The Design and Implementation of a "Domain Names" Resolver†

Mark Painter

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

ABSTRACT

The requests for comment 881, 882 and 883 issued by the Information Sciences Institute address specific and impending needs of the ARPA/DDN internet community for distributed name services. The realization of such services at Berkeley is called the Berkeley Internet Name Domain (BIND). This report describes an implementation of the portion of BIND called the resolver. The resolver is a local agent which is responsible for retrieving information on behalf of the user. Thus, the user interface, inter-machine communication, and, to some extent, the local storage of data, are all functions of a resolver.

During our implementation of name services, we decided to extend the original proposal to allow updates of data by users. Allowing user updates is expected to greatly increase the number of applications for domain style names. Our support for updates includes some straight-forward additions to the protocol that is used to implement name services, and it includes some additions to the functions offered to users by resolvers. These suggested improvements are among the subjects of this report.

1. Introduction

The ideas which are presented in the requests for comment 881, 882 and 883 will be collectively referred to as the "domain names proposal". This title is somewhat arbitrary, and refers to the style in which objects are to be named.

The immediate concern of the domain names proposal, is the impending collapse of the methods which are currently used for distributing name and address information among ARPA/DDN internet users. At the present time, the network information center compiles and distributes a list of all host addresses. There is no uniform method for locating a specific user's mailbox address, nor is there a uniform method for determining which services are supported by a given host. The host table itself is becoming large enough that its management consumes significant resources.

In the broadest terms, the strategy for dealing with these problems in the future is to distribute both the physical location of the data, and the responsibility for its administration. Clearly, the most frequent requests for information will be within organizational boundaries, and between particular organizations which have frequent contact with each other. This suggests distributing

†This work was sponsored in part by the State of California and the NCR Corporation under MICRO grant No. 1-532436-19900, and in part 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.

the data according to natural administrative boundaries. The domain names proposal is a protocol specification for such a distribution of address information.

Distributing data complicates its retrieval. As the data may not be present on the local host, there must be a means of retrieving it from remote hosts. This implies that there must be some method of locating an appropriate remote host. Since the cost of establishing a virtual circuit between processes is believed to be greater than the cost of using "best effort", datagram communication, an unreliable communication mechanism is assumed. Thus, during the processing of a request there may be lost or duplicate communications. For reasons of efficiency, it may be desirable to make local copies of frequently requested data. Also, in the process of locating an appropriate remote host, it is important to detect referrals to hosts which have already been queried. Thus, it is desirable to record what information is requested and the responses received.

These problems are significant enough to warrant a specialized mechanism for their solution. A resolver is such a mechanism. As the resolver is the means by which users will retrieve information, it necessarily includes a user interface. In our case, *users* are C language programmers. The interface to our resolver is a library of subroutines which insulate users from the details of the implementation of name services. These subroutines offer, among other things, the retrival operations which are supported within the database: specifically, *standard queries, inverse queries* and *completion queries* (to be defined in Section 2.2.2.1).

The proposal attempts to be general enough to support future applications, as well as, solve current problems. To this end, there are features in the proposal for adding yet to be defined data types, and there are features to facilitate the crossing of protocol boundaries. Noticeably absent from the proposal are features for user level applications of name service. For example, facilities for user updates, and support of process to process level protocols are lacking. Section 2.2.2.2 of this report will discuss extensions to the proposal to support those features. This support includes three new types of query: *addition queries, deletion queries* and *modification queries.*

The remainder of this report is divided into six major sections. Section 2 provides the background for the project. Section 2.1 discusses the organization of the documents which describe the original proposal. Section 2.2 is a description of what services were to be provided by the implementation. This includes both the original proposal, and ideas for its extension. Section 2.3 discusses the relationship between the user's view of services and efficiency and reliability. Section 3 is a statement of the goals of the implementation. Section 4 is a discussion of the current implementation of the resolver. Section 4.1 describes the process structure of the resolver. Section 4.2 is a concrete description of how the current implementation presents name service to the user. The discussion of the implementation will then proceed to issues which do not directly concern the user. Section 4.3 describes the implementation of the protocol and the design choices made in the relationship of resolvers to name servers. Section 4.4 describes the internal details of the resolver's implementation. Section 5 is an attempt to objectively evaluate the utility of the current design. Section 6 discusses future work. Finally, Section 7 is a summary of what was accomplished and what was learned during the project.

2. Background

This section discusses what was to be implemented. The discussion focuses on the abstractions which were to be realized. The realization of these abstractions is described in Section 4.

2.1. Content of Documents Which Describe the Original Proposal

There are three documents on which this work is based. These documents are requests for comment 881, 882 and 883 which were issued by Information Sciences Institute. In the remainder of this report they will be referred to as "rfc's". Since they are fundamental to this report, their organization will be briefly discussed here. Sections 2.2.1.1 and 2.2.1.2 summarize the aspects of these papers which are relevant to the resolver.

2.1.1. Request for Comment 881

Rfc 881 discusses both the need for conversion to the domain names system, and the administrative details of the conversion. The process of conversion is a formidable task. Its difficulty should not be underestimated. However, this has little direct bearing on the design of the resolver. The most compelling message of this rfc is the need to quickly get an effective mechanism operational.

2.1.2. Request for Comment 882

Rfc 882 is a general description of the domain names proposal. The level of the discussion is roughly comparable with the level of the discussion in Section 2.2. If this report should fail to illuminate the reader, rfc 882 is a good second source for the general concepts involved.

2.1.3. Request for Comment 883

Rfc 883 is an elaboration of rfc 882. It contains a preliminary specification of the name server protocol. Therefore, it was the document which was most closely consulted during the implementation. Aside from giving a concrete specification of the name server protocol, rfc 883 contains suggestions for the implementation of name servers and resolvers. The resolver which is described in this report deviates from some of those suggestions. This is discussed in Section 4.3.2.

2.2. Overview of Proposed Design

This section discusses both the design as was proposed in the rfc's, and extensions to the design. Every effort has been made to distinguish my ideas from the domain names proposal. However, in discussing what has been implemented the two are necessarily intertwined.

There are two aspects of the domain names proposal which have a particularly strong influence on the implementation of the resolver. First, the way in which data is organized and maintained affects the mechanics of the resolver. Second, the services which are supported by the database determine what services the resolver can offer the user. These two aspects are discussed separately.

2.2.1. Name Space

A *domain name* corresponds to some logical entity. For example, a person, a particular machine, or an administrative boundary. Data which is appropriate to the object is associated with the domain name. This data may be specified with the domain name and an indication of the "kind" of data which is desired.

Domain names are organized into a tree structure. A domain name is, in fact, a sequence of labels which specify the path which leads to its node in the tree. Each label in the sequence corresponds to a more specific *domain*. Although it is expected that the structure of the domain tree will reflect the administrative and geographic structure of the ARPA/DDN internet community, the correspondence of domains to entities is a database design problem, and any binding of domains to internet addresses is accomplished with the data that is associated with a domain name.

Programs which are responsible for maintaining some portion of the database are called *name servers*. Each name server is associated with a domain name. Name servers are responsible for some, perhaps truncated, subtree of the name space, called a *zone of authority*. The root of a zone of authority is the domain name associated with the name server. Thus, it is possible to determine if a domain name is in or below a zone of authority by comparing it with the domain name of the name server. Each name server has name and address information for the name servers which are responsible for the branches of the tree which are subtrees rooted in its zone of authority. Each name server also has name and address information for name servers above it in the tree, at least the name servers for the root. It is possible for name servers to cache information which is not in their zone of authority. It is also possible for a zone of authority to be maintained by more than one name server, and for a name server to maintain multiple zones of authority. It is even possible for a degenerate name server to contain only cache data.

The essential point is that the data is structured so that it is always possible to determine if a particular name is within one of the zones a particular name server is responsible for. If the data is not in one of the name server's zones, it is always possible to locate a more appropriate name server. If the name of an object is known, then the general strategy for retrieving information about the name is as follows.

The requesting agent queries some known name server. This name server then returns the requested information, if it has it. Otherwise, it returns the name and address of a name server which is closer in the domain tree to the requested information. This process is iterated until the information is located, or determined to be non-existent. This requesting agent is, of course, the resolver, which is the principal subject of this report.

There are other features in the proposal which complicate this basic design. They will be elaborated throughout this report. However, at this point it is important to understand the spirit of the proposal, but not necessary to know all of its details.

2.2.2. Services Available to the User

What follows is a description of what services are available to the user. At first, this discussion will be in general terms, and will not include specifics of the current implementation. Section 4.2 will be a more implementation specific discussion of how these services are presented to the user.

Although the boundaries are sometimes blurred, it is useful to distinguish three categories of service. There are services which are mandated by the domain names proposal. There are services which are proposed as extensions to the domain names proposal. Lastly, there are services which affect the performance and reliability of the resolver, but are to some extent independent of the basic update and retrieval operations.

The distinction between the specification and its extensions is important to the general discussion, but both were implemented. So, the implementation specific portion of the discussion in section 4.2 will only make distinctions between function and policy.

2.2.2.1. Services in the Domain Names Proposal

The services which are mandated by the domain names proposal are three types of queries for interrogating the name server database. These query types are standard queries, completion queries, and inverse queries. In fact, only standard queries are required by the proposal. The other query types are simply recommended. If they are successful, all of these queries will return items of data called *resource records*.

2.2.2.1.1. Standard Queries

Standard queries form the backbone of the domain names system. A user specifies a domain name, the *type* of data desired, and the *class* of the data. In the absence of unrecoverable network failures, the resolver will either be able to return the requested information, or inform the user that the information does not exist. The method of locating an initial name server for the resolver to try is, however, left to the resolver's implementor.

Here classes roughly correspond to host to host level protocols. Types correspond to certain generic kinds of data, such as addresses, user mailbox names, and domain name aliases. The format of the data portion of resource records is both type and class dependent. Also, for a given domain name, there may be more than one resource record of the same type and class.

When requesting data, it is possible to specify a wild card in the type field of the request. For example, it is possible for a user to request all of the resource records which are associated with a domain name and and which have a type field pertaining to mailboxes.

Domain names are a sequence of labels. The rightmost label in the domain name is the root of the domain tree. The leftmost label is the most specific. Thus, domain names are read in the opposite order from UNIX† path names. It is recommended that labels be composed of sequences of letters, digits and hyphens, starting with a letter, where case is not significant. Labels can be at most 63 bytes in length. In their external representation, the labels in a domain name are separated by dots. Internally, each label is a byte which specifies the length of the label, followed by arbitrary data. The root is signified by a label of zero length. Domain names terminate when the root label is found. Normally, the next label starts where the previous one ended. However, the byte which specifies the length of the next label may be tagged to indicate using indirection to locate the next label. This indirection can be used to save space when several names have the same rightmost labels. Thus, the internal representations are called *compressed domain names*.

As was mentioned above, the items of information which are returned are called resource records. Each resource record contains the domain name which the data pertains to, the class of the data, the type of the data, the data's length, and a maximum time that the data can be cached, as well as the data itself. The additional fields are occasionally of interest to the user; for example, when the user requests multiple types of data to be returned by specifying a wild card in the type field of the request.

2.2.2.1.2. Completion Queries

Completion queries are identical to standard queries, except that the domain name needs to be only partially specified. Specifically, only the leftmost characters in a domain name are given to the resolver. The method of locating an appropriate name server for a completion query is left to the resolver's implementor. There is no guarantee that suitable data will be located even if it exists and there are no network failures. However, if completion queries are suitably restricted, they may be a great convenience to the user.

It is expected that the majority of queries will be for data which is local. However, full domain names may be many characters long. So, specifying a complete domain name with every request may be quite bothersome and often unnecessary. Completion queries provide a mechanism to support abbreviated requests. However, generating all possible suffixes may be prohibitively expensive. So, I recommend that the search be restricted to whole label matches and a few predefined label suffixes. For example, at Berkeley we might append "ucb" to any name used by a completion query for which no whole label match could be found. If that failed to produce a match, the search would be abandoned.

There are actually two types of completion queries. One type requires that the domain name which matches the name used by the completion query must be a unique best match. The other allows multiple domain names to match the name used by the completion query. Our implementation of the resolver does not emphasize this distinction, but allows users to specify that multiple matches will be allowed (section 4.2.2.1).

2.2.2.1.3. Inverse Queries

Inverse queries are unreliable in the same way that completion queries are unreliable. In an inverse query, the user specifies the data, its type and class. The resolver then attempts to locate a name server which is capable of determining the domain names which correspond to the data. This is another case in which fields of the resource records which are returned, other than the data field, will be of interest to the user. Once again, the method of choosing an appropriate name server is left to the resolver's implementor.

An example use of inverse queries would be for debugging purposes. If an erroneous communication is received, it might be desirable to present the source of the communication in human readable form. Similarly, inverse queries may be useful in locating erroneous entries in the name server database.

†UNIX is a trademark of Bell Laboratories.

The algorithms for standard, completion, and inverse queries are discussed in depth in rfc883 and rfc882. For the purposes of the present discussion it is merely necessary to have a general idea of their nature.

2.2.2.1.4. Authoritative Name Servers

The domain names proposal allows data to be cached for performance reasons, but there are no mechanisms for guaranteeing the instantaneous consistency of the database. It is expected that most applications of name service will be able to tolerate temporary inconsistencies in the database. However, there is a mechanism for users to specify that they wish to sacrifice performance in favor of reliability.

This mechanism is the notion of *authoritative name servers*. The idea is that each zone of authority will have a set of servers which are responsible for insuring the timelyness of their data for that zone. Other name servers may cache data from the zone, but they do not attempt to refresh the contents of their caches, nor do they necessarily maintain complete information for the entire zone. Thus, the data of non-authoritative name servers is more likely to be out of date, or incomplete. Therefore, users may request that their queries be answered only by authoritative name servers. It should be noted that specifying that the query must be answered by an authoritative name server does not absolutely guarantee that consistent data will be returned. This is because authoritative name servers periodically refresh their databases.

2.2.2.1.5. Primary Name Servers

There is a primary copy of the resource records in each zone of authority. The name server which is responsible for propagating this copy to the authoritative servers for the zone will be called the *primary server* for the zone. Authoritative name servers for a zone contain resource records which can be used to locate the primary server for the zone. In fact, authoritative name servers use this data to periodically contact the primary server, to ensure that their own data is up-to-date.

An important reason for the existence of authoritative name servers which are not also primary name servers is that, if the host of the primary should fail, the data for the zone will still be available. Additional authoritative name servers also increase the availability of data for a zone by distributing the burden of answering queries for that zone among several hosts.

The original proposal does not include explicit mechanisms for users to locate primary copies of resource records. However, the introduction of user updates increases the likelihood that the degree of consistency maintained by authoritative name servers will not be sufficient for all applications. Also, primary servers play a role in how updates are implemented. So, their existence is important to the discussion which follows.

2.2.2.2. Proposed Extensions

This section contains suggestions for modifying the name server protocol. The principal departure from the proposal is the way in which updates are handled. The extensions which are described here have been implemented within the resolver.

2.2.2.2.1. User Updates (Addition, Deletion, Modification)

The design proposed in the rfc's handles updates in a very limited way. Specifically, there are to be text files which will be updated by hand. These files will periodically be read by the primary name server for a zone. The data in the files will replace the primary name server's database. Other authoritative name servers will obtain the new data when they contact the primary name server, and replace their databases. This scheme avoids a number of difficult problems. However, it severely limits the generality of the name service. If each change must be cleared with a system administrator, it will be impossible to support the naming of short-lived objects, such as processes, and impossible to support the naming of objects based on personal preferences, such as files.

For that reason, our group has decided to implement user updates. Many of the problems encountered relate to the implementation of the name servers, and will not be discussed in detail here. The implementation of name servers is expected to be discussed in a separate report [Zhou84]. However, the user's view of updates has a direct bearing on the implementation of the resolver. The general scheme will be discussed here, and the additions to the name server protocol are summarized in Appendix A.

The scheme which was adopted makes visible the distinction between primary name servers and authoritative name servers. Updates must be made to the primary name server. Processes and users which require immediate access to the new information must also be able to distinguish primary name servers.

When an update is made, the new information will be propagated to secondary name servers in much the same way that is suggested in the proposal. The only substantial difference is that changes to the database will be made incrementally, rather than by replacing data for the entire zone.

There is no notion of a transaction which spans multiple messages. This restriction considerably reduces the complexity of the update algorithms. [Bern81] surveys techniques for guaranteeing the serializability of transactions which span multiple messages and multiple machines. All of these techniques are costly in both the complexity of their implementation and their running time.

For user queries, an unreliable communication medium is assumed. This has consequences on the semantics of the update operations. Specifically, update requests may be received more than once. Therefore, the operations must be meaningful when they are executed repeatedly. An update which is acknowledged has been successfully installed in the database at least once.

There are three update operations, namely, addition, deletion, and modification. It is not an error to add a record which is already in the database, nor is it an error to delete something which is not in the database. Similarly, it is not an error to repeatedly make the same modification. However, modification is the only update operation which may fail. It is an atomic deletion and addition, and it is an error for neither the old nor the new data to be in the database. Name servers return a distinct response code, if nothing in the database was changed by an update operation.*

The modify operation provides support for a transaction which logically consists of a read followed by a write. Unlike the techniques proposed in [Bern81], the burden of guaranteeing serializability is carried by the user. Modify can be used to ensure that the data which was read has not changed between messages: since it fails, if the old data is no longer present. At this point the user would reread the data and try again. This does not provably prevent process starvation, as there is no guarantee that the data could not be changed after every read, but in practice starvation is not expected to be a problem. In keeping with the idea that updates must not require the coordination of name servers, both the old and the new data must be on the same host. Thus, there is no possibility that a modify would require messages between name servers.

To simplify the handling of acknowledgements and response codes, all updates must apply to a single domain name, and a single class. This is because resource records with different domain names or classes may be in different zones of authority. Thus, it is possible for their primary servers to be distinct. It is therefore possible that no single response code would apply to all of the records in the update. This restriction is also used to guarantee that the old and new data for modify operations are on the same machine. In the case of additions, resource records are assumed to be added to the database one at a time. Updates are expected to be sufficiently infrequent that the performance penalty for allowing only record-at-a-time additions will be acceptable. For deletions, wild cards are allowed in the type and data fields without violating the assumption that all affected records are on the same machine. This is a user convenience which is easy to implement.

*The resolver conveys this information to users by setting the value of the global variable "errno".

2.2.2.2.2. Support for Locating Primary Servers

Within the specifications of the original proposal, a primary name server may be located by a resolver in a somewhat cumbersome way. The resolver first issues a query for the domain name until it receives an authoritative answer. The zone which the domain name belongs to can be inferred from which server gives the authoritative answer. That server can then be queried for the location of the primary server for the zone. The last step is to direct the original query to the primary name server.

The cumbersomeness of this approach makes it desirable for name servers to be able to distinguish queries which require a primary response. In the case of retrieval queries, this implies another bit flag in the packet header. Update queries may be distinguished by a distinct opcode. In order for name server records which are cached in non-authoritative name servers and resolvers to be useful in identifying primary name servers, it is also desirable to extend data on name servers to include information as to whether or not they are primary. Otherwise, caches will be useless in expediting the search for a primary name server. Authoritative name servers do not require such additional data, because the domain field of an authoritative name server record will always be the best possible match to a domain name in its zone of authority. Thus, queries will be directed to authoritative name servers, if they are present in a cache.

2.2.2.2.3. Support for Process to Process Level Addresses

One of the reasons for allowing user updates is to provide support for the naming of processes. At present, there is not sufficient support for process to process level protocols. For the time being, it is assumed that this will be corrected by adding more information to address type resource records. This works especially well for protocols which are implemented on top of the internet protocol. In this situation, process addresses are simply more specific instances of internet addresses, in that they contain a port number and a protocol number in addition to an internet address [rfc791]. Such records can be distinguished by their greater length by programs which are aware of the distinction. Programs which are unaware of the distinction should not be affected.

One of the most attractive reasons for adding user updates and better support for process level addressing is that it would then be possible to support process rendezvous. One process could bind itself to an unused port, and register with a name server, and another process would then be able to retrieve the address with a symbolic name for the first process. For the purpose of the present discussion, a port number can be thought of as the address of a process on a given host.

Name server support of process rendezvous would be a great improvement to the facilities that are available now for establishing connections between user processes [Leff83]. In order to communicate with a process it is necessary to know what port that process is using to recieve its messages. This can be done by agreeing on a port number ahead of time. However, there is a requirement that processes use port numbers which are unique on their host. So, it is possible for some unrelated process to prevent the rendezvous by using the agreed upon port. The alternative is for a process to use any port which is free at the time, register its port number with some agent, and then the port number could be retrieved from this agent. Unfortunately, there is no convienent mechanism for doing this at the present time. This situation is almost certainly an impediment to the development of distributed applications.

2.2.2.2.4. Disadvantages

In defense of the authors of the domain names proposal, we must say that introducing user updates and process rendezvous forces implementors to solve a number of difficult problems. These include preventing unauthorized users from altering data, purging transient data from the database when it is no longer needed, and making provisions to ensure that changes to the database are not lost when a host crashes. Probably, the most serious of these are security issues. It is beyond the scope of this report to make protocol specifications which address the need for security. Rather, this report assumes that these needs can be met within the framework of the current proposal.

In any case, most of the mechanisms for ensuring data security belong in the name servers and not the resolvers. This is because the data is maintained by the name servers, and the network is not a secure communication channel. [Denn82] provides a good introductory discussion of the distinction between authentication and access control. So, the role of resolvers in security issues will most likely be limited to including user identification and digital signature in queries. In the meantime, applications that do not require security, such as experiments with process rendezvous, can make use of the enhanced update facilities.

2.3. Efficiency and Reliability from a User Viewpoint

This section describes, in general terms, the kinds of efficiency issues that the user may wish to have some control over. These issues are primarily a three-way trade-off between the degree of certainty that the returned data is correct, the overhead of caching information, and the cost of transmitting additional messages over the network.

Section 2.2.2.1.4 mentioned that the domain names proposal allows users to specify that their queries only be answered by authoritative name servers. Section 2.2.2.2.2 suggested that users be allowed to further specify that their queries be answered only by primary name servers. These specifications allow the user some control over the reliability of the data, but increase overhead by forcing messages to be transmitted even if the data is present locally. However, it seems necessary to allow the user such control. For example, if inconsistent data is retrieved from a cache, it seems desirable to allow the user the opportunity to recover by insisting on retrieval from a reliable source.

The user may also wish to have some control over whether or not data is cached. For example, if data is cached on a per-process basis the user could either allow or disallow recent responses from being cached in the address space of his process. Such control over caching might be important for processes in which memory space is a critical resource, but, as before, the price paid is an increased number of messages.

A tangential issue is that system administrators may wish to tune the placement and replication of data to its pattern of use. Once again, replicating data increases the space resources consumed and increases the chances of its being out of date, but reduces message traffic. Administrators may not wish their choices to be over-ridden by users.

In general there is a wide spectrum of possibilities as to the extent that data is replicated, the extent to which users will be responsible for managing its replication, and the extent to which users will be aware of its replication. It is also not clear whether there is a single best solution for all possible applications of name service.

Although the implementation of the resolvers and name servers will necessarily determine some of the options which can be allowed the user, it is wise to embed as few real choices into the design as possible. When there is more experience with various applications of the resolver, it will be possible to define which parameters should be user settable, and which decisions can be made within the resolver. At the present time, it seems most important to provide mechanisms allowing the user to express concerns about efficiency and reliability. The particular options which are available are of secondary importance. If it is later found that a user should not have direct control over some function of the resolver, that option can be reimplemented to be taken as a hint. Obviously, the design should allow more options to be added later.

The mechanisms which are used to give the user control over performance issues can also be used for passing various ad-hoc arguments to the resolver. The rationale for doing this is two-fold. First, it would complicate the use of the resolver if the user were required to specify values for seldom used options with every call. Second, it gives the resolver a flavor of distinct modes of operation. Thus, it is possible to isolate the portions of a program which are strictly concerned with data retrieval from the portions of the program which set the context for that retrieval. This is intended to foster the modular design of application programs.

3. Design Goals

This section states the points of view which motivated the design of the resolver. It was most important to implement a scheme which could be used to replace the existing mechanisms for locating host address information and which could accommodate related applications. However, since it was an academic project, the goal was not simply to produce a software product.

3.1. Ease of Use

The implementation of the resolver involves many problems which are common to almost all utility programs. The most crucial problem is that it must be easy to use. This means that users should not have to assimilate a vast quantity of detailed information before they can make use of it. On the other hand, it would be inappropriate to restrict the functionality of the resolver by making a priori decisions on behalf of the user. So, the foremost concern is to provide a simple, yet powerful, model of name service.

3.2. Efficiency

Once the resolver is successful in meeting user's needs, and becomes a frequently used program, its use should not unduly degrade the performance of the system which supports it. Therefore, it should not make excessive demands on memory, cpu, io, or the internet networks. These efficiency issues are not part of the user's view of name service, and, as much as is possible, they should be treated independent of the user interface. However, there are policy issues which users should be able to influence, without resorting to recoding the resolver.

3.3. Experiment

Many problems with a design may not be detected until some of the design is implemented. So, one of the goals of the project is to detect any such difficulties. There are at least two areas where problems may arise. Some aspects of the protocol design could force a cumbersome and inefficient implementation. Similarly, the host operating system may not provide suitable mechanisms for an implementation. Thus, an implementation of the resolver is partly an evaluation of the protocol specifications. The ease with which the resolver is implemented is also a partial evaluation of the adequacy of the interprocess communication mechanisms provided by the operating system (in our case, Berkeley UNIX 4.2 BSD).

Given a particular protocol specification and a particular operating system, there is a wide range of possible implementations. So, one of the goals of the project is to identify, and where possible evaluate, some of these alternatives. Caching strategies are of particular interest in the implementation of a resolver. Section 4.3.2 discusses the alternatives. Our implementation included a per-process cache, and Section 5 provides some data about the effectiveness of this technique. Other techniques would have been more difficult to implement, and were not experimented with.

4. Implementation Issues

In this section, design decisions and their rationale are discussed. While the Section 2 discussed what services were to be provided, this section will discuss how they were provided.

4.1. Basic Structure

The most basic choice in the design of the resolver is its process structure. There are, at least, three possible process structures for the resolver, each with its advantages and disadvantages. First, each process that communicates with name servers could include code and data for a complete resolver in its address space. Second, the majority of the resolver's code and data could be included in the UNIX kernel. Third, the resolver could run as a separate process, and each process that communicates with name servers would include a small amount of code for communicating with the resolver process. In this case, the resolver could either be a child of the client process or a daemon that was started at boot time.

The choice of process structure is a trade-off between the ability to share code and data, communications costs, and the hazards of altering a successful operating system. It is desirable to share code and data to eliminate redundant copies, and thus save on memory space requirements. It is also desirable to share data so that the performance of one process might benefit from data cached by another.

If each client program links a copy of the resolver into its own address space, the cost of communicating with the resolver is the cost of a subroutine call, but it is not possible to share either code or data between processes. It is not possible to share the code because each text segment will be different, and it is not possible to share the data because Berkeley UNIX 4.2 BSD does not provide mechanisms for sharing data between processes. Even if it were to do so, it is not clear that there would be sufficient access control to prevent malfunctioning processes from fatally interfering with the operation of resolvers which were running in other processes.

If the resolver runs as an independent process, it is possible to share code, because all resolver processes will have the same text segment. Furthermore, if there is a single resolver process for each machine, its data will effectively be shared between all processes which use the resolver. The problems with this approach are that the cost of communicating with this resolver process will be substantial, and processes will be serialized by access to the resolver process.

A kernel implementation avoids the performance pitfalls of the two approaches above. It is possible to share code and data in a controlled manner, the cost of communicating with the resolver is the cost of trapping to change the mode of the processor, and the kernel already maintains multiple threads of control. However, altering the kernel impedes efforts to standardize Berkeley UNIX, and, once the resolver is part of the kernel, the resolver will be much more difficult to modify and debug.

The resolver need not run in privileged mode, but would benefit from the ability to share portions of its address space between processes. Shared memory, with enforced access control, would make a per machine cache feasible. Shared memory would also eliminate the need for redundant copies of resolver code. At present, the only efficient mechanism for sharing memory between processes is to access the shared memory when running in kernel mode. That is why I suggest that it may eventually be desirable to move portions of the resolver into the kernel. An operating system which supported user defined monitors would be ideal. However, no realistic Berkeley UNIX 4.2 BSD process structure for the resolver seems entirely satisfactory.

The present implementation of the resolver links a complete resolver into the address space of each client program. This approach was chosen because it is the simplest to implement, can probably be adapted to either of the other alternatives, and it is not known if the ability to share code and data is essential for resolver performance.

4.2. User Interface to the Resolver

The resolver is implemented as a set of C language subroutine calls. Here, those calls are described in considerable detail. This level of detail will provide a tangible basis for what would otherwise be a vague philosophical argument about the value of a layered implementation, and the value of providing a relatively uninterpreted view of the underlying abstractions.

As in the design of a great many other interfaces, there is the question of what should be a primitive operation, and what operations should users be responsible for. The potentially large number of types and classes makes it impractical to give explicit support to each and every format (each of these fields occupies 16 bits). It is also not possible to predict the requirements of all applications. For that reason, the resolver presents a relatively uninterpreted view of the returned data.

There is one important exception to this rule. It seems desirable to integrate name service with interprocess communication facilities. Berkeley UNIX 4.2 BSD implements all communications objects using file descriptors [Leff83]. The system calls which create file descriptors take structures which contain protocol specific data as arguments. Therefore, programmers are required to know a considerable amount of detail about the underlying protocol to establish connections between processes. The inclusion of functions in the resolver to map domain names to the structures which are used to create file descriptors could greatly increase the transparency of interprocess communication.

At present, Berkeley UNIX supports the internet protocols, UDP and TCP. It also supports communications objects which are based on the UNIX file system. Assuming that resource records for internet addresses are extended to include protocol and port information, as is suggested in section 2.2.2.3, the resolver maps domain names for UDP and TCP communications objects to the structures which are used to create file descriptors. If UNIX class data were included in the name server database, a similar mapping could be performed for UNIX path names.

The justification for directly supporting domain name to machine address mapping is that programs which use name service are very likely to wish to communicate with the objects that they request information on. Also, the resolver itself uses address records to locate name servers. Thus, it is possible to enhance the interprocess communication facilities which are available at little additional cost to the majority of processes which use name service. If it is later found to be desirable to directly support other applications of name service, the "hooks" in the code, which were developed to support address mapping, can be used to support these other applications.

As was discussed in section 2.2.2 the resolver provides two orthogonal types of service. The first category of services consists of mechanisms for invoking name service. The second category of services consists of mechanisms for setting the policy of the resolver. These topics will be discussed separately.

4.2.1. Invoking Name Service

The interface to the resolver is a subroutine library for C language programs. What follows is a description of each subroutine which is intended to be called from programs that use name services. The programs which use the resolver are expected to include system processes, such as mailer daemons, experimental programs which execute on multiple hosts, and interactive programs, such as "finger", which returns name and address information about the owner of an account.

The calls which are provided fall into four general categories. Three of these categories are the general query types which are described in section 2.2.2.1. The fourth category is for changing the contents of the name server database, as described in section 2.2.2.2.1. These are general purpose routines upon which specific applications of the name service can be built.

The notation which is used to describe the subroutine calls is borrowed from the UNIX programmer's manual. The include lines at the start of each description indicate files that contain needed structure declarations, and the routines are written as they are declared, so that the types of their arguments and return values is clearly indicated.

When it is appropriate, the name of a formal parameter is chosen to indicate correspondence to a field in a resource record. Thus, parameters labeled "type" correspond to types as they were defined in section 2.2.2.1.1. Similarly, parameters labeled "class" correspond to classes as they were defined in section 2.2.2.1.1. Symbolic constants for all of the currently defined types and classes are among the things which are contained in the file "resolv.h". So, it is expected that type and class arguments will usually be a symbolic constant from this file. Parameters labeled "dname" are ASCII representations of domain names with labels separated by dots. It is expected that all such strings will be terminated with the character null.

Parameters which do not have a correspondence to a field in a resource record will be discussed as the routines are presented.

4.2.1.1. User calls for generating and interpreting standard queries

```
#include <sys/types.h>
#include <netinet/in.h>
#include "resolv.h"

struct sockaddr_in *dn_in_addr( dname )
      char dname[];

char *std_query( dname, type, class )
      char dname[];
      int type;
      int class;

unsigned char *answer_i( i, pt, user_rr )
      int i;
      char *pt;
      USER_R_REC *user_rr;
```

The pointers that "dn_in_addr" and "std_query" return refer to structures which have been allocated via a call to the standard user level, memory allocation routine, "malloc". The user is free to release the space when the result of the query is no longer needed. This style of memory management keeps the user's data structures disjoint from the resolver's data structures. It also prevents new results from over-writting old, but still useful results. It is intended that the user will be unaware of the internal details of the structures returned. In the case of "std_query", this convention is strictly enforced.

Conceptually, "std_query" returns an indexed set of responses to the query specified by its arguments. This array can be indexed via calls to "answer_i". "answer_i" returns a pointer to the data field of a resource record. Out of bounds references are indicated by a zero return value. Here, "pt" is the pointer which has been returned by "std_query". "i" is the i'th resource record which was returned. The array of responses is indexed in the same manner as arrays in C: the first value of the index is 0. "user_rr" is a structure which contains other information about the resource record, including the length of the data, the domain name, type, and class. The interpretation of the data is entirely the responsibility of the user. Thus, calls to "std_query" require intimate knowledge of the internal structure of the data fields of resource records.

The same model is used with the other general purpose query routines. Specifically, general purpose query routines return a set of responses. The routine "answer_i" may be used to refer to specific members of this set. Also, the user is free to release the structure when the data is no longer needed.

"dn_in_addr" is one of the routines which provides direct support for domain name to address mapping. It is intended to be easier to use than "std_query". The structure which is returned can be used as an argument to any system call which uses the "sockaddr_in" structure. "sockaddr_in" is the structure which is used by user level calls to represent internet addresses. So, it is potentially a quite useful artifact. The user should have to know neither the internal structure of any resource records, nor the internal details of the "sockaddr_in" structure. However, if the domain name is simply that of a host, it may be necessary to supply a port number [Leff83]. Since, the intent is probably to communicate with a process on the host rather than the host itself.

4.2.1.2. User Calls For Generating and Interpreting Inverse Queries

```
#include <sys/types.h>
#include <netinet/in.h>
#include "resolv.h"

char *in_to_dname( addr )
     struct sockaddr_in *addr;

char *inv_query( data, length, type, class )
     char data[];
     int length, type, class;
```

Both "in_to_dname" and "inv_query" return structure that can be traversed using "answer_i", as is described in the section above. "in_to_dname" is the inverse function which corresponds to "dn_in_addr". "inv_query" is for more general inverse queries. Here, "data" is the data portion of a resource record, "length" is the length in bytes of "data", "type" is the type of the resource record, and "class" is its class. Since this is a non standard type of query, it will be necessary to do a little more work to get at the interesting portions of the resource records which are returned. Namely, the domain names will have to be explicitly retrieved from the "user_rr" structure that "answer_i" returns. Both routines return a zero pointer if they were unable to perform the inversion.

Inverse queries will be successful to the extent that they are supported within the name servers which are queried. There is nothing to prevent users from asking unanswerable questions. The user may set the domain that the data is assumed to belong to. When this is done, the resolver will locate a name server which is authoritative for the specified domain, then direct the inverse query to the authoritative name server.* The means of setting the domain to direct inverse queries to, are discussed below in section 4.2.2.

4.2.1.3. User Calls For Generating and Interpreting Completion Queries

```
#include <sys/types.h>
#include <netinet/in.h>
#include "resolv.h"

struct sockaddr_in *dnc_in_addr(dname)
     char dname[];

char *com_query( dname, type, class )
     char dname[];
     int type, class;
```

"dnc_in_addr" and "com_query" are analogous to "dn_in_addr" and "std_query", respectively. The important difference is that these calls require only a partially specified domain name, as was described in Section 2.2.2.1.2, during the general discussion of completion queries. As with inverse queries, the user may set the domain which will be queried, and there is no guarantee that the query will be answered.

*If the specified domain matches the root of the zone of authority for one of the name servers in the resolver's cache, finding an authoritative name server for the domain will not require any queries.

4.2.1.4. User Calls For Updating the Name Server Data Base

```
#include <sys/types.h>
#include <netinet/in.h>
#include "resolv.h"

add_rr( dname, type, class, data, length )
    char dname[];
    int type, class;
    char data[];
    int length;

delete_rr( dname, type, class, data, length )
    char dname[];
    int type, class;
    char data[];
    int length;

modify_rr( ndname, ntype, nclass, ndata, nlength, old_rr )
    char ndname[];
    int ntype, nclass;
    char ndata[];
    int nlength;
    USER_R_REC *old_rr;
```

These are the routines for performing user updates. "add-rr" makes certain that the resource record specified by its argument is present in the database of the primary server for that resource record. "delete_rr" makes certain that the resource record specified by its arguments is not in the database of the primary name server for the record. "modify_rr" atomically replaces the type and data fields of "old_rr" with "ntype" and "ndata" respectively. They do not return data beyond an indication of their success or failure. The implementation of these functions within name servers is discussed in [Zhou84]. Once again, the user is presented a relatively uninterpreted view of the services which are available.

4.2.2. Setting the Resolver Policy

Section 2.2.3 of this report mentioned that there should be mechanisms for users to set resolver policy. This section discusses the mechanisms which are currently available.

There are a large number of options regarding the behavior of the resolver. Our design philosophy is to hide much of this detail from casual users by providing reasonable defaults. More knowledgeable users may alter the behavior of the resolver by resetting these defaults. This can be done either by making subroutine calls in the process which is using the resolver, or by adding directives and data to a resolver configuration file. The latter can be used to change the defaults for the casual user without recompilation.

4.2.2.1. User Calls for Altering the Resolver's Behavior

```
set_resopt( set, clear )
      char set[];
      char clear[];

set_domain( type, domain )
      char type;
      char domain[];
```

"set_resopt" sets and/or clears a number of yes/no resolver options. "set" is the set of options to be set. "clear" is the set of options to be cleared. Options which are both set and cleared will be cleared. Both "set" and "clear" are null terminated strings in which each character corresponds to an option.

The justification for this arrangement is that it is simple and easy to extend. Many of the potential options in the resolver can be stated in a yes/no fashion. For example, either any answer is acceptable, or an authoritative answer is required, or, perhaps, the resolver should only check its cache without sending any messages. The table below lists the options which are available now.

The options fall into three categories. First, it is possible to stipulate the reliability of the name server that data is retrieved from. The relevant distinctions are discussed in Sections 2.2.2.1.4 and 2.2.2.1.5. Second, there are some ad-hoc arguments. There are actually two types of completion queries as was mentioned in Section 2.2.2.1.2. The distinction is made here to avoid offering the user two identical sets of functions with slightly different semantics. Also, when internet address type resource records are extended to include port and protocol information, it may be necessary to specify the protocol that is of interest. Third, users are allowed to control some performance parameters, such as whether data can be cached in the address space of their program, how the resolver initializes itself, and what protocol the resolver uses to communicate with name servers. Some of the background for these options is to be discussed in Sections 4.3 and 4.4.

The ASCII character which sets and clears an option is listed below the description of the option in the table. Options which are the result of not making a specification with regards to related options are followed with the word "default".

Table of Current Resolver Options		
DATA RELIABILITY	AD-HOC	PROCESS ORGANIZATION
authoritative answer only ('a')	allow multiple matches to completion queries ('m')	
primary answer only ('p')	accept only UDP addresses ('u')	disable per-process cache ('c')
local cache only ('l')	accept only TCP addresses ('t')	use virtual circuit communication ('v')
any answer (default)	accept only host addresses (default)	don't use configuration file ('b')

Table 1

When the resolver is first linked into user code, all options are cleared. This provides a first level of default, which can be over-ridden by the configuration file. Some options may imply other options. "set_resopt" does not attempt to draw such conclusions. Our intention was to make this a low level interface to the resolver's state. In fact, these options are simply mapped to flags which are tested at various times during the resolver's execution, and the result of specifying several options which apply to the same topic is unpredictable. Other routines may be built on top of "set_resopt" to present a more coherent interface to resolver policy for specific applications of name service.

"set_domain" sets the domain which is to be queried for inverse and completion queries. The name servers which are the closest match to "domain" will be queried. "domain" is a null terminated string which is an ascii representation of a domain name. "type" is one of 'c' for completion query domain, or 'i' for inverse query domain. "set_domain" is a separate routine to keep the code for "set_resopt" as small and simple as possible. In a version of the resolver which were not linked into user code, a version of "set_resopt" which parsed its input in some non-trivial way might be more attractive.

4.2.2.2. The Resolver Configuration File

The first time that the resolver is invoked by a process, and perhaps at other times during that process' lifetime, the resolver consults a configuration file. The resolver must know both the address of at least one name server, and the address of its host. The configuration file is the mechanism for obtaining this knowledge, and other initialization data.

The configuration file uses a somewhat modified and much restricted form of the master file format described in rfc883. The restrictions are due to the fact that it is undesirable to carry the code and data for parsing the complete master file format in the resolver. The format of the master file is modified to allow directives which are specific to the resolver, and it is divided into sections so that the intended use of each resource record, within the resolver, can be determined from the record's position in the file. The reason of its resemblance to master file format is consistency: both so that users don't have to learn some entirely different format, and so that programs which handle master file format can be easily adapted to automate the handling of resolver configuration files.

Normally the configuration data is obtained from a global file. However, the user can override the global file by supplying his/her own file. If no configuration file is available, or if the user sets an option to disable it, the resolver attempts to locate a name server on its host before giving up. If the resolver is unable to obtain initialization data, the query which invoked the resolver fails and the global variable "errno" is set to contain an indication of the reason for the failure.

Locating a local name server depends on the existence of some independent mechanism for obtaining the host's address. Currently, the resolver simply queries the host table, because there is no other reliable independent mechanism for retrieving this data. So, conversion to the domain names system will have to include the implementation of such a mechanism.*

Two separate mechanisms for configuring the resolver are provided for the sake of versatility. Although caching is discussed in more detail in Section 4.3.2, the following example illustrates a case where the configuration file would not be useful. It is possible to implement a per-machine cache by running a name server process on each machine. A resolver would query the local name server. The local name server would either resolve the query on behalf of the user or return data that was already present in its database. After the local name server resolved the query, any data that was retrieved from remote name servers would be entered into the local name server's cache. In this case, the function of the configuration file would be assumed by the local name server. If this organization of resolvers and name servers is adopted, a much simpler implementation of the resolver would be needed, but it is now possible to experiment with this organization without recoding the resolver.

The configuration file is divided into four sections. The first section contains resource records which pertain to the resolver's host. This solves the technical problem of finding an address for the resolver to use in recieving responses. The second section contains resource records which pertain to the name servers which are to be consulted when resolving standard queries. This is the solution to the problem of finding an initial server to query. The third section contains name servers to direct completion queries to. The fourth section contains name servers to direct inverse queries to. The last two sections are optional, and in both sections the first name server record encountered is taken to be the domain that queries of that type should be directed to, unless the domain is reset by the user. These sections, therefore, are a partial solution to the problem of locating an appropriate domain for non-standard queries.

Specially marked lines in the file may be used as arguments to "set_resopt". This provides an opportunity to set defaults which are not coded into the resolver.

The resolver is unforgiving of errors in its configuration file. When it gets confused, it simply gives up. It is hoped that the primitiveness of the parsing of this file can be lived with, because the file is expected to be small, used by system administrators and other knowledgeable people, and infrequently changed. In short, the configuration file is a crude, but effective means of initializing the resolver.

4.2.3. Summary of User Interface

It should be apparent from the above discussion that the resolver insulates users from the intricacies of communicating with name servers, but the operations which it supports are determined by the operations which are supported within name servers. What it provides are essentially the abstract services which are described in the domain names proposal and in our extensions to the proposal (see Section 2.2.2.2). Features that are not specified in the proposal are left user settable. These include performance policies, the choice of the initial name server, and the choice of name servers for non-standard queries. Thus, it is intended to be a general purpose tool for C programmers. The subroutines which are currently available are summarized in the table below.

*The function which is currently intended to supply a unique identifier for the host, "gethostid" does not always return an internet address.

Policy Routines			
set_resopt()		set_domain()	
Query Routines			
Standard Queries	Completion Queries	Inverse Queries	Update Queries

std_query()
dn_in_addr()
	com_query()
	dnc_in_addr()
		inv_query()
		in_to_dname()
			add_rr()
			delete_rr()
			modify_rr()

Table 2

4.2.4. Satisfying the Immediate Purpose of Domain Names

The immediate purpose of the domain names proposal was to replace a global host table. In order to do that, it is necessary to replace the existing routines for retrieving address information. New programs may use the calls described above, but there are a number of programs which depend on the the old calls. So, the question of how to convert to the new system is an important part of the design of the user interface. This section describes a strategy for rewriting the old routines, that will disrupt existing programs as little as is possible.

In our environment, there are five routines which are currently used to query the host table. Three of these have semantics which are meaningless in the domain names system. Specifically, "sethostent" opens the host table for reading, "gethostent" gets the next entry in the table, and "endhostent" closes the table. The idea behind allowing users to sequentially search the host table was that, if the user had special needs, such as finding the address of a host on a particular network, he could locate the appropriate records by hand. It is difficult to believe that this will be a useful procedure once the database is distributed. Unfortunately, there are no parameters to the calls which would aid in determining what the user is looking for. Therefore, programs which use these calls must eventually be recoded.

Of the other two routines for querying the host table, "gethostbyname" takes a name and returns an address. This can be simulated with a completion query. "gethostbyaddr" takes an address and returns the name of a host. This can be simulated with an inverse query. However, a number of existing programs will probably fail when this is done. The reason is as follows. The order of entries in the host table is significant in that the first match is returned. Any program which used this fact to avoid searching by hand will have difficulties, because multiple entries for the same name may not be returned in the order in which they were stored in the file.

Another difficulty is that the structure these routines return contain a list of aliases for the record. These aliases are not the same as the aliases used by the domain names system. Rather, they are used for the same thing that completion queries are used for: namely, the user may type in an abbreviated name for a host, but there is no requirement that the alias be composed of the initial letters of the full name. Simulating the list of aliases exactly will require careful structuring of the database, and multiple queries, which is probably unacceptably costly. Therefore, conversion to domain names will probably lead to the discovery of programs which depend on particular aliases.

Although the above discussion was in terms of our local environment, the moral should be true for all environments: that is, converting to the domain names system will be a painful process. It should be done in gradual steps. I suggest first developing some new applications to test the resolver in production use. Then, routines which can be approximately simulated with the resolver should be replaced. The last, and most painful, step will be to purge routines which cannot be simulated from the system. At that point, the host table will no longer be necessary.

4.3. The Resolver's Interface to Name Servers

It is now time to discuss the implementation of the other important interface in the resolver. The resolver can be viewed as a layer in the implementation of name services. From this viewpoint, its purpose is to translate high level user requests into standard data formats, and handle the mechanics of getting the data to an appropriate name server. One important aspect of the design is the choice of appropriate mechanisms for the actual transmission of data. Another important aspect is choosing data structures and algorithms for handling the data once it is returned. Every effort has been made to treat the two issues separately.

4.3.1. Implementation of the Protocol

Fortunately, Berkeley UNIX 4.2 BSD provides mechanisms for transmitting data which are adequate for the resolver's purposes [Leff83]. This portion of the design is a matter of formatting a block of data, then making a sequence of appropriate system calls. For the benefit of those readers which are familiar with the environment, the names of the routines used will occasionally be mentioned. However, an attempt will be made to discuss the functions used without relying on the reader's knowledge of the local environment.

The protocol specification is designed to be used with either datagram or virtual circuit communication. For reasons of efficiency, datagrams are the preferred mode of communication. However, this requires that the resolver be capable of handling retransmission when no response is received after some interval, and consequently it must also handle duplicate responses. Name servers are not responsible for insuring that there responses arrive at the proper destination.

The typical processing of a query is as follows. First, the resolver will format the user query. As there is a maximum size for datagram queries, the space for this can be pre-allocated on the user stack. Then, the resolver will create a UDP communications object using the routine "socket", and "bind" the object to a random port. This communications object will be used for the duration of the processing of the query, but "closed" before returning to the user's program.

The user's query may have to be transmitted to several name servers, before it is either answered or determined to be unanswerable. The inner loop of the query processing is as follows. First, obtain the address of the known name server which is closest to the domain of the query. Then, transmit the query using "sendto", and wait for a response for a fixed interval using "select". The transmission is tried some fixed number of times. Each time the query is transmitted, its identification number is incremented. If a response with an appropriate identification number is received before one of the calls to "select" times out, the response will be entered into the resolver's data structures for further processing. Otherwise, the resolver will repeat the procedure using the address of the next best name server. If there is no next best name server, the query will fail, returning an appropriate error value to the user.

When resolving a particular query, it is important to avoid sending the query to a server that has already responded to it. This is handled within the resolver's internal data structures by flagging servers which have been used during the processing of the current query. Thus, infinite loops are avoided.

If all goes well, the address and domain of a name server which can answer the query one way or another will eventually be entered into the resolver's data structures. The resolver will transmit the query to this server, receive an answer and return the result to the user.

There are some exceptional conditions which should be discussed at this point. These are conditions which are flagged in the header of a response, and can be handled within the routine which transmits the query. Name servers return a code which indicates errors detected by them. Some of these codes indicate that it would be futile to continue processing the query, in which case an error code is returned to the user. Other codes indicate that the current server is inappropriate for the query, in which case the next best server is tried. There is one condition which requires special action. Specifically, if the response is too large to fit in a datagram, the protocol requires that a virtual circuit be opened, and the query must be retransmitted to obtain the entire response.

It is also possible to send queries which are larger than the packet size. This is not supported at present, but implementing it would be a simple matter of allocating enough space to format the query, and setting a flag to force virtual circuit transmission. It is not implemented now because the packet size is large enough (512 bytes) to hold all currently defined queries for existing types and classes, and we do not wish to encourage virtual circuit transmissions. So, the implementation of this feature has not been a pressing concern.

The essential point in the above discussion is that the responsibility for obtaining a complete and useful response is isolated within a single body of code. The remainder of the resolver is designed assuming that such a response has been obtained, and is responsible for its interpretation and storage. It is, in fact, possible to use virtual circuits exclusively, and it would be possible to switch to some entirely different mechanism for transmitting data, without recoding more than a single module. The stage is now set for the discussion of what the resolver does with data once it has been obtained.

4.3.2. Division of Labor Between Resolvers and Name Servers

Although much of the division of labor between name servers and resolvers is implicit in the protocol specifications, there is considerable latitude in a couple of important areas. First, it is possible to build some heuristics into both the resolvers and the name servers. Second, it is possible for both resolvers and name servers to cache data. The choice of heuristics and caching strategy are primarily of interest for performance reasons.

One type of heuristic which may be of some use is to anticipate what the next query will be, and to fetch the answer before it is requested. For example, a name server could return related internet address records when a user mailbox name is requested. Or, a resolver might detect domain name pointers, and locate the records which they point to. I believe that this type of heuristic is more appropriate in name servers than in resolvers. Since the number of datagrams sent should be minimized, as long as there is room in the packet, it costs nothing for a name server to include helpful extra data. On the other hand, extra messages are needed by the resolver to obtain data which is anticipated, but not yet requested.

A related heuristic is for resolvers to selectively retain certain types of resource records, and very recent responses. If name servers include in their responses data which is anticipated to be the object of the next query, it is prudent to retain the response until the next query. However, the relatively small size of a datagram response and the expense of opening a virtual circuit limit the usefulness of this strategy. Also, queries which require authoritative answers force messages to be transmitted in any case.

One kind of information which can profitably be retained by resolvers is information about name servers. The number of referrals to name servers can be reduced by directing the query to the most appropriate server first. This will be most useful for processes requesting information that is distant from their local domain. It should be noted that the number of referrals to name servers can be reduced even when processes require authoritative responses.

The subject of resolvers retaining data from previous responses leads to the second division of labor between resolvers and name servers: namely, caching. The most compelling reason for including some sort of caching scheme in the resolver is that messages are required to retrieve cache data from a name server. This is particularly costly if the name server does not run on the same host as the resolver.

The current implementation of the resolver caches data on a per-process basis using the strategies which were outlined above. Specifically, responses are retained for the next few queries and information on name servers is retained for the lifespan of the process the resolver is running in. The user may override this behavior and instruct the resolver to forget everything. Periodic flushing of the resolver's cache may be useful for processes with an extremely long lifetime, and processes which make numerous queries to diverse domains. No caching may be desirable for processes in which space is at a premium, or which only make local queries.

The domain names proposal suggests a scheme where the resolvers transfer information that they have accumulated to the local name server's cache. It is only worthwhile to make such a transfer if the data did not originate from the local name server. In our environment it would cost an extra message to make the transfer, and additional changes to the protocol would be required, as there is currently no mechanism for sending data to be cached to name servers as queries. Therefore, this is not a particularly attractive scheme.

Another scheme would be for the local name server to resolve the query and thus obtain data for its cache. This would tie up the name server when the query was not local. It would also make it difficult to choose an appropriate timeout for datagram queries, as the query could take an arbitrary amount of time to resolve. This problem could be solved at the expense of efficiency by using virtual circuits to communicate with the local name server. A variation on this scheme would be to install a resolver process on each machine, and user processes would communicate with the resolver process via a UNIX pipe. The cache would then be in the local resolver process.

A third alternative is as follows. Local name servers are in a position to identify the domains on which information is frequently requested. This is true, because the local name server will often be the source of the first referral when a resolver requests information from a distant domain. Also, if the local name server maintains a cache, it is reasonable for resolvers to always query the local name server before other name servers are tried. Thus, it seems reasonable to have name servers take the initiative in caching information for zones which they are not authoritative for. This can be handled either by explicit zone transfer, or by having the name server periodically invoke a resolver to query for individual domain names. For the same reasons that it is desirable for resolvers to pay special attention to name servers, name servers should make a particular effort to obtain information on other name servers.

Another possibility is that resolvers which run on the same machine could pool the information that they accumulate about the name space. If the resolver is moved into the kernel, this may be an attractive option, but so long as the resolver maintains its data structures in each user's data space, the cost of transferring data between processes would probably negate the benefits of this strategy.

The best choice of name server and resolver configurations is probably site and application dependent, and is worthy of a careful study which is beyond the scope of this report. The best performance can probably be obtained by implementing the resolver as part of the kernel, and to cache data on a per-machine basis. However, it is not clear that the increased performance would be sufficient to justify the loss of maintainability and portability. The present technique of caching data on a per-process basis is useful only for applications which make predictable sequences of requests, or request data from distant domains. All of the other caching techniques require significant processing to make an entry in a cache, or add a significant overhead to the processing of each query. Therefore, it may not always be useful to cache data.

4.4. Resolver Data Structures and Internal Algorithms

Since the resolver is responsible for storing the results of queries, memory management within the resolver is a substantial part of the resolver's function. In some sense, the memory management portion of the resolver is the glue which holds the user interface and communications interface together. This is especially true if the term memory management is taken broadly enough to include the creation and maintenance of the resolver's internal data structures, as well as the allocation of space and garbage collection. Regardless of terminology, this section discusses the portions of the resolver which could be replaced without alterations to either the user interface or the name server protocol.

A potential criticism of the current design is that the internal structures are too complicated, and the increased performance that they provide is not sufficient to justify their existence. Indeed, they comprise the vast bulk of the code in the resolver, and there are ad-hoc algorithms which would suffice for resolving queries. These ad-hoc algorithms would require much less space.

In defense of the current design, we can state that there are benefits beyond the ability to cache data. For example, when our group decided to add updates to the protocol specification, it was a simple matter to add another algorithm for locating appropriate name servers. Similarly, it was a simple matter to allow users to specify the domain that they wish inverse and completion queries to be directed to. Ultimately, the utility of the present design depends on how precious space is in the resolver's environment. Also, there is no existence proof for the claim that simpler algorithms would both suffice and use less space. Ad-hoc algorithms have a way of growing out-of-bounds when they attempt to cope with special cases.

The remainder of this section will discuss the specifics of the resolver internals. It will start with how space is managed, and proceed to describe what is done with the space.

When the resolver is first entered, it allocates a relatively large block of space for the stôrage of responses and internal data. This is accomplished with a call to a user level storage allocation routine, "malloc". While the resolver is running, it allocates space for its structures by incrementing a pointer to the next available byte of this large block. When there is not sufficient space available on the block, another block is allocated. This procedure is repeated as often as is necessary. When the resolver finishes the processing of the query, it compares the number of blocks which have been allocated against an upper limit. If the limit is exceeded, a garbage collection routine is run.

The discussion which follows requires the reader to have some knowledge of how queries and responses are formatted. Packets have a header section which contains an operation code, a response code, several flags, and the number of resource records in each of the other sections of the packet. The other sections are the question section, answer section, name server section, and additional records section. The name of a section describes the purpose of resource records in it. The question section is somewhat anomalous as it does not contain complete resource records, rather it contains just the domain name, type and class that the query pertains to. Standard queries contain a single entry in the question section, and all other sections are empty. The response to a standard query contains the question section of the query that it is a response to. The general format of a packet is illustrated in Table 3.

Format of Communication Packets Used by Resolvers and Name Servers

Section	**Fields**				
header	codes	question count	answer count	name server count	additional count
question	domain name			type	class
	more question records				
answer	domain name	type	class	ttl	data
	more answer records				
name server	domain name	type	class	ttl	data
	more name server records				
additional	domain name	type	class	ttl	data
	more additional records				

Table 3

The mechanics of the garbage collection routine depend on the fact that queries and responses are stored in the same format in which they are transmitted. In other words, all data is stored in the format which is illustrated above. So, garbage collection is a matter of allocating fresh space, copying the resource records which are to be saved to a single packet structure on the new space, freeing the old space, and rebuilding the resolver's data structures. This is effective because there tends to be a lot of dead space within packets. After initial processing, the header and question sections are unnecessary, and resource records pertaining to the same domain name can be grouped together to take advantage of the compressed domain name format. Fortunately, the routine which builds the resolver's data structures is relatively fast. All structures are built in a single pass over the packet.

There is another good reason for storing resource records in datagram format. Not translating them to some internal format avoids a memory to memory copy of incoming data. In practice, it is usually not necessary to process every byte in a packet. When entering resource records into the resolver's data structures, data fields are skipped without being inspected, the hash function is computed on the basis of the leftmost label only, and when resolving hash table collisions domain names can usually be determined to be unequal by inspecting the first few bytes of the name. This is probably a small savings in comparison to the total overhead of resolving a query, but there seems to be no substantive reason not to take advantage of it.

The resolver's internal data structures are built up by assigning pointers to interesting parts of the packet. The basic resolver data structure is a hash table, in which each bucket is a structure containing pointers to the domain name and to the data fields of a resource record, together with some auxiliary information. Other data structures are built on top of the buckets of the hash table.

This hash table is the basic structure since it is often desirable to look for records pertaining to a particular domain name. Examples include locating the answer to a query, finding an address record which corresponds to a name server record, and grouping resource records for the same domain name together during garbage collection.

The other major data structure which the resolver builds is a domain tree. The domain tree makes it possible to locate the most appropriate domain to direct a query to. Each time a name server for a new domain is located, it is added to the domain tree. If the new domain is the parent of at least one node in the tree, the tree must be restructured. Otherwise, the new node becomes a leaf in the tree.*

The space which is used to format user queries is allocated on the stack, and is re-used with every query. Therefore, it is important that the resource records in the user query do not become part of any structures which are used beyond the processing of the current query. The user query does participate in some structures, but these are discarded upon exiting the resolver. The space that is used in building these structures is reclaimed when garbage collection is run.

There are other structures which are used only for the processing of the current query. The most important of these is a list of servers which have already been tried. The list is not used to check for repeats, but rather to clear the flags of the servers before the next query is processed. The space for this list is also simply discarded when it has served its purpose.

There is one other potential source of dangling references. Specifically, it is occasionally necessary to delete resource records from the resolver's cache. This happens when the cache is updated to reflect the action of a successful user update, and when a name server does not respond to queries. There is no convenient way of locating references to the deleted record. So, the record is flagged as deleted, but it is not removed from any data structures until garbage collection is run. At that time, it is simply not re-entered into the cache. Deleted records are ignored when they are retrieved from the cache.

* The relationship between two domain names can be efficiently determined by recording the number of labels in the each domain name, skipping the extra labels in the longest name, comparing the remaining labels in the names pairwise, incrementing a counter if they are equal, and setting the counter to zero if they are not. The value of this counter when the end of both names is reached is the measure of closeness between the two names. If this counter is equal to the number of labels in the shorter name, that node is the parent. This rather circuitous algorithm is necessary because the root is the rightmost label in a domain name, but the names must be scanned from left to right, because of their format.

Much of the time spent in garbage collection is spent in copying resource records. Some copying of resource records is avoided by maintaining the resource records which come from the resolver's configuration file separately. These records are not recopied, but are simply reentered into the data structures.

There are several data structures which have not been mentioned, and are built to perform various bookkeeping chores, such as a list of blocks that have been allocated to the resolver. Also, there are a number of statically allocated variables which retain the resolver's state between calls. It should be clear that this style of memory management avoids much of the overhead of a general solution to memory management problems. This is only possible because all of the information which is used to build the data structures is contained in the data. The data structures themselves are not used to represent information, but merely allow faster retrievals of it.

This concludes the discussion of the normal handling of the resolver's major data structures. What remains are some special cases which are worthy of brief mention.

It is possible that the operating system, when requested, will not grant the resolver more space. This would be either a result of exhausting virtual memory, or, what is more likely, of the lack of disc space for swapping the process out. There is no guarantee that the resolver's data structures will be in a consistent state when this happens, and hence it is necessary to take special recovery action. When this occurs, the resolver immediately flushes all of its data, and returns an error to the user. The mechanism for doing this is a pair of assembly language routines, one of which saves all of the registers on entry to the resolver. The other pops the stack back down to the site of the call to the first routine. The error return can be distinguished by a different return value. The justification for this extremely machine dependent solution is that otherwise a great deal of code space and processing time would be spent with every call in an attempt to gracefully back out of a very rare disaster, and there is no provision in the C language for a goto which crosses routine boundaries.

The point is that all calls to the resolver pass through a single routine. That routine saves enough of the machine's state that it is possible to recover gracefully from a disaster. When a disaster happens the process stack is popped down to the stack frame of this routine, "mk_query", and the resolver's data structures and open file descriptors are cleaned up.

Another, less serious disaster occurs if the packet size of the response exceeds the size of the blocks that the resolver normally allocates. This is only possible with responses received on a virtual circuit. The solution is simply to allocate a block which is big enough to accommodate the response. This is possible, since the first two bytes of a virtual circuit response indicate the length of the response, but it may cause the situation which was just described. If sufficient space can be obtained, the garbage collection algorithm will be invoked when the current query has been processed, and things will settle back to normal operation.

One problem with the style of operation which was outlined above is that it is difficult to associate with resource records information which is not part of the resource record. This is especially true if the information is intended to survive garbage collection. At present there is no such information. However, it may be desirable to add such data in the future. For example, it may be desirable to associate timeout intervals with particular name servers. The solution is probably to extend the data fields and embed the information within the resource record itself. This would require a small amount of special handling within the garbage collection routine, but would allow other routines to remain in their present form.

4.5. Summary of Implementation

The resolver is a layer in name service. It is implemented as code to be linked into a user's program. The resolver is itself divided into three layers, a set of user level routines, routines which handle the actual transmission of data, and routines to manage data accumulated from responses. The user level routines provide the abstractions which are supported within the name servers, and some control over the internal operation of the resolver. Since these routines insulate users from the details of the resolver's implementation, it is possible to support applications of name service while the design of the resolver evolves.

The current implementation links a fully functional resolver into a user's address space. This includes the ability to cache responses from previous queries. This is only one of the many possible designs.

5. Evaluation

The previous sections have discussed various aspects of the design of the resolver. The resolver's purpose, the user interface, the protocol, and its internal operation, have all been discussed. The rationale for the parts of the design that are not specified by the protocol has been given. This section attempts to answer the question: "how good is it?".

In many ways this is the most important section of this report. Without data to back it up, personal judgement is mere opinion, and not engineering. Unfortunately, very little data is available at the present time on the performance of the resolver. There are no other resolver designs with which to compare benchmarks. Nor have we attempted to use analytic or simulation techniques to estimate the complexity of the resolver's task. The real evaluation will have to wait for the resolver to come into production use. In time, it will be possible to discover mistakes and make improvements.

Similarly, the real evaluation of the protocol itself, will come after there is more experience with its use. What can be said now is that it is implementable in our environment, and it seems extensible.

It was mentioned in section 3.3 that the implementation of the resolver could be seen as a partial evaluation of the interprocess communication mechanisms in Berkeley UNIX 4.2 BSD. My experience here is that these mechanisms are functionally adequate, but difficult to learn and use. My hope is that the resolver will be a partial solution to these problems.

It is now time to discuss the limited data which is available on the resolver. This data is limited to the cpu time and memory space requirements of a test process using the resolver. Only the time consumed by the cpu is included since is extremely difficult to estimate the additional load placed on other components of the system. The space requirements can be determined by examining the program's symbol table. However, the time requirements must be determined by experiment. The purpose of the experiments we have run so far were two-fold. First, to give a rough idea of the complexity of the resolver. Second, to evaluate the caching strategy which was used.

It is very time consuming to set up experimental databases and to run name servers which contain these databases on appropriate hosts. So, the experiments are limited to very simple situations: specifically, we submitted fifty standard queries for data located on a name server which is running on a separate machine on the same Ethernet. Two cases were considered. First, the data is never present in the resolver's cache. Second, it is always present in the resolver's cache after the first query.

With the results from the first experiment it is possible to estimate the lookup cost in the case of a local area network where a central name server is used. It will also be possible to estimate the extent to which the resolver's caching mechanism contributes to that cost.

The second experiment makes it possible to estimate the fixed costs associated with making a query in a long running process. The initialization costs are relatively small and can be amortized over the fifty queries. So, this experiment makes it possible to estimate the savings which will be realized by the presence of a very successful cache.

The two experiments allow us to estimate the hit ratio which would be needed to justify using the cache. It should be remembered that there are many mitigating factors. The statistics vary somewhat depending on the load of the system, and version of the kernel. A new set of device drivers was installed during the course of the experiments and this affected the running times. The profiling package uses the system clock to sample which routine is running and therefore the time attributed to a particular routine is not precise. The resolver is not a fully tuned piece of code. Therefore, a concentrated effort to optimize the routines which handle the resolver's cache might significantly decrease the portion of the running time which is spent in them. The benefits of directing a query to the most appropriate name server first are not considered. Lastly, the load generated on other components of the system is not measured and may be considerable. With the above in mind, here are the results of the two experiments.

	Fifty Queries and Forty Nine Cache Hits	Fifty Queries and Fifty Cache Misses
Total Running Time	0.43 seconds	1.91 seconds
Percent of Time Spent in Garbage Collection	00.0 %	11.8 % (0.22 seconds)
Time Saved due To Cache Hits	1.26 seconds	0.00 seconds

Table 4

In the above table, the time spent in garbage collection is a good estimate of the time cost of caching, because the other activities of the resolver are necessary even if the caching mechanism is turned off. The time spent in garbage collection does not account for some costs of caching, such as additional hash table collisions, but such costs are very small in comparison to the cost of garbage collection. The time spent in the process with the successful cache is a good estimate of the time cost of initializing the resolver and manipulating its data structures. Thus, the time cost of sending 49 queries and receiving 49 responses can be estimated by subtracting the time spent in garbage collection and the time spent in the process with the successful cache from the running time of the process with the unsuccessful cache.

An estimate of the hit ratio which is necessary to justify using the cache can be made as follows. Obtain the time to send and receive a query by dividing 1.26 seconds by 49 queries yielding 0.0257 seconds per query. Then determine how many queries can be sent and received in 0.22 seconds (the time spent in garbage collection) by dividing 0.22 by 0.0257. The result is it would take about 8.5 cache hits out of 50 queries to break even. This translates to a hit ratio around 17 percent. I make no claim that this is anything more than a very rough estimate.

Assuming that this estimate is reasonable, two conclusions follow. First, the potential benefits of caching resource records are great enough to justify the effort of implementing a successful strategy for doing so. Second, it is likely that there are applications of name services which do not make predictable sequences of queries and do not query distant domains. The the small, per-process cache which has been implemented will not be useful in these applications, because it will be impossible to maintain a high enough cache hit ratio to justify using the cache. So, the current caching strategy is not entirely satisfactory.

The other purpose of the experiments was to provide information about the complexity of the resolver. The cpu time required by the resolver has already been indicated in Table 4.

The space requirements of the resolver are substantial. The working set of the test process is about 100K bytes. This is somewhat mitigated by the fact that much of this space is occupied by system routines which the resolver calls, and users are also likely to need. Also, some of this space is occupied by debugging code. The size of the code and data that the resolver would directly contribute to a user process is close to 23K bytes. When the caching mechanism is used another 12K bytes will be allocated on the heap, unless a large virtual circuit response is received. In theory, the heap space could grow to over 64K bytes, but this is unlikely. When caching is disabled, the heap space requirements will be slightly smaller, but will similarly depend on the size of responses. Still, the space the resolver occupies is large enough that sharing code and data between processes would be desirable if there were an easy way to do it.

The bottom line in this discussion is that the resolver is a relatively expensive utility to invoke. However, this is a function of its task. The time complexity of the resolver compares well with the time complexity of sending and receiving a datagram. It is harder to find an easy comparison for the resolver's space requirements. There are probably some savings to be had, but I believe that most of the space occupied by the resolver is required by the complexity of its tasks.

6. Future Work

There is a great deal more work that needs to be done on this project. This section points out the things which have been left undone.

6.1. Moving Towards Production Use

The resolver is a tool, and tools are worthless without applications. There are many potential applications, but none have yet been implemented. This is a fundamental next step in the development of the resolver.

Once, the resolver has been tested in production use, a substantial amount of work will need to be done to replace the host table. Changing over to the domain names system will be a delicate procedure. The difficulty of doing this should not be underestimated, and was a good incentive for including things in the proposal which were not immediately necessary.

One such feature is user updates. It is encouraging that the mechanics of performing user updates can be easily incorporated into the framework of the domain names proposal. However, the usefulness of user updates will be severely restricted until the related security problems are solved. It would be best if these problems were solved before the resolver is embedded in the system.

6.2. Further Experiments

Another important, but less pressing, concern is with tuning the performance of both resolvers and name servers. There are a large number of ways this can be done. Section 4.3.2 of this report discussed a variety of techniques for obtaining cache data. It is not known which of these techniques is best, either in general or for a particular set of applications or a particular environment.

Section 2.2.3 mentioned that there was a three-way trade-off between reliably obtaining consistent data, space used in storing redundant copies of data items, and the number of messages which would be transmitted. So, not only are the best mechanisms for caching data not known, but also the wisdom of caching the data has not been proved.

A related issue is the placement of the original data in primary and authoritative name servers. There are many possibilities here. A particular organization could have one centralized name server, or there could be several different name servers, each of which was responsible for some logical subset of the data, or each machine could have a name server which was responsible for the data local to that machine. The consequences of such choices have not been considered and are not known.

The above paragraphs are not meant to be an exhaustive list of the possibilities for increasing performance. They are meant to point out that there are very many design choices still to be made. There is a real need for a systematic way of evaluating and testing possible organizations of resolvers and name servers.

6.3. A Shared Memory Implementation

Section 4.1 mentioned the possibility of moving the resolver's code and data into the Berkeley UNIX kernel. The motivation for this was to avoid having a copy of the resolver code in every process which used the resolver and to provide an efficient way of maintaining a per machine cache. It is not known if such a move would be worth the loss of flexibility. Also, some features in the current design of the resolver would have to be changed. Specifically, the resolver's data structures are not designed with mutual exclusion in mind, and the current garbage collection strategy is not appropriate for a cache in which data persists for more than a few rounds of garbage collection, since it is wasteful to continually recopy data.

So, there are two things to be done in this area. First, determine if the resolver should be part of the kernel. Second, if it is to be moved into the kernel, make the necessary changes in the implementation. This choice is intimately related to an investigation of caching techniques and strategies.

7. Conclusion

The work that has been done so far is an important first step. A tool must exist before it can be applied. The current design is a basis for comparison and testing. It does not specify what security problems must be solved, but it does provide a context for their definition and eventual solution.

Although this report has suggested extensions to the original proposal, the basic design of the name server protocol seems sound. At least, the portion of the proposal dealing with the resolver is workable. A resolver can be implemented with moderate effort, and the resolution of a query makes moderate demands on memory space and processing time.

Acknowledgements

Finally, there are several people whose assistance deserves to be acknowledged. Dave Riggle, Doug Terry and Songnian Zhou are the other members of our group. We have had many helpful conversations, and my work has very much depended on their complementary efforts. Luis Filipe Cabrera and Domenico Ferrari have read earlier versions of this report, and have made many suggestions which have improved its quality. Paul Mockapetris authored the supporting rfc's. Ruth Brungardt was very helpful during our efforts to debug communications with the name server being implemented at ISI. However, I alone am responsible for any errors in this document.

References

[Bern81]	P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," Computing Surveys, June 1981
[Denn82]	D. Denning, "Cryptography and Data Security," reading, chapter 1, Addison-Wesley Publishing Company, 1982
[Leff83]	S. Leffler, R. Fabry, W. Joy, "A 4.2bsd Interprocess Communication Primer," University of California at Berkeley, July 1983
[RFC 768]	J. Postel, "User Datagram Protocol" RFC 768, USC/Information Sciences Institute, August 1980
[RFC 791]	J. Postel, "Internet Protocol - DARPA Internet Program Protocol Specification," RFC 791, USC/Information Sciences Institute, September 1981
[RFC 793]	J. Postel, "Transmission Control Protocol" RFC 793, USC/Information Sciences Institute, September 1981
[RFC 881]	P. Mockapetris, "Domain Names - Plan and Schedule," RFC 881, USC/Information Sciences Institute, November 1983
[RFC 882]	P. Mockapetris, "Domain Names - Concepts and Facilities," RFC 882, USC/Information Sciences Institute, November 1983
[RFC 883]	P. Mockapetris, "Domain Names - Implementation and Specification," RFC 883, USC/Information Sciences Institute, November 1983
[Rigg84]	D. Riggle, "A Name Server Database," University of California at Berkeley, May 1984
[Zhou84]	S. Zhou, "The Design and Implementation of the Berkeley Internet Name Domain (BIND) Servers," University of California at Berkeley, May 1984

Appendix A: Proposed Changes to Name Server Protocol

Additions to Packet Header

New bit flag(s): *Primary Required*

When this bit is set authoritative name servers deny their authority for the domain unless they are the primary server. This is for standard queries which require immediate access to the updated data, as with process rendezvous.

This bit flag is not strictly necessary, but it considerably simplifies the method of locating a primary name server for queries which are not updates. The alternative is a three step process which both complicates the coding of the resolver and leads to an increased number of messages. The three steps are to locate an authoritative name server, query that server for the start of authority record for the zone, and then to send the query to the primary name server.

It should be noted that in the case of process rendezvous, the issue can be avoided by requiring processes to register in a zone that is not replicated. However, there may be other applications where this is not an acceptable solution.

new opcodes:

updatea

This opcode indicates the addition of a single resource record to the name server database. Packet consists of header and an additional resource record, which is the resource record to be added to the name server data base.

updated

Here, the packet consists of header and an additional resource record, which is the resource record to be deleted from the name server data base. If the data portion of the resource record in the packet is of zero length, the deletion is to apply to all resource records of the given type and class for the given domain name. It should also be possible to specify wild cards in the type and class fields.

updatem

Here, the packet consists of header and two additional resource records. The first is the resource record in the data base to be modified. The second is what it is to be modified to. As with delete, no data implies replace all records of the given type, class and domain name with the single new record. To avoid the possibility that the modification could span machine boundaries, the domain names of the two records must be identical.

A modify may fail if neither the old or new data is present. In this case, the response is "no such domain name".

New Return Code(s): *NO CHANGE*

This return code is advisory only, it means that an update did not affect the data base. Perhaps, a response was lost and this was the second time the server received the update etc.

In general, it would be desirable to have more specific return codes. The existing proposal does not distinguish data that is not present from a type, class or domain name which is not present. Such distinction would allow for more informative error messages.

Additional Data in Presently Defined Types of Resource Records

At present, the name server records for the primary name server and other name servers for the domain are indistinguishable. This fact greatly complicates methods for locating the primary name server for a zone. Therefore, name server records should be extended by one octet to include an indication of whether or not they are primary for their zone.

There is not sufficient support for process level protocols. In general, such support should be provided by extending address type resource records to optionally include a process address in addition to a host address. The most compelling reason for extending the current address type, rather than providing new types for each process level protocol, is [illegible] while type and class information is orthogonal, process level protocols have little meaning without the underlying host level protocols.

In the internet case, this means extending the data section of address records by 3 octets. One octet to distinguish the protocol, and 2 octets for the port number. Addresses which contained process specific information could be distinguished by their greater length.

The Update Query Algorithm

Updates are processed as standard queries, except by authoritative name servers, which deny their authority unless they are the primary server. Name servers which deny their authority return a name server and address resource record for the primary name server for the zone.

An authoritative answer to an update query is an indication that the update was successful. In the case of add this means that the new record is in the data base. In the case of delete, the record is not in the data base. In the case of modify, the old version is not in the data base, and the new one is. The body of the answer is simply the body of the query.