
    // nbr has an entry to our subtree but with a
    // different id → constraint violated by us
10: else if nbr.address[bit] != self.address[bit] &&
          nbr.cost[bit] != ∞ &&
          nbr.ids[bit] < self.levelId(bit) then
    return invalid
  end if
end for
return valid
```

Listing 5 shows how the validity of an address is verified by checking it against the routing tables of all neighbors. For each neighbor, find the highest order bit where the current node's address differs from the neighbor's address. If the addresses are identical, then the address is invalid if the neighbor has a lower identifier (line 5). Otherwise, check the neighbor's announced routing table for the entry that should contain our subtree. If an entry exists, but contains a different identifier, nid, then we have detected an addressing conflict that needs to be resolved. The current address is invalid if the locally computed identifier is larger than nid.

Note that Listing 4 computes identifiers for each subtree that a node is a member of using only local routing table information.

Let us consider a small example, to illustrate how identifiers propagate through the network and are used to resolve addressing conflicts. For ease of presentation, assume there is a network where all nodes share the address prefix "0xxx". At time t, two nodes, a and b, with identifiers $id_a < id_b$ respectively, appear on opposite sides of the network. Following Listing 2, a and b will each select address "1000". This is a violation of the prefix subgraph constraint, but cannot yet be detected. After one period, the neighbors of a and b, have updated their routing tables to have top-level entry with identifiers id_a and id_b respectively. Still, no node has detected an inconsistency, as the presence of a and b is not yet known throughout the network. More periods pass, and eventually some node, c, will receive conflicting updates: some with id_a in the top level entry, some with id_b. Node c will put the smaller of id_a and id_b, say id_a in its top level entry, and broadcast its new routing table. In the next step, the nodes that sent c the entry with id_b, will receive c's update, and change its routing table correspondingly. Eventually, the id_a entry will reach all the way to node b, which will then determine that it

has an invalid address, and pick a new, unoccupied one.

Merging Networks Efficiently. DART handles the merging of two initially separate networks as part of normal operations. In a nutshell, the nodes in the network with the higher identifier join the other network one by one[8]. The lower-id network absorbs the other network slowly: the nodes at the border will first join the other network, and then their neighbors join them recursively.

Dealing with Split Networks. Here, we describe how we deal with network partitioning. Intuitively, each partition can keep its addresses, but one of the partitions will need to change its network identifier. In this situation, there are generally no constraint violations. This reduces to the case where the node with the lowest identifier leaves the network. Since the previous lowest identifier node is no longer part of the network, the routing update from the new lowest identifier node can propagate through the network until all nodes are aware of the new network identifier.

B. Balancing and Optimizing the Address Allocation

In future versions of our protocol, we will include techniques for optimizing the address allocation according to certain criteria. So far, our mechanisms aim only to maintain legitimate addresses, and they typically only need to respond to link breakage and link formation events. As described above, we currently greedily minimize the expected size of the resulting routing table at each node. However, we may want to reallocate addresses proactively to improve: a) the balancing of the address tree, and b) the length of the routed paths. Our current approach does not consider the path stretch caused by route aggregation and thus may not provide an optimal choice based on the resulting path lengths. It is worth mentioning that even without such optimizations, our scheme performs well.

VII. Populating and maintaining the Routing Table

While packet forwarding is a simple matter of looking up a next hop in a routing table, maintaining a consistent routing state does involve a moderate amount of sophistication. In addition to address allocation, loop detection and avoidance is crucial to correct protocol operation. In this section, we will discuss how the routing table is populated, and how routing loops are avoided.

Listing 6 Refresh()

```
    if ValidateAddress(address) != valid then
      address ← SelectAddress();
    end if
    reset(distance[], id[], route_log[][])
 5: for neighbor in neighbors do
      MergeRoutingTable(neighbor.update)
    end for
```

[8]Ideally, we would like to use the network size as a joining criterion in order to minimize the number of nodes that need to change addresses. Although we are investigating this option, the cost of determining the network size may not be worth the effort.

DART nodes use periodic routing updates to notify their neighbors of the current state of their routing table. If, within a constant number of update periods, a node does not hear an update from a neighbor, it is removed from the list of neighbors, and its last update discarded. Every period, each node executes Refresh(), the function described in Listing 6.

Listing 7 Routing Update Structure

```
struct RoutingUpdate {
    address_bit[ADDR_SIZE]
    cost[ADDR_SIZE]
    id[ADDR_SIZE]
    route_log[ADDR_SIZE][ADDR_SIZE]
}
```

Refresh() checks the validity of its current address, populates a routing table using the information received from its neighbors, and broadcasts a routing update (Listing 7).

Listing 8 PopulateRoutingTable(neighbor**)**

```
    update ← neighbor.update
    /* The level of the boundary the update just crossed */
    diff_level = GetDiffLevel(address, update.address)
    /* Create entry for the neighbor's subtree */
    if neighbor.LevelId(diff_level) ≤ ids[diff_level] then
      distance[diff_level] ← 1
 5:   /* set all bits of log diff_level to 0 */
      route_log[diff_level] ← 0
      /* set bit diff_level in log diff_level to 1 */
      route_log[diff_level][diff_level] ← 1
      id[diff_level] ← neighbor.LevelId(diff_level)
10: end if
    /* Update our table with neighbor's routing info */
    for i := (ADDR_LEN - 1) to (diff_level + 1) do
      if update.route_log[i][diff_level] == 0 then
        if id[i] > update.id[i] or
            ( id[i] == update.id[i] and
              distance[i] > update.distance[i] )
         then
          next_hop[i] ← neighbor.id
15:       id[i] ← update.id[i]
          distance[i] ← update.distance[i] + 1
          /* Copy the log */
          route_log[i] ← update.route_log[i]
          /* Set the proper log bit to 1 */
          route_log[i][diff_level] ← 1
          /* Clear out all lower bits (different parent tree) */
          for i := (diff_level-1) to 0 do
20:         route_log[i][i] ← 0
          end for
        end if
      end if
    end for
```

Let's see how a node updates its routing table upon receiving the routing update of a neighbor. When populating the routing table (Listing 8), the entry for each level, i, in the received

routing update is inspected is sequence, starting at the top level. For neighbors where the address prefix differs at bit i, we create a new routing entry, with a one-hop distance. It also has an empty route log, with the exception of bit i, which represents the level-i subtree boundary that was just crossed. The subtree identifier is computed using the id array in the update, using the procedure in 4. After this, the procedure returns, as the remaining routing information is internal to the neighbor's subtree, and irrelevant to the current node.

For nodes with the same address prefix as the current node, we go on to inspect their routing entry for level i. First, we ensure that the entry is loop free. If so, then keep the routing entry as long as the identifier of the entry is the same or smaller than what is already in the routing table, and as long as the distance (or some other metric of interest, see section X), is smaller.

A. Loop-Freedom under mobility

Routing loops, by definition, occur when a packet visits the same node more than once. In the case of DART, no guarantees can be made with respect to nodes, as routing is done based on addresses, and nodes can change address while a packet is in flight. This is a problem in general with mobility, as paths often break while packets are in transit.

In the static case, the route logs used in DART prevent the formation of routing loops. Let's study the various conditions that can occur due to mobility, and see how these are handled.

New link created. The creation of a new link (or in the wireless case, the detection of a new neighbor) will cause nodes to add new entries, or replace longer routes with shorter routes. Neither of these cause routing loops.

Link torn down. When a link is torn down, some entries in the routing tables of affected nodes may no longer be valid. These will have to be replaced by other routes. Route logs ensure that a node cannot accidentally accept a route that it had originated. However, packets are not (in the current implementation), protected against loops by route logs. Under rare circumstances, packets may end up following looped paths although no loop exists in the routing tables. This would require links to be created and torn down in rapid sequence, but is nonetheless possible. To protect against such events, a TTL field in the packet is decremented for each hop. When the TTL reaches zero, the packet is discarded.

Node address changed. Finally, there is the case of nodes changing address. Address changes and packet forwarding happen at very different time scales. Nevertheless, if a node was to have packets in its buffer while changing its address, these packets could potentially end up visiting the same nodes more than once. This is easily addressed by simply dropping any packets in the queue when an address change is necessary. Interestingly, our simulation results do not indicate that loops are a significant performance issue, so in our current implementation, such loops are handled by the TTL field.

VIII. Simulation Results

We conduct our experiments using two simulators. One is the well known ns-2 network simulator. The other is a simulator which we built to handle larger topologies, and to provide a graphical user interface for interactive experimentation. We initially developed our protocol using our own simulator, and later wrote a "wrapper" to embed it in ns-2. Our own simulator runs the same address allocation and routing code that we use in the ns-2 simulator, but replaces the intricacies of the mac and physical layers with a simple reliable message exchange, thereby improving simulation times.

In ns-2, we used the standard distribution, version 2.26. We used the standard values for the Lucent WaveLAN physical layer, and the IEEE 802.11 MAC layer code, together with a patch for a retry counter bug recently identified by Dan Berger at UC Riverside[9]. For all of the ns-2 simulations, we used the Random Waypoint mobility model with up to 800 nodes and a maximum speed of 5 m/s, a minimum speed of 0.5 m/s, a maximum pause time of 100 seconds and a warm-up period of 3600 seconds[10]. The duration of all the ns-2 simulations was 300 seconds[11], wherein the first 60 seconds are free of data traffic, allowing the initial address allocation to take place and for the network to thereby organize itself. The size of the simulation area was chosen to keep average node degree close to 9. For example, for a 400-node network, the size of the simulation area was 2800x2800 meters. This was done in order to maintain a mostly connected topology. Mobility parameters were chosen to simulate a moderately mobile network. DART is not suitable for networks with very high levels of mobility, as little route aggregation benefits are to be had when the current location of most nodes bear little relation to where these nodes were a few seconds ago.

Our simulations focus on the address allocation and routing aspects of our protocol, not including the node lookup layer, which is replaced by a global lookup table accessible by all nodes in the simulation. The choice of lookup mechanism (for example distributed, hierarchical, replicated, centralized, or out-of-band) should be determined by network characteristics, and performance may vary depending on what mechanism is used.

Here follows a summary of our findings. DSDV, due to its periodic updates and flat routing tables, experiences very high overhead growth as the network grows beyond 100 nodes, but nevertheless performs well in comparison with other protocols in the size ranges studied. AODV, due to its reactive nature, suffers from high overhead growth both as the size of the network, and the number of flows, grows. While AODV performs very well in small networks, the trend suggests that it is not recommendable for larger networks. DSR, in our simulations, performed well in small networks, and never experienced high overhead growth, likely due to its route caching functionality. However, due to excessive routing failures, DSR demonstrated unacceptable performance in larger networks. Finally, DART, demonstrated its scalability benefits in terms of no overhead growth with the number of flows, and logarithmic overhead growth with network size. Still in development, DART did not outperform, but performed

[9] Available for download at http://www.cs.ucr.edu/d̃berger

[10] The minimum speed and the warmup period were used to avoid the speed decay problem identified in [28]

[11] Although the de facto standard is 900 second simulations, we were forced to reduce this to in order to limit execution times and log file sizes.

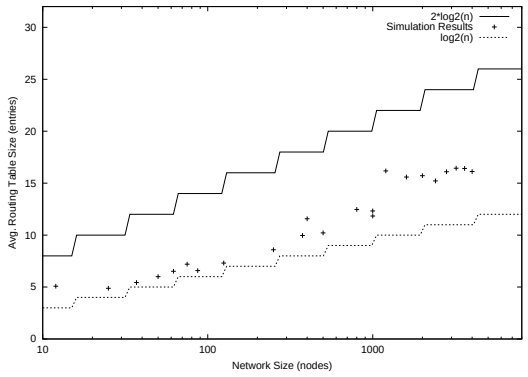

Fig. 5. The routing table size grows logarithmically with the size of the network.

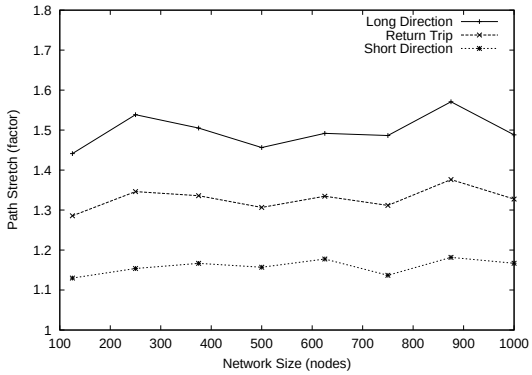

Fig. 6. Path stretch vs. network size. We observe a constant average path stretch of 30-35%. Return-trip denotes the stretch on a path going from source, to destination, and then back again.

on par with other protocols for the larger simulation scenarios. The trend suggests that DART would continue to scale well in scenarios beyond the capacity of our simulation environment.

A. Address Space Utilization

To evaluate the address space utilization effectiveness of the heuristic address allocation scheme described in section VI, we used our custom made, high-performance simulator. We set up a series of experiments in static topologies ranging in size from 12 nodes up to 4,000 nodes, and measured the average size of the routing tables of all the participating nodes. In these experiments, we used 64-bit addresses and chose parameters such that the average node degree was between 6 and 8, which is commonly used to ensure connectivity.

The routing table size indicates the number of empty siblings, or equivalently, the number of "free" bits in a node's address, and is thus a good metric to determine the effectiveness of the address allocation scheme. The average routing table size is also a good indicator of the overhead traffic incurred at each node, since empty entries can be communicated using a single bit, and thus incur essentially no extra overhead. Figure 5 shows the results of these experiments. As we can see, the average routing table size in all of our simulation runs falls between $\log_2 n$ and $2\log_2 n$.

Figure 5 clearly demonstrates that our current address allocation heuristic results in an efficient use of the address space, which in turn results in compact routing tables in the participating nodes. Due to time and hardware constraints, we were unable to perform simulations with more than 4,000 nodes, but we expect larger simulations scenarios to show the same general trend.

B. Path Stretch due to Aggregation

The use of routing by address prefix is a potential source of routing inefficiency, since we don't keep track of the optimal route for every destination. This effect is called *path stretch*, and is defined as routing path length over shortest path length. We created a set of static random topologies with sizes ranging from 125 to 1000 nodes in our custom-built simulator. We then sampled the path stretch between 1000 randomly selected node pairs. Figure 6 shows the average path stretch as network size increases. **We see a constant 30-35% increase in the average path length, due to the extensive route aggregation necessary to achieve logarithmic routing table sizes.** This comes out to 3-4 hops in a 1,000 node network, or 1-2 hops in a 100 node network. To put this in perspective, 20% of paths in the Internet see a stretch of more than 50% due to policy routing [29].

However the path stretch exhibits an interesting asymmetry; by measuring path stretch in both directions, we determined that one direction had a path stretch of 50%, whereas the other direction saw a stretch of 15%. We expect to be able to use this to our advantage on bi-directional connections, such as TCP, through the use of loose source routing, to bring down the average path stretch. In addition, our current work does not optimize the address allocation with respect to path length. Such techniques are part of our future work, and outside the scope of this paper.

C. Routing Overhead

All our experiments were performed for FTP as well as UDP/CBR flows, to accurately capture the effects of flow elasticity. In particular, simulations with elastic flows tend to favor shorter connections, as TCP is better able to ramp up the send rate on faster, and less lossy, paths. For the UDP/CBR flows, we varied the rate and number of flows, but kept the total offered load constant at 250 kbit/s. Flows had a uniformly and randomly selected start time between 50 and 180 seconds into the simulation, and stayed active until the simulation ended. The ns-2 simulator was configured to use standard 1 mbit/s 802.11 interfaces.

Since we are primarily concerned with routing overhead, we start by comparing DART routing overhead with that of AODV, DSDV and DSR. In the following experiment, we compare routing protocol scalability with respect to network size. **In Figure 7 shows how the flat routing utilized in**

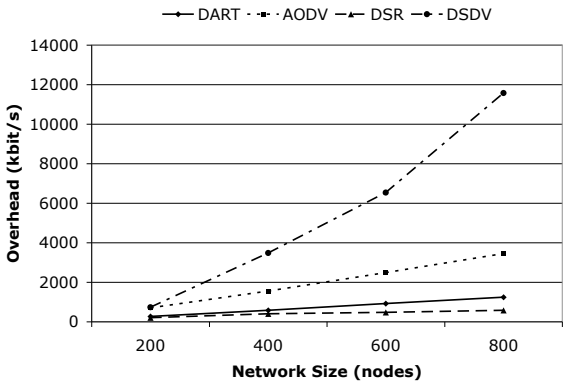

Fig. 7. Overhead vs. Node Network Size: 100 UDP/CBR flows. Total DART overhead grows as $n \log n$.

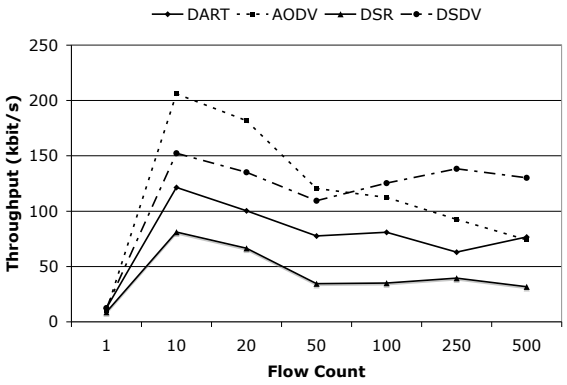

Fig. 9. Throughput vs. Flow Count: UDP/CBR flows, 200 Nodes.

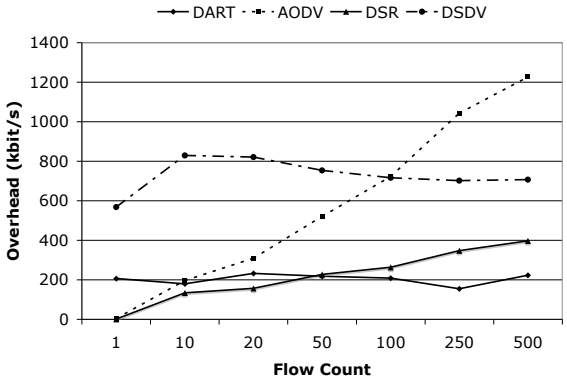

Fig. 8. Overhead vs. Flow Count: UDP/CBR flows, 200 Nodes. Total DART overhead is constant with respect to flow count.

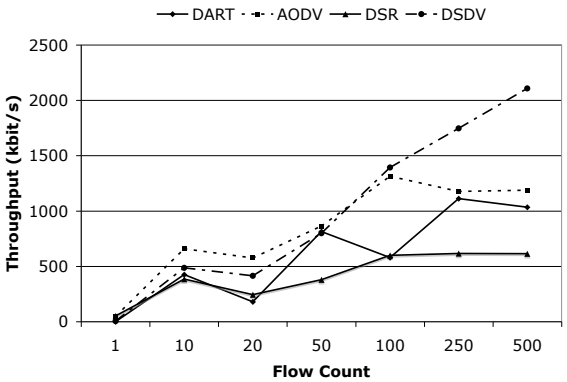

Fig. 10. Throughput vs. Flow Count: FTP flows, 200 Nodes.

DSDV causes total routing overhead to grow quadratically with network size. DART, on the other hand, maintains a relatively low overhead throughout the simulated range. Naturally, total overhead will grow at least linearly with network size, as each node periodically performs a local broadcast of its routing table. However, DART overhead *per node* grows logarithmically, as suggested by the logarithmic size routing tables reported in Fig 5. For this experiment, the overhead of AODV grows approximately proportional to the number of flows (50), and the size of the network. DSR overhead, due to aggressive route caching policy, grows at a rate similar to that of DART. However, as we shall see, this policy also results in frequent routing failures due to bad cache entries, and consequently dismal overall performance.

In our next experiment, we examine routing protocol overhead with respect to the number of flows. Here, we used a network size of 200 nodes.

Figure 8 shows that AODV and DSR overhead has an approximately linear relationship with flow count, whereas the overhead of DSDV and DART are unaffected by this parameter, due to their proactive route establishment. The overhead of the reactive protocols overtake that of DART very quickly, but even the high constant overhead of DSDV in a 200 node network is exceeded by AODV as the number of active flows exceeds 100.

D. Throughput

DART was designed with scalability in mind, and no tuning has been applied to optimize for throughput. Nevertheless, we are interested in comparing the relative performance of DART and several popular routing protocols.

Next, we study the throughput achieved by the four protocols, using a varying number of UDP/CBR and FTP flows respectively. **Figure 9 shows the proactive DSDV and DART remaining largely unaffected as the number of flows increases. As the number of flows increases, AODV's**

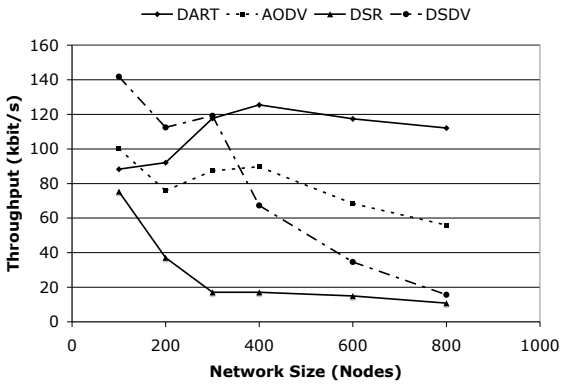

Fig. 11. Throughput vs. Network Size (Nodes): 100 UDP/CBR flows.

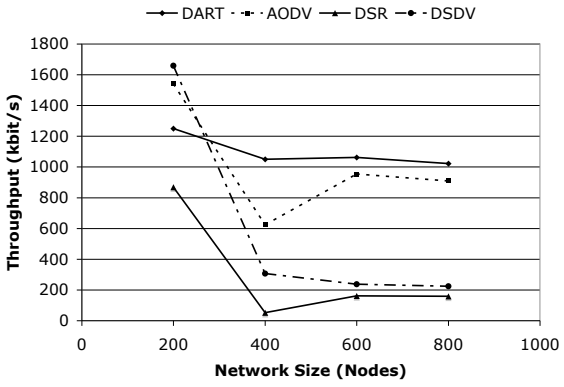

Fig. 12. Throughput vs. Network Size (Nodes): 100 FTP flows.

overhead eats up its initial performance advantage. A slight decrease in throughput for the UDP/CBR case is expected, as inter-flow interference will increase with increasing number of flows. For the FTP case, illustrated in Figure 10, all protocols improve with flow count, since the likelihood that short flows will appear increases. These flows will achieve relatively high throughput, due to the elasticity of FTP transfers. However, it is not clear that achieving high throughput on short, perhaps one-hop, paths is a useful measure of success for a routing protocol.

Finally, we study throughput achieved under varying network size. Here, we choose to keep the number of connections at a fixed 100, and vary only the network size. This should give us a good idea of protocol scalability with respect to network size. When connection end-points are chosen randomly and uniformly, it is natural for any protocol to see reduced throughput with increasing network size, due to increasing average path length, and increasing routing protocol overhead. On the other hand, there is also a gain in spatial diversity, if node density is kept constant, which cancels out some of this loss. In our experiments, all protocols except DART experience throughput reduction as network size increases. Referring to figure 7, we believe this is due to excessive overhead on for AODV and DSDV, whereas DSR is experiencing a high level of routing failures as network size goes beyond 200. **In figures 11 and 12 DART shows performance similar to other protocols for the smaller network sizes, but shows a decidedly more favorable trend as network size increases.** AODV manages to achieve good throughput in the FTP scenario, likely due to short routes getting a majority of the traffic. However, in the UDP scenario, where such adaptation is not performed, DART easily outperforms all other protocols. Due to the small and constant overhead of DART, DART is the only protocol out of the four that shows promise for large networks.

With these throughput results, we want to demonstrate that dynamic addressing is a feasible and promising approach to creating a scalable routing protocol, in that its performance is on par with flat routing protocols, while its overhead is significantly less than these. We would like to point out that while DSR and AODV have many years of optimization work behind them, DART as presented here is in its most basic form, naturally leading to a performance disadvantage in smaller networks. We fully expect there to be significant opportunities for optimization of our protocol, and we outline some of these in the following section.

IX. Overhead and Worst Case Analysis

In this section, we analyze the performance of our address allocation scheme analytically and with qualitative arguments. The analysis suggests that dynamic addressing seems very promising for scalability.

First, we examine two types of topologies that pose a challenge to our address allocation scheme. We provide a solution to the case of star-like topologies, and argue that string topologies can be expected not to be a problem in realistic scenarios.

Second, we compare the overhead incurred with proactive and reactive ad hoc routing protocols. We develop an analytical framework and find the regime in which proactive protocols are more efficient than their reactive counterparts in terms of overhead. We argue that operations in this regime are typical in practical, large scale, scenarios.

A. Topology and Address Allocation

We examine the efficiency of dynamic addressing in terms of the address space we need for assigning legitimate unique addresses to n nodes.

Lower bound. How many bits of address do we need in order to give every node in a size n network a unique address? The tight lower bound is obviously $\log_2 n$ bits.All flat addressing schemes can be expected to achieve this lower bound.

Dynamic addressing needs a larger address space given the prefix subgraph constraint. The constraint precludes nodes that are far apart from having nearby addresses in the address

space. Therefore, any arbitrary available addresses is not necessarily legitimate for any new or re-locating node.

How much larger can the address space become? This depends on the topology of the network. We study some typical and extreme topologies to obtain an intuitive feeling.

Uniformly Random Topologies. For the case where the network can be described as a uniformly random topology, we refer to the simulation results in section VIII. These results, although clearly representing sub-optimal solutions, nevertheless show an average routing table size of less than $2\log_2 n$, or $O(\log_2 n)$.

Star-like Topologies. A star topology presents a different challenge for dynamic addressing. A star consists of one *central* node in the middle, and a large number of *peripheral* nodes connected to the central node without having any connectivity between themselves. Due to the prefix subgraph constraint, the peripheral nodes cannot belong to the same address subtree, unless the central node is included. Assuming that the central node has address [000..0], its neighbors will be compelled to choose addresses like [100..0],[010..0],[001..0]...[000..1]. There are only l such addresses, and this is the limit on the number of peripheral nodes that we can support.

Is this a realistic scenario? We claim that for a high-degree node with disconnected neighbors to exist in an 802.11 network, it must have more than one network interfaces. We then present a solution to this specific problem. The solution depends to some extent on the type of network, the related technology, and the environment of deployment.

Omnidirectional antennas. The star-like topology is not a concern in a typical ad hoc network with omnidirectional antennas. In this context, it is unrealistic to have more than a handful of neighbors that do not hear each other[12]. If we consider natural obstacles and other effects, the number of such neighbors can increase, but we need a really peculiar landscape for the number to become large.

Multiple Network Interfaces. Due to the inherent scalability problems in today's wireless MAC and physical layers, we are compelled to consider networks where wires and fixed directional antennas play a role in providing additional bandwidth.

This could, for example, involve a purely wired router with several wired interfaces, a wireless base station, or a wireless node with directional antennas. In these cases, the node could have an arbitrary number of neighbors that would be disconnected were it not for this node.

The solution, which solves this problem completely, is to assign a distinct address to each network interface. The node with several interfaces can assign valid addresses to its interfaces according to any criteria it wishes, and, most importantly, can balance the address space across all the interfaces. By enforcing a locally balanced address space, it ensures a locally optimal address allocation, thereby almost completely eradicating the risk of running out of address space.

String topologies. A string topology is our worst case scenario. This is not due to the prefix subtree constraint, but is specific to the particular order in which we choose to assign addresses in the current version of our protocol. Consider a string of nodes $u_0, u_1, \ldots, u_{n-1}$, placed in that order. Assume that u_0 initiates the network, and takes address [000..0]. Then the subsequently joining nodes, will get addresses [100..0], [110..0],[111..0]...[111..1], for u_1 to u_{n-1} respectively, according to our address allocation scheme.

With l-bit addresses, the address space could potentially be depleted at the most recently joined node when the network size is $l+1$. With $l = 128$, the routing table can hold strings of at least 129 nodes, and at most 256 nodes, depending on the position of the [000...0] node. One might expect that string topologies of this length will be extremely uncommon.

In section X, we describe a patch that can enable nodes to join and communicate without having a unique address, by sharing an address with a neighbor.

B. The Overhead of Proactive and Reactive Routing

Here, we make a comparative analysis of the communication overhead of reactive protocols and proactive protocols. Our dynamic addressing falls in the proactive routing category, which is often criticized as power inefficient, since they exchange messages even when there is no traffic. Reactive routing is widely regarded as the technique of choice for ad hoc networks, but these protocols all rely on some form of flooding to identify paths on demand. We will demonstrate that the use of flooding for route establishment causes scalability problems in large networks with many active connections.

The focus here is the communication overhead, which we define as the number of non-data bytes transferred. This is necessary to account for the size of the control packets, since in some cases this increases with the size of the network. This definition also captures the additional overhead of data packets in source routing.

We start by identifying a key parameter: the arrival rate of connections, or the connection establishment frequency (CEF). The overhead of reactive protocols is tightly coupled with the connection arrival rate. Each new connection requires at least one route search which in the reactive protocols requires a flooding of the network[13], which uses $O(n)$ messages in an n node network. In a proactive protocol, the number of update messages is $O(n)$ per update period and it is independent of the number of connections. Let us define one update period to be our **unit of time**. Intuitively, if one flooding route lookup is performed per unit of time by any node in the network, reactive routing begins to exhibit higher message overhead than proactive routing.

Note that the analysis here is qualitative. We attempt to capture the general trends of the behavior of the two approaches. Although simplifications are inevitable, the analysis is representative of the nature of the two approaches.

For simplicity, we do not consider mechanisms that do not affect the asymptotic performance. For example, we expect that route caching, path overhearing and local route repairs

[12]This is easily proven with geometry. In the simple case, one can show that this number is at most 5.

[13]We can deploy caching, expanded ring search or other techniques in an attempt to limit the extent of a flood, but asymptotically the cost is the same.

can lead to significant, but nevertheless constant factor improvements.

Reactive protocol overhead. Let us define some parameters that define the performance of the reactive protocols.

$rrc(n)$The cost of a single route request.
$cef(n)$The rate of connection establishment.
$rrr(n)$The rate of repeated route requests.

Here, $cef(n)$ is related to the size of the network, the total offered load, and the average connection duration. $rrr(n)$ is primarily related to the size of the network and node movement and other causes of route failure. Excessive offered load could also have an effect since it is known to cause false link failures in wireless networks. With the above definitions, we can quantify the per-time-unit routing overhead, $React(n)$, of reactive protocols as follows:

$$React(n) = O(rrc(n) \cdot cef(n) + rrc(n) \cdot rrr(n)) \quad (1)$$

Data Packet Overhead. Reactive protocols with source routing can have significant data packet overhead. In source routed protocols, such as DSR, the overhead of sending the route with every packet dominates this equation when mobility is low and traffic volume is high. The per-packet overhead grows linearly with the path length,

$$PackOver(n) = O(path(n)).$$

How does the average path length, $path(n)$, grow as a function of the size? This depends on both the topology and the distribution of pairs of nodes that communicate. For asymptotic analysis, it is fair to assume that the average distance between communicating pairs is a constant fraction of the diameter of the network.

In a two dimensional ad hoc network with homogenous omnidirectional nodes, we expect that the path length will be $path(n) = O(\sqrt{n})$ if the nodes are uniformly distributed. In this environment, strict source routing is probably not feasible for large networks. For the remainder of this discussion, we will focus on the routing message overhead only.

Proactive protocol overhead. The overhead of a proactive protocol, can be described with a single parameter, $size(n)$, the average size of a single routing update. Hence, we have the following formulation for the per-time-unit overhead of proactive protocols,

$$Proact(n) = O(n \cdot size(n)). \quad (2)$$

Depending on the approach taken, the average routing table size can vary significantly.

Flat addressing. Recall that some approaches for proactive routing, such as DSDV [13], use flat addressing. The size of the routing table, $size(n)$, increases linearly with the number of nodes n; $size(n) = O(n)$. Asymptotically, for a really large n, nodes are so busy transmitting the routing table, that they cannot transmit anything else.

Dynamic addressing or hierarchical routing. As mentioned earlier in this section, the average routing table size when using dynamic addressing is $O(\log n)$.

When do proactive protocols incur less overhead than reactive protocols? The question is captured in the following inequality.

$$Proact(n) \leq React(n) \quad (3)$$

We need to assign values to these quantities in order to identify the regime in which the inequality holds. As explained above, it is reasonable to assume that the message overhead of a reactive route lookup is: $rrc(n) = O(n)$. Accordingly, inequality 3, skipping the O-notation, and dividing by n, becomes the following:

$$size(n) \leq cef(n) + rrr(n) \quad (4)$$

We already know that for hierarchical routing based on dynamic addressing, $size(n) = O(\log_2 n)$, so we arrive at the following[14]:

$$\log_2 n \leq cef(n) + rrr(n) \quad (5)$$

When is this condition satisfied? Clearly, it is true for a sufficiently high connection establishment rate. We believe that it is true in any realistic network for sufficiently large n. To see why, consider a network where all nodes have a small, constant probability of establishing a connection during a unit of time. In this network, the connection establishment rate increases linearly with network size, whereas the size of the proactive routing updates grows logarithmically with network size. According to asymptotic analysis, at some point the cost of establishing connections in the reactive protocol will surpass the cost of the periodic routing updates in the proactive protocol. The actual sizes and connection establishment rates necessary to achieve this depend on the protocols involved and can be determined through experimentation. We conclude that for sufficiently large networks and/or high connection establishment rates, proactive routing using our dynamic addressing approach is likely to scale better than any purely reactive routing protocol.

X. Discussion

In this section, we briefly discuss several optimization mechanisms, and implementation issues for our addressing approach. First, we outline ideas of how we can optimize the routing update frequency by making it adaptive to the network needs. Second, we discuss how we can improve the network stability by assigning node identifiers according to the expected behavior of the node. Third, we outline our ongoing efforts on security. Finally, we discuss implementation issues of our approach and discuss its interoperability with the Internet.

[14]Hierarchical routing will invariably incur path stretch. However, our simulation results indicate that path stretch is constant with respect to network size in our protocol.

A. Optimizing Routing Updates

Here, we present two opportunities for improving the performance of our dynamic addressing scheme.

Adaptive Routing Update Frequency. We are currently evaluating the merit of a locally adaptive scheme for the routing update rate. Determining the correct frequency for the routing updates is important for good performance. A fixed update frequency will not be suitable for all operational conditions. A high frequency means good response to highly mobile scenario, but it could lead to waste of resources in a slower moving phase.

Triggered Updates for Improved Convergence Time. We are evaluating mechanisms to improve the convergence speed of the routing information. Apart from the periodic updates, we are considering triggered updates in response to routing changes. Such a mechanism exists in DSDV, which also uses periodic routing updates [13]. The downside is that triggered updates increase the overhead of the protocol, and could cause detrimental ripple effects throughout the network.

B. Assignment of Node Identifiers and Robustness

The assignment of node identifiers can have significant impact on the performance, since the ”lower-id” rule is often used to resolve a conflict. By assigning lower identifier numbers to more reliable nodes, we can achieve increased performance and stability. For example, stationary base stations are highly reliable and less likely to move away. If we have several base stations with low identifiers and interconnect them by reliable means, we can ensure that the address space in an entire region maintains a balanced and stable structure, even as high-speed mobile nodes move through it.

In contrast, we want to assign high identifiers to “volatile” devices such as mobile phones and PDAs. These move both quickly and frequently, and are likely to be turned off. By assigning higher identifiers to these types of units, their volatile behavior will not affect the network at large. The assignment of these identifiers can be done during manufacturing, just like the MAC address of network interface cards.

C. Use of Routing Metrics

In this work, we have only made use of a hop-count metric for routing. Other metrics exist that could substantially improve routing performance, and nothing precludes the use of such metrics in a dynamic address routing protocol. It should be noted that DART, like any other protocol that uses hierarchical types of routing, cannot provide guarantees for finding the minimum cost path to any given destination. Instead, DART finds the minimum cost path to a given subtree, and once in that subtree finds the minimum path cost to the next, lower-level, subtree. This will invariably lead to some amount of path stretch, as illustrated in Section VIII. Path stretch is the price we must pay for small routing tables.

D. Using Additional Neighbor Information

The routing scheme described above limits the internal routing table of nodes to one entry per sibling subtree. However, some nodes will likely receive routing updates from several nodes within the same sibling subtree. A trivial optimization would be to allow nodes to use all the routing information received from their neighbors, to find the best next hop. However, the information broadcast in their routing updates would still contain only one entry per level, with no modification as compared to the original scheme.

E. Coping with Temporary Route Failures

In some cases, a route to the given destination address may not be available, even though the network is connected, and all address allocations are correct. Such temporary route failures can be the result of route propagation delay; when a shorter route breaks, there is an interval of time where nodes are not aware of the route breakage, and a new longer route has not yet been established.

In this situation, the default action by a router that finds itself without a valid path would be to drop the packet, and potentially send a ”no such route” message back to the sender. However, if such failures are common, given a certain mobility scenario, it may be a better idea to delay for some amount of time to allow the new route to be established.

F. Coping with Destination Address Changes

In other cases, the destination address may not be routable because the node that used to be there has changed its address. Here, the last router on the path is faced with a choice; it can notify the sender that there is no route to the destination address, thereby prompting the sender to perform a new lookup operation. Alternatively, it can do the lookup operation on its own and forward the packet to the destination node’s new address. Clearly, the source needs to be notified of the address change. However, in a scenario of bi-directional communication, such as with TCP flows, it would be more economical to have the destination node notify the source in the next outgoing packet, than to have the source perform a new lookup operation. In addition, dropping the packet may not be desirable if low packet loss is important in higher layers of the network stack. Finally, given the lookup table optimizations described in the next section, there can be a significant cost advantage to performing a lookup operation from a node close to the intended destination, as opposed to a node far away from said destination. All these factors need to be taken into consideration when deciding whether to fail easy and drop a packet, or to buffer the packet and more reliably forward them to their new destination address.

G. Handling Address Space Exhaustion

We will now provide a solution to temporarily extend connectivity even when the address space is locally exhausted. The key idea is that an existing node can act as a gateway for a joining node that cannot obtain a legitimate address. This is in many ways similar to a Network Address Translation (NAT) firewall. As far as the larger network is concerned, the gateway simply has many identifiers mapped to its address. In the subnet on the inside of the gateway node, a separate

address space is used, with plenty of space for new nodes. When a gateway receives a packet from the larger network, it looks up the "inner" address of the specified identifier, and forwards it to this address in the inside network. We omit further details due to space constraints.

H. Security

Our focus is to establish the feasibility of dynamic addressing as a way to achieve scalability in ad hoc routing. Security is a constraint that needs to be addressed in a practice, but it extends beyond the scope of this paper. The goal of our current security work is to provide the routing layer with "sabotage resistance". Here, sabotage resistance means a robustness against false route advertisements, such that an attacker can only affect a limited portion of the network, over a limited time span. Recently, several pioneering routing security approaches have been developed [30] [31] [32] [33] and we are using their results to guide our effort.

I. Implementation and Deployment Issues

We intend to develop and release a prototype implementation of our protocol for Linux and Mac OS X in the near future. For a realistic implementation of the protocol, it will be crucial to be able to: (i) support the use of IP-based applications such as web browsers and email readers, (ii) provide a way to access Internet resources, and (iii) connect several of these networks over the Internet.

We expect to solve (i) by hiding the workings of our routing protocol to the application layer. Essentially, we let the node identifiers be IP addresses in the `10.*.*.*`[15] range, and wedge our routing layer between the IP and mac layers in the protocol stack, thereby hiding the dynamically allocated routing address from the higher layers and preserving compatibility. Issue (ii) can be handled by Network Address Translation on gateway nodes connected to the Internet. Finally, our plan is to solve (iii) by way of an overlay network of gateways that tunnel packets through the Internet. All of these solutions are proven techniques, and this makes the integration our protocol with the current Internet infrastructure a feasible goal.

XI. Conclusion

In this paper, we propose Dynamic Address RouTing, an initial design toward scalable ad hoc routing. We outline the novel challenges involved in a dynamic addressing scheme, and proceeded to describe efficient algorithmic solutions. We show how our dynamic addressing can support scalable routing. We demonstrate, through simulation and analysis, that our approach has promising scalability properties and is a viable alternative to current ad hoc routing protocols.

First, we qualitatively compare proactive and reactive overhead and determine the regime in which proactive routing exhibits less overhead that its reactive counterpart. Large scale simulations show that the average routing table size with DART grows *logarithmically* with the size of the network. Further simulations show a constant average path stretch of about 30-35%, which is reasonable when compared to what is observed in the Internet today.

Second, using the ns-2 simulator, we compare our routing scheme to AODV, DSR and DSDV, and observe that our approach achieves superior throughput, and with considerably smaller overhead, in networks larger than 400 nodes. The trend in simulated overhead, together with the analysis provided, strongly indicate that DART is the only feasible routing protocol for large networks. Finally, we describe a number of proposed optimizations to our protocol, which can further improve the performance of our dynamic addressing approach.

The motivation behind this work was to challenge the status quo in ad hoc routing. We believe that dynamic addressing can be the basis for ad hoc routing protocols that for massive ad hoc and mesh networks.

[15]This is range of addresses reserved for local use in IP networks.

XII. Acknowledgements

This work was supported by the NSF CAREER grant ANIR 9985195, DARPA award NMS N660001-00-1-8936, NSF grant IIS-0208950 TCS Inc., DIMI matching fund DIM00-10071, and DARPA award FTN F30602-01-2-0535.

References

[1] Jakob Eriksson, Michalis Faloutsos, and Srikanth Krishnamurthy, "Scalable ad hoc routing: The case for dynamic addressing," in *IEEE InfoCom*, 2004.
[2] Nicholas Negroponte, "Being wireless, 2002," www.wired.com/wired/archive/10.10/wireless.html.
[3] PersonalTelco Project, "Personaltelco," www.personaltelco.com.
[4] "Consume.net project: Trip the loop, make your switch, consume the net!," www.consume.net.
[5] "Wireless anarchy," www.wirelessanarchy.com.
[6] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister, "Smart dust: Communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, 2001.
[7] Ram Ramanathan and Martha Steenstrup, "Hierarchically-organized, multihop mobile wireless networks for quality-of-service support," *Mobile Networks and Applications*, vol. 3, no. 1, pp. 101–119, 1998.
[8] Guangyu Pei, Mario Gerla, Xiaoyan Hong, and Ching-Chuan Chiang, "A wireless hierarchical routing protocol with group mobility," in *WCNC*, 1999.
[9] G. Pei, M. Gerla, and X. Hong, "Lanmar: Landmark routing for large scale wireless ad hoc networks with group mobility," in *ACM MobiHOC'00*, 2000.
[10] X. Hong, M. Gerla, G. Pei, and C. Chiang, "A group mobility model for ad hoc wireless networks," 1999.
[11] J. Eriksson, M. Faloutsos, and S. Krishnamurthy, "Peernet: Pushing peer-2-peer down the stack.," in *IPTPS*, 2003.
[12] C. Perkins, "Ad hoc on demand distance vector routing," 1997.
[13] Charles Perkins and Pravin Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," in *ACM SIGCOMM'94*, 1994.
[14] David B Johnson and David A Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*, vol. 353. Kluwer Academic Publishers, 1996.
[15] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward, "A distance routing effect algorithm for mobility (DREAM)," in *ACM/IEEE MobiCom*, 1998.
[16] Y.-B. Ko and N.H. Vaidya, "Location-aided routing (LAR) in mobile ad hoc networks," in *ACM/IEEE MobiCom*, 1998.
[17] Xiaoyan Hong, Kaixin Xu, and Mario Gerla, "Scalable routing protocols for mobile ad hoc networks," *IEEE NETWORK*, vol. 16, no. 4, 2002.
[18] Z. Haas, "A new routing protocol for the reconfigurable wireless networks," 1997.
[19] Guangyu Pei, Mario Gerla, and Tsu-Wei Chen, "Fisheye state routing: A routing scheme for ad hoc wireless networks," in *ICC (1)*, 2000, pp. 70–74.

[20] Paul F. Tsuchiya, "The landmark hierarchy : A new hierarchy for routing in very large networks," in *SIGCOMM*. 1988, ACM.
[21] Benjie Chen and Robert Morris, "L+: Scalable landmark routing and address lookup for multi-hop wireless networks," 2002.
[22] L. Kleinrock and F. Kamoun, "Hierarchical routing for large networks: Performance evaluation and optimization,," *Computer Networks*, vol. 1, 1977.
[23] Aline C. Viana, Marcelo D. de Amorim, Serge Fdida, and Jos F. de Rezende, "Indirect routing using distributed location information," *ACM Mobile Networks Applications, Special Issue on Mobile and Pervasive Computing*, 2003.
[24] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica, "Geographic routing without location information," in *ACM MobiCom*, 2003.
[25] James Newsome and Dawn Song, "Gem: graph embedding for routing and data-centric storage in sensor networks without geographic information," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA, 2003, pp. 76–88, ACM Press.
[26] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM'01*. ACM, 2001.
[27] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing*, May 1997, pp. 654–663.
[28] Jungkeun Yoon, Mingyan Liu, and Brian Noble, "Random waypoint considered harmful," in *INFOCOM*, 2003.
[29] H. Tangmunarunkit, R. Govindan, and S. Shenker, "Internet path inflation due to policy routing," .
[30] Yih-Chun Hu, David B. Johnson, and Adrian Perrig, "Sead: Secure efficient distance vector routing in mobile wireless ad hoc networks," in *Fourth IEEE Workshop on Mobile Computing Systems and Applications* (WMCSA '02), June 2002, pp. 3–13.
[31] Yih-Chun Hu, Adrian Perrig, and David B. Johnson, "Ariadne: A secure on-demand routing protocol for ad hoc networks," in *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking* (MobiCom 2002), Sept. 2002.
[32] Lidong Zhou and Zygmunt J. Haas, "Securing ad hoc networks," *IEEE Network*, vol. 13, no. 6, pp. 24–30, 1999.
[33] Sergio Marti, T. J. Giuli, Kevin Lai, and Mary Baker, "Mitigating routing misbehavior in mobile ad hoc networks," in *Mobile Computing and Networking*, 2000, pp. 255–265.