

# **A Self-Configuring and Self-Administering Name System**

by

**Michael L. Feng**

Submitted to the Department of Electrical Engineering and Computer  
Science  
in Partial Fulfillment of the Requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science  
at the Massachusetts Institute of Technology  
February 2001  
Copyright 2000 M.I.T. All rights reserved.

Author \_\_\_\_\_  
Department of Electrical Engineering and Computer Science  
November 2, 2000

Certified by \_\_\_\_\_  
Dr. Amar Gupta  
Thesis Supervisor

Accepted by \_\_\_\_\_  
Arthur C. Smith  
Chairman, Department Committee on Graduate Thesis

**A Self-Configuring and Self-Administering  
Name System**

by  
Michael L. Feng

Submitted to the Department of Electrical Engineering and Computer Science  
in Partial Fulfillment of the Requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science  
at the Massachusetts Institute of Technology  
February 2001

**ABSTRACT**

A distributed system that stores name-to-address bindings and provides name resolution to a network of computers is presented in this thesis. This name system consists of a network of name services that are self-configuring and self-administering. The name service consists of an agent program and BIND, the current implementation of DNS. The DNS agent program automatically configures a BIND process during the start-up process and automatically reconfigures and administers the BIND process depending on the changing state of the network. This name system has a scalable and fault-tolerant design and communicates using standard Internet protocols.

Thesis Supervisor: Dr. Amar Gupta

Title: Co-Director of Productivity From Information Technology (PROFIT) Initiative

## **Acknowledgments**

First, I would like to thank Dr. Gupta for providing me the opportunity to work on a challenging and rewarding project. Also, I would like to thank Jim Kingston, Kevin Grace, and Michael Butler for providing the facilities, technical support, and direction for this thesis. A special thank you goes to Ralph Preston, Jared Burdin, and Sam Weibenson for making working on this project an enjoyable experience. Finally, I would like to thank my family and Jenna Yoon for their support in my endeavors.

# Table of Contents:

1 - Introduction.....	8
1.1 – Motivation.....	9
1.2 – Network without an Administrator .....	10
1.3 - Proposed Solution.....	11
2 - Research.....	14
2.1 - Evolution of DNS .....	14
2.1.1 - ARPAnet – HOSTS.TXT .....	15
2.1.1 - Grapevine.....	16
2.2 – Domain Name System (DNS).....	18
2.2.1 - The Name Space .....	18
2.2.2 - Name Servers.....	20
2.2.3 - Resolvers.....	22
2.3 Arrangement of DNS name servers .....	24
2.3.1 Root Servers .....	24
2.3.2 Organization’s Name Servers .....	25
2.3.3 Research on Design for Name Servers .....	25
2.4 – Dynamic Updates in DNS.....	26
2.4.1 – Dynamic DNS Early Research: PHASE.....	26
2.4.2 – DNS UPDATE Protocol .....	27
2.4.3 - Dynamic DNS with DHCP.....	28
2.4.4 - Secure DNS Updates .....	29
2.4.5 - Integrity Between Dynamic DNS and DHCP .....	30
2.4.6 - Vendor Products Integrating Dynamic DNS and DHCP.....	31
2.4.7 – Solution for Mobile Environments .....	32
2.5 – Using DNS with Other Systems .....	32
2.5.1 - Attribute-base Naming.....	33
2.5.2 - Web Servers.....	34
2.6 - Administrator-Less DNS .....	35
2.6.1 – <i>nssetup</i> .....	36
2.6.2 - DNS Administration using Artificial Intelligence .....	37
2.6.3 - ZEROCONF – Zero Configuration Networking .....	38
2.7 – NetBIOS.....	39
2.8 – AppleTalk.....	41
3 – Implementation .....	44
3.1 - Background.....	44
3.2 - Specifications/Requirements .....	46
3.3 - Solution/Approach.....	47
3.3.1 – Name Service Interface .....	47
3.3.2 – Controlling the BIND Process .....	49
3.3.3 – Internal Root Servers .....	49
3.3.4 – Initial Configuration.....	50
3.3.5 – DNS Messages .....	51
3.3.6 – Agent Messages .....	52

3.3.7 – Secondary Name Servers for Zones.....	53
3.4 – State Information.....	53
3.4.1 – Name Service Information.....	54
3.4.2 – Zone Information .....	55
3.4.3 – Message Logs .....	57
3.4.4 – Miscellaneous .....	57
3.5 – Agent Messages .....	58
3.5.1 - <i>discover</i> .....	59
3.5.2 - <i>leave</i> .....	59
3.5.3 - <i>getroot</i> .....	60
3.5.4 - <i>becomeslave</i> .....	61
3.5.5 - <i>negotiatemaster</i> .....	61
3.5.6 - <i>forceslave</i> .....	63
3.6 – Scenarios .....	63
3.6.1 - Start-up .....	65
3.6.1.1 - No Name Services Discovered .....	65
3.6.1.2 - At Least One Configured Name Service Discovered .....	67
3.6.1.3 - Only Non-Configured Servers Discovered.....	68
3.6.2 - Configured .....	69
3.6.2.1 - No Servers Discovered .....	70
3.6.2.2 - Discover Only Unconfigured Servers.....	70
3.6.2.3 - Discover Configured Servers.....	70
3.6.2.4 - Leave Message .....	74
3.6.3 – Get Slave process.....	76
3.7 DNS messages.....	77
3.8 Implementation Notes.....	82
4 – Analysis.....	83
4.1 – Name System .....	83
4.2 – Testing the Name System .....	85
4.3 – Memory Usage of Name System .....	88
4.4 – Bandwidth and Processor Load of Name System.....	89
4.4.1 – DNS Query and Update Messages.....	89
4.4.1.1 – Special Processing for DNS Update Message .....	91
4.4.2 – Zone Transfers .....	91
4.5 – Analysis of Agent Messages .....	92
4.5.1 – Configuration.....	93
4.5.1.1 – Start-up Scenario: No Name Services Discovered .....	95
4.5.1.2 – Start-up Scenario: At Least One Configured Name Service Discovered .....	95
4.5.1.3 – Start-up Scenario: Only Non-Configured Servers Discovered .....	98
4.5.1.4 – Comparison of Configuration Process With Other Systems.....	99
4.5.2 – Administration .....	101
4.5.2.1 – Configured Scenario: No Servers Discovered .....	103
4.5.2.2 – Configured Scenario: Discover Only Unconfigured Servers.....	103
4.5.2.3 – Configured Scenario: Discover Configured Servers.....	103
4.5.2.4 – Configured Scenario: Leave Message .....	105

4.5.2.5 – Comparison of Administration Process with Other Systems.....	106
4.6 – Comparison with ZEROCONF.....	109
4.7 – Comparison with NetBIOS .....	110
4.8 – Comparison with AppleTalk .....	112
5 – Future Work .....	114
5.1 – <i>Discover</i> Messages.....	114
5.2 – Use of Internal Roots .....	115
5.3 – Extension of Name System.....	115
5.4 – Security Issues.....	116
5.5 – Miscellaneous .....	116
6 – Appendix.....	117
6.1 – Agent Messages .....	117
6.2 – Sample <i>named</i> Configuration Files.....	119
7 – Bibliography.....	121

# List of Tables

Table 1 – Server Record Fields.....	54
Table 2 – Types of Server Record Lists.....	55
Table 3 – Header Fields .....	58
Table 4 – Possible Opcodes .....	59
Table 5 – Output of top: Memory usage of name service.....	88
Table 6 – Participants Background .....	101
Table 7 – Time use for configuring a name server .....	101
Table 8 – Messages for Reconfiguration .....	105

# List of Figures

Figure 1 – Sample Network .....	11
Figure 2 – The Two Processes of the Name Service .....	13
Figure 3 – DNS database vs. UNIX filesystem .....	19
Figure 4 – Delegation of a Domain into Zones .....	20
Figure 5 – Resolution of mikefeng.mit.edu in the Internet .....	23
Figure 6 – Update Message Format .....	27
Figure 7 – Interaction Between DNS and DHCP .....	29
Figure 8 – Information from Manager forwarded to Name Service .....	45
Figure 9 – Interaction Between Agent, BIND Process, and Other Processes .....	48
Figure 10 – Sample Set-up of Name Services .....	50
Figure 11 – Agent Message Format .....	58
Figure 12 – States in the Start-up Scenario .....	64
Figure 13 – New Manager is First to Enter Network .....	65
Figure 14 – Obtaining Root Server Information from Configured Manager .....	67
Figure 15 – Discover States in Configured Scenario .....	71
Figure 16 – Obtain Root Server Information from Configured Manager .....	72
Figure 17 – SOA query to Root Server .....	73
Figure 18 – negotiatemaster message .....	73
Figure 19 – Leave States in Configured Scenario .....	75
Figure 20 – States for DNS Messages .....	79
Figure 21 – Testing the Name System .....	85
Figure 22 – $N_m$ vs. $n$ for At Least One Configured Found during Start-up .....	97
Figure 23 – $N_m$ vs. $n$ for None Configured during Start-up .....	99
Figure 24 – $N_m$ vs. $n$ for $m=3$ in Configured State .....	104

# 1 - Introduction

Even with all the recent technological advances in the field of computers to make the computer field more user-friendly, it is still a difficult task to set up a computer network. For example, it is not a trivial task to configure a Local Area Network (LAN) for a company or organization. There are many pieces that go into a computer network, and expertise is required to configure and administer each part of the network. Also, since the parts of network are interdependent on each other, the behavior of each part in relation to other parts has to be closely monitored. Examples of objects that make up a computer network are mail servers, web servers, routers, DNS servers, and ftp servers. These objects provide services for users, and individual hosts/users connect to these services to communicate with other machines and obtain information from the network.

When putting together a network, the first thing that needs to be determined is what services are required for the particular network in question. After this is determined, each service needs to be configured to serve the network and handle user requests. As soon as a service is configured, the service will have to make its existence and location known to the rest of the network [29]. This is so that the services will know of each other, and users will know where to obtain these services. Following this process of service discovery, a communication protocol needs to be agreed upon by the users and the network services to allow them to talk with each other. Once all the required services have been configured and are running, hosts can join onto the network and perform client tasks.

Currently, this process of configuring a network is performed by trained network administrators who manually configure services and notify hosts of which services exist

and how to communicate with the services. Software packages exist that assist in the building of a network [19], but many times, this task of configuring a new network continues to be a time-consuming one that can be very frustrating and full of unexpected errors. Furthermore, the role of a network administrator is not finished after the initial set-up of the network. The administrator also has the task of maintaining the network. For instance, if a service mysteriously becomes error-prone, the administrator is required to detect this error and fix the problem. Also, the administrator needs to monitor incoming and outgoing hosts in the networks and has to be able to add or delete services depending on the needs of the users. Much practice and experience is required to set up and to maintain a network, and it would be useful if this expertise could be captured in a program that could perform many of the administrator's tasks.

In particular, the configuration and administration of the naming portion of the network can be automated so that no human administrator is required to set up naming services. New hosts can join onto the network, and other hosts and services will be able to contact the new hosts by their names.

## **1.1 – Motivation**

Hence the motivation of this thesis is to design and implement a name system that is self-configuring and self-administering. The name system will consist of a network of name services that reside on different servers machines. The name service will consist of an agent program that controls a local DNS process. The agent programs handle the task of configuring and administering the name services, and the agent programs decide how to control the name servers based on the changing conditions of the network. This name system will work in the framework of a larger autonomous network. Before describing

the name system, it would be useful to describe the larger administrator-less network the name system will be operating in.

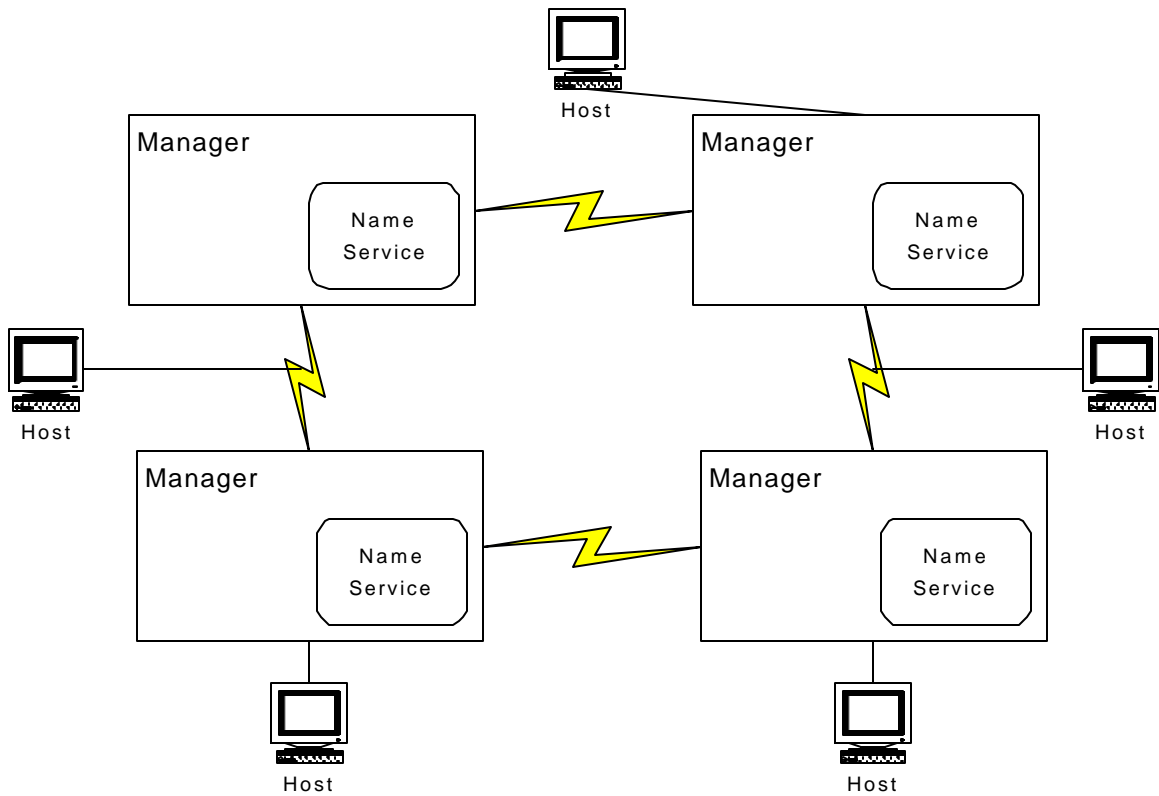
## **1.2 – Network without an Administrator**

The self-configuring and self-administering name system will operate in an administrator free network. This administrator-less network consists of two different types of machines. The first type of machine is called a manager and acts as the server. The second type of machine is the host, or user, and acts as the client. The managers form the backbone of the network and provide services to the hosts in the network. Each host will have a predetermined name and access to a manager, and the manager facilitates the interaction between hosts. These machines can form any topology, and like the standard communication protocol currently being used, these machines will use the IP protocol to communicate with each other.

At a minimum, each manager will be required to have a discovery mechanism, packet routing capabilities, an address allocation scheme, and knowledge of the hosts in the system. First, the discovery mechanism is used to determine what other managers exist in the system. Once the manager knows about the other managers in the system, the manager will be able to route packets to their correct destination. When a new host joins the network, a manager will use the address allocation feature to assign an IP addresses to the new host. After the address assignment, the manager needs to store that host name and address information so that there will be a record of how to contact the new host in the future. Also, other managers will need access to this host name-to-address information so that they too will know how to contact the new host.

### 1.3 - Proposed Solution

A solution to the problem of storing the host name-to-address information among all the managers is the presented in this thesis. This solution requires that each manager has already discovered all the other managers in the system, that the packet routing is working, and that the new host was assigned an IP address correctly. These requirements were satisfied through efforts by engineers at the Research Corporation.



*Figure 1 – Sample Network*

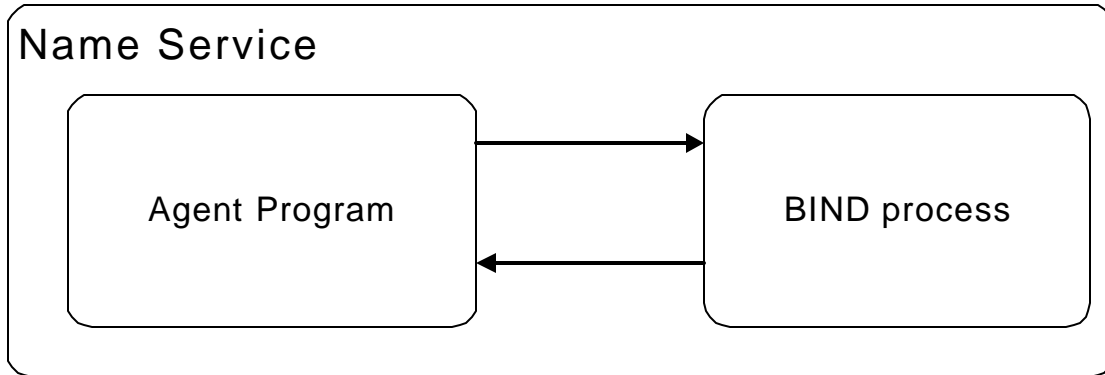
The proposed solution runs as a name service on all the managers in the network [Fig. 1]. This name service stores the name-to-address bindings of each host in the network and provides name to address resolution for host names. The name service will be notified whenever a new host is assigned an IP address. The name service will store

this information into its database and in turn notify other managers of the location of this new name-to-address information. All the other name services on all the other managers will then know about the new host. So, a user will be able to contact any machine in the network by simply knowing the name of that machine. The user will give the name service a host name, and the service will return the IP address associated with the host name. And with that IP address, the user will be able to communicate with the desired machine.

This name service system will have to be scalable and fault-tolerant. The network should be able to handle a large number of hosts, and thus the name service should be able to handle many name-to-address bindings. Also, name information should not be lost completely if one manager/name service fails. To achieve fault-tolerance, replication of information will be needed. Also, the name service will also need to handle updates and deletions of name data very quickly. Many of these requirements for the name service have already been solved by the Domain Name Server (DNS) system [21][22]. DNS achieves scalability by using a hierarchical storage and look-up of names and addresses. DNS achieves fault-tolerance by having multiple servers store the same information. Also, DNS is able to handle dynamic updates, and these updates can be transferred to replicated servers within minutes. The Berkeley Internet Domain (BIND) system, the current implementation of DNS, fulfills all of these requirements well.

However, BIND assumes a slow changing network topology and relies on an administrator to maintain its configuration files. With the configuration-free and administrator-free network, managers and hosts might enter and leave the system freely causing the need to change configurations and name bindings. The proposed solution

takes this into account by running an agent program that will change the BIND configuration file based on the current network state. Every time the agent program makes a change to the configuration files, the agent sends a signal to BIND to reread its configuration files. So, the proposed solution is a name service running on each manager that consists of BIND and an agent program that controls what BIND does [Fig. 2].



*Figure 2 - The Two Processes of the Name Service*

The thesis will be split into four sections. First, there will be background on the domain name server (DNS) system and research being done on using DNS in different ways. Second, the name service system will be explained in detail. Third, there will be an analysis on how well the name service system works. Fourth, there will be a future works section.

## **2 - Research**

As stated in the introduction, the focus of this thesis is to develop a program that works with the Berkeley Internet Name Domain package (BIND), the current implementation of the Domain Name System (DNS), to handle name-to-address resolution and dynamic name updates. Before the functions of the program are described, there has to be an understanding of the different parts and terminology of the Domain Name System. Also, there are many extensions to DNS that allow DNS to work in many different environments. In this chapter, a brief overview of the history of name-to-address storage leading to the development of DNS, a detailed explanation of DNS components, the set-up of DNS, the dynamic update DNS extension, DNS integrated with other systems, and descriptions of research performed on administration-free and configuration-free DNS are given. In addition, the naming schemes used in the NetBIOS and AppleTalk systems will be discussed.

### **2.1 - Evolution of DNS**

In a typical network, each host or server computer has a name that identifies that host or server in the network. In addition, each machine is assigned an IP address, a 32-bit number that says where that machine is located in the network. A computer uses this IP address whenever it wants to communicate with another computer. In this sense, a person wanting to access files from a server would have to memorize the 32-bit number assigned to the server. For people in general, memorizing this number would be difficult because IP addresses can be up to 12 digits long. It would be simpler just to remember the name of the machine and have a service that returns the IP address of a machine given

the name. This service would store all the mappings between host names, a format humans find convenient, and Internet addresses, which computers use.

In the past, there have been different solutions for this name to address mapping service. These solutions solved the problem for their current situations, but changing conditions in the networks they were used for made these solutions outdated. Two notable name-to-address systems are described in this section.

### **2.1.1 - ARPAnet – HOSTS.TXT**

Back in the 1970s, the Internet, known as the ARPAnet at the time, was a small network consisting only of a few hundred hosts. Because of the small size of the ARPAnet, the simple solution of storing the name-to-address mappings of every host into a single file, HOSTS.TXT, was sufficient. Communication with other hosts by name involved looking in the HOSTS.TXT file for the address associated with the name. This HOSTS.TXT file was maintained by Stanford Research Institute's Network Information Center (NIC), with administrators e-mailing changes to the NIC and periodically *ftping* the NIC server to obtain a copy of the current HOSTS.TXT file [21].

However, as the ARPAnet grew, this scheme became unworkable. The size of HOSTS.TXT file grew larger and larger, and the network traffic and processor load generated by the update process of the file was becoming unbearable [2]. Also, name collisions were becoming a problem, and it was becoming increasingly difficult to maintain the consistency of the HOSTS.TXT file across all the hosts in the expanding network. Fundamentally, this single file mechanism was not scalable, and while the solution worked fine for a small network, a new more scalable and decentralized solution was required as the ARPAnet expanded.

### 2.1.1 - Grapevine

Another solution to the name-to-address mapping problem was the Grapevine system developed and used by Xerox PARC in the early 1980s. The Grapevine system was a distributed and replicated network that provided facilities for message delivery, naming, authentication, and resource location [5]. Grapevine consisted of a set of computer servers that were placed in the Xerox network based on the load and topology of the clients being served. These computers combined to make up the registration database. The names in the registration database were structured as a two-level hierarchy in the form of  $F.R$ , where  $R$  is a registry name and  $F$  is a name within the registry. There could be many registries in the Grapevine system, and the name information of these registries would be stored on different Grapevine servers. A set of servers would store the name information of a registry, and these servers update each other of any name changes in that registry. Replicating the registry information on different servers helped to make the system fault-tolerant. If one Grapevine server failed, there would be another server who knew about the name information of the failed server.

Also, only a small set of Grapevine servers was in charge of a particular registry. This distribution of registry information made the system decentralized and easier to manage [5]. Whenever a name was added to a registry, only a few servers had to know about the name. This kept the name replication traffic to a minimum and allowed groups to control their own registry information. In addition to storing the name information for individual registries, each Grapevine server stored the Grapevine registry information. This registry information contained which Grapevine servers each registry was assigned to. So, even if a Grapevine server did not have name information on a particular registry,

that server could use its Grapevine registry information to find out where in the network to locate the name information for that registry. Thus, even though name information was distributed throughout the network, each Grapevine server would know, directly or indirectly, about all the registry name-to-address mappings in the system.

The Grapevine system was implemented and worked well in the Xerox network to handle name-to-address mappings, but their solution was not being used outside of Xerox. There was a later version of Grapevine called Clearinghouse, but like Grapevine, Clearinghouse was only used in a company intranet [25]. Grapevine's naming scheme was not standard, and the Grapevine's design still needed to be refined. Even though the Grapevine system did not become the standard, many of the ideas used in developing Grapevine would be used in the Domain Name System (DNS) proposed by Paul Mockapetris in 1984. DNS would become the Internet standard for storing name-to-address mapping and handling name-to-address resolution.

## **2.2 – Domain Name System (DNS)**

The main purpose of Domain Name System (DNS) [21][22] is to provide a mapping from the human readable domain names to the numerical IP addresses used to identify hosts on the Internet. Also, DNS can store mailbox information, aliases, or any other information about a domain name. Because of the large number of domain names and frequency of updates to names in the Internet, DNS was designed as a distributed database. Each portion of the name space is administered by a different organization, and the organization takes care of any updates that occur and any other maintenance required in their assigned name space. Each organization runs name servers that handle queries for domain name information, and the hosts requesting the name data use programs called resolvers to locate which name server to contact. This section will describe in detail the three main components of the DNS system: the name space, name servers, and resolvers.

### **2.2.1 - The Name Space**

The name space for the Internet is a variable-depth, inverted tree structure similar to the one used in a Unix file system [Fig. 3]. At the top is the root node, and each node is also the root of a new subtree of the overall tree. Each node represents a part of the domain name system called a domain, and each node has an associated label that names the domain. Domain names, used as indexes into the DNS database, are hierarchical and each domain has a unique domain name [21]. The format of the domain name is well known and uses the dotted notation. The full domain name of any node in the tree is the sequence of labels on the path from that node to the root. Domains can hold information about both names and subdomains. The data associated with the names in the domain name space are called resource records. The resource record can be of many different

types, including the IP address, name server, or mailbox information for a host. The client accesses this information by specifying the resource record type.

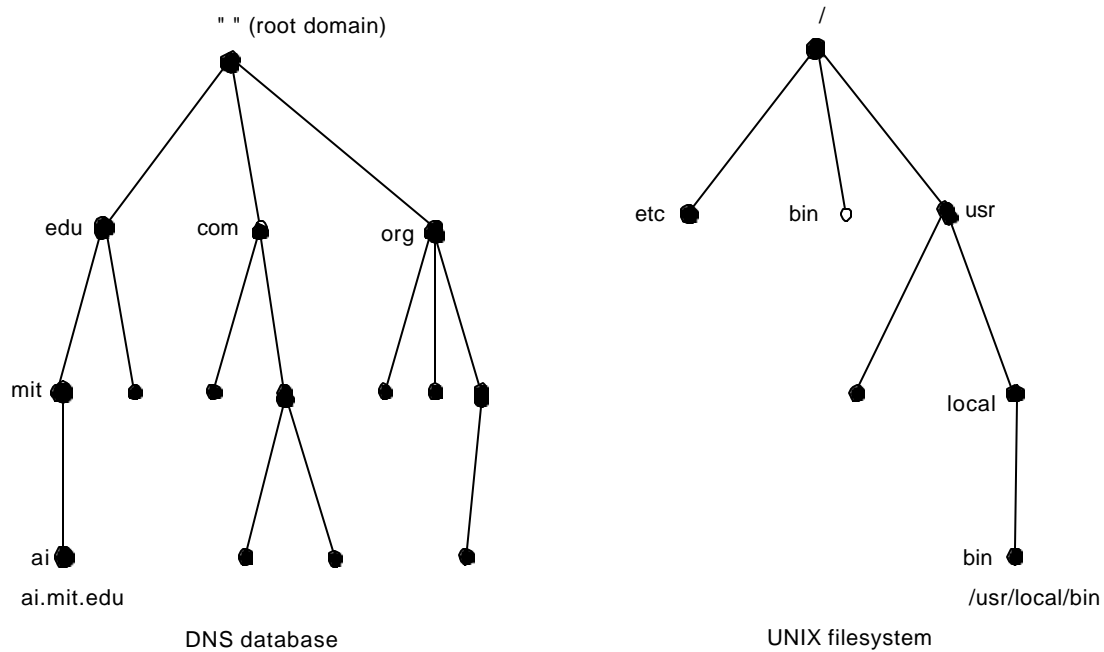


Figure 3 – DNS database vs. UNIX filesystem

Each domain can be administered by a different organization, and each organization is free to name hosts and can further break its domain into subdomains as it sees fit. A domain is called a subdomain if it is contained within another domain, and the subdomain's name label includes the name label of the parent domain. The organization can either have authority over the subdomain or delegate responsibility over the subdomain to another organization. Because domain names are unique, the organization can perform this task of naming hosts and subdomains without the worry of name collisions. The organizations will also maintain name servers that store the domain name information.

## 2.2.2 - Name Servers

Name servers are the repositories of information that make up the domain database. Each name server has complete data about the parts of the domain name space for which it is responsible. The information for a part of the name space is called a zone, and if a name server has the name information for a zone, then the name server is said to be authoritative for that zone. Name servers can be authoritative for multiple zones.

The difference between a zone and a domain is important, but subtle [2]. A zone

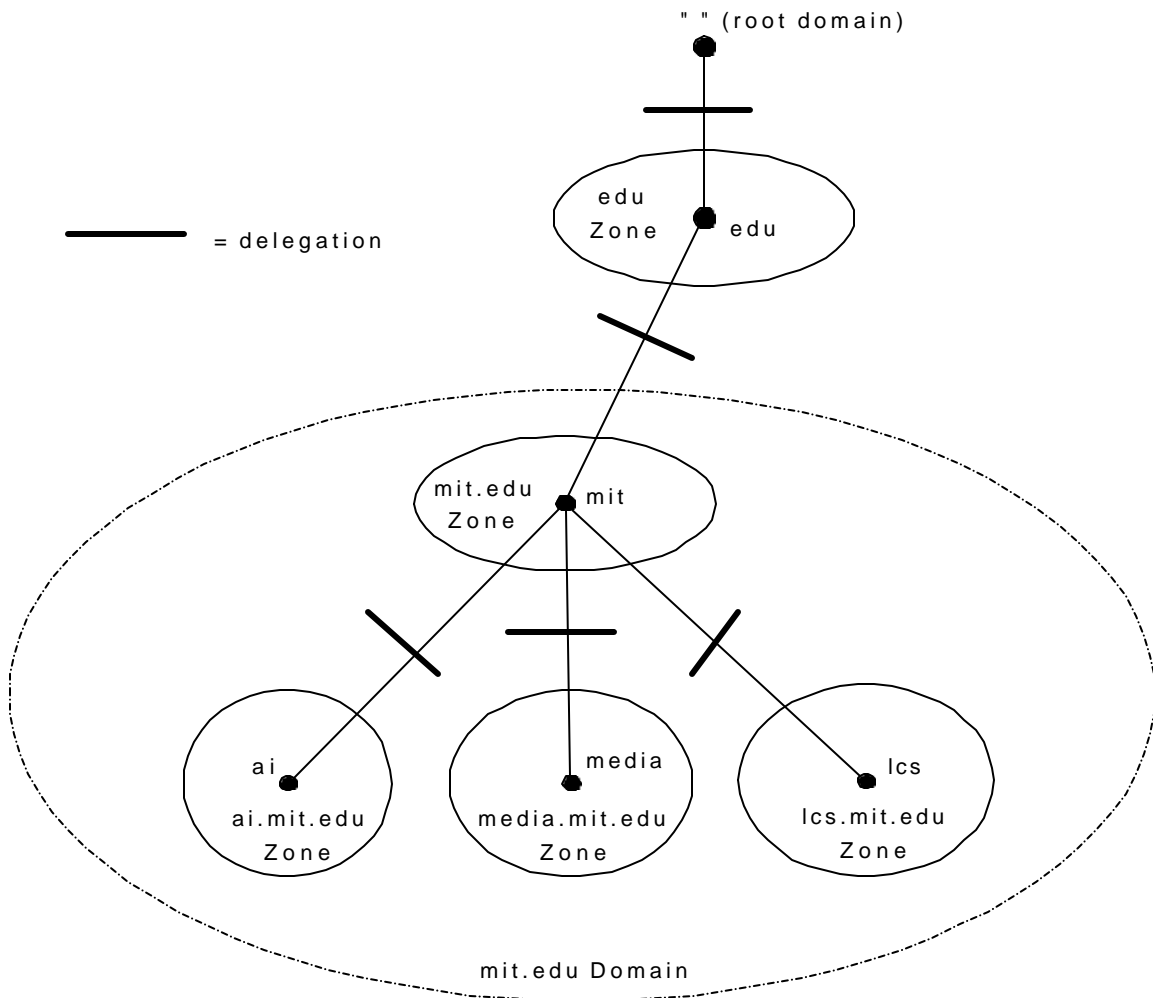


Figure 4 – Delegation of a Domain into Zones

is a subset of a domain. Zones contain all the domain name information that the domain with the same domain name contains, except for the domain name information in delegated subdomains. Domains can be delegated into subdomains maintained by other name servers. These subdomains will make up their own zones and subdomain zones. The parent zone name server stores pointers to the name servers that are authoritative for a subdomain. Storing these pointers is important for the name resolution process. For example, Figure 4 shows how the *mit.edu* domain is split and delegated into smaller zones. Authority for the *ai.mit.edu*, *media.mit.edu*, and *lcs.mit.edu* zones are delegated onto different name servers. The name server for the *mit.edu* zone will have information on which name servers are authoritative for the delegated zones.

Each zone has two or more authoritative nameservers that are responsible for keeping information about that zone up-to-date [21]. This replication of nameservers allows for fault-tolerance in the system. There are two types of authoritative name servers, the primary and the secondary. The first type of name server, called the primary name server, holds a master file that contains all the resource records for that zone. Administrators manually edit this master file whenever there are any changes in the names or name data of the zone. Whenever any changes to the master file are made, the primary name server rereads this file to obtain the new state of the zone. The second type of name server, the secondary name server, periodically retrieves a copy of the master file from another server, called the master server. This process of transferring name information is called a zone transfer. The server requesting a zone transfer is called the slave server, and the server receiving the zone transfer request is called the master server. Typically, a secondary name server will request zone transfers from the primary name

server, but it is also possible for a secondary name server to fetch data from another secondary server.

### 2.2.3 - Resolvers

A resolver is a program, typically a system routine, that communicates between name servers and their clients [2]. Usually, resolvers run on network hosts and initiate DNS queries or lookups on behalf of network applications. The resolver knows about at least one local name server where it can direct DNS queries. When queried, the local name server might either respond by answering the question directly, referring the client to another set of name servers, or signal an error condition.

Figure 5 shows an example of a sample query. First, the client machine notifies the resolver that it wants to know the IP address for the domain name *mikefeng.mit.edu*. The resolver sees this client request and sends a query to the local name server. The local name server does not have information for this name, so it queries a root server for information on *mikefeng.mit.edu*. From the root server, the local name server receives the name server information for the *edu* domain. Next, the local name server queries the *edu* name server. The local name server is once again referred to another name server, this time to the *mit.edu* name server. Finally, when the local name queries the *mit.edu* name server, the IP address for *mikefeng.mit.edu* is returned to the local name server. That answer is then returned back to the resolver. After this process, the client machine will know the IP address of the domain name *mikefeng.mit.edu*.

This name resolution process can either be done recursively or iteratively [2]. In a recursive query, the local name server has to respond with the requested data or with an error message. In an iterative query, if the local name server does not know the requested

information, it can refer the resolver to another name server. The resolver receives a list of other name servers to query, and is responsible for sending out additional queries to this list of name servers.

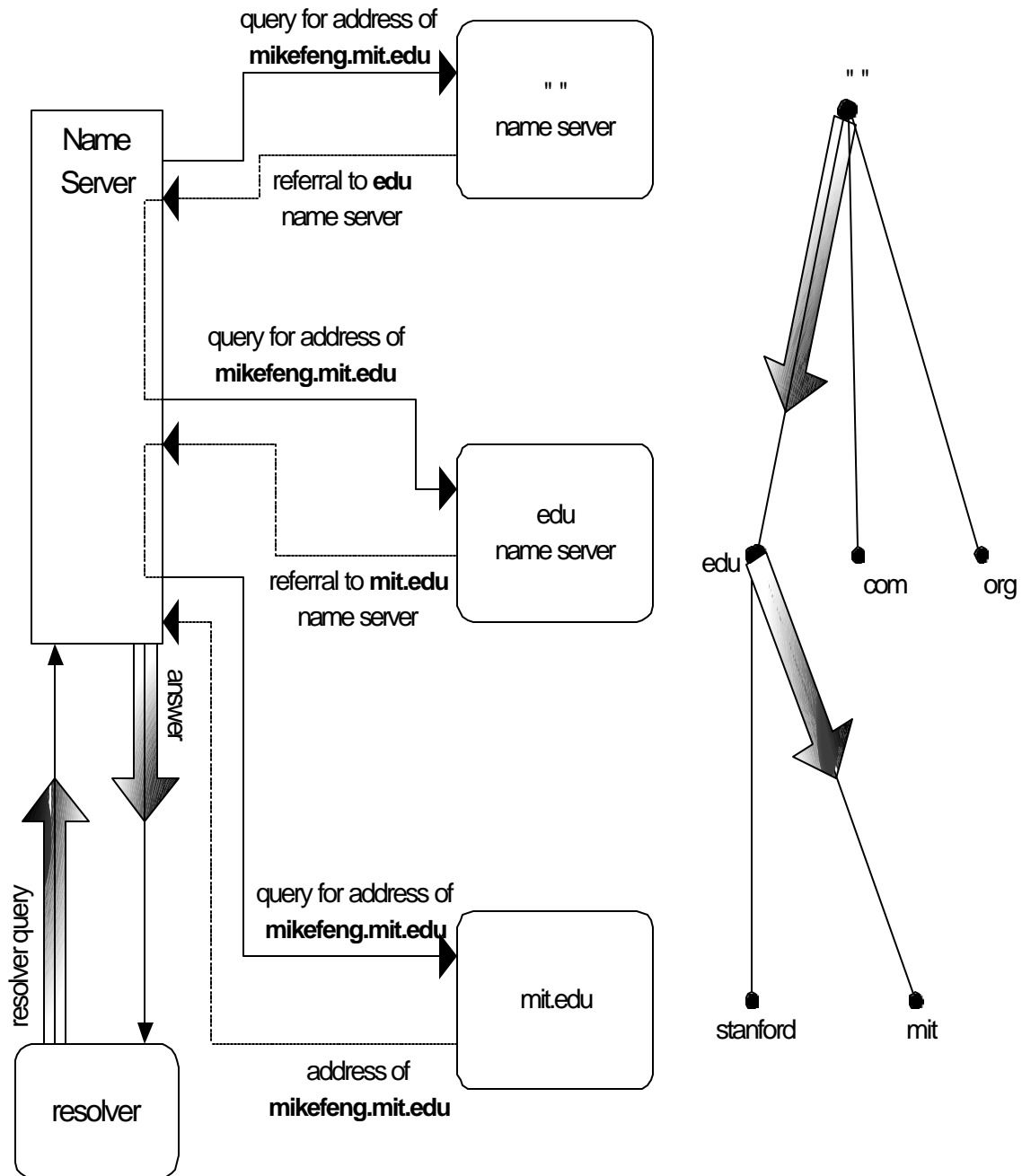


Figure 5 – Resolution of `mikefeng.mit.edu` in the Internet

## **2.3 Arrangement of DNS name servers**

The DNS consists of resolvers and name servers working with each other to store information about the name space. There is much flexibility in how these resolvers and name servers can be arranged. In general, resolvers will reside on each client machine, and as long as the resolver is properly configured with a local name server, the resolver will be able to resolve any name in the Internet. Name servers can be placed in any location as long as they can service the resolvers and the computers in their zone. A requirement for name servers is that they are replicated to ensure fault-tolerance. In case a zone's DNS name server fails, there will exist another server that will have the zone's name data and will be able handle name resolutions and updates. There is a collection of name servers that store name information for the root domain, and organizations can be flexible in how they control the name servers for the lower-level domains.

### **2.3.1 Root Servers**

There is a set of root servers that service the root domain. These root servers know which name servers hold the authoritative data for the top-level domains and are very important to the name resolution process. Whenever a name server does not know of a name, that name server can query a root server, and the root server will respond with information on where to find the requested name information. Because root servers play such a large part in resolving names, there are thirteen of these root servers in the Internet [2]. They are spread out geographically across the world and are maintained by InterNIC. Whenever an organization with a new domain name comes into existence, the organization will register this domain name with the InterNIC, and InterNIC will add this information into its root servers.

### **2.3.2 Organization's Name Servers**

After the domain location is stored in the root servers, the organization is free to organize its name servers to service its domain name space. An organization's name servers can be arranged in any way, but there are some general policies to follow to ensure availability. For an organization, there should at the minimum be two name servers so that the failure of one name server does not mean all name resolution will fail. It is not uncommon for an organization to run from five to seven servers [2]. In addition, there should one name server for each network or subnet the organization has, and name servers should run on large time-sharing machines like file servers. Taking these guidelines into consideration will speed up look-up times for names and help prevent servers from becoming overloaded with name requests. Also, it is useful to run one name server offsite in case the network's name servers become unreachable.

### **2.3.3 Research on Design for Name Servers**

An interesting proposal for a new design for DNS involves storing the entire DNS database on a set of replicated servers [16]. Storing all of the DNS data will make the DNS system highly responsive to changes in DNS information and significantly reduce DNS lookup times. These replicated servers will be geographically distributed and distribute any updated name information over a satellite channel or over terrestrial multicast. This proposal was inspired by the continued decrease in the cost of disk storage and advances in broadcasting information. For now, this proposal is still just an idea, but this proposal is an example of the flexibility allowed when setting up name servers.

## **2.4 – Dynamic Updates in DNS**

With its network of resolvers and name servers, the DNS system became the standard for storing and resolving names in the Internet. However, as the Internet and user needs have changed through the years, the DNS system has been tweaked to adapt to these changes. Many refinements and additions have been made to the DNS system. One important extension to DNS is the ability to handle dynamic updates [37]. In this section, the dynamic update extension to DNS and some of its applications are discussed.

DNS was originally developed with the assumptions that data in the system generally changes very slowly and that IP addresses were statically assigned to hosts. Anytime a host changes its IP address or any other name data, the host's owner will have to notify the zone's administrator. The administrator then manually edits the zone's master file to reflect the new changes and sends a signal to the name server for the server to reread its configuration files.

As network topologies became more variable and computers became more portable and mobile, this scheme of notifying a person and manually editing a file became insufficient [10]. Some mechanism was needed to dynamically update the DNS databases. With this new mechanism, updates to the name space would happen faster, and there would be no element of human error involved with the update process. There was research done into this area of dynamic DNS, and in 1997, the standard update protocol format was decided on.

### **2.4.1 – Dynamic DNS Early Research: PHASE**

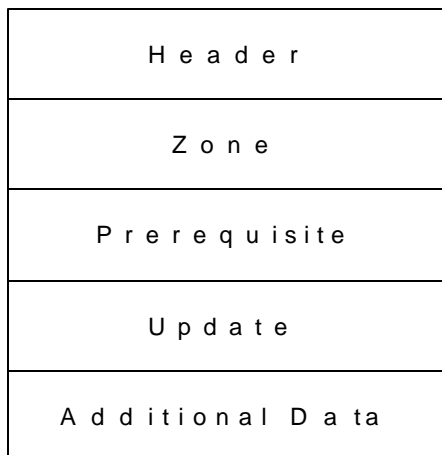
An example of some early research in the area of dynamically updating DNS was the Portable Host Access System Environment (PHASE) [36]. The PHASE system

consists of two components, the Portable Host Access Component (PHAC) and the Dynamic Domain Name Server (DDNS). The PHAC is in charge of providing a temporary IP address to each computer in the system, and the DDNS automatically manages the IP addresses and host name information in the system. The DDNS acts as a conventional domain server in that DDNS maintains name-to-address mappings and provides name-to-address resolution, but is different in that DDNS can handle a special Update query. Whenever a new host joins the system, the PHAC will first assign the host an IP address and then send to the DDNS an update query in the form:

```
LOGIN <Domain Name> <IP Address> <Password>
```

The DDNS stores the name and address information in its database, and the password is included for security purposes. The PHASE system was only designed to handle eight portable hosts, but the ideas presented in the PHASE system are similar to the ones in the current DNS update protocol.

### 2.4.2 – DNS UPDATE Protocol



*Figure 6 – Update Message Format*

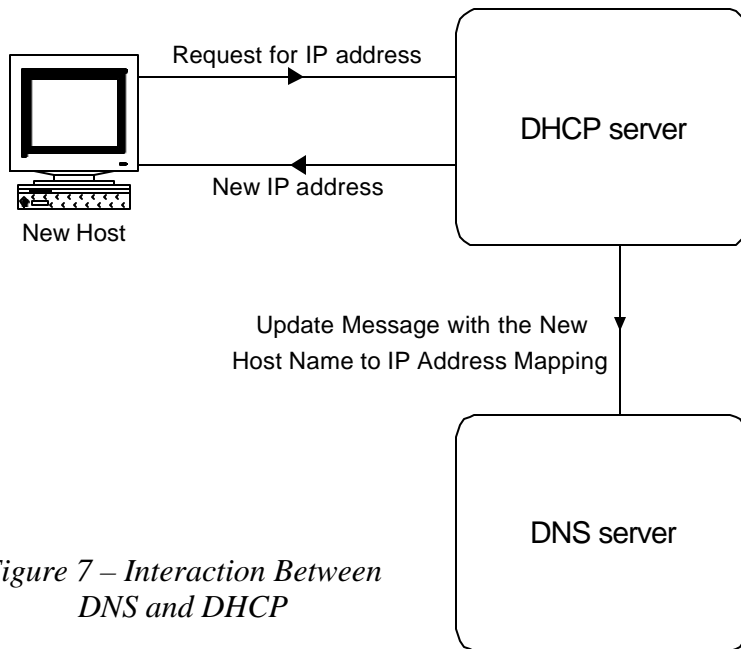
As the frequency of changes to name information in the Internet increased, so did the need for a standard DNS update protocol. In 1997, a standard for the DNS update protocol was agreed upon [37], and implementations of DNS that handle this update message presently exist. On the server side, the name server was expanded to handle an extra update message. The format of the DNS update message is shown in Figure 6. The

header section will specify that the message is an UPDATE and describe the sizes of the rest of the sections. The zone section specifies the zone to be updated. The prerequisite section specifies what resource records have to be present or not present in order for the update to occur. The update section contains the update information for the zone, and the additional data section contains any other data that might also be necessary to complete the update.

For a client to perform a dynamic update, the client sends an update message to an authoritative name server. The name server sees the update message, and if the message has appropriate name data, the name server will add this record into its database. In addition to adding resource records, the update protocol can be used to delete resource records and set prerequisite conditions that allow updates to occur only if the prerequisite conditions are true. The format of the update message is very similar to the format of a regular DNS query, allowing for interoperability between name servers that do and do not support updates and making the process of upgrading of a name server to support DNS updates easy.

### **2.4.3 - Dynamic DNS with DHCP**

DNS with dynamic update capabilities is commonly used with the Dynamic Host Configuration Protocol [9]. DHCP is used to dynamically assign IP addresses to new hosts in the network. Allowing DHCP to dynamically assign IP addresses to hosts can greatly simplify the process of administering a network. Also, DHCP is particularly useful in a mobile computing environment [30] where computers constantly move in the network and need a new IP address for each connection made. DHCP servers work by listening for discover messages from new hosts. Upon receipt of these discover



*Figure 7 – Interaction Between DNS and DHCP*

messages, the DHCP server will offer an IP address for the new host to use. If the new host accepts the IP address, the DHCP server will remember this information and refrain from assigning

this IP address to any other host. The DHCP server, however, will not store which name has been bound to which address. The DNS system is often used for this task of storing name-to-address bindings. By taking advantage of dynamic DNS, DHCP servers can inform the whole network of any new host-to-address mappings by directly sending update messages to DNS servers [Fig. 7]. Once this information is stored in DNS, clients can use resolvers to query the network’s host information.

### 2.4.4 - Secure DNS Updates

The main problem in using DHCP and DNS together is the security issue. Administrators have to take good care to ensure that any updates to the DNS name servers come from trusted entities and that the name servers are able to recognize and discard bogus updates. The solution for the security issue is presented that involves using a set of keys to authenticate an updater [11]. Digital signatures are stored in the DNS as SIG resource records and are used to encrypt and decrypt update messages for a zone. This solution, however, only allows hosts to perform name updates to names that are in

the same zone as the name servers. To attempt to solve this problem, an extension to secure DNS has been proposed that allows hosts to update the database of the DNS servers in different zones [34]. In this extension, two scope controls are attached onto a DNS header to determine which zones can be updated securely. Also, DNS supports an update address access list. The name server can only process update messages from IP addresses on the access list.

### **2.4.5 - Integrity Between Dynamic DNS and DHCP**

In addition to security issues, there are integrity issues between DHCP and DNS. When an individual computer is entering and leaving a network quickly, the DHCP server might end up assigning many different IP addresses to that machine, with only the most recent one being valid. DNS is required to know which address is the most recent one, and old stale data needs to be thrown out of the database. Setting a short time-to-live for a resource record helps to keep stale data to a minimum, and the process of transferring zones from primary to secondary name servers ensure any updates reach all the name servers for the zone.

However, there is still a time period where the DNS system will not have the correct view of the network address assignments. There has been research done into minimizing this time period of inconsistency and improving the integrity between DHCP and DNS. A solution of integrating the DHCP process of assigning IP addresses with the DNS system has been proposed [26]. In this solution, before a DHCP server confirms the assignment of an IP address to the new host, the DHCP server will first update the DNS server with the new address assignment. The DHCP and DNS processes will seem like

one entity, and the DNS server is guaranteed to have the exact same information as the DHCP server.

#### **2.4.6 - Vendor Products Integrating Dynamic DNS and DHCP**

Vendor products exist that provide both DHCP and dynamic DNS in a single software package. These solutions include the following: Meta IP [20], Net ID [23], Shadow IPserver [33], Lucent QIP [19], and Join [15]. To solve and avoid some of the security and integrity issues involved with using DNS and DHCP together, these vendor products closely integrate the functionality of DHCP and DNS by combining the tasks of assigned and storing IP addresses. These products are of varying degrees of complexity in terms of the features that are offered. Features include fault tolerance for the system, load balancing for servers, and enabling remote management. The configuration of these systems is done centrally by an administrator. The administrator will set up the server software according to the needs of the network, and once the initial configuration is complete, the software will handle any new host updates and perform any required name maintenance in the system. These solutions are designed with company intranets in mind and help individual organizations manage their portion of the name space.

As evidenced by the number of vendor products that incorporate DHCP and DNS, using DHCP and DNS together is an effective solution for managing networks. Hosts can freely join and leave a network without worrying about notifying an administrator of changes, and after the initial configurations of the DNS and DHCP servers, administrators can let the server software do much of the work. However, according to Charles Perkins, this combined DHCP and DNS solution is not optimal for mobile computing environments [28]. As mobile computers move, they require new IP

addresses on the fly. Once a new IP address is assigned, the old address becomes stale and should not be used anymore. However, an intermediate name server might have cached the old address information, leading to incorrect name resolutions. Caching could be disallowed, but this increases the amount of network traffic taken up for name resolutions. Also, Perkins argues the security required for DNS updates is tricky to implement.

### **2.4.7 – Solution for Mobile Environments**

For mobile environments, Perkins suggests using the Mobile IP protocol [27] as the mechanism for keeping track of the IP addresses for mobile hosts. Each host in the Mobile IP environment will have a *home address* on its *home network*. On the home network, there will be a *home agent* that maintains a registry of the mobile host's current location. As a mobile host moves from network to network, the host will receive a new *care-of address* and register this new care-of address with its home agent [14]. Thus, the home agent will always know the current location of the mobile host. Other hosts requesting contact with the mobile host will use the home address for communication, and the home agent will redirect the incoming messages to the host's care-of address. This provides simplified communication with the mobile node at all times.

### **2.5 – Using DNS with Other Systems**

Presently, DNS is used as the standard for storing name-to-address mappings and name resolution. The hierarchical and distributed design of DNS allows the system to be scalable, and having replicated name servers allows DNS to be fault-tolerant and ensures availability. These design principles have also been used in other systems, and other systems have modified and extended DNS for their own needs. Attribute-based naming

systems and web server load balancing are examples of systems that have used and modified DNS.

### **2.5.1 - Attribute-base Naming**

The standard way for naming hosts in the Internet is a label that describes where the host is in relation to the root domain. However, there have been other research proposals for the naming of hosts in a network. One popular idea for naming hosts is based on the attributes of a host. For example, the aim of the Profile Naming Service [31] was to provide an attribute-based naming service where hosts would be known by traits like their name, address, or phone. Based on these attributes, the service would know how to contact the host in question.

Another application of the attribute-based naming idea is for resource discovery. With resource discovery, a newly joined network host can learn the location all the network services. The main idea behind resource discovery is that each resource or service has a name, e.g. printer, fileserver, web server, etc., and these resource names are bound to the host names that provide these resources. A design has been proposed for a resource discovery protocol using URNs, or Uniform Resource Names [29]. URNs describe resources, and hosts requesting a resource will query a name server using the URN and receive from the name server a usable host name representation. From here, the host can use the new name to contact the requested resource. For example, a person who wants to know how to access the IBM research web page will send a URN query in the form:

```
n2c://ibm/http/research/homepage
```

The name server handling URN requests will see the URN and return a response that will a response in the form of a URL:

<http://www.research.ibm.com>

The host can then use the DNS system to resolve the name given in the URL and access the desired web page.

Another more recent proposal for an attribute-based naming system is the Intentional Naming System (INS) [1]. INS, which uses an intentional name language based on a hierarchy of attributes and values for its names, is a resource discovery and service location system for dynamic and mobile networks of devices and computers. INS has its own set of name servers and resolvers working to store and lookup attributes. The name servers have the capability to send updates amongst themselves, and the resolvers are able to self-configure their topology depending on the changing needs of the network. Services will periodically contact the name servers to update the name servers of any new service information, and hosts use the intentional naming language to contact the services. Even though the INS system has all the components of the DNS system, INS is intended to complement, not replace the Internet DNS. INS maps service name-attributes to name records, and DNS maps the name records into IP addresses.

### **2.5.2 - Web Servers**

DNS has also been used to distribute load among web servers. For popular web sites to support high request rates, replication of information among multiple Web servers is necessary. One way to implement this replication is to organize the multiple web servers in a cluster. In this arrangement, there is a virtual single interface to all the web servers, and the request load is balanced among all the servers. The role of DNS in a web

cluster is handling the storage of all the web servers' IP addresses. Each web server will be known by a single domain name (the name of the web site), and DNS will store each web server's IP address under that web site name. When the name server is queried for an address for that web site name, the name server will respond with an IP address from the list of cluster's web server addresses.

How the DNS name server chooses which IP address to use has been the subject of much research. Optimizing this process of choosing an IP address will help achieve load balancing among all the web servers, and thus allow efficient processing of all web requests to the cluster. A simple approach to solve the load-balancing problem is to use a round-robin strategy where the IP addresses returned is the next one on the list of a simple rotation [17]. So, 1/Nth of the DNS requests get each of the N different IP addresses. However, this round-robin scheme might not work because a part of the network might be particularly congested or a web server might be experiencing problems. Another approach is for the DNS server to compute the network round-trip time between the different web servers and to return the IP address of the server with the minimum round-trip time [8]. An even more complicated approach is to remember state information like requester history and cache time-to-live information in order to make a more informed decision on which web server to choose [6]. Also, an integrated solution using DNS and the HTTP protocol together has been proposed to redirect web requests to the correct server [7].

## **2.6 - Administrator-Less DNS**

To this point, much of the functionality of the Domain Name System (DNS) has been explained. In addition to storing name-to-address bindings and providing name

resolution, DNS can be extended to handle dynamic updates, assist in service discovery, and provide load balancing to servers. In all of these cases, an administrator has manually configured DNS to perform its task. The administrator knows which domain he is authoritative over, controls the placement of the name servers, and decides what extensions and additional features a name server can handle. Also, an administrator is available to correct any errors or failures that might occur in the system. The idea of an administrator-free DNS has been proposed in research in different forms. A program called *nssetup* [12] that automates some common DNS configuration steps has been implemented, and an expert system and neural network have been used to help diagnose common DNS errors [24]. Also, the ZEROCONF project [38] is in the process of standardizing protocols that would allow networking in the absence of configuration and administration.

### **2.6.1 – *nssetup***

Researchers in Japan have been trying to tackle network problems by using the Zero Internet Administration approach [12]. This approach reduces the work of a network administrator by simplifying configuration tasks and eliminating repetitive tasks through automation. The approach was applied to the case of DNS, and the result was a program with a graphical interface that helped DNS administrators perform their job. Their tool, called *nssetup*, could generate a DNS database file from the machine's host file, keep the root cache file up-to-date, and maintain the reverse address lookup table. These were all repetitive tasks that could be automated. To check the correctness of the configuration, *nssetup* contained a feature that checked if the name server was up and

running. In addition, *nssetup* provided a graphical user interface for configuring the resolver and adding new hosts into the database.

The *nssetup* program was used in an experiment to compare how fast a person could configure a name server using *nssetup* and not using *nssetup*. The results showed that it was considerably faster to configure a name server using *nssetup* than not using it. Someone with minimal experience in network management could configure a name server in three minutes. With this program, the task of configuring and updating DNS is much less time-consuming and less error-prone.

### **2.6.2 - DNS Administration using Artificial Intelligence**

The *nssetup* tool helps to configure DNS, but the tool does not give any support on what to do if an error occurred in the DNS system. An artificial intelligence system that will detect errors that occur in DNS, diagnose the error, and fix the error has been proposed in research [24]. This system consists of a neural network module and a rule-based system for monitoring and diagnosing problems that occur in the DNS system. Rule-based systems are useful for autonomous network management problems because error and solution patterns are naturally expressed in IF-THEN rules and network characteristics can easily be captured in a knowledge base. Also, as network patterns change, it is possible to add or remove a rule from the rule-base without changing the overall structure of the program or the control flow [18]. Since it is difficult to model some of the errors in DNS, it is useful to use neural networks to learn any errors that are not in the rule-base already.

This artificial intelligence system works by reading the log file of a name server running BIND. Any error information in the log file is passed along to the neural

network and expert system. If the expert system does not know of the error, then the neural network learns the error. If the expert system does know of the error, then the expert system will fire its set of rules and output a report of the cause of the problem and the name server that created the problem. Also, suggestions for the correction of the error are given. This information can be passed along to the administrator and facilitate in the process of returning the DNS system into a working state. This system was tested with real data DNS log files, and the system was able to recognize over 99 percent of errors. This error information would be very useful for administrators to diagnose and fix errors in DNS.

### **2.6.3 - ZEROCONF – Zero Configuration Networking**

The ZEROCONF Working Group of the Internet Engineering Task Force (IETF) is currently working on standardizing protocols that would allow networking in the absence of configuration and administration [38]. Using the zeroconf protocol, computers could join together and immediately begin performing basic network functions without any human configuration. Also, the network will continue to exist and be able to add or subtract computers with any human administration. To achieve this goal, the ZEROCONF working group has laid out four main areas of work [13]:

1. IP host configuration
2. Domain name to IP address resolution
3. IP multicast address allocation
4. Service discovery

Of particular relevance to the topic of this thesis is the name-to-address resolution portion. The zeroconf requirements state that domain name to IP address resolution must

be able to occur even though a host is not configured with the IP address of a DNS server. There must be some scheme, existing across all the zeroconf computers, that will store zeroconf name-to-address mappings and provide resolution until a DNS server is able to be located. Once a DNS name server is located and contacted, the zeroconf computers from then on will use the DNS name server for name-to-address resolutions and updates [13].

In addition to issues involved with name resolution and detecting DNS servers, there is the issue of assigning names to the zeroconf computers. The zeroconf computers are required to create or determine the existence of a default domain name to use. Once that is determined, the computer must choose a name, verify that the name is unique within the default domain, and be able to choose another name if it is duplicate. All hosts operating within a domain must use the same default name, and sub-domain delegation is not done within a zeroconf zone.

These requirements of the name-to-address resolution portion of zeroconf project are still subject to change since the zeroconf protocol is still in the developmental stages, but outlining these requirements give a good idea of the challenges involved in the field of self-configuring and self-administering networks.

## **2.7 – NetBIOS**

NetBIOS was designed for use by a group of personal computers, sharing a broadcast medium. The computers can communicate using either the UDP or TCP protocols, and computers using the NetBIOS protocol are identified by names that are assigned dynamically in a distributed fashion. NetBIOS applications employ NetBIOS mechanisms to locate resources, establish connections, send and receive data with an

application peer, and terminate connections [32]. The naming scheme for NetBIOS resources is of particular interest.

NetBIOS resources are referenced by name, and an application, representing a resource, will register one or more names that it wishes to use. The name space is flat and uses sixteen alphanumeric characters. For an application to use a name, the application has to go through a process of registration. During registration, a bid is placed for a name. The bid may be for exclusive (unique) or shared (group) ownership. All applications go through this bidding process in real time, and implicit permission is granted when no objections are raised. This means that if no objects are received after waiting a period of time after a bid is made, the application is free to use the name. Applications are allowed to call the name service primitives Add Name, Add Group Name, and Delete Name.

The NetBIOS system has three types of end-nodes that support NetBIOS service interfaces and contain applications. The three types are broadcast (“B”) nodes, point-to-point (“P”) nodes, and mixed mode (“M”) nodes. Also, there is a NetBIOS Name Server (NBNS) included in the system. The NBNS is flexible in that it can simply act as a bulletin board on which name/address information can be freely posted, or the NBNS can assume full control of the management and validation of names [32]. The three types of nodes work with the NBNS to provide an accurate view of the names in the system. Before a name can be used in the system, the name has to be registered to make sure there are no conflicts. The B nodes perform name registration by broadcasting claim requests and solicit defenses from any node already holding the name. The P nodes perform name registration by contacting and asking the NBNS, and the M nodes will first

broadcast claim requests and then contact the NBNS. Once the name is registered, either the B nodes or the NBNS will have knowledge of the new name.

The registration process is just one part of the NetBIOS name system. There also has to be a way for nodes to find the IP address of a node given the name of the node. To perform name resolutions in the NetBIOS system, the different end-nodes use different techniques. The B nodes solicit name information by broadcasting a request. The P nodes ask the NBNS, and the M nodes first broadcast a request, and if the desired information is not obtained, the M node asks the NBNS. Names also have to be released or deleted from the name system if a node fails or is turned off. To release names, B nodes broadcast a notice, the P nodes send notification to their NBNS, and M nodes both broadcast a notice and inform their NBNS. The basic functions of registering, resolving, and deleting names form the foundation of the NetBIOS name system.

## **2.8 – AppleTalk**

The AppleTalk network system allows a set of computers to form a network and be able to communicate with each other and network devices. AppleTalk was designed to allow users to exchange and share information on an application level. Even if the lower level link and transport layers on machines are different, the machines can use the AppleTalk protocol to communicate. The architecture of the AppleTalk system has a decentralized design, and AppleTalk adopts a “plug-and-play” capability, wherein the user can plug a computing device into a network system and use it immediately, without any of the complications of configuration [3]. The AppleTalk system has a naming protocol called the Name Binding Protocol (NBP) that provides a way of converting a name into the address required by transport protocols. Also, the AppleTalk system has a

Zone Information Protocol (ZIP) that helps to group and identify a collection of resources.

Network-visible entities in the AppleTalk system are assigned entity names. Entity names have the format: object:type@zone. Each entity has a type associated with it, and entities belong in zones. The AppleTalk system handles the binding of these entity names with Internet addresses, and each node in the system will maintain a names table containing name-to-entity Internet address mappings. The names directory (ND) is the distributed database that stores all the name-to-address mappings in the system. This directory can be distributed among any number of machines. NBP does not require the use of name servers. However, its design allows the use of name servers if they are available [35].

The name binding protocol provides four basic services:

- name registration
- name deletion
- name lookup
- name confirmation

The name registration process first involves inserting a new name of a node into the node's names tables, and then the NBP process enters the new name-to-address mapping into the ND. Name deletion means deleting a name from the ND, and name confirmation checks the validity of a binding. For name lookup, NBP is used to perform a search through the ND for the named entity. On a single network, a name lookup packet is broadcast over the network asking for the information for a name.

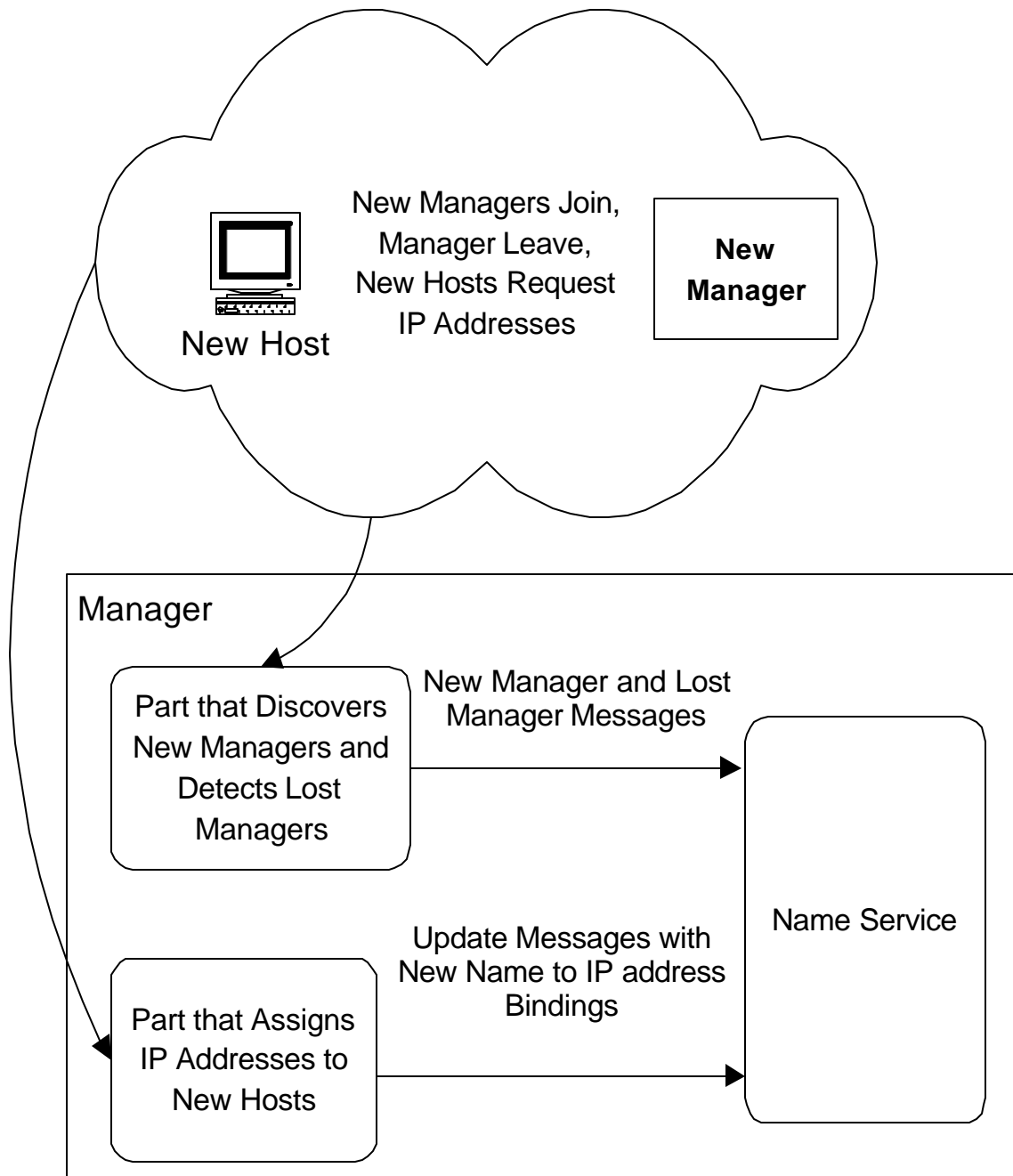
To perform name lookup packet over a network with multiple zones, the Zone Information Protocol (ZIP) is used. ZIP uses a data structure called the zone information table (ZIT) to store which computers are associated with each zone. These tables reside on each router together with the routing table. Each AppleTalk zone consists of a collection of AppleTalk networks, and a particular AppleTalk network belongs to exactly one zone [3]. NBP will use the ZIP protocol to find out where the network is for a zone. Once the network is located, the name lookup packet is broadcast over that network asking for name information. If that name information is found, NBP will return the name information to the original sender. The Name Binding Protocol (NBP) and the Zone Information Protocol (ZIP) are used in the AppleTalk network system to provide name registration and name lookup capabilities.

## 3 – Implementation

In this section, an implementation of a system that can store Internet name-to-address mappings and provide name resolution is presented. Virtually no configuration is required to set up this name system, and no administration is needed to maintain the system. This name system consists of a set of computers running name services that communicate with each other to distribute name information. These name services converse with each other using a protocol designed for this system, and the name services use the standard Internet DNS protocol to answer queries and handle updates from hosts. This name system works within in the framework of a larger network system that provides full network capabilities to a community of computers. Before describing the name system in depth, it would be useful to describe the full network system and the role of the name system within the full network system.

### 3.1 - Background

For the past year and a half, there has been an undertaking at the Research Corporation to devise a way to form a network without any human configuration and for the network to maintain itself without any human administration. Ideally, someone could connect a set of computers together, and within a short time, the computers will discover each other and form a network. Also, if any new computers join the system or if any computers leave the system, the other computers will be able to reconfigure themselves accordingly. Currently, the strategy for implementing this network system is to have a backbone of computers that provide network services and to have the hosts connect to the backbone. The computers making up the backbone of the network system are called *managers*, and they are the machines that will discover other computers, provide packet



*Figure 8 – Information from Manager forwarded to Name Service*

routing, dynamically assign IP addresses to new hosts, keep a database of the hosts in the system, and provide name-to-address resolution.

Thus, the name system will be a subset of the whole network system. A name service will reside on each manager [Fig.1], and the sum of the name services will form

the name system. In order for the name services to know of each other and the state of the network, the other features of the network will be used. Every time a manager discovers another manager, a message will be sent to the name service informing the name service of the new manager [Fig. 8]. Also, every time a manager detects a missing manager, a lost manager message will be sent to the name service. This way, each name service will know the location of every other name service in the network.

Another manager feature that the name system uses is the dynamic assignment of IP address to hosts [Fig. 8]. When a new host connects to the backbone of the network, one of the managers will assign an IP address to the new host. That host name and IP address binding has to be forwarded to the name service residing on the manager. This name information will take the standard DNS update format, and upon receipt, the name service will store the new name-to-address mapping in its database. Requiring update messages to be sent ensures that the name system will know about every host in the network.

## **3.2 - Specifications/Requirements**

With the role of the name system clearly defined in the context of the whole network system, the requirements of the name system can be outlined. At the very minimum, the name system must be able to store name-to-address mappings and provide name-to-address resolution for the manager network system. Also, the name system should be able to handle dynamic updates. The formats of the domain names, addresses, queries, and updates used in the name system will be the same as the formats used for the Domain Name System (DNS). In order to support an arbitrary number of hosts and managers, the name system should be scalable. To ensure availability in case a name

server fails, the name system should be fault-tolerant. Finally, the name system should require no human configuration and no human administration.

### **3.3 - Solution/Approach**

To meet the requirements of the name system, an existing solution is incorporated with a new program. The name service runs on the UNIX operating system and consists of two processes running on each manager in the name system. One process is the Berkeley Internet Name Domain (BIND) program. BIND is the industry-standard implementation for DNS on UNIX and runs on many Internet name servers. The other process is an agent program that changes BIND's configuration files in response to discover messages and new update messages from the network. Since BIND is the standard implementation for DNS, BIND is able to speak in standard DNS formats. The most recent version of BIND, version 8.2, can handle dynamic updates. Since the DNS is designed to be scalable and fault-tolerant, using BIND makes the name system scalable and fault-tolerant. The agent program will handle the configuration and administration of BIND.

#### **3.3.1 – Name Service Interface**

Fig. 9 shows the interaction between agent program, BIND and other manager processes. The agent program uses Berkeley UDP sockets to listen to and send two different types of messages. The first message type, the agent message format, is primarily used to communicate between agents on different managers. Also, the discovery component of the manager uses the agent message format to send notification of new and lost managers to the agent program. Port number 54 is used to send and receive these agent messages. The second message type is the standard DNS message

format. The agent program acts as a filter for DNS messages and listens on port number 53, the standard DNS port, for the DNS messages. All DNS queries and updates will come in on this DNS port, and agent program decides what to do with these DNS messages. If the agent decides that the BIND process running on the same manager can service the DNS request, the agent will forward the DNS message to the BIND process. The BIND process will see the DNS request and respond to the request, and the response message will eventually be returned back to the original requester. If the agent decides that the BIND process cannot service the DNS request, the agent program will redirect the DNS message to another name service's agent. The BIND process will listen on port number 55, and only agent programs can send messages to this port.

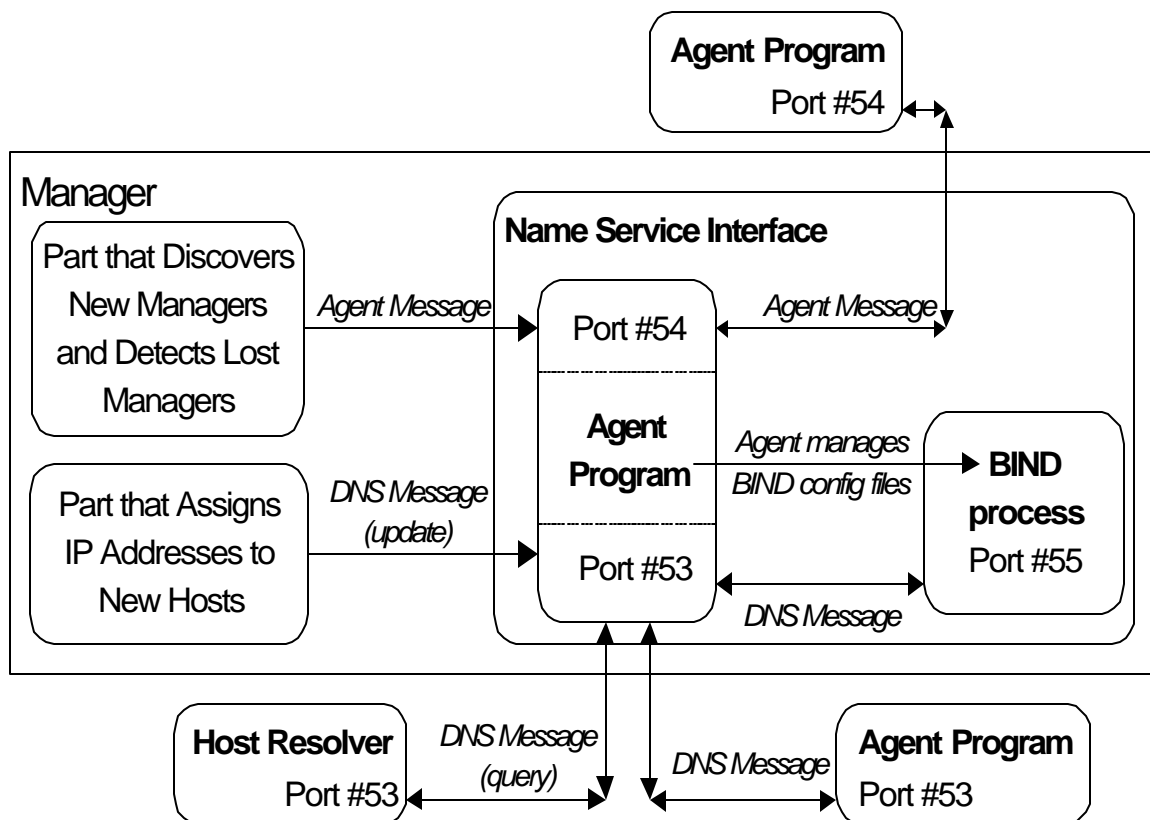


Figure 9 – Interaction Between Agent, BIND Process, and Other Processes

### 3.3.2 – Controlling the BIND Process

To start the BIND process, the command *named* is run. Before *named* can be run, one configuration file, *named.conf*, and a set of database files have to be configured. The *named.conf* file lists all the zones the name server is authoritative for and whether the name server is a slave or master for that zone. A sample *named.conf* file is shown in the Appendix. A database file contains all the name information for a zone, and there will be a separate database file for each zone in the *named.conf* file. If the *named.conf* file or the database files are changed in any way, a SIGHUP signal can be sent to the *named* process. This signal tells the *named* process to reread its configuration files and database files.

The agent program controls BIND through the use of the *named.conf* file and database files [Fig. 9]. During start-up, the agent program will use a Perl script to generate a *named.conf* file and the appropriate database files. Once these files are generated, the agent program will automatically start the *named* process. BIND is then ready to answer queries and handle updates to the name database. Once the *named* process has started, it might be the possibility that the name service will need to handle another zone. The agent program will append that zone information to the *named.conf* file, create the appropriate database file, and send a SIGHUP signal to the *named* process. With the agent program in control, BIND will always be configured properly to store names and handle all the network name resolutions.

### 3.3.3 – Internal Root Servers

Presently, the name system is designed to only handle the host names and addresses in the manager network system. There are no requirements that the managers

or any of the hosts must be able to connect with the Internet and outside world. With this in mind, the name system is implemented using the idea of internal root servers. Certain name services in the manager network act as the root servers and are authoritative for the root domain [Fig. 10]. These internal root servers store the root level names and keep track of where top-level domain name information is located. With this information, the root servers will know how to resolve every name in the network. So, as long as a name service knows the location of at least one root server, the name service will be able to provide name resolution for the network's name space. Also, with internal root servers, there are no restrictions on what name each host can be assigned to, and there is automatic registration for names.

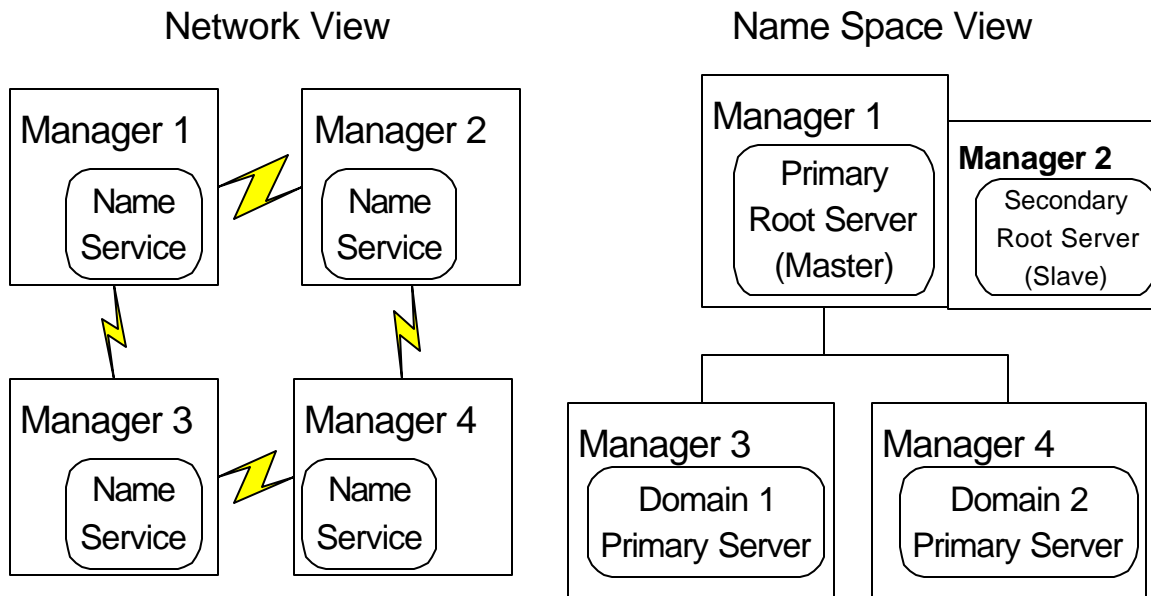


Figure 10 – Sample Set-up of Name Services

### 3.3.4 – Initial Configuration

For a name service to function in the name system, the name service will either have to be a root server or know about other root servers. This idea forms the foundation

of the name system, and the name service's goal during the initial configuration is to learn about the root servers in the system. The initial configuration for a name server starts when a new manager is placed in the network system. The agent will receive discovery notifications from the manager. Once the agent program collects all the information on the other managers in the network, the agent will decide how to configure BIND.

If the agent program does not receive any notifications about any other managers, then that means there are no other managers in the network. So, the name service is the only name service in the system, and the agent will configure the BIND process to run as an internal root server for the network.

If the agent program does receive notifications from other services, then the agent will ask the other existing agents for their root server information. The agent program will then use this root server information to configure the BIND process.

### **3.3.5 – DNS Messages**

Once the name services are configured with root servers, the name services are ready to handle DNS queries and updates [Fig. 9]. The agent program will directly listen to any DNS messages that come in to the name server. If the message is a query, the agent forwards the query to the BIND process. The BIND process will answer the query and return the answer to the agent. The agent will then send the answer back to the original sender of the query. If the message is an update, the agent program will try to find the name service in the system that can handle the update. If a name service is found, the agent will redirect the update message to the correct name service. If a name service is not found, then the agent will change the local *named.conf* so that the agent's

BIND process is able to handle the update message. Once the reconfiguration of the BIND process is complete, the agent will directly send the DNS update message to the reconfigured BIND process.

### **3.3.6 – Agent Messages**

In addition to handling DNS messages, the agent program will listen to messages from other agents and managers. The most important of these messages are the *discover* and *leave* messages. *Discover* messages notify the agent of any new managers, and thus any new name services, that join the network. *Leave* messages notify the agent of any lost managers. These messages allow the agent to have a current view of the network topology and warn the agent of any conflicts or errors in the network.

A conflict might occur if a configured name service's agent hears a *discover* message informing the agent that an already configured server has joined the network. In this network joining situation, there is the possibility both services are primary masters for the same zone. Only one primary master per zone is allowed, and it is the agent program's job to detect this type of conflict and negotiate a new master for the zone. The *negotiatemaster* and *forceslave* agent messages are used for conflict resolution.

A DNS error may occur when managers leave the network. In this network partition situation, there is the possibility that a zone's slave server will be missing its master server. A primary master server is required for a zone to handle updates, and depending on the server IDs of each manager, the agents in the network will elect a new master for the zone, and notify any other slaves of the new master. The *becomeslave* agent message is used in this case.

In addition to the DNS messages mentioned above, the *getroot* message will be used by an agent to share the root server information among all the agents.

### **3.3.7 – Secondary Name Servers for Zones**

Besides listening and reacting to DNS and agent messages, the agent program will assign secondary (slave) servers for each zone the agent's name service is the primary authoritative master for. If the BIND process is a primary master for a zone, the agent program will use the *becomeslave* agent message to periodically ask other name services to be the slave for that zone. The agent program asks other services until there are enough slave servers for that zone. In the DNS system, a name service can be a master or slave for an arbitrary number of zones, and so there is no restriction on which name services can be authoritative for a certain zone.

After the initial configuration process, the main tasks of the DNS agent program are to assign secondary name services for zones and to listen and respond to agent and DNS message. The details of each task and the conditions for performing these tasks are discussed in the subsequent sections. The states the agent program keeps, the messages the agent sends, and a detailed description of all the possible scenarios the agent program can be in will be discussed.

## **3.4 – State Information**

To help perform the agent tasks, the agent program stores some useful state information. The agent program will store the information for name services in the network, store zone information, maintain a message log, and other miscellaneous information.

### 3.4.1 – Name Service Information

The agent program stores information about all the name services in the system. The agent obtains this information through *discover* messages from the manager and by asking other agents in the system. The information stored for an individual name service is shown in Table 1.

Table 1 – Server Record Fields

Server Record Field	Description
Name	The name of the manager the name service resides on.
IP Address	The IP address of the manager the name service resides on.
Server ID	The unique ID of the manager the name service resides on. The unique ID will be the MAC address of the network interface used by the name service.
Status Flag	A status flag saying whether or not the name service has configured its root servers. If the name service has configured its root servers, then the status flag will have the value: <i>srv_rs_configured</i> . If not, then the status flag will have the value: <i>srv_start_up</i> .

These four fields make up a *server record*, and this name service information comes from the manager the name service resides on. For usability, the agent program stores the network's name service information in four separate groups of server records. The agent program uses these four server groups to quickly look up information about the

other servers in the network and to exchange information with other agent programs. The four groups can have overlapping server records and are shown in Table 2.

*Table 2 – Types of Server Record Lists*

<b>Server Record List Type</b>	<b>Description</b>
Own Servers	This group of server records contains the agent program's own server information.
Known Servers	This group of server records contains the server information for all the other name servers in the network.
Root Servers	This group of server records contains the server information for all the root servers in the network. This information is useful to have when configuring BIND and when sharing root server information among agent programs.
Newly Discovered Servers	This group of server records contains server information for all the name servers that have just been discovered by the discovery portion of the manager. Every time a new name server enters the system, the agent will be notified, and it is here that the new server information is stored. Once the agent processes this newly discovered information, the server record will be deleted from the newly discovered servers and added onto the Known Servers.

### **3.4.2 – Zone Information**

The agent program controls which zones the name service is authoritative for, and this name service zone information is stored as a data structure in the agent program. The following information about the zone will be stored:

1. **Zone Name** – The name of the zone the name service is authoritative for.
2. **Master/Slave Flag** - Whether the name service is a master or slave for the zone

- a. If the name service is a *master*, the agent will also store:
    - i. **Slave Information** – The server record of the slaves for the zone.
    - ii. **Number of Slaves** – The number of slaves that exist for the zone.
    - iii. **Number Slaves Required** – The number of slaves required for the zone.
    - iv. **Need More Slaves** – Boolean flag saying whether or not any more slaves are needed for the zone.
  - b. If the name service is a *slave*, the agent will also store:
    - i. **Master Information** – The server record of the primary master for the zone
3. If available, the **Parent Zone Name** and **IP address** of the parent zone's name service.

The redundancy of the slave number information is not necessary but makes it easier for the agent to recognize whether or not more slaves are needed. The parent zone information is needed for informing a parent zone name server of any new replicas in the system. Informing the parent zone name server ensures proper delegation in the system.

A name service can become authoritative for a zone either by receiving a DNS update message for a zone that does not exist or by receiving an agent message asking the service to become a slave for a zone. The agent program will update this zone information data structure whenever the name server becomes authoritative for a zone. The agent program uses this zone data structure to generate and update the *named.conf* configuration file and the database files used by BIND.

### **3.4.3 – Message Logs**

The agent program maintains a set of message logs. Each agent message and DNS message contain a message ID, and the message logs store this message ID and the IP address of the host the message came from.

There are message logs that keep track of all the agent messages that have been sent and responded to. Keeping a message log provides duplicate response message detection and ensures no tasks are unnecessarily repeated.

For DNS messages, there is a message log that stores the IP address of where the message came from. The agent program acts as a filter for DNS messages and can redirect DNS messages to any BIND process. When the BIND process returns a response message to the agent, the agent uses the message log to find where the original request came from and forwards the DNS response to the original sender.

### **3.4.4 – Miscellaneous**

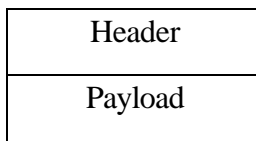
The agent program also has an event list. The agent uses this event list to schedule events that should be done after waiting a certain amount of time. The agent can also schedule periodic events. This event list is used by the agent to schedule the checking of newly discovered servers and the periodic event of finding slaves for the name server's master zones.

The agent program also has a timeout list that stores agent messages. There are certain agent messages that require a response, and if a response to an agent message has not been received within a timeout interval, the timeout list will resend the agent message. This resending continues until a response is received or the message dies. This

timeout list is useful in case there is a temporary network failure that prevents the delivery of a message.

### 3.5 – Agent Messages

To communicate with other agent programs and the managers, the agent program uses a set of agent messages. The agent message has a very simple format with a header section and a payload section [Fig. 11].



*Figure 11 – Agent Message Format*

*Table 3 – Header Fields*

<b>Header Field</b>	<b>Description</b>
Message ID	The unique message ID for the message.
Opcode	The operation code for the message.
Send/Response Flag	Flag saying whether the message is send request or a response. This flag can take on two values: “Send” or “Response”

The header section contains three fields delimited by a semicolon [Table 3]. The three fields are the Message Id, Opcode/Type of the Message, and a flag saying whether the message is a Send or Response message. The payload will hold the data for the message, and the format of the payload is different for each possible Opcode.

The Opcode for the agent message can be six different values [Table 4], and an overview of the possible Opcodes is given. The exact format of each agent message is shown in the Appendix.

Table 4 – Possible Opcodes

Opcode	Quick Description
<i>discover</i>	Informs agent of new managers/name services.
<i>leave</i>	Informs agent of lost managers/name services.
<i>getroot</i>	Used to share root server information across the network.
<i>becomeslave</i>	Agent sends to another agent to <i>make</i> the other agent a slave server.
<i>negotiatemaster</i>	Used to resolve master conflicts.
<i>forceslave</i>	Agent sends to another agent to <i>force</i> the other agent to be a slave server.

### 3.5.1 - *discover*

*Discover* messages are sent by the manager to the agent program informing the agent of newly discovered managers in the network. Since name services are located the managers, the agent will know of any new name services in the system. The payload of the *discover* message consists of the server records of any new managers in the system. The format of the server record is explained in Section 3.4.1.

When the *discover* message is received, the agent will add the payload information onto the Newly Discovered Servers list and call the CheckDiscovers function. The CheckDiscovers function checks through the newly discovered name services and decides, depending on the state of the name service and the state of the newly discovered name services, if any configurations should be changed. The conditions and results of these reconfigurations are described in detail in the next section.

### 3.5.2 - *leave*

*Leave* messages are sent by the manager to the agent program informing the agent of any lost managers in the network. Since name servers are located the managers, the

agent will know of any lost name services in the system. The payload of the *leave* message consists of the server records of any lost name services in the system.

The agent program will be able to reconfigure the name server if the lost name server is somehow related. For example, if the lost name server was the primary master for a zone that the agent was a slave for, the agent will be able to find a new master for the zone. The conditions and results of these reconfigurations are described in detail in the next section.

### **3.5.3 - *getroot***

*Getroot* messages are sent from one agent program to another agent program requesting the root server information. The *getroot* message is primarily used during the start-up process and when a manager joins the network. For a name server just starting up, requesting and receiving the root server information from another name service allows the starting-up name service to reach a configured state and to start handling name resolutions. Obtaining another server's root server information also helps to resolve any zone conflicts. The use of root server information is explained in more detail in the next section.

The payload of the *getroot* message takes on two forms depending on whether the message is a Send or Response message. With the Send flag, the payload of the *getroot* message will be empty. The sender of the *getroot* message is requesting information, and so the payload does not require any data. With the Response flag, the payload of the *getroot* message will be the server records for the root server information. When an agent receives a *getroot* message with the Send flag, the agent will place its root server information in the payload and send a *getroot* message with the Response flag back to the

sender. The original agent sender can then use the root server information to configure the name server.

### **3.5.4 - *becomeslave***

This message is sent by an agent program whose name service is the primary master for a zone, and the name service needs a slave for that zone. Having another name service act as a slave creates another copy of the zone information on the network, making the name system replicated and more fault-tolerant.

To make another name service become a slave for a zone, the agent program will send a *becomeslave* message with the Send flag to the agent residing on the other name service. The payload of this message will have the zone name, the server record for the agent's manager, and parent zone information if any is available. The receiving agent sees this message and will configure the BIND process to be a slave for the zone name and use the included server record as the master server record for the zone. Also, the receiving agent will store the parent zone information. After completing these tasks, the receiving agent will send back a *becomeslave* message with the Response flag and its own server record included in the payload. When the sending agent sees this response message, the sending agent will add the included server record into its slave information.

### **3.5.5 - *negotiateaster***

This message is sent when an agent program detects a zone conflict. When two configured name services join, it is possible that they both are primary masters for the same zone. Having two primary masters is not allowed in the DNS system, so the two name services will have to negotiate a single master and merge their information

together. The agent decides which name service remains the primary master using the *negotiatemaster* message.

When an agent detects a master zone conflict, the agent will send a *negotiatemaster* message to the agent on the conflicting name service. The payload of the *negotiatemaster* message includes the conflicting zone name, how many slaves for the zone the sender has, and the sender's own server record. When the receiving agent receives the message, the receiving agent will respond with a *negotiatemaster* message with a payload that includes the conflicting zone name, the number of slaves for the zone the receiving agent has, and the receiving agent's server record.

Both agents will compare the number of slaves they have for the zone with the number of slaves the other agent has. If the agent sees that it has more slaves than the conflicting agent, then, the agent will declare itself to be the master for the zone. If the agent has fewer slaves, then the agent will not do anything. If the two agents have the same number of slaves, then the agents will compare the server ID of the conflicting manager with the server ID of its own manager. If the agent sees that it has a smaller ID than the conflicting server's ID, then the agent will not do anything. If the agent sees that it has a larger ID, then the agent will declare itself to be the master for the zone.

If the agent has declared itself the master, the agent will merge the conflicting name server's zone information with its own and then send a *forceslave* message to the agent on the conflicting server. The zone information is merged by manually requesting a zone transfer and then sending update messages with the zone transfer data to the remaining master. The *forceslave* message forces the conflicting master to be a slave for the zone.

### **3.5.6 - *forceslave***

This message is sent by an agent after using the *negotiatemaster* message to resolve a master server conflict. After deciding that it should remain a zone's master server, an agent will send a *forceslave* message with the Send flag to the conflicting agent. The payload of the message includes the zone name, the server record of the agent's manager, and parent zone information if any is available. The receiving agent sees this message and will configure the BIND process to be a slave for the zone name and use the included server record as the master server record for the zone. Also, the receiving agent will store the parent zone information. The new slave will then send back a *forceslave* message with the Response flag. The payload of this response message will have the server records of all the slaves the name server used to be the master to. The sending agent will add these server records into its slave information and send the *becomeslave* message to all these new slaves. Calling the *becomeslave* message informs the slaves that they have a new master.

## **3.6 – Scenarios**

Now that an overview has been given on the possible agent messages that can be sent by the agents, the scenarios where these agent messages are used can be explained. The agent can be in one of two states, the start-up state and the configured state. The state of agent is stored in the status flag of the *server record* information for the agent. When a new manager joins a network, an agent will be in the start-up state. As the agent learns about other name services, it will enter a configured state. During the configured state, new name services may join the network, or name services may leave the network. The agent is able to react to these changes by reconfiguring the BIND process. The

section will give a detailed explanation of the possible things that can happen during the start-up and configured states.

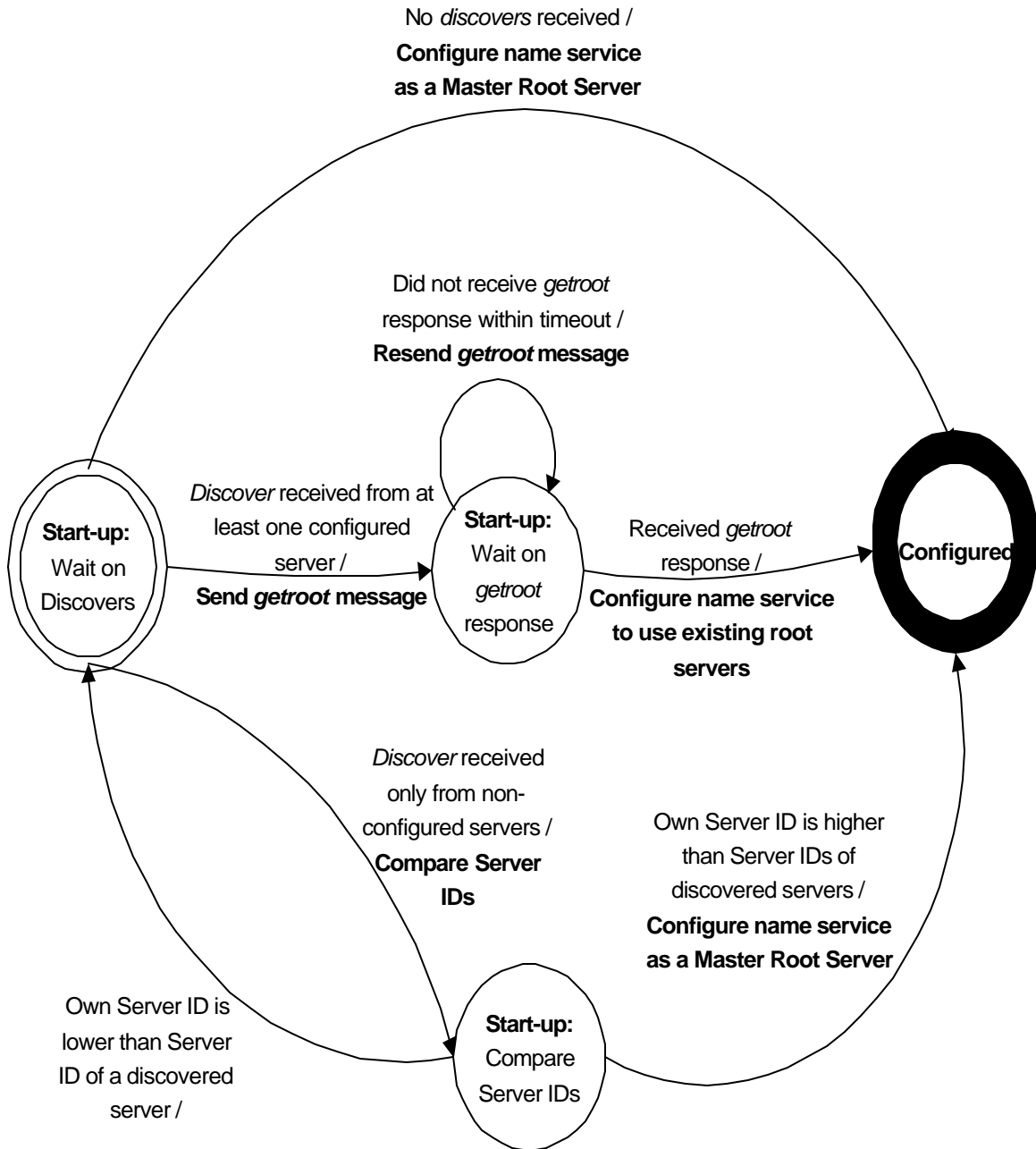


Figure 12 – States in the Start-up Scenario

### 3.6.1 - Start-up

The start-up stage is when a new manager joins the network system. During this stage, the new manager will discover what other managers exist in the network. For each manager that is discovered in the network, the agent will receive a *discover* message. As explained Section 3.5.1, the *discover* message includes the server record of any discovered managers. Since name services reside on the managers, the agent will know of all the other name services in the network. There are three possible ways the agent program will act depending what is discovered. These three ways are shown in Fig. 12 and described in detail in the subsections.

#### 3.6.1.1 - No Name Services Discovered

If the agent program did not receive any *discover* messages, then that means the agent's name service is the first and only one in the network. The agent will configure its name service as an internal root server for the network [Fig. 13].

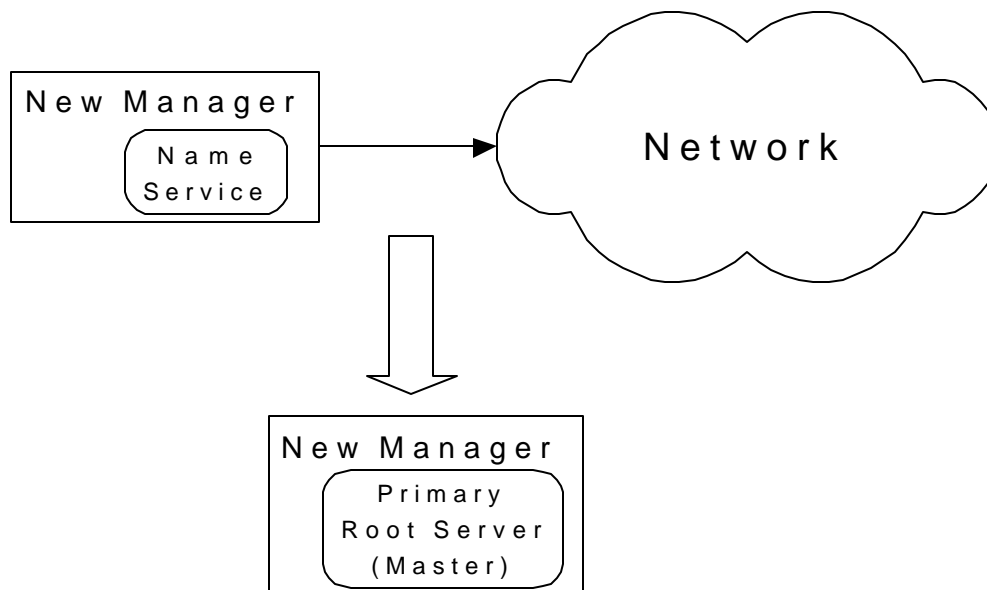


Figure 13 – New Manager is First to Enter Network

To do this, the agent configures the BIND process to be the primary master for the root domain by adding this statement into the *named.conf* file:

```
zone "." in {  
    type master;  
    file "db.root";  
};
```

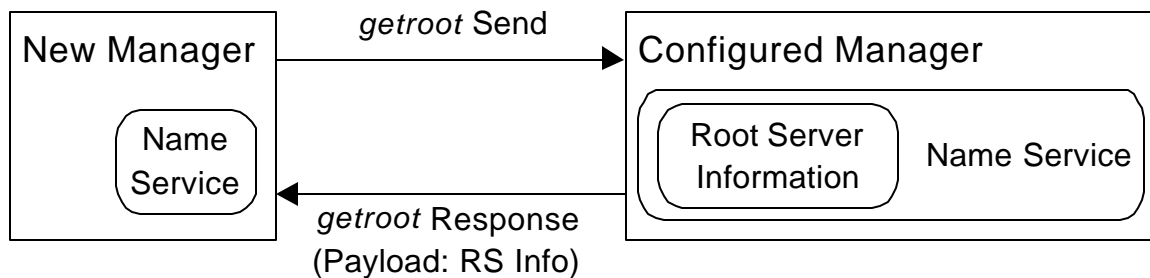
Note: The single period denotes the root domain.

Also, the *db.root* file is generated by the agent program and will contain a Start of Authority (SOA) record, a name server (NS) record, and an address (A) record. A sample *db.root* file is shown in the Appendix. The SOA record is used by the DNS system to indicate the primary master for a zone and specify how often slaves perform zone transfers. In this case, SOA record for the *db.root* file will say that the newly joined name server is the primary master for the root domain and for any slaves to update their data every two minutes. The NS record for the *db.root* file will say that this name server can directly handle queries for the root domain. The A record for the *db.root* file will show the name to IP address mapping between this name server and its IP address. The NS and A records are included so resolvers know which name server to contact and the name server's IP address when the resolver wants to resolve a name in the root domain.

Once the *named.conf* and *db.root* files are generated, the agent program forks off a process and runs the command *named* to start the BIND process. The BIND process will be set to listen for DNS messages on the same interface as the agent, but on a different port number, port number 55. The agent's name service is now configured, and the server record of the manager is modified to take note of this change from a start-up state into a configured state.

### 3.6.1.2 - At Least One Configured Name Service Discovered

If the agent program receives discover message(s) and at least one of the discovered name services is configured to handle name resolutions, then that means internal root servers already exist in the network. Since internal root servers already exist, the agent does not have to configure itself as a root server and can ask another agent for the root server information of the network [Fig. 12]. To obtain the root server information, the agent sends a *getroot* message to the agent on the configured name service with the highest server ID [Fig. 14].



*Figure 14 – Obtaining Root Server Information from Configured Manager*

This *getroot* message can be sent to any arbitrary service, and sending a message to the highest ID service ensures that only one *getroot* message will be sent in the network. The agent on the configured name service will receive the *getroot* message and use its state information to respond. As explained in Section 3.4.1, the agent stores the server record for each root server in the network. The configured agent will place this root server record information in the response to the *getroot* message. The agent on the unconfigured name service will receive this response, and use the root server data to configure itself. As the first configuration step, the agent will add the following to the `named.conf` file:

```
zone "." in {  
    type hint;  
    file "db.cache";  
};
```

This statement tells the BIND process to look in the file *db.cache* for hints on how to resolve names in the network. Then, the *db.cache* file is generated to include the newly acquired root server information from the *getroot* response. A sample *db.cache* file is shown in the Appendix. The *db.cache* will include name server (NS) and address (A) records. The NS records will say which name services to contact when trying to resolve a name in the root domain, and the A record will store the IP address of that name service. Once the *named.conf* and *db.cache* files are changed, the agent program will run the command *named* to start the BIND process. The agent's name service is now configured to handle name resolutions, and the server record of the manager is modified to take note of this change from a start-up state into a configured state. This change in status of the server record will be propagated to the other managers in the network so that the other agents in the network know that this name service is configured.

### **3.6.1.3 - Only Non-Configured Servers Discovered**

If the agent program receives discover message(s) but none of the discovered servers are configured to handle name resolutions, then that means internal root servers do not exist in the network. This scenario occurs when two or more new managers join the network at the same time. In this scenario, one of the new name services will have to take on the role of the internal master root server. All the starting-up name services will know of each other's server records through discover messages, and so the server ID will be used as the metric to decide who becomes an internal root master. The name service with the highest server ID will be the one that becomes the internal root master, and all

the other services with the lower server IDs will wait until a configured server appears in the network [Fig. 12]. The agent on the name server with the highest server ID will configure BIND just as it did in the first scenario. A root zone master statement will be added into the *named.conf* file and a *db.root* file to make the BIND process into the primary master for the root domain.

Once the *named.conf* and *db.root* files are changed, the agent program will run the command *named* to start the BIND process on the highest ID name service. The agent's name service is now configured, and the server record of the manager is modified to take note of this change from a start-up state into a configured state. This change in the server record will be propagated to the other managers in the network so that the other agents in the network know that this name service is configured. When the other agents receive notice of this new configured name service, the agents with the lower IDs will be in the second scenario. These unconfigured agents will then use the *getroot* message to ask the newly configured agent for its root server information and use that information to add a root hint statement in the *named.conf* file and to generate a *db.cache* file. Once these files are modified, the agents with the lower IDs will start the *named* process and be able to handle name resolution requests. These agents will move from a start-up state to a configured state, and this change in state will be propagated to the rest of the network.

### **3.6.2 - Configured**

The configured state is when a manager has already joined the network system, and the name service on the manager is able to handle name resolutions for the network. Either the agent has configured the name service to be an internal root server, or the name service knows the location of the root servers. During this stage, other managers might

join or leave a network, and the agent will be notified of these changes through the *discover* and *leave* messages. There are three possible ways the agent program will act depending what is discovered, and the state diagram for the discover scenarios are shown in Fig. 15. Also, the agent program handles the *leave* message.

### **3.6.2.1 - No Servers Discovered**

There is no change in the state of the network, and so, in this case, the agent program takes no action.

### **3.6.2.2 - Discover Only Unconfigured Servers**

In this case, new managers have joined the network. The agents on the new managers will ask an agent on a configured manager for root server information and use that information for configuration purposes. This process is detailed in Section 3.6.1.2. Configured managers do not need to worry about changing their configuration; they only need to respond to *getroot* messages from unconfigured managers.

### **3.6.2.3 - Discover Configured Servers**

In this case, a configured agent sees already configured name servers the agent has never seen before in the system. This scenario occurs when two configured networks join. It is possible that zone conflicts occur where more than one name service is the master for the same zone. The agent will have to be able to detect and resolve any conflicts between these name services. The whole process of detecting zone conflicts is shown in Fig. 15.

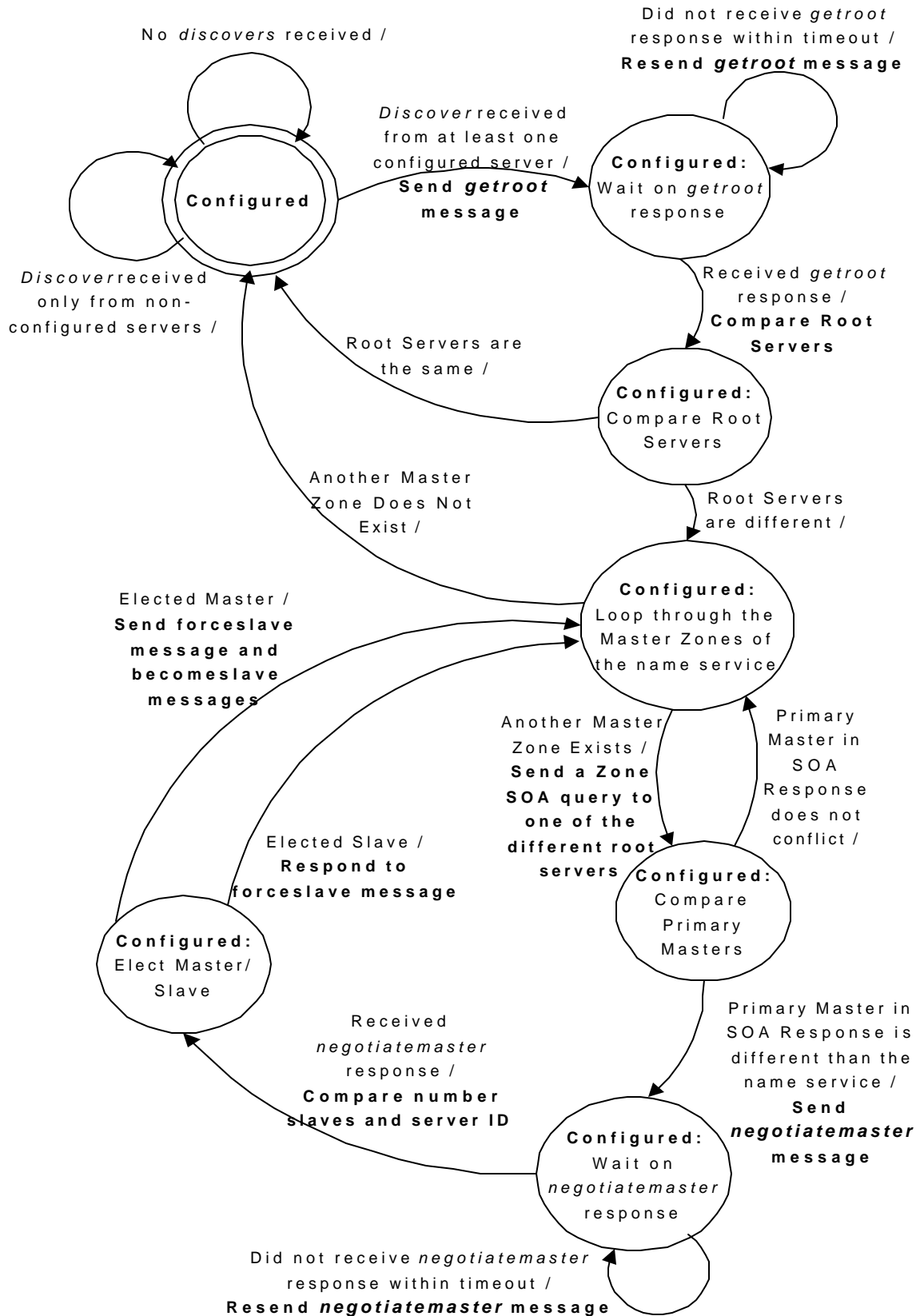
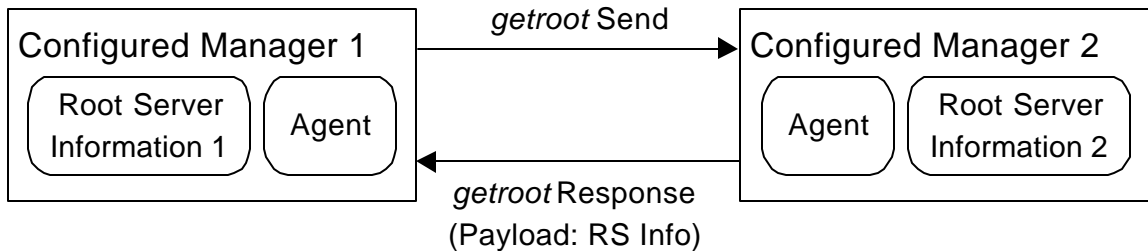


Figure 15 – Discover States in Configured Scenario

To detect any zone conflicts, the agent will go through a process of checking its own zones. First, the agent will send a *getroot* message to the agent on the newly discovered configured manager with the highest server ID [Fig. 16]. This way, only one *getroot* message will have to be sent. The agent will compare the root server information in the *getroot* response message with its own root server information. If the root server information is different from its own, the agent will query the differing root servers to see if any conflicts exist.



*Figure 16 – Obtain Root Server Information from Configured Manager*

For each zone the name service is the primary master for, the agent will send a Start of Authority (SOA) query to a differing root server [Fig. 17]. The response to the SOA query will say which name service is the primary master for that zone. The differing root server will answer the SOA query. If the answer is empty, then the differing root server does not know of the zone, and no conflict exists. If there is a SOA answer, the agent will look at the part of the SOA answer where it says which name server is the primary master for the zone. If the primary master in the SOA answer is the same as the agent’s name service, then there is no conflict. Else, if the primary master is different, then there is a conflict.

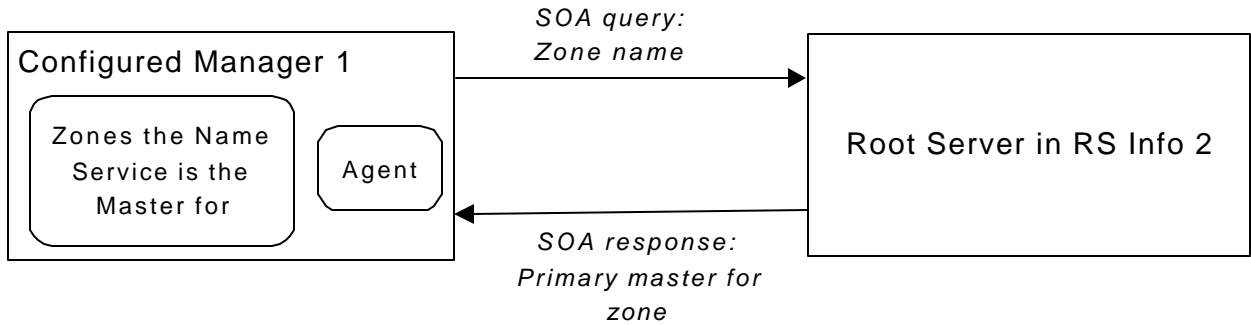


Figure 17 – SOA query to Root Server

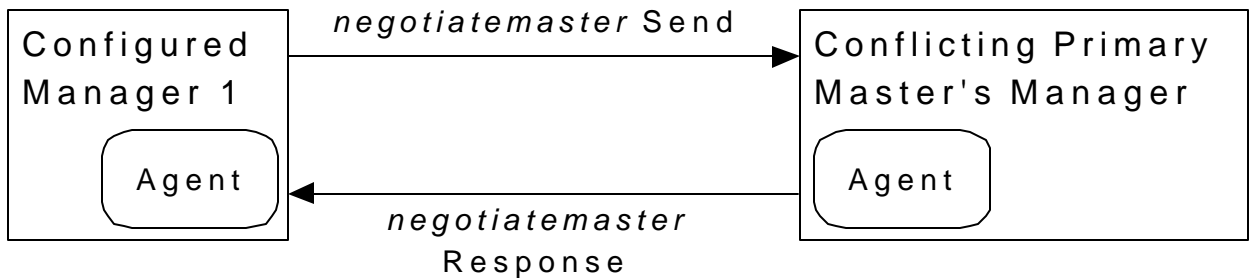


Figure 18 – negotiatermaster message

To resolve the conflict, the agent will send a *negotiatermaster* message to the conflicting name service's agent [Fig. 18]. The *negotiatermaster* message has information on the number of slaves the agent has for the zone and the agent's server ID. When the receiving agent sees the *negotiatermaster* message, the receiving agent will send back a *negotiatermaster* message with its number of slaves and server ID. The two agents will compare the information, and the agent with the most number of slaves or higher server ID will declare itself to be the master for that zone. Then, the new master agent will have the responsibility of merging the name data between the two conflicting servers. To merge the data, the new master agent will request a zone transfer from the losing agent's BIND process. A zone transfer returns all of the losing agent's name information for that zone, and the new master agent will use dynamic updates to add this information into its own BIND process. Once this merging of data is complete, the new master agent will send a *forceslave* message to the losing agent. The losing agent will become a slave, and

the losing agent will send a *forceslave* response that includes all the slaves the losing agent had. The new master agent will see this response and call the *becomeslave* message on all these slaves. The use of the *negotiatemaster* and *forceslave* messages is also explained in Sections 3.5.5 and 3.5.6.

#### **3.6.2.4 - Leave Message**

The agent program receives a *leave* message whenever a manager leaves or becomes lost in the network. The agent might need to reconfigure itself or change the zone state information based on the lost manager. For each zone the agent's name service is authoritative for, the agent will check to see if the lost server was authoritative for the zone. The possible states for when an agent sees a *leave* message is shown in Fig. 19.

If the agent is the master for a zone and the lost service is a slave for the zone, the agent will delete the lost service's server record information from its slave zone information. The agent will then send an update delete message with type NS to notify the BIND process that the lost name service no longer exists.

If the agent is a slave for a zone and the lost service is also a slave for the zone, the agent does not have to make any changes. The agent on the zone's master name service will make the changes to the system.

If the agent is a slave for a zone and the lost service is the master for the zone, then the agent will have to go through the process of electing a new master name service for the zone. All the other agents on the zone's slave name services will also go through the process of electing a new master. To try to elect a new master, the agent will first send a Name Server (NS) query to its own BIND process asking which name services are

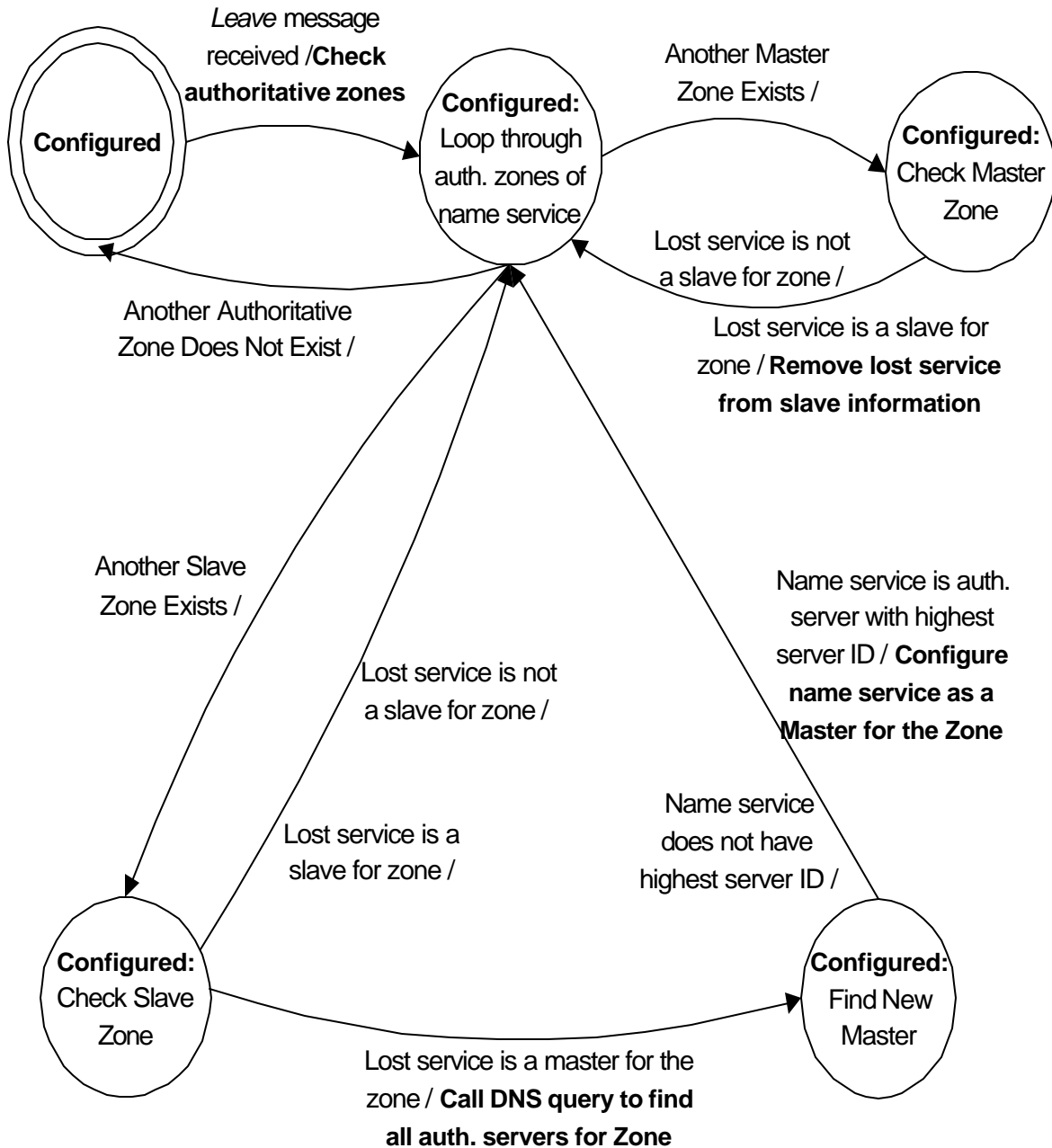


Figure 19 – Leave States in Configured Scenario

authoritative for the zone. The BIND process will return a list of name services that are authoritative for the zone. The agent will remove the lost service's record and the agent's own record from the list. The agent will then use its Known Server information to look up the server IDs for each remaining name service in that list. If the agent's own server ID is larger than the server IDs for all the other servers, the agent will make its BIND

process the new primary master server for the zone. After configuring the BIND process to be the primary master server, the agent will send *becomeslave* messages to the other name services authoritative for that zone. The agents on the other name services will change their *named.conf* file in response to the *becomeslave* messages, and thus be reconfigured as slaves to the new master for the zone. If the agent's own server ID is smaller than the server ID of another server, then the agent does not need to do anything until the agent receives a *becomeslave* message from the agent with the highest ID.

There is a special case if all the root servers for the network become lost. The agent will be to detect this through maintenance of the root server information, and when all the roots become lost, the agent will reenter into the start-up state. The agent will rediscover the other name services in the network and go through the same start-up configuration process explained in Section 3.6.1.

### **3.6.3 – Get Slave process**

Once an agent is configured with root servers, the agent will run a periodic event that finds slave servers for the agent's master zones. Periodically, the agent program will call the GetSlave function. The GetSlave function will loop through each of the agent's master zones. For each zone the agent's name server is the primary master for, the agent will check the Boolean flag saying whether or not the name service requires any more slave servers for the zone. If the zone needs another slave server, the agent will choose a random server from the Known Servers list that is not already a slave for the zone. The agent will then send the *becomeslave* message to the chosen server. When the other name server receives the *becomeslave* message, the agent on the other name server will add the following lines to the *named.conf* file.

```
zone "zone_name" in {
    type slave;
    masters { (master's IP address); };
    file "db.zone_name" };
```

The `zone_name` is the name of the zone the server is going to be a slave for, and the master's IP address will be the IP address of the agent sending the `becomeslave` message. The database file will not have to be generated because the slave will obtain a copy of the file from the master name server. Once the `named.conf` file is changed for the new slave, the agent on the new slave server will send a `SIGHUP` signal to the `named` process telling the process to reread the `named.conf` file. Once the slave configuration is complete, the agent for the slave server will respond with the `becomeslave` message. The agent on the master server will see the message and update its slave information with the information included in the `becomeslave` response. The agent's zone information will be updated and the name server (NS) and address (A) information for the new slave will be sent as a DNS update to the BIND process. To maintain proper delegation, the name server and address updates will also be sent to the name service of the parent zone. If the master has enough slave servers, then the flag specifying the need for more slaves will be set to false.

### 3.7 DNS messages

In addition to handling agent messages, the agent program will also listen to and send DNS messages. The agent program will listen to DNS messages on the standard DNS port number of 53. DNS messages can take on five possible Opcodes: *query*, *iquery*, *status*, *notify*, and *update*. The agent message only needs to perform processing on the *update* Opcode. All the other Opcodes can be forwarded directly to the BIND process. The possible states for the DNS messages are shown in Fig. 20.

For DNS messages with the Opcodes of *query*, *iquery*, *status*, and *notify*, the agent program will store the message ID of the message and the IP address the DNS message's sender. The agent will then forward the DNS message to port number 55 on the same network interface. The BIND process will be listening on this port number. The BIND process will answer the DNS query and send the response back to the port number 53 where the agent is listening for the response. The agent will recognize that the message is a response and use the stored message ID and IP address sender information to redirect the response to the original sender of the DNS message.

For DNS messages with the Opcode of *update*, the agent will do some processing with the message. Each update message has a Zone section [Fig. 6]. This Zone section specifies the zone name of where the update will occur. A name service running BIND can only accept updates for zones the name service is the primary master for. Since there are no restrictions on which name service updates are sent to, the agent's role is to ensure that update messages are directed to the name service that is the primary master for the zone name in the update message.

To accomplish this task, the agent will first extract the zone name from the Zone section of the update message. The agent then checks if the agent's BIND process is authoritative for the zone. If the agent's name service is a master for the zone, then the update message is forwarded directly to the agent's BIND process. The BIND process will see this update message and update its name database accordingly. After BIND is finished updating, the BIND process will send an update response message notifying the agent of the result of the update. If the agent's name service is a slave for the zone, the agent will use its Zone state information to find out who the zone's master is and redirect

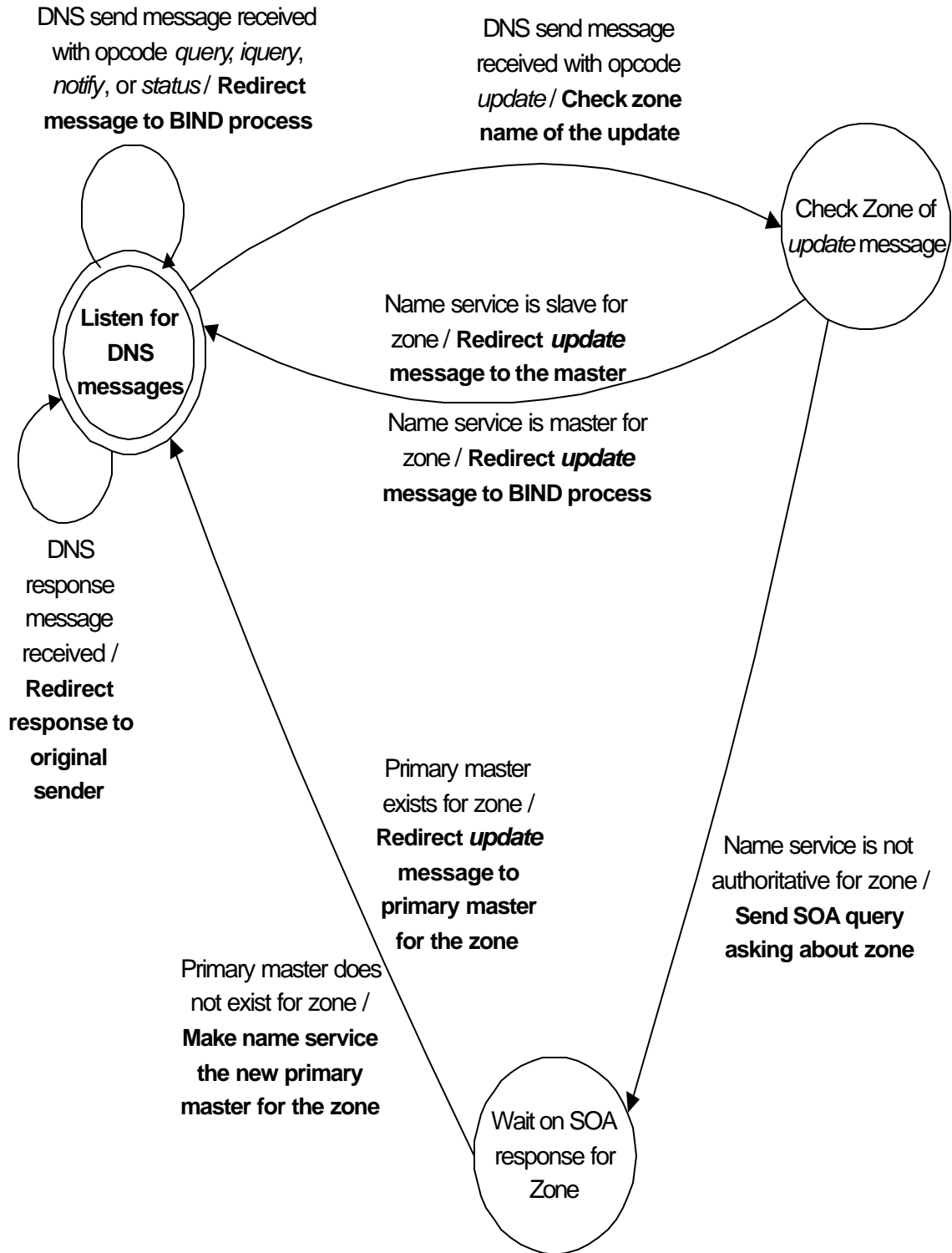


Figure 20 – States for DNS Messages

the update message to the agent on the master name service. The agent on the master name service forwards that update to the master name service's BIND process, and that BIND process will perform the necessary update and send a update response message.

If the agent's name service is not authoritative for the zone, then the agent will send a Start of Authority (SOA) query to its BIND process. The SOA query asks the DNS system who the primary master for a zone is. The question section for the SOA query will be the zone name of the update message. The BIND process will perform the standard DNS resolution to find the SOA information on the zone name. The BIND process then sends a DNS response message with the results of the SOA query back to the agent. The response message will say whether or not there exists a primary master for the zone name. If there exists a primary master in the network, then the response message will return a no error (NO\_ERROR) flag and include the name and IP address of the primary master name server. If there does not exist a primary master in the network, the response message will return a domain non-existent (NX\_DOMAIN) error flag.

If the agent sees that a primary master exists in the network, then the agent will forward the update message to the location of the primary master. The location of the primary master for the zone is included in the SOA response message.

If the agent sees that a primary master does not exist for the zone, then the agent will need to create a new zone in the network. To create a new zone in the network, the agent will configure its BIND process to be the primary master for the zone. The agent will add the following lines to the *named.conf* file:

```
zone "zone_name" in {  
    type master;  
    file "db.zone_name";  
};
```

The `zone_name` is the zone name specified in the update message. Also, the `db.zone_name` file is generated by the agent program and will contain a Start of Authority (SOA) record, a name server (NS) record, and an address (A) record. The SOA record will indicate that the agent's name service is the primary master and the NS and A records will indicate that the agent's name server can handle DNS messages for the zone and the IP address of the name server. A sample database file is shown in the Appendix.

Once the `named.conf` file is changed for the new master, the agent will send a SIGHUP signal to the `named` process telling the process to reread the `named.conf` file. The agent's name service is then ready to handle any DNS queries of the new zone name. If the new zone name has a parent zone, then the name service of the parent zone will be notified of this new zone. For example, if a new zone `company.com` is created, the agent who is the new master for the `company.com` zone will send a DNS update message to master name service for the zone `com`. The DNS update message will include a name server (NS) record saying which name service can handle queries for the zone `company.com`, and the agent will know the location of the `com` master server through a SOA query. Sending this information to the parent zone ensures proper delegation in the name system. Also, the flag specifying the need for more slaves in the new zone will be set to true. The next time the `GetSlaves` function is called, the agent will find slave servers for this new zone.

### 3.8 Implementation Notes

The agent program was implemented using Gnu C++ on the Unix operating system. The program consists of a set of classes that control the state of the agent and store network information. The main class for this agent program is the Agent class. The Agent class controls the flow of the program and handles the input and output of agent and DNS messages. The UDP protocol was used to send and receive agent and DNS messages, and the standard UNIX Berkeley socket library was used to create the necessary UDP sockets. The Agent class retrieved messages by using a select call on each socket to check the availability of data on the socket. Also, callback functions were used to perform time-dependent and periodic events.

The Agent class uses a number of storage classes that store the state of the network. These storage classes include the Servers class, the Domains class, and the MsgLog class. The Servers class helps to store what other managers and name services exist in the network. The Domains class helps to store what zones the agent is authoritative over. The MsgLog class stores information on the messages that have been received by the agent program. The C++ standard template library (STL) was used extensively in the implementation of these storage classes. In particular, the map, list, and vector template classes proved to be very useful in storing information.

To generate the configuration and database files for *named*, embedded Perl was used. Embedded Perl allows a C++ program to call Perl commands and subroutines. All of the Perl commands and subroutines required to generate the *named* files were placed in a file called *agent.pl*, and the agent program used embedded Perl to call the subroutines in *agent.pl* every time the *named* configuration files needed to be changed.

## 4 – Analysis

The implementation of the agent program successfully solves the problem of providing a scalable and fault-tolerant way to store name-to-address mappings and to handle name-to-address resolutions in the absence of human configuration or administration. The agent program handles the configuration and administration of a DNS name server implementation called BIND, and BIND stores name information and answers name queries with scalability and fault-tolerance in mind. In this section, how the name system works in the network will be discussed. In addition, there will be an analysis on how well the agent program works in terms of the amount of network traffic generated during reconfiguration and the amount of time required to reconfigure a name service. Also, the agent program will be compared to other systems that perform comparable tasks.

### 4.1 – Name System

The name system works in the context of a network system that is self-configuring and self-administering. A name service resides on each manager in the network, and the name service depends on the other parts of the manager to inform the name service whenever another name service joins the network or a network host has been assigned an IP address. When a name service joins the network, all of the other name services will be notified of this event through the *discover* message, and when a network host connects and receives an IP address through DHCP, the name service will receive a DNS update message that has information about the new network host. Currently, the *discover* message and DNS update functionality is being developed, and this development nearing completion. The implementation of this name service

notification feature and other parts of the manager was done separately from the implementation of the agent program. Fortunately, the name system was designed to work independently of how the other parts of the manager were implemented. The only communication between the name system and the other portions of the network system is the *discover* message and DNS update messages, and these two message formats are standard. The name service does not need to know anything else from the rest of the network system.

Even though the name system has not had a chance to be integrated with the rest of the network system, the functionality of the name system could still be tested independently from the other parts of the network system. A modification was made to the agent program so that the agents in the name system would know about each other, and an additional program, called *command*, was written to send DNS updates to the agent program. To take the place of the *discover* messages received from the manager, the agent program would broadcast its own server information in the form of a *discover* message to the other agent programs in the network. For testing on a small network, this broadcasting of the *discover* message does not create an unbearable amount of network traffic. To simulate the DNS update message sent by the manager, a program called *command* was created. The *command* program reads input from a command line, uses the input to create a DNS update message, and sends that DNS update message to the name service. A user specifies the name and IP address of a machine in the command line.

## 4.2 – Testing the Name System

With the modification of the agent program and the addition of the command program, it was possible to test the name system. The name system was tested using three UNIX machines. The components and set-up of the machines is shown in Fig. 21. These UNIX machines would act as the manager the name service resides on. Each of the three machines was given a name and IP address, and the three machines were connected together to form a local network. BIND was installed on each of the three machines, and a copy of the agent program was placed on each machine. To start a name service on a machine, the agent program is run. Once the agent program is started, the

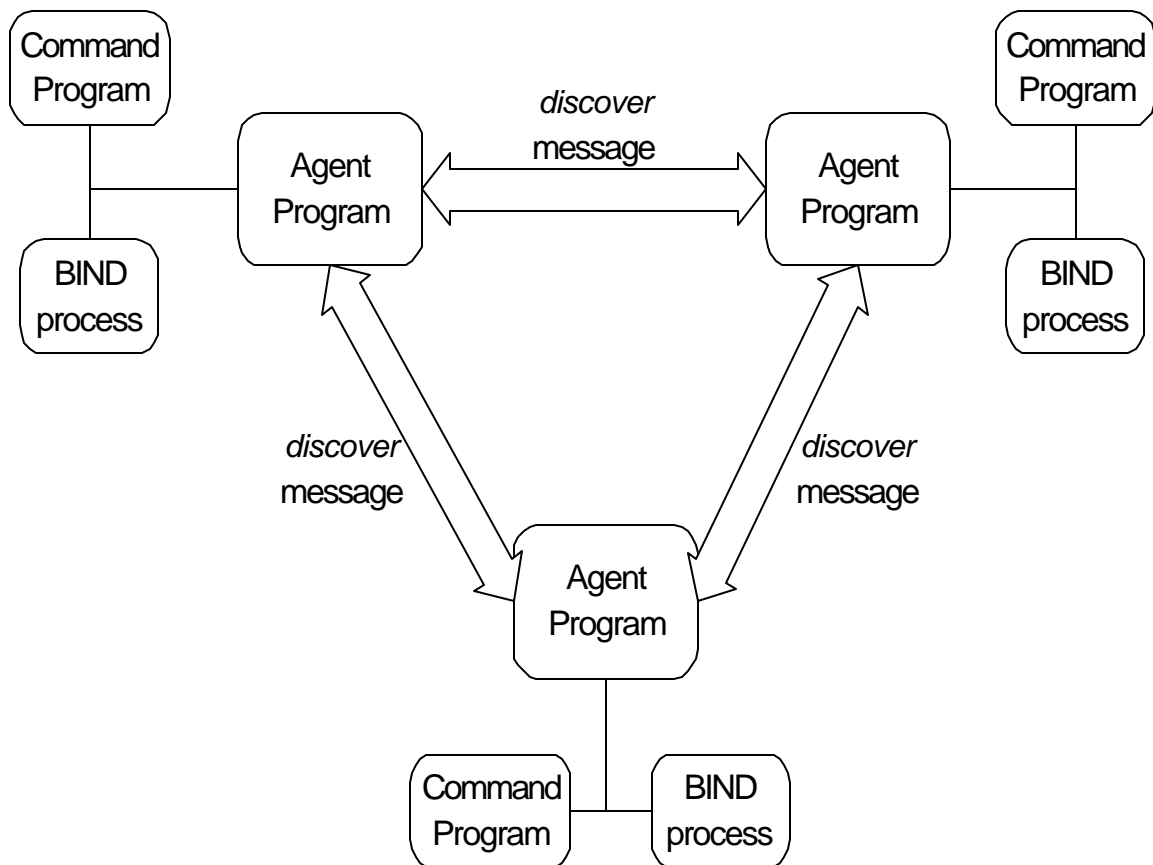


Figure 21 – Testing the Name System

agent sends *discover* messages to the IP addresses of the other two machines notifying the other agents of a new name service in the system. The *discover* message has the name, IP address, and server ID of the new name service. When the other agent programs see the *discover* message, the other agents will send their own *discover* message to the network. This way, each agent will know of each other. Based on these *discover* messages, the agent will know how to configure BIND and start a *named* process on the machine. The *named* process will then be able to handle DNS messages.

To simulate name services entering and leaving a network, the agent program was run at different times on the three machines. The three machines start off with the agent program not running on them. To make sure that each agent handled the start-up scenario correctly, the agent program was run a minute apart on the three machines. The first running agent program would see itself as the first name service in the system. This first agent correctly configured the BIND process to be the primary root master for the network. The second and third agent programs would be start-up services joining a network with a configured service. By checking the debug messages, it was verified that the second and third agents sent a *getroot* message to a configured agent and received an appropriate response. Using this root information, the second and third agent programs correctly configured the BIND process.

To check that a root master was properly chosen in the case where name services joined at the same time, the agent program was run within five seconds of each other. Through the *discover* messages, the three agents saw that all the services were in the start-up stage. It was verified that the agent with the highest ID declared itself to be the master, and the other agents used the *getroot* message to obtain root information. After

the agent programs configured their BIND processes, the command program was used to send DNS update messages to the name service. The DNS tool, nslookup, was used to confirm that the name updates were stored in the name database correctly and to send DNS queries to the name service.

Also, tests were performed to show that the agents reconfigured the BIND process correctly in the case of missing or lost name services. To test this functionality, the agent program was stopped on a machine. Before the agent program exited, the agent sent a *leave* message to the remaining agents in the network. Depending on which machine stopped running the agent, the remaining agents were able to successfully elect a new master or update their slave information. The agents on the machines were able to reconfigure their BIND processes properly regardless of which machine the agent was stopped on.

To test the merging functionality of the agent program, physical connections between the machines were disconnected and reconnected. Each machine started off with the agent program not running. The cables connecting the three machines were disconnected so that the three machines could not communicate with each other. Next, the agent program was started on each machine. Each agent configured its BIND process to be the primary root master. Then, the cable connecting a pair of machines was reconnected, and a *discover* messages were sent to each machine. The two configured agents were able to negotiate a master and merge their name data successfully. The remaining agent was then reconnected to the network as well, and the negotiation process was completed successfully as well.

Now that the agent program works correctly in the framework of the name system and the whole network, the performance of the name system can be analyzed. The name system should not be a bottleneck for network communication and should be able to handle a fair amount of name queries and updates. The name system does not place a large load on the processors of the machines in the network. Also, the name system can handle a decent amount of DNS traffic and is easily extensible to handle more traffic.

### 4.3 – Memory Usage of Name System

An individual name service does not place a large load on the CPU time or memory usage of a machine. The two components of the name service, the agent program and the BIND process, took up approximately 2.4% of the total memory of the machines used for testing the name system. The machines used for testing all had Intel Pentium 450 MHz processors with 128 Megabytes of RAM. To show the memory usage of the programs, the agent program was run on a machine, and the program *top* was run to give a memory and CPU usage report. The output of the *top* program is shown in Table 5. As shown in Table 5, the agent program occupied 1.7 Megabytes of memory and the BIND process, *named*, occupied 1.4 Megabytes of memory.

*Table 5 – Output of top: Memory usage of name service*

Size	RSS	share	% CPU	% MEM	
1784	1784	1456	0.0	1.3	agent
1448	1448	708	0.0	1.1	named

Both of these processes were idle when the *top* was run, and so the CPU usage is zero for both processes. In times of heavy agent and DNS message traffic, the CPU

usage is around 5%. During zone transfers, the memory usage may go up to two or three times the idle usage because *named* forks off new *named* processes to handle zone transfers, but 128 Megabytes of RAM should be sufficient to support this extra load. These numbers support the assertion that the name service does not place a large load on a machine. The machine is not overloaded or overworked by the name service and will not be a bottleneck to communication in the network. The assumption was made that machines running name services should have a minimum set of specifications, but presently, computers with 450 MHz processing capabilities are standard, and multi-user server machines often have more than 128 Megabytes of RAM. Standard computers should be able to run the agent program and the BIND process without many problems.

## **4.4 – Bandwidth and Processor Load of Name System**

An individual name service should be able to answer a fair amount of DNS queries and updates. In addition, name services should be able to perform zone transfers. The zone transfer is the mechanism used by BIND to replicate name information through the network. Also, the name service should be able to respond to agent messages. When the name services in the network require reconfiguration, a set of agent messages will be transferred from agent to agent. The following sections describe each of these different types of communication and the load they place on the name system.

### **4.4.1 – DNS Query and Update Messages**

The machines the agent programs were tested on all used 100/10 Megabit Ethernet cards. Since all of the machines were able to use 100 Megabit-capable Ethernet cards, the bandwidth for each machine is 100 Megabit/second. Assume that 100 bytes is the rough average size for a DNS packet [2]. Since there are 8 bits/byte, each DNS

packet is approximately 800 bits. To figure out the optimal bandwidth for DNS packets in the system, the following calculation is performed using the optimal estimate of the network conditions:

$$(100 \text{ Megabits} / \text{second}) / (800 \text{ bits} / \text{DNS packet}) = 125,000 \text{ DNS packets} / \text{second}$$

The number of DNS packets per second seems very large, but this number is derived with the assumption of optimal network conditions. It is unlikely that 100 Megabits/second will be available for use because other messages might be coming in from the network interface card and physical network connections might be slow. Also, the processor will not be able to handle 125,000 DNS packets per second. The processor can handle roughly several hundred DNS packets per second. So, the bottleneck for handling DNS packets will be the processor and not the network connection. Given that the processor can handle several hundred DNS packets per second, the name service should be able to comfortably serve about the same number of hosts simultaneously performing name intensive tasks like web surfing. Thus, for a network with a thousand hosts, there needs to be three or four name services running in the system to answer name queries and updates. Also, running three or four name services provides replication for the name data in case one of the services fails or becomes unreachable.

Running three or four name services is reasonable for a network of a thousand hosts. For replication purposes, the absolute minimum of name services to run for any network should be two. Adding one or two to the minimum number only helps the system to be more fault-tolerant. Also, for a typical organization, it is not uncommon to run five to seven name servers [2]. The company where the agent program was

developed has around a thousand employees, and the company uses four name servers to service the hosts in the internal network.

#### **4.4.1.1 – Special Processing for DNS Update Message**

In the name system, the *named* process does not directly handle DNS queries and updates. The agent program will first see any DNS messages and forward along the message to the *named* process. When the agent program sees a DNS query, the agent simply forwards the query to the *named* process controlled by the agent program. The forwarding process is more complex with DNS updates. When the agent program sees a DNS update, the agent will check if the agent's *named* process is authoritative for the update zone. If the *named* process is authoritative, then the update will be forwarded to the master name service. No extra DNS traffic will be generated in this case.

Otherwise, a SOA query is made and sent to the *named* process asking it to find the primary master for the update zone. If the update zone already exists in the system, then the update message is forwarded to the primary master for the update zone. Else, the agent will reconfigure its BIND process to handle a new zone and send a DNS update message to the name service for the parent zone. In the worst case, only two extra DNS messages are generated for a DNS update. This extra traffic is negligible, and so the special processing that might be required for a DNS update message will not affect the performance of the name system and whether more name services will be needed to handle DNS messages.

#### **4.4.2 – Zone Transfers**

In addition to handling DNS updates and queries, name services will also have to handle zone transfers from other name services. Zone transfers happen when a master

name server for a zone transfers its zone data to the slave servers for the zone. A zone transfer is sent as many DNS packets in a row. If a master name server has 100 name records for a zone, then the zone transfer consists of sending each name record as a separate DNS packet. As long as the processor can handle the packets and there is enough bandwidth available to send the zone transfer packets, the name service will be able to handle the zone transfers. From the previous analysis in this section, there will be sufficient bandwidth and processor power to handle zone transfers in the name system.

However, if the processor does become overloaded or the network does become congested, the two *named* processes performing a zone transfer have mechanisms for congestion control and message reliability. In BIND, a zone transfer is implemented by establishing a TCP/IP connection between two *named* processes. The TCP/IP protocol guarantees the reliable transfer of messages and can slow the send rate of a message based on network conditions. Thus, for the name system, zone transfers between name services will be reliable and properly replicate name information in the network.

## 4.5 – Analysis of Agent Messages

The name service will also have to handle agent messages. Agent messages use a text-based message format with each field delimited by a semicolon. Typically, the agent message will be about 100 bytes long, the same size as an average DNS packet. A sample agent message is shown here:

```
3;getroot;response;1;name;ip_address;server_id;1
```

This message returns information on the root servers in the system. Assuming that the name is 10 characters long, the IP address is 15 characters long, and the ID of the server is 17 characters long, the above agent message will be 67 bytes long. Taking into

account the additional headers for the transport protocol are added on and variations in the agent payload length, 100 bytes is a conservative estimate for the length of an agent message. So, an agent message size-wise is very similar to a DNS packet.

For the most part, agent message traffic will be very low. Once the name services in the network have been configured and working properly, the agent program does not need to do much, and agent messages are not transferred in this steady state. Agent messages are only really used to reconfigure the name services when a new name service enters or an old name service leaves the network. In these cases, there will be a burst of agent message traffic and other traffic generated from the other parts of the network system. For example, when the name service on a new manager enters the network, the discovery portion of the new manager will have to make the manager known to the other managers and to find the other managers in the network. There is a nontrivial amount of traffic associated with this discovery process, and this effect has to be taken into consideration when analyzing the amount of traffic the agent program generates. In the following sections, the traffic generated by the agent program during configuration and administration will be discussed. Also, the amount of time the agent needs to configure and administer a name service will be discussed in detail, and a comparison between the name system and existing research will be presented.

#### **4.5.1 – Configuration**

When a new name service enters the network, the name service will go through a configuration process. The agent will first see what other name services have been discovered, and then based on this discover information, the agent will configure its BIND process accordingly. For the testing of the agent program, the agent waited ten

seconds for *discover* messages. Ten seconds is ample time for sending and hearing the *discover* messages and for starting different agent programs simultaneously. This wait time is known as the check-discover delay time and can be denoted by the variable  $D_d$ . After the check-discover delay time passed, the agent program checked what other name services existed in the network and configured and started the *named* process. Also, before the agent generates a *named.conf* file and calls *named* to start its BIND process, the agent might need to obtain information from other agents in the network. In these cases, agent message traffic will be generated, and it could take some time for the agent to configure its BIND process.

In addition to the agent message traffic generated by the configuration process, there will be discover traffic generated by the discovery portion of the manager. In order for the agent program to work properly, the agent needs to know of the existence of every other name service in the network. The discovery portion of the manager is required to provide the agent with this information, and the amount of time and number of messages required to obtain this information should be quantified. Since the discovery portion of the manager was developed independently of the name service and is still being worked on, it is difficult to quantify the time and traffic of the discovery process.

However, some useful metrics can be used as ways of thinking about the discover process. To represent the amount of time it takes for discover information to propagate through the network, the variable  $T_d$  will be used. For discover information to spread through the network, the information will pass through a series of managers. The discover information passing through a single manager to another manager can be seen as a network hop. Each network hop will take a certain amount of time, and since this

network hop time is variable, the variable  $T_d$  will be expressed in terms of the unit hop-time. For example, if there are ten machines connected all in a row, the time it will take for a message to reach from one end of the network to the other end will be nine hop-times. The  $T_d$  for this case will be nine hop-times.

To represent the number of messages that need to be sent, the variable  $N_m$  will be used. The variable  $N_m$  will represent how many machines need to hear the *discover* message for a new manager. For example, if a new name service joins a network that already has ten managers, then the ten managers will have to hear the discover for the new name service, and the new name service will have to hear the discover for the ten existing name services. So, in this case,  $N_m$  will be equal to twenty. Using these two metrics will help in the thinking of how well the agent configuration process works. With these tools in hand, all the possible start-up scenarios can be looked at in more detail.

#### **4.5.1.1 – Start-up Scenario: No Name Services Discovered**

In this scenario, the name service is the first one in the system. No discover messages are received from the rest of the network, and the agent does not need to send any agent messages to other agents in the network. There is no  $T_d$  or  $N_m$  associated with this scenario, and the amount of time it will take the agent to configure the name service will be the check-discover delay time,  $D_d$ , which is set to ten seconds.

#### **4.5.1.2 – Start-up Scenario: At Least One Configured Name Service Discovered**

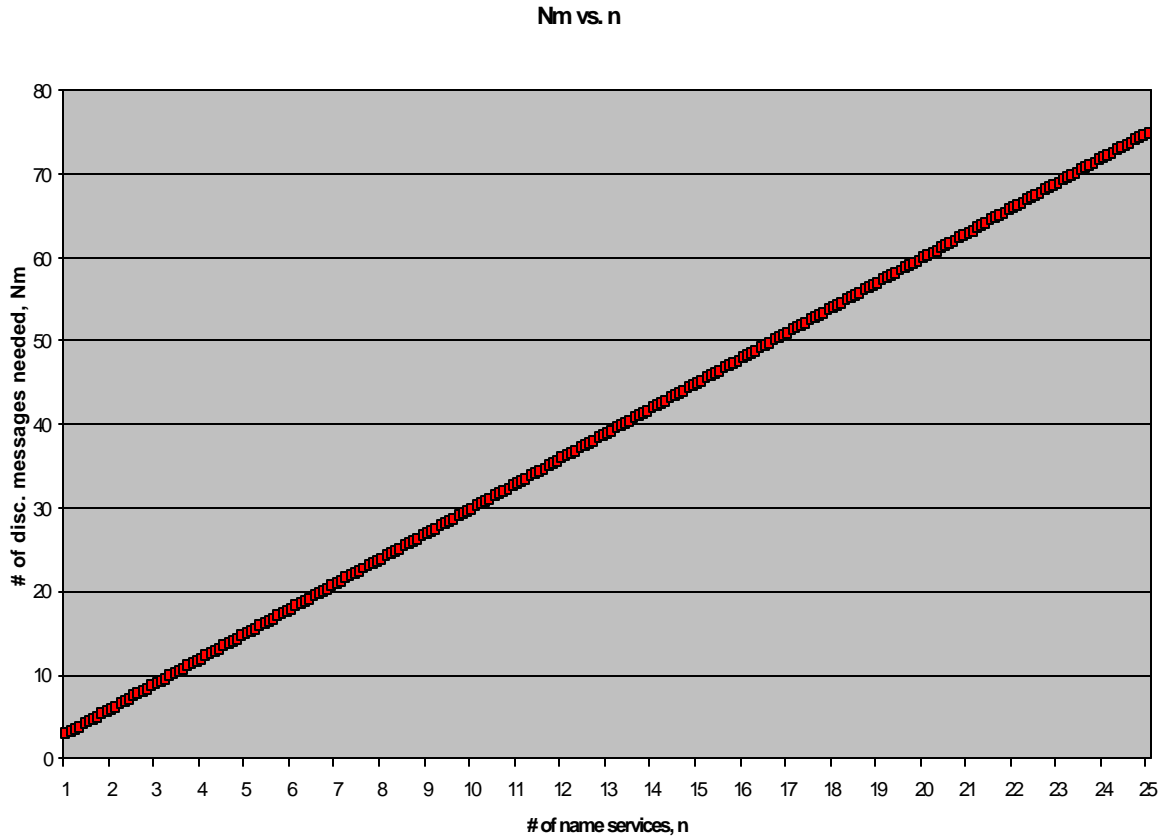
In this scenario, a new name service entering the network will hear *discover* messages about all the other name services in the network. Initially, given there are  $n$  name services already in the network, there were be a minimum of  $2 * n$  *discover*

messages passed to the agents in the program. The  $n$  existing name services each need to hear a *discover* message for the new name service, and the new name service needs to hear a *discover* message for each of the  $n$  existing name services. So,  $N_m$  will start off with the value  $2 * n$ . Since hop-times are on the order of milliseconds and the agent program was tested on a small network, the time to discover,  $T_d$ , will be much less than the check-discover delay time,  $D_d$ . Since  $D_d$  is sufficient time for the discover process to happen, the time  $T_d$  does not have to be taken into consideration in this scenario.

After the check-discover delay time,  $D_d$ , has passed, the new agent will look through its discovered name services and send one *getroot* agent message to the name service with the highest ID. The agent on that name service will respond with a *getroot* message, and the new agent will configure BIND with the root servers given in the response message. The whole configuration will take the time  $D_d$  plus the round-trip time of the *getroot* message. The round-trip time of the *getroot* message will be on the order of milliseconds, and so the total time to configure will be approximately  $D_d$ . After configuration is complete, *discover* messages have to be sent notifying the rest of the network that the new name service is configured and ready to accept name queries. There will be a cost of  $n$  extra *discover* messages that have to be sent.

To summarize, only one agent message will be used in the configuration process in this scenario, and the amount of time to configure the new name service will be  $D_d$ , which is set to ten seconds. Also, the number of *discover* messages needed,  $N_m$ , will be  $3 * n$ , where  $n$  is the number of name services already in the network. A graph of this relation is shown in Fig. 22. Since the processors for the managers can handle several hundred agent messages a second, this amount of network traffic is acceptable for

networks with less than fifty or so existing name services. This is within the limit for an organization running five to seven name services.



*Figure 22 –  $N_m$  vs.  $n$  for At Least One Configured Found during Start-up*

For larger networks, the amount of network traffic generated by requiring that every name service needs to know of another might be unbearable. However, the reason for this requirement is to ensure that the administration function of the agent works properly, and this requirement can be relaxed for the configuration case. As long as a new agent knows about one configured name service, the new agent will have enough information to configure its BIND process. The discovery portion of the manager can be changed to discover less managers, and reducing the discover traffic will make the configuration portion of the name service scalable.

#### 4.5.1.3 – Start-up Scenario: Only Non-Configured Servers Discovered

If multiple name services start up at the same time, the agents of the name services will have to go through the process of electing a master and configuring their BIND processes. If  $n$  name services all enter the network at the same time, each name service will hear  $n - 1$  *discover* messages. So,  $N_m$  will start off with the value  $n * (n - 1)$ . Like the last scenario,  $D_t$  is sufficient time to for the discover process to happen, and so the time  $T_d$  does not have to be taken into consideration in this scenario.

After waiting the check-discover delay time  $D_d$ , the agent with the highest server ID will configure its BIND process to be the root master for the network. Once this configuration is complete, the other agents in the network need to be informed of the newly configured name service. At the minimum,  $n - 1$  *discover* messages will be used to transmit this information. After waiting another delay time  $D_d$ , the  $n - 1$  unconfigured agents will use the *getroot* message to obtain the root information from the master server. As each unconfigured name service becomes configured through the *getroot* information, more *discover* messages will be generated. Since each name service needs to announce its configuration state to everyone else, there will be  $n * (n - 1)$  *discover* messages generated as the name services become configured.

For  $n$  name services all starting at the same time,  $n - 1$  agent messages will be used in the configuration process. The master name service will take the delay time  $D_t$  to configure, while the other name services will take  $2 * D_d$  to configure. Also, the number of *discover* messages needed,  $N_m$ , will be  $2 * n * (n-1)$ . A graph of this relation is shown in Fig. 23. In this scenario, the agents do not take a long time to configure their name services, but a large amount of network traffic might be generated if a large number of

name services start up at the same time. However, there is little chance that this scenario happens. In general, name services enter one by one, and it is unlikely that a bunch of servers will all start up within ten seconds of each other.

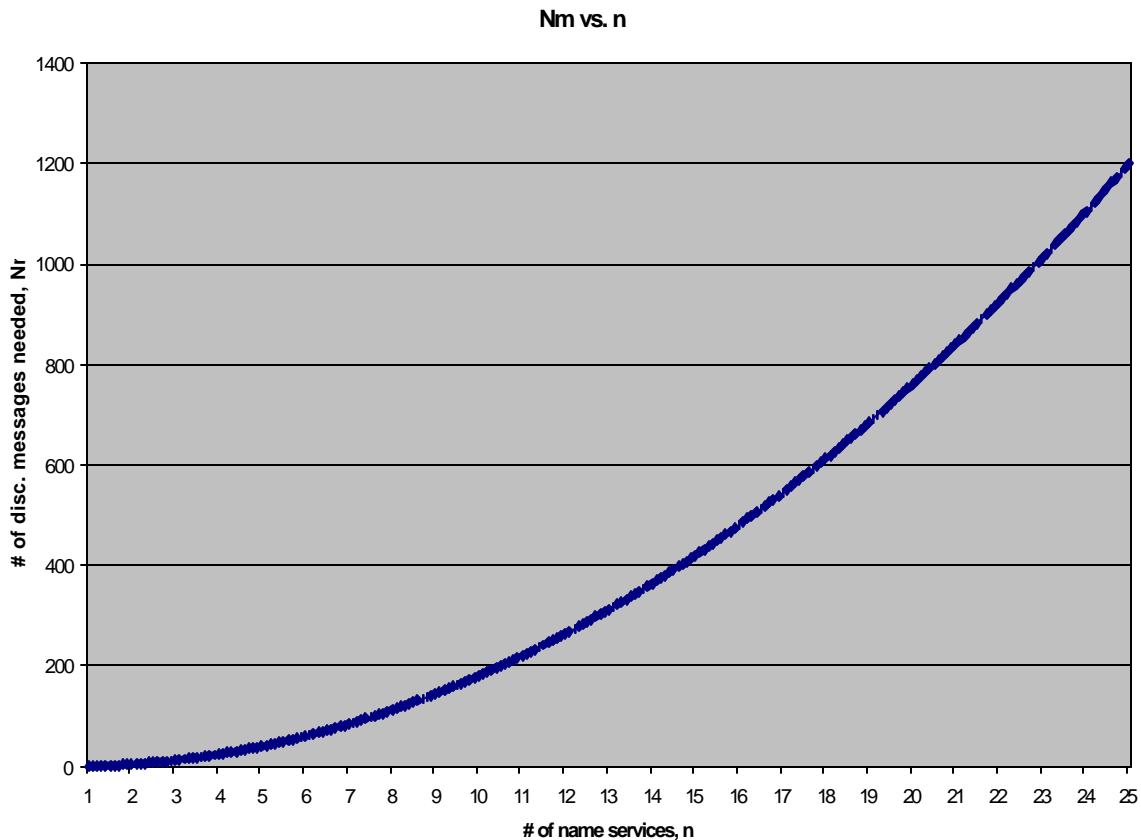


Figure 23 -  $N_m$  vs.  $n$  for None Configured during Start-up

#### 4.5.1.4 – Comparison of Configuration Process With Other Systems

The agent program is able to autonomously configure a DNS process without any human intervention. To accomplish name service configuration, multiple agent programs communicate with each other and exchange information, and based on information from other agents, an agent will generate the *named.conf* file and call the *named* command. No human is required to configure the name services, and configuration is done in a

decentralized manner. This is different from existing configuration solutions in industry and research.

Name service solutions in industry require humans to manually install the server software and configure the server to act as a name server. For example, there is much planning and modeling involved in the configuration process of the Lucent QIP Enterprise system [19]. In order for the QIP system to work effectively, physical and logical relationships between network objects and users have to be determined. This can be a time-consuming process for a network administrator. With the agent program, the only thing to do is to run the agent program. The agent figures out how to configure the name server, and a trained professional is not required for this configuration process.

Also, the time it takes for the agent program to configure a name service is minimal. As explained in the sections preceding this one, the time it takes for the agent to configure the BIND process is either one or two discover delay times. Given that the discover delay time is ten seconds, the whole configuration process will take less than one minute. This configuration time can be compared to the times given in the “Zero Internet Administration” paper [12]. In this paper, the researchers perform an experiment to determine how long it takes two groups of people to configure a name server. The two groups of people are shown in Table 6. One group had experience in network management, while the other group does not. These two groups of people were asked to configure a name server using different tools. The command line tool helps convert name-to-address bindings into the correct format, and the *nssetup* tool is the tool developed by the researchers and is the focus of the paper. The results of the experiment are shown in Table 7 [12].

Table 6 – Participants Background

ID	Computer Use	Position	Background
1	work/research	1 <sup>st</sup> year Masters Student in Computer Science	Extensive use of PC and workstation
2	work/research	2 <sup>nd</sup> year Masters Student in Computer Science	Experience in network management

Table 7 – Time use for configuring a name server

ID	Time (minutes)	
	1	2
without any tool	600	120
using a command line tool	30	8
using <i>nssetup</i>	3	2
using agent program	< 1	< 1

Looking at the table, using the *nssetup* tool saves a considerable amount of time used in configuring a name server. The fourth tool shown in Table 7 was not part of the original experiment but was included to show the time performance of the agent program. Not only does the agent program take less time to configure a name service, no real prior computer experience is needed to know how to use the agent program. Compared with other systems, the agent program is different in that no human is required and a minimal amount of time is used to configure a name service.

#### 4.5.2 – Administration

After the name service goes through the configuration process, the name service enters into the configured state. Once the name service enters the configured state, the

name service can answer name queries and accept name updates. In addition, the agent of the name service will have to perform administrative tasks. The administration of the name system involves maintaining the name database and paying attention to any new name servers that may join the system.

Maintaining the name database is fairly straightforward and follows the rules of dynamic DNS. Tasks involving the maintenance of the name database are done using DNS queries and updates and do not involve agent messages. The manager will use DHCP to assign IP addresses to hosts, and that name-to-address binding information will be sent to the name service in the form of a DNS update packet. The agent will see this update message and direct the message to the correct name service in the network. That name binding will be stored in the appropriate place and will be available for name resolution in the future. If the name has a domain name that is new to the system, the agent can create storage space for the new domain name. For the creation of any new zones, the agent will set up all the required parent and child delegation information between name services. These tasks generally occur in the background and are performed without much effect on the network traffic or load of the system.

When a new name service enters the system or a name service leaves the network, agent programs on configured name services will use agent messages to obtain information on how to reconfigure their BIND processes. How the agent reconfigures its BIND process depends on the *discover* or *leave* messages that it receives. This reconfiguration process can place a non-negligible load on the network. Many messages may be passed around, and the process of finding zone conflicts and electing new masters can be quite complex. Using the same metrics used in the configuration analysis section,

the possible scenarios that can occur for a configured name service can be looked at in more detail.

#### **4.5.2.1 – Configured Scenario: No Servers Discovered**

In this case, nothing has to be done. There are no changes to the state of the network.

#### **4.5.2.2 – Configured Scenario: Discover Only Unconfigured Servers**

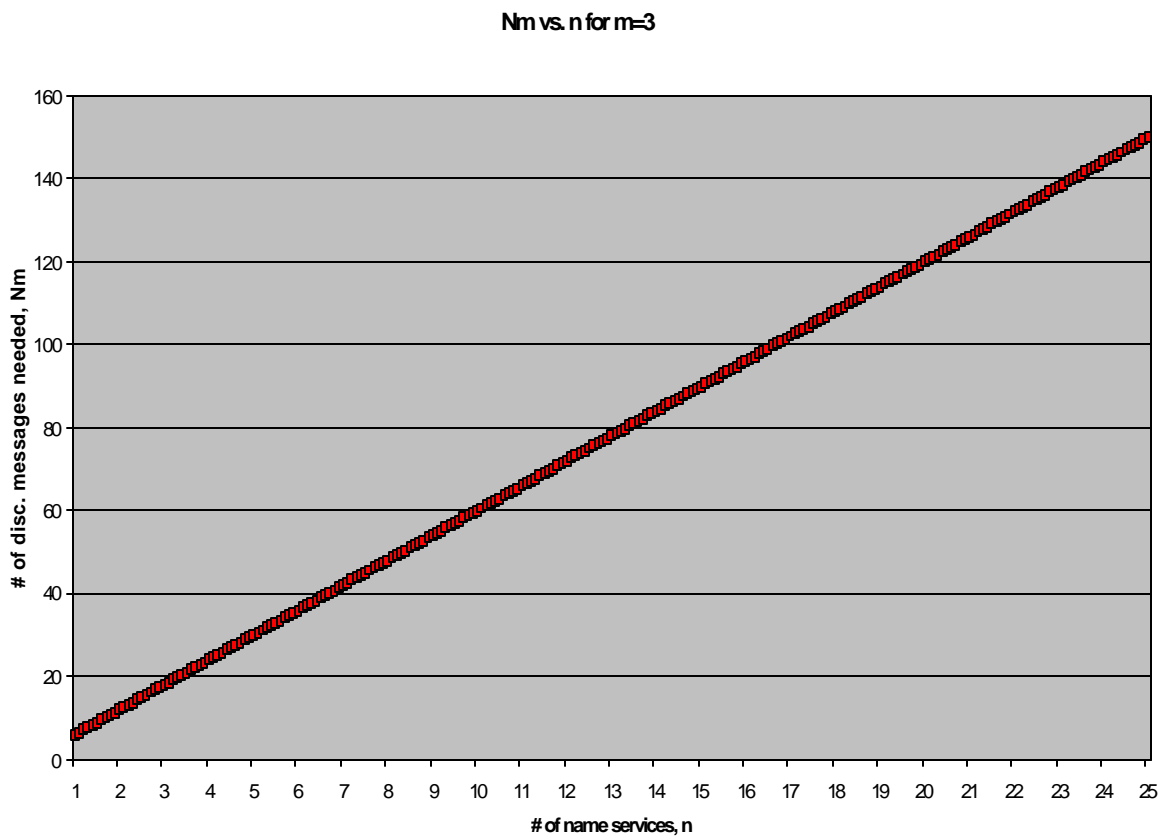
In this case, the agent does not have to perform any reconfiguration. The new unconfigured name service will not have any zone conflicts with the configured name service, and so the configured agent will not have to perform any reconfiguration.

#### **4.5.2.3 – Configured Scenario: Discover Configured Servers**

This case occurs when two configured networks join together. One network has  $m$  configured name services and another network has  $n$  configured name services. Somehow, the two networks become connected, and the name services will have to configure themselves if any zone conflicts exist. Before detecting any zone conflicts, *discover* messages have to be passed around. A name service on one network will have to hear *discover* messages for each name service on the other network. If there are  $m$  machines on one network, then a total of  $m * n$  *discover* messages have to be heard on one network. Similarly,  $n * m$  *discover* messages have to be heard on the other network. So, the total number of *discover* messages,  $N_m$ , that have to be heard is  $2 * n * m$ . This relation is shown for  $m = 3$  in Fig. 24.

For small networks, the discover traffic will not be unbearable. For larger networks, in order to make the discover traffic bearable, the *discover* messages can be made to propagate slower through the network. The time to discover,  $T_d$ , will be longer,

leading to a longer time to reconfigure the name services. This is a satisfactory tradeoff to ensure that the network traffic is not too high. It is important that the network is able to pass messages, but reconfiguring the name services is not necessarily a critical event. Name services remain available to hosts to answer queries, and even though name services might not have all a zone's name data for a period of time, eventually the name service will be reconfigured properly.



*Figure 24 –  $N_m$  vs.  $n$  for  $m=3$  in Configured State*

In addition to the *discover* messages, there are agent messages that have to be sent in order for a name service to reconfigure properly. All the agent messages that are sent and which agents have to send those messages are detailed in Table 8. Each name service sends a *getroot* message to a machine on the other network to obtain the root

server information for the other network. Using the other network's root information, the agent sends SOA queries to the other network to see if any zone conflicts exist. If a zone conflict exists, then the *negotiatemaster* and *forceslave* messages are used to resolve the conflict. Also, a zone transfer is called to merge the name data for the conflicting servers. As long as there are not too many zones being used in the network, the reconfiguration process will not place a large load on the system. Each agent message is small in size, and zone transfers happen all the time. However, if there are many zones, the discover message time,  $T_d$ , can be made longer so that the reconfiguration process does not happen all at once. With these considerations in mind, the reconfiguration of the name services when a zone conflict occurs will be scalable.

*Table 8 – Messages for Reconfiguration*

<b>Message</b>	<b>How many messages sent</b>
<i>getroot</i> message	All configured agents send <i>getroot</i> message to the newly joined network. $m + n$ total <i>getroot</i> messages sent
SOA query	One for each zone in the network
<i>negotiatemaster</i> and <i>forceslave</i> message	One for each conflicting zone in the network.
zone transfer	One for each conflicting zone in the network.

#### **4.5.2.4 – Configured Scenario: Leave Message**

This case occurs when a name service leaves the network. When a name service leaves the network, the remaining  $n$  name services should receive a *leave* message. So, there are  $n$  leave messages that have to be sent. Based on which name service left the

system, the agents will reconfigure their BIND processes. If the leaving name service was a master for a zone, the slaves will elect a new master based on the server ID, and the new master will send *becomeslave* messages to the slaves. The number of *becomeslave* messages sent will be the number of slaves. If the leaving name service was a slave, there will be no agent messages passed, and no reconfiguration of any name services will take place. In this case, not much network traffic or load will be generated, and the network can reconfigure very quickly.

#### **4.5.2.5 – Comparison of Administration Process with Other Systems**

The agent program is able to autonomously administer a name system without any human intervention. New name-to-address bindings are sent to the name system through DNS dynamic updates, and the agent program is in charge of making sure the update messages reach the correct name service. Also, depending on *discover* and *leave* information, agent programs will communicate with each other to detect zone conflicts and reconfigure name services accordingly if conflicts exist. Administration is done in a decentralized fashion, and no human is required to manually enter in new name-to-address bindings and rewrite BIND's configuration files. In regards to the autonomous maintenance of the name database, the agent program is similar to existing administration solutions in industry, but the agent program provides a decentralized approach to administration and a mechanism to easily add new name services.

Existing administration solutions in industry take the integrated DNS and DHCP solution. New hosts joining a network obtain an IP address using DHCP, and a DNS update message is sent to notify the DNS system of the new name-to-address binding. For example, Shadow IPServer integrates DHCP and DNS into a single, unified system

architecture [33]. The agent name service system takes the same approach as commercial solutions in regards to new names in the network. However, commercial solutions cannot handle the creation of new zones and the changing of network as quickly as the agent program. Commercial solutions are centrally managed and primarily used for static networks, and the administrator is required to make a response to changes in the network. On the other hand, any agent can reconfigure a portion of the name system.

Name service solutions in industry are centrally managed by network administrators. The main users of name service solutions are large organizations that want to have an easy way for storing and resolving names in their system. These organizations will hire network administrators to help configure and set-up their name servers. To assist these network administrators, name service solutions are designed with central management in mind. For example, the Meta IP system has an 'Admin Console' that allows any authorized administrator to manage every element within the entire IP network, located on any Meta IP Site within the organization [20]. Also, the Nortel Network's NetID solution provides a single point of management for all IP addressing information, as well as DNS and DHCP configuration and management [23]. This central management scheme works well for an organization that can hire trained people to administer their networks, but the agent program works in the absence of trained people.

The agent program's decentralized approach makes it easy to for a new zone to be created in the system. An agent simply changes the *named.conf* file for its name service to handle a new zone, and the system can automatically store names and provide name resolution for that zone. In a centralized system, the administrator will have to manually

create the new zone in the system. Once the new zone is created, then the integrated DHCP and DNS approach can be used to autonomously add names in the name database. Any agent in the name system can create the new zone, and no administrator is needed before new names for the zone can be added in the name database. Also, the decentralized approach makes it simple to add a new name service in the system and combine existing name services in the network. The agent takes care of all the reconfiguration required when name services join a network and can handle a dynamic name server topology. Existing commercial solutions assume a relatively static name server network, and an administrator is required to perform any reconfiguration when the topology of the name server network changes. In addition to not requiring a human administrator to make changes, the agent name system is much more flexible in adapting to changes in topology and in handling new names than existing commercial solutions.

One feature the agent name system does not have is a fault detection mechanism. The agent does not detect if there are any errors in the configuration or administration of the name servers. The agent only performs the action of configuring and administration. This is in contrast to the DNS network management system implemented using a rule-based system and neural networks [24]. This network management system went through the BIND system log and searched for errors. Any errors were reported and suggestions were given on how to fix the error. This purpose of this system was to detect any hard to catch human errors made in the configuration and administration process of BIND. The purpose of the agent program is not to detect errors but to participate in the process of configuration and administration. By allowing the agent program to autonomously

configure and administer a name service, the common errors that occur during the setup and maintenance of the BIND process can be avoided.

## **4.6 – Comparison with ZEROCONF**

The goal of the Zero Configuration Networking (ZEROCONF) Working Group is to enable networking in the absence of configuration and administration. The goal of this working group is very similar to the goal of the larger project this thesis is a part of. However, the ZEROCONF project is still in the process of developing specifications for the system, where as there has already been an attempt to implement a self-configuring and self-administering network at the Research Corporation. The name portion of the implemented system is the agent and BIND name service system explained in this thesis, and the workings of the agent name service are different than what is specified in the requirements for the zeroconf protocol.

For the zeroconf protocol, there must be some scheme, existing across all the zeroconf computers, that will store zeroconf name-to-address mappings and provide resolution until a DNS server is able to be located [13]. Once a DNS name server is located, then that DNS server is used for name resolutions and updates. The agent name system provides a DNS name server for the hosts in the network to use. The whole point of the agent name system is for the agent to learn the state of the network and configure a DNS name server according. Hosts in the network perform name resolutions and queries right away, and there is no need to try to locate an outside service. Also, the newly configured DNS servers in the network can join the Internet, and hosts in the network will be able to communicate with any machine in the world.

With the agent name system, an independent scalable network can be built quickly. Because name servers are provided, many hosts can join the network and expect to communicate with each other and services immediately. DNS dynamic updates are received every time a new host joins the network, and so the agent name system satisfies the requirement that when a new host joins the network, the rest of the network will know about that name of the host. The zeroconf protocol also has this requirement for new hosts in a network. Also, the agent program configures the name servers in a fault-tolerant and scalable way by making name servers replicate information and maintaining the DNS naming scheme and tree-like structure. The zeroconf protocol does not specify anywhere how to build a naming system in the absence of a DNS server. In the zeroconf protocol, storing names and providing name resolutions among zeroconf machines is a temporary solution. The more permanent solution is for zeroconf computers to locate a DNS name server. After locating the DNS server, the zeroconf computers will achieve a full configuration state and will function like any other computer in the Internet. The agent name system is much more flexible and provides a greater breadth of services to hosts in a network than the zeroconf protocol does.

## **4.7 – Comparison with NetBIOS**

Like the agent name system, the NetBIOS system provides a naming system that end-nodes can use to store and resolve name-to-address bindings. However, the agent name system is more scalable and has a more uniform approach. NetBIOS makes a distinction between the different end-nodes that exist in the system. Nodes in NetBIOS can either be a broadcast node, point-to-point node, or a mixed node. Each of these nodes performs a different task in regards to name registration and resolution. For

different nodes, the name registration occurs in different places, and name resolution uses two different ways of finding name information. In the agent name system, there is no distinction made into what kind of node a machine is, and each name service resolves names and accepts updates in the same way. Also, there is no set way on how the NetBIOS Name Service (NBNS) can be implemented. A single NBNS may be implemented as a distributed entity, such as the Domain Name Service [32]. However, by not specifying how the NBNS works, it might be difficult to integrate NetBIOS services with the rest of the Internet. The agent name system uses the standard DNS concepts and formats, and agent name services can easily communicate with other name servers and hosts in the Internet.

In addition, the agent name system is more scalable than the NetBIOS system. First, there is a flat naming space in the NetBIOS system. There is a limit to how many names can be used, and name conflicts can be frequent. The agent name system uses the standard DNS dot notation for names, and this hierarchical naming scheme avoids conflicts and is scalable. Also, to register a name in the NetBIOS system, broadcasts sometimes have to be sent. For large networks, broadcasts tend to be inefficient and place a large load on the network. Broadcasts can also be sent to resolve names in the NetBIOS system. The agents in agent name system can redirect update messages to the right name service, and agents take advantage of the DNS tree structure to provide scalable name resolutions. The naming system used in NetBIOS works well for a small, specialized network, but for larger generic networks, the agent name system is a better way to provide name storage and resolutions.

## 4.8 – Comparison with AppleTalk

Like the agent name system, the AppleTalk network system has way of assigning names to machines and storing name-to-address bindings for machines in the system. However, the naming conventions used in AppleTalk is different from the ones used in the Internet, and the name storage and resolution scheme is not as scalable as the one used in the agent name system. Entities in AppleTalk take the following form: object:type@zone. This is different from the standard dotted notation used in the Internet today. Also, AppleTalk's Name Binding Protocol uses a different message format from the standard DNS format. The agent name system uses standard DNS protocols and message formats, and the agent system can be more easily integrated with systems in the Internet than AppleTalk systems can.

Also, AppleTalk's name storage and resolution scheme is not as scalable as the agent's scheme. AppleTalk uses a distributed database called the names directory to store the name information for a zone in the network. This names directory can reside on different machines, and if a machine has a copy of the names directory, then the machine will know about all the names in the zone. However, as the zone becomes large, it would be difficult to store all the names in one directory and to maintain consistency among multiple copies. The agent name system uses the properties of DNS to distribute name information among different servers. Also, a copy of the zone information in an AppleTalk system has to be on every router. It is difficult to maintain zone information consistency if many routers and zones exist in the system. The agent name system stores zone information in the same manner as it stores host name information.

Using broadcasts as a means of performing name lookups is inefficient compared to the resolution scheme used for the agent name system. For AppleTalk, if the name in question is in the same zone as the requester, a name lookup packet is broadcast in the network. Else, a directed broadcast is sent in the network zone of the name in question. Even though a broadcast is sent, the original requester only uses one answer. For the agent name system, there is a set of dedicated services used for answering name resolution requests. There is little wasted network traffic used. The AppleTalk system works well for specialized internal networks, but the agent name system is more scalable and can be more easily integrated with the Internet.

## 5 – Future Work

The agent name system works well in storing names and providing name-to-address resolutions in a scalable and fault-tolerant way. However, there are refinements and improvements that can be performed in the future. In the future, the agent system can be less dependent on *discover* messages, can use different root servers, can be extended for more functionality, and can have security features.

### 5.1 – *Discover* Messages

The first improvement is to make the agent name system less dependent on the *discover* messages sent by the managers. Currently, an agent program needs to hear a *discover* message for every other name service that exists in the system. For a very large network, this might not be possible. It would be more useful if an agent could function with only hearing a few *discover* messages. One fix for this problem has already been proposed. Every agent name service can automatically start out as an internal root server, and once the agent hears the existence of another agent name services, the agent name services can fight amongst themselves to resolve the root domain conflict or any other zone conflicts that might occur. Since every name service starts in the configured state, the change from a start-up state to configured state does not need to be propagated to other name services. In addition, the name services will properly configure even if the agent does not know about every other name service in the network. Implementing this scheme will save on the number of *discover* messages used.

Along the same lines, the conflict detection scheme for the zones assumes that an agent will receive a *discover* message for every newly joined name service in the system. One possible way to reduce the amount of *discover* traffic is to only have the root servers

receive *discover messages* and allow the root servers to perform conflict detection and resolution. In the future, the design of the conflict detection scheme and the configuration process can be refined to work more efficiently.

## **5.2 – Use of Internal Roots**

Another future work is to change the agent name system so that it does not use internal roots. For implementation and testing purposes, using internal root simplified much of the logic and naming for the system. In the future, the agent name system will take on a special domain name like *autonomous.company.com*, and the agent will control all the names in the domain. The basic idea of the network will not change, and the only difference is that root domain used internally now has a special name in the future. By allocating a new domain name within a organization, hosts from the outside world will be able to easily communicate with hosts and servers in the autonomous network.

## **5.3 – Extension of Name System**

The agent name system is just a basis for a name system. In the future, DNS extensions such as Mobile IP or a load-balancing web server functionally can be added onto the agent name system. By adding this functionality, the agent name system can be customized to better handle certain network environments. Things might be added onto the agent name system, but the foundation of the name system will remain the same. Also, it would be interesting to look at how cellular phone distribute identity information and apply some of those methods to the agent name system. A related future work would be deciding which real-world applications the autonomous network and agent name system are best suited for.

## **5.4 – Security Issues**

Finally, security issues have to be carefully considered and implemented in the future. In its current form, the agent name system does not worry much about security issues. It would be useful to devise a security procedure that determines which hosts can or cannot join the network. Also, IP spoofing can be a large problem. For example, if someone decides to spoof a root server, it could be disastrous to the integrity of the name system. Shared keys or other security measures can be used to authenticate users, and doing security checks will prevent spoofing and other attacks to the name server process. Also, Secure DNS can be used to for the name system. Secure DNS provides secure name updates and authenticates peer DNS servers. Also, agents have to be able to know which other agents are trusted in the system. In a military setting, these security issues are very important. The integrity of the connection is continually tested by enemy entities. A way to provide secure name services will be needed. These security issues are important and left for future development.

## **5.5 – Miscellaneous**

Some others issues that can be explored are using different transport protocol for delivering messages in the network. Right now, the TCP and UDP protocol is used for communication, but there might be more efficient protocols that can be used for the name system. Also, a fall back option can be devised in case the whole name system fails. For example, essential name information can be cached on a router in order to achieve extremely high availability.

## 6 – Appendix

### 6.1 – Agent Messages

The agent message format will consist of a header section and a payload field. The header section will have a message ID, Opcode, and a Send/Response Flag. These three fields will be delimited by a semicolon and be in plain text format. The payload section can be of variable size, and each field in the payload section will be delimited by a semicolon.

#### Server Record Format

Name
IP Address
Server ID
Status Flag

#### *Discover* Message

Msg_Id	number
Opcode	<i>discover</i>
Send/Response Flag	<i>Send</i>
Payload	Server Records of any newly discovered servers.

#### *Leave* Message

Msg_Id	number
Opcode	<i>leave</i>
Send/Response Flag	<i>Send</i>
Payload	Server Records of any lost servers.

#### *Getroot* Message

Msg_Id	number
Opcode	<i>getroot</i>
Send/Response Flag	<i>Send</i>
Payload	Empty

Msg_Id	number
Opcode	<i>getroot</i>
Send/Response Flag	<i>Response</i>

Payload	Server Records for the root servers in the system.
---------	--

*Becomeslave Message*

Msg_Id	number
Opcode	<i>becomeslave</i>
Send/Response Flag	<i>Send</i>
Payload	Zone Name the agent should be a slave for.
	Server Record for the master name service.
	Parent Name Service Information: Name and IP Address of Parent Name Service.

Msg_Id	number
Opcode	<i>becomeslave</i>
Send/Response Flag	<i>Response</i>
Payload	Zone Name the agent should be a slave for.
	Server Record for the slave name service.

*Negotiatemaster message*

Msg_Id	number
Opcode	<i>negotiatemaster</i>
Send/Response Flag	<i>Send</i>
Payload	Zone Name in conflict
	Number of Slaves the agent has for the zone.
	Server Record of the sending agent.

Msg_Id	number
Opcode	<i>negotiatemaster</i>
Send/Response Flag	<i>Response</i>
Payload	Zone Name in conflict
	Number of Slaves the agent has for the zone.
	Server Record of the responding agent.

### *Forceslave message*

Msg_Id	number
Opcode	<i>forceslave</i>
Send/Response Flag	<i>Send</i>
Payload	Zone Name the agent is forced be a slave for.
	Server Record for the master name service.
	Parent Name Service Information: Name and IP Address of Parent Name Service.

Msg_Id	number
Opcode	<i>forceslave</i>
Send/Response Flag	<i>Response</i>
Payload	Zone Name the agent should be a slave for.
	Server Records for all the slave servers the agent used to be the master for.

## **6.2 – Sample *named* Configuration Files**

### **Sample *named.conf* file**

```
options {
    directory "/home/feng/dns_agent/dbfiles/";
    notify yes;

    check-names master warn;
    check-names slave warn;
    check-names response ignore;

    listen-on port 55 {129.83.47.17; };
    pid-file "/var/run/named.pid";
};

zone "."                               in {
    type master;
    allow-update {129.83.47/24; };
    file "db.root";
};

zone "green"                           in {
    type master;
    allow-update {129.83.47/24; };
```

```

    file "db.green";
};

zone "0.0.127.in-addr.arpa" in {
    type master;
    file "db.127.0.0";
};

zone "17.47.83.129.in-addr.arpa."    in {
    type master;
    file "db.129.83.47.17";
};

```

### **Sample db.root file**

```

. 86400 IN SOA nameservice hostmaster (
    7          ; serial number
    120       ; refresh time
    60        ; retry time
    604800   ; expire time
    86400 )  ; minimum time-to-live

. IN NS nameservice
nameservice IN A 192.168.20.1

```

### **Sample db.cache file**

```

. 3600000 IN NS A.ROOT-SERVERS.NET.
A.ROOT-SERVERS.NET. 3600000 A 198.41.0.4

. 3600000 NS B.ROOT-SERVERS.NET.
B.ROOT-SERVERS.NET. 3600000 A 129.9.0.107

. 3600000 NS C.ROOT-SERVERS.NET.
C.ROOT-SERVERS.NET. 3600000 A 192.33.4.12

. 3600000 NS D.ROOT-SERVERS.NET.
D.ROOT-SERVERS.NET. 3600000 A 128.8.10.90

```

### **Sample db.org file**

```

org. 86400 IN SOA nameservice hostmaster.org. (
    1          ; serial number
    120       ; refresh time
    60        ; retry time
    604800   ; expire time
    86400 )  ; minimum time-to-live

org. IN NS nameservice
nameservice IN A 192.128.14.3

```

## 7 – Bibliography

- [1] W. Adje-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley, “The design and implementation of an intentional naming system”, *Operating Systems Review*, vol. 35, no. 5, pp. 186-201, December 1999.
- [2] P. Albitz, C. Liu, *DNS and Bind*, 3<sup>rd</sup> Edition, O’Reilly, 1998.
- [3] *AppleTalk Network System Overview*, Addison-Wesley Publishing Company, Inc., 1990.
- [4] R. Baldoni, S. Bonamoneta, C. Marchetti, “Implementing Highly-Available WWW Servers based on Passive Object Replication”, *Proceedings of 2<sup>nd</sup> IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 259-262, 1999.
- [5] A.D. Birrell, R. Lvein, R.M. Needham, M.D. Schroeder, “Grapevine: An Exercise in Distributed Computing”, *Communications of the ACM*, vol. 25, no. 4, pp. 260-274, April 1982.
- [6] V. Cardellini, M. Colajanni, P. Yu, “Efficient State Estimators for Load Control Policies in Scalable Web Server Clusters”, *Proc. of 22<sup>nd</sup> Int. Conf. on Computer Software and Applications (COMPSAC)*, pp. 449-455, 1998.
- [7] V. Cardellini, M. Colajanni, P. Yu, “Redirection Algorithms for Load Sharing in Distributed Web-server Systems”, *Proc. of 19<sup>th</sup> IEEE Int. Conf. on Distributed Computing Systems*, pp. 528-535, 1999.
- [8] H. Chao, T.Y. Wu, S.W. Chang, R. Wang, “The Network Topology Based Domain Name Service”, *Proc. of Int. Workshops on Parallel Processing*, pp. 214-219, 1999.
- [9] R. Droms, “Dynamic Host Configuration Protocol”, RFC 2131, March 1997.
- [10] “Dynamic DNS, Infrastructure essential for today’s intranets”, <http://www.nts.com/collateral/ddnstechpaper.pdf>
- [11] D. Eastlake, “Secure Domain Name System Dynamic Update”, RFC 2137, April 1997.
- [12] C.S. Giap, Y. Kadobayashi, S. Yamaguchi, “Zero Internet Administration Approach: the case of DNS”, *Proc. of 12<sup>th</sup> Int. Conf. on Information Networking (ICOIN)*, pp. 350-355, 1998.
- [13] M. Hattig, ed., *Zeroconf Requirements*, draft-ietf-zeroconf-reqts-03.txt, March 2000. (work in progress)

- [14] D.B. Johnson, "Scalable Support for Transparent Mobile Host Internetworking", *Mobile Computing*, ch. 3, 1996.
- [15] "JOIN DDNS", <http://www.join.com/ddns.html>
- [16] J. Kangasharju, K. Ross, "A Replicated Architecture for the Domain Name System", *Proc. of 19<sup>th</sup> Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 660-669, 2000.
- [17] T.T. Kwan, R.E. McGrath, D.A. Reed, "NCSA's World Wide Web server: Design and Performance", *IEEE Computer*, vol 28, no. 11, pp. 68-74, November 1995.
- [18] Y. Lin and M. Gerla, "Induction and Deduction for Autonomous Network", *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1415-1425, December 1993.
- [19] "Lucent QIP Enterprise 5.0: Automating IP Services Management", [http://www.qip.lucent.com/products/qipent\\_6093.pdf](http://www.qip.lucent.com/products/qipent_6093.pdf)
- [20] "Meta IP Technical White Paper", <http://www.checkpoint.com/products/metaip/whitepaper.pdf>
- [21] P.V. Mockapetris, "Domain Names – Concepts and Facilities", RFC 1034, November 1987.
- [22] P.V. Mockapetris, "Domains Names – Implementation and Specification", RFC 1035 November 1987.
- [23] "Network Management: NetID 4.1", <http://www.nortelnetworks.com/products/02/datasheets/2894.pdf>
- [24] N. Nuansri, T. Dillon, S. Singh, "An Application of Neural Network and Rule-Based System for Network Management: Application Level Problems", *Proc. of 30<sup>th</sup> Hawaii Int. Conf. on System Sciences*, vol 5, pp. 474-483, 1997.
- [25] C. Oppen, Y. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment", *ACM Transactions on Orrice Information Systems*, vol. 1, no. 3, pp. 230-253, July 1983.
- [26] C. Park, et.al., "The Improvement for Integrity between DHCP and DNS", *High Performance Computing on the Information Superhighway*, pp. 511-516, 1997.
- [27] C. Perkins, ed., "IP Mobility Support", RFC 2002, October 1996.

- [28] C.E. Perkins, "Mobile networking in the Internet", *Mobile Networks and Applications*, vol. 3, pp. 319-334, 1998.
- [29] C.E. Perkins, H. Harjono, "Resource discovery protocol for mobile computing", *Mobile Networks and Applications*, vol. 1, pp. 447-455, 1998.
- [30] C.E. Perkins, K. Luo, "Using DHCP with computers that move", *Wireless Networks*, vol. 1, pp. 341-353, 1995.
- [31] L. Peterson, "The Profile Naming Service", *ACM Transactions on Computer Systems*, vol. 6, no. 4, pp. 341-364, November 1988.
- [32] "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods", RFC 1001, March 1987.
- [33] "Shadow IPserver", <http://www.nts.com/collateral/ipserverdatasheet.pdf>
- [34] Y. Shim, H. Shim, M. Lee, O. Byeon, "Extension and Design of Secure Dynamic Updates in Domain Name Systems", *5<sup>th</sup> Asia-Pacific Conference on Communications (APCC)*, vol. 2, pp. 1147-1150, 1998.
- [35] G.S. Sidhu, R.F. Andrews, A.B. Oppenheimer, *Inside Appletalk*, Second Edition, Addison-Wesley Publishing Company, Inc., 1990.
- [36] R.D. Verjinski, "PHASE, A Portable Host Access System Environment", *IEEE Military Communications Conference Record (MILCOM)*, vol.3, pp. 806-809, May 1989.
- [37] P. Vixie, et.al., "Dynamic Updates in the Domain Name System (DNS Update)", RFC 2136, April 1997.
- [38] "Zero Configuration Networking (zeroconf) Charter", <http://www.ietf.org/html.charters/zeroconf-charter.html>