

# Landmark Routing Algorithms: Analysis and Simulation Results

Paul F. Tsuchiya  
Ron Zahavi

March 1990

MTR-89W00277

SPONSOR:  
Defense Communications Agency  
CONTRACT NO.:  
F19628-89-C-0001

This document was prepared for authorized distribution.  
It has not been approved for public release.

The MITRE Corporation  
Washington C<sup>3</sup>I Division  
7525 Colshire Drive  
McLean, Virginia 22102-3481

MITRE Department  
and Project Approval:

Robert K. Miller A.



## ABSTRACT

This paper is the third in a series of papers that document the research, development, specification, implementation, and deployment of a new routing technique called Landmark Routing. Landmark Routing is a distributed and adaptive hierarchical routing protocol for use in arbitrarily large networks and internets. Its primary features are that it is robust and durable in the face of rapid topological changes, that it is easy to administer, and that it provides name-based addressing. In this paper, two of the three major algorithms that make up Landmark Routing were simulated. The algorithms resulting from the simulations are specified. The algorithms are analyzed and the simulation results are presented. The conclusion from this work is that Landmark Routing still appears to be a viable technology, and that implementation and testing should commence.

**Suggested Keywords:** Routing, Hierarchical Networks, Hierarchies, Landmark Routing, Landmark Hierarchy, Addressing, Naming, Address Binding, Packet-Switching, Data Communications, Simulation

## ACKNOWLEDGMENTS

The authors would like to thank Richard Wilmer for his painstaking editing of this rather thick document.

## TABLE OF CONTENTS

SECTION	PAGE
List of Figures	xi
List of Tables	xiii
EXECUTIVE SUMMARY	xv
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Scope of this Document	1
1.3 Outline	2
2 LANDMARK ROUTING OVERVIEW	3
2.1 The Landmark Hierarchy	3
2.1.1 The Landmark	3
2.1.2 The Landmark Hierarchy	3
2.1.3 Routing Table	5
2.1.4 Addressing in a Landmark Hierarchy	6
2.1.5 Routing in a Landmark Hierarchy	6
2.1.6 Landmark Hierarchy Example	7
2.2 Landmark Routing Algorithms	7
3 SIMULATION ENVIRONMENT	11
3.1 OPNET Description	11
3.1.1 Protocol Development	12
3.1.2 Node Development	12
3.1.3 Network Development	13
3.1.4 Simulation Development and Execution	13
3.1.5 Simulation Analysis	13
3.2 Landmark Routing Node Design	13
3.2.1 OPNET Node Design	15
3.2.2 Software Modularity	17
3.2.3 Other Simulation Aspects	17
3.3 Landmark Routing Network Design	18
3.4 Running Simulations	18
4 CONFIGURATION, FORWARDING, ROUTING, and HOST ALGORITHMS	21
4.1 Configuration Algorithm	21
4.2 Forwarding Algorithm	21

## TABLE OF CONTENTS (Continued)

SECTION	PAGE
4.2.1 Forwarding Data Packets	22
4.3 Routing Algorithm	24
4.4 Host Algorithm	26
<b>5 HIERARCHY ALGORITHM DESCRIPTION</b>	<b>27</b>
5.1 Landmark Hierarchy Structure	27
5.2 High-level Hierarchy Algorithm Description	29
5.3 Detailed Hierarchy Algorithm Description	31
5.3.1 Information Needed by Hierarchy Algorithm	31
5.3.2 Messages Exchanged by the Hierarchy Algorithm	33
5.3.3 Detailed Hierarchy Algorithm Description	34
5.4 Algorithms We Tried But Didn't Like	42
5.5 Algorithms We Didn't Try But Would Like to Use	42
<b>6 HIERARCHY ALGORITHM PERFORMANCE</b>	<b>47</b>
6.1 Hierarchy Algorithm Analysis	47
6.1.1 Analysis of Hierarchy Changes due to Topology Changes	47
6.1.2 Analysis of Traffic Generated by Hierarchy Changes	49
6.1.3 Analysis of Convergence Time	51
6.2 Static Simulation Results	52
6.3 Dynamic Hierarchy Algorithm Simulations	54
6.3.1 Hierarchy Algorithm Simulation Description	56
<b>7 BINDING ALGORITHM DESCRIPTION</b>	<b>69</b>
7.1 Overview	69
7.2 Detailed Description	72
7.2.1 Host Server Description	72
7.2.2 Resolution Server Description	78
7.2.3 Binding Server Description	82
<b>8 BINDING ALGORITHM PERFORMANCE</b>	<b>85</b>
8.1 Binding Algorithm Analysis	85
8.1.1 Memory Usage	85
8.1.2 Number of Messages	86
8.2 Dynamic Binding Algorithm Simulations	88
8.2.1 Binding Algorithm Simulation Description	88
<b>9 CONCLUSIONS</b>	<b>95</b>

## TABLE OF CONTENTS (Concluded)

SECTION	PAGE
9.1 Future Work	95
Appendix A: Glossary of Mathematical Expressions	97
References	99
Glossary	101

## LIST OF FIGURES

FIGURE NUMBER		PAGE
1	A Single Landmark	4
2	Landmark Hierarchy	5
3	Landmark Routing Example	8
4	Landmark Routing Node: Functional Diagram	14
5	OPNET Node Design	16
6	Landmark Hierarchy Tree Structure	28
7	Relationship of Hierarchy Algorithm to Other Algorithms	29
8	High-Level Landmark Algorithm Flow Diagram	30
9	Detailed Hierarchy Algorithm Flow Diagram	35
10	Child Algorithm	43
11	Estimated Hierarchy Changes	50
12	Effect of Network Type on Hierarchy Changes	53
13	Impact of Randomness on Changes	54
14	Effect of New Parent Distance Threshold on Hierarchy Changes	55
15	Effect of Number of Nodes on Hierarchy Changes	55
16	Effect of Number of Children on Hierarchy Changes	56
17	Effect of Multiple Topology Changes on Hierarchy Changes	57
18	Comparison of Experimental and Analytical Results	58
19	Convergence Time due to Routing Algorithm Update Speed	61
20	Convergence Time due to Routing Algorithm Update Speed	62

## LIST OF FIGURES (Concluded)

FIGURE NUMBER		PAGE
21	Convergence Time and Packets Lost by Number of Children	64
22	Convergence Time and Packets Lost by Network Type	65
23	Convergence Time by Landmark Level	66
24	Number of Packets Lost by Landmark Level and Number of Failures	66
25	Convergence Time by Number of Failures	67
26	Hierarchical Resolution of Addresses Derived Through Hash Function	71
27	Intermediate Hash Space Allows for Even Distribution of Bindings	72
28	How to Handle Frequent Queries	73
29	Binding Algorithm Architecture	74
30	Host Server Algorithm	77
31	Resolution Server Algorithm	80
32	Update or Query Resolution Example	81
33	Binding Server Algorithm	83
34	Binding Performance by Number of Updates and Routing Algorithm Speed	91
35	Convergence Time and Number of Lost Packets by Scenario	93

## LIST OF TABLES

TABLE NUMBER	PAGE
1 Routing Table for Node g of Figure 3	8
2 Data Packet Information	22
3 Hierarchy Algorithm State Variables	32
4 Hierarchy Algorithm Messages	34
5 Hierarchy Algorithm Wait States	37
6 Hierarchy Change: Usual Sequence of Events	51
7 Modifiable Hierarchy Algorithm Simulation Parameters	59
8 Hierarchy Algorithm Simulation Parameter Combinations	60
9 Messages Sent and Received by the Binding Server	75
10 Host Server Data	76
11 Resolution Server Data	79
12 Binding Server Data	82
13 Distribution of HN Bindings Among N Nodes	86
14 Modifiable Binding Algorithm Simulation Parameters	89



## EXECUTIVE SUMMARY

### INTRODUCTION

This paper is the third in a series of papers that document the research, development, specification, implementation, and deployment of a new routing technique called Landmark Routing. Landmark Routing is a distributed-adaptive hierarchical routing protocol for use in arbitrarily large networks and internets. Its primary features are that it is robust and durable in the face of rapid topological changes, that it is easy to administer, and that it provides full name-based addressing. These features arise from the fact that Landmark Routing dynamically establishes its own hierarchy and modifies it as the network undergoes changes. This is the first routing protocol with this capability.

In addition, Landmark Routing has features designed to facilitate its operation in the existing Department of Defense (DoD) Advanced Research Projects Agency (DARPA) and emerging International Organization for Standardization (ISO) internet environments. In particular, it embraces the concept of separately administered networks that require some level of autonomy and protection from each other.

The first two papers (Tsuchiya, 1987a and 1987b) analyze the Landmark Hierarchy and describe possible algorithms for use with the Landmark Hierarchy. In this paper, we present a detailed description and simulation results of two of the algorithms discussed in the second paper—the Hierarchy Algorithm, and the Binding Algorithm (Assured Destination Binding (ADB)). We had hoped to also simulate Alternate-path Distance-vector Routing (ADR), but did not complete that part of the project. Instead, we use the distance-vector routing algorithm used in the SURAN packet radio network called Tier Routing (Westcott and Jubin, 1982; Westcott, 1982) as the lower-level routing algorithm.

### THE LANDMARK HIERARCHY

Landmark Routing is a hierarchical routing scheme based on the Landmark Hierarchy. The Landmark Hierarchy is different from the traditional area hierarchy. In the area hierarchy, nodes are grouped into areas so that 1) all nodes in an area have a routing entry for all other nodes in the area, and 2) between any two nodes in the area there is a path that does not leave the area. Areas are then grouped into super-areas, and so on to form a hierarchy (the telephone network is a well-known example of an area hierarchy). Routers outside of an area view the area as a single entity, thus reducing the amount of routing information needed to address nodes in that area.

Whereas an area is a group of nodes *all of which have a routing table entry for each other*, a Landmark is a node for which a group of nodes *all have a routing table entry for that Landmark*. The group of nodes is determined by a radius  $r$  extending from the Landmark. Therefore, every node  $r$  hops away from the Landmark has a routing table entry for that Landmark. A hierarchy of Landmarks is formed by having all nodes be Landmarks with small radii, a portion of those nodes be Landmarks with larger radii, a portion of those be Landmarks with still larger radii, and so on until there are a few nodes network-wide whose radii covers the whole network. Whereas in the

area hierarchy a node is addressed by its membership in areas, a node in a Landmark Hierarchy is addressed by its proximity to Landmarks.

Because any given node falls outside the radii of most other nodes, its routing table is small. Source nodes route messages to destination nodes by directing the message towards the lowest-level Landmark in the destination node's address. Initially, a message may move towards a high-level Landmark, but as it nears the destination, it will veer towards lower and lower-level Landmarks, and finally to the destination itself.

## **HIERARCHY ALGORITHM DESCRIPTION**

The purpose of the Hierarchy Algorithm is to create the Landmark Hierarchy. For this, each node must determine its hierarchy level, its radius, and its Landmark Address. A high-level description of the Hierarchy Algorithm follows.

Upon power-up, each Landmark randomly chooses a hierarchy level and corresponding radius, large enough to cover a potential parent with high probability. The Landmark then both accepts children at lower levels, and waits to hear of a potential parent. If it hears a potential parent, it requests permission to be adopted by that parent. If the answer is yes, the Landmark reduces its radius appropriately (to just cover the parent), and picks either a global address, or an address that is under its parent's address tree. If the answer is no, the Landmark increments its level by one, increases its radius correspondingly, and tries to find a new potential parent. If either at initial boot time, or after incrementing its level, the Landmark sees no potential parent, it assumes that it is the root of the Landmark Hierarchy, and assigns itself a global address.

Once a Landmark has a parent (or is the root) and an address, it is in its steady state. It can choose a new parent if its old parent disappears, or if the new parent is much closer than the old parent. If the Landmark was not global, this will result in a new address. Even without getting a new parent, it can get a new address by becoming or ceasing to be global, or by its parent getting a new address.

The main features of the Hierarchy Algorithm are:

1. Random selection of hierarchy level. This is the key feature. By randomly selecting the hierarchy level (with decreasing probability at higher levels), we avoid elections, and the accompanying complexity and delay.
2. Modification of radius based on distance from parent. It is this feature that allows us the freedom to randomly select the hierarchy level. If the parent is far away, simply make the radius large.
3. Ability to choose a new parent, by incrementing the hierarchy level if necessary.
4. Ability to become or cease being global.

## BINDING ALGORITHM DESCRIPTION

The problem to be solved by the Binding Algorithm is that since the address of any node can change at any time, the source of a packet must discover the current address of the destination given the name (or ID) of the destination. Of course, the source cannot ask the destination its address, since it would need to already know the destination's address to do this. Therefore, it is necessary for there to be some third party—the name server—that can be found by both the source and the destination. The destination gives its name server its current address (update) and the source asks the name server for this address (query).

We want the Binding Algorithm to be robust and efficient. Flooding—either updates or queries—is robust but not efficient. Creating a name server hierarchy creates potential bottlenecks and single points of failure, and doesn't eliminate flooding since the name servers must flood their current addresses, which of course can change as easily as any other node's.

In our Binding Algorithm, called Assured Destination Binding, we have, for robustness, every node (Landmark) act as a name server. Therefore, the routing table is at least a partial view of the set of potential name servers (partial because of the hierarchy). To determine the correct name server for any given named destination, we hash the name into the address space, resolve the resulting address to one of the entries in the routing table, and send either update or query to that address. If the routing table changes, then certain bindings must be re-resolved and new updates sent out. The resolution works by mapping the address derived from the hash function into the next highest real address (in the routing table).

This algorithm is efficient because updates and queries are not flooded. It is robust because the routing table, which is itself robust, is the basis for discovering the correct name server for any given destination.

Because the routing table is hierarchical, the address derived from the hash function cannot normally be fully resolved by the source of the update or query. This is because the resolution does not work properly unless each node that might do the resolution agrees on the set of addresses to be resolved to. Therefore, the derived address must be resolved one level of the hierarchy at a time, starting from the global level.

In other words, the source of the update or query 1) hashes the name, producing the derived address, 2) maps the derived address into the set of global Landmarks, and 3) sends the update or query towards the chosen global Landmark. When the update or query reaches one of the offspring of the chosen global Landmark, that node is able to resolve the update or query to one of the children of the global Landmark, and sends the update or query to that child. The first offspring of that child that receives the query or update further resolves it to one of its children, and so on. Eventually, the update or query will reach the appropriate name server, regardless of the source of the update or query.

Hashing the names into addresses evenly distributes the derived addresses among the address space. However, since addresses themselves may be clustered, it is also necessary to hash the entries of the routing table (the addresses) into what we call an Intermediate Hash Space (IHS).

Names are then hashed into the IHS rather than directly into the address space, and resolved to the next highest entry in the IHS, which is mapped back into an entry in the routing table. There will be one IHS for each level of the hierarchy.

Routing table entries will be hashed into the IHS multiple times in order to provide a still better distribution of bindings. This is necessary because, since there are a small number of entries at any single hierarchical level, the pseudo-random variation in the hash function can cause an uneven distribution of IHS entries. Multiple hashes can be accomplished by appending a byte to the hash input (the address itself) and incrementing the value in that byte for each additional hash.

To increase robustness and efficiency, several name servers are provided for each binding. This is more efficient because 1) queries can go to the nearest name server (this is only more efficient if there are substantially more queries than updates), and 2) because the period for sending out updates can be longer since the period is based on the probability that all servers for any given binding have crashed. Multiple name servers can be chosen using the same technique as creating multiple IHS entries—append a byte to the name input to the hash function, and increment the byte for each additional server.

While the hash function results in an even distribution of bindings, it doesn't guarantee that all name servers will have a similar workload. This is because certain popular destinations will have more queries generated for them, and those queries will be directed towards a few name servers. To spread this load around, when a name server finds that it is receiving excessive queries for one of its destinations, it gives the binding to the neighbors it is receiving the bindings from. The neighbors will then begin to answer the bindings, and can themselves spread the binding to their neighbors if necessary. The more queries that are generated for a particular destination, the more adjacent servers that will receive the binding and answer the queries.

Once a binding has been handed to a neighbor, it must be kept current for that neighbor as well. We call the original holder of the binding the primary name server, and the subsequent holders of the binding the adjacent name servers. When the primary name server receives the periodic update, or a change for an existing binding, it must tell its adjacent name servers. If an adjacent name server finds that it is not receiving many queries for a binding it is holding, it can choose to no longer hold the binding, and must tell the primary (or up-tree adjacent) name server.

When a node gets a new address, it sends out new updates for its attached hosts. It also determines which of the bindings it is holding no longer resolve to it and informs the sources of those bindings that they need to send out new updates. This way, all address changes result in immediate updating of the bindings. When a node crashes, it obviously cannot inform the source that new bindings must go out, and so bindings are sent out periodically to insure that bindings are up to date in the event of crashes.

The Binding Algorithm is analyzed and simulation results are presented. This analysis focuses on packet overhead and convergence time. We find that the packet overhead is larger than that for the Hierarchy Algorithm, and can be significant when high-level Landmarks crash. Nevertheless, the packet overhead even for worst-case scenarios is not prohibitive. The convergence time is also mainly dependent on the convergence speed of the lower-level routing

algorithm. In addition, the count-to-infinity problem has a bad effect on the Binding Algorithm in that IHS tables can be out of sync for the duration of the counting-to-infinity. Again, ADR is necessary to speed convergence, and to remove the count-to-infinity problem. There are other algorithms that eliminate the count-to-infinity problem, but unfortunately at the expense of fast convergence. The simulations show that the Binding Algorithm behaves as expected. Several suggestions for improvement are made.

## CONCLUSIONS

The main conclusion to draw from this paper is that the algorithms developed so far seem workable and that the next phase of the project, implementation and testing, should begin. Because of time and resource constraints, the simulations were not as thorough as they should have been. We could have run more simulations to give us statistically more solid data, and we could have run more complex scenarios, thus stressing the algorithms more. Even so, the simulations have given good insights into the behaviour of the algorithms, and have helped in the design of good algorithms. Furthermore, the analysis of the completed algorithms also give insights into the expected performance of the algorithms. The implementation and testing phases will serve to stress the algorithms.

In addition to implementing the algorithms discussed in this paper, simulation needs to be done on the Alternate-path Distance-vector Routing (ADR) algorithm sketched out in the previous paper (Tsuchiya, 1987b). More simulation is needed for the Binding Algorithm, in particular simulating with a hierarchy. Finally, simulation with administrative boundaries is still required. Of these, the major part is simulating ADR. The other simulations are relatively simple extensions of the algorithms already simulated.

The major work to be done, however, is implementation and testing in a test bed. This work will be performed during the next few years by the University of Maryland under DARPA contract.

## 1 INTRODUCTION

This paper is the third in a series of papers that document the research, development, specification, implementation, and deployment of a new routing technique called Landmark Routing. Landmark Routing is a distributed-adaptive hierarchical routing protocol for use in arbitrarily large networks and internets. Its primary features are that it is robust and durable in the face of rapid topological changes, that it is easy to administer, and that it provides full name-based addressing. These features arise from the fact that Landmark Routing dynamically establishes its own hierarchy and modifies it as the network undergoes changes. This is the first routing protocol with this capability.

In addition, Landmark Routing has features designed to facilitate its operation in the existing Department of Defense (DoD) Advanced Research Projects Agency (DARPA) and emerging International Organization for Standardization (ISO) internet environments. In particular, it embraces the concept of separately administered networks that require some level of autonomy and protection from each other.

### 1.1 Motivation

This work is supported by the Defense Communications Agency (DCA), and specifically by the Defense Communications System Data Systems organization, which manages the Defense Data Network (DDN). There are two similar problems which have motivated this work.

The DDN consists of several long-haul packet switching networks for use by Department of Defense (DoD) subscribers. These networks are growing, and are scheduled to merge in the future. The size of these networks is becoming such that the efficiency of the existing non-hierarchical routing (Shortest Path First (SPF)) is questioned (Sparta, 1986), (Khanna and Seeger, 1986).

An even more severe problem is the growth of the Internet (the DDN plus connected networks, such as the NSFNET). This growth is greater than that of the DDN alone, and is now pushing the limits of the existing gateway routing protocols, for instance the Gateway-to-Gateway Protocol (GGP), and the Exterior Gateway Protocol (EGP). A further problem here is that, unlike SPF, the gateway routing protocols are not very robust.

Finally, Landmark Routing is attacking both the problems associated with separately administered networks, and those of address administration.

### 1.2 Scope of this Document

The first paper on Landmark Routing is "*The Landmark Hierarchy: Description and Analysis*" (Tsuchiya, 1987a). This paper studies the Landmark Hierarchy alone—that is, without any regard to its use in a dynamic environment. The point of that paper is to determine the efficiency of the hierarchy in terms of the routing table sizes and path lengths. The whole idea behind the use of any hierarchy in routing is to reduce the amount of routing information that must be spread around. The performance penalty paid for this reduction is increased path length. A

hierarchy is overall beneficial if the reduction in overhead from decreased routing information outweighs the increase in overhead from longer paths. If this benefit doesn't exist, then there is no reason to have the hierarchy at all. The first paper shows that the Landmark Hierarchy provides this benefit. In particular, it shows that the routing table sizes and path lengths are slightly better than those seen in area hierarchies.

The second paper (Tsuchiya, 1987b) is a medium to high-level design covering all aspects of Landmark Routing—dynamic management of the hierarchy, the routing algorithms, name-to-address binding, administrative boundaries, and implementation in existing networks. It explores the feasibility of accomplishing all of the goals set for Landmark Routing by stating how each goal can be accomplished. In particular, it concludes that the various algorithms that make up Landmark Routing appear promising.

In this paper, we present a detailed description and simulation results of two of the algorithms discussed in the second paper—the Hierarchy Algorithm, and the Binding Algorithm (Assured Destination Binding (ADB)). We had hoped to also simulate Alternate-path Distance-vector Routing (ADR), but did not complete that part of the project. Instead, we use the distance-vector routing algorithm used in the SURAN packet radio network called Tier Routing (Westcott and Jubin, 1982; Westcott, 1982) as the lower level routing algorithm.

### 1.3 Outline

Section 2 gives an overview of the Landmark Hierarchy. This information is also found in the previous two papers. Section 3 describes the simulation environment within which we tested the algorithms. Section 4 describes the algorithms used with the Hierarchy and Binding Algorithms, but which were not the object of our study. These include neighbor configuration, forwarding, routing, and host algorithms. Section 5 describes in detail the Hierarchy Algorithm. This is the algorithm that forms the Landmark Hierarchy and assigns addresses. It is worth noting that the algorithm described here is different from that described in the second report. Section 6 gives an analysis of the Hierarchy Algorithm, and presents and discusses simulation results. Section 7 describes in detail the Binding Algorithm. This is the algorithm that discovers the current address of named destinations. Section 8 gives an analysis of the Binding Algorithm, and presents and discusses simulations results. Finally, Section 9 concludes that the algorithms seem workable and that it is time to begin implementation and simulation.

## 2 LANDMARK ROUTING OVERVIEW

In this section, we first discuss the Landmark Hierarchy, and the basic idea behind Landmark Routing. We then discuss at a high level the algorithms required to make Landmark Routing work in a dynamic environment.

### 2.1 The Landmark Hierarchy

We first describe the Landmark itself. Then, we describe a hierarchical structure built from Landmarks. Third, we describe how nodes are addressed in a Landmark hierarchy. Finally, we show how routing may take place with the Landmark hierarchy. This description is available in the previous work (Tsuchiya, 1987a), but is repeated here for completeness.

#### 2.1.1 The Landmark

The description of a Landmark is simple. A Landmark is a node whose neighbor nodes within a certain vicinity contain routing entries for that node. Determination of the vicinity is based on hops; that is, the distance between any two nodes that share a link is measured as one.<sup>1</sup>

As an example, consider Node 1 in the network of Figure 1. Nodes 2 through 6 have routing entries for Node 1 (as indicated by the arrowheads) and are therefore able to forward any packets addressed for Node 1 to Node 1. Nodes 7 through 11 do not contain routing entries for Node 1. Therefore, Node 1 is a Landmark which can be "seen" by all nodes within a distance of 2 hops. We refer to Node 1 as a Landmark of radius 2. This is distinguished from an area in the area hierarchy, where the nodes in a given area *all have routing table entries for each other*.

#### 2.1.2 The Landmark Hierarchy

Next, let us consider a hierarchy built from Landmarks. The nomenclature  $LM_i$  refers to a Landmark of hierarchy level  $i$ ,  $i=0$  being the lowest level, and  $i=H$  being the highest level. Throughout this paper, the subscript  $i$  is reserved to mean a hierarchy level. The nomenclature  $LM_i[x]$  refers to a level  $i$  Landmark with label  $x$ .

Each  $LM_i$  has a corresponding radius  $r_i$ . In the Landmark hierarchy, every node in a network is a Landmark  $LM_0$  of some small radius  $r_0$ . Some subset of  $LM_0$ 's are  $LM_1$ 's with radius  $r_1$ , and with  $r_1$  almost always greater than  $r_0$ , so that there is at least one  $LM_1$  within  $r_0$  hops of each  $LM_0$ . Likewise, a subset of the  $LM_1$ 's are  $LM_2$ 's, with  $r_2$  almost always greater than  $r_1$ , so that there is at least one  $LM_2$  within  $r_1$  hops of each  $LM_1$ .<sup>2</sup>

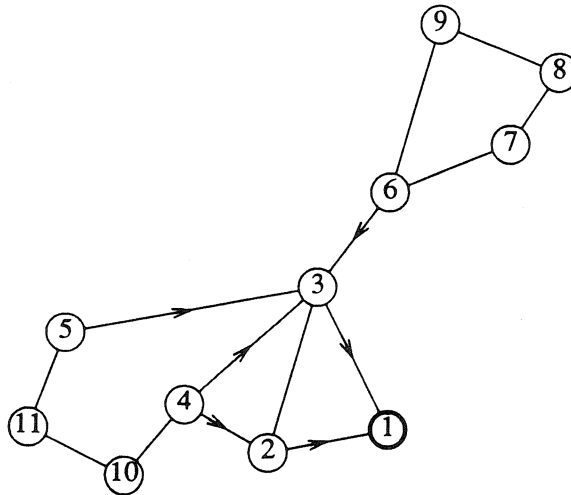
---

<sup>1</sup>We also require that all links be full duplex. The class of routing algorithm required for Landmark Routing (Distance-Vector) does not work with simplex links (see Section 3.2). We do, however, have techniques to account for link metrics other than hops, and for different metric values for each direction of a full duplex link. This requires that multiple routing algorithms be run in parallel.

<sup>2</sup>Note that not all  $r_i$  are necessarily equal. In other words,  $r_i[x]$  may or may not be equal to  $r_i[y]$ .



Figure 1  
A Single Landmark



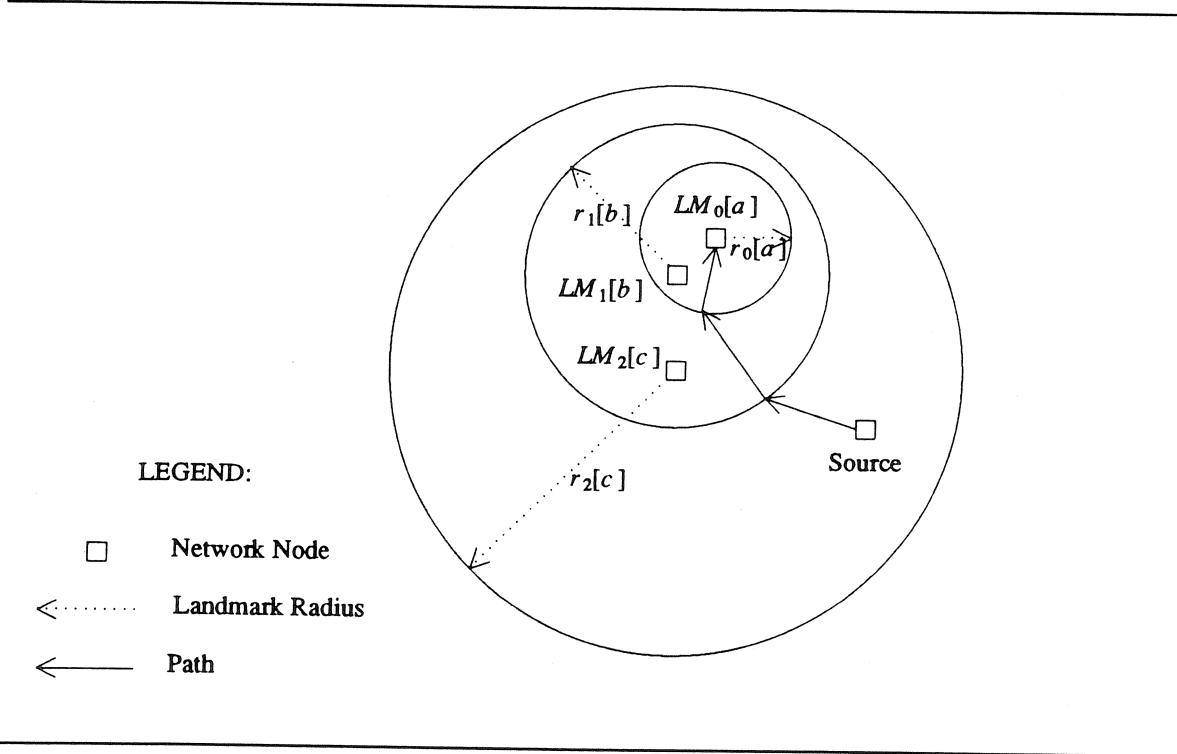
These iterations continue until a small set of nodes are  $LM_i^G$ 's each with an  $r_i^G$ , with  $r_i^G \geq D$ ,  $D$  being the diameter of the network. Because the radius of these Landmarks is larger than the diameter of the network, all nodes in the network can see these Landmarks. We call these global Landmarks, and give them the superscript  $G$ . The reason for this structure will become clear in Section 2.1.5.<sup>3</sup>

Figure 2 illustrates the Landmark hierarchy by showing a portion of a network. This is a two-dimensional representation (meaning that only nodes drawn physically close to each other would share a link). For simplicity, only four of the nodes are shown, and no links are shown. The dotted arrows and circle indicate the radius of the Landmarks; that is, the vicinity within which nodes contain routing entries for that Landmark. For instance, every node within the circle defined by  $r_1[b]$  has an entry for, and can route to,  $LM_1[b]$ . Since a node at level  $i$  is also a Landmark at all levels  $x < i$ , it will have Landmark Labels for each level. Again for simplicity, the nodes in Figure 2 are labeled only with the Landmark Labels which are pertinent to the examples herein.

<sup>3</sup>The network diameter is the distance between the two nodes in the network furthest from each other.

In general, Landmark Labels only need to be locally unique, except at the highest level. The requirements for local uniqueness are discussed in Section 5.

Figure 2  
Landmark Hierarchy



### 2.1.3 Routing Table

Each node in the network keeps a table of the next hop on the shortest path to each Landmark for which it has routing entries. Each node will therefore have entries for every  $LM_0$  within a radius of  $r_0$ , every  $LM_1$  within a radius of  $r_1$ , and so on.

Since every node is an  $LM_0$ , and since every node has entries for every  $LM_0$  within a radius of  $r_0$ , every node has full knowledge of all the network nodes within the immediate vicinity. Likewise, since a portion of all  $LM_0$  are  $LM_1$ , every node will know about some of the network nodes further away. Similarly, each node will have knowledge of even fewer nodes further still, and so on. The end result is that all nodes have full local information, and increasingly less information further away in all directions. This can be contrasted with the area hierarchy where a node on the border of an area may have full local information in the direction within the border, but virtually no local information in the direction across the border.

### 2.1.4 Addressing in a Landmark Hierarchy

In an area hierarchy, the address of a node is a reflection of the areas at each hierarchical level in which the node resides. The telephone number is a well-known example of this. In a Landmark hierarchy, the address of a node is a reflection of the Landmark(s) at each hierarchical level which the node is near. The Landmark Address (or just Address, for short), then, is a series of Landmark Labels:  $LM_G^G[x_G].LM_{i-1}[x_{i-1}]. \dots .LM_0[x_0]$ .

There are two constraints placed on Landmark Addresses. First, the Landmark represented by a given Landmark Label must be within the radius of the Landmark represented by the next lower Landmark Label. For instance, the node labeled  $LM_0[a]$  in Figure 2 may have the Landmark Address  $LM_2[c].LM_1[b].LM_0[a]$ . In this case, we call  $LM_2[c]$  a parent of  $LM_1[b]$ , and we call  $LM_1[b]$  a child of  $LM_2[c]$ . In this paper, the terms parent and child will always refer to two Landmarks, the lower of which is using the higher as part of its address. The address of the node labeled  $LM_0[a]$  could be  $LM_2[c].LM_1[e].LM_0[a]$  if and only if there existed a Landmark  $LM_1[e]$  (not shown) which was within the the radius of the node labeled  $LM_0[a]$ . The reason for this constraint will become clear in Section 2.5. Since more than one Landmark may be within the radius of a lower level Landmark, nodes may have many unique addresses. Multiple addresses could be used to provide some traffic splitting.

Now, the set of nodes that contain a routing table entry for a Landmark is not the same as the set of nodes that use that Landmark for their Landmark Address. In general, the set of nodes that contain a routing table entry for a Landmark is larger than the set of nodes that use that Landmark for their Landmark Address. The former set overlaps with analogous sets for other Landmarks (in other words, a node will typically have routing table entries for several Landmarks at a particular level), while the latter set usually does not overlap with other analogous sets (in other words, the Address tree is a strict tree).

### 2.1.5 Routing in a Landmark Hierarchy

Now we may consider how routing works in a Landmark Hierarchy. Assume we want to find a path from the node labeled *Source* to the node labeled  $LM_0[a]$  in Figure 2. The Landmark Addresses for the node labeled  $LM_0[a]$  is  $LM_2[c].LM_1[b].LM_0[a]$ . The basic approach is the following: *Source* will look in its routing tables and find an entry for  $LM_2[c]$  because *Source* is within the radius of  $LM_2[c]$ . *Source* will not, however, find entries for either  $LM_1[b]$  or  $LM_0[a]$ , because *Source* is outside the radius of those Landmarks. *Source* will choose a path towards  $LM_2[c]$ . The next node will make the same decision as *Source*, and the next, until the path reaches a node which is within the radius of  $LM_1[b]$ . When this node looks in its routing tables, it will find an entry for  $LM_1[b]$  as well as for  $LM_2[c]$ . Since  $LM_1[b]$  is finer resolution, the node will choose a path towards  $LM_1[b]$ . This continues until a node on the path is within the radius of  $LM_0[a]$ , at which time a path will be chosen directly to  $LM_0[a]$ . This path is shown as the solid arrow in Figure 2.<sup>4</sup>

There are two important things to note about this path. First, it is, in general, not the shortest possible path. The shortest path would be represented in Figure 2 by a straight line directly from

*Source* to  $LM_0[a]$ . This increase in path length is the penalty paid for the savings in network resources which the Landmark hierarchy provides. This will be analyzed in Section 4.

The other thing to note is that the path does not necessarily go through the Landmarks listed in a Landmark Address. This is more so if the Landmark vicinity for an  $LM_i$  goes beyond an  $LM_{i+1}$ . This is an important reliability consideration in that a Landmark may be heavily congested or down, and yet a usable path may be found using that Landmark (or, more literally, using previous updates received from that Landmark).

### 2.1.6 Landmark Hierarchy Example

To better illustrate the Landmark Hierarchy, consider Figure 3. This network has 3 hierarchical levels. All nodes (small circles) are  $LM_0$ . Diamonds denote  $LM_1$ , and large circles denote  $LM^G$  (again, the superscript  $G$  means that the Landmark is global). The rightmost address component is the  $LM_0$ , and for this example is unique for each node in the network. The middle address component is the  $LM_1$  and indicates proximity to an  $LM_1$ , and the leftmost address component is an  $LM^G$ , and indicates proximity to an  $LM^G$ . All  $r_0 = 2$  hops, all  $r_1 = 4$  hops, and all  $r_2 = 8$  hops.

Table 1 shows the routing table for Node  $g$  in Figure 3. This length of this table has been optimized by including only one entry per node, even if that node is a Landmark at several different levels. Node  $g$  has less than one-fourth of the total network nodes in its routing table.

Let's consider a routing example where Node  $g$  (with address  $d.i.g$ ) is routing a message to Node  $t$  (with address  $d.n.t$ ). Node  $g$  examines Node  $t$ 's Landmark Address— $d.n.t$ —and does not find entries for either  $LM_0[t]$  or  $LM_1[n]$  in its routing table. Node  $g$  does, however, have an entry for  $LM_2[d]$ , and therefore forwards the message towards  $LM_2[d]$  via Node  $f$ . Node  $f$  also does not have entries for  $LM_0[t]$  or  $LM_1[n]$ , and therefore forwards the message towards  $LM_2[d]$  via Node  $e$ . Node  $e$  does have an entry for  $LM_1[n]$  (but not  $LM_0[t]$ ), and forwards the message towards  $LM_1[n]$  via Node  $d$ . Node  $d$  does have an entry for  $LM_0[t]$ , as does Node  $u$ , and the message is delivered. The resulting path,  $g-f-e-d-u-t$ , is 5 hops, 1 hop longer than the shortest path,  $g-k-i-u-t$ .

## 2.2 Landmark Routing Algorithms

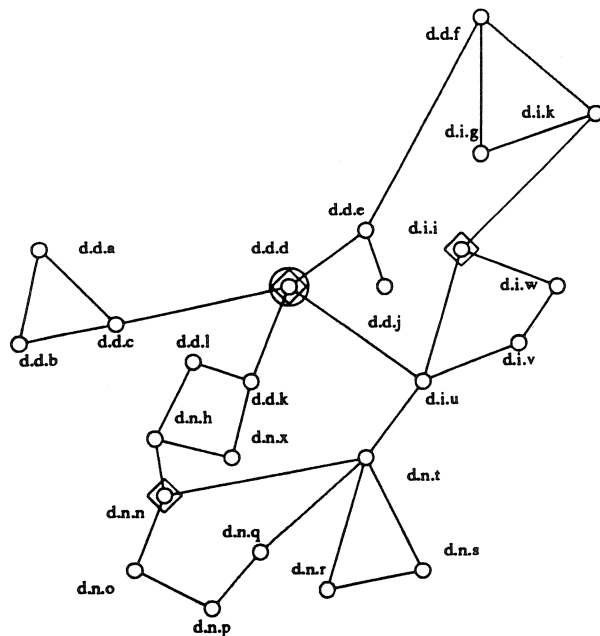
For Landmark Routing to work in a dynamic environment, three things, in this order, have to happen. First, the Landmark Hierarchy must configure itself and assign addresses to nodes in the network (Hierarchy Algorithm). Second, routing tables pointing to those Landmarks must be configured (Routing Algorithm). Finally, sources of packets must have some way of knowing what the current address of a destination is (Binding Algorithm).

The (simplified) sequence of events is as follows:

---

<sup>4</sup>In point of fact, *Source* might well see an  $LM_1[b]$ , because Landmark Labels are only locally unique with respect to their siblings. It would not, however, see an  $LM_2[c].LM_1[b]$ .

**Figure 3**  
**Landmark Routing Example**



**Table 1**  
**Routing Table for Node g of Figure 3**

Landmark	Level	Next Hop
$LM_2[d]$	2	f
$LM_1[i]$	1	k
$LM_0[e]$	0	f
$LM_0[k]$	0	k
$LM_0[f]$	0	f

When a node, say Node A, comes up, it chooses a Landmark Hierarchy level. In our simulation, we randomly pick a level with a probability appropriate for that level (probability 1 for level 0 or greater, probability  $\frac{1}{k}$  for level 1 or greater, probability  $\frac{1}{k^2}$  for level 2 or greater, and probability  $\frac{1}{k^h}$  for level h or greater, for some constant  $k$ ). In a real implementation, this would probably be based on the hash of the node's ID, so that the level it chooses could be externally

verified. Node A also picks an initial radius according to its level, large enough to cover its eventual parent with high probability. This is the first part of the Hierarchy Algorithm.

Next, the Routing Algorithm configures routing table entries for Node A in nodes within the radius chosen by Node A. This is done with a traditional distance-vector (Old ARPANET, Bellman-Ford, etc.) type routing algorithm, except that the routing information is not propagated beyond the number of hops indicated by the radius. In other words, Node A will tell its neighbors that it is zero hops from Node A. Nodes A's neighbors will create a routing table entry for node A, and tell their neighbors that they are one hop from Node A. Those neighbors will create routing table entries for Node A, and tell their neighbors that they are two hops from Node A, and so on. After some convergence time, every node within the radius of Node A will have a routing table entry pointing towards Node A. In addition, the routing algorithm will also cause Node A to know about other nodes, in particular zero or more nodes one level higher than it (a potential parent) in the Landmark Hierarchy.

If Node A sees a potential parent Landmark (after some time), it attempts to become a child of that Landmark. If Node A does not see a potential parent, it assumes itself to be the root of the Landmark Hierarchy. In any event, one or more nodes will become roots of the Landmark Hierarchy, and the rest of the nodes will establish parents. The roots then assign themselves Landmark Addresses, and all of their offspring adopt Landmark Addresses as well.

Once the Hierarchy Algorithm determines its Landmark Address, it tells the Binding Algorithm. It also tells the Binding Algorithm other information about the state of the Landmark Hierarchy—its direct ancestors and offspring. The Binding Algorithm then sends out binding updates for its hosts, and subsequently receives binding updates to store for other nodes' hosts.

When data packets arrive at a node from one of its hosts, the Binding Algorithm will send out a binding query if it does not already know the Landmark Address for the destination host. Another Binding Algorithm will answer the binding query, the data packet will be filled in with the appropriate Landmark Address, and the packet will be delivered.

The reason for presenting this simple description of the Landmark Routing algorithms is to prepare the reader for understanding Section 3, which describes, among other things, the node architecture used in the simulations. The main points are that 1) there are three algorithms, the Hierarchy Algorithm, the Routing Algorithm, and the Binding Algorithm, and 2) that these algorithms exchange information. In particular, the Hierarchy and Routing Algorithms both depend on information from the other, and the Binding Algorithm depends on information from the Hierarchy Algorithm.

### 3 SIMULATION ENVIRONMENT

To test and develop Landmark Routing, we implemented our algorithms on an event-driven simulation tool. We used a commercial simulation package called OPNET.<sup>5</sup>

OPNET satisfied the following simulation requirements:

1. The simulation environment must be realistic with respect to the underlying environment as seen by Landmark Routing. Because Landmark Routing is well decoupled from the physical characteristics of the network, this was not a difficult requirement to satisfy.
2. We must be able to simulate many nodes (at least 50) to exercise enough levels of the hierarchy.
3. We must be able to develop portable code for later testing in a real implementation. OPNET satisfied this requirement because it allows the user to code algorithms in C.
4. We must be able to decouple the various algorithms used by Landmark Routing for independent testing and modification.
5. The simulation results must be easily observed and analyzed.

In this section we first describe OPNET, and then describe the simulation environment we set up for testing our implementation on OPNET.

#### 3.1 OPNET Description

OPNET is a package for developing, running, and analyzing event-driven simulations of distributed communications systems. OPNET allows the user to do the following:

1. Define protocols, either by drawing a finite state machine, or using C language (or a combination of both).
2. Define nodes as a combination of user-built protocols and OPNET-provided modules such as traffic generators, queues, transmitters, receivers, antennas, and sinks.
3. Define networks by positioning nodes in 3-dimensional space, defining radio-coverage patterns for nodes and jammers, placing point-to-point links between nodes, and attaching nodes to broadcast LANs.

---

<sup>5</sup>OPNET is a trademark of MIL 3, Inc., Washington, D.C.

4. Run event-driven simulations on networks, during which nodes and jammers can move, radio characteristics can change, traffic patterns can change, and so on.
5. Analyze simulation outputs by plotting the outputs on graphs, and by running the output data through OPNET-defined and user-defined filters, such as averagers, differentiators, summers, and so on.

In the following descriptions of OPNET features, we mention for completeness both features we used and those we did not use.

### 3.1.1 Protocol Development

The user can define protocols either by drawing a finite state machine through an icon/mouse-based interface, or by using C language (or a combination of both). The finite state machine interface is useful when the protocol to be developed has already been defined in terms of a finite state machine. Within each state, the user specifies the actions to be executed in C language. OPNET takes the finite state machine representation and generates C code.

OPNET provides the user with a library of C functions which allow the protocol to interface with the simulation environment. Some of these functions place the protocol in what is essentially a communicating operating system environment. In other words, the user has a set of C functions that allow him to generate, modify, send, receive, and destroy packets, and to generate interrupts and be interrupted, just as would be seen in any real communications box. This is what allows us to write code that is easily portable to another system.

Other functions allow the protocol to modify the simulation environment, by changing node characteristics such as position or fault status, by changing radio characteristics, by writing simulation data to an output file, and by ending the simulation. OPNET also provides a library of routing functions.

In addition, the user has access to all C functions available in UNIX, since OPNET runs on UNIX.<sup>6</sup> One can write printf statements, can fork processes, and can define shared memory between protocols, even those in different nodes. This is very useful in circumventing the network to learn the state of other nodes for debugging purposes.

### 3.1.2 Node Development

The user can define nodes as a combination of user-built protocol modules and OPNET-provided modules such as traffic generators, queues, transmitters, receivers, antennas, and sinks. Nodes are defined through an icon/mouse-based interface. The node-building tool allows the user to architect a node, in terms of both software (by placing protocol modules and defining communications streams between them) and hardware (by placing transmitters, receivers, and antennas, and defining communications streams between them). It is this modularity in node

---

<sup>6</sup>UNIX is a trademark of AT&T Bell Laboratories.



design that allows us to decouple the various aspects of our algorithm so that each can be tested and modified separately.

The user can define transmitter and receiver characteristics such as bit error rate, bandwidth, frequency range, modulation type, spread spectrum, jammer type, and others. The user can also define antenna coverage patterns, transmitting power, and so on.

### **3.1.3 Network Development**

The user can define networks by positioning nodes in 3-dimensional space, defining radio-coverage patterns for nodes and jammers, placing point-to-point links between nodes, and attaching nodes to broadcast LANs. Networks can be built either through the icon/mouse-based user environment, or by writing a program in C language on UNIX that defines node placement and connectivity. The program method allows one to build large networks (many hundreds of nodes) without having to draw each component.

### **3.1.4 Simulation Development and Execution**

The user can run event-driven simulations on networks, during which nodes and jammers can move, radio characteristics can change, traffic patterns can change, and so on. During simulation, OPNET calculates connectivity characteristics between nodes based on position, transmitter power, frequency, jammers, and so on. OPNET generates traffic based on one of several probability distribution functions, or the user can design his own traffic generation module. OPNET also generates output data both from its own modules (for instance, queue length for queues, packets sent and received, and number of errors for transmitters and receivers), and from user-defined protocols.

### **3.1.5 Simulation Analysis**

After a simulation is executed, simulation outputs can be analyzed by plotting the outputs on graphs, and by running the output data through OPNET-defined and user-defined filters, such as averagers, differentiators, summers, and so on. Raw output can also be made available to user programs that are run from UNIX.

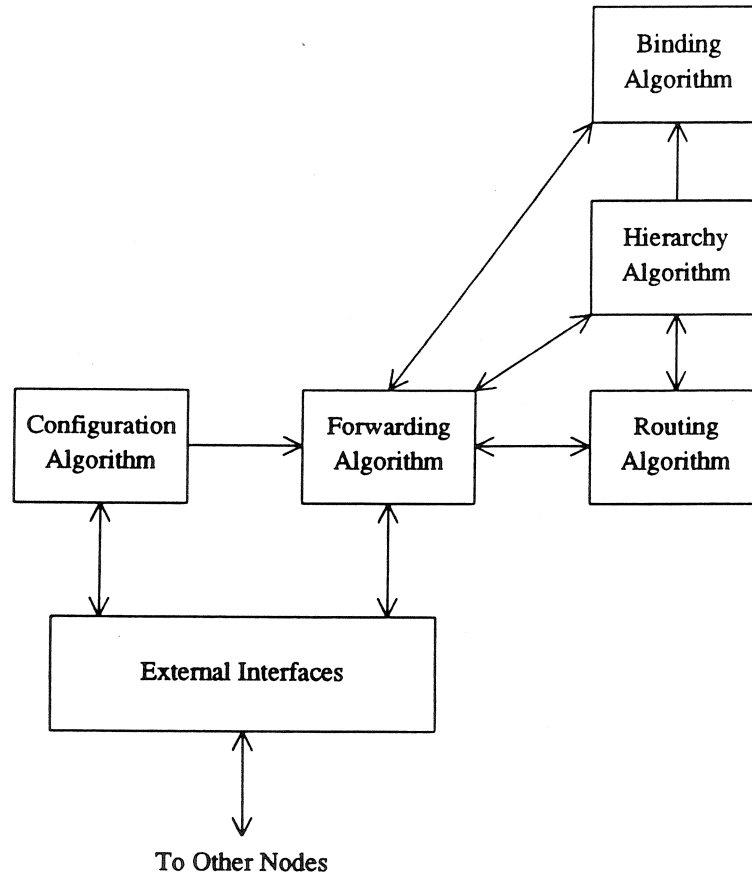
## **3.2 Landmark Routing Node Design**

Figure 4 shows a functional diagram of a Landmark Routing node. It shows the three major algorithms, the Routing Algorithm, the Hierarchy Algorithm, and the Binding Algorithm, all briefly discussed in Section 2. It also shows the Forwarding Algorithm, Configuration Algorithm, and interfaces to other nodes.

The main task of the Forwarding Algorithm is to receive packets from its neighbors (directly connected nodes), and either deliver them to the higher layers if it's node is the destination, or forward them on to other neighbors if it is not. To do this, the Forwarding Algorithm requires two inputs, one from the Configuration Algorithm and one from the Routing Algorithm. The Configuration Algorithm simply tells the Forwarding Algorithm who its neighbors are. The

Figure 4  
Landmark Routing Node: Functional Diagram

---



---

Routing Algorithm tells the Forwarding Algorithm which neighbors are the next hop to various destinations. The Forwarding Algorithm builds a routing table with this information.

The Configuration Algorithm is simple. It sends messages over all of the external interfaces, and receives such messages sent by its neighbors (understand that some of the external interfaces may not have attached neighbors). The Configuration Messages 1) inform neighbors of each other's existence, and 2) give the Node ID of the neighbor. When the Configuration Algorithm learns of a neighbor coming up or going down, it tells the Forwarding Algorithm (and the Routing and Hierarchy Algorithms, although for simplicity we don't show that in Figure 4).

Finally, the External Interfaces provide the physical connectivity to other nodes. In all of the simulations presented in this report, the connections between nodes are single full-duplex point-to-point links (as opposed to a broadcast medium, for instance).

### 3.2.1 OPNET Node Design

Figure 5 shows the node as it was used in the OPNET simulations. Figure 5 shows the same architecture as Figure 4, but with more detail.

Figure 5 shows five kinds of OPNET modules; 1) software processes, 2) a traffic generator, 3) transmitters, 4) receivers, and 5) FIFO (First In First Out) queues. These modules are connected by streams (lines with arrows) indicating where messages may flow.

Each software process shown operates as a software process in the traditional operating sense—it is scheduled and interrupted separately from the other software processes. Each software process represents a chunk of C code that executes the algorithm.

Notice that both the Hierarchy Algorithm and the Binding Algorithm are modeled as two software processes, one perfect and one real. This is so that we have control experiments to compare our results against. The Perfect Hierarchy and Perfect Binder perform perfectly. The Real Hierarchy and Real Binder are the algorithms we developed. In any given simulation, either the perfect or the real algorithm is chosen, not both. A global configuration parameter, set at run time, determines which is chosen. We put both perfect and real algorithms in the same node to avoid having four different types of nodes, which in the end would have resulted in 16 simulation executables (all combinations of 4 networks and 4 nodes) and the extra compiling and memory required.

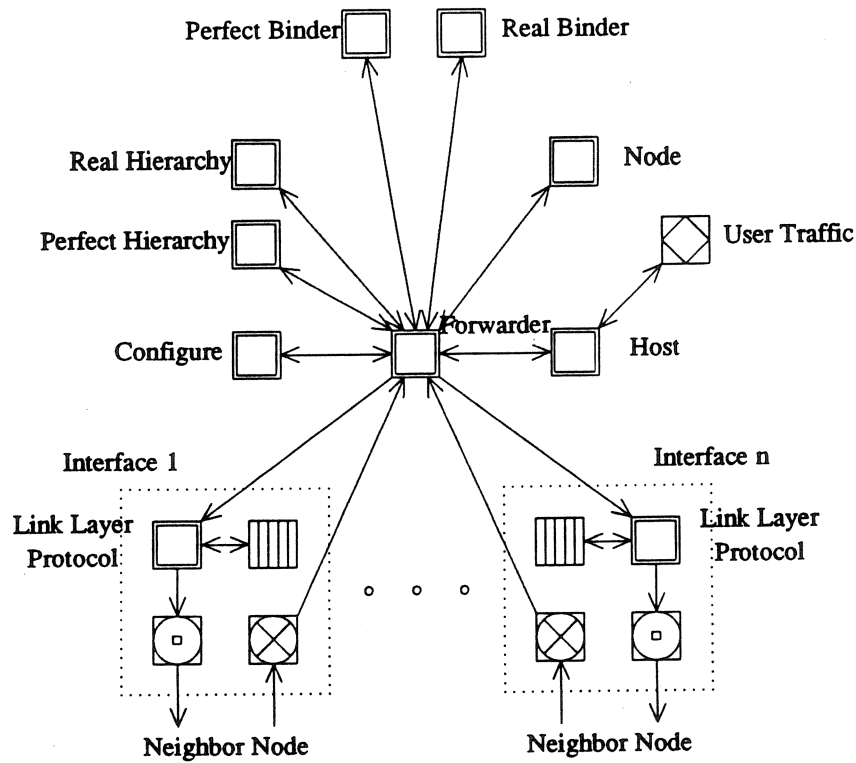
Figure 5 shows a Host module that is missing in Figure 4. This module emulates host traffic that is sent to the node for delivery to other hosts via other nodes. The hosts could have been modeled as separate nodes connected to the nodes via external links. We chose to incorporate the host into our node for simplicity.

The Host software emulates multiple hosts, and generates packets to send to other hosts. The purpose of this traffic is not to load the network. Rather, it is to test the correctness of the algorithms. The packets are generated using an OPNET traffic generator. When a packet is generated, it is sent to the Host module, which determines the destination host to which to address the packet before delivering the packet to the Forwarder Algorithm.






Figure 5 shows that all streams go through the Forwarder module. We did this because initially this made certain statistics gathering easier. Later versions of OPNET made this unnecessary, but we never changed our node model.

Finally, each external interface consists of a Link Layer software module, a FIFO queue, a transmitter, and a receiver. The Link Layer software does not actually run a link layer protocol. It simply emulates the effect of a connection-oriented link protocol in that it always delivers packets reliably and in order, but can introduce delay to packets to emulate other packets on the queue, or link level retransmissions. By not actually running a link protocol, and by artificially introducing delay over the link rather than generate loading traffic, we get the appropriate link characteristics much more efficiently. In real systems, protocols like LAP-B, X.25, TCP, or ISO TP4 can be used to provide reliable, connection-oriented communications.

Figure 5  
OPNET Node Design



LEGEND:

-  Software Process
-  Traffic Generator
-  Transmitter
-  Receiver
-  FIFO Queue

The Link Layer software process uses the attached FIFO queue to hold packets that have been slated for delay. The transmitter module takes into account the delay over the link due to the packet length, as well as queues up packets internally for one-at-a-time delivery over the link. The receiver module interrupts the Forwarder module upon receipt of the end of the packet.

### 3.2.2 Software Modularity

All of the software modules interface via a message passing interface. No memory is shared between modules. This results in less efficient simulations, because 1) sometimes two modules will hold the same information, and 2) because sending a message between modules is less efficient than simply going to shared memory.

The advantage, however, is modularity. With a message interface, it is easier to change one algorithm without affecting the others. The fact that we have multiple algorithms of the same type (real and perfect) is testimony to this. The modularity also 1) makes protocol specification easier, because the inputs and outputs to the algorithm are clearly delineated, and 2) will make porting to a real system easier, because each module is self contained.

### 3.2.3 Other Simulation Aspects

In addition to the above simulation aspects, we provided skewed node clocks, and node crashes.

**3.2.3.1 Simulating Clocks.** One important characteristic of the distributed environment is that nodes' clocks are not synchronized, and that they drift away from each other. OPNET has two interrupt paradigms, continuous periodic interrupt and single interrupt. When a module is defined (initially placed with a mouse), its periodic interrupt interval (or lack thereof) is defined. Setting a periodic interrupt with period  $T$  will cause the module to be interrupted during simulation at time  $0, T, 2T, \dots$ , seconds.<sup>7</sup> If all nodes' clocks were based on an OPNET periodic interrupt, then the nodes would all be synchronized.

To provide skewed clocks, we initially set a periodic interrupt to bring our nodes up at time  $0$ . Once a node comes up, it cancels the periodic interrupt, and sets a single interrupt `BOOT_TIMER` with a random time. The purpose of this is to simulate nodes booting at different times. When `BOOT_TIMER` interrupts the node, it then sets a one second periodic timer, but places a small random positive or negative skew on the one second timer. All protocol timer functions are based on this skewed one second timer. As a result, all nodes, and therefore all protocol events, are asynchronous.

**3.2.3.2 Simulating Crashes.** Another characteristic of our environment is that nodes and links can crash and come back up. As far as our distributed algorithm is concerned, there is no difference between a link crashing because the link itself broke, or because the lower-layer connection was dropped. When a node crashes, the algorithm loses all of its state, and reinitializes

---

<sup>7</sup>Time here refers to simulated time.

when the node comes back up. For realism, nodes and links must crash and recover in a non-deterministic fashion. For debugging and for generating worst case scenarios, however, we must also be able to deterministically crash nodes and links. These features were easy to provide in the OPNET/UNIX programming environment.

### 3.3 Landmark Routing Network Design

In OPNET, after a node is designed (and compiled), it can be used to build networks. As previously mentioned, the networks we used for testing are based entirely on directly connected, point-to-point (non-broadcast) links. We used four networks for the data we present in this paper. (We used other, smaller networks for debugging purposes.) The four networks are designed to isolate the impact of the number of nodes, the node degree (the number of neighbors each node has), and the diameter of the network. One of our networks had 50 nodes, an average node degree of 3, and a diameter of 7 (n50d7e3). To see the impact of the number of nodes, another network had 25 nodes, but the same node degree and diameter (n25d7e3). To isolate node degree, a third network had 50 nodes and a diameter of 7, but an average node degree of 6 (n50d7e6). Finally, our fourth network had a diameter of 14, 50 nodes, and an average node degree of 3 (n50d14e3).

These networks are quasi-random in that, within the parameters stated, the links are chosen quasi-randomly. The technique for generating such networks is described in the first report of this series (Tsuchiya, 1987a). Basically, it involves creating a loop with  $N$  nodes and  $N+1$  links, and then adding more links quasi-randomly until the desired node degree is achieved. If a smaller diameter is desired, the added links span more of the loop. If a larger diameter is desired, the added links span less of the loop. In our networks, no node had more than eight neighbors.

The choice of number of nodes represents a compromise between having enough nodes to build a meaningful hierarchy (several levels), and having efficient simulations. The choice of average node degree is based on existing point-to-point networks such as the ARPANET. The choice of diameter is somewhat dependent on the number of nodes and average node degree, and ours are chosen within the feasible range given the other parameters.

All of our links ran at 1.54Mbps (T1 speed). As such, we did not stress the algorithms by constraining link bandwidth.

### 3.4 Running Simulations

Within the framework of the networks chosen, there are numerous algorithm parameters that can be changed. These parameters are described in the sections discussing the algorithms themselves, and we won't go into them here. We designed our code so that all of these parameters could be set at run time, based on the contents of text files. Our experiment philosophy was to pick a baseline set of parameters, and then modify each one to see the impact of that parameter.

Some of the run-time text configuration files contained failure scenarios. Therefore, for any given set of parameter values, we could run several different failures.

During our simulations, we gathered many statistics, in both OPNET analyzable (raw data files) and text forms. The text forms generally gave us summary data that we could peruse quickly. The OPNET analyzable form gave us graphical displays of parameters during the whole course of the simulation. Thus, we could see exactly what happened before and after a particular failure.

## 4 CONFIGURATION, FORWARDING, ROUTING, and HOST ALGORITHMS

In this section, we discuss the details of the Configuration, Forwarding, Routing, and Host Algorithms. Since the operation of these algorithms impacts the operation of the Hierarchy and Binding Algorithms, it is important to understand them.

### 4.1 Configuration Algorithm

The Configuration Algorithm 1) determines the existence and ID of neighbor nodes (those nodes directly connected via a link), and 2) determines the existence and ID of connected hosts.

For finding neighbors and hosts, the Configuration Algorithm mimics an up/down protocol that pings neighbors and hosts periodically to indicate their presence. When a neighbor event occurs, either a link going up or down, or a neighbor node going up or down, the Configuration Algorithm discovers this event within  $T_{cn}$  seconds. If a link goes up or down, the two neighbors on either end of the link will not discover this at the same time. For all of the experiments described in this paper, we set  $T_{cn} = 2$  seconds.

Upon a neighbor event, the Configuration Algorithm notifies the Forwarding and Routing Algorithms. The notification states whether the neighbor is up or down, gives the ID of the neighbor, and the link over which the neighbor can be reached.

In our experiments, we did not crash hosts. We did, however, emulate mobile hosts. When a host moves, the disconnection is discovered in an average of  $T_{ch}$  seconds, and the reconnection is discovered in an average of  $T_{ch}$  seconds. It is possible for the reconnection to be discovered before the disconnection, and vice versa. We set  $T_{ch}$  to 5 seconds.

Upon a host event, the Configuration Algorithm notifies the Binding and Forwarding Algorithms. The notification contains whether the host is being added or removed, and the host ID.

### 4.2 Forwarding Algorithm

The Forwarding Algorithm is the “switch” part of the node. It receives packets from other nodes and from higher-level algorithms, and delivers them to the appropriate destination. The forwarding may require the use of what is traditionally called a routing table (a list of destinations, masks, and appropriate next hop), but what we will call a forwarding table. The Forwarding Algorithm is connectionless in that it does not reserve memory resources for packet flows, and does not exert flow control on its neighbor nodes or on hosts.

The Forwarding Algorithm receives control input (as distinguished from data packets) from 1) the Configuration Algorithm, 2) the Routing Algorithm, 3) the Hierarchy Algorithm, and 4) the Binding Algorithm. From the Configuration Algorithm, the Forwarding Algorithm learns 1) the IDs of its neighbors, and which links they are reachable over, and 2) the IDs of the hosts associated with it. From the Routing Algorithm, the Forwarding Algorithm gets the Forwarding Table. From the Hierarchy Algorithm, the Forwarding Algorithm learns the node’s (its own) Landmark Address.



From the Binding Algorithm, the Forwarding Algorithm learns the Landmark Addresses of destination Hosts to which it is forwarding packets.

#### 4.2.1 Forwarding Data Packets

Data packets (packets sent between the various algorithms) can be divided into two parts, the Forwarder Header and the Data part. The Forwarder Header here is analogous to the header of a network layer protocol like IP. The information in the Forwarder Header is shown in Table 2. The Forwarder Header has source and destination IDs and source and destination Landmark Addresses. This is indicative of a departure from existing data protocols where the “address” both uniquely identifies and locates the source or destination. In Landmark Routing, the Landmark Address changes while the ID does not. Therefore, we explicitly separate the two functions in the header. Notice that Hosts take on the Landmark Address of their Node, and therefore multiple Hosts may have the same Landmark Address.

*Table 2*  
**Data Packet Information**

---

Field	Description
source_id	ID of packet source (Host or Node)
dest_id	ID of packet destination (Host or Node)
source_addr	Landmark Address of packet source (Node only)
dest_addr	Landmark Address of packet destination (Node only)
proto_type	Protocol type (Binder, Hierarchy, etc.)
last_hop	ID of last forwarder to handle packet
num_hops_traveled	Number of hops already traversed
data	Rest of packet (protocol specific information)

---

The `proto_type` determines which protocol within a node the packet is destined. The protocol specific information is in the data part of the packet.

The `last_hop` field is used to discover and quench ping-pong loops, where two Forwarders are passing a packet back and forth to each other. When a Forwarder forwards a packet, it puts its ID in the `last_hop` field. Then the next forwarder receives the packet, it checks to see if it is going to forward the packet to the forwarder identified in the `last_hop` field. If it is, an error is logged, and the packet is thrown away.

The `num_hops_traveled` is used to discover longer loops. Every time a Forwarder forwards a packet, it increments the `num_hops_traveled` field. If the `num_hops_traveled` field is incremented beyond the maximum allowable number of hops (some number greater than the largest expected diameter of the network), then the packet is assumed to be looping, an error is logged, and the

packet is thrown away. These loop checks are necessary because the Routing Algorithm (Section 4.3) often has loops.

**4.2.1.1 Filling in the Destination Landmark Address.** When the Forwarder receives a packet from a Host, it must fill in the Landmark Address part of the header based on the destination Host ID. Packets received from any other protocol already have the necessary header information filled in. To do this, the Forwarder first looks in its local cache to see if it already has the Landmark Address for the destination Host. If it does not, it sends a query to the Binding Algorithm requesting the Landmark Address, internally queues the packet, and processes other packets. After searching for the requested information, the Binding Algorithm eventually returns 1) a positive reply containing the Landmark Address, 2) a negative reply indicating that it could not find the requested address, or 3) nothing. If the Forwarding Algorithm receives the negative reply from the Binding Algorithm, it pulls the data packet from the queue and throws it away. Since the Forwarding Algorithm ages all data packets in its queue, the data packet will eventually time out and be thrown away if a reply is never received.

If a positive reply is received, the Forwarding Algorithm 1) caches the binding information in the reply for subsequent data packets, 2) pulls the appropriate data packets from the queue and fills in their `dest_addr` fields with the Landmark Address, and 3) routes the packet. The cached binding information is aged. This caching didn't affect our simulations, since in our simulations hosts rarely send packets to the same destination twice. The third step (routing the packet) is the same thing a Forwarder does when it receives a data packet from a neighbor Forwarder.

**4.2.1.2 Routing a Data Packet.** When routing a data packet, the Forwarder first checks the packet's `dest_id` to see if it matches its own or one of its Hosts. If it matches one of its Hosts, the packet is forwarded to the Host. If it matches its own, then the `proto_type` is used to determine which protocol should receive the packet (Configure, Forward, Route, Hierarchy, or Bind), and the packet is sent.

If 1) the `dest_id` does not match, 2) the `dest_addr` does match the Forwarder's Landmark Address, and 3) the `proto_type` is for a Host, then the Forwarder throws away the packet and sends a `HOST_BAD_ADDR` message to the Forwarder that originated the packet to let the Forwarder know that its cached entry is wrong. Upon receipt of a `HOST_BAD_ADDR` message, the cached entry referred to by the Host ID in the message is deleted. This will cause the Forwarder to send another query to the Binding Algorithm when another data packet for that destination is received from a Host.

If the above conditions hold except that the `proto_type` is not for a Host, then the packet is thrown away, because the various algorithms will work out the problem.

If neither the `dest_id` nor `dest_addr` matches, then the packet must be forwarded to another node. The Forwarder scans the forwarding table, which contains tuples of the form `<addr,mask,next_hop>`. These are arranged in order of largest mask first, because the larger the mask, the more detailed the routing information in the entry. The Forwarder applies the mask to the `dest_addr` and compares it with `addr`. If they match, then `next_hop` is returned. If not, the next entry is tried. If no match is found, the packet is thrown away, and a `HOST_BAD_ADDR` message

is sent to the originating node so it can flush the bad entry from its binding cache. The packet is sent over the link leading to the node identified by next\_hop.

**4.2.1.3 Establishing the Forwarding Table.** The Routing Algorithm sends the Forwarding Algorithm messages with entries for its forwarding table. The messages are of type add, change, and remove. They contain the three items in the forwarding table tuple: addr, mask, and next\_hop. The Forwarding Algorithm does not age forwarding table entries. They are removed only by explicit command from the Routing Algorithm.

### 4.3 Routing Algorithm

Although we designed a high-performance distance-vector routing algorithm for this project (Alternate-Path Distance-Vector Routing) (Tsuchiya, 1987b), we did not have time to implement it for this paper. However, any routing algorithm of the distance-vector variety can be used in Landmark Routing (with minor modifications to account for radii). For reasons not given here, we chose to use a distance-vector routing algorithm called Tier Routing, developed for the SURAN Packet Radio Project (Westcott and Jubin, 1982; Westcott, 1982). The base code we used for this was written by BBN. We modified it to take into account Landmark Hierarchy constraints, and to run on OPNET. Our version of Tier Routing did not have the code for directional flooding, nor did it have the code that updates the routing table based on information gleaned from data packets, and therefore performed worse than would a full version of Tier Routing.

Tier Routing is not what we would call a high-performance distance-vector routing algorithm. It uses timer-based rather than event driven updates, and therefore propagates routing information more slowly than it otherwise might. It also suffers from the count-to-infinity problem. Both of these conditions result in slow convergence (Tsuchiya, 1987b). The count-to-infinity problem, however, also causing a thrashing of routing table entries where a routing table entry will exist for a particular destination, then it will be purged, then return again, only with a longer distance. We mention this because the thrashing has an impact on the operation of the Binding Algorithm.

The performance of the Routing Algorithm directly affects the performance of the Hierarchy Algorithm, and to a lesser extent, the Binding Algorithm. However, using a low-performance Routing Algorithm for this research is a blessing in disguise. As long as we understand what impact the Routing Algorithm performance has on the Hierarchy and Binding Algorithm performance, we can predict the performance of the Hierarchy and Binding Algorithms given a better Routing Algorithm. However, using a poor-performance Routing Algorithm allowed us to discover holes in the Hierarchy and Binding Algorithms that we may not otherwise have discovered.

Normally, one would take into consideration link usage, CPU usage, memory usage, and convergence time when characterizing a routing algorithm. With respect to the impact of the Routing Algorithm on the Hierarchy and Binding Algorithms, we are only concerned with convergence time. The Hierarchy and Binding Algorithms can act only after information is received from the Routing Algorithm. This causes the Hierarchy Algorithm to configure slower, and the Binding Algorithm data-bases to be in inconsistent states for longer periods of time.

The nature of distance-vector routing algorithms is that information about a network change propagates outward from the locality of that change. If a link comes up, the nodes on the ends of the link will learn it first, followed by their neighbors, followed by their neighbor's neighbors, and so on. Define the average time it takes for routing information to travel one hop to be  $T_{update}$ . Good news (a node or link coming up, resulting in a decreased distance to destinations) travels at the rate of  $T_x$  seconds/hop. Therefore, if the diameter of a network is  $D$  hops, all nodes will learn of a new node within  $D T_{update}$  seconds.

For event-driven distance-vector routing schemes, nodes asynchronously trade routing information once every  $T_r$  seconds. On hearing of new information from a neighbor, a node will, on the average, pass it on to another neighbor in  $\frac{T_r}{2}$  seconds. Therefore,  $T_{update} = \frac{T_r}{2}$ . In simulations of Tier Routing, we found this estimate to be accurate.

It is harder to characterize the spread of bad news (a link or node going down, resulting in an increased distance to destinations). While it remains true that information will, on the average, take  $T_{update}$  seconds to travel one hop, the nature of the count-to-infinity problem means that the source of bogus news about a destination can come from any node, not just from the locality of the network change. Further, as the count-to-infinity goes on, bogus news will emerge from different places.

In our simulations of the count-to-infinity problem, we found that the network often exhibited oscillatory behavior towards the down destination. Bad news about the down destination spreads outwards from the destination until most nodes purge the entry. Then a routing update loop occurs, causing bogus now-good news to re-enter most nodes. Then a wave of purges spreads from yet another location, followed by another wave of bogus news. All the while the distance to the destination by the nodes that manage to have an entry for the destination is increasing. Eventually, either the distance limit is reached (whatever infinity is defined as) thus causing a full purge of the down node from the routing tables; or all nodes manage to purge the down destination simultaneously (no bogus news starts up). Normally, the latter will eventually occur, although there is no way to predict when.

In our simulations of count-to-infinity, we tested three networks. Each had 50 nodes. One had diameter  $D = 7$  and node degree  $E = 3$ , one had diameter  $D = 14$  and node degree  $E = 3$ , and the third had diameter  $D = 7$  and node degree  $E = 6$ . We ran each network twice, once with  $T_r = 2$  seconds, and once with  $T_r = 8$ . We measured the rate at which the distance seen to a crashed destination increased. For  $T_r = 8$ , the distance increased at 0.64 hops/second. For  $T_r = 2$ , the distance increased at 0.19 hops/second. Further, the rate of increase was constant during the count-to-infinity—it did not increase and decrease. The three networks behaved almost identically.

The important things about the propagation of bad news with respect to impact on the Hierarchy and Binding algorithms is 1) that it can take a long time to completely purge knowledge of a down node, 2) that the distance seen to that node increases until the node is purged, and 3) during this time, some nodes will have entries for the down node, while others will not.

#### **4.4 Host Algorithm**

The purpose of Hosts is to generate traffic to test the correctness and performance of the algorithms, not to load the network with traffic. Hosts are fed by an OPNET traffic generator with a uniform probability distribution between 0 and 10 seconds, therefore generating packets at an average rate of one per five seconds. When a Host receives a packet from the packet generator, it will randomly pick a destination Host to send the packet to, put the ID of the destination Host in the `dest_id` field of the packet header, and send the packet to the Forwarder. If the packet is correctly received at the destination Host, some statistics are logged, such as packet delay and path length, and the packet is returned to the system. Comparison of the number of packets sent and the number received gives one indication of performance.

If a Popular Host is configured, then all other Hosts will always send their packets to the Host designated as the Popular Host. The Popular Host tests the adjacencies feature of the Binding Algorithm. This feature automatically spreads bindings over many nodes if there are many queries for that binding.

## 5 HIERARCHY ALGORITHM DESCRIPTION

The purpose of the Hierarchy Algorithm is to create the Landmark Hierarchy. For this, each node must determine its hierarchy level, its radius, and its Landmark Address.

### 5.1 Landmark Hierarchy Structure

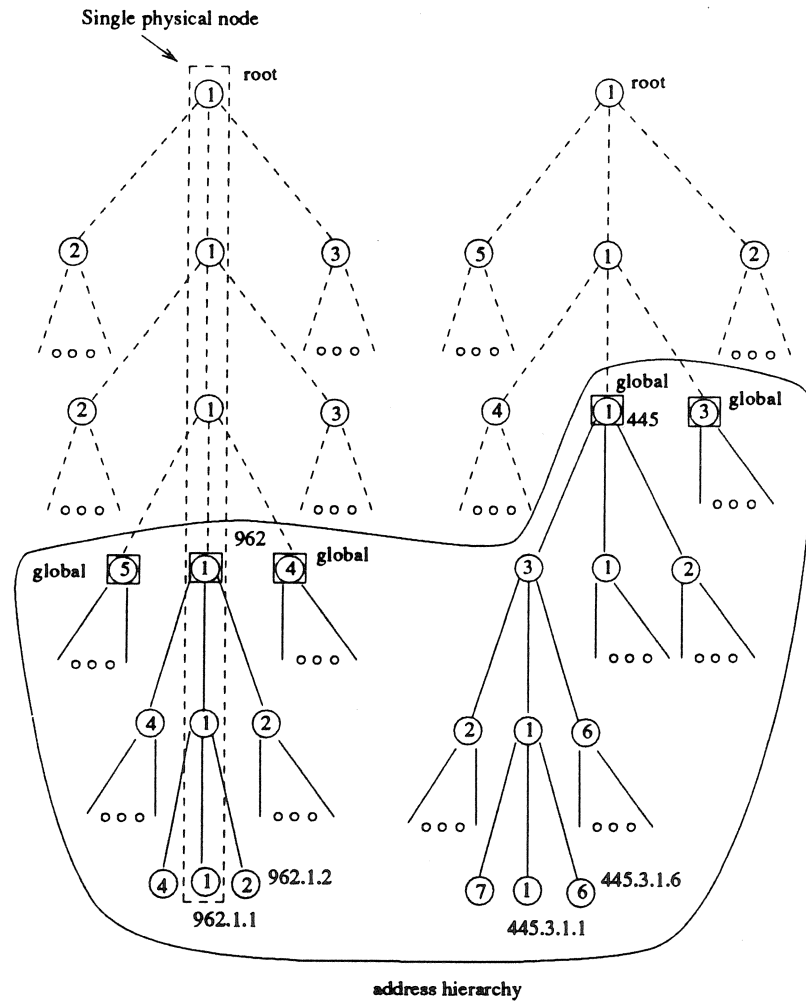
In Section 2, we give a brief description of Landmark Routing: the Landmark Hierarchy, the Landmark Addresses, how large radii must be, the structure of the routing tables, and how paths are chosen. Figure 6 gives more detail about the tree structure of the Landmark Hierarchy. It shows a six-level Landmark Hierarchy. Every Landmark is either a root of the Landmark Hierarchy, or has one and only one parent. (As discussed in Section 2, the radius of a Landmark must cover its parent. Radii are not shown Figure 6.) In Figure 6, there are two roots. In general, there are as many roots as there are highest level Landmarks.

Each Landmark has 1) an ID, 2) a Hierarchy Address (or `hier_addr`), and 3) a Landmark Address (or `lm_addr`). The ID uniquely identifies the Landmark among all other Landmarks, and never changes. The `hier_addr` tells where the Landmark is in the Landmark Hierarchy with respect to its root Landmark. The `hier_addr` is not necessarily unique among all Landmarks, because roots cannot be distinguished by looking at the `hier_addr`. The `hier_addr` is used to describe the parent-child relationships all the way up to a root Landmark, and is therefore useful in reconfiguring the Landmark Hierarchy when necessary. The `lm_addr` is the address used in the actual forwarding of data packets (as distinguished from control packets such as routing updates). It, and not the `hier_addr`, is what appears in the data packet header. The `lm_addr` is unique among all Landmarks. (Much of the justification for this structure is sprinkled throughout the rest of this section. Please bear with us.)

Both the `hier_addr` and the `lm_addr` consist of a series of labels. The value of each label distinguishes the children of a given Landmark. While the `hier_addr` describes the parent-child relationships of the entire Landmark Hierarchy, the `lm_addr` only describes the Landmark Hierarchy up to and including a Landmark partway up the Landmark Hierarchy—the global Landmark. This is highlighted by the balloon in Figure 6. Instead of a label distinguishing it from its siblings, the global Landmark has a larger label distinguishing it from all other global Landmarks. It is because of this that `lm_addr`'s are unique.

The purpose of global Landmarks, and the resulting dual addresses, is to minimize the number of nodes dependent on any other single node for their addresses. The reason is to minimize the number of addresses that can change at any one time, and therefore minimize the chaos and overhead brought about by any address change. We particularly want to minimize the traffic caused by binding updates, since a surge in binding traffic can congest the net, and can therefore take a long time to converge. For example, assume there are no globals, and that the left root in Figure 6 crashes. In this case, all of the nodes under it (half the nodes in the network) would get new addresses, and the resulting binding effort would be enormous. With globals, however, only the global labeled 962 and its (non-global) offspring would get new addresses. If there are  $\sqrt{N}$  global Landmarks (where  $N$  is the total number of nodes), then a single crash will affect only  $\sqrt{N}$  other nodes (100 nodes in a 10000 node network).

Figure 6  
Landmark Hierarchy Tree Structure



Some addressing conventions: In our simulations, each Landmark has a maximum of 7 children. Therefore, only three bits (1 through 7) are needed to label a child. Networks with node degrees of 5 or 6 or more may use 4 bits (or 15 children children per Landmark). In our simulations, Landmarks always assign the label 1 to themselves (a level  $i$  Landmark is its own child at levels below  $i$ ), and labels 2-7 to other Landmarks. By doing this, one can by inspecting an `lm_addr` determine if the Landmark is global, and if not what level Landmark it is. One can always determine the level of a Landmark by inspecting its `hier_addr`. A Landmark is global if all of its labels (after the global label) are 1's. The level of a Landmark is the position, counting from the left, of its rightmost non-1 label. For instance, 962.1.1 is a global Landmark, 445.3.1.1 and 962.1.2 are not. 445.3.1.1 is a level 2 Landmark, 962.1.2 is a level 0 Landmark, and one cannot tell what

level 962.1.1 is without looking at its hier\_addr. Its hier\_addr, however, is 1.1.1.1.1—we know that it is a level 5 Landmark, and also that it is a root.

The whole point behind this clever use of 1 and non-1 labels is to avoid explicitly sending around global/not-global and level information in Hierarchy Update packets. We have since decided that this is a stupid idea, as it requires constantly having to parse the address to gather this information, and unduly constrains the address structure.

As we will see, maintaining global Landmarks contributes a large portion of the complexity of the Hierarchy Algorithm. However, for the above-mentioned robustness reasons, we don't see that it can be avoided. The Hierarchy Algorithm, again, is the process of choosing a Hierarchy level, choosing a parent, choosing a radius, and choosing an address, in part by deciding whether or not to become global.

## 5.2 High-level Hierarchy Algorithm Description

Figure 7 shows the Hierarchy Algorithm (for a single Landmark), the other algorithms it communicates with, and the data communicated. Within the same Landmark, the Hierarchy Algorithm communicates with the Routing Algorithm and the Binding Algorithm. The Hierarchy Algorithm also communicates with Hierarchy Algorithms in other Landmarks.

Figure 7  
Relationship of Hierarchy Algorithm to Other Algorithms

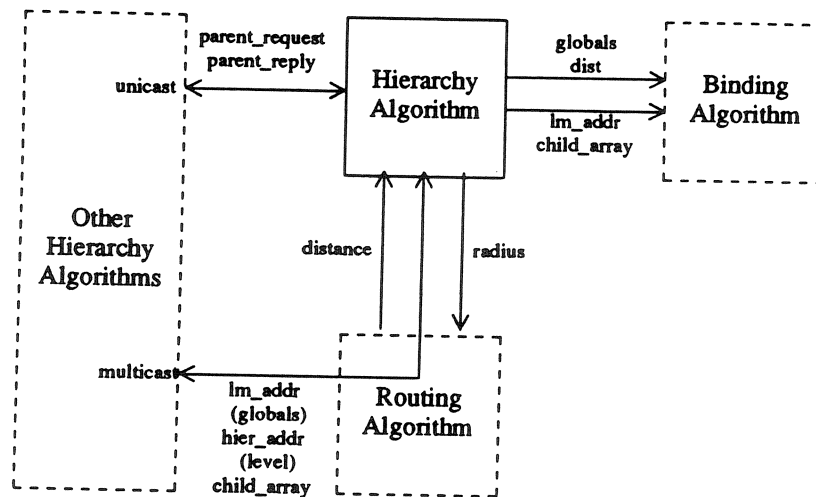
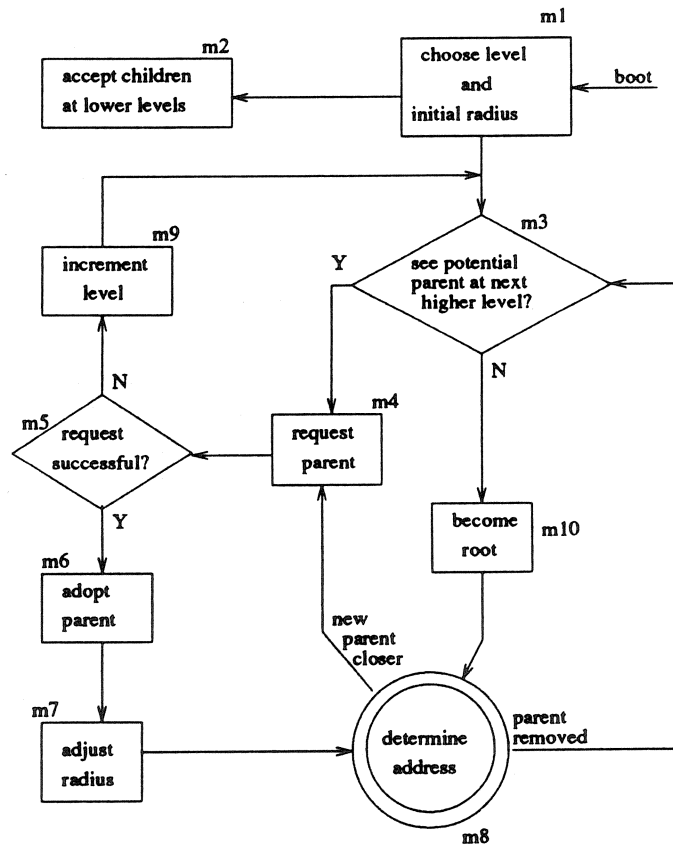


Figure 8 is a high-level flow diagram of the Hierarchy Algorithm. Upon power-up, each Landmark randomly chooses a hierarchy level and corresponding radius, large enough to cover a potential parent with high probability (box m1 in Figure 8). The Landmark then both accepts children at lower levels (box m2), and waits to hear of a potential parent. If it hears a potential



parent, it requests permission to be adopted by that parent (m4). If the answer is yes, the Landmark reduces its radius appropriately (to just cover the parent), and picks either a global address, or an address that is under its parent's address tree (m8). If the answer is no, the Landmark increments its level by one, increases its radius correspondingly, and tries to find a new potential parent. If either at initial boot time, or after incrementing its level, the Landmark sees no potential parent, it assumes that it is a root of the Landmark Hierarchy, and assigns itself a global address.

*Figure 8*  
**High-Level Landmark Algorithm Flow Diagram**



Once a Landmark has a parent (or is a root) and an address, it is in its steady state. It can choose a new parent if its old parent disappears, or if a potential new parent is much closer than the old parent. If the Landmark was not global, this will result in a new address. Even without getting a new parent, it can get a new address by becoming or ceasing to be global, or by its parent getting a new address.

The main features of the Hierarchy Algorithm are:

1. Random selection of hierarchy level. This is the key feature. By randomly selecting the hierarchy level (with decreasing probability at higher levels), we avoid elections, and the accompanying complexity and delay.
2. Modification of radius based on distance from parent. It is this feature that allows us the freedom to randomly select the hierarchy level. If the parent is far away, simply make the radius large.
3. Ability to choose a new parent, by incrementing the hierarchy level if necessary.
4. Ability to become or cease being global.

### 5.3 Detailed Hierarchy Algorithm Description

#### 5.3.1 Information Needed by Hierarchy Algorithm

Table 3 shows the state information kept by the Hierarchy Algorithm. Listed first is the `hier_addr`. It is an array of  $H_{hier} - 1$  labels,  $H_{hier}$  being the maximum number of hierarchy levels. In our implementation,  $H_{hier} = 11$ . The reason we only need  $H_{hier} - 1$  labels for  $H_{hier}$  levels is because the root level is implicit, and doesn't require a label. Each label has  $\lceil \log_2 C_{max} \rceil$  bits, where  $C_{max}$  is the maximum number of children per parent. In our implementation,  $C_{max} = 7$ , and therefore each label has 3 bits.

The variable `level` is the hierarchy level. This is the level randomly chosen at boot time, and possibly incremented later on. As already discussed, it can be derived from the `hier_addr`, and is therefore not explicitly carried. Also as discussed, we think `level` should be explicitly carried.

The next variable is `lm_addr`, also discussed previously. It has  $H_{lm}$  labels. We used  $H_{lm} = 7$ . The lower 6 labels are also 3 bits (this must correspond to the lower 6 `hier_addr` labels). The highest (global) label is 14 bits. Therefore, the `lm_addr` is 32 bits long—long enough to fill the DoD IP Address space.

The variable `global_status` indicates whether or not the Landmark is global. It can be derived from the `lm_addr`.

The next variable is `radius`, which is described in Section 2 and later in this section.

Next is the `parent_id`. This variable identifies the Landmark's parent. Other information about the parent is in the array `lm_entries[ ]`, which is indexed by `id`.

The next variable is `parent_state`. It has three possible values: `VACANT`, `PENDING_VERIFICATION`, and `OCCUPIED`. `VACANT` means that a potential parent has not even been identified. `parent_state` is `VACANT` if the Landmark is a root. `parent_state` is `PENDING_VERIFICATION` when a potential parent has been identified, but has not yet been

Table 3  
Hierarchy Algorithm State Variables

Hierarchy Algorithm Data	
Object	Description
Derived Object	
hier_addr	array of $H_{hier}$ labels each label has $\lceil \log_2 C_{max} \rceil$ bits
level	hierarchy level $i_{hier}$
lm_addr	array of $H_{lm}$ labels labels $(H_{lm} - 2)$ to 0 have $\lceil \log_2 C_{max} \rceil$ bits label $(H_{lm} - 1)$ has $\lceil \log_2 G_{max} \rceil$ bits
global_status	binary, global or not global
radius	integer, $r \geq 1$
parent_id	use to find parent information in <code>lm_entries[ ]</code>
parent_state	VACANT, PENDING_VERIFICATION, OCCUPIED
lm_entries[ ]	list of known Landmarks index by Landmark ID $LM[id]$
hier_addr	same as above; if parent, derive own hier_addr from this
lm_addr	same as above; if parent, and I am not_global, derive own lm_addr from this
distance	distance to Landmark
child_array[ ] [ ]	bit array, index by level and label binary value, child exists or not (derive num_children)
parent_status	VALID_PARENT, NOT_VALID_PARENT
child_entries[ ] [ ]	[level] [label]
child_id	use to find child information in <code>lm_entries[ ]</code>
child_state	VACANT, PENDING_VERIFICATION, OCCUPIED

secured. After a potential parent answers positive to a `parent_request_msg`, then the `parent_state` becomes OCCUPIED. This is the normal steady state.

The above variables describe oneself. Strictly speaking, a Landmark also knows its own id. However, since this id applies to other algorithms, and is not modified by the Hierarchy Algorithm, we do not list it. Information about other Landmarks are kept in an array called `lm_entries`. The entries in `lm_entries` are distinguished by the id of the Landmark. The first two variables in an `lm_entry[ ]` are `hier_addr` and `lm_addr`.

Next is the variable `distance`. This is the distance, in hops, to the Landmark. Notice we do not care what the radius of other Landmarks are—only whether or not they can be seen.

The next `lm_entry[ ]` variable is `child_array[ ][ ]`. `child_array[ ][ ]` gives the labels of all the children at each level. It does this through a bit map: one byte for each level, one bit, positioned by label value, for each child. The `child_array[ ][ ]` of all entries in `lm_entries[ ]` is used as part of the determination of whether or not to become global. This is explained in Section 5.3.3.1.

By counting the ones in a `child_array[ ][ ]` entry, the `num_children` for a Landmark at each level can be calculated. The `num_children` is used to determine if a Landmark would reject a parent request (because it had the maximum number of children). It is not a very important piece of information, because a parent reject is rare. However, since `num_children` is derived from `child_array[ ][ ]`, no extra bandwidth is required to convey `num_children`. Still, it adds some extra complexity, and on retrospect we don't think it should be used.

The final `lm_entries[ ]` variable is `parent_status`. This is normally `VALID_PARENT` unless the Landmark has answered no to a `parent_request_msg`. In that case, it is set to `NOT_VALID_PARENT` until another parent is found and verified, after which it is set to `VALID_PARENT` again. This prevents repeatedly asking the same Landmark to be a parent.

Finally, we keep information about our children in the variable `child_entries[ ][ ]`. `child_entries[ ][ ]` is indexed both by level and by label. The first variable in `child_entries[ ][ ]` is `child_id`, which is used to find other child information in `lm_entries[ ]`. The second variable is `child_state`. Like `parent_state`, it tells whether or not a child at a given level and with a given label exists, and if so, whether it has been verified. Unlike `parent_state` (except for root), `VACANT` is a valid steady state.

### 5.3.2 Messages Exchanged by the Hierarchy Algorithm

Table 4 shows the messages exchanged by the Hierarchy Algorithm. The `parent_request_msg` is used to request that another Landmark be your parent. The `parent_reply_msg` is the reply (yes or no) the parent gives to the child. These are exchanged directly with other Landmarks.

The `hier_update_msg` is used to disseminate Hierarchy information to other Landmarks. The Routing Algorithm disseminates the Hierarchy information, except for the radius, which it uses to determine which Landmarks receive the information. The Routing Algorithm receives Hierarchy

**Table 4**  
**Hierarchy Algorithm Messages**

MESSAGE	INFORMATION	COMMENTS
parent_request_msg	hier_addr level	to/from another Hierarchy Algorithm
parent_reply_msg	yes/no level label hier_addr lm_addr	to/from another Hierarchy Algorithm
hier_update_msg	hier_addr lm_addr global_status level child_array[ ][ ] radius distance	to/from Routing Algorithm (for spreading to other Hierarchy Algorithms)  to Routing Algorithm only from Routing Algorithm only
bind_update_msg	my lm_addr my child_array[ ][ ] all global lm_addr all distance to globals	to Binding Algorithm

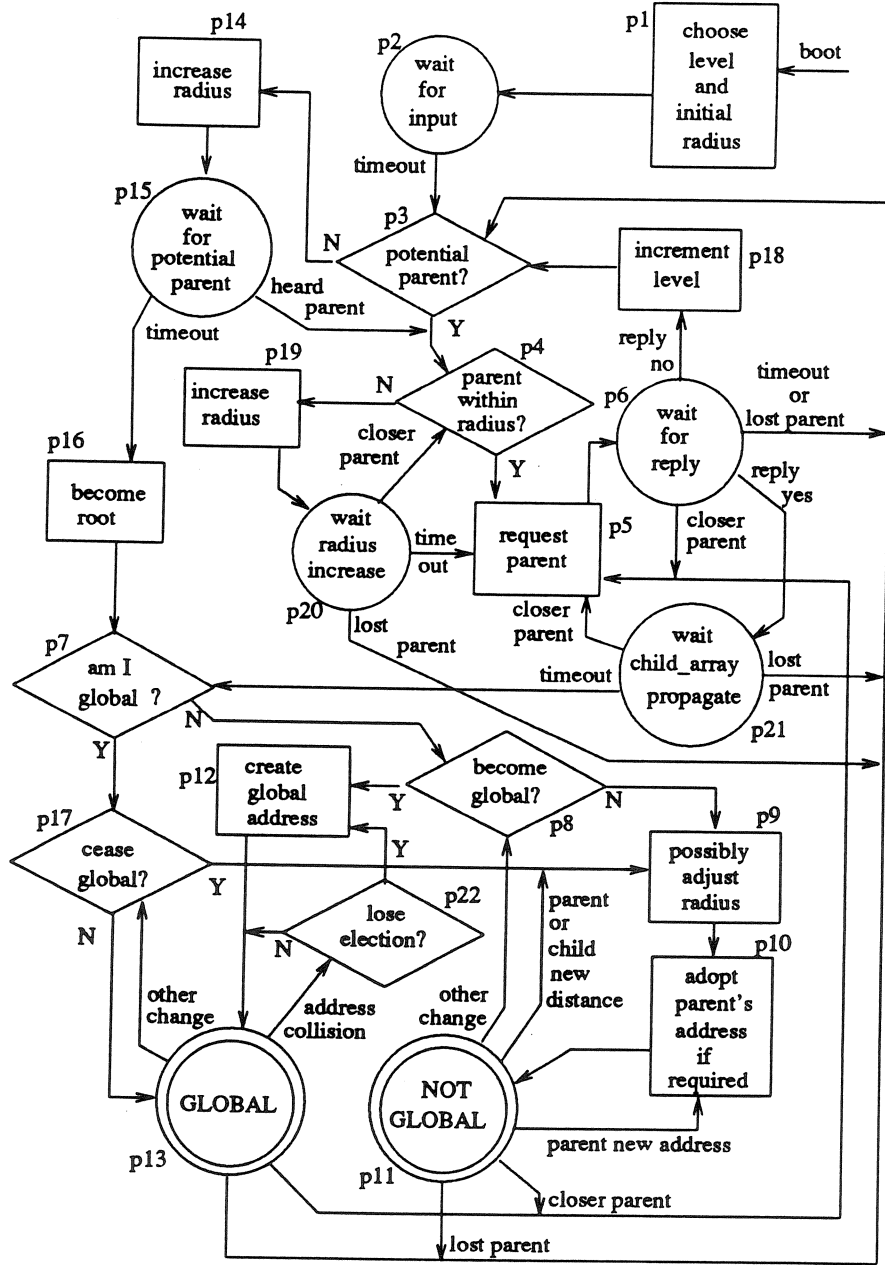
information from other Landmarks, and adds to that the distance to those Landmarks, which it calculates. The bind\_update\_msg is sent to the Binding Algorithm.

### 5.3.3 Detailed Hierarchy Algorithm Description

Figure 9 shows a more detailed Hierarchy Algorithm flow. The single circles are wait states. The boxes indicate actions taken. The diamonds are decision points. The double circles are steady states.

Upon boot, a level is randomly chosen, and an initial radius is calculated ( $p_1$ ). Of course all Landmarks are at least level 0. Higher levels are randomly chosen with decreasing probability. In our implementation, to choose a level, a Landmark initially sets variable level to 0. Then with probability  $1/C$  it becomes a level 1. If successful, then with another probability  $1/C$ , it becomes a level 2. This continues until either 1) the Landmark does not achieve the next highest level, or 2) the Landmark reaches the highest possible level. Therefore, a Landmark is level 0 with probability  $\frac{C-1}{C}$ , level 1 with probability slightly less than  $\frac{1}{C}$ , level 2 with probability slightly less than  $\frac{1}{C^2}$ , and level  $i$  with probability slightly less than  $\frac{1}{C^i}$ . In our implementation, we typically set  $C = 4$ ,

Figure 9  
 Detailed Hierarchy Algorithm Flow Diagram



and experimented with values of  $C = 2$  and  $C = 8$ . With  $C = 4$ , the probability of becoming a level 10 Landmark is around 1 out of  $10^6$ .

We then create a `hier_addr` commensurate with our level. If we are level 0, `hier_addr = 0`, if level 1, `hier_addr = 1`, if level 2, `hier_addr = 1.1`, and so on.

Finally, we calculate our initial radius. A Landmark needs to choose a radius that is large enough to cover all potential children with high probability, so that the children can discover the parent. Since this radius is quickly reduced when a parent is discovered, choosing a large initial radius doesn't cause much unneeded overhead except in the case where many Landmarks come up at once (on the order of within a minute of each other). In our implementation, we set `radius = 2` for level 0, `radius = 4` for level 1, and then incremented radius by 3 for each higher level. We think that increasing the radius by 2 for each higher level would probably be sufficient, even for networks with large diameter relative to the number of nodes.

After `hier_addr`, `level`, and `radius` are chosen, this information is handed to the Routing Algorithm, which distributes the information flood-style to neighbor Landmarks along with other distance-vector routing information. The neighbors flood it to their neighbors, and so on, until the radius is reached. All neighbors receiving the information store it in `lm_entries[ ]`, and act on it accordingly. Notice that other initial values sent are: `lm_addr = NULL`, `num_children = 0`, and `global_status = NOT_GLOBAL`.

Next, the Landmark enters the wait state `NB_DUMP_WAIT` (p2) (see Table 5). The purpose of this state is to obtain routing and hierarchy information from the neighbors. Typically, a Landmark comes up in a fully configured Landmark Hierarchy, and this allows the Landmark to do nothing until it learns of the whole hierarchy. Otherwise, a Landmark normally responds to Hierarchy information when it arrives. The length of the `NB_DUMP_WAIT` wait state `NB_DUMP_WAIT` is 1 second longer than the maximum time it takes to receive a full update from a neighbor. In our implementation, since we used a timer driven routing algorithm, we wait `NB_DUMP_WAIT = 2T_{update} + 1`, where  $2T_{update}$  is the maximum length of time it takes to pass a routing update one hop, and  $T_{update}$  is the average length of time it takes to pass a routing update one hop. With an event-driven update strategy, this information would be exchanged upon neighbor configuration.

The Landmark will, however, handle `parent_request_msg` from prospective children.

After obtaining full hierarchy information, the Landmark decides if it sees a potential parent by scanning the information in `lm_entries[ ]` and picking the closest potential parent (p3). A Landmark is a potential parent if 1) its level `lm_entries[potential_parent_id].level` is higher than the potential child's level and 2) it doesn't have a full compliment of children at level `level+1` (in other words, `lm_entries[potential_parent_id].num_children[level+1] < 7`). If it has a potential parent, then it 1) puts the id of the potential parent in `parent_id`, 2) sets `parent_state` to `PENDING_VERIFICATION`, and checks `lm_entries[potential_parent_id].distance` to see if the potential parent is within radius hops (p4).

Table 5  
Hierarchy Algorithm Wait States

Wait States		
State	Value	Description
NB_DUMP_WAIT	$T_{update} + 1$	After boot, to obtain neighbor routing tables (p2)
RADIUS_WAIT	$T_{update} \times parent\_dist + 1$	Increase radius to cover parent (p20) Can be canceled if hear of closer parent
PARENT_REPLY_WAIT	10	Timeout in case requested parent (p6) never answers
CHILD_ARRAY_WAIT	$T_{update} \times parent\_dist + 1$	Wait for parent to propagate child_array info (p21)
ROOT_WAIT	$T_{update} \times radius + 10$	Wait to verify root status (p15)
CHILD_VERIFY_WAIT	$T_{update} \times child\_dist + 60$	Wait for new child to be verified (c5) Canceled if child verified

If the potential parent is not within radius hops, then it must increase radius to cover the potential parent (p19), and then wait (RADIUS\_WAIT) for the Routing Algorithm to increase distribution of the Hierarchy information (p20). By doing this, we insure that the parent has the child in `lm_entries[ ]` when it receives a `parent_request_msg` from the child. We had initially designed the Hierarchy Algorithm so that the parent could handle a parent request even without having the requesting child in its `lm_entries[ ]` data base, but this overly complicated the Hierarchy Algorithm because the parent kept additional state and made additional checks about all of its children.

In our implementation, radius is set to  $\max(distance \times 1.5, distance + 1)$ . The 1.5 multiplier is a good balance between robustness and small routing tables (Tsuchiya, 1987a). The +1 add insures that there is at least one additional hop of coverage beyond the parent. We did not experiment with different values. The RADIUS\_WAIT time is  $(T_{update} \times parent\_dist) + 1$ .

During RADIUS\_WAIT, it is possible to hear of a significantly closer potential parent. In our implementation, a potential parent is significantly closer if  $(new\_parent\_distance \times 1.5) < old\_parent\_distance$ . This multiplier is meant to reduce the probability of flopping back and forth between two parents because a link is going up and down. We did not experiment with this value. If a new parent is closer, then we cancel the RADIUS\_WAIT, replace the old parent\_id with the new one, and check to see if the new potential parent distance is less than radius (p4). In any event, the Landmark will eventually determine that a potential parent is within its radius, and send a `parent_request_msg` to the potential parent (p5).

During RADIUS\_WAIT, it is also possible for the Routing Algorithm to indicate that the parent has become unreachable (either because it has crashed, or the topology has caused it to be



further away than its radius). In this case, the Landmark changes parent\_state to VACANT, and again searches for the closest potential parent (p3).

Next, the Landmark waits for the parent\_reply\_msg (p6, PARENT\_REPLY\_WAIT). During PARENT\_REPLY\_WAIT, four things may happen.

1. It is again possible that the Landmark will hear of a closer potential parent. This time, the Landmark does not need to check that radius is big enough. It simply replaces parent\_id, sends a parent\_request\_msg to the new potential parent (p5), and enters PARENT\_REPLY\_WAIT again (p6).
2. The parent answers NO. (In our implementation, the PARENT\_REPLY\_WAIT timer is set to 10 seconds. In real implementations, this would be set higher, and the Landmark would be more persistent about sending request\_parent\_msg's to get a response.) In either case, the action is the same. The Landmark 1) changes parent\_id to NULL, 2) sets all lm\_entries[ ].parent\_status to VALID\_PARENT, 3) increments its level (p18), and 4) searches for a new potential parent (p3).

This is the only occasion (other than at boot time) where a Landmark may change its level. The idea here is that if a Landmark has a full set of children, then there are probably not enough Landmarks at that level in that area of the network. Rather than simply be adopted by a parent even further away, it makes more sense to just go ahead and become a higher level Landmark.

Notice that there is no way for a Landmark to lower its level. There was a mechanism in the early designs where a Landmark would lower its level if it had too few children. This of course added complexity. We saw little use for it, because 1) the incrementing of levels is self limiting because more Landmarks at a level means fewer children per Landmark, and 2) even if the levels were somehow getting too high, Landmarks would crash and start over at lower levels long before the hierarchy got really top heavy. In fact, Landmark levels can be lowered by management intervention if necessary.

3. The parent is lost (Routing Algorithm indicates that parent is unreachable), or the Landmark times out. The Landmark changes parent\_state to VACANT, and again searches for the closest potential parent (p3).
4. The parent answers YES. First, the Landmark waits for the parent to propagate its new child\_array information before fully accepting the new parent (p21). The reason for this is that a child knows that it is no longer a child if the parent's child\_array no longer includes the child. The existence of the child\_array information verifies the parent for the child.

After CHILD\_ARRAY\_WAIT, the Landmark accepts the parent by changing parent\_state to OCCUPIED. The Landmark also changes hier\_addr to match that of the

parent's above its level, and uses the label given it in the `parent_reply_msg` to fill in `hier_addr` at its level.

At this point, the Landmark has found a place in the Landmark Hierarchy, and must now get a Landmark Address. First, the Landmark must decide if it should become global if it is not (p7, p8), or cease being global if it is (p7, p17). We discuss exactly how this decision is made in Section 5.3.3.1. Ultimately, however, there are two possibilities. Either the Landmark is or becomes global and it keeps or picks its own `lm_addr`, or it is or does not and adopts the parent's `lm_addr`.

If the Landmark was not global and becomes global, it creates a global address for itself (p12), and waits for any change (p13). This is a steady-state, and there is no timer associated with it. To create a global address, the Landmark must choose a global label. It picks a global label by hashing its own id. If the result matches a global label already chosen by some other Landmark, it chooses another global label, for example by incrementing its id and hashing again.

If the Landmark was not global and chooses not to become global, or if it was global and chooses to cease being global, then it 1) possibly modifies radius to cover the parent appropriately ( $\text{radius} = \max(\text{distance} \times 1.5, \text{distance} + 1)$ ) (p9), and 2) creates `lm_addr` by assuming the parent's `lm_addr` above level, and keeping its own at level and below (p10). It then enters steady state (p11).

If the Landmark was global and decides to stay global (p17), then it enters steady state without doing anything else. This would be the case if, for instance, a global Landmark lost its parent and found a new one. Because it stays global, neither it nor any of its offspring change addresses.

Before discussing what happens in steady state (p13 or p14), let's consider a thread in the logic that we have bypassed so far. After the initial wait `NB_DUMP_WAIT` (p2), or after the Landmark has lost a parent, it looks in the `lm_entries[ ]` data base for a potential parent. If the Landmark doesn't find one, then it increases radius (p14) and waits (`ROOT_WAIT`) in case another Landmark is doing the same (p15). If during `ROOT_WAIT` it hears a potential parent, then it tries to join that parent as already described (p4, etc.).

If the Landmark does not hear of any potential parent, it assumes that it is a root of the hierarchy (p16), and either becomes global if it is not (p8, p12, p13), or stays global if it is (p17, p13) (a root is always global). If in fact after all this there was a potential parent in the network that had not been discovered, the potential parent will be discovered after it becomes a root (and therefore becomes a global). In this case, the Landmark behaves just as it would if it had found a closer parent had it not been a root (go to p5).

Now we may discuss what happens in steady state (either p13 or p14). There are two events that evoke the same response whichever of the two states the Landmark is in: a closer parent (or any parent, if root), or losing the existing parent (not applicable if root). In the former case, the Landmark requests the new potential parent (p5). In the latter, it looks for a new potential parent (p3). Both of these cases have been discussed.

If the steady state is NOT\_GLOBAL, then the Landmark must worry about radius. If the Routing Algorithm reports a new distance for the parent or child, then the Landmark must re-evaluate radius (p9). For example, if the parent is further away, but not far enough to warrant getting another parent, the Landmark may need to increase radius. Also, if the parent gets a new lm\_addr (because, say, its parent got a new lm\_addr), then the Landmark must follow suit and get a new lm\_addr (p10).

In the GLOBAL steady state, the Landmark must worry about other global Landmarks. If it discovers another global Landmark with the same global label, then the two global Landmarks must resolve the address collision. This may occur if two Landmarks become global at the same time, or more commonly if two networks or network partitions are joined. The two global Landmarks resolve the collision by first comparing their levels. The Landmark with the lower level must change its global label (p12), because that will usually result in the smallest number of lm\_addr changes. This will occur if for example a small network partition rejoins the network. If the levels are the same, then they must elect a winner. In our simulation algorithm, we do this by simply comparing id's. The higher one must change its address. A more fair way (remove advantage of having a low id) is to have each Landmark modify its own id as a function of the other Landmark's id (for instance, shift the other Landmark's id left 1, and logically OR it with our own), hash the results of the modified id's, and compare the hash results. Either way, the main result, that all lm\_addr's are unique, is accomplished.

Both states have an event nebulously labeled "other change". This refers to some change in the lm\_entries[ ] data base that changes the number of known Landmarks, either directly or because a Landmark declares more or less children. Since this information in part determines global\_status, the Landmark must re-evaluate global\_status (p17 for the GLOBAL, p8 for NOT\_GLOBAL). We have already discussed the possible outcomes of that evaluation.

**5.3.3.1 Evaluating global\_status.** The goals behind choosing global Landmarks are to 1) choose the appropriate number of global Landmarks (roughly  $\sqrt{N}$ ), and 2) choose the global Landmarks so that each has roughly the same number of offspring. Of course, we would like this to be done with low overhead, with no oscillation, and with simplicity.

Several different algorithms were designed. We are not totally happy with any of them, including the one we ended up with. While what we have seems workable, we believe that this algorithm could use more work.

Choosing globals requires the following steps:

1. Each node calculate the number of nodes in the network ( $N$ ), and thus the number of required globals ( $\sqrt{N}$ ).
2. Each node determine if more or less globals are needed by comparing the actual number of globals with the required number of globals.
3. If more or less globals are required, each node determines if it should become or cease being global.

Each step is independent of the others in that it doesn't matter how the previous step is accomplished, only that the appropriate information be determined by the previous step. Step 2 is trivial, since the number of globals in the network can be counted in the `lm_entries[ ]` data base. Also, it is trivial to pick certain globals—for instance, the roots and their children are always global.

The initial approach to Step 1 (Tsuchiya, 1987b) had each Landmark calculate the number of offspring it had, and pass that information up to its parent, who would make its calculation for its offspring, pass that up to its parent, and so on. When each global had calculated its number of offspring, it would include this in its hierarchy information. Each Landmark counted the number of offspring of each global, and then knew the total number of nodes. Each Landmark's calculation would have some hysteresis built in so as not to send updates all the way up tree every time a child went up or down.

This approach is actually pretty good, and in retrospect is probably the best way to do step one. We tried another approach because we wanted to see if we could do away with the child-to-parent messages altogether.

In an initial approach to step 3, the Landmarks would create a strict ordering of which would become or cease being global next. All globals at the lowest global level would order themselves by node id. Then, each global would in order assign one of its children global. When all had assigned one child global, each would assign another global, and so on until all children were global. Then the process would continue at the next lower level. They would cease being global in the reverse order. This algorithm, however, seems much more complex than is necessary.

Our current implementation estimates the number of nodes in the network (step 1) by using the `child_array[ ][ ]`'s for each of its `lm_entries[ ]`. It estimates the average number of children at each level directly from the `child_array[ ][ ]`'s. First, it counts all of the roots and uses that as the number of highest level ( $i = H$ ) Landmarks. Since it knows from the `child_array[ ][ ]` how many children a root has, it knows how many  $H-1$  Landmarks there are. From the level  $H-1$  Landmarks in `lm_entries[ ]`, it finds the average number of children, and estimates the number of level  $H-2$  Landmarks by assuming that all level  $H-1$  Landmarks have the same average number of children. This continues all of the way down the hierarchy to to local level to estimate the total number of nodes in the network  $N_{est}$ . We then calculate the required number of globals as  $G_{req} = \sqrt{N_{est}}$ .

Count the number of globals in `lm_entries[ ]` ( $G_{act}$ ). If  $G_{req} > k_1 G_{act}$  or  $G_{req} < k_2 G_{act}$ , then  $G_{req} - G_{act}$  globals are needed (Our implementation uses  $k_1 = 1.5$  and  $k_2 = .7$ . This prevents oscillation.)

If globals are needed or must be removed, then each Landmark determines if it must change its global status. In our implementation, we first make a simple check to see if we are at the level at which globals must be added or removed. If we are not at that level, then we do nothing. If we are, then we use the previous estimate of the number of nodes at our level, determine what percentage of these nodes must become or cease being global, and then randomly become or cease being global with a probability equal to that percentage.

**5.3.3.2 Child Algorithm.** Figure 10 shows the algorithm used for dealing with children. This algorithm is run separately for each child in `child_entries[ ][ ]`. Upon booting, each Landmark has no children (c1). After it chooses and advertises a hierarchy level, a Landmark may receive `parent_request_msg`'s. When it does, it checks to see if its level is high enough (c2) or whether it already has a full quota of children (c3). Either of these two checks can result in a no answer to the child (c8). Otherwise, it answers yes with one of its available labels (c4), sets `child_state` to `PENDING_VERIFICATION`, and waits for child verification through the Routing Algorithm (`CHILD_VERIFY_WAIT`) (c5). Verification comes when the child's `hier_addr` at level and above matches `hier_addr`. If the child is never verified (timeout), then `child_state` is set back to `VACANT` (c1). If the child is verified, `child_state` is set to `OCCUPIED` (c6), and a check is made to see if the radius must be extended to include the new child (c7). If the Routing Algorithm indicates that the child has disappeared, or the child has a new `hier_addr`, then `child_state` is set back to `VACANT` (c1).

#### 5.4 Algorithms We Tried But Didn't Like

This algorithm is different from the one documented in the previous report (Tsuchiya, 1987b). The main difference is the elimination of elections. In fact, the first algorithm we implemented and tried to simulate included elections. The election criteria was not that of distance, as described in the previous report, but the number of children per parent. Nonetheless, the complexity required to run the elections was enough that simply debugging the implementation on the simulator was quite difficult.

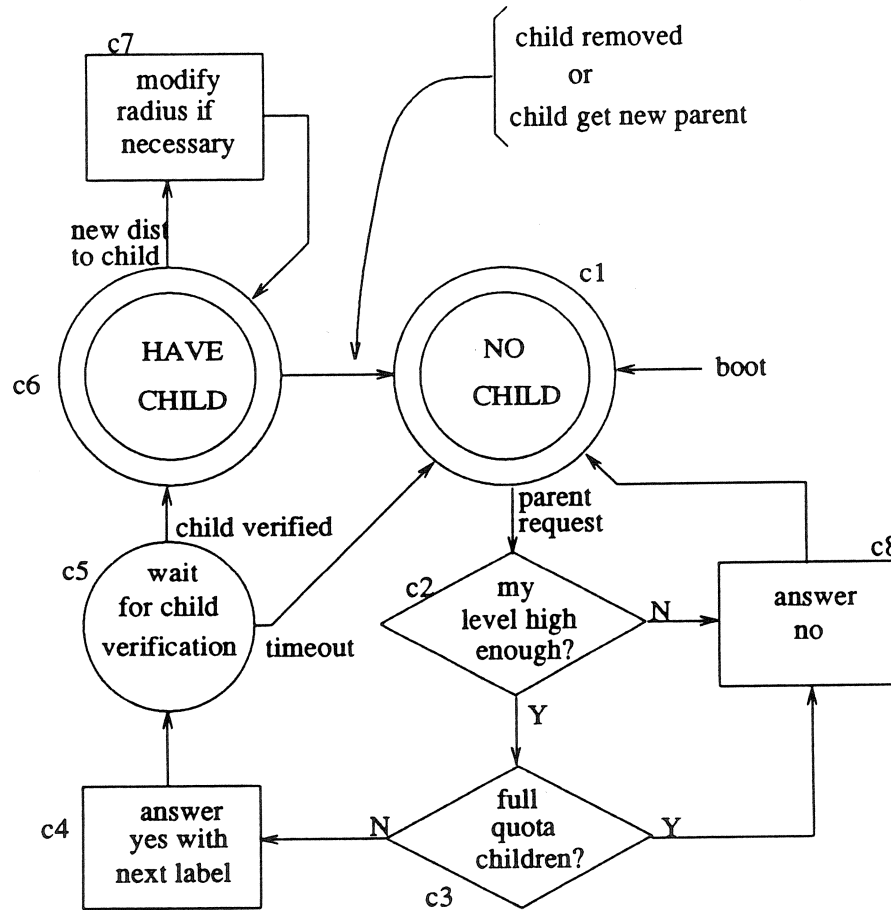
In addition, the first implementation included a means of keeping two parents at once, and thus having two addresses at once. This was done by doubling the label space of the first label below the global label (from 3 bits to 4). Values 1 through 7 were for the primary address space, and values 8 through 15 were for the secondary address space.

The purpose of this was to send both addresses to the binding algorithm in advance of losing either parent so that rebinding could be smoothed out over time, thus avoiding the surge in binding messages that we like to avoid. This was again too complex. We had to make sure, for instance, that the second parent (or uncle, as it was called) was reachable via a neighbor different from that of the primary parent so that one link crash wouldn't partition both parents. Further, it doubled the binding traffic required on the average, and didn't always prevent surges.

#### 5.5 Algorithms We Didn't Try But Would Like to Use

There are several improvements we could make. The main one is to use a better Routing Algorithm. This is even more true for the Binding Algorithm. Table 5 shows the timer values for the various wait states. Nearly all of them depend on a factor of the Routing Algorithm parameter  $T_{update}$ . Three of them in particular (`NB_DUMP_WAIT`, `RADIUS_WAIT`, and `CHILD_ARRAY_WAIT`) normally exit via the timeout rather than some event. Also, most new information, even event-driven information, about other Landmarks is learned from the Routing Algorithm. If an event-driven Routing Algorithm was used,  $T_{update}$  could be reduced from seconds to milliseconds, thus speeding the whole algorithm.

Figure 10  
Child Algorithm



Surprisingly, the count-to-infinity problem did not bother the Hierarchy Algorithm (although without a certain modification it was murder on the Binding Algorithm). As the counting goes up, an entry for the counting-up destination will appear and disappear, each time with a larger distance. However, it doesn't have to count up very far before its children will have abandoned it for a closer parent, and so the counting-up Landmark effectively leaves the hierarchy well before it is done counting to infinity.

In our current algorithm, we only send one `parent_request_msg`. Our simulation rarely drops packets, and so this was not particularly a problem. Generally, however, a transport connection between the child and the parent should be set up for the duration of the request-reply transaction to deal with lost packets.

We could eliminate the RADIUS\_WAIT state if when a parent\_request\_msg was sent, the backwards route to the child was remembered along the way. This way, the parent, and all Landmarks between the parent and the child, would know the route to the child, and the parent\_reply\_msg could go out right away. To do this, instead of sending the parent\_request\_msg through the network layer directly to the parent, it could visit each Hierarchy Algorithm along the way, which would then update the Routing Algorithm with the appropriate information.

In our implementation, the Landmarks at the highest two levels are automatically global. In a real implementation, this should be increased to the highest 3 or 4 levels, depending on the number of children per parent, so that 60 or 70 Landmarks automatically become global. If we had done this in our simulations, then all of the Landmarks would have become global, and the simulations wouldn't have been very interesting. Given that the intended use of Landmark Routing is for networks with thousands of Landmarks, 60 or 70 automatic globals is appropriate. Put another way, if a network can't handle the traffic generated by 60 or 70 globals, then it probably shouldn't be using Landmark Routing at all.

In our implementation, each parent is responsible for telling its children of its new address. This is inefficient in some cases, because when a Landmark gets a new address, all of its offspring will of course be getting the new address, and they can all determine what their new address is directly from the Landmark getting the new address. Therefore, rather than filter this information down through each generation of Landmarks, the Landmark getting the new address can flood it to all of its offspring, thus speeding convergence.

Another real-network consideration is that of preventing a single Landmark from causing too much havoc, say by becoming a high level Landmark, adopting many children, and then disappearing, causing those children to thrash between one address and another. Or, a Landmark could cause havoc by pretending to be many global Landmarks, thus causing many others to go away. Or, a Landmark could hand out duplicate addresses, thus denying service to some other Landmarks.

The whole key behind Landmark Routing is that each Landmark is capable of making certain decisions without consulting with other Landmarks. This is what simplifies the algorithm. However, it is this very key that, unchecked, gives a Landmark the power to cause problems.

The first two problems could potentially be handled through some kind of authentication. The authentication could only be enforced for only those Landmarks that wanted to become global, thus lessening the load on the trusted servers. The trusted servers themselves could become known by becoming high level Landmarks themselves, or by widely establishing binding information about themselves.

When a Landmark wished to become global, it would first authenticate itself with a trusted server. The server would then determine if it is appropriate for the Landmark to be at the level it has chosen for itself, and check to see whether the Landmark has been becoming global too often. If the global request is ok, then the trusted server would give the global an encrypted key that the global would put in its hier\_update\_msg, thus allowing other Landmarks to decide whether to accept the new global or not. This process could conceivably be repeated on a local basis for lower

levels. This authentication weakens the robustness of Landmark Routing because the authentication servers become potential bottlenecks and points of failure.

The neighbors of a parent can detect when it has handed out duplicate addresses because they will hear routing updates from different Landmarks (the parent's children) but with identical addresses. Therefore, the third problem can at least be detected if Landmarks always check new entries or changes in entries for duplicate addresses.



## 6 HIERARCHY ALGORITHM PERFORMANCE

This Section has three parts: 1) An analysis of the hierarchy algorithm, 2) the results of static simulations of the hierarchy algorithm, and 3) the results of dynamic simulations of the hierarchy algorithm.

### 6.1 Hierarchy Algorithm Analysis

In this analysis we are interested in three things:

1. The likelihood that a topology change will result in a hierarchy change. Here we consider both parent changes and address changes.
2. The time it takes for the hierarchy to change. Here, we only consider the time it takes from the topology change to the establishment of new addresses. We don't consider the additional time it takes for the binding algorithm to converge.
3. The amount of traffic generated when the hierarchy changes. Again, here we only consider traffic germane to the hierarchy. We do not consider binding traffic.

#### 6.1.1 Analysis of Hierarchy Changes due to Topology Changes

We first consider the impact on the hierarchy of a link going down. Assume that each level  $i-1$  Landmark ( $LM_{i-1}$ ) is an average of  $d_i$  hops from its parent  $LM_i$ . There are therefore an average of  $d_i$  links between any  $LM_{i-1}$  and its parent  $LM_i$ .

We assume that if a link between an  $LM_{i-1}$  and its parent goes down, then the  $LM_{i-1}$  will get a new parent. In other words, we assume that there are a negligible number of multiple paths of similar lengths to other nodes. This assumption is less valid as node degrees go up with respect to the network diameter. Since more paths to a parent reduce the number of parent changes, by ignoring multiple paths our analysis will give worse results than should be seen in actual practice.

Assume there are  $L$  links in the network. Then, for any  $LM_{i-1}$ , if a link goes down, there is a  $d_i/L$  probability that the down link was one of the links between the  $LM_{i-1}$  and its parent, and therefore there is a  $d_i/L$  probability that a parent change occurs. Assume that there are  $T_i$  level  $i$  Landmarks. Then each link going down will result in  $T_{i-1} \frac{d_i}{L}$  new parents at each level  $i$ , and

$$P_{link} = \sum_{i=1}^H T_{i-1} \frac{d_i}{L}$$

total new parents, where  $H$  is the highest hierarchy level.

This is an average over all links. In fact, the closer the crashed link is to a high level Landmark, the more parent changes there will be.

Not every Landmark that gets a new parent will get a new address. Global Landmarks keep their own addresses. Also, some Landmarks will get new addresses even though they kept the same

parents, because their parents themselves get new addresses. We assume that if a Landmark gets a new address because it is separated from its parent, then all of its offspring will receive new addresses.

Assume each  $LM_i$  ( $i > 0$ ) has  $C$  children at level  $LM_{i-1}$ . Then each  $LM_{i-1}$  has a total of  $C^{i-1}$  offspring (including itself), and  $T_{i-1} = \frac{N}{C^{i-1}}$ . The total number of address changes due to a crashed link, then, is

$$\begin{aligned} A_{link} &= \sum_{i=1}^{H_G-1} \frac{N}{C^{i-1}} \frac{d_i}{L} C^{i-1} + k_{H_G} \frac{N}{C^{H_G-1}} \frac{d_{H_G}}{L} C^{H_G-1} \\ &= \frac{N}{L} \left[ \sum_{i=1}^{H_G-1} d_i + k_{H_G} d_{H_G} \right], \end{aligned}$$

where  $H_G$  is the hierarchy level at which some  $LM_i$ , ( $i = H_G$ ), become global. Since it is possible that not all  $LM_{H_G}$  are global, the last term is multiplied with a factor  $k_{H_G}$  representing the percentage of non-global Landmarks at level  $H_G$ .

$H_G$  and  $k_{H_G}$  are determined as follows. We know that there are  $\sqrt{N}$  total global Landmarks. It follows, then, that the fraction of  $LM_i$  that are global is  $\frac{\sqrt{N}}{T_i}$ , and the fraction that are not is  $k_i = \frac{T_i - \sqrt{N}}{T_i}$ .  $H_G$  is the level where  $k_{H_G} > 0$  and  $k_{H_G+1} < 0$ .

When a link crashes, the only hierarchy changes are those where a child is separated from its parent. When an  $LM_i$  crashes, we have in addition to these changes the fact that all of that node's children must find new parents. An  $LM_i$  is also an  $LM_{i-1}$ , and  $LM_{i-2}$ , and so on. Therefore, an  $LM_i$  has  $iC$  children.

The number of nodes that get new parents after an  $LM_i$  crashes is

$$\begin{aligned} P_{node,i} &= \sum_{k=i}^H T_{k-1} \frac{d_k - 1}{N} + iC \\ &= \sum_{k=i}^H \frac{d_k - 1}{C^{k-1}} + iC \quad (i > 0), \end{aligned}$$

where the first term is due to nodes being separated from their parents, and the second term is due to the parent itself crashing. The first term was derived similarly to that for links, except that 1) there are only  $d_i - 1$  nodes between an  $LM_{i-1}$  and its parent  $LM_i$ , and 2) only nodes at or above the level of the crashed node can be separated from their parent (those below that level are the children of the crashed node itself). The second term is simply the average number of children an  $LM_i$  has.

The number of Landmarks that get new addresses when an  $LM_i$  crashes, then, is

$$\begin{aligned} A_{node,i} &= \sum_{k=i}^{H_G-1} \frac{d_k - 1}{C^{k-1}} C^{k-1} + k_{H_G} \frac{d_{H_G} - 1}{C^{H_G-1}} C^{H_G-1} + \min(C^i, C^{H_G}(1+kC)) \\ &= \sum_{k=i}^{H_G-1} (d_k - 1) + k_{H_G} (d_{H_G} - 1) + \min(C^i, C^{H_G}(1+kC)) \end{aligned}$$

The last term is the number of offspring whose addresses are dependent on the crashing node. An  $LM_{H_0}$  has an average of  $C^{H_0}$  offspring, all of which are dependent on the  $LM_{H_0}$  for their addresses. An  $LM_i$ , for  $i > H_G$ , can have still more address-dependent offspring. An  $LM_{H_0+1}$  is also an  $LM_{H_0}$  and has  $C^{H_0}$  offspring at that level. But since not all  $LM_{H_0}$  are global, an  $LM_{H_0+1}$  will additionally have an average of  $kC$  level  $H_G$  children that are not global, and who depend on it for their addresses. Therefore, the maximum possible value for the last term is  $C^{H_0}(1+kC)$ .

It seems likely that nodes and links coming up will not have as much impact as nodes and links going down. When a node crashes, its children have no choice but to look elsewhere for a new parent. When a new node comes up, or a link coming up causes a node to become closer, only if it is closer by some percent than existing parents will nodes change to it. In other words, there is hysteresis built into the process of choosing new parents: a node will stay with its current parent unless a new parent is closer by some percentage. We do not model this in our analysis.

In any event, we can think of no reason that nodes and links coming up should cause more changes than nodes and links crashing. Therefore, we will use the above analysis for both deletions and additions, again giving us an overestimate of hierarchy changes.

To solve the above equations, we need to know  $d_i$ . If there are  $N$  nodes and  $T_i$  Landmarks at level  $i$ , then each  $LM_i$  has  $\frac{N}{T_i}$  nodes closer to it than to any other node. There is therefore some distance  $\hat{d}_i$  so that

$$v(\hat{d}_i) = \frac{N}{T_i},$$

where the function  $v(x)$  is the number of nodes within  $x$  hops. We have an approximation of  $v(x)$  (Tsuchiya, 1987a).

We can approximate  $\hat{d}_i$  by finding two values  $v(h_1)$  and  $v(h_2)$  such that  $v(h_1) \leq \frac{N}{T_i} \leq v(h_2)$ , and then interpolating between  $h_1$  and  $h_2$  to get  $\hat{d}_i$ . Since  $\hat{d}_i$  represents the furthest a node can be away from an  $LM_i$  without being closer to another  $LM_i$ , we know that  $\hat{d}_i \geq d_i$ . By using  $\hat{d}_i$  as an approximation for  $d_i$ , we overestimate the number of hierarchy changes—actual results should be better than our estimate.

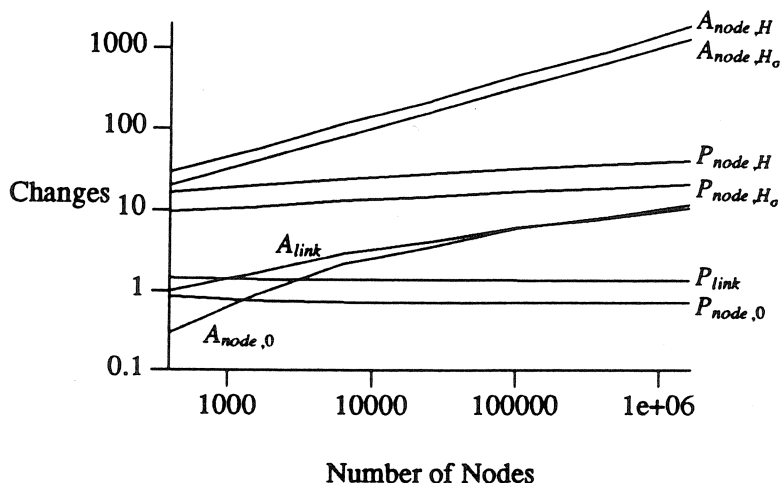
Figure 11 plots  $P_{link}$ ,  $A_{link}$ ,  $P_{node,i}$  and  $A_{node,i}$  for  $i = 0$ ,  $i = H_G$ , and  $i = H$ . These values are for node degree 3 and diameter  $3\log_2 N$ .

### 6.1.2 Analysis of Traffic Generated by Hierarchy Changes

Now, we consider how much traffic is generated by hierarchy changes. We do not consider traffic generated by the routing algorithm or the binding algorithm.

When an  $LM_{i-1}$  gets a new parent  $LM_i$ , it sends a parent request and receives a parent reply—two messages. The old parent receives and sends no messages. It learns of the disappearing child

Figure 11  
Estimated Hierarchy Changes



either through timeout or through the normal operation of the routing algorithm. These two messages travel a distance of  $d_i$  hops. Therefore, a change in parent generates  $2d_i$  packets.

When an  $LM_i$  gets a new address and a new parent, it must inform all of its  $C^i$  offspring of that new address. This can be done with an efficient flood, where each message traverses each link twice, as in the ARPANET SPF algorithm (McQuillan, 1980). In fact, it should be possible to use the information in the routing tables to make a spanning-tree multicast, but we will be conservative and assume the flood. There are  $C^i$  nodes that must receive the message, and each node shares  $E$  links with neighbors, the message must cross  $\frac{EC^i}{2}$  links twice, so  $EC^i$  packets are generated.

When an  $LM_i$  gets a new address but without getting a new parent, it is because its parent got a new parent, or its parent's parent got a new parent, and so on. There is no reason for an  $LM_i$  with a new address but not a new parent to tell its offspring of the new address. The ancestor that ultimately got the new parent and new address will do so. Therefore, of the  $\min(C^i, C^{H_o}(1+kC))$  nodes in the third term of  $A_{node,i}$  that get new addresses, only  $C \min(i, H_G + k)$  of them must actually flood the new address information out.

A change in a link causes  $X_{link} = P_{link} + A_{link}$  hierarchy changes, resulting in

$$X_{link} = P_{link} d_i + A_{link} EC^i$$

packets. A change in an  $LM_i$  causes

$$X_{node,i} = P_{node,i} d_i + EC^i \left[ \sum_{k=i}^{H_o-1} (d_k - 1) + k_{H_o} (d_{H_o} - 1) + C \min(i, H_G + k) \right]$$

packets.

The second term (floods due to address changes) is the dominant term in both these equations. However, this flooding is spread evenly over the nodes that receive the message. Therefore, *most links transmit only 2 packets*. If a higher level Landmark crashes, more packets will be generated, but they will be spread over a larger number of nodes.

### 6.1.3 Analysis of Convergence Time

Finally, we are interested in the length of time it takes to converge. The usual sequence of events from booting to complete reconfiguration of the hierarchy and the times associated with each event are given in Table 6.

*Table 6*  
**Hierarchy Change: Usual Sequence of Events**

Step	Event	Time
1	Choose radius	0
2	Get routing info from neighbor	$T_r + 1$
3	Establish radius	$(T_{update} r_i) + 1$ $(r_i = 1.5d_i)$
4	Request parent/get reply	$d_i T_{packet}$
5	Advertize address	$(T_{update} d_{i-1}) + 1$
6	step 4 for new children	$d_{i-1} T_{packet}$ (or less)
7	step 5 for new children	$(T_{update} d_{i-2}) + 1$ (or less)

$T_r$  is the maximum time to send a routing update one hop, and  $T_{update}$  is the average time to send a routing update one hop.  $T_{packet}$  is the average time to send a packet one hop.

Steps 1 through 3 occur when the node boots. Steps 4 and 5 occur when a node tries to obtain a new parent. Of course, the booting node must execute steps 4 and 5, and so must all nodes that become children of the booting node (steps 6 and 7). Therefore, the time from boot to convergence of the hierarchy is:

$$4 + T_{update} (1 + 1.5d_i + d_{i-1} + d_{i-2}) + T_{packet} (d_i + d_{i-1})$$

If we assume an event driven routing algorithm with  $T_{update} = .5\text{sec}$ , assume  $T_{packet} = .5\text{sec}$  as well, and assume that  $d_i = 2d_{i-1}$ , then we get  $4.5 + 2d_i$  seconds. If  $d_i$  is 10 hops, then the hierarchy can converge in about 25 seconds (this would be a very high level Landmark). If  $d_i$  is more like 3 hops, then the hierarchy can converge in about 10 seconds.

The major factor in convergence time is  $T_{update}$ . If the routing algorithm is slow, as was the case with our simulations, then convergence will be proportionally slow. Furthermore, if something unusual happens, like the potential parent crashing after the parent request but before the parent reply, then it will take longer.

## 6.2 Static Simulation Results

We performed static simulations of 100, 200, and 400 node networks to determine the number of hierarchy changes that result from topology changes. In particular, we experimented with five networks. Three of them had diameters of 20 and average node degrees of 3. In addition, one of the 200 node networks had a diameter of 40 (and node degree 3), and the third 200 node network had a node degree of 6 (and a diameter of 20).

In our experiments, we pick Landmarks by randomly choosing a level for each node according to the number of children per Landmark. We then pick the  $\sqrt{N}$  highest level Landmarks and label them global. We then assign each  $LM_{i-1}$  to its closest  $LM_i$ . We then change the topology by adding and removing links, local Landmarks ( $LM_0$ ), global Landmarks, and root Landmarks. Finally, we determine how many Landmarks have new parents and new addresses.

For a given set of parameters, we do this 500 times, keeping the average and maximum values for the 500 runs. This set of 500 runs is one experiment.

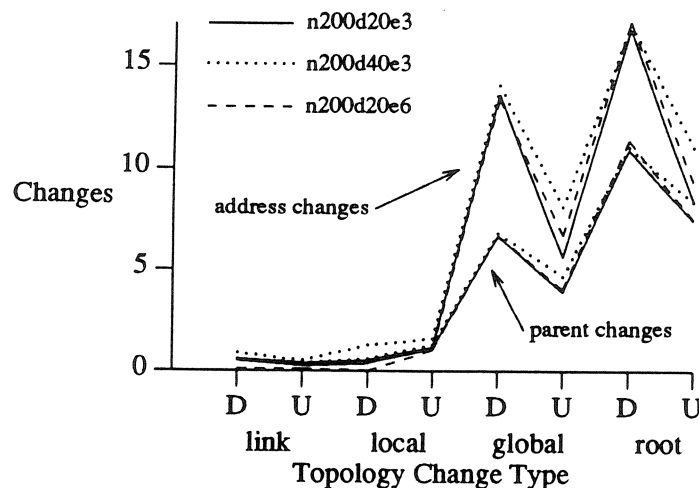
In addition to the different network types, we manipulated 1) the number of topology changes (crashed or brought-up nodes or links) (1, 2, 4, and 8), 2) the amount a potential new parent must be closer than the existing parent before it will be chosen (0%, 25%, 50%, 100%), and 3) the average number of children per Landmark (2, 4, 8, and 16). The control experiment was 1 change, a 50% new parent distance threshold, 4 children, and the 200 node network with diameter 20 and average node degree 3. Each other experiment changes only one parameter.

In most of our graphs, we show only average performance. In general, the maximum number of changes for 500 runs is as much as four or five times greater than the average. Figure 12 shows 1) the difference between average and maximum values, and 2) the difference between individual experiments with the same parameters (D means down, where we crashed a node or link, and U means up). We see that the maximum value can be significantly higher than the average. The maximums seen here are typical of all of our experiments, and so we don't bother to show them each time.

The dashed line in Figure 12 is the same experiment as the solid line, but with a different random number generator seed. The dotted line is for an experiment with the same parameters as the other two, but using a different network topology (although the number of nodes, diameter, and node degree are the same). We see that the average values are virtually identical. Therefore, we can trust that the rest of our results are accurate (not dependent on random variation). The maximum values are spread out a bit, but still relatively close to each other.

Figure 13 shows the difference between networks with different numbers of nodes. We show both the number of parent changes and address changes. The main thing to note is that the

Figure 12  
Effect of Network Type on Hierarchy Changes



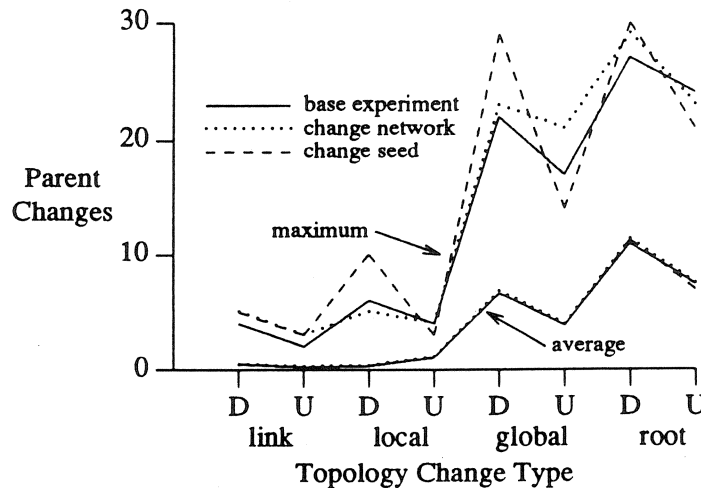
difference in magnitude between the parent and address changes is nowhere near the difference in magnitude between the number of nodes. Our equations state that the changes should never be greater than  $O(\sqrt{N})$ , because of the global Landmarks. For the crashing global scenario, we see on the average that this is exactly how many Landmarks get new addresses. For the crashing root scenario, more Landmarks get new addresses because the algorithm doesn't assign exactly  $\sqrt{N}$  offspring to each global, and since the roots are higher in the hierarchy than most globals, they have more offspring. Still, the number of Landmarks that get new addresses even for the root crashing case is nowhere near the difference in magnitude between the number of nodes.

Figure 14 shows the difference between networks with the same number of nodes but different node degree and diameter. Figure 14 shows very little difference. This means that, at least for a modest range of values, one probably need not be terribly concerned about such network parameters as the node degree and diameter.

Figure 15 shows the effect of changing the distance threshold for choosing new parents. Here, we clearly see that introducing hysteresis into the parent changing function reduces the number of parent changes. We note that fewer changes here results in less efficient hierarchies, because children are further away from their parents. Notice that the threshold has no effect when the parent crashes. This is of course because when a parent crashes, the child has no choice but to find a new parent.

Figure 16 shows the effect of the average number of children on hierarchy changes. In theory, it should make no difference what the number of children is, because on the average each node will

Figure 13  
Impact of Randomness on Changes



still have the same number of children. However, when the number of children approaches, and in the case of 16 children, surpasses  $\sqrt{N}$ , then some globals (those at level 1 and especially those at level 2) will have more children, and those at level 0 will have none. However, those with more children will tend to pull the average higher more than those with 0 children will pull it lower.

Figure 17 shows the effect of multiple topology changes on the hierarchy. We don't show the data for roots, because there weren't enough roots to remove four or eight of them. We see here that independent topology changes independently add to the number of hierarchy changes. In other words, the effect of topology changes is proportional to the number of topology changes.

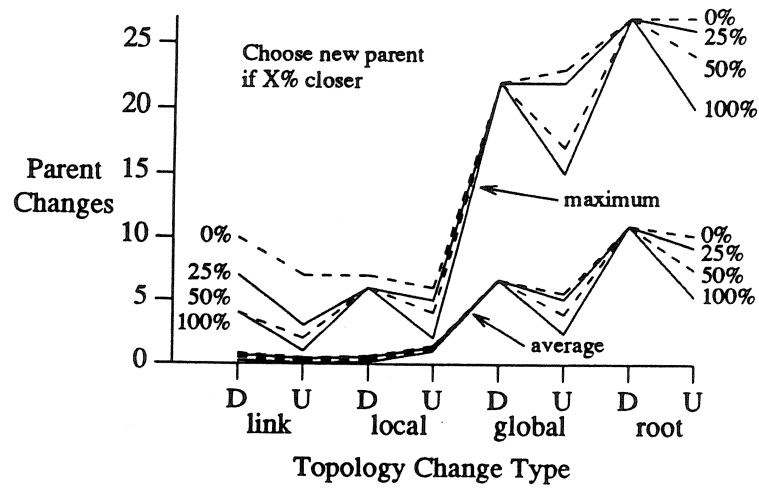
In Figure 18, we compare our experimental results with the results from our equations. Recall that our analytical results are supposed to be somewhat higher than actual results. This is generally true for Figure 18, although we see that for the root crashing experiment, our experimental results are slightly higher than predicted. The equations do not quite adequately account for the variation in the number of offspring each global has. Since of course the roots will be on the high end of that variation, we see that their crashes result in more address changes than we predicted. Still, the predicted and actual average results are quite close.

### 6.3 Dynamic Hierarchy Algorithm Simulations

This Section is split into two parts, simulation description and simulation results.



**Figure 14**  
**Effect of New Parent Distance Threshold on Hierarchy Changes**



**Figure 15**  
**Effect of Number of Nodes on Hierarchy Changes**

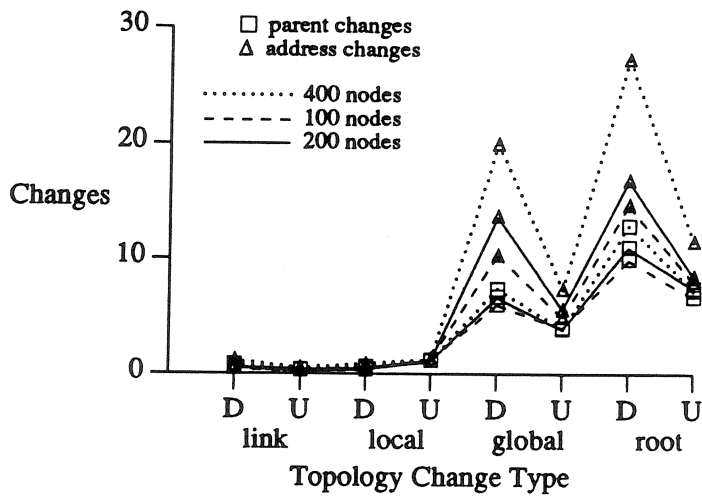
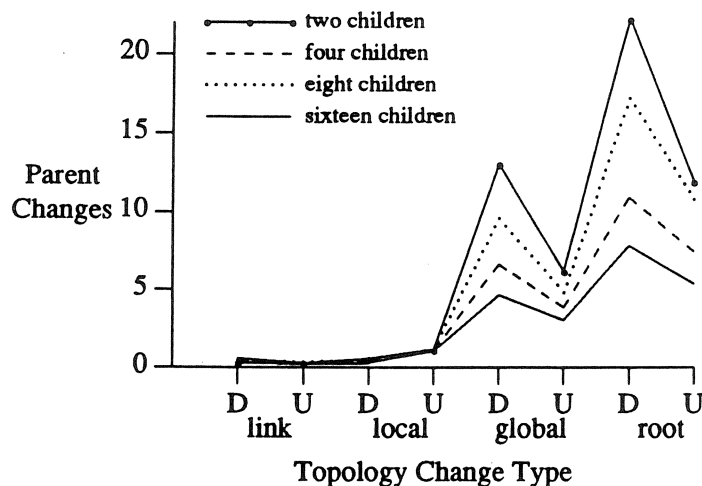


Figure 16  
Effect of Number of Children on Hierarchy Changes



### 6.3.1 Hierarchy Algorithm Simulation Description

The simulations 1) measure the performance of the Hierarchy Algorithm for some environment, and 2) determine the impact of various parameters on Hierarchy Algorithm performance.

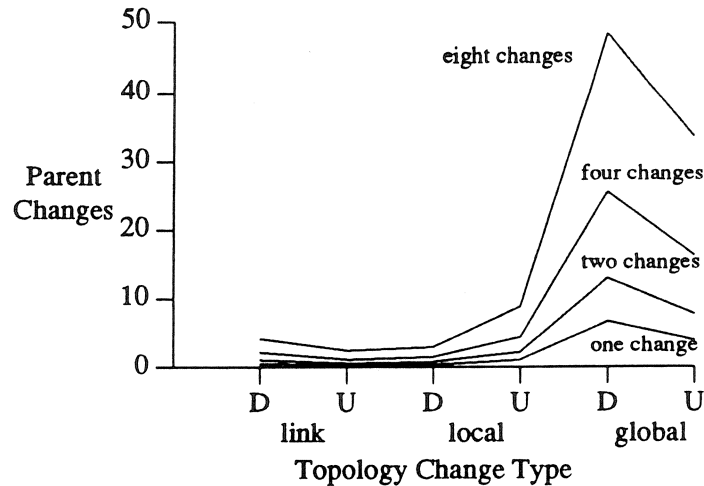
**6.3.1.1 Simulation Parameters.** Table 7 describes the parameters that are varied in the simulations.

We simulated four different networks. The network n50d7e3 is a control network. It has 50 nodes ( $n$ ), a diameter of 7 ( $d$ ), and an average node degree of 3 ( $c$ ). Each of the other three networks varies from n50d7c3 in one aspect. n25d7c3 has 25 nodes instead of 50, n50d14c3 has a diameter of 14 instead of 7, and n50d7c6 has an average node degree of 6 instead of 3. By using these four networks, we hope to learn the impact of each network parameter on the Hierarchy Algorithm.

Over most of the simulation parameters, we simulated both with and without the Hierarchy Algorithm. By comparing these two groups, we can see the additional impact of using the Hierarchy Algorithm on top of the Routing Algorithm over just using the Routing Algorithm alone. We produce the effect of running without the Hierarchy Algorithm by simply making all Landmarks global.

The average time to forward a routing update ( $T_{update}$ ) has a strong impact on the performance of the Hierarchy Algorithm. We simulate with  $T_{update} = 1$ ,  $T_{update} = 2$ , and  $T_{update} = 4seconds$ .

Figure 17  
 Effect of Multiple Topology Changes on Hierarchy Changes



The average number of children per Landmark ( $C$ ) impacts the number of levels in the Hierarchy, and impacts the number of times a child is rejected. We use values of  $C = 2$ ,  $C = 4$ , and  $C = 8$ .

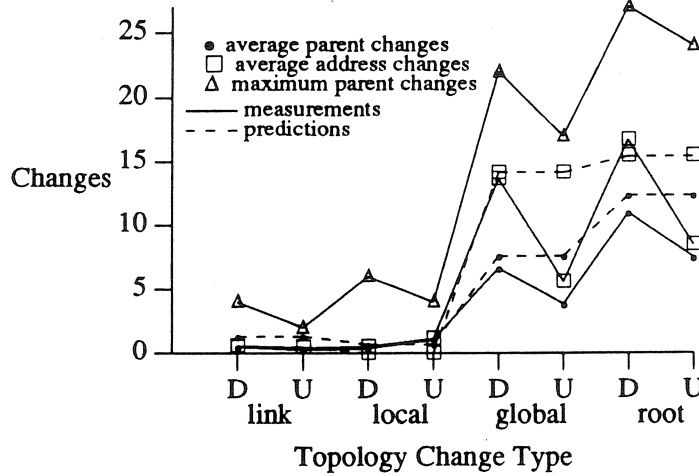
Finally, we produce different dynamic changes to the network topology during simulation (what we call scenarios). Since the Hierarchy Algorithm is meant to run in a dynamic network environment, these scenarios are what really measures Hierarchy Algorithm performance.

The baseline scenario is to simply boot the network and let it reach steady state, but not make any changes after that. The other scenarios all produce a controlled change after steady state has been reached. This way we clearly see the impact of each type of change.

To test the impact of changes in different types of Landmarks, we both crash and bring up one local (level 0), one global, and one root Landmark. To test the impact of changing different amounts of the network, we both crash and bring up one Landmark, two Landmarks, and four Landmarks. Finally, we both partition the network (roughly in half) and join two partitions.

Notice that in all of these experiments, we assume a perfect binding algorithm. In other words, all nodes know of the current addresses of all other nodes immediately. While on one hand this gives us artificially good results, it does serve to isolate the effects of the Binding Algorithm from those of the Hierarchy Algorithm

Figure 18  
Comparison of Experimental and Analytical Results



**6.3.1.2 Parameter Combinations.** Table 8 shows the various parameter combinations used in the simulations. We did not simulate all possible permutations of all parameters. Instead, we picked a reasonable baseline set of parameters, and then varied one parameter at a time from that baseline. The potential disadvantage of this is that we don't get to see the impact of combinations of parameters.

The baseline parameters are network n50d7c3, both with and without the Hierarchy Algorithm, average routing update time ( $T_{update}$ ) of 1 second, average 4 children per Landmark ( $C$ ), and the network boot, crash 1 Landmark, and bring up 1 Landmark scenarios.

The single-parameter deviations from this baseline are as shown in Table 8. There are three exceptions to the pattern of deviations. First, we don't bother to run without the Hierarchy Algorithm when we change the number of children per Landmark, because this parameter has no effect. Second, we don't run without the Hierarchy Algorithm when we change local and root Landmarks, again because these don't apply to the no Hierarchy case (where every node is a global Landmark). For the network partition and partition join scenarios, we used network n50d14c3 instead of network n50d7c3 because fewer links were needed to create the partition or join. Most network partitions or partition joins will occur because of a single link going down (the last one) or coming up (the first one).

Ten simulations were run for each parameter combination. We call each group of ten simulations an experiment. We present the median, and sometimes the maximum, value.

Table 7  
Modifiable Hierarchy Algorithm Simulation Parameters

PARAMETER	VALUES	COMMENTS
Network	n25d7e3 n50d7e3 n50d14e3 n50d7e6	(nodes/diameter/node degree) Different networks tell how the number of nodes, diameter, and the average node degree impact algorithm
Hierarchy	yes/no	no hierarchy is control experiment to evaluate performance of hierarchy
$T_{update}$	1,2,4	Average time to forward routing update
$C$	2,4,8	Average number of children per Landmark
Scenarios	Network Boot Crash 1 Local Landmark Crash 1 Global Landmark Crash 1 Root Landmark Bring Up 1 Local Landmark Bring Up 1 Global Landmark Bring Up 1 Root Landmark Partition Network Join 2 Partitions Crash 2 Landmarks Crash 4 Landmarks Bring Up 2 Landmarks Bring Up 4 Landmarks	No link or node changes The remaining scenarios occur after the network has reached steady state

In general, we saw a large variance within each group of simulations. As a result, our data has a larger margin of error than we would like. Nevertheless, it does give us a general idea of Hierarchy Algorithm performance.

**6.3.1.3 Results.** We are interested in the kind of results we cannot get from the static simulations: time to convergence and number of packets lost.

Figures 19 and 20 show the impact of the convergence speed of the lower level Routing Algorithm ( $T_{update}$ ) on the Hierarchy Algorithm. Figure 19 gives time-lines for several experiments. The main measure of convergence speed is the time of the last dropped packet (shown by "LP"). However, we also show the time of the last parent change ("pc"), and the last binding update ("bs").

In Figure 19, we show data for both with and without the Landmark Hierarchy. We also show data for two scenarios, the network boot, and crashing one node (a global, in the case of the hierarchy experiment).

Table 8  
Hierarchy Algorithm Simulation Parameter Combinations

Network	Hierarchy	$T_{update}$	$C$	Scenario
n50d7c3	yes/no	1	4	Network Boot Crash 1 (global) Landmark Bring Up 1 (global) Landmark
The above are the six base experiments. Unless otherwise noted, all remaining experiments are single-parameter deviations from these.				
n25d7c3 n50d14c3 n50d7c6				
		2 4		
	(yes only) (yes only)		2 8	
(n50d14c3) (n50d14c3)	(yes only) (yes only) (yes only) (yes only)			Crash 1 Local Landmark Crash 1 Root Landmark Bring Up 1 Local Landmark Bring Up 1 Root Landmark Partition Network Join 2 Partitions Crash 2 Landmarks Crash 4 Landmarks Bring Up 2 Landmarks Bring Up 4 Landmarks

The first thing to note is that convergence time is much faster for the network with no hierarchy. This is to be expected, as the Landmark Hierarchy cannot even start until after the Routing Algorithm has finished its job.

The second thing to note is that convergence time is roughly proportional to the average update time ( $T_{update}$ ) of the Routing Algorithm. This is to be expected (see Table 6), as most of the timing is dependent on  $T_{update}$ .

The third thing to note is that in the crash scenarios, the last binding update sent is long after the last packet failure. This is because of the count-to-infinity problem with the distance-vector algorithm we used (the SURAN packet radio algorithm). Count-to-infinity manifests itself, from the point of view of a single node, as a destination disappearing, reappearing with a longer distance, disappearing again, reappearing with a still longer distance, and so on. Since the Binding Algorithm sees this as nodes entering and leaving the hierarchy, it responds each time the destination appears or disappears. Clearly, Landmark Routing should have a distance-vector

Figure 19  
**Convergence Time due to Routing Algorithm Update Speed**

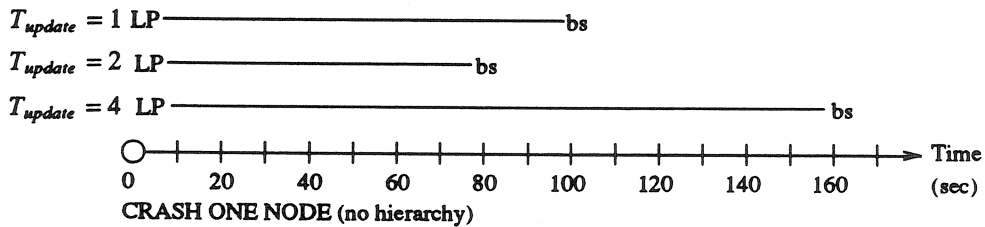
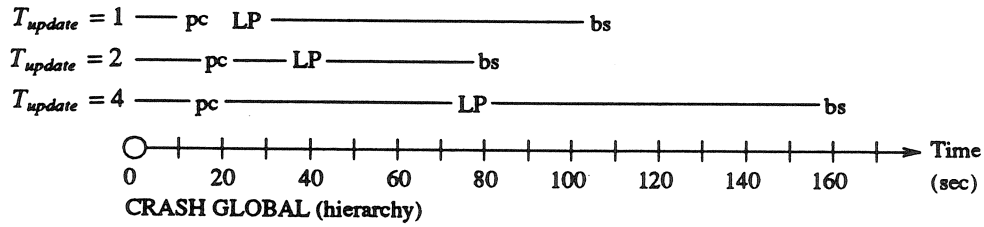
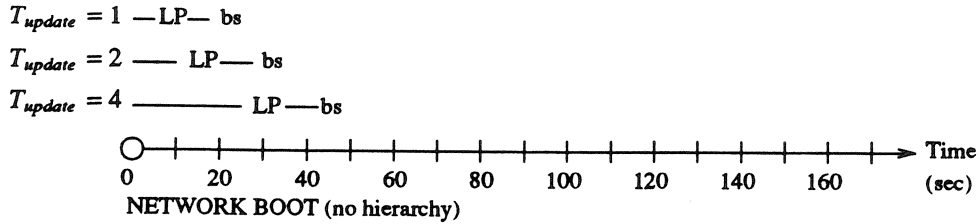
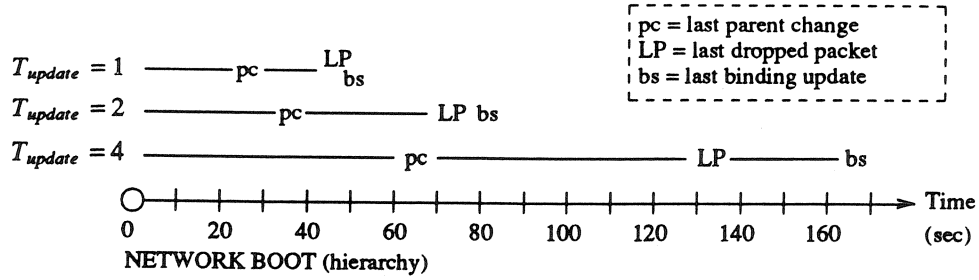
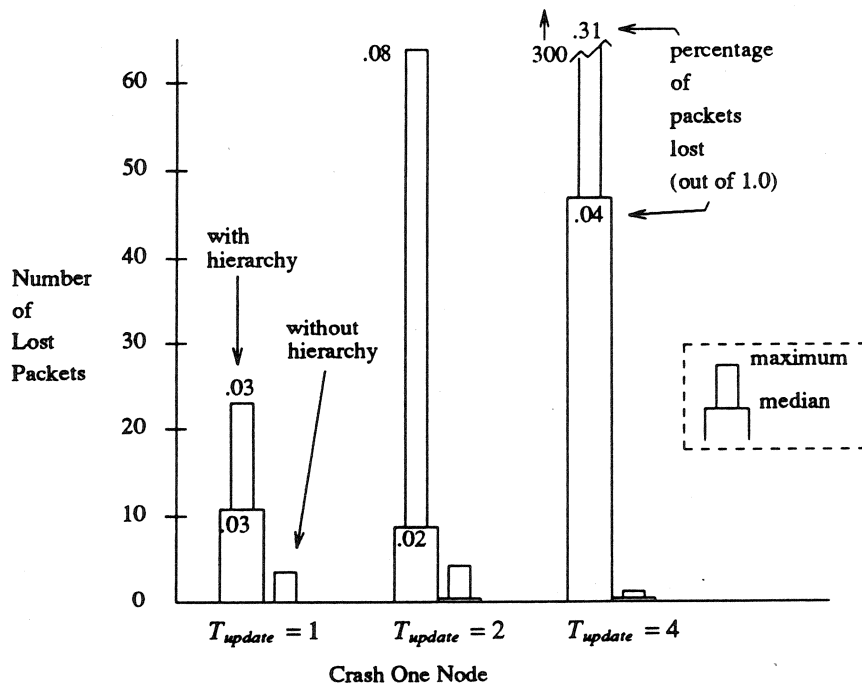


Figure 20  
 Convergence Time due to Routing Algorithm Update Speed



routing algorithm that does not exhibit count-to-infinity. It is not certain why the average last binding update is later for  $T_{update} = 1$  than  $T_{update} = 2$ . However, given that the duration of count-to-infinity varies widely from run to run, it could easily be due to random variation.

Finally, notice that the delay between the last parent change (pc) and the last lost packet (LP) is large, and is dependent on  $T_{update}$ . This is because of the length of time it takes to establish new addresses, which in turn is because new addresses are learned through the routing updates, one level at a time (each Landmark learns from its immediate parent). As discussed in Section 5, this could be speeded up if address changes were quickly flooded from the Landmark with the new address to all of its offspring.

In Figure 20, we show the number and percentage of packets lost after a node crash. Here, we show both the median and maximum values.

First, notice that in general a small percentage of packets were lost after the crash but before convergence. This figure is obviously highly dependent on the traffic pattern and on the specific failure. We used a random traffic pattern (all nodes send to any other node with equal probability).

Notice also that fewer packets were lost with  $T_{update} = 2$  than with  $T_{update} = 1$  (median value).



This can be attributed to the large random variation between experiments with the same parameters. In fact, the average value for  $T_{update} = 2$  is larger than that for  $T_{update} = 1$  (16.40 vs. 11.38).

Finally, notice that one of our simulations exhibited an unusually large percentage of lost packets (31%). In this case it was because the crashed global was the root and had an unusually large number of offspring. These large percentages are less likely to occur with large networks, because it is less likely that a single node will have a large percentage of network nodes as its offspring.

In Figure 21, we show the impact of the average number of children on convergence time and number of lost packets. While there is not a large difference between  $C = 2$ ,  $C = 4$ , and  $C = 8$ , it does seem in general that more children results in faster convergence and fewer lost packets.

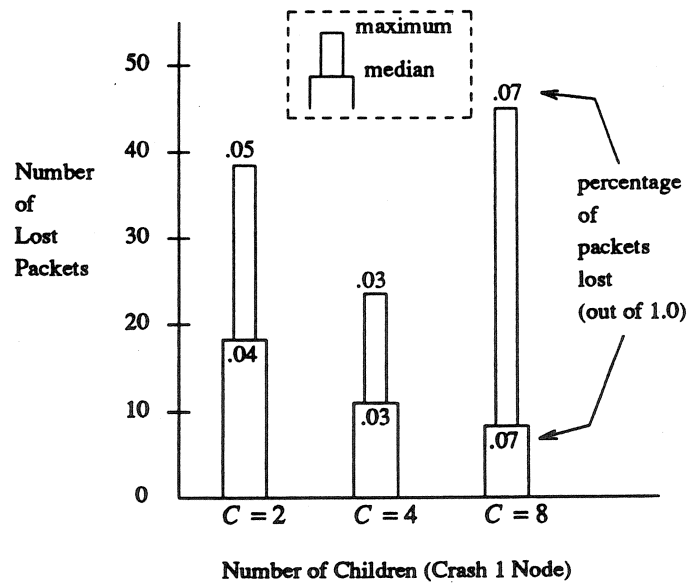
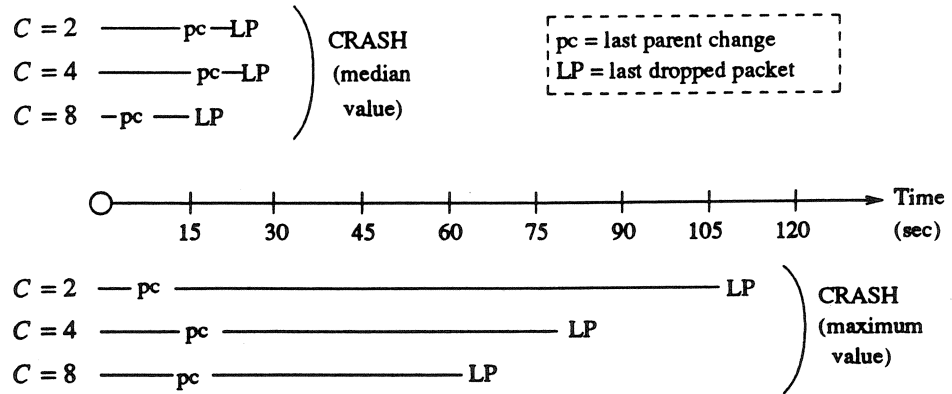
In Figure 22, we show the impact of number of nodes, diameter, and node degree, on convergence time and lost packets. First, notice that the 25 node network converged faster. This is probably due to fewer levels of hierarchy. Second, notice that the 25 node network lost a larger percentage of packets. This is due to the fact that any one global has a larger percentage of network nodes dependent on it for its address (20% on the average). Third, notice that the larger diameter network took longer to converge and lost more packets than the other 50 node networks. This is because, with larger diameter, it takes longer for the routing updates to propagate outwards.

Finally, Figures 23 through 25 show the impact of the type and magnitude of topology change on convergence time and number of lost packets.

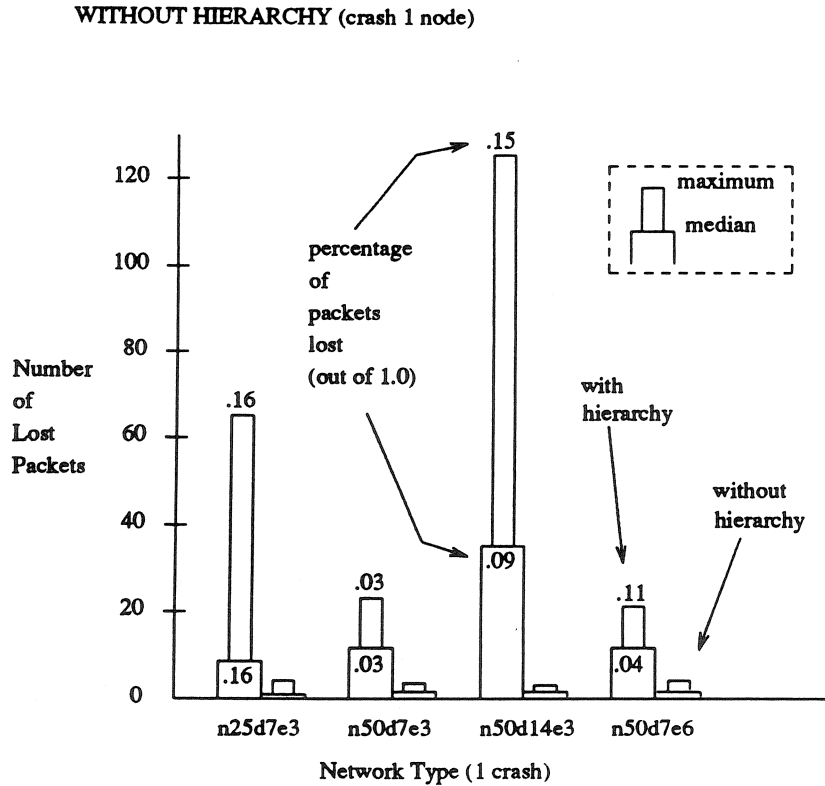
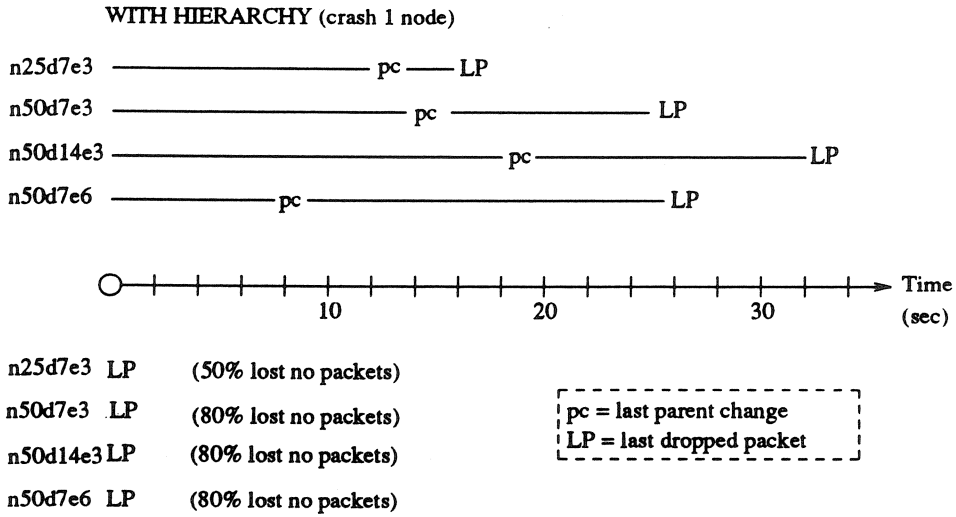
Figure 23 and the last three bars of Figure 24 clearly show the impact of crashing Landmarks at various levels. The higher the Landmark, the larger the impact. Our data shows significantly worse performance when the root is crashed vs. any global. Again, the difference in performance would be less for larger networks.

Figure 25 and the first four bars of Figure 24 show the impact of the number of failures. The data here does not show the clear trend seen for hierarchy level. This is because each failure acts more-or-less independently of each other, and thus convergence time is not affected much. We expect convergence time to take slightly longer with more changed nodes because convergence time will be the maximum of each change. It is surprising to see that the median number of lost packets is less for 4 nodes crashing than for 2 nodes crashing. However, the median 4 nodes crashing test converged more quickly than the median 2 nodes crashing test. Notice that the percentage of lost packets is slightly larger for 4 nodes crashing, as would be expected.

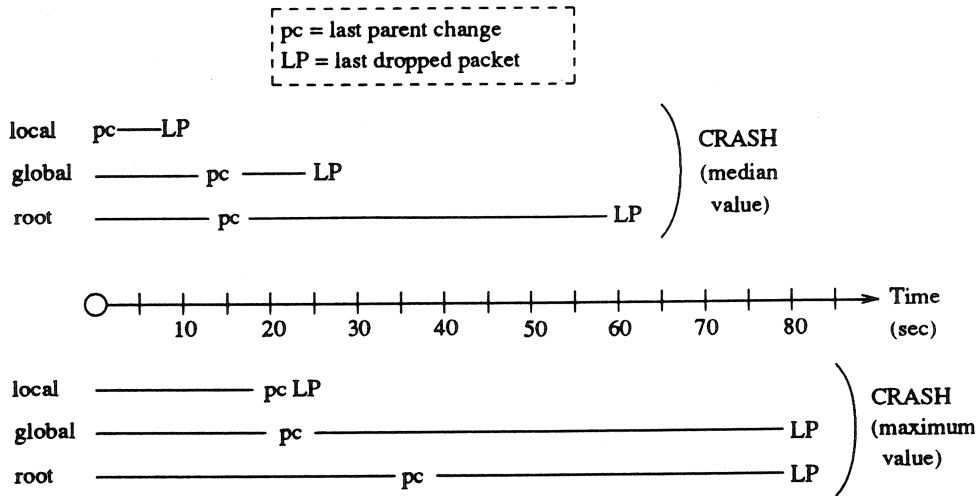
**Figure 21**  
**Convergence Time and Packets Lost by Number of Children**



**Figure 22**  
**Convergence Time and Packets Lost by Network Type**



**Figure 23**  
**Convergence Time by Landmark Level**



**Figure 24**  
**Number of Packets Lost by Landmark Level and Number of Failures**

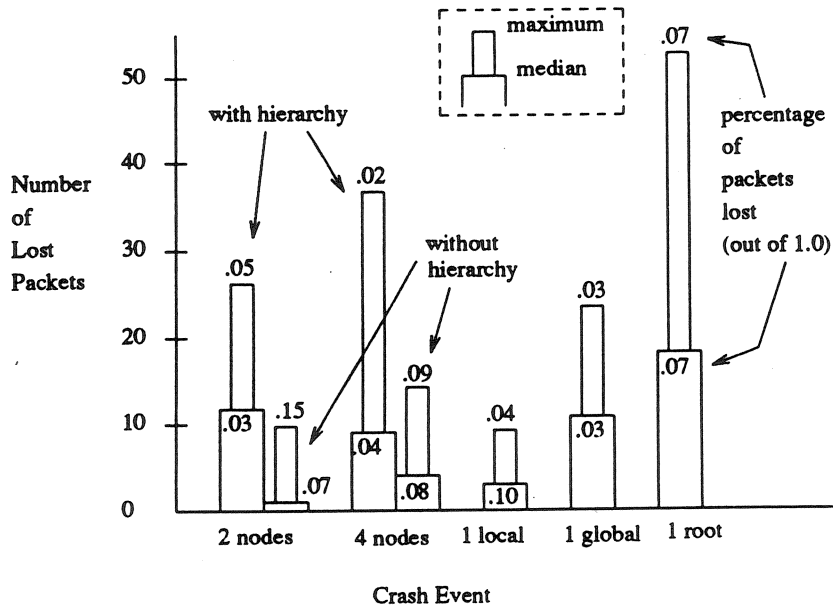
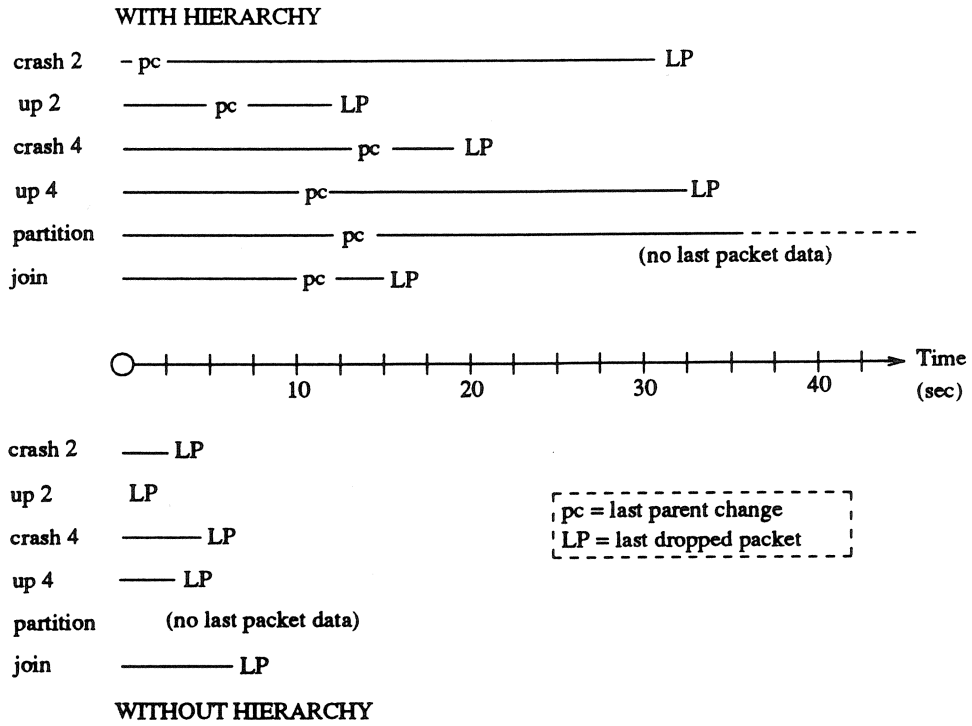


Figure 25  
 Convergence Time by Number of Failures



## 7 BINDING ALGORITHM DESCRIPTION

### 7.1 Overview

While the Hierarchy Algorithm changed markedly from the description in the previous work (Tsuchiya, 1987b), the Binding Algorithm changed only slightly. Therefore, we will only summarize that algorithm, highlighting those things that have changed.

The problem to be solved by the Binding Algorithm is that since the address of any node can change at any time, the source of a packet must discover the current address of the destination given the name (or ID) of the destination. Of course, the source cannot ask the destination its address, since it would need to already know the destination's address to do this. Therefore, it is necessary for there to be some third party—the name server—that can be found by both the source and the destination. The destination gives its name server its current address (update) and the source asks the name server for this address (query).

We want the Binding Algorithm to be both robust and efficient. Flooding (either updates or queries) is robust but not efficient. Creating a name server hierarchy creates potential bottlenecks and single points of failure, and doesn't eliminate flooding since the name servers must flood their current addresses, which of course can change as easily as any other node's.

In our Binding Algorithm, called Assured Destination Binding, we have, for robustness, every node (Landmark) act as a name server. Therefore, the routing table is at least a partial view of the set of potential name servers (partial because of the hierarchy). To determine the correct name server for any given named destination, we hash the name into the address space, resolve the resulting address to one of the entries in the routing table, and send either update or query to that address. If the routing table changes, then certain bindings must be re-resolved and new updates sent out. The resolution works by mapping the address derived from the hash function into the next highest real address (in the routing table).

This algorithm is efficient because updates and queries are not flooded. It is robust because the routing table, which is itself robust, is the basis for discovering the correct name server for any given destination.

Because the routing table is hierarchical, the address derived from the hash function cannot normally be fully resolved by the source of the update or query. This is because the resolution does not work properly unless each node that might do the resolution agrees on the set of addresses to be resolved to. Therefore, the derived address must be resolved one level of the hierarchy at a time, starting from the global level.

In other words, the source of the update or query 1) hashes the name, producing the derived address, 2) maps the derived address into the set of global Landmarks, and 3) sends the update or query towards the chosen global Landmark. When the update or query reaches one of the offspring of the chosen global Landmark, that node is able to resolve the update or query to one of the children of the global Landmark; the node then sends the update or query to that child. The first offspring of that child that receives the query or update further resolves it to one of its children, and

so on. Eventually, the update or query will reach the appropriate name server, regardless of the source of the update or query (see Figure 26).

Hashing the names into addresses evenly distributes the derived addresses among the address space. However, since addresses themselves may be clustered, it is also necessary to hash the entries of the routing table (the addresses) into what we call an Intermediate Hash Space (IHS) (see Figure 27). Names are then hashed into the IHS rather than directly into the address space, and resolved to the next highest entry in the IHS, which is mapped back into an entry in the routing table. There will be one IHS for each level of the hierarchy.

Routing table entries will be hashed into the IHS many times in order to provide a still better distribution of bindings. This is necessary because, since there are a small number of entries at any single hierarchical level, the pseudo-random variation in the hash function can cause an uneven distribution of IHS entries. Multiple hashes can be accomplished by appending a byte to the hash input (the address itself) and incrementing the value in that byte for each additional hash.

To increase robustness and efficiency, several name servers are provided for each binding. This is more efficient because 1) queries can go to the nearest name server (this is only more efficient if there are substantially more queries than updates), and 2) because the period for sending out updates can be longer since the period is based on the probability that all servers for any given binding have crashed. Multiple name servers can be chosen using the same technique as creating multiple IHS entries—append a byte to the name input to the hash function, and increment the byte for each additional server.

While the hash function results in an even distribution of bindings, it doesn't guarantee that all name servers will have a similar workload. This is because certain popular destinations will have more queries generated for them, and those queries will be directed towards a few name servers. To spread this load around, when a name server finds that it is receiving excessive queries for one of its destinations, it gives the binding to the neighbors it is receiving the bindings from. The neighbors will then begin to answer the bindings, and can themselves spread the binding to their neighbors if necessary. The more queries that are generated for a particular destination, the more adjacent servers that will receive the binding and answer the queries (see Figure 28).

Once a binding has been handed to a neighbor, it must be kept current for that neighbor as well. We call the original holder of the binding the primary name server, and the subsequent holders of the binding the adjacent name servers. When the primary name server receives the periodic update, or a change for an existing binding, it must tell its adjacent name servers. If an adjacent name server finds that it is not receiving many queries for a binding it is holding, it can choose to no longer hold the binding, and must tell the primary (or up-tree adjacent) name server.

When a node gets a new address, it sends out new updates for its attached hosts. It also determines which of the bindings it is holding no longer resolve to it and informs the sources of those bindings that they need to send out new updates. This way, all address changes result in immediate updating of the bindings. When a node crashes, it obviously cannot inform the source that new bindings must go out, and so bindings are sent out periodically to ensure that bindings are up to date in the event of crashes.

**Figure 26**  
**Hierarchical Resolution of Addresses Derived Through Hash Function**

---

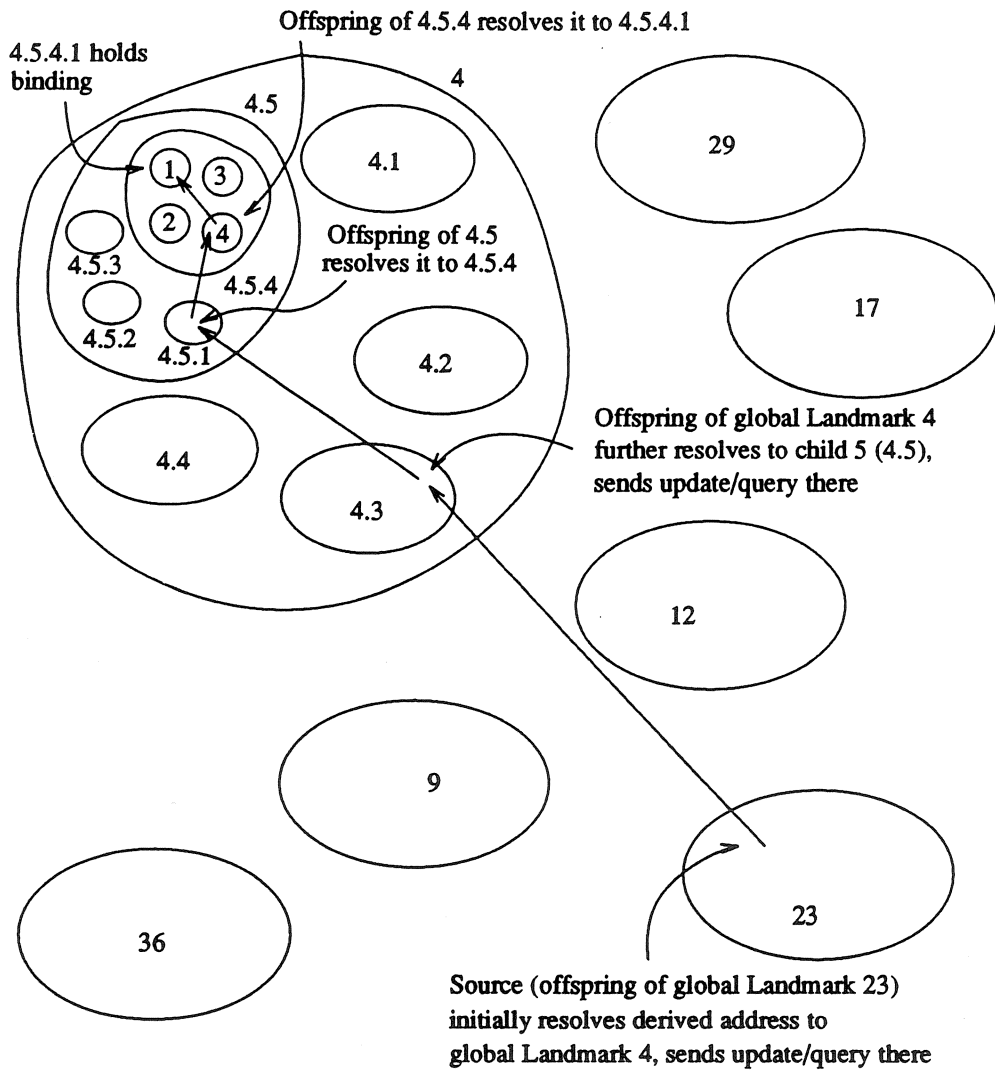
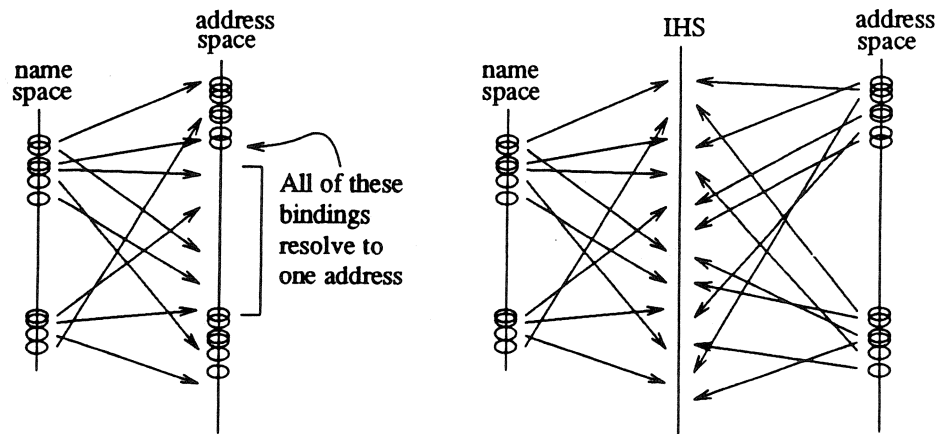




Figure 27

## Intermediate Hash Space Allows for Even Distribution of Bindings



Both names and addresses likely to be clustered, but if names hashed directly into address space, addresses at bottom of clusters will be overloaded.

By hashing both names and addresses into an Intermediate Hash Space, we get a relatively even distribution of bindings

## 7.2 Detailed Description

The name server can be partitioned into three functions: the host server, the resolution server, and the binding server (see Figure 29). The host server maintains a list of the node's attached hosts, and sends out updates for the hosts. The binding server receives updates and answers queries. The resolution server resolves updates and queries to the appropriate binding servers.

Table 9 shows the messages sent and received by the binding server, and the contents of those messages.

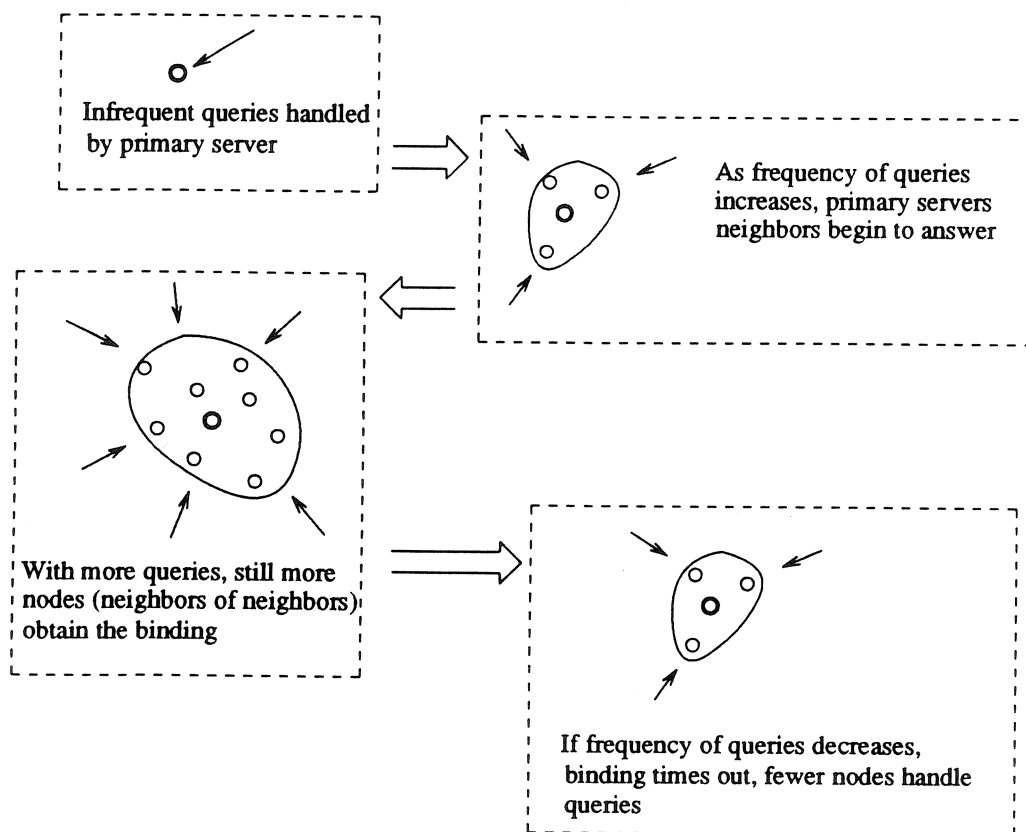
### 7.2.1 Host Server Description

The host server acts as an agent for the node's attached hosts. It registers and deregisters attached hosts, and sends updates and queries to the resolution server on behalf of the hosts.

Table 10 shows the data held by the host server. Figure 30 shows the host server algorithm, which is also described below.

When a host is added, its ID and time out period (`host_life_time`) is added to the `hosts[ ]` array. This timer is used to determine if the host has crashed or left. This timer is reset any time a packet is received from the host (this not shown in Figure 30). Then the host server sends to the resolution server  $K_B$  primary updates (`pri_update_msg`) binding the host ID to its `lm_addr` (that of the node itself). Each update has a different key (see Table 10). It is the key concatenated with the `host_id` that forms the input to the hash function executed by the resolution server. The host server

Figure 28  
How to Handle Frequent Queries



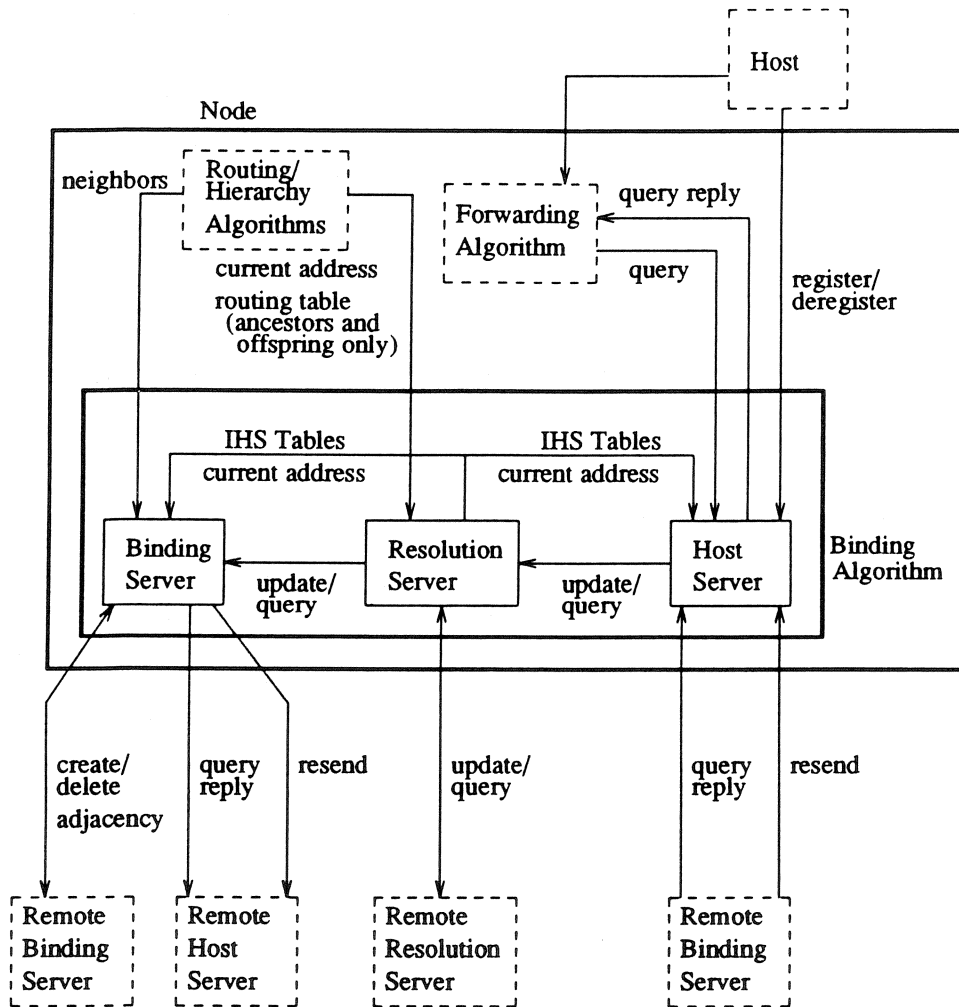
fills in all fields of the `pri_update_msg` except the `dest_lm_addr`. That field is filled in by the various resolution servers that handle the `pri_update_msg` on its way to the destination binding server. The host server also sets the  $K_B$  `bind_life_time` timers, one for each of the  $K_B$  updates. These times should be set with some jitter to prevent updates from synchronizing.

When the `bind_life_time` timer expires, the host server sends to the resolution server a `primary_update_msg` for the concerned host/key pair.

When the `host_life_time` timer expires, the concerned host is removed from the `hosts[ ]` array.

When the node gets a new Landmark Address (`lm_addr`), it must send out new updates for all of its hosts. Perhaps the best order to send them out in is to do all of the hosts for one key, then all of the hosts for another key, and so on. The reason for this is that at least one correct update gets out for all hosts as quickly as possible.

Figure 29  
Binding Algorithm Architecture



In a bandwidth limited environment it may be necessary to spread these updates out over time to avoid congestion. At the same time, we do not want to prevent communication while updates are going out. Therefore, when a Landmark gets a new address, it may be possible to send an update to the previous address, which will get resolved to some real address. That update would say "anything sent to the previous address should be forwarded to the current address". For this to work, the Forwarding Algorithm would need to operate in such a way that it did resolution for all packets, so that packets that are temporarily misaddressed would go to the Landmark closest to the previous address and be forwarded on.

*Table 9*  
**Messages Sent and Received by the Binding Server**

Messages To/From Other Binding Algorithms		
MESSAGES	INFORMATION	COMMENTS
pri_update_msg	key dest_lm_addr host_id host_lm_addr host_life_time source_host_server_id source_host_server_lm_addr	key associated with host composed in transit  life time of host in reply to send resend
adj_update_msg	key host_id host_lm_addr host_life_time	key associated with host  life time of host in reply
query_msg	key dest_lm_addr host_id source_host_server_id source_host_server_lm_addr	key associated with host composed in transit  to send reply
adjacency_delete_msg	host_id key	key associated with host
reply_msg	type host_id host_lm_addr key keys_used host_life_time	positive/negative  key associated with host which keys used until now life time of host if positive
resend_msg	key host_id	key associated with host

Table 10  
Host Server Data

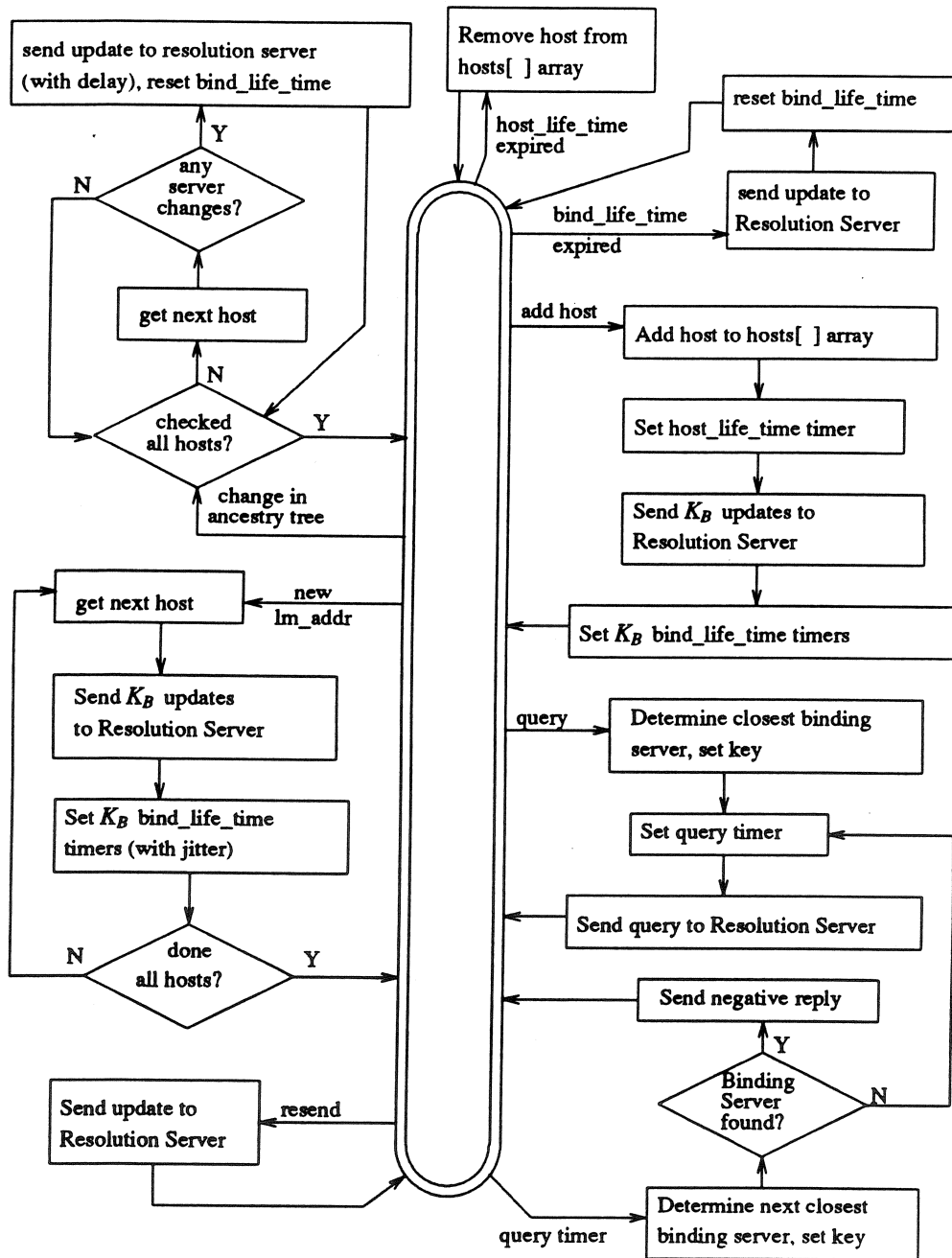
Host Server Data	
Object	Description
Sub Object	
lm_addr	own Landmark Address
ID	own ID
hosts[ ]	array of attached hosts
host_id	ID of host
host_life_time	host configuration timeout
IHS Tables[ ][ ]	index by level, entry (shared with resolution and binding servers)
bind_life_time	period between host updates
queries[ ]	list of outstanding queries
keys[ ]	list of used keys
host_id	destination host ID

Without this function, it is not necessary for the Forwarding Algorithm to do resolution because the only other packets that need resolution, updates and queries, can be locally addressed to the neighbor that is the next hop towards resolution. In our simulations, we did not do the previous address update function.

The host server must respond to changes in its ancestry tree. The ancestry tree consists of all of a Landmark's children, its ancestors (up to the global level), and its ancestor's siblings (this includes all globals). The reason for this is that a host server can tell in some cases when it is necessary to resend an update without being told by the binding servers holding its bindings. When a change occurs in the ancestry tree, the host server checks each host binding and determines if the change causes any of its entries to resolve to another server. If it does, then the host server, after a small delay, sends a fresh update to the resolution server, which will resolve it to the new binding server.

The small delay is necessary because, when a host server learns of a change in the ancestry tree, it is because some kind of topological change has occurred, and the Routing and Hierarchy

Figure 30  
Host Server Algorithm



Algorithms are converging. During convergence, different nodes will have different views of the world (in particular, different IHS tables), and as a result will resolve updates and queries differently. By delaying before sending out an update, most of these resolution differences can be presented (at the expense of increased delay in convergence of the Binding Algorithm). If even after delay there are still inconsistencies in the IHS tables, this will be discovered by the resolution server during the resolution process.

For those cases where the host server cannot determine that one of its host's bindings has a new binding server, the old binding server will know, and will send the host server a resend. Upon reception of the resend, the host server resends the update, which will be resolved to the new server.

Finally, the host server must send queries when the Forwarding Algorithm does not know the `lm_addr` for a given destination. When the host server receives a query, it first determines which of the  $K_B$  binding servers is closest. (This is clearly an optimization that favors bandwidth over processing. In environments where processing is the scarce resource, any of the  $K_B$  binding servers can be picked.) The host server fills in the key field of the query message with the key for the closest binding server and sends the query to the resolution server for delivery. The host server also sets the query timer in case the query is not answered. If the query is answered, the host server forwards the answer on to the Forwarding Algorithm (not shown in Figure 30). If the query is not answered, the timer will go off, and the host server will choose the next closest binding server if one exists, or return a negative reply to the Forwarding Algorithm if one does not.

### 7.2.2 Resolution Server Description

The resolution server receives updates and queries, determines where they should go, and forwards them on if necessary. To do this, the resolution server needs the IHS tables. It derives the IHS tables from the `ancestry_tree`, which it receives from the Hierarchy Algorithm. Table 11 shows the data used by the resolution server. Figure 31 shows the resolution server algorithm, which is also described below.

When the resolution server receives a new `ancestry_tree` entry from the Hierarchy Algorithm, it hashes the `lm_addr` label  $K_H$  times, changing the appended one-byte key each time, and adds the  $K_H$  entries to the IHS table at the appropriate level. It then sends the new IHS table entries to the host and binding servers so they can determine if they need to resend updates. The multiple hashes results in a relatively even distribution of bindings among the entries in the IHS table. Likewise, the resolution server removes entries from the IHS tables when necessary.

When the resolution server receives an update or query, 0 or more `dest_lm_addr` fields will already be filled in. The resolution server determines if it can fill in more fields. To see how, see Figure 32. Here, an update binding host V to address 2433.4.3.1 must be sent out. When the resolution server of Node A (2433.4.3.1) receives the binding from the host server, the `dest_lm_addr` field is blank. Therefore, Node A starts to resolve `dest_lm_addr` at the global level. Node A hashes V, resulting in some index into the global IHS, which in this example resolves to global label 801. Node A fills in the global field of `dest_lm_addr`, and sends the update to the next resolution server on the path towards global Landmark 801. The rest of Figure 32 is self-explanatory, and demonstrates most of the various paths in the flow chart of Figure 31.

Table 11  
Resolution Server Data

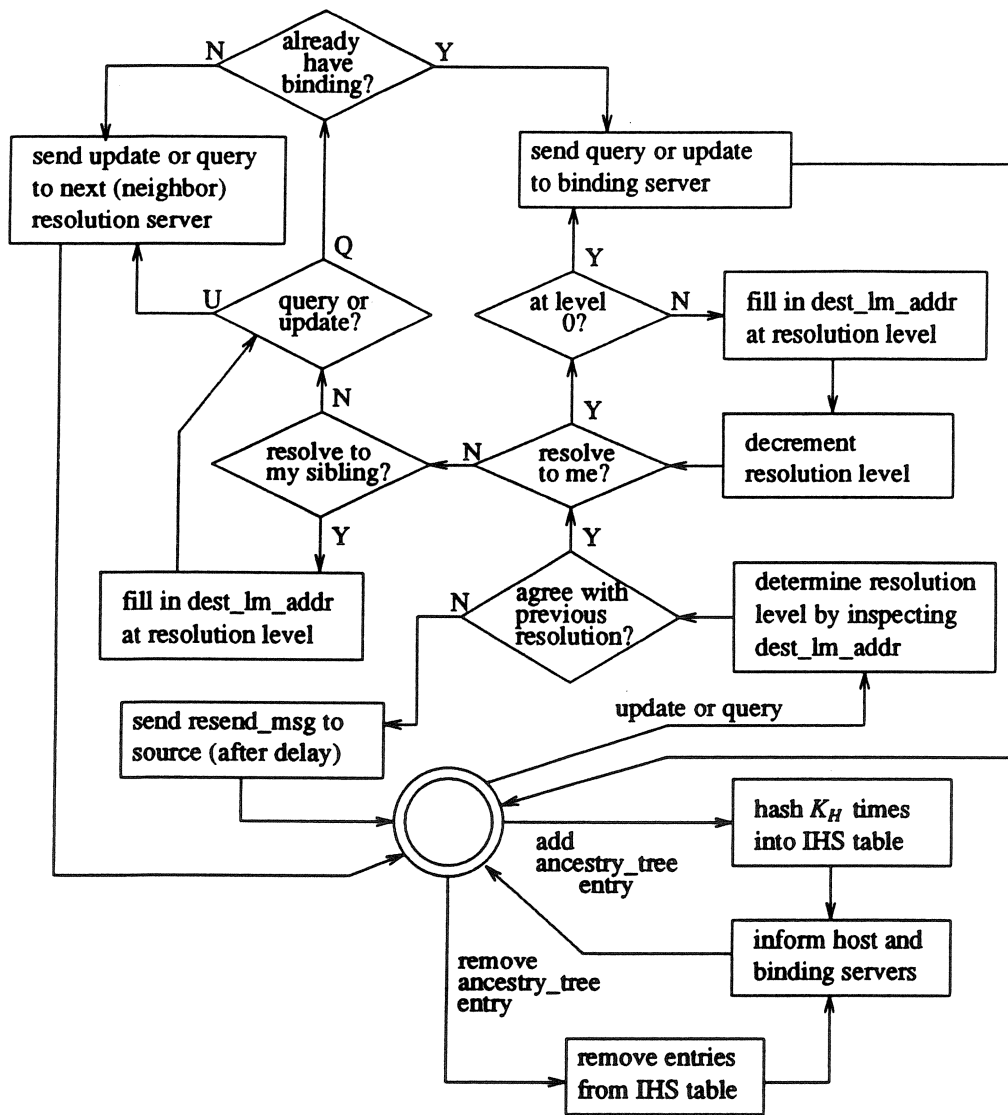
Resolution Server Data	
Object	Description
Sub Object	
lm_addr	own Landmark Address
ID	own ID
ancestry_tree[ ][ ]	indexed by level, entry children, ancestors, and ancestor's siblings
lm_addr label	
level	
IHS Tables[ ][ ]	index by level, entry (shared with host and binding servers)
lm_addr label	
key	tells which of $K_H$ hashes

Of course, during convergence of the Routing and Hierarchy Algorithms, neighboring resolution servers can have different IHSs (with respect to common ancestor levels). In these cases, two resolution servers can resolve an update or query differently. It is important, especially with updates, to discover these inconsistencies and correct them. Therefore, when a resolution server receives an update or query, it checks the resolution made by the previous resolution server to make sure they get the same result. If they do not, then, after a small delay, a resend is sent back to the source of the update or query. The delay is required to let the Routing and Hierarchy Algorithms converge. Upon receiving this resend, the source host server sends the update or query again.

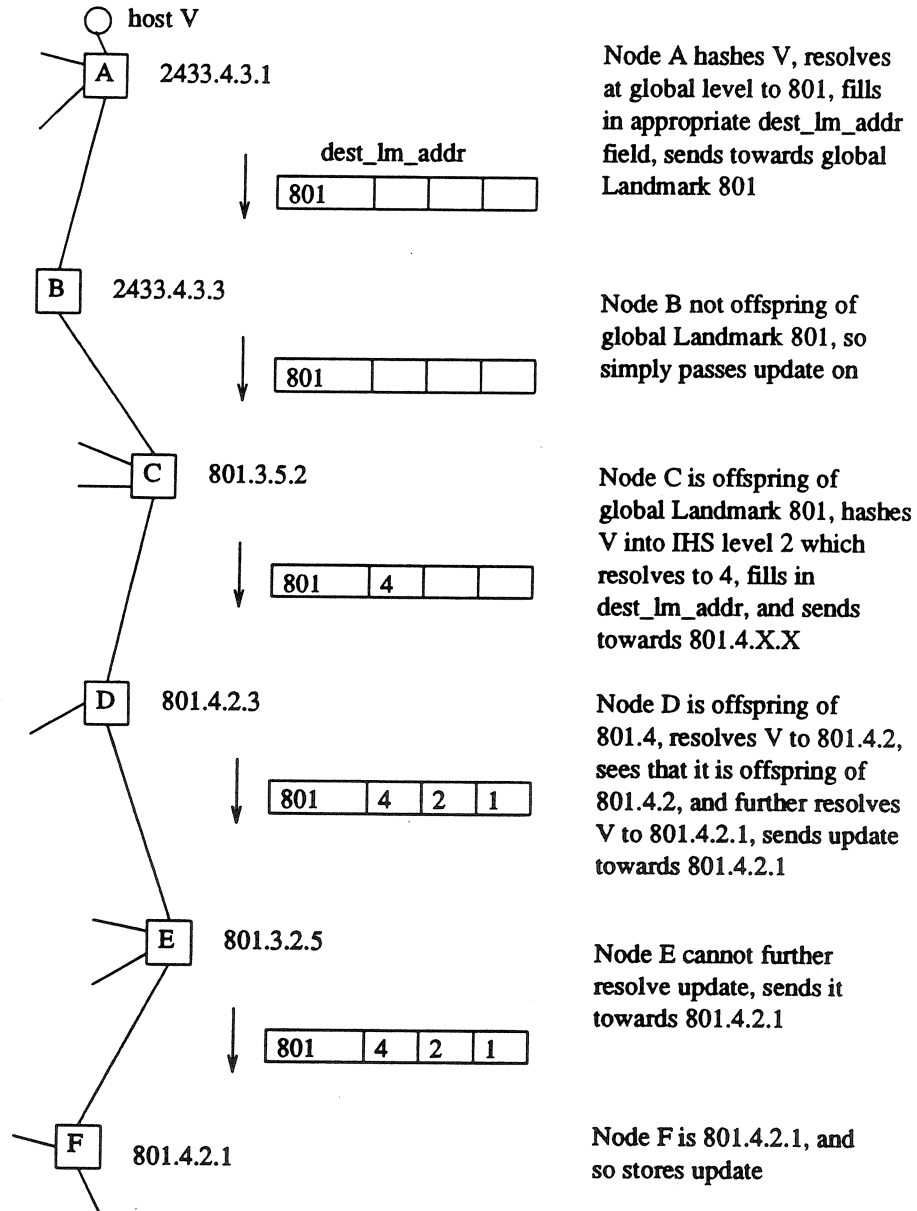
There is a difference between the handling of an update and the handling of a query. If it is a query, then the resolution server checks (through the binding server) whether or not that binding is already being held (as an adjacency). If it is, then the query is sent to the binding server to be answered. If not, then the resolution server puts its own ID and Landmark address in the `adj_server_id` and `adj_server_lm_addr` fields of the query packet. This allows the binding server that answers the query to send an adjacency update back to the neighbor binding server. Otherwise, updates and queries are handled identically by the resolution server.



Figure 31  
Resolution Server Algorithm



**Figure 32**  
**Update or Query Resolution Example**



### 7.2.3 Binding Server Description

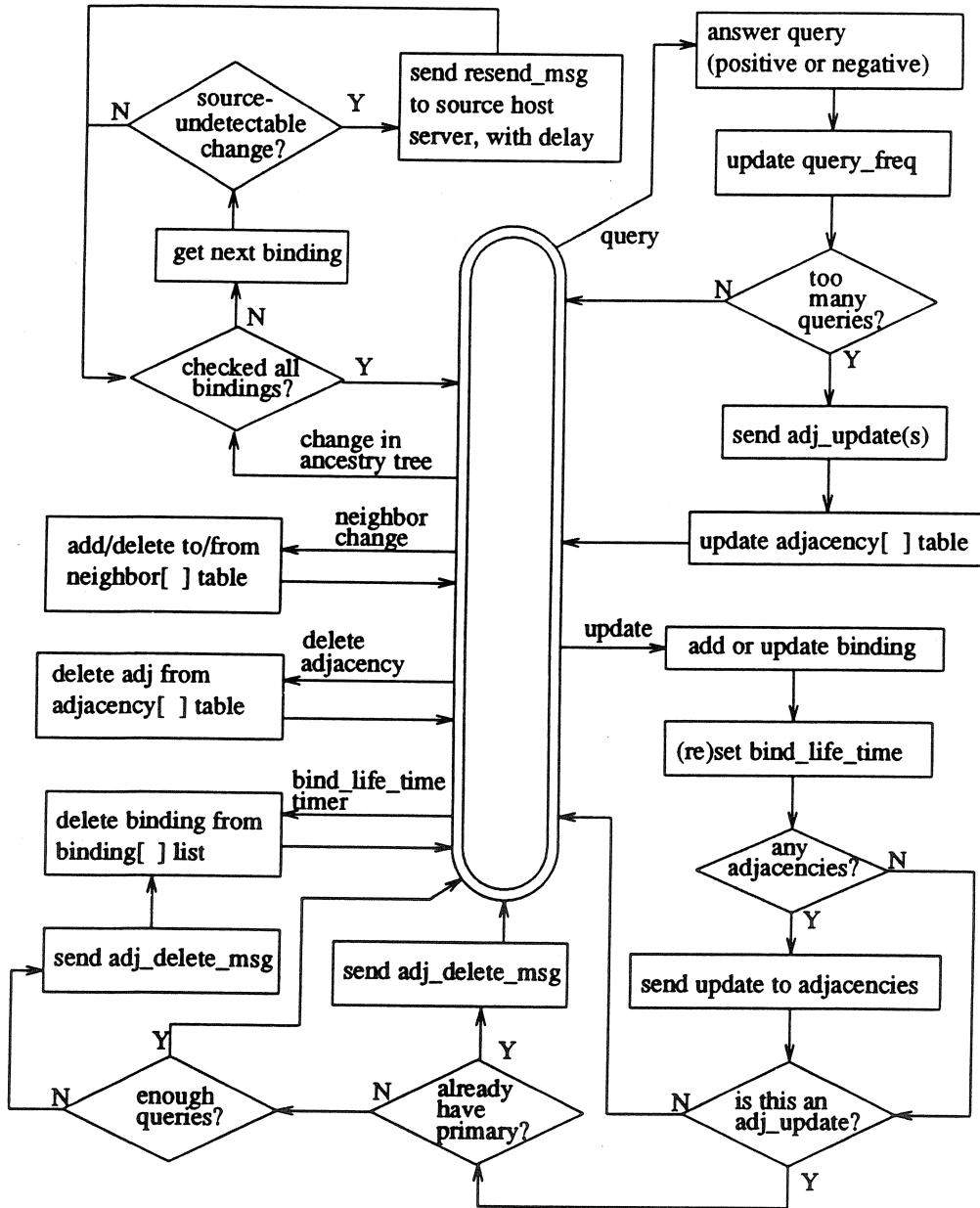
The binding server holds the bindings and answers queries. Table 12 shows the data used by the binding server, and Figure 33 shows the binding server algorithm which is also described below.

*Table 12*  
**Binding Server Data**

Binding Server Data	
Object	Description
Sub Object	
lm_addr	own Landmark Address
ID	own ID
IHS Tables[ ][ ]	index by level, entry (shared with resolution and host servers)
bindings[ ]	list of bindings index by host ID/key pair
ID	id of host in binding
lm_addr	current address of host
bind_life_time	expiration time
query_freq[ ]	how many queries received (over time per neighbor)
primary	uptree binding server null if I am original binding server
adjacency[ ]	list of adjacency binding servers
neighbors[ ]	list of neighbor binding servers index by id

When the binding server receives an update, it either adds it to its bindings[ ] list and sets the bind\_life\_time timer, or it updates an existing entry in bindings[ ] and resets the bind\_life\_time timer. This timer is for the purposes of deleting the entry if periodic updates are not received. If

Figure 33  
Binding Server Algorithm



adjacencies have been set up for this particular binding, then the update is forwarded on to them. If the update received is itself an adjacency update, then a check is made to see if this binding has already been sent by a different primary binding server. If it has, then an `adj_delete_msg` is sent back to the binding server that sent this update. If not, then `query_freq` is checked to see if enough queries have been received since the last update to warrant keeping the binding. If not, an `adj_delete_msg` is sent to the primary binding server, and the binding is deleted from the `bindings[ ]` table.

When the binding server receives a query, it looks in its `bindings[ ]` table to find the answer. If the answer exists, it is sent back as a positive reply to the host server that originated the query. If not, then a negative reply is sent back. Note that the query came to the binding server by way of one or more resolution servers, but is returned directly to the host server, since the host server's address is known. Next, the binding server updates `query_freq`. This parameter is used to determine if an adjacency update must be sent out. If `query_freq` has become too high, then an adjacency update is sent to the neighbor binding server over which the queries have been received, and the neighbor is added to the `adjacency[ ]` table.

When the binding server receives a change in the ancestry tree, it acts similarly to what the host server does when it receives a change. That is, it looks at its bindings to see if any no longer resolve to it and therefore need to be resent (by the host server), and deleted. This would be the case if either a new entry in the IHS tables captured some of the bindings that had previously resolved to it, or if the binding server got a new address, causing its bindings to resolve elsewhere. In either case, it may be possible for the host servers that originated the bindings to know themselves that their bindings no longer resolve to the binding server. The binding server can detect this (the same way that the host server can detect it), and need not send out a resend message. Otherwise, the binding server will send a resend message, after a small delay to let the Routing and Hierarchy Algorithms converge, to the host server, letting it know that it must send out a new update.

The remaining three events are straightforward. If a neighbor change message is received, then the binding server either adds or removes the neighbor from the `neighbors[ ]` table. If an `adjacency_delete_msg` is received, then the adjacency is deleted from the `adjacency[ ]` table. If the `bind_life_time` timer goes off, then the binding is removed from the `bindings[ ]` table.

## 8 BINDING ALGORITHM PERFORMANCE

This Section has three parts: 1) An analysis of the binding algorithm, 2) the results of static simulations of the binding algorithm, and 3) the results of dynamic simulations of the binding algorithm.

### 8.1 Binding Algorithm Analysis

In this analysis, we are interested in memory usage, number of packets generated, and convergence speed. However, there is really nothing to analyze with respect to convergence speed, because the binding algorithm will have converged shortly after the Routing Algorithm has converged. The extra delay comes from 1) short wait periods to insure that the Routing Algorithm has in fact converged, and 2) the time it takes to send out bindings.

#### 8.1.1 Memory Usage

We are interested in the memory used for each of the three parts of the binding algorithm: the host server, the binding server, and the resolution server.

On the average, each host server will have  $\frac{H}{N}$  entries, where  $H$  is the total number of hosts, and  $N$  is the number of nodes. Of course, this number varies from node to node depending on how many hosts configure with each node.

Binding servers carry entries both in their capacity as primary and alternate binding servers. Of course, the number of alternate binding entries is dependent on the number of popular destinations. We assume for this analysis simply that some fraction  $K_A$  of bindings will be alternates.  $K_A = 0.1$  seems to be a likely number.

For every host entry there are  $K_B$  bindings generated, and so there are  $K_B \frac{H}{N}$  primary binding entries per binding server, and  $K_B(1+K_A) \frac{H}{N}$  binding entries per binding server. Since the bindings are randomly distributed among all binding servers, we are interested in knowing the maximum number of bindings any binding server can be expected to hold.

To determine this, we ran an experiment where  $HN$  bindings were randomly distributed over  $N$  binding servers using the UNIX rand function. We experimented with values of  $N = 10$ ,  $N = 100$ ,  $N = 1000$ , and  $N = 10000$ , and  $H/N = 1$ ,  $H/N = 10$ , and  $H/N = 100$ . For each combination of  $H$  and  $N$ , we ran 100 trials, and recorded the maximum of each trial. The results are shown in Table 13.

First, we see that the maximum number of bindings that any one binding server holds is independent (or very nearly so) of the number of binding servers. Second, we see that the ratio of the maximum number of bindings over the average number of bindings shrinks as  $\frac{H}{N}$  increases. If there are 100 hosts per binding server, then a binding server only need have space for twice that, or 200, to handle virtually all of the bindings it will receive. In general,  $\max(200, 2K_B \frac{H}{N})$  entries is a

Table 13  
Distribution of HN Bindings Among N Nodes

	$H/N = 1$	$H/N = 10$	$H/N = 100$
$N = 10$	5	26	148
$N = 100$	7	23	140
$N = 1000$	7	26	143
$N = 10000$	9	29	153

sufficient number of binding entry spaces for any values of  $H$  and  $N$ . In the rare event that a binding server does not have space for a binding, then the backup servers can service queries.

The resolution server holds the IHS table. This table holds  $K_H R_H$  entries, where  $K_H$  is the number of times a routing table entry is hashed into the IHS, and  $R_H$  is the number of entries in the ancestry tree. However,  $R_H = C \log_C N$ , where  $C$  is the average number of children per Landmark, and  $\log_C N$  is of course the number of hierarchy levels. Therefore, the resolution server has on the average  $K_H C \log_C N$  entries. The maximum number of entries should never be even twice this, as each Landmark is limited to roughly  $2C$  children, and because the actual number of levels will not vary much from  $\log_C N$ .

All told, then, a node requires on the average

$$M_B = \frac{H}{N} + K_B(1+K_A)\frac{H}{N} + K_H C \log_C N$$

entries for the binding algorithm. Assuming  $K_B = 3$ ,  $K_A = 0.1$ , and  $K_H = 5$ , we get  $M_B = 4.3 \frac{H}{N} + 5C \log_C N$ . If we assume that  $\frac{H}{N}$  is a constant, then we see that the memory required for the binding algorithm is  $O(\log N)$ .

### 8.1.2 Number of Messages

First, we are interested in the number of periodic updates generated. If we assume that updates are generated at the rate of  $T_P$  updates per second per binding, then we have a total of  $K_B H T_P$  updates per second network-wide. If we further assume that each of these updates travels an average of  $\frac{D}{2}$  hops, where  $D$  is the diameter of the network, then we have  $K_H T_P \frac{D}{2}$  packets per second, and each link handles  $K_H T_P \frac{D}{2L}$  packets per second, where  $L$  is the total number of links in the network.

Now we calculate the number of packets due to queries. Assume that each host generates on the average  $T_Q$  queries per second. Assume further that, since queries go to the closest binding server, each query travels  $\frac{D}{K_B}$  hops (this is a rough but not unreasonable estimate). Then the

number of packets per second per link from queries is  $HT_Q \frac{D}{K_B L}$ .

Both periodic updates and queries are spread over time. They occur independently of topology changes. Now we are interested in the number of updates generated because a node gets a new address. When a node gets a new address, three things happen. First, it must send out  $K_B \frac{H}{N}$  updates as a host server. Second, assuming that all of the bindings the node was holding as a binding server will be resent,  $K_B \frac{H}{N}$  resend messages must be sent out. Finally, all of the  $K_B \frac{H}{N}$  bindings that it was holding will be sent to new binding servers. Again, each of these travels  $\frac{D}{2}$  hops, so we have a total of  $1.5DK_B \frac{H}{N}$  packets per address change.

Notice that this last equation is not expressed per link. That is because two thirds of these packets will not be spread evenly around the network—they will come from the node whose address changed.

We are interested in determining the number of packets sent over a link that is connected to a node experiencing an address change. Of the  $1.5DK_B \frac{H}{N}$  packets generated from one address change,  $DK_B \frac{H}{N}$  of them come from the node whose address changed, and an average of  $\frac{1}{E}$  of these will cross each attached link, where  $E$  is the average node degree. Therefore, the total number of packets seen by a link connected to a node whose address changes is  $DK_B \frac{H}{N} (\frac{1}{2L} + \frac{1}{E})$ . This link sees the most traffic of all links because, as one moves further away from the node with the changed address, the packets coming from that node are spread over more links.

If the node that gets a new address is an  $LM_1$  or higher, then multiple nodes (the offspring of the changed Landmark) will get new addresses all at once. All of these nodes will be sending out binding packets, and the links attached to the initiating node will be in the “center” of these changes, and will carry the most packets as a result. We are interested in the number of packets that this link sees.

Let's assume for simplicity that the link we are interested in is exactly in the center of the changes, rather than try to deal with the fact that more changes will come from the end of the link with the node initiating the changes. Let's also consider the 2 nodes attached to the link to be 0 hops away, the neighbors of those nodes to be 1 hop away, and so on.

There will be 2 nodes 0 hops away contributing packets to the link. There will be  $2(E-1)$  nodes 1 hop away contributing packets,  $2(E-1)^2$  nodes 2 hops away, and in general  $2(E-1)^x$  nodes  $x$  hops away. (This is similar to the portion of  $v(x)$  that grows geometrically.) However, only  $\frac{1}{E}$  of a node's packets will cross a link 0 hops away,  $\frac{1}{E(E-1)}$  will cross a link 1 hop away, and in general  $\frac{1}{E(E-1)^x}$  of a node's packets will cross a link  $x$  hops away. Therefore, a link in the center of a group of address changes will carry  $\frac{2(E-1)^x}{E(E-1)^x}$  or  $\frac{2}{E}$  packets for all nodes  $x$  hops away. If the  $X$



offspring for a given node are within  $x$  hops, then the link in the center of the changes will carry on the average  $\frac{2x}{E}$  of the packets sent by each offspring, plus a fraction of the packets sent by the host servers that have new binding servers, for a total average of

$$K_B \frac{H}{N} \left( \frac{XD}{2L} + \frac{4x}{E} \right)$$

packets. All other links will carry fewer packets.

Notice on one hand that this analysis did not consider the part of  $v(x)$  that flattens out. However, in general the increase in the number of nodes contributing packets to the link (numerator) is offset by the number of links that those packets are dispersed over (denominator). This will continue to hold true even after  $v(x)$  flattens out.

The number of packets actually seen by any one link can be much higher than this average. The actual amount depends on the topology. For instance, there may be topological funnel points where a large percentage of the binding updates may cross. On the other hand, one might expect such funnel points to be able to handle larger amounts of user traffic, so the effect of additional binding updates may not be so bad.

To get a worst-case feel for the kind of numbers we are talking about here, consider a network of 10,000 nodes, 100 hosts per node, node degree 3 (15,000 links), and diameter 30. Each global will have an average of 100 offspring, but let's assume that the global with the most offspring has 400. The 400 offspring are within 8 hops of the global. Assuming that  $K_B = 3$ , the center link will carry 3300 packets. Assuming further that these packets are spread out over say half a minute, we get roughly 100 packets per second. This is not an unreasonable number for T1 or faster links.

## 8.2 Dynamic Binding Algorithm Simulations

This Section is split into two parts, simulation description and simulation results.

### 8.2.1 Binding Algorithm Simulation Description

The simulations 1) measure the performance of the Binding Algorithm for some environment, and 2) determine the impact of various parameters on Binding Algorithm performance.

**8.2.1.1 Simulation Parameters.** Table 14 describes the parameters that are varied in the simulations.

The four networks simulated are the same as for the hierarchy simulations.

Over most of the simulation parameters, we simulated both 1) with the Binding Algorithm, and 2) with perfect knowledge of host addresses (the perfect binder). In other words, as a control, we compared the performance of the Binding Algorithm with a case where every node knew the address of every host at all times. If a host address changes, all nodes know instantly, without exchange of traffic.

Table 14  
Modifiable Binding Algorithm Simulation Parameters

PARAMETER	VALUES	COMMENTS
Network	n25d7c3 n50d7c3 n50d14c3 n50d7c6	(nodes/diameter/node degree) Different networks tell how the number of nodes, diameter, and the average node degree impact algorithm
Adjacencies	yes/no	running with or without adjacencies
$\frac{H}{N}$	1,5	number of hosts per node
$K_B$	1,3	the number of bindings per host
$K_H$	1,5	the number of entries in IHS for each routing table entry
$T_{update}$	1,4	Average time to forward routing update
Scenarios	Network Boot Change 1 Node Address Crash 1 Node Crash 2 Nodes Crash 4 Nodes Bring Up 1 Node Bring Up 2 Nodes Bring Up 4 Nodes Popular Destination	No link or node changes The remaining scenarios occur after the network has reached steady state

The simulations we performed for the Binding Algorithm unfortunately do not have a hierarchy. Because of time and resource limitations, we were not able to fully debug the code for resolution in a hierarchy in time, and instead run without a hierarchy, as though all nodes were global Landmarks. We believe that the major impact of running with the hierarchy is to slow down convergence, which would generally worsen performance.

The algorithm for handling adjacencies in our simulations is different than that described in Section 7. Instead of sending an adjacency update when a certain rate of queries is surpassed, a binding server sends one every time it receives a query. This adjacency update is sent to the neighbor that last handled the query. A binding server receiving an adjacency update simply holds on to it for a short period of time.

Our simulations also do not do the delays in sending out update or resend messages because of changes in the IHS or discrepancies between the resolution of two neighbors. The reason is that we discovered through our simulations that the delays are a good idea, but didn't have time to recode and resimulate. That being said, the Hierarchy Algorithm does delay sending the Binding Algorithm information about a Landmark appearing, but only because of the severe count-to-

infinity problem that exists in the Routing Algorithm we implemented. This count-to-infinity problem manifests itself as routing table entries (and therefore IHS entries) appearing and disappearing over and over, but each time with a larger distance. By delaying sending information about new routing table entries for a time longer than the count-to-infinity period, one can avoid most of the thrashing.

The scenarios are similar to those performed for the Hierarchy Algorithm simulations, with two exceptions. In one scenario, the address of one of the nodes is changed without any nodes coming up or going down. In another scenario (popular destination), all queries are for one destination host. This scenario is for the purpose of evaluating the adjacency update algorithm.

**8.2.1.2 Parameter Combinations.** The control set of parameters is adjacencies yes,  $K_B = 3$ ,  $\frac{H}{N} = 5$ ,  $K_H = 5$ , and  $T_{update} = 1$ . The impact of these parameters was tested by modifying the parameters one at a time, resulting in a total of 6 possible parameter sets. For every parameter set, we ran groups of simulations with all four networks, and for all nine scenarios. Within each group (a given parameter set, network, and scenario) 5 simulations were run, each time changing the random seed. We calculated the average and standard deviation for each set of 5 simulations.

In general, we saw a large variance within each set of simulations. As a result, our data has a larger margin of error than we would like. Nevertheless, it does give us a general idea of Binding Algorithm performance.

**8.2.1.3 Results.** We are mainly interested in two kinds of results: time to convergence and number of packets lost.

We don't show any data comparing the four networks. Generally speaking, we saw the same differences between the four seen in the data for the hierarchy simulations. That is, the longer diameter network takes longer to converge, and the larger node degree network converges more quickly. The reason, as with the hierarchy algorithm, is that convergence speed of the binding algorithm depends on convergence speed of the underlying routing algorithm.

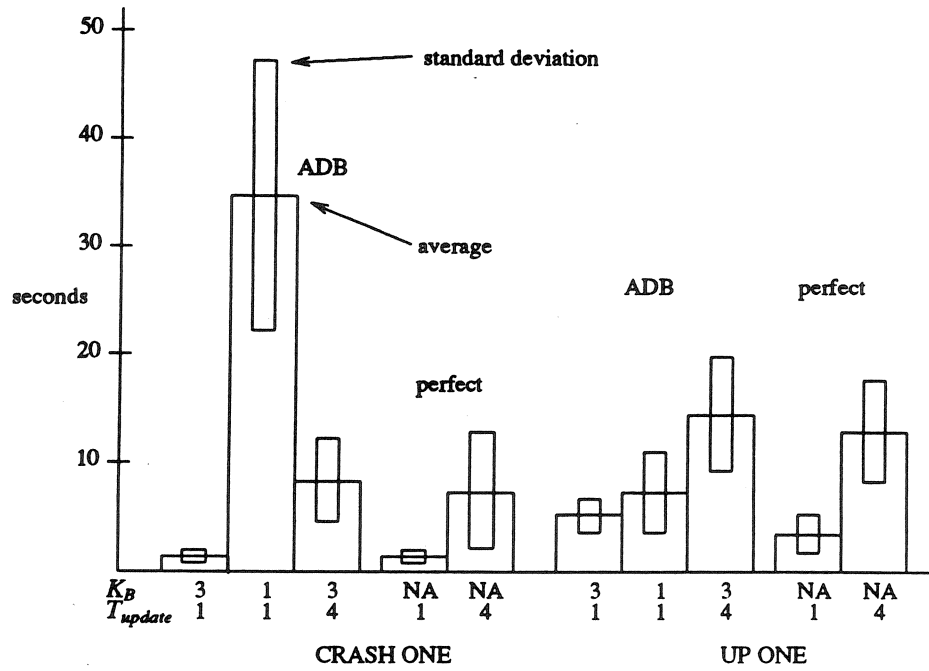
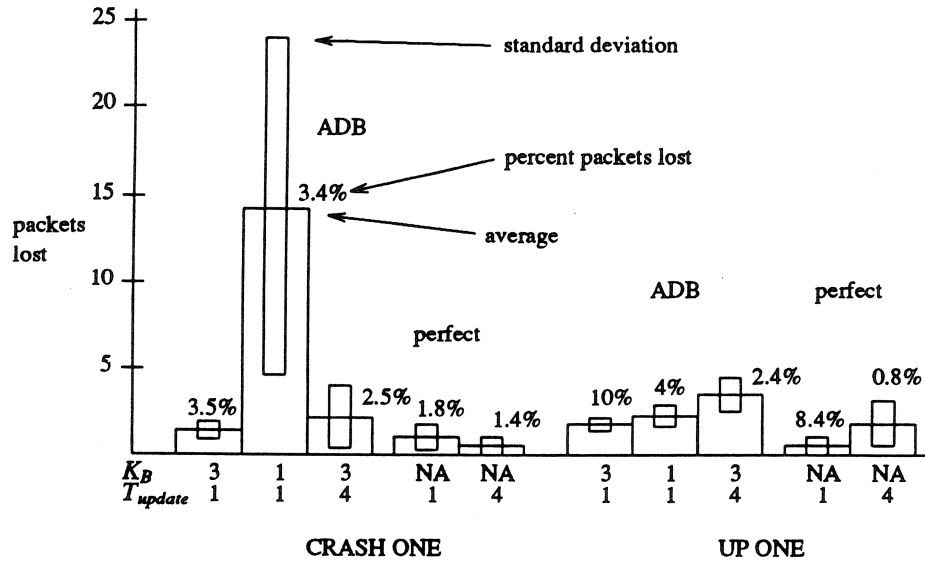
Figure 34 shows the impact of changing  $K_B$  and  $T_{update}$  on convergence time and number of lost packets. Both graphs are for the 50 node, diameter 7, node degree 3 network.

Notice in the graphs of Figure 34 that performance is worse when either the number of updates per host ( $K_B$ ) is reduced or the routing update speed  $T_{update}$  is increased. Notice also that the perfect binder shows better performance than the Binding Algorithm. These results are generally expected, although it is interesting to note that the perfect binder doesn't perform all that much better than the Binding Algorithm.

A very interesting data point is that for  $K_B = 1$ . Notice that the crash-one scenario for  $K_B = 1$  performs much worse than the others, and in particular, it performs much worse than the up-one scenario for  $K_B = 1$ . The root cause of the bad performance here is the count-to-infinity problem. During count-to-infinity, some IHS tables will be out of sync with others, and some percentage of queries will fail. When there are  $K_B = 3$  bindings per host, the probability that all of them will fail

Figure 34

Binding Performance by Number of Updates and Routing Algorithm Speed



is small. When there is only  $K_B = 1$  binding per host, then the probability of failure is much higher, as is indicated by the increased failure rate for that case in Figure 34.

Figure 35 shows convergence time and number of lost packets for the various scenarios, namely change of address (AD), crash one, two, and four nodes (C1, C2, and C4), and bringing up one, two, and four nodes (U1, U2, and U4). Again, the data in Figure 35 is for the 50 node, diameter 7, node degree 3 network.

First let's compare the up scenarios with the crash scenarios. For the perfect binder, the up scenarios consistently lost fewer packets than the crash scenarios. This is because in distance vector algorithms, bad news causes more thrashing than good news. However, for the Binding Algorithm (ADB), the up scenarios lost more packets than the crash scenarios. This is because of the delay in sending the new routing table information from the Hierarchy Algorithm to the Binding Algorithm to avoid the count-to-infinity problem. (Recall from Figure 34 that this didn't completely get rid of the effect of count-to-infinity, just reduced it.)

Next, notice that changing (crashing or bringing up) more nodes resulted in more lost packets and longer convergence. However, the increase in lost packets does not keep pace with the increase in changed nodes. This is to be expected, since the distance-vector algorithm converges more-or-less independently for each crash. Notice that the differences in convergence times for the various scenarios is not as marked as the differences for lost packets.

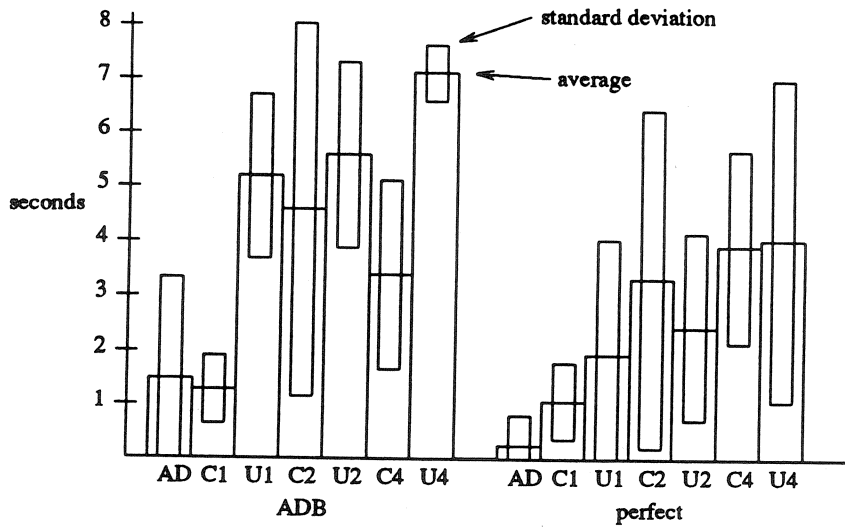
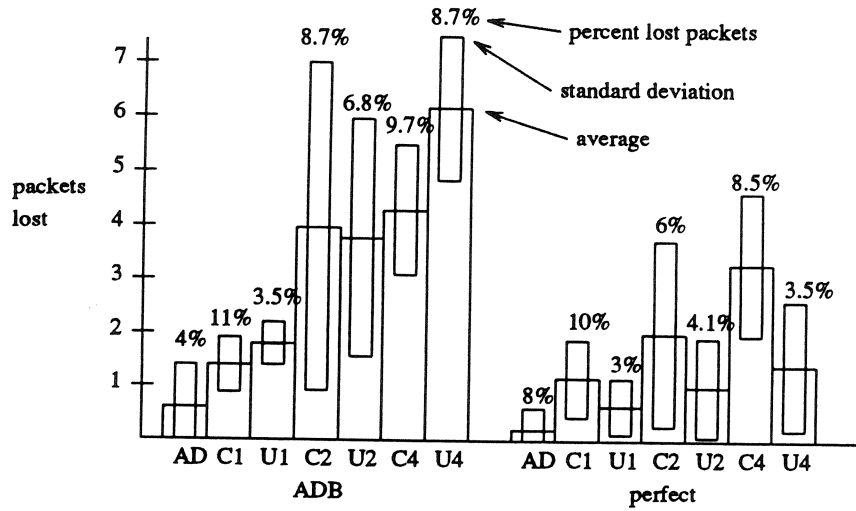
Finally, we compare the Binding Algorithm with the perfect binder. For the crash cases the Binding Algorithm generally loses about twice the number of packets as the perfect binder, and loses three or four times as many packets for the up cases. Correspondingly, the Binding Algorithm takes up to twice as long to converge as the perfect binder.

Generally, the performance of the Binding Algorithm is quite satisfactory. Notice that convergence times in all cases are less than 10 seconds, and that the percentage of lost packets during convergence is typically 7 or 8 percent. These numbers are well within the tolerances of a transport connection.

For the popular scenario, we measured the ratio of the number of replies made by the node with the most replies over the average number of replies for all nodes. With adjacency updates turned off, the ratio averaged around 17. With adjacencies turned on, the ratio was around 5. In spite of this, nodes held only 2 percent more bindings with adjacencies turned on.

Finally, we measured a significant number of mis-matched IHS tables during convergence. For the crash one scenario, mis-matched IHS tables accounted for 1/3 of the updates sent. We believe that this number would drop significantly if the delays discussed in Section 7 were implemented.

Figure 35  
 Convergence Time and Number of Lost Packets by Scenario



## 9 CONCLUSIONS

Detailed technical conclusions are given in the relevant sections and won't be repeated here.

The main conclusion to draw from this paper is that the algorithms developed so far seem workable and that the next phase of the project, implementation and testing, should begin. Because of time and resource constraints, the simulations were not as thorough as they should have been. We could have run more simulations to give us statistically more solid data, and we could have run more complex scenarios, thus stressing the algorithms more. Even so, the simulations have given good insights into the behaviour of the algorithms, and have helped in the design of good algorithms. Furthermore, the analysis of the completed algorithms also give insights into the expected performance of the algorithms. The implementation and testing phases will serve to stress the algorithms.

### 9.1 Future Work

In addition to implementing the algorithms discussed in this paper, simulation needs to be done on the Alternate-path Distance-vector Routing (ADR) algorithm sketched out in the previous paper (Tsuchiya, 1987b). More simulation is needed for the Binding Algorithm, in particular simulating with a hierarchy. Finally, simulation with administrative boundaries is still required. Of these, the major part is simulating ADR. The other simulations are relatively simple extensions of the algorithms already simulated.

The major work to be done, however, is implementation and testing in a testbed. This work will be performed during the next few years by the University of Maryland under DARPA contract.

## APPENDIX A

### Glossary of Mathematical Expressions

- $A_{link}$  The number of nodes that get new addresses when a link crashes or comes up.
- $A_{node,i}$  The number of nodes that get new addresses when an  $LM_i$  crashes or comes up.
- $C$  The average number of children per Landmark.
- $C_{max}$  The maximum number of children per Landmark. In our simulations,  $C_{max} = 7$ .
- $d_i$  The average distance from each node to its closest  $LM_i$ .
- $\hat{d}_i$  The distance from an  $LM_i$  for which nodes are closer to it than to any other Landmarks.
- $D$  The diameter of a network (the maximum distance between any two nodes).
- $E$  The average node degree of a network, where the node degree of a single node is the number of nodes it is connected to.
- $G$  The superscript  $G$  indicates global level. For instance,  $LM_i^G$  is a global Landmark.
- $G_{act}$  The current number of global Landmarks.
- $G_{req}$  The number of required global Landmarks ( $G_{req} = \sqrt{N_{est}}$ )
- $H$  The total number of hierarchy levels. The maximum number of levels in our implementation is  $H_{hier} = 11$ . The maximum number of levels in the `lm_addr` is  $H_m = 7$ . The level at which Landmarks become global is  $H_G$ .
- $i$  The Landmark Hierarchy level.
- $L$  The number of links in a network.
- $LM_i$  A Landmark of level  $i$ .
- $LM_i[x]$  An  $LM_i$  with `ID = x`.
- $N$  The total number of nodes in a network.  $N_{est}$  is the estimated number of nodes in the network (used when choosing globals).



$P_{link}$	The number of nodes that get new parents when a link crashes or comes up.
$P_{node,i}$	The number of nodes that get new parents when an $LM_i$ crashes or comes up.
$r_i$	The average radius of a group of $LM_i$ . When this appears in the text, the set of $r_i[x]$ which are being averaged is understood by context. If a single network or simulation is the context, $r_i$ is averaged over all Landmarks $LM_i[x]$ in the network. When an experiment (a group of simulations) is the context, $r_i$ is averaged over all Landmarks for all simulations in the experiment.
$r_i[x]$	The radius of a Landmark Vicinity for a particular $LM_i[x]$ .
$R$	The average total number of entries in node's routing tables ( $R[x]$ for a specific node). (When written as a subscript, it refers to the root level of the Landmark Hierarchy.)
$R_i$	The average total number of $LM_i$ in a node's routing table ( $R = \sum_{i=0}^H R_i$ ).
$T_{ch}$	Maximum time for the Configuration Algorithm to discover a host. Set to 5 in our simulations.
$T_{cn}$	Maximum time for the Configuration Algorithm to discover a neighbor node. Set to 2 in our simulations.
$T_i$	The total number of $LM_i$ in a network.
$T_r$	The period between routing updates in a periodic routing update scheme.
$T_{update}$	The average time it takes routing information to travel one hop. In general, $T_{update} = \frac{T_r}{2}$ .
$T_{packet}$	The average time it takes a packet to travel one hop.
$v(x)$	The number of nodes within $x$ hops of a node.

## REFERENCES

- Khanna, A. and J. Seeger (1986), *Large Network Routing Study Design Document*, Report No. 6119, Cambridge, MA: BBN Communications Corporation.
- McQuillan, J., I. Richer, and E. Rosen (May 1980), "The New Routing Algorithm for the ARPANET," *IEEE Trans. Comm.*, COM-28, pp. 711-719.
- Sparta Incorporated (1986), *Design and Analysis for Area Routing in Large Networks*, McLean, VA: Sparta Incorporated.
- Stine, Robert H. Jr. and Paul F. Tsuchiya (1987), *Assured Destination Binding: A Technique for Dynamic Address Binding*, MTR-87W00050, McLean, VA: The MITRE Corporation.
- Tsuchiya, P. F. (1987a), *The Landmark Hierarchy: Description and Analysis*, MTR-87W00152, McLean, VA: The MITRE Corporation.
- Tsuchiya, P. F. (1987b), *Landmark Routing: Architecture, Algorithms, and Issues*, MTR-87W00174, McLean, VA: The MITRE Corporation.
- Westcott, J. and J. Jubin (1982), *A Distributed Routing Design for a Broadcast Environment*, 1982 IEEE Military Communications Conference Record 32CH1734-3, Vol. 3, pp. 10.4-1 through 10.4-5.
- Westcott, J. (1982), *Issues in Distributed Routing for Mobile Packet Radio Networks*, 1982 Proceedings of Comcon, 25th IEEE Computer Society International Conference on Computer Networks, pp. 233 - 238.

## GLOSSARY

<b>ADB</b>	<b>Assured Destination Binding</b>
<b>ADR</b>	<b>Alternate-Path Distance-Vector Routing</b>
<b>ANSI</b>	<b>American National Standards Institute</b>
<b>BBN</b>	<b>Bolt Beranak and Newman Inc.</b>
<b>DARPA</b>	<b>Defense Advanced Research Projects Agency</b>
<b>DCA</b>	<b>Defense Communications Agency</b>
<b>DDN</b>	<b>Defense Data Network</b>
<b>DoD</b>	<b>Department of Defense</b>
<b>EGP</b>	<b>Exterior Gateway Protocol</b>
<b>FIFO</b>	<b>First-In-First-Out (Queue)</b>
<b>GGP</b>	<b>Gateway-to-Gateway Protocol</b>
<b>IP</b>	<b>Internet Protocol (DoD)</b>
<b>ISO</b>	<b>Organization for International Standardization</b>
<b>LAN</b>	<b>Local Area Network</b>
<b>LAP-B</b>	<b>Link Access Protocol, Balanced Mode</b>
<b>NSFNET</b>	<b>National Science Foundation Network</b>
<b>SPF</b>	<b>Shortest Path First</b>
<b>TCP</b>	<b>Transmission Control Protocol</b>
<b>TP4</b>	<b>ISO Transport Protocol, Class 4</b>

## DISTRIBUTION LIST

### *MITRE Washington*

A-10 B. Horowitz  
G. MacDonald  
C. Zraket

D-14 E. Brady  
R. Granato  
R. Harris  
J. Quilty

W-30 R. Binder  
G. Holt  
B. Moran  
T. Nyman  
F. Powers  
J. Rubin  
A. Schoka  
L. Stine  
E. Wells  
L. Wentz  
D. Wood

W-31 J. Ackermann  
S. Bhanji  
D. Gomberg  
R. Hansen  
W. Lazear  
P. Mellinger  
R. Miller (2)  
B. Mishra  
G. Reichlen  
A. Whitaker  
R. Wilmer  
Technical Staff  
Associate Technical Staff

W-33 J. Gravallese  
J. Just  
S. Turner

W-34 R. Evans  
T. Minton  
R. Pesci

W-35 C. Bowen  
M. Meltzer  
A. Messeh  
B. Price  
C. Smith  
M. Stella  
R. Ward

W-36 D. Jurenko  
W. Kinzinger

W-37 H. Marsh  
M. Zobrak  
D. Zugby

W-79 W. Blankertz

W-110 H. Duffield  
R. Paroline

W-144 E. Boyle

W-147 M. Abrams  
S. Schaen  
R. Shirey  
J. Vasak

### *MITRE Bedford*

D-110 P. Brusil  
G. Koehr

D-111 J. Woodward

**DISTRIBUTION LIST (Concluded)**

D-114 H. Bayard

D-115 S. Ames

Records Resources (2)

*External (Sponsors)*

COL T. Herrick, Code DD

Mr. E. Schonborn, Code DDD

Ms. G. Hix, Code DDI

LTCOL R. Law, Code DDE

COL A. Maughan, Code DDR

LTCOL J. Saunders, Code DDL

LTC D. Williams, Code DDC

Mr. J. Milton, Code DDO

LTC D. Gwin, Code DDN

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS N/A			
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Prepared for authorized distribution. It has not been approved for public release.			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MTR-89W00277		5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation		6b. OFFICE SYMBOL (If applicable) W-31	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) 7525 Colshire Drive McLean, Virginia 22102-3481		7b. ADDRESS (City, State, and ZIP Code)			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION (DCA) Defense Communications Agency		8b. OFFICE SYMBOL (If applicable) DDE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) 8th & South Courthouse Road Arlington, Virginia 22204		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.	PROJECT NO 359Z	TASK NO	WORK UNIT ACCESSION NO
11. TITLE (Include Security Classification) (U)Landmark Routing Algorithms: Analysis and Simulation Results					
12. PERSONAL AUTHOR(S) Paul F. Tsuchiya, Ron Zahavi					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 90-03-01		15. PAGE COUNT 35
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This paper is the third in a series of papers that document the research, development, specification, implementation, and deployment of a new routing technique called Landmark Routing. Landmark Routing is a distributed and adaptive hierarchical routing protocol for use in arbitrarily large networks and internets. Its primary features are that it is robust and durable in the face of rapid topological changes, that it is easy to administer, and that it provides name-based addressing. In this paper, two of the three major algorithms that make up Landmark Routing were simulated. The algorithms resulting from the simulations are specified. The algorithms are analyzed and the simulation results are presented. The conclusion from this work is that Landmark Routing still appears to be a viable technology, and that implementation and testing should commence.  Suggested Keywords: Routing, Hierarchical Networks, Hierarchies, Landmark Routing, Landmark Hierarchy, Addressing, Naming, Address Binding, Packet-Switching, Data Communications, Simulation					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL			22b. TELEPHONE (Include Area Code)		22c. OFFICE SYMBOL