The Design and Implementation of the Berkeley Internet Name Domain (BIND) Servers†

Songnian Zhou

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

ABSTRACT

Name servers are system processes that maintain databases of information about objects existing in computer networks, and answer user queries concerning them. Naming services are becoming increasingly important as computers are connected into networks and eventually into internetworks. This report describes our design and implementation of the DARPA Internet name servers in the Berkeley Unix environment. The naming service and the information it provides are both distributed and replicated. User update queries, as well as retrieval queries, were implemented, and zones of network information are kept up to date by incremental zone refresh operations.

1. INTRODUCTION AND BACKGROUND

1.1. The Need for Network Naming Services

With the proliferation of computer networks and internetworks, more and more computers are connected together through all types of communication media, ranging from simple copper wires to coaxial cables, microwave channels, and satellite stations. This trend opens up unlimited opportunities for people to communicate with each other, to share computing resources, and to develop new applications. Many organizations have connected their computers together using local area networks, and several long haul networks, such as the DARPA Internet, CSNET, and Telenet, connect together numerous installations in the United States and abroad. The increasing use of personal workstations makes it possible to achieve fast and predictable response but at the same time imposes greater demand on network services, such as file servers, print servers, and network information servers and so on. The workstations have to be connected to networks in order for them to share the network resources and to communicate with other users; the usefulness of a stand-alone workstation is quite limited. It is not an exaggeration to say that our era is the era of communications.

The first problem in communication is that of naming and addressing, i.e., of how should the party one wants to communicate with be named, and how its location be specified. Specifically, in a large computer network environment, we are faced with the problem of how to assign names to the great number of objects in the network without causing any confusion, and how to provide addressing information for them quickly and efficiently. An object in a network is any entity that can be specified or invoked by its name. Hosts, printers, user mailboxes, software packages, and processes are all examples of network objects. The rapidly growing size and topological

†This work was partly sponsored by the Defense Advanced Reasearch Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under Contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the US Government.

complexity of networks makes it clear that such problems are significant enough to be studied seriously. Their best accepted solution is the introduction of yet another kind of network service, called *naming service* or *directory service*.

Naming services have been provided ever since the beginning of computer networks, often in an *ad hoc* manner. For example, in the early days of the DARPA Internet, a table of all the hosts on the network together with their Internet addresses was maintained at a central location and distributed to every host. Every communication between two hosts involved searching the local copy of this table. To avoid obsolescence, every copy of the table had to be updated periodically. As the size of the Internet grew, it became obvious that such a table was unmanageable. What is really needed is a systematic and distributed approach to providing naming services.

That of providing network addresses given a name is only one, albeit important, of the services that can be provided by network name servers. For example, information about user mailboxes, the locations and availabilities of various network resources such as printers, graphic displays, and application software packages can all be maintained and provided by name servers. Interprocess communication can also be assisted by name servers. A process that is ready to communicate with other processes may indicate its willingness by registering with a name server and another process ready to communicate with the first can find the address of the first process by sending a query to a name server.

1.2. Network Naming Conventions

The first problem to be solved for network naming service is that of naming convention, that is, the set of rules that must be followed in assigning names to network objects. In the past, many methods have been proposed and used. The UUCP protocol, for example, adopted a *relative naming convention* in which the name of an object is actually a concatenation of the hosts that form a route from the source to the object being named. Such a scheme is called relative as the name of an object is not unique but rather depends on where it is used. Since relative names are non-transferrable, that is, the name of an object used at one place cannot be used at another place without change, relative naming conventions are not convenient. Another class is that of *absolute naming conventions*, in which every object is given a name that is unique throughout the entire network environment. For instance, in the Grapevine system designed at Xerox PARC [Birrell *et al.* 82], the name space is a two level hierarchy and every object name is the concatenation of two parts, a *registry name* and a local name within that registry. While such a scheme serves well in an environment like the Xerox internetwork, the two-level restriction seems unreasonable for a more diversified environment with many organizations and different types of hosts. The DARPA Internet is such a case.

Another problem with naming conventions is that of naming authority. One way to solve this problem is to create a central authority to which every request for a name is directed, just as is done at the present time in the Arpanet. A name server at the Network Information Center (NIC) at SRI International maintains a master database of the addresses of all the hosts on the network, and answers queries from all sites. Such a centralized scheme is simple and has been widely used but has severe limitations. The central name server is a critical resource in the network, and the reliability and performance aspects of this solution are unsatisfactory [Pickens *et al.* 79]. It is highly desirable to distribute the naming authority to a number of regional agents while still maintaining the global uniqueness of names.

The hierarchical naming convention proposed by Su and Postel [Su and Postel 82] is a general scheme that meets both of the above two requirements. In this scheme, the name space is organized as a tree according to administrative structures. Each node, called a *domain*, is associated with a label, and the name of a domain is simply the concatenation of all the labels of the domains from the root to this domain, listed from right to left and separated by dots. The labels need only be unique under the same parent, thus naming authority may potentially be delegated at every domain. The whole space is partitioned into a number of areas called *zones*, each starting at a domain and extending down to the leaf domains or to the domains where other zones start. Zones usually represent human administrative boundaries and associated authorities.

For example, Berkeley may have a zone "ucb.arpa" and the computing center at Berkeley may have a zone "cc.ucb.arpa" under "ucb.arpa", each being maintained independently by a responsible person, the *zone manager*. Such a scheme is general in that there is no limitation on the number of levels of the domain space and the number of subdomains that a domain may have. It is also distributed because each zone may have more than one name server that provides naming service and maintains relevant information for the zone. From this point on, we use the term zone to refer to an area in the domain space as described above, as well as all the information concerning the objects in that area.

1.3. DARPA Internet Name Servers

Name servers are system processes that keep databases of information about objects existing in the networks and answer user queries concerning them. Collectively, name servers provide network naming services to users. Mockapetris at ISI has proposed a basic specification and design for DARPA Internet name servers in two Requests for Comments, [Mock 83, 1] and [Mock 83, 2]. The Internet naming service is supported by a number of name servers that are distributed throughout the internetwork. There exists a general mapping between the zones and the name servers, that is, a zone may be stored at one or more name servers, and a name server may contain zero or more zones. We call a name server containing a zone an *authoritative name server* for that zone. It is authoritative for all the information within that zone and should therefore constantly try to keep it up to date. For performance and reliability reasons, a zone is usually stored in several name servers, the degree of replication depending on the frequency of its use and its importance for network operation. The zone at the root of the domain tree, for instance, may be highly replicated to avoid frequent remote queries and to be able to continue operation when one or more name servers fail. The presence of multiple copies of the same zone presents fundamental problems when updates occur. This will be discussed in the next section. Besides authoritative zones, a name server also contains some cached information which can be used to answer user queries but for which the name server is not authoritative. A time-out scheme is used for such cached data. Notice that a name server may contain only cached data but no zones. Such degenerate name servers are possibly useful on small machines like personal workstations.

Information about network objects is stored as *resource records* (RR). Each resource record consists of domain name, class, type and data fields. Every domain has zero or more resource records attached to it. Different kinds of information are distinguished by the type field. For instance, resource records for host addresses have a type value "A" and those for user mailboxes have a type value "MB". Two special types are used for domain space data management. The "SOA", or *Start Of Authority*, type signifies the origin of a zone, that is, the domain where a zone starts; the "NS", or *Name Server*, type signifies the point of the delegation of authority. A record of NS type contains, in its data field, the domain name of the host on which resides a name server that has the zone starting at the current domain. We will discuss the uses of these two types of resource records in the next section. A complete list of all the currently defined resource record types can be found in Appendix IV.

Besides name servers, the naming service has two other parts. One is the user interface called the *resolvers* which consists of a group of subroutines that can be invoked by the users. Resolvers assemble queries on behalf of the users, send them out to name servers, receive responses from name servers and analyze them, and finally return answers to users. More sophisticated resolvers also cache resource records and information about name server structure for improved performance. There is usually one resolver on each host, and there may be several versions of resolvers of differing complexities on hosts of different capacities. The other part of the naming service is the set of databases maintained by name servers. There is a separate logical database for each zone and for the cached resource records. The relationships among the resolvers, the name servers and the databases are depicted in Figure 1. The three parts of the naming service deal with different aspects of it and such a division of functionality facilitates modular design and implementation. This paper will concentrate on the design and implementation of the name server part. For descriptions of the design of the resolvers and the name server databases, see [Painter 84] and [Riggle 84], respectively.

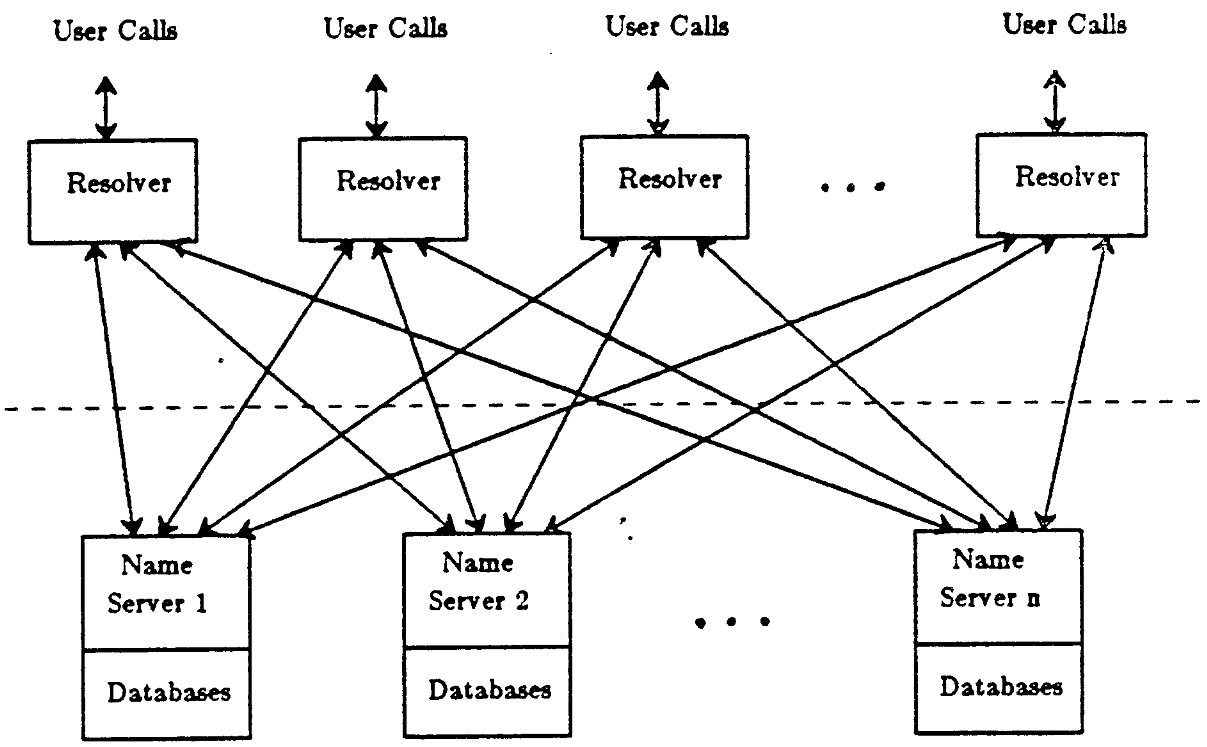

Figure 1. Relationship between the resolvers, the name servers, and the databases.

From the above description, it is evident that the design proposed by Mockapetris is intended to be general purpose. A reasonable implementation of it should be able to accommodate new applications without fundamental changes. For example, to store new types of network information, we need only specify new resource record types and modify a few sections of the name server programs if special processing for such new types are required.

A version of the Internet name domain server has been implemented for the Berkeley Unix environment. While the basic protocol and specifications in RFC883 were followed in our design, a number of modifications and extensions were made, mainly in the update and the domain data management aspects. We discuss in detail such extensions and the design choices involved in the next sections. In Section 2, we describe the algorithms for user query processing and the management of domain space data. In Section 3, some implementation issues are discussed. Future work is mentioned in Section 4, which is followed by the conclusions in Section 5.

2. USER QUERY PROCESSING AND DOMAIN DATA MANAGEMENT

The basic function of a name server is to provide information about network objects by answering user queries. This will be discussed first in this section. In the specifications of RFC882 and RFC883, only retrieval queries are allowed. That is, users cannot change the contents of the name server databases *interactively*. We feel that user update queries are necessary and therefore should be included. We will describe the basic design for domain data management first, then discuss Berkeley extensions and design alternatives.

2.1. The Processing of User Retrieval Queries

2.1.1. Basic Query Format

Both a query and a response to a query consist of five sections. The first section is the header, which contains a query identification number, a query type code, a response code, and a number of flags, one of which, the authority flag, indicates, in a response, whether the name server is authoritative for the resource record(s) requested. The second, or the *question* section contains a domain name, a class code, and a type code. The other sections are the answer, the name server, and the additional sections, each containing a number of resource records. The uses of these records are indicated by the header, but in most cases the answer section of a response contains records that provide an answer to the query; the records in the name server section provide information about other name servers that should be queried; and the additional section contains pertinent information which is not strictly an answer to the query. The header is always present in a query or response, whereas any of the other sections may be absent depending on whether they are needed or not. Figure 2 shows a schematic of the general query format. For detailed information about query format and contents, see [Mock 83, 2].

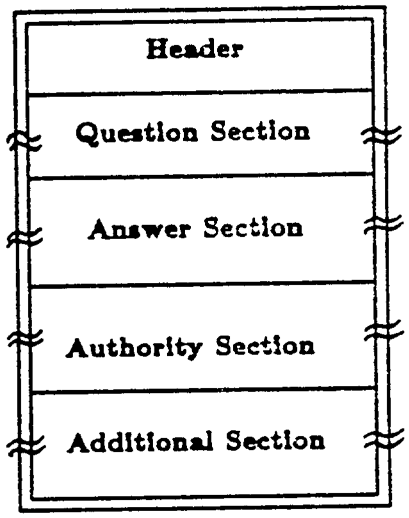

Figure 2. General format of a query.

2.1.2. Determination of the Zone to Search

Since a name server may contain several zones, the first step in query processing is to decide which zone to search. The zones of the domain space usually form a partition of the space, that is, there is no overlapping between zones, and the whole space is covered by zones. Therefore, a domain belongs to one and only one zone. The domain name in the question section is compared with the origin of each zone in the server. The zone with the longest match is the only one in this server which may possibly contain the domain being sought. For example, suppose the requested domain is "d.c.b.a", and the name server contains zones with origins "a","e.c.b.a", and "b.a", then the zone "b.a" is selected. In this example, it is clear that zone "a" delegated part of its authority to zone "b.a" and that zone "e.c.b.a" is beside the domain being sought. It is possible that no matching zone is found. In this case, this name server cannot decide whether such a domain exists or not and its cache is searched for relevant resource records. At the very least, the NS records of the root domain together with the address records of the hosts on which they run will be returned to the resolver to allow it to continue its search for an authoritative name server.

2.1.3. Determination of the Zone Authority

Suppose a matching zone is found. This does not necessarily mean that the name server is authoritative for the domain. It is possible that authority has been delegated to another zone at some domain along the path from the zone origin to the domain being sought. To check for this, the server should search down this path for NS type resource records, which indicate the starting point of another zone. If such a record or records are found, they are returned to the resolver along with their address records. If no such NS records are found, then we are certain that this name server is authoritative for the domain, if it exists. The zone database is searched for the resource record(s) requested and they are returned if found or an error message is returned.

2.1.4. Other Types of Retrieval Queries

We described above the processing of the most common type of queries, the *standard* queries. Depending on application requirements, other kinds of retrieval queries may be needed. Two of them are described in the proposed specification. The first type is that of *inverse queries*, which require inverse mapping from domain data to domain names. A name server is presented with a specific data field of a resource record and will search its databases, probably through a secondary index, for resource records with such data and return them to the resolver. The other type of query is that of *completion queries*, which require the name server to expand a partial domain name, like a local shorthand, into a full domain name according to a set of *closeness criteria*, and perform standard query processing on this name. Other types of queries are also possible. For example, a particular application may need information about all the print servers within an organization. This kind of "yellow page" service may make the addition of new types of queries necessary. We decided not to include these types of queries in our implementation for the time being, but we keep the door open for future expansions.

2.2. Iterative versus Recursive Query Processing

In the algorithm described above, queries are processed *iteratively*. That is, a name server returns to the inquiring resolver either the requested resource records or information about other name server(s) where the query should next be directed. It is the responsibility of the resolver to follow the forwarding references until reaching the name server with the information needed or getting an error message back. However, resolvers with limited functionalities are sometimes needed because of restrictions in the storage size and the computing power of the host. For instance, a small machine such as a workstation may not be able to support a full-scale resolver. In this case queries may be processed *recursively*. If the first name server being queried by a resolver does not have the information, it queries other name servers trying to get the information and eventually returns it to the resolver. Besides alleviating some of the burden of a resolver, another advantage of recursive query processing is that it can facilitate caching of resource records received from other name servers. Such records may be needed in the future, and multiple queries can be avoided in that case. On the other hand, such a scheme blurs the clear functional boundary between name servers and resolvers, since now the name servers have to be able to send queries to other name servers, just like resolvers. Recursive query processing is optional, and is not currently implemented in the BIND servers. We will study recursive query processing together with caching problems in more detail in Section 4, where future work will be discussed.

2.3. Domain Space Data Management

2.3.1. The Design Proposed in RFC882 and RFC883

2.3.1.1. Domain Data Updates

In the proposed design, users of the name servers cannot update resource records directly. They can only retrieve information via queries. For every zone, there exists a text file called the *zone master file*, which lists all the records in the zone in a well defined format. When a name server is booted, it reads in the specific master files and initializes its zone databases. All changes to a zone are made to its master file, using a text editor. This can be executed only by the zone manager. Therefore, any user who wants to make a change to the zone data must submit a request to the zone manager, who determines whether such an update is permitted, and edits the master file on the user's behalf. To keep the zone database up to date, the name server must periodically check if its master file has been edited recently. If so, it reads in the master file to create a new zone database. When this is done, a lock is set to prevent user queries, and the new database is switched into use, the space of the old database being reclaimed.

Clearly, the above scheme is simple and reliable. Since the legitimacy of updates is checked by a human manager, the authentication and access control mechanisms required for the name servers can be greatly simplified. But such a simplification is achieved by imposing severe restrictions on the maintenance of domain space data. The update operations are performed manually and therefore may take quite a long time, especially so if the manager is temporarily unavailable. Moreover, since the loading of a zone database which contains a large number of resource records may be expensive, the operation cannot be performed very frequently, and this is the cause of yet another delay in the realization of an update.

2.3.1.2. Support for Replicated Zones

As mentioned in Section 1, most zones are stored in a number of name servers. This means that several name servers may be authoritative for the same zone. The purpose of replication is obtaining better performance and greater availability. Such improvements are not achieved without costs, however. Because of the existence of multiple copies of the same zone, efforts must be made to keep every copy up to date and consistent with the others. Two problems are involved here:

1) How does each name server which is authoritative for a zone get its resource records?

2) How are updates to the master file propagated to each of its database images?

One way of solving these problems is to distribute the master file to each name server host and let the name server there read in the copy of the master file. This can be done manually by the zone manager periodically or after a certain number of updates to the master file. Yet a better solution would be to let the name servers themselves perform the data propagation. In the proposed design, one of the authoritative name servers is distinguished from the others in that this server loads its database directly from the master file, while the others contact this server to get the resource records from its database. We call the name server having access to the master file the *primary name server* for the zone, and the others the *secondary name servers*. When each of the secondary servers is booted, it sets up a connection with the primary server and sends over to it a maintenance query identifying the zone it desires. If the primary server has loaded such a zone and the query is from a legitimate server, it sends the nodes in the zone one by one. The secondary name server receives these records, creates a new zone database and then performs a switch just as described before in the previous subsection. Independently of whether or not such an initial zone transfer is successful, the secondary servers will thereafter perform the same zone transfer periodically. Note that the initial zone transfer of a secondary server may be unsuccessful either because the primary server is not alive or because it dies in the middle of the transfer. Some savings can be achieved by not transferring a zone to the secondary servers if the primary name server has not loaded a new version of the master file since the last transfer. Similarly, the primary server need not load the master file if this file has not been edited since the last loading. For brevity, we refer to the copy of a zone residing in its primary server as the *primary zone*, and to those in the secondary servers as the *secondary zones*.

The above description is not very detailed, but we can already see a number of problems. First of all, a whole zone has to be transferred to each secondary name server to keep it up to date. Note that a zone may contain a large number of resource records. For example, the Berkeley zone may contain several dozens of hosts and servers, hundreds of workstations, and hundreds of user mailboxes and user aliases, adding up to several thousands of resource records. In this case, whole zone transfers may impose a heavy load on the network and the hosts involved. Measurements and experiences with distributed information systems have shown that updates are much less frequent than retrievals [Terry 84]. This is reasonable in our environment since most of the objects, like host addresses, user mailboxes, and service ports, are relatively stable, and do not change over long periods of time. A whole zone that is transferred may differ from the copy that already exists on a secondary server only by a few resource records, while the rest of the records are simply copied over. As an example, suppose the Berkeley zone is stored in a secondary server, and during the refresh interval only ten of the one thousand records are changed. In this case, 99% of the information transferred over the network is not needed by the secondary servers.

A second problem with the proposed design is that, in every zone transfer, resource records are transmitted node by node using reliable communication protocol, such as TCP [Postel 81]. Since the overhead of processing a message for transmission is essentially independent of the message size [Almes and Lazowska 79] , heavy load may be placed on the communication protocols.

Because of the deficiencies in the update and replication management of the original design, we decided to make some modifications and extensions in the Berkeley implementation. These are discussed in the next section.

2.3.2. The Berkeley Modifications and Extensions

2.3.2.1. Interactive User Updates

From the above description of the proposed design, we conclude that the latency of domain data updates may be intolerably long. This is even more the case if information with a short expected life, like the communication ports of a process used in process rendezvous, is supported by the name servers. Here the problem stems from the human involvement in update operations. In the Berkeley implementation we automated update operations by permitting user update queries. Three types of the queries were defined, as shown in Table 1.

OPERATION	OPCODE	SEMANTICS
addition	UPDATEA	add a RR, return NOCHANGE if already exists
deletion	UPDATED	delete a RR, return NOCHANGE if nonexistent
modification	UPDATEM	replace old data section of RR with new one atomically

Table 1. Update query types and their semantics.

The format of update queries are the same as that of standard queries, and the resource records to be added or deleted are stored in the additional section. For modification operations, both the old and the new records are provided to facilitate complete error checking. We require that the old and new records have exactly the same domain name, class, and type, but differ in their data fields. We feel that this is adequate, as modifications are meant to be atomic operations. Indeed, if different domain names and/or classes are allowed, the operation may involve two zones, and increase the hazard of leaving the database in an inconsistent state if a crash occurs in the middle of the modification. Such more general kind of modification can always be performed by doing an addition followed by a deletion. So, we lose no functionality by imposing the above restriction.

It is possible that the response to an update query is lost because the communication between resolvers and name servers normally uses connectionless protocols, such as datagrams

[Postel 80]. In such cases, a resolver may time out and send the update query again. This should be tolerated as a fact of life. Instead of returning an error message in such a case, we decided to define the new response code "NOCHANGE". For addition, it indicates that the resource record is already in the database. For deletion, it means that the record is absent, and, for modification, it means that the old record is absent and the new one present. Error checking is performed as carefully as possible to ensure consistency in the zone database. No redundant resource records, that is, multiple identical records in the same zone database, are permitted. This restriction makes the implementation easier and the semantics clearer, while causing no foreseeable damage to the generality of the naming service.

An important question is which name servers can accept update queries. Although we can allow all the authoritative name servers to accept updates and operate on their copies of the zone databases, this scheme would increase the system's complexity significantly. Propagation of updates becomes a serious problem. Suppose that there are n authoritative name servers. If we perform the zone refresh operations periodically, every copy of the zone may contain some recent information that none of the other name servers has. Therefore, in the worst case, $n(n-1)$ zone refreshes will have to be performed during every period. If we decide to propagate each update immediately, $n-1$ transfers are needed for each update. Since update propagation is crucial to name server operations, we need to use reliable communication protocols, like those which establish virtual circuits [Postel 81]. For each zone refresh operation, a connection would have to be set up between the two parties involved. Such a connection is expensive, and it is desirable to minimize the frequency of zone refreshes while still providing timely information to users. A reasonable solution is to buffer the updates and propagate them in batches. We will discuss our approach later.

Besides causing potential performance degradation, allowing multiple name servers to process update queries may even create logical errors. As an example, consider the following scenario. A user decides to modify a resource record. He performs an update query at one site in the network and succeeds. He then goes to another site and discovers that his previous modification was wrong. So, he performs another update query which, unfortunately, goes to another name server. These two updates will eventually propagate to a third name server. However, because of the asynchronous nature of the propagations and the random communication delays, the second modification may arrive before the first. As a result, now the third server database contains a wrong record but gives the user no warning.

Considering the above problems, we decided to restrict all updates to be directed only to the primary name server, the server which derives its zone data directly from the master file. Now the picture becomes much clearer. The price we pay is possibly one more query for a user update. If updates are relatively infrequent, as reported in [Terry 84], such a price is probably easily affordable. The details of our design are described below.

2.3.2.2. Incremental Zone Refresh

The decision to allow interactive user updates and to require updates to be performed only through the primary name server changes the original picture of domain data management drastically. First of all, the role of the zone master file becomes much less important. Now, the most up-to-date image of the zone data is not this text file but rather the zone database of the primary server. The only reasons that the master file is kept now are to provide a source of zone data for loading the database when the primary server is booted for the first time or if the zone database is unrecoverably damaged, and to provide a human readable version of all the records in the zone which may be slightly out of date. Instead of calling it the master file, we call it the *backup file* of the zone. At appropriate times, for instance at regular intervals or after a certain number of updates, the primary name server dumps its database to create a new version of the backup file.

While describing the proposed design of zone update propagations, we stated that it would be grossly inefficient to transfer a whole zone from the primary server to each secondary server for every refresh operation. This would be even more the case if the nodes of a zone are transferred one by one. We decided to use *incremental refresh queries* instead of whole zone transfers after the initial transfer. When an update arrives at a primary name server, in addition to performing the update to its database, the server also records the update in a data structure called the *update list*. Each secondary server periodically requests a connection to the primary server and sends a refresh query identifying itself and the zone desired. The primary server then places all the updates since last refresh for this secondary server into the answer section of the response and sends it back. Multiple messages will be used if there are too many updates to be stored into one response message. Pointers into the update list are maintained for each secondary server to keep track of how much of the update list this server has received, and the storage occupied by update records that have been distributed to all the secondary servers is reclaimed. The structure of the update list and its management will be discussed in the next section.

Performance-wise, the incremental refresh scheme has two advantages. One is that the whole zone transfers are avoided, thereby reducing network and host load significantly. The other is that the updates are propagated in batches, and their frequencies are controlled by each secondary server, so that frequent connection costs are avoided. However, the timeliness of update propagation is affected. This is remedied by defining an "important" class of retrieval queries. We designate a bit in the header of a query as the *primary required*, or PR, flag. If this flag is set in a query, a secondary server for the zone will not answer it but rather redirect it to the primary server where the most up to date information exists. Thus, without sacrificing essential functionality, we have reduced the frequency of zone refresh operations.

Four observations are in order here. First, the secondary servers for a zone are used as caches, i.e., used to improve performance and accessibility. They get their initial data as well as all the updates from the primary server. Data propagation has a radiation pattern, flowing out of the primary server to all the secondary servers, and to the backup file. Secondly, absolute data consistency among all the copies of the zone data is not guaranteed; instead, we ensure only the eventual convergence of the data. Such a relaxation of the multiple copy consistency requirement seems tolerable in our naming service environment, and greatly simplifies the design, as well as improving system efficiency and service availability. Thirdly, the terms primary servers and secondary servers are meaningful only with respect to a particular zone. While a server is secondary for one zone, it may well be primary for another. This generality promises to make the name servers highly adaptive to working requirements imposed by organizational and administrative structures. For instance, an industrial corporation and its research division may have separate zones and name servers and the research division server may have the zone for the corporation as a secondary zone, vice versa. Lastly, it should be pointed out that permitting interactive user updates substantially increases the complexity of the authentication and access control mechanisms needed in order for the name servers to be usable in the real world. The zone manager definitely would not appreciate it if an "ordinary" user deleted his mailbox, or modified the root name server record. No authentication and access control has been implemented in the current version of the BIND servers but we discuss the problem briefly in Section 4. In Figure 3, we show the name servers' structure in relation to a particular zone, and the data propagation pattern.

2.4. Compatibility with the Proposed Design

The Berkeley extensions to the proposed ISI design are based on functionality and performance considerations. The Berkeley implementation is upward compatible to the proposed design so that name servers of different implementations can maintain secondary zones for each other and communicate with each other using the same protocol. We shall now discuss two problems that need to be resolved for the Berkeley implementation to achieve full compatibility.

First, in the ISI design a serial number field in the SOA type resource record for a zone is incremented every time that zone is reloaded from its master file. A secondary server ready to perform a refresh operation will first retrieve this SOA record from the primary server and compare its serial field with the serial field in its own SOA record. If the two are equal, no new version of the master file has been loaded since the last refresh, and therefore the zone need not be

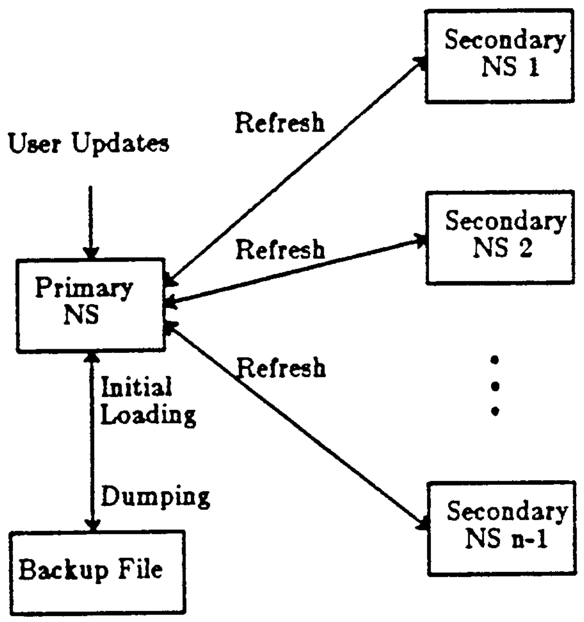

Figure 3. Name server structure in relation to a particular zone.

transferred. In the Berkeley implementation, every user update query changes the content of a zone database, and thus corresponds to the loading of a new version of the master file. Therefore, we increment the serial number of the zone with every update. For efficiency, we may not want to increment the serial number after every update but only after a certain number of updates. This number of updates is settable to allow a balance to be reached between the timeliness of the information in the secondary servers and the frequency of zone refresh operations.

The second difference is the way maintenance queries are processed in the two designs. A Berkeley version primary server should be able to answer maintenance queries from a secondary server of the proposed design. Two sets of routines are provided, and the type of the secondary server can be detected by its first query over a virtual circuit connection. For a Berkeley type server, the query operation code in the header is ZONEINIT or ZONEREF, and for the proposed implementation, it is QUERY with the type field in the question section set to SOA.

Similarly, a secondary server of the Berkeley implementation must be able to refresh its zone from a primary server of the ISI design. When first started, the Berkeley type server will try a ZONEINIT query and get an error message back. It can then switch protocol to get the zone and initialize its database. From this point on, the secondary server will use the same protocol for every maintenance query for the zone.

The two different styles of zone transfers have effects on the underlying databases, and are discussed in [Riggle 84].

3. SOME IMPLEMENTATION ISSUES

3.1. Name Server Operation Overview

In this section we present a general description of the operation of a name server and identify a number of issues to be discussed in later sections.

When a name server is first started, it reads in a boot file which specifies all the zones that this server has, followed by a number of resource records to be loaded as the initial contents of the cache database. These records usually provide information about the name servers of the higher level domains so that this server can forward a query that it is unable to answer. Then the name server loads its zone databases in order. For a primary zone, its backup file specified in the boot file is used as the source of the data. For a secondary zone, the server tries to get data from the zone's primary server. After all zones are taken care of, the name server creates two communication ports with a well-known port number. One is a datagram port for receiving user queries; the other is a virtual circuit port for receiving maintenance queries, that is, initial zone transfers and zone refresh queries. Periodically the name server is interrupted by a timer that is set at the server initialization time and determines whether the dumping intervals for its primary zones or the refresh intervals for its secondary zones have expired. If so, it acts accordingly. It should be noticed that each zone has its own dumping or refresh interval. Appendix III provides an annotated list of the name server's program modules and functions.

3.2. Name Server Initialization and Data Structures

In order for a name server to be able to start its operation, a minimal amount of information about its environment has to be provided. This is included in the boot file, which is read in when the name server first starts. Some data structures are initialized that will later direct all of the name server operations. The specification of the boot file's format is given in Appendix I. For each zone, a record structure is created that includes all the relevant information about the zone.

3.3. The Process Structure of Name Servers and Its Impact on Server Operations

As shown in the above description, a name server has to carry out the following three activities:

1) user query processing: both retrievals and updates;
2) zone data maintenance: periodically writing primary zones to backup files and refreshing secondary zones;
3) name server maintenance query processing: both zone transfers and incremental refreshes.

Ideally, we would like to have multiple processes cooperating to perform these operations; for example, there could be one for user query processing, one for maintenance query processing, and a third for zone data maintenance. In this way, we would achieve concurrency and thus high availability of naming services because user queries and server refresh queries can still be processed during maintenance operations. Such cooperation requires that the processes be able to share zone databases and name server data structures. For example, the user query process will perform user updates and store them in the update list to be used later by the maintenance query process for propagation. Unfortunately, this is very difficult to implement in Berkeley Unix [Leffler *et al.* 83], since each process has its own private address space, and thus no memory sharing between processes is possible. Communication between processes can be realized by shared files or by messages, but they either fail to meet our needs or are too inefficient. For simplicity, we chose to use a single process for a name server and to multiplex the various activities within that process. Although some degree of parallelism can still be achieved by forking off a child process to perform a specific task while keeping the parent available to user queries, this parallelism is bound to be very limited because the child now has a copy of its parent's address space and therefore any change in the parent's or child's address space will not affect the address space of the other. Moreover, a "*fork*" operation in Unix is fairly expensive for the whole address space of the parent has to be copied over. The newer system call "*vfork*" is not appropriate here. Although the address space of the parent process is not fully copied with "*vfork*", no parallelism can be achieved as the parent process will be suspended while the child process is using its resources.

The decision to use a single process for the name server has important implications on its behavior. For example, maintenance queries and user queries cannot be processed simultaneously. A name server usually waits for a query to arrive by initiating the blocking call "*select*". When the call returns, the server determines what kind of query has arrived and proceeds to serve it. If both maintenance and user queries are waiting, maintenance queries are taken care of first, since they are considered more important. It should be noted that a maintenance query may take a long time to process, especially an initial zone transfer. But since during normal operation zones at different servers need only be refreshed, and the number of updates in each response to a refresh query is not likely to be large if updates are infrequent, we do not expect to suffer the serious degradation in user query processing that would be caused by long delays.

However, queries may be delayed when a name server is performing zone data maintenance. As mentioned above, a name server is periodically interrupted to maintain its zone data. For a primary zone, its dumping interval is decremented by the interrupt interval and if the value becomes negative, the whole zone is dumped to the backup file. This is a time consuming operation if the zone contains a large number of resource records. Similarly, for a secondary zone, its refresh interval is decremented and a refresh query is sent to the primary server if the value becomes negative. This involves requesting a connection, sending out the query, waiting for the response, and performing all the updates in the response. Again, this may take quite some time. Such operations have to be performed on every zone and, in the mean time, all the incoming queries are queued up. Fortunately, we expect that zone data maintenance operations will be performed at relatively long intervals, and hence their impact will be felt only occasionally.

As discussed above, we recognize the constraints imposed by the single process structure of the name servers; however, since such a decision is based on the operating system characteristics, there seems little that we can do about it. If in the future shared memory between processes is implemented in Berkeley Unix, the name servers may be reimplemented with a relatively small amount of effort to take advantage of that feature. We feel that the constraints of multiplexing several activities within a single process is tolerable. To verify this, performance measurements in a production environment are needed but are unfortunately unobtainable at present, since no applications of the name servers have been implemented.

3.4. The Update List and Its Management

An update list is created and maintained by a name server for each of its primary zones to record all the recent updates to be propagated to the secondary servers. There are two requirements for its data structure:

1) It should be flexible enough to accommodate the fluctuating rates of update queries without losing any of them.

2) It should not waste too much space. The space occupied by updates that are already propagated to all the secondary servers should be reclaimable.

The first data structure we considered was a cyclic list. This is a static data structure containing a number of records for storing updates. A pointer to the next available record is kept. When reaching the end of the list, this pointer wraps around, overwriting old records with new ones. The size of such a list is difficult to decide. If it is too small, we run the risk of overflowing the list. If it is too large, memory is wasted. Clearly, a cyclic list does not meet the requirements stated above.

The data structure we used in the Berkeley implementation is shown in Figure 4. It is essentially a chain of structures, each of which contains a number of update records. This chain grows and shrinks as new updates come and old updates go. The primary name server maintains a pointer in the corresponding zone record, called the *primary pointer*, which points to the next available location for updates, as well as a number of pointers, one for each secondary server, to remember where the last refresh operation of that secondary server stopped. Such a pointer is called a *secondary pointer* and is initialized to the primary pointer when an initial zone transfer is performed for this secondary server. When the secondary pointer that lags behind the most passes over a structure, the space occupied by the structure is reclaimed.

When first started, the primary server has no knowledge about the secondary servers except for the NS type resource records in the zone. At the time when an initial zone transfer request

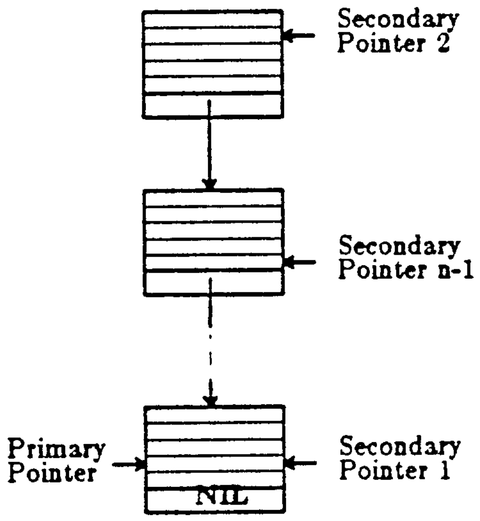

Figure 4. Data structure for the update list of a zone

comes, the primary server checks whether there is an NS resource record with the data field being the secondary name server domain name provided in the query. If so, the zone is transferred, and a new record is created for this secondary name server to store its secondary pointer. Such records form a chain, and a pointer to this chain is kept in the zone record. For every maintenance query, this chain is searched to get the appropriate pointer into the update list. More uses of the chain of records will be discussed in the next section.

3.5. Recovery in Case of Host Crash

A difficult problem that distributed systems designers have to solve is that of recovery from network partitions due to failures in communication media and/or host crashes. The LOCUS project at UCLA, for example, spent large amounts of time and talent trying to solve such a problem for their distributed file systems [Walker *et al.* 83]. Fortunately, in our case, the clear distinction between the primary server and the secondary servers for a zone greatly simplifies the problem. Below we study the problem separately for primary and secondary servers.

3.5.1. Crash Recovery of Primary Name Servers

When the host on which a primary server for a zone runs crashes, the server dies, but the zone database is usually preserved on disk. The secondary servers for the zone will keep trying to connect to the primary server at every refresh interval for the zone. After a certain number of unsuccessful tries, the secondary servers will understand that the primary server must be down. In such a case, the secondary servers should move the whole zone to their cache databases and use a timeout mechanism to delete them eventually. A secondary server, however, should not give up the zone completely. It should keep trying to connect to the primary server, probably at much longer intervals than those used for the refresh operation. Once the primary server is restarted after a host crash, it will check for the existence of the zone database, and will not perform an initial loading from the backup file if the database already exists. The next connection attempt by a secondary server will succeed, and the whole zone will be transferred, thus resuming normal service. It should be noted that a whole zone transfer has to be performed in this case since the primary server might have lost some of the recent updates that have been propagated to the secondary servers. This may happen if the database on the permanent storage is badly damaged or the most recent updates have not been flushed out to the permanent storage. Therefore, to achieve consistency, all the secondary zones should be reinitialized to the primary zone, even if this means loss of some updates.

It is possible that the host of a primary name server experiences a short breakdown, and that the secondary servers do not notice it. When the primary server receives the next refresh query from a secondary server, it finds that it does not have such a secondary server on its chain of secondary servers for the zone since this chain was in memory and therefore was lost in the breakdown. In this case, the primary server refuses the query and forces the secondary server to perform a whole zone transfer. The old copy of the zone database in this secondary server should be replaced. This mechanism also prevents a malfunctioning secondary server from trying a refresh operation before a whole zone transfer.

3.5.2. Crash Recovery of Secondary Name Servers

Recovery of a secondary name server is much simpler. When restarted, such a server performs a whole zone transfer to replace its old zone database. While zone refresh before zone transfer is prohibited by the primary server, multiple zone transfers are allowed since a secondary server may experience a brief breakdown that is not detected by the primary server. Again we notice here that every secondary server is responsible for the maintenance of its own zones, and the primary server simply checks the validity of a maintenance query when it arrives, but makes no active effort to keep the secondary servers up to date. This is a form of distributed management of replicated data, and is designed to relieve a primary server from part of its burden.

4. FUTURE WORK

4.1. Authentication and Access Control

Name servers store and manage critical information in networks, and therefore the access to server databases must be well controlled. Authentication deals with the verification of the real identities of users. The simplest example of authentication is the user name-password mechanism that most computer systems use to verify the identities of their users. Access control specifies the information that a user is authorized to access, and the types of permitted accesses. In an academic environment, it can be expected that most of the naming information, like host addresses, user mailboxes, and server ports, may be readable by all users, hence the read access control is not a serious problem; however, with user update queries implemented in the BIND servers, authentication and access control become indispensable.

No authentication and access control mechanisms have been implemented in the BIND servers, but this must be done to make these servers usable in a production environment.

4.2. Caching

Internet naming services will be widely used for many applications, and information about all the objects in the whole DARPA internetwork will be available to any authorized user. This may mean a large volume of remote access traffic. Every time a student at Berkeley, for example, wants to send mail to MIT, the remote MIT name server has to be queried to get the location of the destination mailbox. As in other information systems, for example in the Grapevine system developed at Xerox PARC [Birrell *et al.* 82], we can expect locality in naming information accesses [Terry 84]. There are two aspects of locality. One respect is temporal locality, that is, information that is being used is likely to be used again in the near future. The other is spatial locality, that is, information near that currently being accessed is likely to be needed soon. Considering the large ratio between the time for a remote access over a long-haul network and that for a local access, it makes sense to cache frequently used remote information in local name servers for improved performance.

Two problems arise with respect to caching. The first has to do with the granularity. The smallest cachable unit is a resource record. Other choices include a node in the domain space, a whole zone, or a group of zones. The second issue is the management of cached data. It should be pointed out that caches in name servers are usually read-only, and that there are no rigid limits on their sizes as in hardware caches. Below we investigate several approaches to caching.

4.2.1. Secondary Zones

As we observed in Section 2, the replication of a zone on several name servers is a form of caching. Each of the secondary servers gets the zone from the primary server and thereafter refreshes it from the primary server. Here the granularity of caching is a whole zone, and the cached data is managed by the refresh scheme. This form of caching should be used with prudence, for the cost of maintaining a secondary zone is relatively high. A situation in which this type of caching is appropriate is that of two organizations, each having its own zone and having strong ties of cooperation, for example, two research facilities of a big corporation, or two universities with extensive academic exchanges. Other factors also have to be considered in establishing secondary zones, e.g., organizational structures and accessibility requirements.

4.2.2. Database Shared with the Resolvers

Caching can be done in the resolvers; resource records returned from queries are both returned to the users and stored in a local cache. Painter discusses this subject in [Painter 84]. Caching in resolvers is bound to be limited, since a resolver is a collection of subroutines to be linked into user programs that use naming services, thus, caching is done on a per-process basis. In RFC883, Mockapetris proposed the use of a database shared between a name server and a resolver. The shared database contains both zones that are maintained with a refresh scheme and resource records acquired by the resolver and maintained by a timeout mechanism. We feel that such a shared database approach complicates the clean functional separation between the resolvers, seen as local agents that make queries on behalf of the users, and the name servers, which are system processes providing naming service; furthermore, this complication does not seem to be accompanied by any obvious benefits.

4.2.3. Caching by Recursive Query Processing

We mentioned in Section 2 that user queries are usually processed in an iterative fashion. If the first server being asked cannot answer a query, it redirects the resolver to another server, and will not involve itself in resolving that query any longer. Thus, the next time the server receives the same query, the same expensive iterative steps will have to be performed again. Based on this observation, it is reasonable to suggest that the resource records that the resolver finally receives as the answer be stored in the cache of the first server. One simple way to do this is to have the resolver send the records to the first server. One more message is needed to do this. Moreover, the roles of resolvers and name servers become more complicated. In addition to regular queries, a resolver can send information to a name server. Whether this method can provide better performance depends on the user's access behavior. If there exists a strong temporal locality, this method may appreciably reduce remote query traffic. Otherwise, the cost of one more message per remote query may outweigh the occasional benefits. The advantage of this method is obviously its simplicity.

A more elegant way to do caching is by recursive query processing. Instead of returning a forwarding reference, the first server without the required information will track down a chain of servers to find this information for the resolver. The resource records are also stored in the local cache. This means that a name server has to be able to act as a resolver too.

While being conceptually elegant, recursive query processing has two problems. First, the response time that a resolver can expect for a query is more unpredictable. If the requested information is stored in the server that is first queried, a response is usually returned quickly. If, however, this server does not have the information, then it will query other servers trying to resolve the query. In this case, the resolver has to wait a variable amount of time without knowing what is going on. Secondly, for an implementation like ours in which a single process is used for each server, the potentially long delays due to recursive query processing may be unbearable. Forking a child process does not help much in this case, for the two processes will have completely separate address spaces.

From the above discussion on the possible ways to do caching, we conclude that there is no single method that satisfies all our requirements. A combination of them may give better results. Caching is a difficult problem in general, but is probably indispensable to achieve acceptable performance in a production environment. As a research topic in its own right, caching should be given a substantial amount of attention.

4.3. Possible Applications

BIND servers are intended to be a general purpose facility that can incorporate many types of applications. For each application, specific user interface programs will be written that invoke naming services. As new applications are added, the name servers should remain relatively stable. Below we discuss a few of the possible applications, most of which are replacements and extensions of currently available system services, often implemented in an *ad hoc* manner. We are confident that as experience with name servers and their uses grows, more applications will be proposed and implemented.

4.3.1. Internet Host Address Binding

The classical use of name servers is to perform the mapping from host names to their addresses. Currently, a complete table of all the hosts in the DARPA Internet and their addresses is maintained by the Network Information Center and distributed to every host. When the address of a host is needed, the local copy has to be searched. Name servers can easily perform this function in a distributed manner by assigning each host to a domain in the domain space and storing an address type resource record for it. The lookup of an address is a simple name server query. Local servers can cache frequently used remote host addresses to improve performance. Similar information, such as the network addresses of file server and print server ports, can also be maintained.

4.3.2. User Information and Mailboxes

General information about users, such as their addresses, telephone numbers, and job titles, is often useful. In 4.2BSD, such information is stored in the password file and accessed by the "*finger*" command. We can create a new resource record type, say UINFO, and store the above information into these records. For each user, a new domain within his/her organization can be created, probably with his/her login name. Organizing users according to administrative structures detaches them from specific hosts and therefore offers greater flexibility and ease of use. Similarly, information about user mailboxes, mailing lists, and user aliases can also be supported by name servers.

4.3.3. Process Rendezvous

Interprocess communication can be assisted by name servers. Processes willing to communicate with each other may register themselves with name servers and find potential partners there. A process can be represented by the host on which it runs† and its communication port number. The inclusion of relatively transient information like that about processes poses new challenges to the name server's database management. Timeliness of the information now becomes critical. We have not implemented process rendezvous, and much study is called for in this area.

4.4. Performance Measurement and Evaluation

Many of the design choices we made for the BIND servers were based on assumptions about the working environments of the naming service, and the naming information access patterns. Although these assumptions seem reasonable, they must be verified using measurement data

†Here we assume that a process resides on one host throughout its life time. A more sophisticated scheme is needed if process migration is allowed.

derived from production environments. Unfortunately, no application using the BIND servers has been implemented yet; thus, a comprehensive evaluation of the server's performance has not been possible.

5. CONCLUSIONS

We have described the design and implementation of the Berkeley Internet Name Domain servers. A number of important design decisions have been made, including the incorporation of interactive user update queries, the distinction between primary and secondary name servers, which is useful in update query processing, the management scheme for replicated zones, the protocol for incremental zone refresh, and the elimination of master files for zone data. These decisions were based on experiences obtained from other distributed information systems and on intuitive arguments. While the decisions seem reasonable, their validity must be and can only be tested by measurements of the naming service operating in a production environment. Even though the above decisions represent departures from the specifications in RFC882 and RFC883, compatibility nevertheless must be maintained so that name servers of different implementations can communicate with each other. This was achieved by implementing two sets of routines for maintenance queries, and automating the switching between different protocols. The crash recovery problem has been dealt with in a simple manner and the adequacy of the solution is to be verified. Throughout the implementation of the name servers, the constraint of using a single process for each server has been painfully felt. It may still be early to fully appreciate its influence.

The basic name servers are running, and the proposed design has been proved to be sound. We have made the very important first step towards an extensive, easy-to-use computer network naming service. With the backbone established, various applications can be built, and the name servers used in production environments. The experiences to be derived from using the name servers will shed light on the best direction for future work.

A number of important issues are still to be resolved, however. Since the user update queries have been implemented, the authentication and access control problems must be solved in order for the naming service to be usable in the real world. Caching schemes are to be designed for the purpose of achieving satisfactory server performance in a widely distributed internetwork environment. Once the name servers go into regular use, their performance should be measured so that various design and implementation decisions can be evaluated, and the servers can be tuned for optimal operation. Great efforts are still needed to make the BIND servers an integral part of a distributed computing environment.

6. ACKNOWLEDGEMENTS

I would like to thank Professor Domenico Ferrari, my research advisor, for his support of this project and for his guidance. Special thanks are due to Professor Luis F. Cabrera, my second reader, whose many criticisms and suggestions have made this report more readable. Douglas Terry motivated the BIND server project and provided invaluable advice throughout its course. The discussions with Mark Painter and David Riggle have been very helpful. The financial support of the Defense Advanced Research Projects Agency is gratefully acknowledged. The responsibility for any errors in this report lies with nobody else but myself.

7. REFERENCES

[Almes and Lazowska 79]
Almes, G. T. and Lazowska, E. D., "The Behavior of Ethernet-like Computer Communications Networks," *Proceedings of the Seventh Symposium on Operating Systems Principles*, Pacific Grove, California, Dec. 1979, pp. 66-81.

[Birrell *et al.* 82]
Birrell, A., Levin, R., Needham, R. M., and Schroeder, M. D. "Grapevine: An experience in Distributed Computing," *Comm. of ACM* 25, 4, pp.260-274.

[Leffler *et al.* 83]
Leffler, S., Joy, W., and McKusick, M. "4.2BSD Berkeley Unix Programmer's Manual," July 1983.

[Mock 83, 1]
Mockapetris, P. "Domain Names - Concepts and Facilities," RFC882, USC/Information Sciences Institute, November 1983.

[Mock 83, 2]
Mockapetris, P. "Domain Names - Implementation and Specification," RFC883, USC/Information Sciences Institute, November 1983.

[Painter 84]
Painter, M. "The Design and Implementation of a 'Domain Names' Resolver", Master Report, Computer Science Division, Univ. of Calif., Berkeley, May 1984.

[Pickens *et al.* 79]
Pickens, J. R. Feinler, E. J., and Mathis, J. E. "The NIC Name Server-- A Datagram Based Information Utility," *Proc. of Fourth Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1979, pp. 275-283.

[Postel 80]
Postel, J. "User Datagram Protocol," RFC768, USC/Information Sciences Institute, August 1980.

[Postel 81]
Postel, J. "Transmission Control Protocol," RFC793, USC/Information Sciences Institute, September 1981.

[Riggle 84]
Riggle, D. "A Name Server Database, Master Report, Computer Science Division, Univ. of Calif., Berkeley, May 1984.

[Su and Postel 82]
Su, Z., and Postel, J. "The Domain Naming Convention for Internet User Applications," RFC819, USC/Information Sciences Institute, August 1982.

[Terry 84]
Private communication, 1984.

[Walker *et al.* 83]
Walkers, B., *et al.* "The LOCUS Distributed Operating System," *Proc. of the Ninth Symp. on Operating Systems Principles*, Bretton Woods, New Hampshire, October 1983, pp. 49-70.

APPENDICES

Appendix I:

Boot File Format

The information included in the boot file for a name server is purposely kept at a minimum. A boot file for a name server consists of three sections. The first section is simply one line specifying the domain name of the host on which this name server runs and the time interval at which the name server is interrupted for maintenance operations. The name server host domain name is needed when making maintenance queries. The second section lists all the zones for which this server is authoritative. For each zone, the first line specifies the domain name of the zone origin, the name of the zone database, and the keyword "PRIMARY" or "SECONDARY" to indicate whether this server is the primary server for this zone or not. For a primary zone, the next line specifies the name of its backup file. For a secondary zone, the next line specifies the domain name of the primary server's host and its address.

The three sections are separated by lines starting with a "$" sign. Lines starting with spaces or semicolons are ignored and strings are freely separated by spaces and/or tab characters.

An example of a boot file is listed below.

```
; boot file for name server with
;       secondary zone 'arpa', and
;       primary zone 'ucb.arpa', and
;       secondary zone 'cc.ucb.arpa';
; name server located at domain 'ucbarpa.ucb.arpa' and
; interrupted every 5 minutes
;
ucbarpa.ucb.arpa        300
$
arpa        arpadb      SECONDARY
ucbcalder.ucb.arpa      128.32.0.12

ucb.arpa    ucbdb       PRIMARY
ucbfile

cc.ucb.arpaccdb SECONDARY
ucbmonet.ucb.arpa       128.32.0.7
$
;
; followed by resource records to be loaded into the cache database,
; omitted here.
;
```

Appendix II:

Maintenance Query Formats

For uniformity, we adopt, for maintenance queries, formats similar to that for user queries, but with some modifications. The query identification number and some of the flags in the header are not needed because we use a reliable virtual circuit protocol for maintenance operations. The opcode is used, however, to specify whether this is an initial zone transfer or an incremental refresh. Two new opcodes, ZONEINIT and ZONEREF are defined, respectively. Following the header are not the question section or resource records but rather two domain names. The first is the origin of the zone required, and the second is the secondary server's host domain name.

The format of a response to an initial zone transfer and to an incremental refresh are slightly different. For an initial transfer, the response consists of a header and a number of resource records in the answer section. For an incremental refresh, the response consists of a header and a number of update records, each being simply a resource record preceded by a code specifying the type of update. Both of the old and the new records are transmitted for a modification.

The response code in the header may be one of several values. If everything is correct, it should be "NERROR". If the primary server is unable to answer the query due to internal failures, "SERVFAIL" is returned. If the primary server is not willing to provide the zone or its updates, "REFUSED" is returned. It is possible that the resource records or the updates of a zone cannot fit into one message. In this case the truncation flag "tr" in the header is set and multiple messages will be sent for the zone.

Appendix III:

An Annotated Listing of Name Server Program Modules and Functions

nameser.c:

This module contains the main function of the name server.

"*main*":

It reads in the boot file to initialize the name server data structures, loads or transfers zone databases, creates virtual circuit and datagram ports, sets interrupt interval, and then starts serving queries. All interprocess communication protocols are handled here.

query.c:

This module includes all the functions for user query processing.

"*proc_query*":

It parses a query, searches the appropriate zone database or cache, and assembles the response.

"*check_auth*":

It checks whether or not this server is authoritative for the domain name in query and returns a code indicating one of several conditions:

–authoritative and zone loaded;

–authoritative but zone not loaded;

–authority delegated to a zone not stored in this server;

–no appropriate zone found, cache to be searched.

"*search_cache*":

It searches the cache database for pertinent resource records and stores them into response.

"*forward*":

It puts forwarding NS and A resource records in response, used for iterative query processing.

"*pforward_q*":

It forwards a query to the primary name server.

"*aforward_q*":

It forwards a query to another name server.

"*store_rr*":

It stores a resource record derived from zone database into response message.

"*update_init*":

It performs initial steps of update query processing, including query parsing, authority checking, and query forwarding if not the primary server.

"*rec_update*":

It records an update in the zone update list to be transferred to secondary servers in refresh operations.

dbrts.c

This module contains all the functions for name server maintenance and maintenance query processing.

"*boot_ns*":

It reads in the boot file and initializes all the server data structures.

"*maintain*":

This is the name server interrupt handler. It checks dumping intervals for primary zones and refresh intervals for secondary zones, and invokes appropriate functions accordingly.

"*refresh_zone*":
It requests a virtual circuit connection to the primary server. For an initial zone transfer, it gets the whole zone from the primary server. For an incremental refresh, it gets the recent updates from the primary server.

"*zone_init*":
It loads the resource records in a zone transferred from the primary server into the zone database.

"*zone_ref*":
It refreshes the zone by performing all the updates received from the primary server.

"*zone_updates*":
It serves a maintenance query from a secondary server. If initial transfer is requested, it sends all the resource records in the zone. If incremental refresh is requested, it sends all the recent updates.

"*store_zone*":
It stores the resource records in a zone into response to be sent to a secondary server.

"*store_updates*":
It stores the recent updates of a zone into response to be sent to a secondary server.

"*isi_zonetrans*":
It serves a maintenance query from a secondary server of the ISI implementation (to be implemented).

"*store_domain*":
It stores all the records of a domain into the response (to be implemented).

"*isi_getzone*":
It sets up a virtual circuit with a primary server of the ISI implementation and gets a zone (to be implemented).

"*load_domain*":
It loads into the zone database all the resource records of a domain received from a primary server of the ISI implementation (to be implemented).

funcs.c:
This module contains all the basic utility functions to be used by the above modules.

"*dn_compare*":
It compares two domain names and returns the number of matching labels. Zero is returned if the second name is the root (null domain), and minus one is returned if it is not a substring of the first name.

"*filldata*":
It stores the data section of a resource record into a message according to specified format decided by record type.

"*conv_data*":
It converts the data section of a resource record from the message format into the string format to be stored into the zone database.

"*check_class*":
It checks the validity of a resource record class value.

"*check_type*":
It checks the validity of a resource record type value.

debug.c:
This module contains all the functions for name server debugging, mainly functions that print out relevant information.

"*print_ns*":
It prints out all the name server data structures.

"*pup_list*":
It prints out the content of the update list for a zone.

"*print_p*":
It prints out a query or response.

"*print_q*":
It prints out the question section of a query or response.

"*print_ud*":
It prints out an update record in a query or response.

"*print_rr*":
It prints out a resource record in a query or response.

compress.c:
This module contains functions that operate on domain names. It is written by Mark Painter and used by both the resolvers and the name servers. Irrelevant functions are omitted.

"*dn_skip*":
It returns pointer to the first byte after a compressed domain name.

"*dn_expand*":
It converts domain name from the compressed message format into the string format.

"*dn_comp*":
It converts domain name from the string format into the compressed message format.

Appendix IV:

Currently Defined Resource Record Types and Their Encodings

(Adapted from RFC883)

TYPE	VALUE	MEANING
A	1	a host address
NS	2	an authoritative name server
MD	3	a mail destination
MF	4	a mail forwarder
CNAME	5	the canonical name for an alias
SOA	6	marks the start of a zone of authority
MB	7	a mailbox domain name
MG	8	a mail group member
MR	9	a mail rename domain name
NULL	10	a null resource record (RR)
WKS	11	a well known service description
PTR	12	a domain name pointer
HINFO	13	host information
MINFO	14	mailbox or mail list information