MIT Sloan School of Management

Sloan Working Paper 4213-01
August 2001

A SELF-CONFIGURING AND SELF-ADMINISTERING NAME SYSTEM WITH DYNAMIC ADDRESS ASSIGNMENT

Paul Huck, Michael Butler, Amar Gupta, Michael Feng

A Self-Configuring and Self-Administering Name System with Dynamic Address Assignment

Paul Huck
Oracle Corporation

Michael Butler
MITRE Corporation

Amar Gupta
Massachusetts Institute of Technology

Michael Feng
Oracle Corporation

Abstract

A distributed system that stores name-to-address bindings and provides name resolution to a network of computers is presented in this paper. This name system consists of a network of name services that are individually self-configuring and self-administering. The name service consists of an agent program that works in conjunction with the current implementation of the Domain Name System (DNS). The DNS agent program automatically configures a Berkeley Internet Name Domain (BIND) process during start up and dynamically reconfigures and administers the BIND process based on the changing state of the network. The proposed name system offers high scalability and fault-tolerance capabilities and communicates using standard Internet protocols.

1 Introduction

As we approach the close of their second decade of use, the underlying technology of the Internet, specifically the TCP/IP protocol suite, has shown a tremendous resilience that attests to the framers' foresight. Their achievement is even more remarkable when one considers that this architecture was conceived at a time when there were remarkably few network hosts, very low bandwidth links, and relatively few types of digital transactions. Computer mobility was unheard of, and corporations and campuses were generally centralized at a single geographic location.

Today, the Internet is rapidly approaching a billion hosts; link bandwidths vary from paltry hundred bits per second to gigabits per second; links carry real-time voice, video, and data traffic including financial, medical, criminal records, which place demands on privacy as well as performance; host mobility is commonplace, and organizations are now geographically distributed are connected by virtual private networks. With the advent of "Internet Ready" embedded processors, we can expect these increases in size, diversity, and fluidity of the Internet to continue.

The underlying architectural concepts of the Internet are as sound now as ever, but weaknesses (growing pains?) are to be expected as the Internet continues to evolve. One such area involves address space management in the presence of host mobility, micro networks (e.g., home networks), and network reconfiguration. We begin the discussion with a brief historical perspective on address management, recap the state of things today, and explore an alternative for the future.

In the infancy stages of the Internet, address space management was deceptively simple. It was envisioned that all interfaces on all hosts would have unique, permanently assigned IP addresses[1]. Thus, large blocks of addresses were assigned to an institution or a corporation to administer as they saw fit. The size of the allocated block was determined by the organizations expected needs and fell into three classes, A, B, and C. This approach worked exceptionally well since it distributed the management of the four billion available addresses among the participating organizations giving them both the responsibility and authority for managing a portion of the address space. Although such allocations were wasteful[2], the perception was that the available 32-bit address space was inconceivably large.

A few years ago this inefficient management of address space lead to a short-term crisis; it appeared that the then "inconceivably large" address space was being rapidly exhausted. One of the solutions, IPv6, offered a presently "inconceivably large" 128-bit address space as an alternative. Although several IPv6 implementations are now available, IPv6 has not—and in fact might never—displace IPv4 and its paltry address space.

One of the ways that the "address space crisis" in IPv4 was averted was the rethinking of one of IP's initial assumptions—the desire for a globally unique, permanent address assignment for every host interface. It is arguable whether this change in assumptions was a conscious effort or it occurred accidentally, but address space management changed fundamentally with the introduction of DHCP, masquerading firewalls, and VPNs. We now routinely violate the permanency and even the uniqueness of address assignment by recycling IP addresses both temporally and spatially.

Such transient assignments have, however, exposed some undesirable artifacts in other protocols; specifically, TCP's concept of a connection is integrally coupled to the invariance of source and destination address as presented in the packet header. Hence, the loss of a DHCP lease (e.g., by a dropped PPP connection) effectively severs any open TCP connections. Mobile-IP is, in large part, a work around for TCP's dependence on address permanence. It offers TCP the illusion of a permanent IP address while simultaneously providing routing changes based on the changes in IP address. (With MobileIP, Mobile hosts retain their permanent addresses but their packets are transiently tunneled to the "care of" address of a mobile agent at their point of attachment to the network.)

---

[1] Excepting experimental subnets in the 192.168.x.y address space

[2] A Class A address allocation contains 16 million host addresses

Dynamic address assignment is, however, an exceedingly powerful technique that offers solutions to not only the shrinking address space problem, but the larger address space management and network configuration problems as well. Route aggregation, for example, could be optimized if address reassignment were possible. ATM (Asynchronous Transfer Mode) protocols took such dynamic address assignment to the logical conclusion making all interface addresses locally assigned and explicitly supporting the concept of trunking, and thereby avoiding the address space problem in its entirety.

One might ask why the concepts of interface addresses and host names are both preserved. The two techniques are complementary and address different problems. Internet addresses, the 32-bit values that appear in IP packets, provide a convenient, fixed-size, short hand indication of a packet's destination and, therefore, prescribe its migration through the network. When the address space is carefully administered, addresses also provide sufficient information for route aggregation or "trunking", routing traffic based on only a few bits of the address, which greatly reduces the size of routing tables in both hosts and routers. This shorthand representation is compact and very convenient for machine processing.

However, numeric addresses, whether permanently or transiently assigned, are an exceptionally inconvenient representation for humans. Dynamically assigned addresses only exacerbate the problem. Host names provide much better mnemonics—as well as a much larger total address space and the opportunity for many-to-many name-to-address mappings as well as dynamic address assignment.

Presently in the Internet, dynamic address assignment has been applied primarily to end-user machines using either DHCP or Mobile IP, and IP masquerading is used in firewalls at the enterprise boundaries. Virtual servers (where a single virtual URL maps to one of several physical hosts) offer another example of dynamic address binding, though the implementation differs in details. Although such uses have shown the power of dynamic address binding, we still do not exploit dynamic address binding to obviate address management in network design.

Serendipitously, the early inclusion of this mnemonic "crutch" which allowed named hosts has also provided the cornerstone for dynamic address assignment. The hierarchically organized DNS servers universally accomplish the mapping between the large and mnemonic name space and the smaller and compact addresses space. In order to reap the benefits of a more dynamically assigned address space; we will need a more fluid name-to-address mapping than in presently available.

This paper addresses the key issues associated with modifying DNS to enable more dynamic address assignment.

2 Overview

The Autonomous Network Management initiative attempts to develop a collection of protocols and software tools that will enable networks to be rapidly configured and maintained by untrained personnel. Industry efforts such as DHCP and the IETF's ZeroConf Working Group have attempted to address some issues related to automated configuration, but these are focused exclusively on providing a solution for endhosts and they ignore other network components. The approach taken by us to automate the network planning and management task splits the problem into four inter-related subtasks: address assignment; host name registration; network augmentation; and resource redistribution for load balancing. In this paper, we focus on the first two aspects. Our approach focuses on providing dynamic interface address assignment for all network assets, so that the network components can themselves efficiently renegotiate addresses each time the network is extended. To facilitate the latter, a more flexible DNS hostname to address binding scheme must be incorporated.

For more than a decade, the Berkeley Internet Name Domain (BIND) has been the de facto standard for DNS implementations [28]. Nearly all modern networks use the public domain BIND implementation or commercial products derived from BIND. In fact, BIND has become virtually synonymous with the industry standards that define the DNS architecture. The DNS specification was written to allow product interoperability and thus permit and encourage flexibility in each DNS implementation. Thus, it is possible to develop new DNS implementations with added features, while remaining compliant with existing standards.

In particular, it would be desirable if a DNS service could be run with no lengthy configuration process and no need for user intervention after it has started. The current implementation of BIND requires a network administrator to write several configuration files in order to function correctly. These configuration files tell BIND the zone it is responsible for, the location of other DNS servers on the network, and the location of the root name server. Setting up this configuration can be a tedious process, as every time a new host is added or removed from the network, these files must be updated. In addition, if the network topology changes (such as by adding or removing name servers or by merging two networks together), significant changes are required to the configuration of BIND as the DNS zones and ownership of the zones may have changed too. This is certainly not ideal for home networks or networks consisting of mobile or wireless clients. In a home-networked environment, hosts are frequently leaving and entering the network as appliances are turned on and off, or as new hosts are added. Mobile clients also will join and leave networks often as they are transported across network boundaries. In an ideal environment, hosts should be free to join and leave the network, and should immediately gain access to the services; this requires that the DNS be able to configure itself on a dynamic basis.

3 DNS

This section describes the hierarchical naming system and distributed architecture of the current DNS standard. It also examines the operations involved for performing updates to the network.

DNS was developed in 1984 by Paul Mockapetris as a distributed database that could resolve the address of any computer on the network [22]. It was created to replace the ASCII host files that associated host names with their respective IP addresses. These host files resided on every host in the network, and if a name was not listed in the file, that host could not be reached. As the Internet grew to become a worldwide network, the process of maintaining the host files became increasingly unwieldy, leading to a growing need for the DNS.

The DNS directory can be thought of as a tree, with each node on the tree representing a "domain" [1]. Each domain is identified and located in the tree by its domain name, which uses the familiar dotted notation (www.mit.edu, for example.). As we read the domain name from right to left, we can follow the path down the DNS tree to the correct node (See Figure 1). The "edu" domain is one of many top-level domains; others include "com", "gov", "org", and "uk". The full and unique domain name of any node in the tree is the sequence of labels on the path from that node to the root.

The hierarchical tree of domain names can be referred to as the domain name space. Each domain in the space has host data associated with it. These host data are defined in resource records. There are many different types of resource records, the most common being the address-record (A-record), which associates the domain name with an IP address.

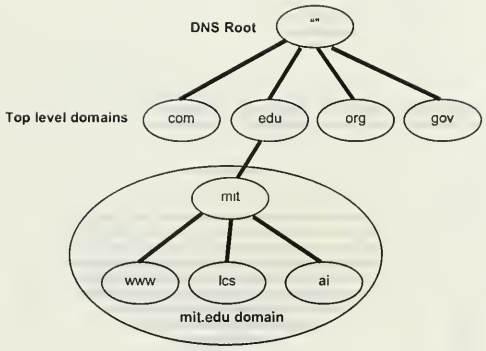


Figure 1 - DNS Hierarchy

The other distinguishing characteristic of the DNS architecture is its distributed implementation. DNS servers may be operated by any organization that owns a domain. Each DNS server is given authority for one or more "zones". A zone is a collection of one or more DNS domains that share the same parent domain, along with the associated resource records. A DNS server receives authority over a zone when the network manager responsible for the domain that contains that zone delegates the authority to that particular DNS server. Therefore, the DNS infrastructure is distributed both geographically and administratively [28].

3.1 Name Servers

Each zone in the domain name space has at least two name servers authoritative for it, a primary one and a secondary one. Authoritative name servers are the only name servers guaranteed to contain the correct resource records for their zone. When querying a name on the Internet, a resolver can only be assured that the address is correct if the answer comes from an authoritative name server for the zone that is being queried (See Figure 2). To improve performance, other name servers may cache resource records, but each cached entry has time-to-live to prevent staleness of data. A name server stores all the resource records for the zone for which it is authoritative. The primary name server is an authoritative server for its zone(s) and reads its zone data from the zone files [1]. When changes are made to the zone's resource records, they must be made to the primary's zone file. This data is sent to secondary name servers upon request. A secondary name server is also authoritative for its zone(s); however, it obtains its zone data via data transfers from other name servers. A secondary name server interrogates the primary or other secondary servers periodically to determine if its zone's data have changed. A network administrator can set the period of this interrogation. A single name server can act as both the primary and secondary server for two or more different zones.

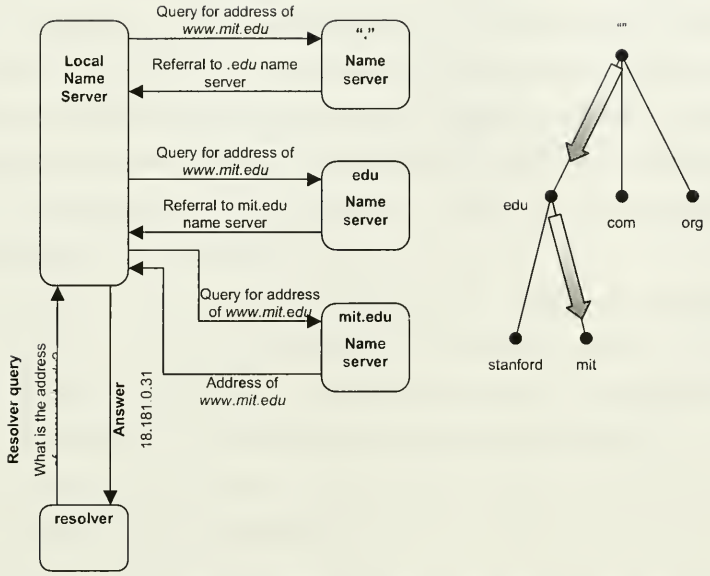


Figure 2 - Resolution of www.mit.edu through DNS (from [12])

3.2 Dynamic DNS and Zone Transfers

The original DNS specification was written with static networks in mind. It was assumed that hosts would join and leave the network infrequently. However, in modern networks, computers are free to join and leave the network, and many new devices are being connected. In order to deal with dynamically changing networks, several extensions to DNS have been implemented. Specifically, RFC 2136 [34] defines the DNS Update protocol that allows for dynamic updates to the DNS. A dynamic update is a deletion or addition of resource records to a zone. It can be sent by a DNS client, DNS server, or a DHCP server (see Section 4). The *update* signal is sent to the primary name server, which receives the signal and permanently updates its zone file. The signal may be sent to the primary server directly or passed through one or more secondary servers until it reaches the primary. When a primary server fulfills an *update* request, it can use the *notify* signal to inform its secondary servers that the zone information has changed.

In order for multiple name servers to maintain consistency of their records, zone transfers are performed on a periodic basis. A zone transfer is the transfer of resource records from a primary name server to a secondary name server. A full zone transfer occurs when all resource records are sent. Instead of sending all the resource records, it is possible for the primary name server to perform an incremental zone transfer. This will transfer only those records updated since the last zone transfer. A secondary name server requests an incremental zone transfer and a primary server chooses whether it will perform a full zone transfer or an incremental one. It is recommended that full zone transfers be performed no more than once every 15 minutes and at least once every 24 hours [3].

The following is a brief example of how the current version of BIND propagates changes to the resource records to all authoritative name servers. The process is illustrated in Figure 3.

1. The primary server receives a request for the addition or deletion of a host. This may come from administrator manually editing the zone file, or through an *update* message received. If the *update* message is received at a secondary server, it will pass it to the primary server if it knows its location, or to another secondary server. This is continued until the primary server receives the *update* signal. Once the zone file is changed, the serial number of the file is incremented.

2. The primary name server reads the edited zone file. The frequency at which the server rereads its zone file and checks for zone changes is a configurable parameter of BIND.

3. The primary server will send a *notify* message to all known secondary servers. The primary server will wait some time between sending each *notify* to reduce the chance of multiple secondary servers requesting zone transfers at the same time.

4. If the secondary server(s) support(s) the *notify* signal, a zone transfer is immediately initiated. Otherwise, the secondary server will discard the *notify* and wait until the next scheduled zone transfer time.

5. The secondary server then notifies any other secondary servers that may be dependent on it for zone transfers. This multi-level transfer is continued until all secondary servers have received the changed records.

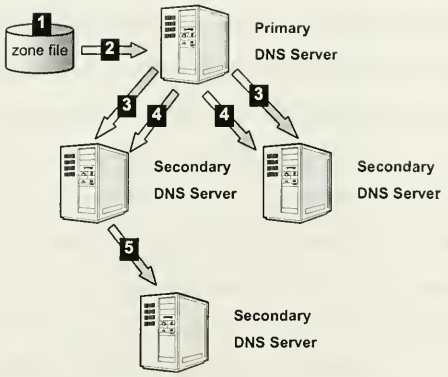


Figure 3 - BIND Updates (from [28])

While the dynamic update system may lessen the amount of administrative work for the name servers, it does not make them administrator free. Each name server in the network must be configured with the address of either the primary DNS server or other secondary DNS servers for its zone. Name servers are not free to join and leave the network. If a secondary name server is added to the network, either the primary DNS server or other secondary DNS must be configured to recognize the presence of the new secondary name server so that the primary DNS server can receive *updates* properly. Also, if the primary name server is removed and another added in its place, an administrator must manually change the configuration of every secondary server so that it is informed of the new primary DNS server. In addition, the current DNS system requires extensive initialization effort to ensure proper operation. Zones must be properly allocated with specific primary and secondary servers and a clear domain hierarchy must be defined. These tasks all require extensive knowledge and effort from an administrator.

The above problems are addressed by the system described in the next section. It uses a combination of an agent program monitoring BIND and dynamic DNS protocols to maintain a true self-configuring and self-administrating naming system.

4 Self-Configuring and Self-Administering Name System

4.1 Network Overview

The naming system described in this section was conceived as part of a larger network system that is under development at a company codenamed as Research Corporation in this document. The proposed design provides a full array of network services to the hosts on the network. This is done with virtually no manual configuration and no administration. The network design allows for a group

of computers to be physically connected together (through Ethernet or other network media) and after a short time, they will all be properly configured in a working network. On this network there will be two types of nodes: managers and hosts (See Figure 3 and Figure 4). Managers form the backbone of the network and provide services, such as name to address resolution, to the hosts. Besides the name to address resolution service, the managers also provide dynamic IP configuration for new hosts, discovery mechanisms for new managers, packet routing, and a database of all hosts in the system. The services directly related to the naming service are automatic IP configuration and manager discovery, as these will be the services that the name system will directly interact with. The goal of the network system is to allow any computer, either a host or a manager, the freedom to join or leave the network, without the need for any external administration. The network should be able to detect the presence of a new node or the absence of an existing one, and deal with either type of change in an appropriate manner. In addition, the network system, and in particular the name system, should deal gracefully with the merging of two networks. Conflicts should be detected and resolved quickly.

In order to provide the self-configuring and self-administrating name service, each manager in the system is designed to run three types of services: the IP configuration service, the manager discovery service, and the agent based name service. The IP

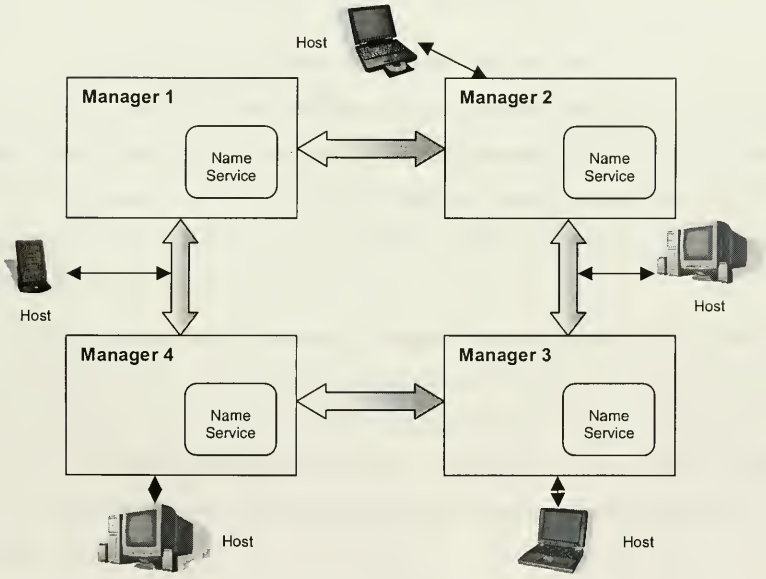


Figure 4 - Sample Network

configuration service is handled by a standard implementation of the dynamic host configuration protocol (DHCP). DHCP is a network protocol specified by the IETF that provides hosts with configuration parameters for their IP interface [6]. This includes an IP address, a domain name, a subnet mask, a default gateway, and a location of DNS server. DHCP also allows a host to retain a previously

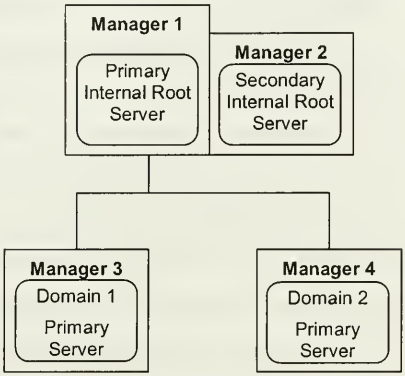


Figure 5-Name Space View of Systems shown in Figure 4

configured domain name while receiving an IP address. After receiving the necessary information from a DHCP server, a previously unconfigured host's IP interface has all the necessary parameters in order to begin transmitting and receiving on the network. DHCP requires no prior configuration; as a host locates a DHCP server by broadcasting discover messages on the local network. All modern operating systems and even most embedded devices support DHCP. Overall, DHCP meets the goals and design objectives of the desired self-configuring and self-administrating network.

The process of manager discovery is accomplished by having the manager discovery process periodically broadcast "discover packets" to each interface. These packets contain the source address and a unique network number that the manager resides on. All other manager discovery processes will receive the packet, and if the network number is known, the packet will be dropped. If, however, the network number is unknown, that manager will respond to the source of the "discover" with a "discover reply". This reply includes the network number for the new manager. This allows the manager discovery process to inform the local name service of any new managers that appear on the network.

The design and the operation of the name service are discussed in the next subsection.

4.2 Name Service Interface

The name service runs on every manager in the system and consists of two concurrently running processes. The first is the BIND implementation of DNS. As stated before, BIND is currently used in the overwhelming majority of name servers on the Internet. It provides a scalable, stable, and fault-tolerant basis for the name service. The second process is an agent program that reacts to network conditions and configures BIND automatically. Figure 6 shows the relationship and manner of communication between BIND, the agent program, the other local manager processes, as well as other managers in the network. The agent program uses Berkeley UDP sockets to listen for two different formats of messages. On port 53, the standard DNS port, the agent program listens for DNS messages. It acts as a filter for the DNS messages, sending queries directly to the BIND process, and processing update messages from the IP configuration process. On port 54, the agent program listens for any agent messages coming from either the manager discovery service, or other agents in the system. The agent messages allow an agent process to gain knowledge of other agents and offer a method of communication between agents. The IETF has designated port 54 to be used for XNS Clearinghouse [27]. We chose to use port 54 because XNS is uncommon on modern servers. If this service is needed, another port may be specified for agent communication. The details of both DNS and agent message processing are discussed in following sections.

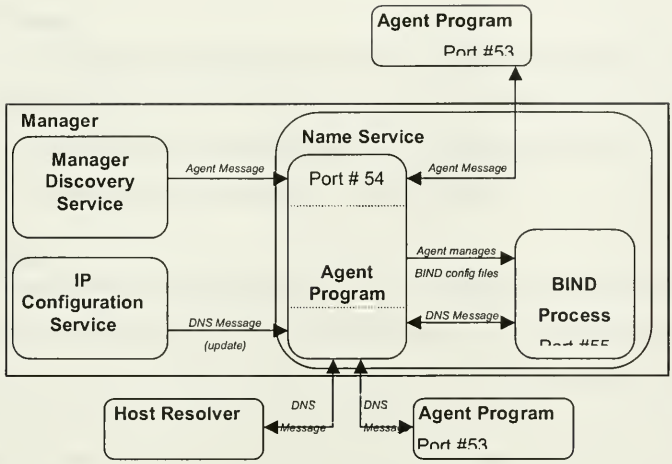


Figure 6 - Relationship between agent, BIND, and other processes (from [12])

4.3 Internal Root Servers

The name servers in the system are only authoritative for the IP addresses of the subset of hosts configured by the naming system. If the network is connected to a larger network, such as the Internet or other networks serviced by the naming system, DNS servers outside the local network must be queried. 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 internal root servers and are authoritative for the entire domain that is serviced by the self-configuring naming system. These are not true DNS root servers as they are not authoritative for every domain name on the Internet, but only for those that exist inside the domain serviced by the self-configuring naming service. So, as long as a name service knows the location of at least one internal root server, the name service will be able to provide name resolution for the network's name space. For example, if the entire mit.edu domain is using the self-configuring naming system, a host in the lcs.mit.edu sub-domain may wish to know the IP address of a computer in another sub-domain serviced by the naming system (such as media.mit.edu). This host will query any local name system, and since the manager is not authoritative for the media.mit.edu domain, it will query the internal root server of the mit.edu domain. This server may resolve any domain name in the mit.edu domain by directing the resolver to the proper media.mit.edu authoritative name server. For resolving names outside the domain, such as Internet host names, the Internet root servers may be queried.

4.4 DNS Messages

The only DNS messages that the agent program processes directly are DNS updates. All other messages are simply directed to the local BIND process on port 55. These messages will simply be DNS queries and BIND will process the query, return an answer to the agent process, which in turn will forward the response to the inquirer. The agent process is the only entity with access to the BIND process. To all hosts on the network, it appears as if BIND is running on the normal port.

4.4.1 DNS Update

DNS updates, however, require special processing by the agent. DNS updates only occur when the IP configuration service or manager discovery service detects the addition or deletion of a host or manager on the network. The respective local manager service then assembles the appropriate DNS update message containing the resource record that must be added, modified, or deleted and sends it to the agent program. Although DNS updates may be sent to any DNS server that is authoritative for a zone, they will always end up being processed by the primary master for the zone, as that is the only server that has the definitive set of resource records for a particular zone. As stated earlier, standard DNS implementations specify that any secondary server which receives a DNS update message must forward it directly to the primary master if its location is known, or alternatively through the secondary server chain until it reaches the primary master [34]. The secondary servers will only learn of the update through *notify* signals originating from the primary master.

To try to improve the efficiency of this mechanism, the agent program will examine the update message, and if it applies to RR in the zone that it is authoritative for, it will send it directly the primary master for that zone. Each agent process has knowledge of its primary master as this is part of its state information described in Section 4.6. When the agent program is not authoritative for the zone receiving the update, then it must search for the primary master for that zone on the network. To do this, the agent program will send a Start of Authority (SOA) DNS query to the local BIND process. The SOA asks the DNS for the address of the primary master for a particular zone. Upon receipt of the SOA, BIND will perform standard DNS resolution to find the information on the zone name. If the zone exists, BIND will return to the agent process the IP address of the primary master for the requested zone, and the agent process will forward the *update* to that address. If, however, the DNS update message applies to a zone that does not exist on the network, BIND will return the non-existent domain error flag (NX_DOMAIN), and the agent program will configure the local BIND process to be authoritative for the new zone. The reconfigured BIND process will then process the update request. This allows for dynamic creation of zones and is especially useful at resolving all naming conflicts when two networks are merged.

4.5 Agent Messages

Agent messages are used for inter-agent communication and as a mechanism for agent discovery. Figure 7 shows the structure of an agent message. It consists of a three-field header followed by a payload section. The header fields are explained in Table **1**. The payload holds the data from the message. The data are specific to each Opcode. The data sent with each agent message are explained in more detail in the following sections.

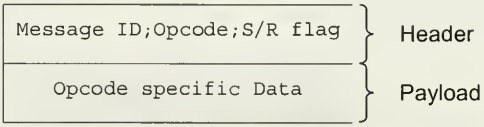


Figure 7 - Agent Message Format

Table 1 – Agent Message Header Fields

Header Field	Description
Message ID	The unique message ID for the message
Opcode	The operation code for message. Can have one of six values; see **Table 2**
S/R flag	Send/Response Flag. A flag indicating whether the message is a send request or response to an earlier message

The Opcode may take on one of six values, which are described in Table 2. Currently the header is formatted as ASCII text separated by semicolons. A sample agent message header is shown here:

```
3;getroot;response
```

The total length of this header will be 18 bytes (one byte for each character). However, it is possible to represent each field with a more efficient binary representation. If the message ID were encoded as a 12 bit binary number (allowing for 2^{12} unique IDs), the opcode as a 3 bit number, and the S/R flag as a one bit field, the header size could be reduced to two bytes. The case of ASCII text was chosen to simplify the implementation and debugging of the system.

Table 2 - Opcode Values

Opcode	Brief Description
Discover	Informs agent of new managers
Leave	Informs agent of lost managers
getroot	Used to request the location of internal root server
becomeslave	Sent to an agent to make it a slave to the sender
negotiatemaster	Used to resolve primary master conflicts
forceslave	Sent by the winner of two conflicting servers, to force the loser into becoming a slave

4.5.1 Agent Message Behavior

The manager discovery process will send the local agent process *discover* and *leave* messages whenever it discovers that a manager has joined or left the system, respectively. These messages include the information about the new or lost server such as the zone it is authoritative for and also whether it was a master or a slave for that zone. They allow an agent to gain a current view of the network topology and can also help to warn against conflicts and errors arising in the network. For example, if a manager that was a primary master for a zone disappeared, a slave for that zone would receive the appropriate *leave* message and negotiate with its peers to elect a new primary server for that zone.

The *negotiatemaster* and *forceslave* messages are designed to be used in such situations. The *negotiatemaster* is used when two managers discover that they are primary masters for the same zone. This may happen when two previously unconnected networks are physically joined. The two agents will exchange *negotiatemaster* messages and elect a new master. The *negotiatemaster* message includes metrics to help determine the optimal master, such as the number of slaves the server currently has. The winner of the election then sends a *forceslave* to the loser and requests the latter's zone data, so that such data may be merged with the existing data. The merge is accomplished via a zone transfer from the loser to the winner. This assumes that all hosts have unique names, which is reasonable assumption based on ongoing developments stated in [34]. An example is shown in Figures 8 (a) thru (e).

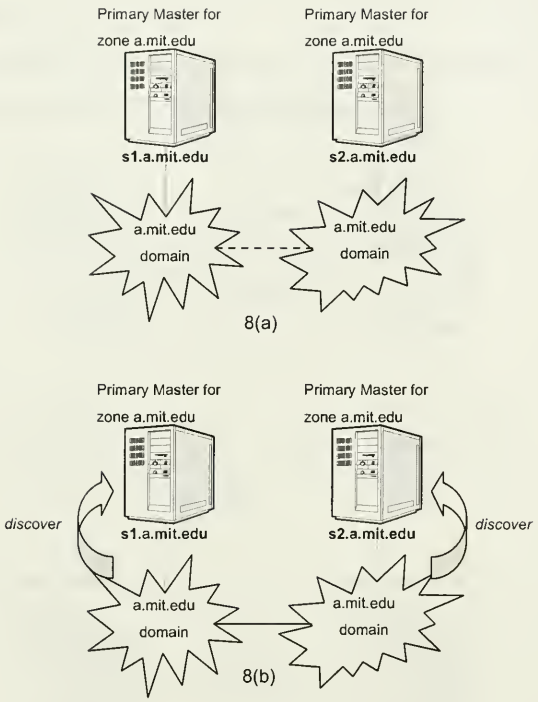


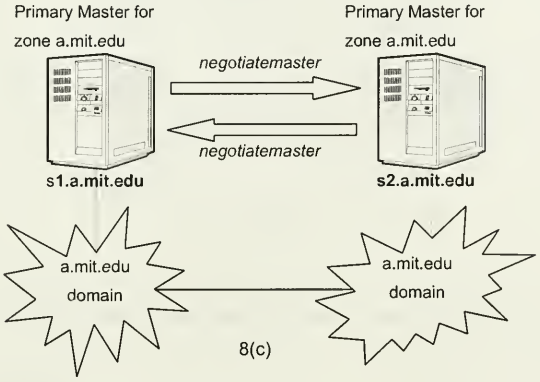
Figure 8(a) - Merging of two previously unconnected networks with name conflicts begin with new physical connection joining the two.

Figure 8(b) - Agent programs in both networks receive *discover* messages informing them of the newly discovered managers.

The *becomeslave* message can also be used to make an agent a slave for a zone. However, while the *forceslave* message is only used between two competing primary servers, the *becomeslave* can be sent to any agent in the system. It is not used for elections, but only to notify agents of new primary servers. If a new primary server for a zone leaves the network and a new one is elected, the *becomeslave* message is passed to all the old slaves of the former primary server and to the slaves of the loser in the election. The *becomeslave* message informs them of the new primary server and allows the agents to reconfigure themselves accordingly.

The *getroot* message is used by agents to share the internal root server information. This is useful when a new unconfigured agent is introduced to the system and wishes to know the internal root server. The *getroot* function operates on the basis that any

configured DNS server that is queried with the *getroot* message will have a reference to a root server; this root server can be another server or the root server itself. This assumption is valid because the server is either authoritative or not authoritative for the particular root. In the former case, the authoritative root server will simply return its own information to the querying server. In the other case, it will have a root server reference that will be used to query for the relevant zone information.



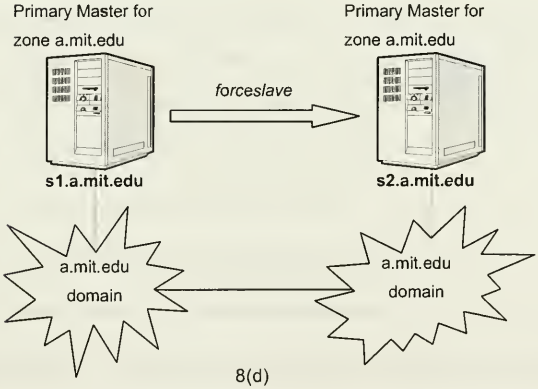


Figure 8(c) - The conflicting primary master servers send each other *negotiatemaster* messages.

Figure 8(d) - The winner of the election sends a *forceslave* message to the loser.

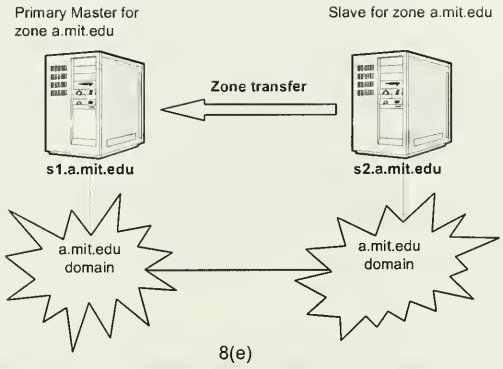


Figure 8(e) -The new slave performs a zone transfer with the master.

4.6 State Information

To assist with the agent tasks, each agent stores state information about itself and other managers in the network, as well as a log of all messages it has received. In particular, the agent will keep a record for all managers that it knows to exist on the network. This *server record* contains the name, IP address, a unique server ID, and a status flag that tells if the server is configured or just starting up (See Table 3). Each server record is placed into one or more of the following categories:

- Own Servers - contains the agent program's own server information.
- Known Servers - list of server records for every manager on the network.
- Root Servers - server records for every domain root server on the network
- Newly Discovered Servers - list of recently discovered managers. Once the agent processes the *discover* message, the server record will be moved to one of the above categories.

Table 3 - Server Record Fields

Server Record Field	Description
Name	Name of the manager the name service resides on.
IP Address	IP address of the manager the name service resides on
Server ID	Unique ID of the manager the name service resides on. The unique ID may be derived from the MAC address of the network interface in use by the manager.
Status Flag	*Configured:* name service is configured with the location of an internal root server. *Start-up:* name service is unaware of a location of an internal root server.

In addition to keeping records for every manager, each agent also keeps information on every zone it is authoritative for. If the server is a slave in that zone, the zone information includes the server record of the primary master for the zone. If the zone is the master for the zone, then the zone information includes the number of slaves for the zone, the *server records* of all the slaves, and the number of slaves required. If the number of slaves is less than the number required, the agent will attempt to send *becomeslave* messages to other agents.

4.7 Operation

At any time, the agent program can be in two different states: the Configured state and the Start-up state. The state of the agent is stored in the in the *server record* information for the agent as described in Section 4.6.

4.7.1 Initial Configuration

The key piece of information any agent needs to operate is the location of an internal root server. On startup, the primary goal of any new agent is to find the location of an internal root server. Once an internal root is found, the agent is put into Configured state. An agent can only be in the Configured state if it knows the location of the internal root server, otherwise it is in the Unconfigured state.

Figure 9 depicts the startup scenario and the states of the transition between unconfigured and configured. When a new agent is started, it will wait for *discover* messages from the discovery process on the local manager. When at least one *discover* message is received, the new agent will send one *getroot* message to one of the discovered agents. If a specified timeout period expires and no other manager has been located, the agent will assume it is the only running manager on the system and configure itself to be the internal root. If the new agent hears only *discover* messages from Unconfigured managers, then they will compare Server IDs and elect the server with the highest ID to be the internal root server. This may happen when two or more managers join the network at the same time.

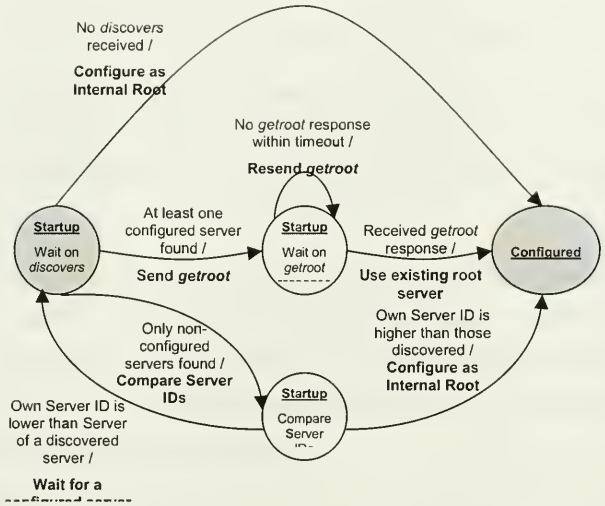


Figure 9 - Startup Scenario

4.7.2 Configured

Once an agent has entered the Configured state, it is ready to handle any name resolution query on the network. Even if it does not have the requested name in its database, it can query the internal root server and find an answer by working recursively down the DNS tree. When the agent is in the Configured state, it listens for queries, DNS updates, *discover* and *leave* messages, and (if it is a primary master for a zone) periodically runs the getSlave function. The getSlave function is used to find more slaves for a primary master. For every zone, the agent is a primary master for the getSlave function will check the zone information to see if more slaves are needed. If more slaves are needed, the agent picks a random server out of the Known Servers list that is not already a slave for the zone. It then sends a *becomeslave* message to that server.

The behavior of the agent in response to both DNS queries and updates is described in Section 4.4. The arrival of a *discover* or *leave* message signals a change in network topology, as managers have either joined or left the network. In the case of a *discover* message, there are two options: the discovery of a configured manager, or the discovery of an unconfigured manager. The case of the

discovered unconfigured manager is rather simple as an agent need only record the *server record* of the new manager and respond to any *getroot* messages it may receive. When a configured manager is discovered, the process is slightly more complex.

Configured managers are discovered when two configured networks are joined. Network unions are revealed by the manager discovery process as two managers will have different network IDs that were previously unknown to the other manager. In this case, it is possible to have a conflict where two managers are primary masters for the same zone. Therefore an agent needs to have a mechanism to detect and resolve this conflict.

Figure 10 illustrates the procedure that is initiated when a configured agent receives *discover* messages. When an agent is a primary master for a zone and receives a *discover* message with information on new configured managers, the agent will send a *getroot* message to the newly discovered server with highest ID (the server ID is used so that only one server is contacted, any other metric could be used as well). Only primary masters need to participate in the conflict detection scheme, as any slave's master will detect the conflict and transfer the new network information in a future zone transfer.

Zone conflicts can only occur if the two networks have different internal root servers. If both networks shared the same internal root server, it would be impossible for zone conflicts to occur as a single DNS root tree will not allow the same zone to have two different primary masters. When the two internal root servers are different, an agent must make sure that it is the only primary master for its zone. Therefore, the agent will send a SOA query to the differing root server for every zone that it serves as primary master for. The differing root server will then respond with the address of the manager that it considers to be the primary master for that zone, or an error message indicating that it is not aware of the zone. If the SOA response is an error message or if the address matches the querying agent, then no conflict exists. If the SOA message is an error, this indicates the other root server is not aware of the zone on the network. However, it can still resolve names in that zone by recursing down the DNS tree from the top node.

If, however, the differing root server indicates that it believes another manager to be the primary master for the zone, a conflict exists and must be resolved. The conflict is resolved by the *negotiatemaster* master message described in Section 4.5.

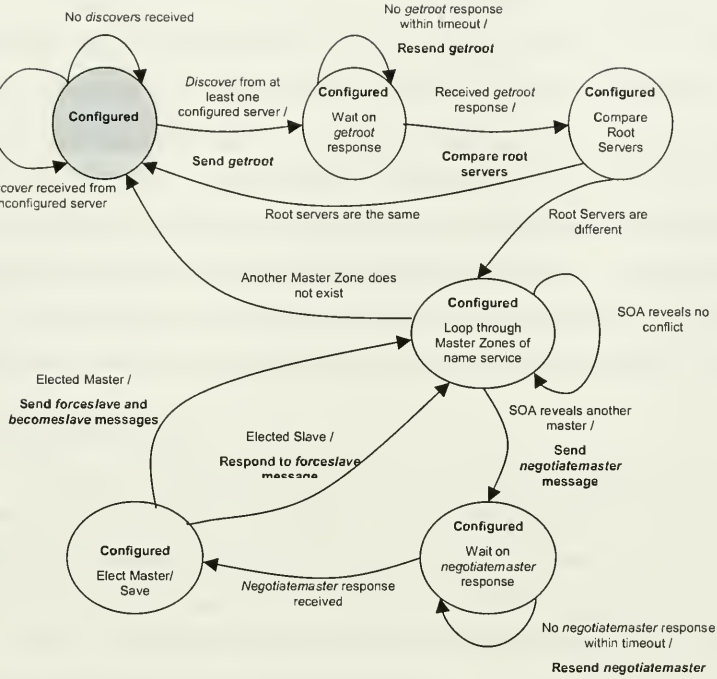


Figure 10 - *Discover* message handling in configured state

An arrival of a *leave* message may also require an agent to reconfigure itself. If a master server loses a slave, it will delete the slave's server record from its state information and send an *update* to the BIND process informing it of the lost name server. If an agent is a slave, it only needs to be concerned with *leave* messages that inform it that its master has left the network. All other *leave* messages will be processed by its master. When a primary master has left the network, all the slaves will become aware of this change and will need to elect a new master for their zone. The slave with the highest server ID becomes the new master. The highest server ID can be calculated by looking at the Known Servers state information (see section 4.6). This new primary master will then send *becomeslave* messages to every other slave informing them of their new master.

4.8 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 need 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 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. 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 to handle updates to the name database. Once the *named* process has started, the possibility exists 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 to handle all the network name resolutions.

4.9 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, the list, and the 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.

4.10 Security Concerns

The current implementation of the system has no security mechanisms built in. Securing a network designed to be configuration-free and administrator-free creates a conflict, as any meaningful security mechanism will require some sort of configuration [37]. The best that anyone can hope to accomplish is to minimize the configuration overhead necessary to keep the system secure. For a closed environment such as a corporate intranet, perhaps the easiest security model is to have none at all, simply relying on physical security mechanisms to control access to the machines [32]. This will work well for an environment that is well controlled and where one can physically control access to the network. If a corporation only allows approved machines and approved users to plug into the Ethernet, it is impossible for an outside intruder to gain access to the network.

However, as wireless networks become increasingly popular, controlling physical access to networks becomes impossible [14]. Insecure wireless networks allow anyone in range to send and receive data. This allows for theft of services (such as an individual using his or her neighbor's internet connection), as well as the potential for malicious hosts to be introduced to the system.

Another reason for incorporative security capabilities is when the network managed by the agent-naming system is connected to another larger network, such as the Internet. If the system is left insecure, any host in the world could attempt to violate the network's security mechanisms.

When a network is managed by the agent naming system, it should be *at least* as secure as a standard IP network. That is, the agent naming system should not open up any new holes into the DNS system. Currently, most DNS implementations are completely insecure, relying only on redundancy and physical security [35]. DNS servers are kept in a trusted, safe location and distributed throughout the network. Normal DNS servers only allow administrators at the physical machine to change the configuration of the DNS server. However, because the agent naming system allows other agents to change its configuration, it is prone to attacks from outside. If a malicious user could introduce a "rogue agent" into the system, it could do significant damage.

The managers in the agent naming system have two main functions. First, they provide IP configuration information to new hosts that join the network, and second they provide the name to address resolution service for the network. Each of these services is vulnerable to attack, and for a fully secure network, both must be secured.

There are two possible methodologies for securing the managers. The first is to use IPSec to provide network layer security for all traffic [19]. The second is to secure the individual protocols that provide IP configuration and name to address resolution [7][16][35][36].

Using IPSec, all traffic between managers is authenticated and optionally encrypted [19]; this ensures that no malicious user can place a rogue manager into the network, for it will not be able to authenticate itself. In addition, hosts can authenticate managers, so that when they receive their IP configuration, they can make sure it is correct and that it came from a known manager.

The other option is to utilize secure versions of DNS and DHCP. RFC 2535 defines DNS extensions that allow for host and DNS server authentication [10], while RFC 3007 specifies extensions to [10] that will provide authentication for *update* messages [36]. Authentication for DHCP messages is currently being proposed by the IETF [7]. These protocols may prove to be lighter weight than IPSec; however, they do not provide confidentiality of data. This is a deliberate design decision by the IETF because they believe DNS data to be public [10]. If confidentiality of data is desired, IPSec must be used to encrypt all data. IPSec may also be easier to implement, as it is a broad solution covering all network traffic. The only requirement is that the TCP/IP stacks of both the host and manager support IPSec. When IPSec is not used for security, any additional services added to the system must provide their own mechanism and protocol for authentication.

5 Related Work

5.1 ZEROCONF

The Internet Engineering Task Force's (IETF) ZEROCONF Working Group is currently proposing a protocol to enable networking in the absence of configuration and administration [38]. Although they have yet to propose a specific implementation or specification of the protocols, they have set a list of requirements for ZEROCONF protocols [15]. The goal is to allow hosts to communicate using IP without requiring any prior configuration or the presence of network services. Of particular relevance to this paper is the name to address resolution problem. The ZEROCONF requirements specifically state that there should be no need for preconfigured DNS or DHCP servers. In fact, the ZEROCONF protocols are required to work in the absence of DNS servers. However, when these services are present, the ZEROCONF requirements direct that hosts should use them. Thus, the ZEROCONF requirements offer temporary and inferior solutions to the name resolution problem until a complete name resolver is located, such as a DNS server.

ZEROCONF protocols face two challenges when determining name to address bindings. The first is obtaining a unique address and/or hostname on the network. This is handled extremely well in modern networks by the use of a DHCP server; however, ZERCONF protocols must not rely on the presence of DHCP server. Therefore, the working group recommends using either IPv6 or IPv4 auto-configuration mechanisms [14]. IPv6 auto configuration is vastly superior as it allows hosts to obtain a link local address (useful only on a single network) using address auto configuration [14] and a routable address by discovering routers using Neighbor

Discovery [24]. IPv4 auto configuration is still in the research state by the IETF, but the initial specifications allow a host to only get a link-local address [4]. This will prevent the host from communicating with any device not directly on the same network as it. Also, while IPv6 provides nearly all-necessary network parameters such an address, domain name, default router, and DNS server location (if present), IPv4 provides only an address. Thus, if a host configured with IPv4 auto configuration leaves a network and rejoins, it may have a new address, while a host configured by IPv6 will have a permanent method of contact, its domain name. While IPv6 clearly offers more advantages, it is expected that IPv4 will dominate for some more time [4]. This is because IPv6 is relatively new standard and the lack of fully complaint IPv6 routers and hosts on most networks has prevented it from gaining widespread acceptance.

Once a ZEROCONF host has obtained an address on the network, it still needs to discover other hosts and resolve domain names. The ZEROCONF requirements state that hosts should use multicast to resolve names in the absence of a DNS server. In order to support this requirement, an IP host will also need to listen for such requests and to respond when the request corresponds to the host's own name. The IETF currently has two ongoing activities in this area: multicast DNS [11] and "IPv6 Node Information Queries" [5]. In each case, all hosts will run a "stub" name service that only responds when it fields a request for its hostname. The stub service does not provide any name to address resolution for other hosts on the network.

While the naming service proposed in this paper may fit into the broad goals of the Zero Configuration Working Group, it has several key differences. The most obvious one is that the ZEROCONF requirements are designed to work with a network composed of entirely client devices, with no service providers or managers in the network. By design, the agent program is run on a network manager, and provides DNS services for the entire network. While the ZEROCONF requirements specify that hosts need no previous configuration, they do rely on more complete solutions such as DNS and DHCP for long-term operation and scalability. However, the ZEROCONF Working Group has set no requirements that these servers be self-configuring and self-administrating. This is precisely the problem that the agent program attempts to solve. In a network managed by the self-configuring naming service described in this paper, both hosts *and* managers are administrator free and may join and leave the network freely. Therefore it is possible to have a network that runs the agent program on the DNS servers and also meets the ZEROCONF requirements. Hosts could use the ZEROCONF protocols to obtain an address until the discovery of a manager. Once a manager is found, the host is free to resolve any name on the network using standard DNS calls.

5.2 Non-IP Networks

The requirements laid out by the ZEROCONF Working Group stress the importance of having computers networked together "just work" in the absence of service providers such as DNS and DHCP [15]. Two protocols in use today provide this level of

functionality. The AppleTalk suite of protocols is simple to operate in small networks. Plugging a group of Macs into an Ethernet hub will get one a working AppleTalk network without the need to setup specialized servers like DNS [14]. As a consequence, AppleTalk networks can be used in homes, schoolrooms, and small offices – environments where IP networks have been absent because they are too complicated and costly to administer. NetBIOS provides similar functionality and ease of use on Microsoft Windows machines.

However, because nearly all computers used today are connected to the Internet, they also require TCP/IP to be configured, as this is the protocol of the Internet. Therefore, the benefits of AppleTalk and NetBIOS are overshadowed as application developers will need to support two protocols: TCP/IP to access the Internet, and either AppleTalk or NetBIOS to access the local network. This is the motivation behind our research as well as the ZEROCONF Working Group and other efforts to make the IP suite of protocols simple to configure. Allowing developers to concentrate on one protocol for all communications will make application development simpler and more efficient. In addition, both protocols have issues when scaling to networks the size of the Internet. AppleTalk relies on broadcast packets to perform name resolution, thus creating a large number of unnecessary packets being sent to each host. A properly configured NetBIOS network can be designed such that no broadcasts are necessary. However, NetBIOS relies on a flat sixteen-character alphanumeric name space, which will undoubtedly lead to naming conflicts, particularly when networks are merged.

5.3 Easing DNS Administration

The above solutions were all replacements for a full DNS system; either by using other methods of name to address mapping over IP or by using different protocols altogether. However, because IP networks and DNS have been integrated into nearly all modern network systems, there have been other efforts to ease the configuration and administration of DNS.

Researchers in Japan have attempted to tackle DNS administration problems by simplifying configuration tasks and eliminating repetitive tasks through automation [13]. They developed a program with a graphical interface that reduced the work of a DNS administrator. Their tool, called *nssetup*, automates repetitive tasks such as generation a DNS database file from the machine's host file, keeping the root cache file up-to-date, and maintaining the reverse address lookup table. To check the correctness of the configuration, *nssetup* contained a feature that checked if the name server was up and running. In addition, *nssetup* provides a graphical user interface for configuring the resolver and adding new hosts into the database. The *nssetup* developers showed that it was considerably faster to configure a name server using *nssetup* than without it. They state they have reduced the configuration time from two hours to three minutes.

However, *nssetup* does not truly reach the goal of zero administration. It simply provides a nice user interface and a good set of default configuration values for BIND. Every time new DNS servers are added, an administrator must configure them as well as every

DNS server already running on the network so that they are properly integrated into the network. The agent program requires no prior configuration when DNS servers are added. Simply starting the agent program is all that is needed to be done; it will locate other managers on the network and they will configure themselves accordingly, all without user intervention. The time it takes the agent program to configure a name service is less than the time using *nssetup*, and does not require a human administrator.

5.4 Commercial Solutions

There are several commercial products available today that aim to simplify the operation and administration of DNS servers. Most provide solutions that include both DHCP and dynamic DNS in a single software package. These products are aimed at corporations that run a large intranet with their own DNS servers. By including both dynamic DNS and DHCP, they allow hosts to freely join and leave network with no manual configuration. DHCP will assign the new host an IP address, and send DNS *update* signals to notify DNS server of the new host so that it may be located on the network. Figure 11 shows the integration between DNS and DHCP in the commercial solutions with the addition of remote administration. Some popular products include Lucent QIP Enterprise 5.0, Check Point Meta IP 4.1, and NTS IP Server.

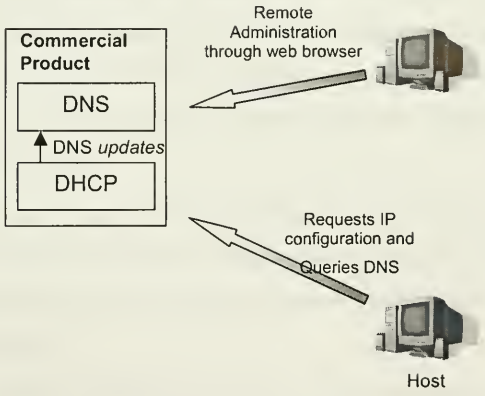


Figure 11 - Integration between DHCP and DNS

It is important to note that Meta IP includes some changes to the standard DNS protocol to allow for additional functionality. Specifically, the developers of Meta IP developed a proprietary way for primary and secondary servers to notify other slaves of zone updates. However, recent BIND implementations support the notify message which provides similar functionality [33]. Because the agent name service runs BIND, it also supports this feature.

The NTS IP Server is built from the ground up and is not based on any previous implementation of BIND. This allowed the developers to implement two proprietary extensions into the DNS protocol. The first DNS extension developed by NTS allows servers to have "peer backups". These are essentially duplicates of the server, and unlike secondary servers, they are always consistent with the main server. Essentially, "peer backup" allows for more than one primary server. Thus if one fails, the other is available for updates. The other DNS extension allows for zone "co-serving". A zone can be split into a number of pieces each served by a different server. However, any of the servers may be queried and may appear to be primary master for the zone. They will communicate amongst each other and return the correct answer to the resolver. For larger zones, this may lead to a performance improvement as the load can be balanced amongst a number of different servers. However, these extensions are not Internet standards, and only those servers running the NTS IP Server will be able to use them.

5.4.1 Comparisons to the agent name system

In addition the products described above, there exist other similar solutions to ease the administration and operation of DNS [18][25]. Nearly all of these vendor products offer features that appeal to the corporate environment. These features include user profile support, directory services, and remote administration. While these features are certainly advantageous in the corporate environment, they do not pertain directly to the name system. In addition, the goal of these commercial solutions is to provide a central point of administration for the entire network.

In contrast, the agent name system takes a decentralized approach to the administration of the network. No one manager is in charge of the entire management and most network operations, such as creating a new zone and electing a new master require communication amongst multiple managers. The decentralized approach allows for the relevant portions of the network to be involved in the decision making process.

Another key difference between the agent name system and those described in this section is the procedure for introducing new name servers into the system. Both the commercial systems and the agent name system require no administrative input when hosts join and leave the network. However, only the agent name system allows servers to do the same. In all of the above products, extensive reconfiguration is required when a server exits or joins the network.

While most of the commercial products provide a method to handle *update* messages while the primary server is down, none of them provides a permanent solution like the agent name system. They simply store the *updates* until the primary comes back on line or allow the secondary server to process them. However, if the secondary also fails, the updates will be lost. In contrast, the agent name system quickly promotes a new server to primary allowing all *updates* to be recorded permanently. Networks managed by the commercial solutions must be carefully planned so that the master/slave server relationship exists for all zones. Servers must be configured so that they are aware of the locations of the other servers. In the best case, the commercial products provide a human administrator a nice graphical user interface for performing the necessary configuration. In the worst case, manual editing of the configuration files is needed. Whatever the case is, the agent name system provides a simpler solution. No human administration is necessary when name servers exit or join the network, as the managers will communicate amongst themselves and adapt dynamically to the changing network topology. In addition, the agent name system solution works usually existing DNS standards and implementations. This ensures compatibility and takes advantage of the proven stability of the existing implementations.

5.5 Performance Comparison

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 [20]. 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.

5.6 Memory Usage of Name System

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 4. As shown in Table 4, the agent program occupied 1.7 Megabytes of memory and the BIND process, *named*, occupied 1.4 Megabytes of memory.

Table 4 – 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. Standard computers should be able to run the agent program and the BIND process without many problems.

6 Conclusion

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 comparison to approaches proposed by other researchers, the agent name system offers superior functionality for large networks and can also be more easily integrated with existing Internet solutions. The name system offers solutions for many of today's networks. It offers a simple solution for homes and small businesses that cannot afford a permanent IT staff to administer their network, while being scalable enough to handle networks of much larger size and complexity.

7 Acknowledgements

We would like to thank Ralph Preston, Sam Wiebenson, Jeffrey Yu and Kevin Grace for their help and assistance with this paper.

8 References

[1] P. Albitz, C. Liu, *DNS and Bind*, 3rd Edition, O'Reilly, 1998.

[2] *AppleTalk Network System Overview*, Addison-Wesley Publishing Company, Inc., 1990.

[3] *BIND Administrator Reference Manual*, Nominum BIND Development Team, January 2001.

[4] S. Cheshire and B. Aboba, "Dynamic Configuration of IPv4 Link-Local Addresses", draft-ietf-zeroconf-ipv4-linklocal-02.txt, March 2001. (work in progress)

[5] M. Crawford, "IPv6 Node Information Queries", draft-ietf-ipngwg-icmp-name-lookups-07.txt, August 2000.

[6] R. Droms, "Dynamic Host Configuration Protocol", RFC 2131, March 1997.

[7] R. Droms, ed., "Authentication for DHCP Messages", draft-ietf-dhc-authentication-16.txt, January 2001. (work in progress)

[8] "Dynamic DNS, Infrastructure essential for today's intranets", http://www.nts.com/collateral/ddnstechpaper.pdf

[9] D. Eastlake, "DNS Security Operational Considerations", RFC 2541, March 1999.

[10] D. Eastlake, "Domain Name System Security Extensions", RFC 2535, March 1999.

[11] L Esibov, B. Aboba, and D. Thaler, "Multicast DNS", draft-ietf-dnsext-mdns-00.txt, November 2000.

[12] M. Feng, "A Self-Configuring and Self-Administering Name System", http://scanner-group.mit.edu/PDFS/FengM.pdf, Master's Thesis, Massachusetts Institute of Technology, 2001.

[13] C.S. Giap, Y. Kadobayashi, S. Yamaguchi, "Zero Internet Administration Approach: the case of DNS", *Proc. of 12th Int. Conf. on Information Networking (ICOIN)*, pp. 350-355, 1998.

[14] E. Guttman, "Zero Configuration Networking", Proceedings *of INET 2000*, 2000.

[15] M. Hattig, ed., *Zeroconf Requirements*, draft-ietf-zeroconf-reqts-07.txt, March 2001. (work in progress)

[16] K. Hornstein, et al, "DHCP Authentication Via Kerberos V", November 2000.

[17] P. Huck, "Zero Configuration Name Services for IP Networks", http://scanner-group.mit.edu/PDFS/HuckP.pdf, Master's Thesis, Massachusetts Institute of Technology, 2001.

[18] "JOIN DDNS", http://www.join.com/ddns.html

[19] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

[20] "Lucent QIP Enterprise 5.0: Automating IP Services Management", http://www.qip.lucent.com/products/qipent_6093.pdf

[21] "Meta IP Technical White Paper", http://www.checkpoint.com/products/metaip/whitepaper.pdf

[22] P.V. Mockapetris, "Domain Names – Concepts and Facilities", RFC 1034, November 1987.

[23] P.V. Mockapetris, "Domains Names – Implementation and Specification", RFC 1035 November 1987.

[24] T. Narton, E Nordmark, and W. Simpson, "Neighbor Discovery for IP Version 6 (IPv6)", RFC 2461, December 1998.

[25] "Optivity NetID", http://www.nortelnetworks.com/products/01/unifiedmanagement/collateral/onetid_brief.pdf

[26] "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods", RFC 1001, March 1987.

[27] J. Reynolds and J. Postel, "Assigned Numbers", RFC 1700, October 1994.

[28] "Shadow IPserver", http://www.nts.com/collateral/ipserverdatasheet.pdf.

[29] G.S. Sidhu, R.F. Andrews, A.B. Oppenheimer, *Inside Appletalk*, Second Edition, Addison-Wesley Publishing Company, Inc., 1990.

[30] M. Stapp and Y. Rekhter, "The DHCP Client FQDN Option", draft-ietf-dhc-fqdn-option-01.txt, March 2001. (work in progress)

[31] R. Thayer, N. Doraswamy, and R. Glenn, "IP Security: Document Roadmap", RFC 2411, November 1998.

[32] H. Toivanen, "Secure Zero Configuration", http://citeseer.nj.nec.com/401221.html.

[33] P. Vixie, "A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)", RFC 1996, August 1996.

[34] P. Vixie, et.al., "Dynamic Updates in the Domain Name System (DNS Update)", RFC 2136, April 1997.

[35] P. Vixie, et al, "Secret Key Transaction Authentication for DNS (TSIG)", RFC 2845, May 2000.

[36] B. Wellington, "Secure Domain Name System (DNS) Dynamic Update", RFC 3007, November 2000.

[37] A. Williams, "Securing Zeroconf Networks", draft-williams-zeroconf-security-00.txt, November 2000. (work in progress)

[38] "Zero Configuration Networking (zeroconf) Charter, http://www.ietf.org/html.charters/zeroconf-charter.html

A Appendix

A.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 below.

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: *srv_rs_configured*. If not, then the status flag will have the value: *srv_start_up*.

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 below.

Server Record List Type	Description
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.

A.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 is 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 also stores:
        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 also stores:
        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.

A.3 Agent Messages

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

Server Record Format

Name
IP Address
Server ID
Status Flag

***Discover* Message**

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

***Leave* Message**

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

***Getroot* Message**

Msg_Id	number
Opcode	getroot
Send/Response Flag	***Send***
Payload	Empty

Msg_Id	number
Opcode	getroot
Send/Response Flag	***Response***
Payload	Server Records for the root servers in the system.

***Becomeslave* Message**

Msg_Id	number
Opcode	becomeslave

Send/Response Flag	***Send***
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	becomeslave
Send/Response Flag	***Response***
Payload	Zone Name the agent should be a slave for.
	Server Record for the slave name service.

Negotiatemaster **message**

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

Msg_Id	number
Opcode	negotiatemaster
Send/Response Flag	***Response***
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	forceslave
Send/Response Flag	***Send***
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	forceslave
Send/Response Flag	***Response***
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.