Variable-length Address Routing for Mobile Ad Hoc Networks

Ulisses Amaya Suvidhean Dhirakaosal Wojciech Jawor

Department of Computer Science,
University of California,
Riverside, CA 92521.
{uamaya|sdhirak|wojtek}@cs.ucr.edu

December 12, 2003

Abstract

We consider a Mobile Ad Hoc Network (MANET) with dynamically assigned addresses. Unique identifiers and addresses of the nodes in the network are separated. Addresses have variable length and are assigned based on the relative position of the nodes. When a node changes its location it may also change its address. We propose efficient algorithms for address assignment, routing, and address lookup.

1 Introduction

A mobile ad hoc network (MANET) is a self-organizing network formed by a collection of wireless mobile nodes without the aid of centralize administration. Each node in the MANET is autonomous (serving as a router and a destination node) and may move randomly resulting in a fully dynamic topology. One example of a MANET could be PDAs communicating with cell-phones and laptops at a busy coffee shop joint. Another example of MANET would be military units that need to keep in contact with each other while patrolling around a given terrain.

Most of the work done in this area have used static identifiers and addresses for their mobile nodes. Static addressing allows for quick implementation of existing networking protocols in MANETs at the expense of scalability. Since the sizes of MANETs are expected to become very large in the future, current static addressing schemes will not be a viable choice.

To address the main problem of scalability, we propose a variable-length dynamic addressing scheme. Such a scheme not only alleviates the scalability problem, but also reduces overhead since the space used for addresses is smaller than a static addressing scheme in a similarly sized MANET. Our variable-length addressing scheme includes efficient algorithms for address assignment, packet routing, and address lookup. The address space in our network is a binary tree. Address assignment and packet routing both take advantage of this organization for efficiency. The efficiency of our address lookup protocol is a combined result of a well designed hash function and a simple prefix matching method.

Previous work. It is widely believed that the routing protocols for MANETs should follow similar conventions and methods as those used in wired networks [1] and a myriad of papers has been published that follows that track ([5], for example). [1] is also one of the first papers that considers a variable address architecture for MANETs. However the proposed scheme requires a node to flood the network to find the next available address. A general discussion on routing protocol performance issues for MANETs can be found in [3] and an evaluation of proposed methods in [2]. Finally, in [4] an architecture with dynamic address assignment is considered. However, the addresses in that work are of fixed length.

Organization of this paper. The following sections are organized as follows – Section 2 is an overview and explanation of notations used in this work. Section 3 details the implementation of variable address assignment and how to deal with conflicting addresses. Section 4 details how a packet is routed through the network and how to avoid loops. Sections 5 and 6 explain how to merge two networks and how to split one network into two. Section 7 details how the addresses are looked up and retrieved. Finally, Section 8 presents our conclusion.

2 Overview and notation

Our architecture will maintain the following property *the addresses of nodes that are geographically close to each other share a common prefix.* The addresses of the nodes will change as the 'neighborhood' of the node changes. We thus separate the node addresses and the node identities.

The network will contain two main subnets $0*$ of nodes with addresses with prefix 0 and $1*$ of nodes with prefix 1. Each of these networks will recursively be divided into smaller subnets.

The addresses in the network will form an abstract address tree with labels 0 or 1 on the edges (see Fig. 1). The tree is a binary tree in which the address of a node is a string formed by the labels on the path from the root to that node. We assume that the address of the root is an empty string ϵ. In our notation the label of left subtree is always 0 and the label of the right is 1. Using the address tree, we may introduce a concept of *opposite subnets*. An opposite subnet of a node with address *add*, is obtained by taking any prefix of *add* and negating the last bit. For example, for a node with address 0110 the opposite subnets are 1, 00, 010, and 0111.

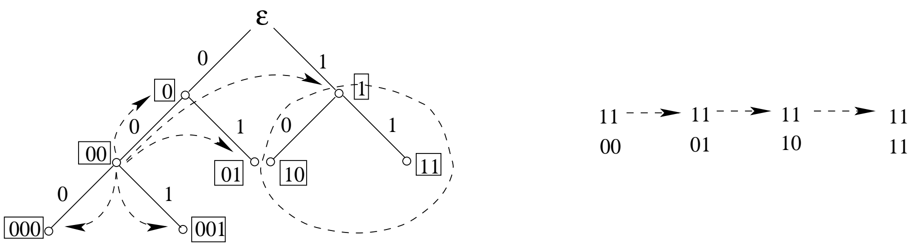

Figure 1: The address tree. The dashed arrows represent the entries in the routing table for the node 00. An example path traveled by a packet from node 00 to 11 is shown on the right.

Each node of the network contains a routing table. The table contains an entry for each of the opposite subnets of that node, two entries for the lower level subnets (the subnets containing the children of the node and not the node itself) and a list of entries for the nodes with addresses that are prefixes of the address of the particular node. For example a node 011 will contain two entries for the subnets $1*$, $00*$, two entries for the subnets $0110*$, $0111*$ and entries for the nodes 0, 01.

The important property of the routing paths established in the network is that nodes from a particular subtree will communicate using the nodes from this subtree only. This will help avoid routing loops. A loop will exist if a packet leaves and reenters a subtree.

3 Address Assignment

When a new node joins the network it chooses one of its neighbors to be the address allocator. The prefix of the assigned address is always some prefix of the address of the allocator. This assumption is meant to help to solve the problem of even address distribution throughout the network. It will also speed up the process of creating the routing table for the new node and keep the routing paths relatively short.

The address allocator must ensure that the address it advertises to the new node is not already occupied. The suffix of the address it assigns is chosen randomly in the following manner. The node flips a coin and decides whether the first bit of the suffix is 0 or 1. Next, it tries to determine whether this particular address is already being used in the network by consulting the routing tables of the neighbors and/or contacting the node with the address directly. If the address is already being used, the next random bit is chosen and the procedure is repeated. If the addresses are distributed evenly throughout the network the number of random bits the allocator chooses is $O(\log n)$ on average.

The procedure to determine the next unoccupied address does not have to be carried out when the allocator receives an address request. The node should, however, periodically check if the address it would advertise is not being assigned to a different new node by a third party.

The new node may choose the address allocator in many ways. First it may randomly choose one of its neighbors. Second, it may choose the neighbor with the longest or shortest address. Third, it may try to acquire an addresses from some subset of its neighbors and choose the shortest one. [1]

When the neighborhood of a node changes due to mobility the node should requests a new address. This ensures that the nodes close to each other share common prefixes, which keeps the length of the routing path relatively short. If a node cannot contact any other node, it changes its address to 0 and creates a new network. One way to detect the change of the neighborhood is to keep track of the position of the node's address allocator. This must be done using the allocators ID, not its address, as it may change.

The requirement of containing the address of the allocator as the prefix of the new address poses the following threat. If node 0 dies then there is no node that assigns addresses with one bit prefix 0. Therefore, if, for example, node 00 dies, its address will never be reused. This may eventually lead to starvation of some ranges of addresses. An easy way to go around the problem is to allow the nodes to assign addresses not only containing the address of the node as the prefix, but also to assign addresses that are prefixes of the address of the node allocator.

Many tasks that a node needs to perform require the list of neighbors of the particular node. This list is always denoted by L in the pseudo code. The following procedure is run by a node to obtain a new address.

[1] We suspect that the three different approaches may not yield drastically different results. The nodes sharing a common prefix occupy same locations, so the different neighbors will probe the same subnets for a free address.

```
Procedure ObtainAddress
  if (—L— == 0)
    myAdd = 0
    myNetworkID = GenerateNetworkID()
    InitializeRoutingTable()
  else
    l = ChooseAllocator(L)
    (myAdd,myNetworkID) = RequestADDID(l)
    UpdateDNS(myID, myAdd)
    InitializeRoutingTable()
  endif
```

The following procedure is used to construct an address.

```
Procedure FindNewAddress
  tmpAdd = myAdd
  if (!isOccupied(myAdd shr 1)) return (myAdd shr 1)
  // shr deletes the least significant bit (shifts right)
  repeat
    suffix = randomly chose 1 or 0
    tmpAdd = tmpAdd appended by suffix
    inUse = isOccupied(tmpAdd)
  until (!inUse)
  return tmpAddr
```

```
Procedure isOccupied(add)
  repeat
    ping host add
    if host reachable return true
  until (unsuccessful P times)
  return false
```

Conflict of addresses. Our protocol does not give a strict guarantee that at any time two nodes are not assigned the same address. Once a duplication of an address is detected, one of the two nodes must give up its address and obtain a new one.

We propose the following way of detecting if two nodes have the same address. When a node transmits a packet to the destination it includes the destination identifier in the packet header. When a node receives a packet it checks whether its identifier and the identifier from the packet header match. If they do not match, then the node contacts its DNS (see section 7) to check when the latest update for the packet identifier occurred. If the update took place after the node was assigned the new address then there is a high probability that a conflict of addresses exists in the network.

```
Procedure IsSameAddress(header)
  if (header.ID != myID) && (header.Add == myAdd)
    retrieve DNS information
    if (DNS.updateTime > myAdd.creationTime)
      ObtainAddress()
    endif
  endif
```

4 Routing

The routing table of a node contains entries with the following information: subnet identifier, the next hop on the path to the subnet, and a subtree tag. As said before, the entries in the routing table can be divided into the following categories: entries for each of the opposite subnets of that node, two entries for the lower level subnet (the subnets containing the children of the node and not the node itself) and a list of entries for the nodes with addresses that are prefixes of the address of the particular node. For example the routing table of a node 01101 would contain an entry for each subnet $1*$, $00*$, $010*$, $0111*$, $01100*$, entries for the child subnets $011010*$, $011011*$ and entries for each of the nodes 0, 01, 011, 0110.

It is easy to see that the size of the routing table is $O(\log n)$, where n is the total number of nodes, if the address tree is balanced.

It is also easy to see that the routing tables of two nodes with addresses that contain a common prefix (like the address of the address allocator and an address of a new node), contain similar entries, for the prefix part. The second part of the routing table, that corresponds to the randomly allocated code must be built in addition to the prefix entries. This is illustrated by the following example.

Suppose that a new node obtains address $N = 0110100$ from the node $A = 01101$. The routing table of A contains entries: $1*$, $00*$, $010*$, $0111*$, $01100*$, 0, 01, 011, 0110, 011010, 011011. The routing table of N should contain all of the above entries plus $011011*$, $0110101*$, 011010, 01101000, 01101001. The next hop in the routing table of N for the first group is A (unless this introduces a potential loop or the entry in the routing table of A is blank). The next hop for the second group is determined by examining the routing tables of other neighbors. Some of the entries may be blank.

Avoiding loops Recall that the routing paths established in the network satisfy the following property: nodes from a particular subtree communicate using the nodes from this subtree only. This property allows us to detect loops easily. If a packet leaves a certain subtree and then reenters the subtree again we have a loop. We may efficiently detect loops using *tag* entries in the routing tables. The tags will basically be a compact representation of the subtrees visited by a packet.

If the packet is currently in a node with address of length N the tag will be a binary string of length $N + 2$. The interpretation of the tag will be as follows: if kth bit of the tag (where $1 \leq k \leq N$) is set to 1, the kth opposite subtree on the path from the root to the current node has been visited. If $N + 1$st bit is set then the *left* subtree of the current node has been visited; if the $N + 2$nd bit is set the *right* subtree of the current node has been visited.

The tag TAG of a packet is maintained using the following algorithm (m is the length of the *new* TAG)

1. If the packet travels up the tree, TAG gets shorter and the $m + 1$st, or $m + 2$nd bit is set.

2. If the packet travels down the tree, the tag is extended by 0s, to match the length of the path from the root to the current node

3. If the packet leaves a subtree, TAG is truncated to match the length of the path from the root to the node and the bit ith bit (corresponding to the subtree just left) is set.

The packet enters the same subtree twice if the same bit is about to be set for the second time.

The algorithm can be illustrated with the following example. Packet starts from node 0110101 and the tag is $0000000 - 00$ (the last two bits have special meaning and are separated with a dash in this example). If the packet travels up the tree to node 011, the tag becomes $000 - 10$, since the packet just left the left subtree. If the packet now enters the right subtree and travels to node 01110, the tag is modified to $00010 - 00$. If the packet now leaves the subtree 01 by entering node 00, for example, the tag is modified to $01 - 00$

The real network packets do not have to carry the tags with them. The tags will instead be associated with the entries in the routing tables, and the nodes will be able to check efficiently if the advertised paths do not form loops.

5 Merging two networks

We assume that each network has its own unique identifier which is passed from node to node at the moment of address allocation. The identifier can for example be a string representing the time and place of creation of the first node in the network.

When two networks with different identifiers meet a prefix 0 is added to one of them and a prefix 1 to the other. This operation preserves the invariant that a local group of nodes shares common prefix and that addresses in the network are unique. Next the routing tables need to be updated. This is done by adjusting the entries in all routing tables of one network by adding a prefix 0 and in the other by adding a prefix 1. At the end an advertisement of the entries $0*$ and $1*$, 0 and 1 must be transmitted in the respective subnets.

```
Procedure DetectMerge()
  Let l ∈ L
  if (l.NetworkID != myNetworkID)
    Greet(l)
  endif
  if (l.NetworkID < myNetworkID)
    broadcast order to extend addresses with 0 and to change
    network ID to l.NetworkID
  else
    broadcast order to extend addresses with 1
  endif
```

6 Splitting the network

When a set of nodes looses the connection to certain subnets it may be necessary to split the network into two smaller networks. When a set of nodes disconnects from the subnet $0*$ we generate a new network identifier and remove prefix 1 from all nodes of the subnet. The routing tables are also updated by removing the blank entries $0*$ and 0. Similar updates are necessary if the nodes loose the connection with subnet $1*$.

```
Procedure DetectSplit()
  if (RteTable[!add & 1] == ϵ for T time units)
    generate new NetworkID and broadcast it with the order to
    shift addresses by 1 and to change network ID
  endif
```

7 Address lookup

As said before, the identifiers and the addresses of the nodes in our architecture are separated. This introduces the problem of finding the address of a node when all we know is its identifier.

The general idea for the address lookup is very simple. All nodes use a common hashing function $\mathcal{H}$ that based on the identifier of a node determines the address of a node that keeps the current dynamic address of the node with the given identifier. The function $\mathcal{H}$ obviously cannot depend on the address of a node, only on its identifier.

The use of the function $\mathcal{H}$ is illustrated by the following example (Fig. 2). A node with address $addA$ and identifier $iddA$, transmits to the node with address $\mathcal{H}(iddA)$, its address $addA$. When another node wants to contact the node with identifier $iddA$, it contacts the node with address $\mathcal{H}(iddA)$ first, and retrieves the current address of $iddA$.

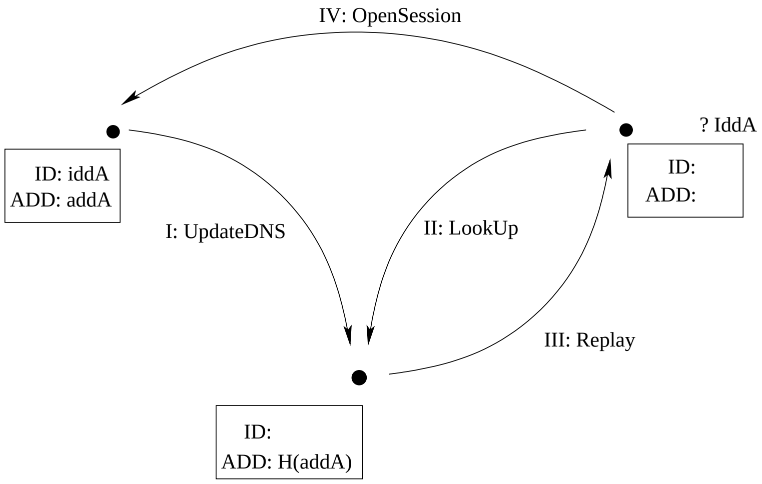

Figure 2: Address lookup. The procedure is carried out in three phases. I: the update of the DNS; II: lookup of the address; III: the reply of the DNS; IV: the start of the communication session

A problem may arise when there is no node with address $\mathcal{H}(iddA)$. It may be solved by introducing another argument of the function $\mathcal{H}$ - the number of iterations. When a node tries to retrieve the address of the node $iddA$ for the first time, it tries to contact $\mathcal{H}(iddA, 1)$. If this fails after a number of attempts, the node contacts $\mathcal{H}(iddA, 2)$, and so on, until some limit of iterations is achieved.

The challenge here is to design a hashing function $\mathcal{H}$ such that the dynamic addresses are distributed evenly throughout the network. An additional problem is that the identifiers of the nodes may be variable length.

We propose to use the following approach that does not require the node to send multiple DNS requests. We assume that the identifier is represented by a binary string. First, the identifier is run through a function h. The function h is a hashing function known to all nodes that produces a random string using its argument as a seed. The value produced by h is the same every time the same seed is used. Now the address of the DNS is the node which address makes the longest prefix of the value of h.

This still does not guarantee that the DNS is defined uniquely. Consider an example when a node chooses its DNS to be 0110, and the last existing hop on the route to that node is the node 0111 which will become the DNS. However, if another node sends a request to 0110 its packet may get as far as 0111, but it may also stop at 01110 which does not have the DNS entry for the initial node. To avoid this situation the last node on the path to the original DNS, will redefine the DNS to be the longest common prefix of the original DNS and the node's address. In the example above this will be 011.

The node initially sends the packet to the address $h(iddA)$. The packet is routed through the subnets by matching the leading bits of the destination. When the maximum number of bits has been matched (and the current node cannot forward the packet any further) it reroutes the packet using the longest common prefix procedure. When the address of the DNS matches the current node's address it becomes the DNS for the source of the packet.

The following procedures are run when the DNS needs to be updated.

Procedure UpdateDNS
  Send (myID, myAdd) to H(myAdd)

Procedure ForwardDNSUpdate
  if (Update.dest == myAdd)
    become the DNS
    return
  end if
  if (next hop on path to Update.dest exists)
    Forward the packet to the next hop
  else
    Update.dest = longest common prefix(myAdd, Update.dest)
    Keep forwarding
  endif

This procedure is used to forward a packet to the appropriate DNS

Procedure ForwardDNSReq
  if (Req.dest == myAdd)
    return DNS[Req.ID]
    return
  end if
  if (next hop on path to Req.dest exists)
    Forward the packet to the next hop
  else
    Req.dest = longest common prefix(myAdd, Req.dest)
    Keep forwarding
  endif

8 Conclusion

We propose a variable-length addressing scheme for mobile ad hoc networks. Our methods allow MANETs to be scalable with a reduced overhead (relative to static addressing schemes). We have detailed efficient algorithms for: 1) Variable-length address assignment, including conflict resolution. 2) Loop-free routing of packets. 3) Address lookup scheme where the nodes acting as the address resolver are uniformly distributed across the network.

Future work should include evaluation of the architecture with respect to other existing schemes, identifying bottlenecks and improvements. Considering address tree balancing may be of interest. It would also be interesting to analyze the performance of the protocols analytically to provide additional support for our claims.

References

[1] J. Boleng. Efficient network layer addressing for mobile ad hoc networks. In *Proceeding of International Conference on Wireless Networks (ICWN'02)*, pages 271–277, 2002.

[2] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva. Multi-hop wireless ad hoc network routing protocols. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'98)*, pages 86–97, 1998.

[3] S. Corson and J. Macker. Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations. RFC 2501, 1999.

[4] J. Eriksson, M. Faloutsos, and S. Krishnamurthy. Scalable ad hoc routing: The case for dynamic addressing. In *INFOCOM*, 2004.

[5] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.