Network Working Group
Request for Comments: 883

P. Mockapetris
ISI

November 1983

DOMAIN NAMES - IMPLEMENTATION and SPECIFICATION

This memo discusses the implementation of domain name servers and resolvers, specifies the format of transactions, and discusses the use of domain names in the context of existing mail systems and other network software.

This memo assumes that the reader is familiar with RFC 882, "Domain Names - Concepts and Facilities" which discusses the basic principles of domain names and their use.

The algorithms and internal data structures used in this memo are offered as suggestions rather than requirements; implementers are free to design their own structures so long as the same external behavior is achieved.

***** WARNING *****

This RFC contains format specifications which are preliminary and are included for purposes of explanation only. Do not attempt to use this information for actual implementations.

TABLE OF CONTENTS

INTRODUCTION

Overview

The goal of domain names is to provide a mechanism for naming resources in such a way that the names are usable in different hosts, networks, protocol families, internets, and administrative organizations.

From the user's point of view, domain names are useful as arguments to a local agent, called a resolver, which retrieves information associated with the domain name. Thus a user might ask for the host address or mail information associated with a particular domain name. To enable the user to request a particular type of information, an appropriate query type is passed to the resolver with the domain name. To the user, the domain tree is a single information space.

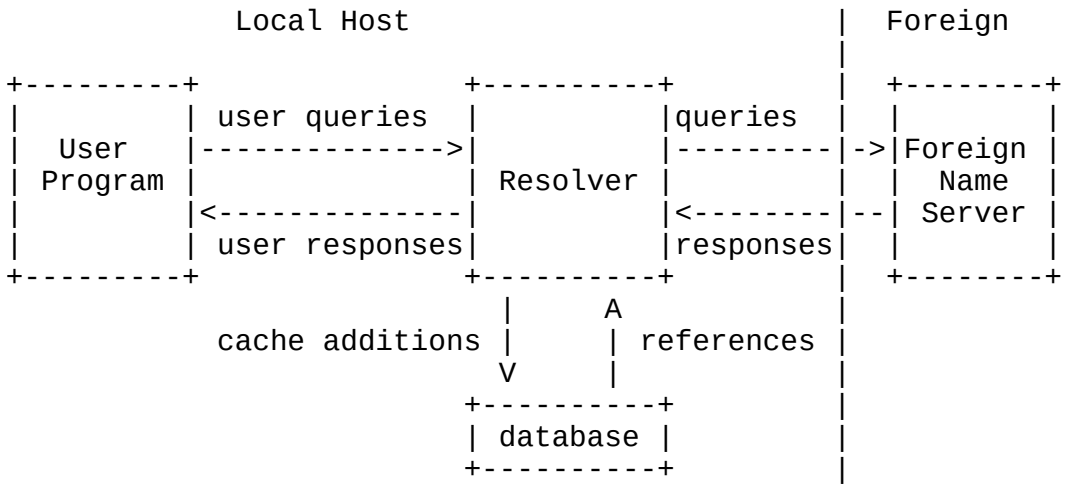
From the resolver's point of view, the database that makes up the domain space is distributed among various name servers. Different parts of the domain space are stored in different name servers, although a particular data item will usually be stored redundantly in two or more name servers. The resolver starts with knowledge of at least one name server. When the resolver processes a user query it asks a known name server for the information; in return, the resolver either receives the desired information or a referral to another name server. Using these referrals, resolvers learn the identities and contents of other name servers. Resolvers are responsible for dealing with the distribution of the domain space and dealing with the effects of name server failure by consulting redundant databases in other servers.

Name servers manage two kinds of data. The first kind of data held in sets called zones; each zone is the complete database for a particular subtree of the domain space. This data is called authoritative. A name server periodically checks to make sure that its zones are up to date, and if not obtains a new copy of updated zones from master files stored locally or in another name server. The second kind of data is cached data which was acquired by a local resolver. This data may be incomplete but improves the performance of the retrieval process when non-local data is repeatedly accessed. Cached data is eventually discarded by a timeout mechanism.

This functional structure isolates the problems of user interface, failure recovery, and distribution in the resolvers and isolates the database update and refresh problems in the name servers.

Implementation components

A host can participate in the domain name system in a number of ways, depending on whether the host runs programs that retrieve information from the domain system, name servers that answer queries from other hosts, or various combinations of both functions. The simplest, and perhaps most typical, configuration is shown below:

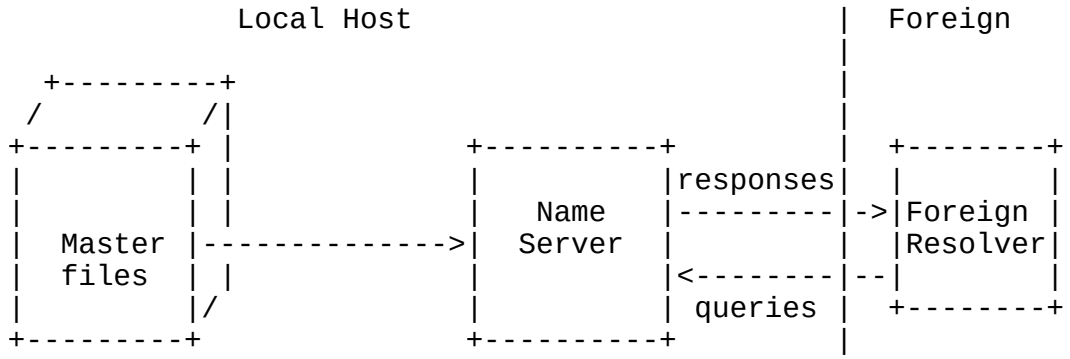


User programs interact with the domain name space through resolvers; the format of user queries and user responses is specific to the host and its operating system. User queries will typically be operating system calls, and the resolver and its database will be part of the host operating system. Less capable hosts may choose to implement the resolver as a subroutine to be linked in with every program that needs its services.

Resolvers answer user queries with information they acquire via queries to foreign name servers, and may also cache or reference domain information in the local database.

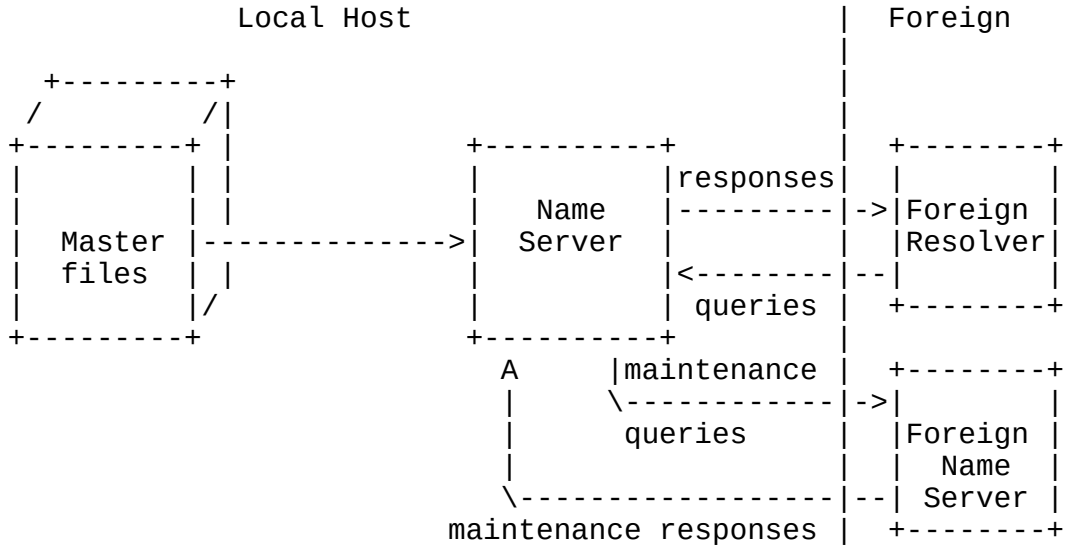
Note that the resolver may have to make several queries to several different foreign name servers to answer a particular user query, and hence the resolution of a user query may involve several network accesses and an arbitrary amount of time. The queries to foreign name servers and the corresponding responses have a standard format described in this memo, and may be datagrams.

Depending on its capabilities, a name server could be a stand alone program on a dedicated machine or a process or processes on a large timeshared host. A simple configuration might be:



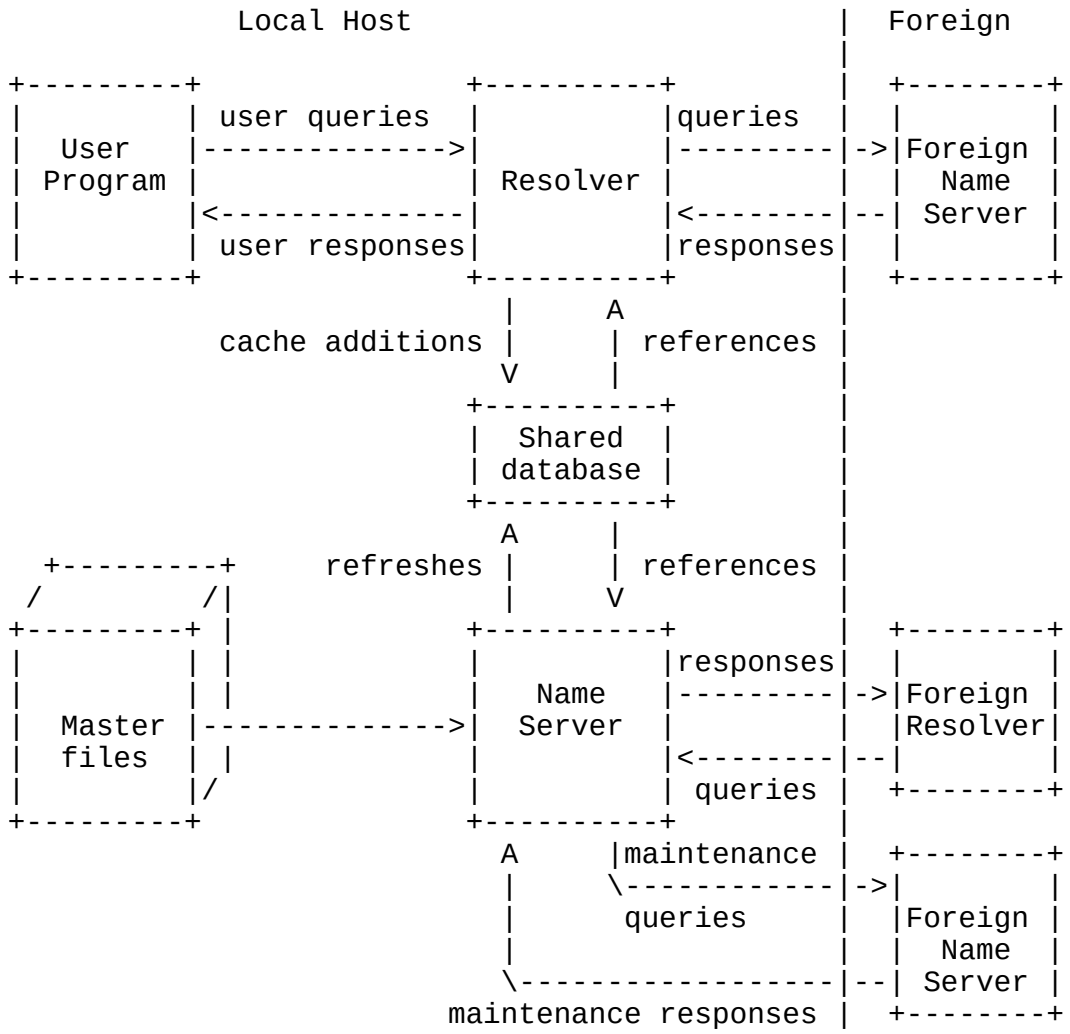
Here the name server acquires information about one or more zones by reading master files from its local file system, and answers queries about those zones that arrive from foreign resolvers.

A more sophisticated name server might acquire zones from foreign name servers as well as local master files. This configuration is shown below:



In this configuration, the name server periodically establishes a virtual circuit to a foreign name server to acquire a copy of a zone or to check that an existing copy has not changed. The messages sent for these maintenance activities follow the same form as queries and responses, but the message sequences are somewhat different.

The information flow in a host that supports all aspects of the domain name system is shown below:



The shared database holds domain space data for the local name server and resolver. The contents of the shared database will typically be a mixture of authoritative data maintained by the periodic refresh operations of the name server and cached data from previous resolver requests. The structure of the domain data and the necessity for synchronization between name servers and resolvers imply the general characteristics of this database, but the actual format is up to the local implementer. This memo suggests a multiple tree format.

This memo divides the implementation discussion into sections:

NAME SERVER TRANSACTIONS, which discusses the formats for name servers queries and the corresponding responses.

NAME SERVER MAINTENANCE, which discusses strategies, algorithms, and formats for maintaining the data residing in name servers. These services periodically refresh the local copies of zones that originate in other hosts.

RESOLVER ALGORITHMS, which discusses the internal structure of resolvers. This section also discusses data base sharing between a name server and a resolver on the same host.

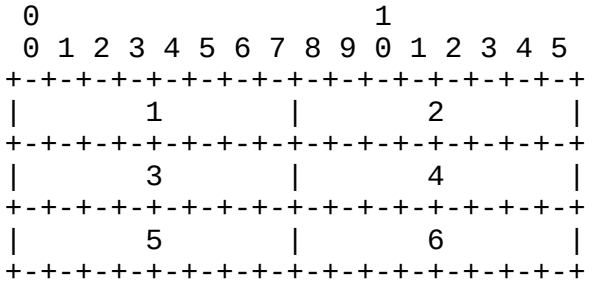
DOMAIN SUPPORT FOR MAIL, which discusses the use of the domain system to support mail transfer.

Conventions

The domain system has several conventions dealing with low-level, but fundamental, issues. While the implementer is free to violate these conventions WITHIN HIS OWN SYSTEM, he must observe these conventions in ALL behavior observed from other hosts.

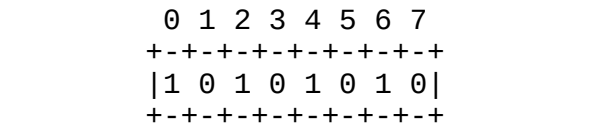
********** Data Transmission Order **********

The order of transmission of the header and data described in this document is resolved to the octet level. Whenever a diagram shows a group of octets, the order of transmission of those octets is the normal order in which they are read in English. For example, in the following diagram the octets are transmitted in the order they are numbered.



Transmission Order of Bytes

Whenever an octet represents a numeric quantity the left most bit in the diagram is the high order or most significant bit. That is, the bit labeled 0 is the most significant bit. For example, the following diagram represents the value 170 (decimal).



Significance of Bits

Similarly, whenever a multi-octet field represents a numeric quantity the left most bit of the whole field is the most significant bit. When a multi-octet quantity is transmitted the most significant octet is transmitted first.

********** Character Case **********

All comparisons between character strings (e.g. labels, domain names, etc.) are done in a case-insensitive manner.

When data enters the domain system, its original case should be preserved whenever possible. In certain circumstances this cannot be done. For example, if two domain names x.y and X.Y are entered into the domain database, they are interpreted as the same name, and hence may have a single representation. The basic rule is that case can be discarded only when data is used to define structure in a database, and two names are identical when compared in a case insensitive manner.

Loss of case sensitive data must be minimized. Thus while data for x.y and X.Y may both be stored under x.y, data for a.x and B.X can be stored as a.x and B.x, but not A.x, A.X, b.x, or b.X. In general, this prevents the first component of a domain name from loss of case information.

Systems administrators who enter data into the domain database should take care to represent the data they supply to the domain system in a case-consistent manner if their system is case-sensitive. The data distribution system in the domain system will ensure that consistent representations are preserved.

Design philosophy

The design presented in this memo attempts to provide a base which will be suitable for several existing networks. An equally important goal is to provide these services within a framework that is capable of adjustment to fit the evolution of services in early clients as well as to accommodate new networks.

Since it is impossible to predict the course of these developments, the domain system attempts to provide for evolution in the form of an extensible framework. This section describes the areas in which we expect to see immediate evolution.

DEFINING THE DATABASE

This memo defines methods for partitioning the database and data for host names, host addresses, gateway information, and mail support. Experience with this system will provide guidance for future additions.

While the present system allows for many new RR types, classes, etc., we feel that it is more important to get the basic services in operation than to cover an exhaustive set of information. Hence we have limited the data types to those we felt were essential, and would caution designers to avoid implementations which are based on the number of existing types and classes. Extensibility in this area is very important.

While the domain system provides techniques for partitioning the database, policies for administrating the orderly connection of separate domains and guidelines for constructing the data that makes up a particular domain will be equally important to the success of the system. Unfortunately, we feel that experience with prototype systems will be necessary before this question can be properly addressed. Thus while this memo has minimal discussion of these issues, it is a critical area for development.

TYING TOGETHER INTERNETS

Although it is very difficult to characterize the types of networks, protocols, and applications that will be clients of the domain system, it is very obvious that some of these applications will cross the boundaries of network and protocol. At the very least, mail is such a service.

Attempts to unify two such systems must deal with two major problems:

1. Differing formats for environment sensitive data. For example,

network addresses vary in format, and it is unreasonable to expect to enforce consistent conventions.

2. Connectivity may require intermediaries. For example, it is a frequent occurence that mail is sent between hosts that share no common protocol.

The domain system acknowledges that these are very difficult problems, and attempts to deal with both problems through its CLASS mechanism:

1. The CLASS field in RRs allows data to be tagged so that all programs in the domain system can identify the format in use.

2. The CLASS field allows the requestor to identify the format of data which can be understood by the requestor.

3. The CLASS field guides the search for the requested data.

The last point is central to our approach. When a query crosses protocol boundaries, it must be guided though agents capable of performing whatever translation is required. For example, when a mailer wants to identify the location of a mailbox in a portion of the domain system that doesn't have a compatible protocol, the query must be guided to a name server that can cross the boundary itself or form one link in a chain that can span the differences.

If query and response transport were the only problem, then this sort of problem could be dealt with in the name servers themselves. However, the applications that will use domain service have similar problems. For example, mail may need to be directed through mail gateways, and the characteristics of one of the environments may not permit frequent connectivity between name servers in all environments.

These problems suggest that connectivity will be achieved through a variety of measures:

Translation name servers that act as relays between different protocols.

Translation application servers that translate application level transactions.

Default database entries that route traffic through application level forwarders in ways that depend on the class of the requestor.

While this approach seems best given our current understanding of

the problem, we realize that the approach of using resource data that transcends class may be appropriate in future designs or applications. By not defining class to be directly related to protocol, network, etc., we feel that such services could be added by defining a new "universal" class, while the present use of class will provide immediate service.

This problem requires more thought and experience before solutions can be discovered. The concepts of CLASS, recursive servers and other mechanisms are intended as tools for acquiring experience and not as final solutions.

NAME SERVER TRANSACTIONS

Introduction

The primary purpose of name servers is to receive queries from resolvers and return responses. The overall model of this service is that a program (typically a resolver) asks the name server questions (queries) and gets responses that either answer the question or refer the questioner to another name server. Other functions related to name server database maintenance use similar procedures and formats and are discussed in a section later in this memo.

There are three kinds of queries presently defined:

1. Standard queries that ask for a specified resource attached to a given domain name.

2. Inverse queries that specify a resource and ask for a domain name that possesses that resource.

3. Completion queries that specify a partial domain name and a target domain and ask that the partial domain name be completed with a domain name close to the target domain.

This memo uses an unqualified reference to queries to refer to either all queries or standard queries when the context is clear.

Query and response transport

Name servers and resolvers use a single message format for all communications. The message format consists of a variable-length octet string which includes binary values.

The messages used in the domain system are designed so that they can be carried using either datagrams or virtual circuits. To accommodate the datagram style, all responses carry the query as part of the response.

While the specification allows datagrams to be used in any context, some activities are ill suited to datagram use. For example, maintenance transactions and recursive queries typically require the error control of virtual circuits. Thus datagram use should be restricted to simple queries.

The domain system assumes that a datagram service provides:

1. A non-reliable (i.e. best effort) method of transporting a message of up to 512 octets.

Hence datagram messages are limited to 512 octets. If a datagram message would exceed 512 octets, it is truncated and a truncation flag is set in its header.

2. A message size that gives the number of octets in the datagram.

The main implications for programs accessing name servers via datagrams are:

1. Datagrams should not be used for maintenance transactions and recursive queries.

2. Since datagrams may be lost, the originator of a query must perform error recovery (such as retransmissions) as appropriate.

3. Since network or host delay may cause retransmission when a datagram has not been lost, the originator of a query must be ready to deal with duplicate responses.

The domain system assumes that a virtual circuit service provides:

1. A reliable method of transmitting a message of up to 65535 octets.

2. A message size that gives the number of octets in the message.

   If the virtual circuit service does not provide for message boundary detection or limits transmission size to less than 65535 octets, then messages are prefaced with an unsigned 16 bit length field and broken up into separate transmissions as required. The length field is only prefaced on the first message. This technique is used for TCP virtual circuits.

3. Multiple messages may be sent over a virtual circuit.

4. A method for closing a virtual circuit.

5. A method for detecting that the other party has requested that the virtual circuit be closed.

The main implications for programs accessing name servers via virtual circuits are:

1. Either end of a virtual circuit may initiate a close when there is no activity in progress. The other end should comply.

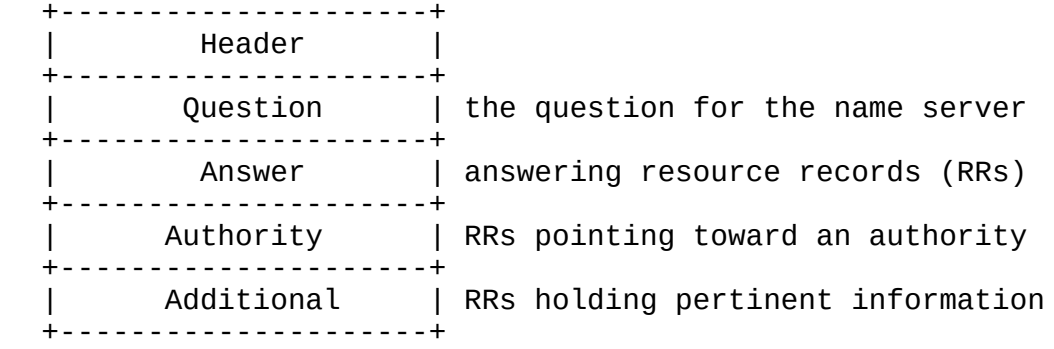
The decision to initiate a close is a matter of individual site policy; some name servers may leave a virtual circuit open for an indeterminate period following a query to allow for subsequent queries; other name servers may choose to initiate a close following the completion of the first query on a virtual circuit. Of course, name servers should not close the virtual circuit in the midst of a multiple message stream used for zone transfer.

2. Since network delay may cause one end to erroneously believe that no activity is in progress, a program which receives a virtual circuit close while a query is in progress should close the virtual circuit and resubmit the query on a new virtual circuit.

All messages may use a compression scheme to reduce the space consumed by repetitive domain names. The use of the compression scheme is optional for the sender of a message, but all receivers must be capable of decoding compressed domain names.

Overall message format

All messages sent by the domain system are divided into 5 sections (some of which are empty in certain cases) shown below:



The header section is always present. The header includes fields that specify which of the remaining sections are present, and also specify whether the message is a query, inverse query, completion query, or response.

The question section contains fields that describe a question to a name server. These fields are a query type (QTYPE), a query class (QCLASS), and a query domain name (QNAME).

The last three sections have the same format: a possibly empty list of concatenated resource records (RRs). The answer section contains RRs that answer the question; the authority section

contains RRs that point toward an authoritative name server; the additional records section contains RRs which relate to the query, but are not strictly answers for the question.

The next two sections of this memo illustrate the use of these message sections through examples; a detailed discussion of data formats follows the examples.

The contents of standard queries and responses

When a name server processes a standard query, it first determines whether it is an authority for the domain name specified in the query.

If the name server is an authority, it returns either:

1. the specified resource information

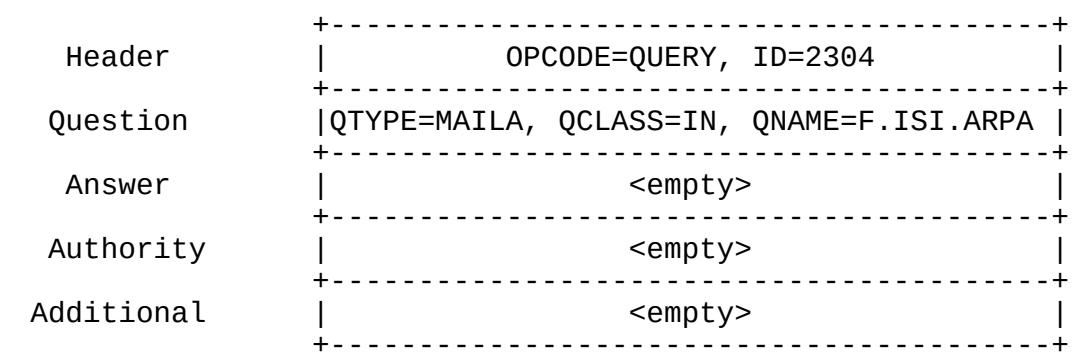
2. an indication that the specified name does not exist

3. an indication that the requested resource information does not exist

If the name server is not an authority for the specified name, it returns whatever relevant resource information it has along with resource records that the requesting resolver can use to locate an authoritative name server.

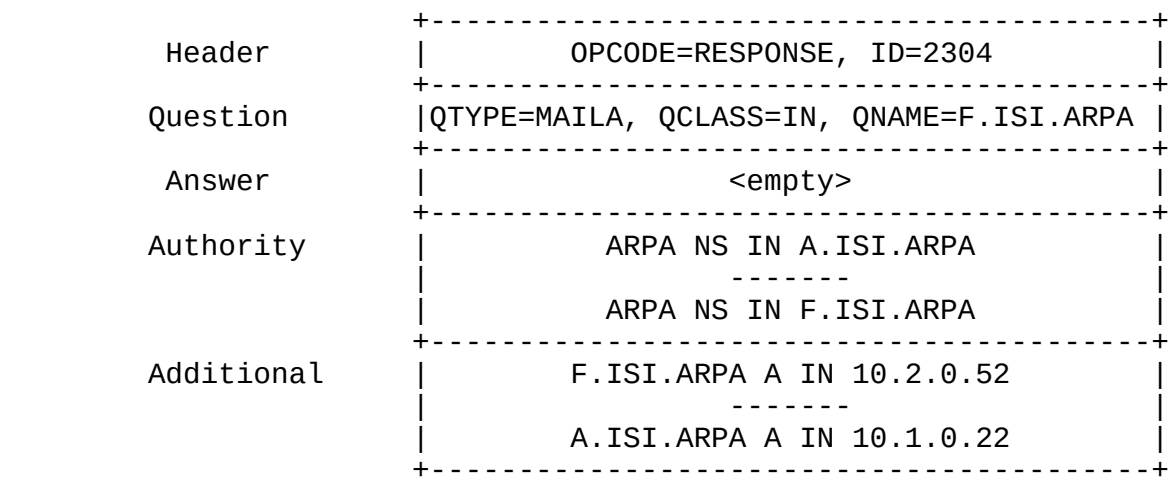
Standard query and response example

The overall structure of a query for retrieving information for Internet mail for domain F.ISI.ARPA is shown below:



The header includes an opcode field that specifies that this datagram is a query, and an ID field that will be used to associate replies with the original query. (Some additional header fields have been omitted for clarity.) The question section specifies that the type of the query is for mail agent information, that only ARPA Internet information is to be considered, and that the domain name of interest is F.ISI.ARPA. The remaining sections are empty, and would not use any octets in a real query.

One possible response to this query might be:



This type of response would be returned by a name server that was not an authority for the domain name F.ISI.ARPA. The header field specifies that the datagram is a response to a query with an ID of 2304. The question section is copied from the question section in the query datagram.

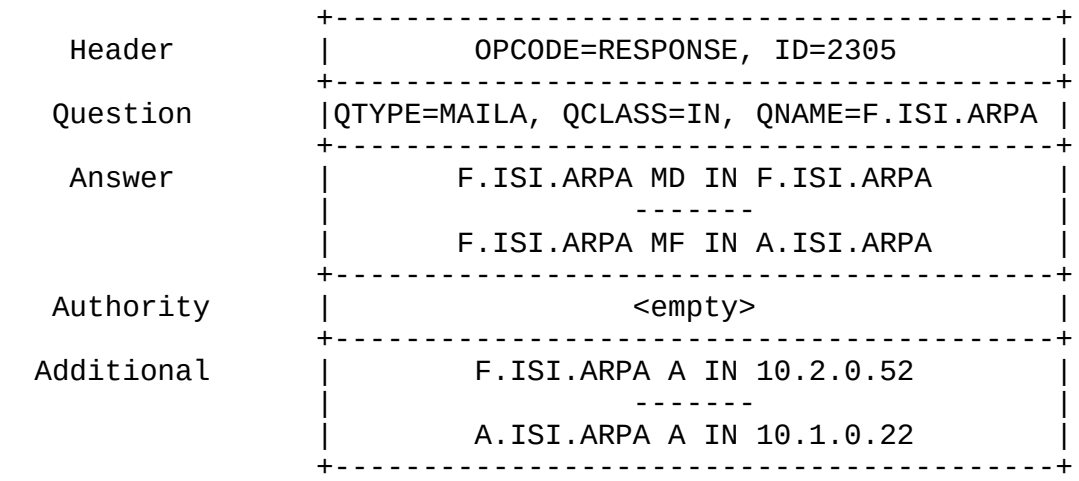
The answer section is empty because the name server did not have any information that would answer the query. (Name servers may happen to have cached information even if they are not authoritative for the query.)

The best that this name server could do was to pass back information for the domain ARPA. The authority section specifies two name servers for the domain ARPA using the Internet family: A.ISI.ARPA and F.ISI.ARPA. Note that it is merely a coincidence that F.ISI.ARPA is a name server for ARPA as well as the subject of the query.

In this case, the name server included in the additional records section the Internet addresses for the two hosts specified in the authority section. Such additional data is almost always available.

Given this response, the process that originally sent the query might resend the query to the name server on A.ISI.ARPA, with a new ID of 2305.

The name server on A.ISI.ARPA might return a response:



This query was directed to an authoritative name server, and hence the response includes an answer but no authority records. In this case, the answer section specifies that mail for F.ISI.ARPA can either be delivered to F.ISI.ARPA or forwarded to A.ISI.ARPA. The additional records section specifies the Internet addresses of these hosts.

The contents of inverse queries and responses

Inverse queries reverse the mappings performed by standard query operations; while a standard query maps a domain name to a resource, an inverse query maps a resource to a domain name. For example, a standard query might bind a domain name to a host address; the corresponding inverse query binds the host address to a domain name.

Inverse query mappings are not guaranteed to be unique or complete because the domain system does not have any internal mechanism for determining authority from resource records that parallels the capability for determining authority as a function of domain name. In general, resolvers will be configured to direct inverse queries to a name server which is known to have the desired information.

Name servers are not required to support any form of inverse queries; it is anticipated that most name servers will support address to domain name conversions, but no other inverse mappings. If a name server receives an inverse query that it does not support, it returns an error response with the "Not Implemented" error set in the header. While inverse query support is optional, all name servers must be at least able to return the error response.

When a name server processes an inverse query, it either returns:

1. zero, one, or multiple domain names for the specified resource

2. an error code indicating that the name server doesn't support inverse mapping of the specified resource type.

Inverse query and response example

The overall structure of an inverse query for retrieving the domain name that corresponds to Internet address 10.2.0.52 is shown below:

```
                  +-----------------------------------------+
    Header        |          OPCODE=IQUERY, ID=997          |
                  +-----------------------------------------+
    Question      |                 <empty>                 |
                  +-----------------------------------------+
    Answer        |        <anyname> A IN 10.2.0.52         |
                  +-----------------------------------------+
    Authority     |                 <empty>                 |
                  +-----------------------------------------+
    Additional    |                 <empty>                 |
                  +-----------------------------------------+
```

This query asks for a question whose answer is the Internet style address 10.2.0.52. Since the owner name is not known, any domain name can be used as a placeholder (and is ignored). The response to this query might be:

```
                  +-----------------------------------------+
    Header        |         OPCODE=RESPONSE, ID=997         |
                  +-----------------------------------------+
    Question      |   QTYPE=A, QCLASS=IN, QNAME=F.ISI.ARPA  |
                  +-----------------------------------------+
    Answer        |       F.ISI.ARPA A IN 10.2.0.52         |
                  +-----------------------------------------+
    Authority     |                 <empty>                 |
                  +-----------------------------------------+
    Additional    |                 <empty>                 |
                  +-----------------------------------------+
```

Note that the QTYPE in a response to an inverse query is the same as the TYPE field in the answer section of the inverse query. Responses to inverse queries may contain multiple questions when the inverse is not unique.

Completion queries and responses

Completion queries ask a name server to complete a partial domain name and return a set of RRs whose domain names meet a specified set of criteria for "closeness" to the partial input. This type of query can provide a local shorthand for domain names or command completion similar to that in TOPS-20.

Implementation of completion query processing is optional in a name server. However, a name server must return a "Not Implemented" (NI) error response if it does not support completion.

The arguments in a completion query specify:

1. A type in QTYPE that specifies the type of the desired name. The type is used to restrict the type of RRs which will match the partial input so that completion queries can be used for mailbox names, host names, or any other type of RR in the domain system without concern for matches to the wrong type of resource.

2. A class in QCLASS which specifies the desired class of the RR.

3. A partial domain name that gives the input to be completed. All returned RRs will begin with the partial string. The search process first looks for names which qualify under the assumption that the partial string ends with a full label ("whole label match"); if this search fails, the search continues under the assumption that the last label in the partial sting may be an incomplete label ("partial label match"). For example, if the partial string "Smith" was used in a mailbox completion, it would match Smith@ISI.ARPA in preference to Smithsonian@ISI.ARPA.

   The partial name is supplied by the user through the user program that is using domain services. For example, if the user program is a mail handler, the string might be "Mockap" which the user intends as a shorthand for the mailbox Mockapetris@ISI.ARPA; if the user program is TELNET, the user might specify "F" for F.ISI.ARPA.

   In order to make parsing of messages consistent, the partial name is supplied in domain name format (i.e. a sequence of labels terminated with a zero length octet). However, the trailing root label is ignored during matching.

4. A target domain name which specifies the domain which is to be examined for matches. This name is specified in the additional

section using a NULL RR. All returned names will end with the target name.

The user program which constructs the query uses the target name to restrict the search. For example, user programs running at ISI might restrict completion to names that end in ISI.ARPA; user programs running at MIT might restrict completion to the domain MIT.ARPA.

The target domain name is also used by the resolver to determine the name server which should be used to process the query. In general, queries should be directed to a name server that is authoritative for the target domain name. User programs which wish to provide completion for a more than one target can issue multiple completion queries, each directed at a different target. Selection of the target name and the number of searches will depend on the goals of the user program.

5. An opcode for the query. The two types of completion queries are "Completion Query - Multiple", or CQUERYM, which asks for all RRs which could complete the specified input, and "Completion Query - Unique", or CQUERYU, which asks for the "best" completion.

   CQUERYM is used by user programs which want to know if ambiguities exist or wants to do its own determinations as to the best choice of the available candidates.

   CQUERYU is used by user programs which either do not wish to deal with multiple choices or are willing to use the closeness criteria used by CQUERYU to select the best match.

When a name server receives either completion query, it first looks for RRs that begin (on the left) with the same labels as are found in QNAME (with the root deleted), and which match the QTYPE and QCLASS. This search is called "whole label" matching. If one or more hits are found the name server either returns all of the hits (CQUERYM) or uses the closeness criteria described below to eliminate all but one of the matches (CQUERYU).

If the whole label match fails to find any candidates, then the name server assumes that the rightmost label of QNAME (after root deletion) is not a complete label, and looks for candidates that would match if characters were added (on the right) to the rightmost label of QNAME. If one or more hits are found the name server either returns all of the hits (CQUERYM) or uses the closeness criteria described below to eliminate all but one of the matches (CQUERYU).

If a CQUERYU query encounters multiple hits, it uses the following sequence of rules to discard multiple hits:

1. Discard candidates that have more labels than others. Since all candidates start with the partial name and end with the target name, this means that we select those entries that require the fewest number of added labels. For example, a host search with a target of "ISI.ARPA" and a partial name of "A" will select A.ISI.ARPA in preference to A.IBM-PCS.ISI.ARPA.

2. If partial label matching was used, discard those labels which required more characters to be added. For example, a mailbox search for partial "X" and target "ISI.ARPA" would prefer XX@ISI.ARPA to XYZZY@ISI.ARPA.

If multiple hits are still present, return all hits.

Completion query mappings are not guaranteed to be unique or complete because the domain system does not have any internal mechanism for determining authority from a partial domain name that parallels the capability for determining authority as a function of a complete domain name. In general, resolvers will be configured to direct completion queries to a name server which is known to have the desired information.

When a name server processes a completion query, it either returns:

1. An answer giving zero, one, or more possible completions.

2. an error response with Not Implemented (NI) set.

Completion query and response example

Suppose that the completion service was used by a TELNET program to allow a user to specify a partial domain name for the desired host. Thus a user might ask to be connected to "B". Assuming that the query originated from an ISI machine, the query might look like:

```
                  +-----------------------------------------+
   Header         |         OPCODE=CQUERYU, ID=409          |
                  +-----------------------------------------+
  Question        |       QTYPE=A, QCLASS=IN, QNAME=B       |
                  +-----------------------------------------+
   Answer         |                 <empty>                 |
                  +-----------------------------------------+
  Authority       |                 <empty>                 |
                  +-----------------------------------------+
 Additional       |             ISI.ARPA NULL IN            |
                  +-----------------------------------------+
```

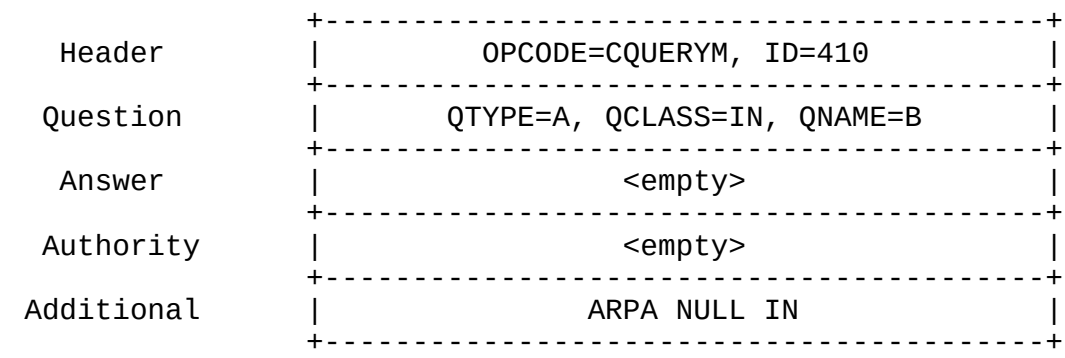
The partial name in the query is "B", the mappings of interest are ARPA Internet address records, and the target domain is ISI.ARPA. Note that NULL is a special type of NULL resource record that is used as a placeholder and has no significance; NULL RRs obey the standard format but have no other function.

The response to this completion query might be:

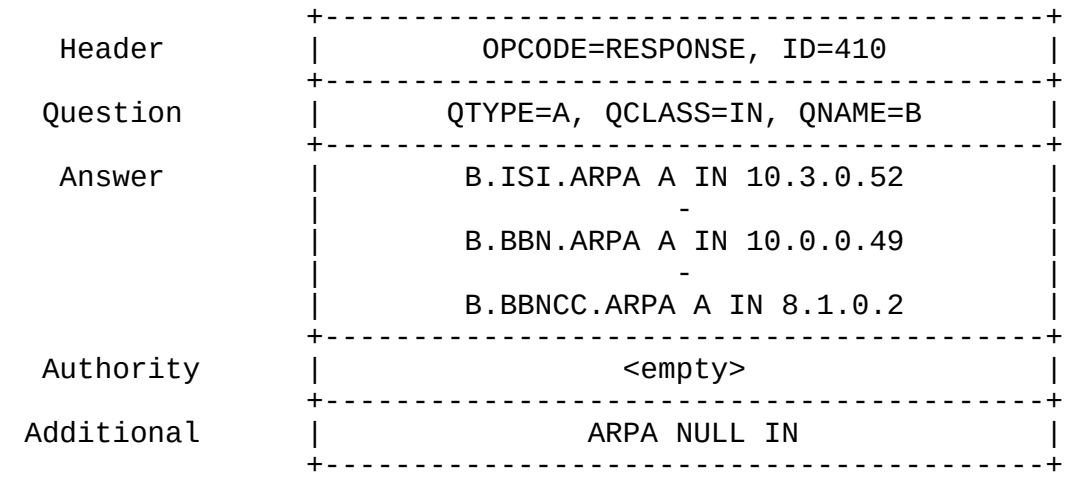
```
                  +-----------------------------------------+
   Header         |         OPCODE=RESPONSE, ID=409         |
                  +-----------------------------------------+
  Question        |       QTYPE=A, QCLASS=IN, QNAME=B       |
                  +-----------------------------------------+
   Answer         |        B.ISI.ARPA A IN 10.3.0.52        |
                  +-----------------------------------------+
  Authority       |                 <empty>                 |
                  +-----------------------------------------+
 Additional       |             ISI.ARPA NULL IN            |
                  +-----------------------------------------+
```

This response has completed B to mean B.ISI.ARPA.

Another query might be:



This query is similar to the previous one, but specifies a target of ARPA rather than ISI.ARPA. It also allows multiple matches. In this case the same name server might return:



This response contains three answers, B.ISI.ARPA, B.BBN.ARPA, and B.BBNCC.ARPA.

Recursive Name Service

Recursive service is an optional feature of name servers.

When a name server receives a query regarding a part of the name space which is not in one of the name server's zones, the standard response is a message that refers the requestor to another name server. By iterating on these referrals, the requestor eventually is directed to a name server that has the required information.

Name servers may also implement recursive service. In this type of service, a name server either answers immediately based on local zone information, or pursues the query for the requestor and returns the eventual result back to the original requestor.

A name server that supports recursive service sets the Recursion Available (RA) bit in all responses it generates. A requestor asks for recursive service by setting the Recursion Desired (RD) bit in queries. In some situations where recursive service is the only path to the desired information (see below), the name server may go recursive even if RD is zero.

If a query requests recursion (RD set), but the name server does not support recursion, and the query needs recursive service for an answer, the name server returns a "Not Implemented" (NI) error code. If the query can be answered without recursion since the name server is authoritative for the query, it ignores the RD bit.

Because of the difficulty in selecting appropriate timeouts and error handling, recursive service is best suited to virtual circuits, although it is allowed for datagrams.

Recursive service is valuable in several special situations:

In a system of small personal computers clustered around one or more large hosts supporting name servers, the recursive approach minimizes the amount of code in the resolvers in the personal computers. Such a design moves complexity out of the resolver into the name server, and may be appropriate for such systems.

Name servers on the boundaries of different networks may wish to offer recursive service to create connectivity between different networks. Such name servers may wish to provide recursive service regardless of the setting of RD.

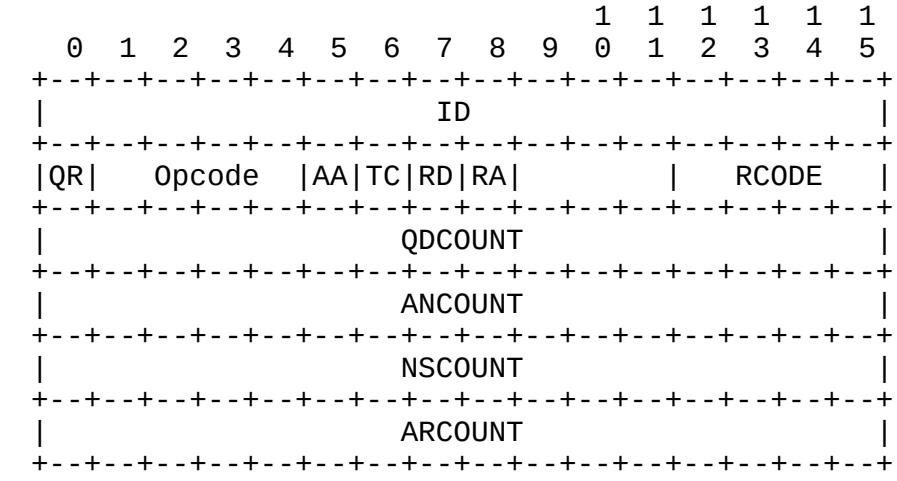
Name servers that translate between domain name service and some other name service may wish to adopt the recursive style. Implicit recursion may be valuable here as well.

These concepts are still under development.

Header section format

```
+-----------------------------------------------+
|                                               |
|             *****  WARNING  *****             |
|                                               |
|  The following format is preliminary and is   |
| included for purposes of explanation only. In |
| particular, the size and position of the      |
| OPCODE, RCODE fields and the number and       |
| meaning of the single bit fields are subject  |
| to change.                                    |
|                                               |
+-----------------------------------------------+
```

The header contains the following fields:



where:

ID - A 16 bit identifier assigned by the program that generates any kind of query. This identifier is copied into all replies and can be used by the requestor to relate replies to outstanding questions.

QR - A one bit field that specifies whether this message is a query (0), or a response (1).

OPCODE - A four bit field that specifies kind of query in this message. This value is set by the originator of a query and copied into the response. The values are:

0 a standard query (QUERY)

1 an inverse query (IQUERY)

2 an completion query allowing multiple answers (CQUERYM)

2 an completion query requesting a single answer (CQUERYU)

4-15 reserved for future use

AA - Authoritative Answer - this bit is valid in responses, and specifies that the responding name server is an authority for the domain name in the corresponding query.

TC - TrunCation - specifies that this message was truncated due to length greater than 512 characters. This bit is valid in datagram messages but not in messages sent over virtual circuits.

RD - Recursion Desired - this bit may be set in a query and is copied into the response. If RD is set, it directs the name server to pursue the query recursively. Recursive query support is optional.

RA - Recursion Available - this be is set or cleared in a response, and denotes whether recursive query support is available in the name server.

RCODE - Response code - this 4 bit field is set as part of responses. The values have the following interpretation:

0 No error condition

1 Format error - The name server was unable to interpret the query.

2 Server failure - The name server was unable to process this query due to a problem with the name server.

3 Name Error - Meaningful only for responses from an authoritative name server, this code signifies that the domain name referenced in the query does not exist.

4 Not Implemented - The name server does not support the requested kind of query.

5 Refused - The name server refuses to perform the specified operation for policy reasons. For example, a name server may not wish to provide the information to the particular requestor, or a name server may not wish to perform a particular operation (e.g. zone transfer) for particular data.

6-15 Reserved for future use.

QDCOUNT - an unsigned 16 bit integer specifying the number of entries in the question section.

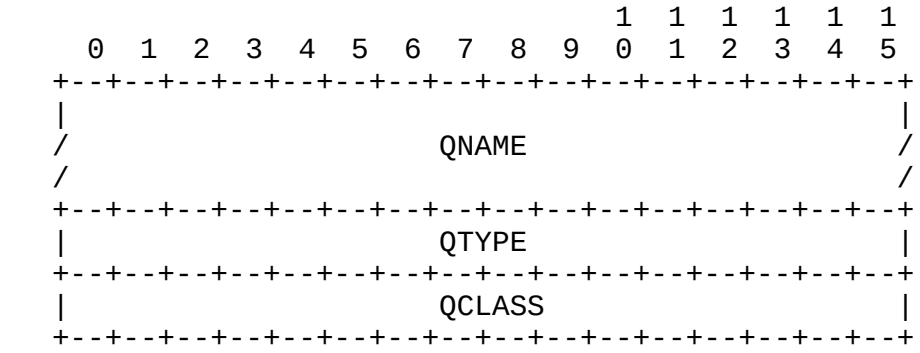
ANCOUNT - an unsigned 16 bit integer specifying the number of resource records in the answer section.

NSCOUNT - an unsigned 16 bit integer specifying the number of name server resource records in the authority records section.

ARCOUNT - an unsigned 16 bit integer specifying the number of resource records in the additional records section.

Question section format

The question section is used in all kinds of queries other than inverse queries. In responses to inverse queries, this section may contain multiple entries; for all other responses it contains a single entry. Each entry has the following format:



where:

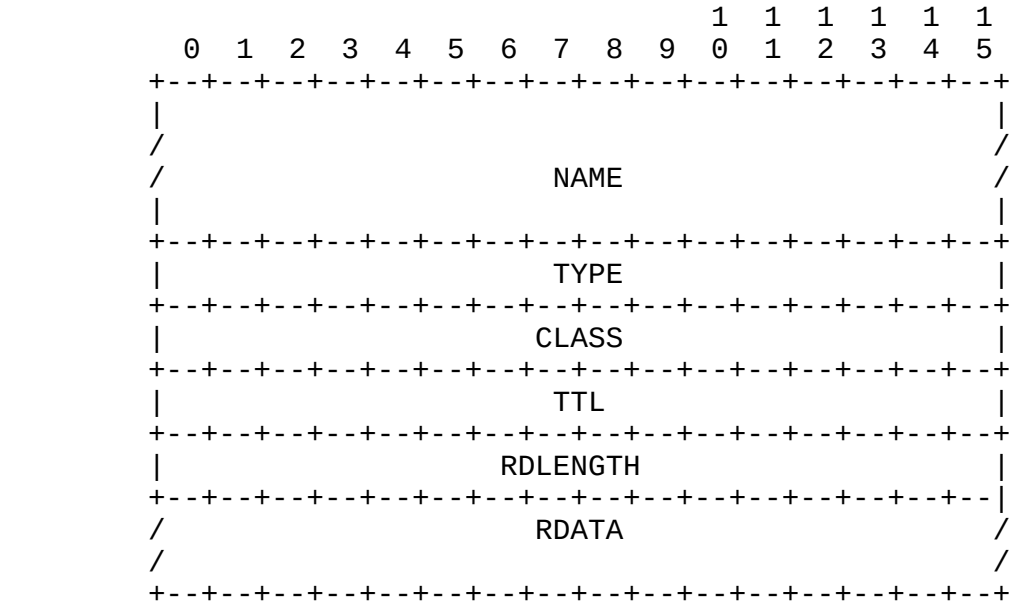
QNAME - a variable number of octets that specify a domain name. This field uses the compressed domain name format described in the next section of this memo. This field can be used to derive a text string for the domain name. Note that this field may be an odd number of octets; no padding is used.

QTYPE - a two octet code which specifies the type of the query. The values for this field include all codes valid for a TYPE field, together with some more general codes which can match more than one type of RR. For example, QTYPE might be A and only match type A RRs, or might be MAILA, which matches MF and MD type RRs. The values for this field are listed in Appendix 2.

QCLASS - a two octet code that specifies the class of the query. For example, the QCLASS field is IN for the ARPA Internet, CS for the CSNET, etc. The numerical values are defined in Appendix 2.

Resource record format

The answer, authority, and additional sections all share the same format: a variable number of resource records, where the number of records is specified in the corresponding count field in the header. Each resource record has the following format:



where:

NAME - a compressed domain name to which this resource record pertains.

TYPE - two octets containing one of the RR type codes defined in Appendix 2. This field specifies the meaning of the data in the RDATA field.

CLASS - two octets which specify the class of the data in the RDATA field.

TTL - a 16 bit unsigned integer that specifies the time interval (in seconds) that the resource record may be cached before it should be discarded. Zero values are interpreted to mean that the RR can only be used for the transaction in progress, and should not be cached. For example, SOA records are always distributed with a zero TTL to prohibit caching. Zero values can also be used for extremely volatile data.

RDLENGTH- an unsigned 16 bit integer that specifies the length in octets of the RDATA field.

RDATA - a variable length string of octets that describes the resource. The format of this information varies according to the TYPE and CLASS of the resource record. For example, the if the TYPE is A and the CLASS is IN, the RDATA field is a 4 octet ARPA Internet address.

Formats for particular resource records are shown in Appendicies 2 and 3.

Domain name representation and compression

Domain names messages are expressed in terms of a sequence of labels. Each label is represented as a one octet length field followed by that number of octets. Since every domain name ends with the null label of the root, a compressed domain name is terminated by a length byte of zero. The high order two bits of the length field must be zero, and the remaining six bits of the length field limit the label to 63 octets or less.

To simplify implementations, the total length of label octets and label length octets that make up a domain name is restricted to 255 octets or less. Since the trailing root label and its dot are not printed, printed domain names are 254 octets or less.

Although labels can contain any 8 bit values in octets that make up a label, it is strongly recommended that labels follow the syntax described in Appendix 1 of this memo, which is compatible with existing host naming conventions. Name servers and resolvers must compare labels in a case-insensitive manner, i.e. A=a, and hence all character strings must be ASCII with zero parity. Non-alphabetic codes must match exactly.

Whenever possible, name servers and resolvers must preserve all 8 bits of domain names they process. When a name server is given data for the same name under two different case usages, this preservation is not always possible. For example, if a name server is given data for ISI.ARPA and isi.arpa, it should create a single node, not two, and hence will preserve a single casing of the label. Systems with case sensitivity should take special precautions to insure that the domain data for the system is created with consistent case.

In order to reduce the amount of space used by repetitive domain names, the sequence of octets that defines a domain name may be terminated by a pointer to the length octet of a previously specified label string. The label string that the pointer

specifies is appended to the already specified label string. Exact duplication of a previous label string can be done with a single pointer. Multiple levels are allowed.

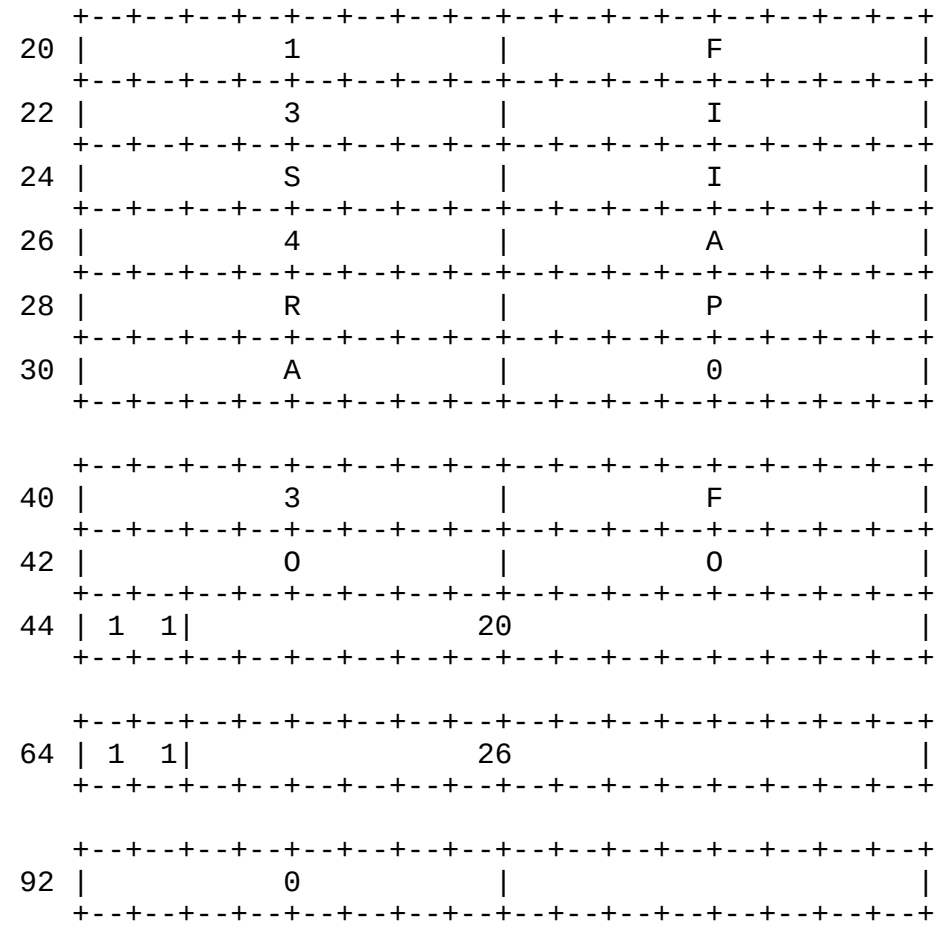
Pointers can only be used in positions in the message where the format is not class specific. If this were not the case, a name server that was handling a RR for another class could make erroneous copies of RRs. As yet, there are no such cases, but they may occur in future RDATA formats.

If a domain name is contained in a part of the message subject to a length field (such as the RDATA section of an RR), and compression is used, the length of the compressed name is used in the length calculation, rather than the length of the expanded name.

Pointers are represented as a two octet field in which the high order 2 bits are ones, and the low order 14 bits specify an offset from the start of the message. The 01 and 10 values of the high order bits are reserved for future use and should not be used.

Programs are free to avoid using pointers in datagrams they generate, although this will reduce datagram capacity. However all programs are required to understand arriving messages that contain pointers.

For example, a datagram might need to use the domain names F.ISI.ARPA, FOO.F.ISI.ARPA, ARPA, and the root. Ignoring the other fields of the message, these domain names might be represented as:



The domain name for F.ISI.ARPA is shown at offset 20. The domain name FOO.F.ISI.ARPA is shown at offset 40; this definition uses a pointer to concatenate a label for FOO to the previously defined F.ISI.ARPA. The domain name ARPA is defined at offset 64 using a pointer to the ARPA component of the name F.ISI.ARPA at 20; note that this reference relies on ARPA being the last label in the string at 20. The root domain name is defined by a single octet of zeros at 92; the root domain name has no labels.

Organization of the Shared database

While name server implementations are free to use any internal data structures they choose, the suggested structure consists of several separate trees. Each tree has structure corresponding to the domain name space, with RRs attached to nodes and leaves. Each zone of authoritative data has a separate tree, and one tree holds all non-authoritative data. All of the trees corresponding to zones are managed identically, but the non-authoritative or cache tree has different management procedures.

Data stored in the database can be kept in whatever form is convenient for the name server, so long as it can be transformed back into the format needed for messages. In particular, the database will probably use structure in place of expanded domain names, and will also convert many of the time intervals used in the domain systems to absolute local times.

Each tree corresponding to a zone has complete information for a "pruned" subtree of the domain space. The top node of a zone has a SOA record that marks the start of the zone. The bottom edge of the zone is delimited by nodes containing NS records signifying delegation of authority to other zones, or by leaves of the domain tree. When a name server contains abutting zones, one tree will have a bottom node containing a NS record, and the other tree will begin with a tree location containing a SOA record.

Note that there is one special case that requires consideration when a name server is implemented. A node that contains a SOA RR denoting a start of zone will also have NS records that identify the name servers that are expected to have a copy of the zone. Thus a name server will usually find itself (and possibly other redundant name servers) referred to in NS records occupying the same position in the tree as SOA records. The solution to this problem is to never interpret a NS record as delimiting a zone started by a SOA at the same point in the tree. (The sample programs in this memo deal with this problem by processing SOA records only after NS records have been processed.)

Zones may also overlap a particular part of the name space when they are of different classes.

Other than the abutting and separate class cases, trees are always expected to be disjoint. Overlapping zones are regarded as a non-fatal error. The scheme described in this memo avoids the overlap issue by maintaining separate trees; other designs must take the appropriate measures to defend against possible overlap.

Non-authoritative data is maintained in a separate tree. This tree is unlike the zone trees in that it may have "holes". Each RR in the cache tree has its own TTL that is separately managed. The data in this tree is never used if authoritative data is available from a zone tree; this avoids potential problems due to cached data that conflicts with authoritative data.

The shared database will also contain data structures to support the processing of inverse queries and completion queries if the local system supports these optional features. Although many schemes are possible, this memo describes a scheme that is based on tables of pointers that invert the database according to key.

Each kind of retrieval has a separate set of tables, with one table per zone. When a zone is updated, these tables must also be updated. The contents of these tables are discussed in the "Inverse query processing" and "Completion query processing" sections of this memo.

The database implementation described here includes two locks that are used to control concurrent access and modification of the database by name server query processing, name server maintenance operations, and resolver access:

The first lock ("main lock") controls access to all of the trees. Multiple concurrent reads are allowed, but write access can only be acquired by a single process. Read and write access are mutually exclusive. Resolvers and name server processes that answer queries acquire this lock in read mode, and unlock upon completion of the current message. This lock is acquired in write mode by a name server maintenance process when it is about to change data in the shared database. The actual update procedures are described under "NAME SERVER MAINTENANCE" but are designed to be brief.

The second lock ("cache queue lock") controls access to the cache queue. This queue is used by a resolver that wishes to add information to the cache tree. The resolver acquires this lock, then places the RRs to be cached into the queue. The name server maintenance procedure periodically acquires this lock and adds the queue information to the cache. The rationale for this procedure is that it allows the resolver to operate with read-only access to the shared database, and allows the update process to batch cache additions and the associated costs for inversion calculations. The name server maintenance procedure must take appropriate precautions to avoid problems with data already in the cache, inversions, etc.

This organization solves several difficulties:

When searching the domain space for the answer to a query, a name server can restrict its search for authoritative data to that tree that matches the most labels on the right side of the domain name of interest.

Since updates to a zone must be atomic with respect to searches, maintenance operations can simply acquire the main lock, insert a new copy of a particular zone without disturbing other zones, and then release the storage used by the old copy. Assuming a central table pointing to valid zone trees, this operation can be a simple pointer swap.

TTL management of zones can be performed using the SOA record for the zone. This avoids potential difficulties if individual RRs in a zone could be timed out separately. This issue is discussed further in the maintenance section.

Query processing

The following algorithm outlines processing that takes place at a name server when a query arrives:

1. Search the list of zones to find zones which have the same class as the QCLASS field in the query and have a top domain name that matches the right end of the QNAME field. If there are none, go to step 2. If there are more than one, pick the zone that has the longest match and go to step 3.

2. Since the zone search failed, the only possible RRs are contained in the non-authoritative tree. Search the cache tree for the NS record that has the same class as the QCLASS field and the largest right end match for domain name. Add the NS record or records to the authority section of the response. If the cache tree has RRs that are pertinent to the question (domain names match, classes agree, not timed-out, and the type field is relevant to the QTYPE), copy these RRs into the answer section of the response. The name server may also search the cache queue. Go to step 4.

3. Since this zone is the best match, the zone in which QNAME resides is either this zone or a zone to which this zone will directly or indirectly delegate authority. Search down the tree looking for a NS RR or the node specified by QNAME.

   If the node exists and has no NS record, copy the relevant RRs to the answer section of the response and go to step 4.

   If a NS RR is found, either matching a part or all of QNAME, then QNAME is in a delegated zone outside of this zone. If so, copy the NS record or records into the authority section of the response, and search the remainder of the zone for an A type record corresponding to the NS reference. If the A record is found, add it to the additional section. Go to step 2.

   If the node is not found and a NS is not found, there is no such name; set the Name error bit in the response and exit.

4. When this step is reached, the answer and authority sections are complete. What remains is to complete the additional section. This procedure is only possible if the name server

knows the data formats implied by the class of records in the answer and authority sections. Hence this procedure is class dependent. Appendix 3 discusses this procedure for Internet class data.

While this algorithm deals with typical queries and databases, several additions are required that will depend on the database supported by the name server:

QCLASS=*

Special procedures are required when the QCLASS of the query is "*". If the database contains several classes of data, the query processing steps above are performed separately for each CLASS, and the results are merged into a single response. The name error condition is not meaningful for a QCLASS=* query. If the requestor wants this information, it must test each class independently.

If the database is limited to data of a particular class, this operation can be performed by simply reseting the authoritative bit in the response, and performing the query as if QCLASS was the class used in the database.

* labels in database RRs

Some zones will contain default RRs that use * to match in cases where the search fails for a particular domain name. If the database contains these records then a failure must be retried using * in place of one or more labels of the search key. The procedure is to replace labels from the left with "*"s looking for a match until either all labels have been replaced, or a match is found. Note that these records can never be the result of caching, so a name server can omit this processing for zones that don't contain RRs with * in labels, or can omit this processing entirely if * never appears in local authoritative data.

Inverse query processing

Name servers that support inverse queries can support these operations through exhaustive searches of their databases, but this becomes impractical as the size of the database increases. An alternative approach is to invert the database according to the search key.

For name servers that support multiple zones and a large amount of data, the recommended approach is separate inversions for each

zone. When a particular zone is changed during a refresh, only its inversions need to be redone.

Support for transfer of this type of inversion may be included in future versions of the domain system, but is not supported in this version.

Completion query processing

Completion query processing shares many of the same problems in data structure design as are found in inverse queries, but is different due to the expected high rate of use of top level labels (ie., ARPA, CSNET). A name server that wishes to be efficient in its use of memory may well choose to invert only occurrences of ARPA, etc. that are below the top level, and use a search for the rare case that top level labels are used to constrain a completion.

NAME SERVER MAINTENANCE

Introduction

Name servers perform maintenance operations on their databases to insure that the data they distribute is accurate and timely. The amount and complexity of the maintenance operations that a name server must perform are related to the size, change rate, and complexity of the database that the name server manages.

Maintenance operations are fundamentally different for authoritative and non-authoritative data. A name server actively attempts to insure the accuracy and timeliness of authoritative data by refreshing the data from master copies. Non-authoritative data is merely purged when its time-to-live expires; the name server does not attempt to refresh it.

Although the refreshing scheme is fairly simple to implement, it is somewhat less powerful than schemes used in other distributed database systems. In particular, an update to the master does not immediately update copies, and should be viewed as gradually percolating though the distributed database. This is adequate for the vast majority of applications. In situations where timliness is critical, the master name server can prohibit caching of copies or assign short timeouts to copies.

Conceptual model of maintenance operations

The vast majority of information in the domain system is derived from master files scattered among hosts that implement name servers; some name servers will have no master files, other name servers will have one or more master files. Each master file contains the master data for a single zone of authority rather than data for the whole domain name space. The administrator of a particular zone controls that zone by updating its master file.

Master files and zone copies from remote servers may include RRs that are outside of the zone of authority when a NS record delegates authority to a domain name that is a descendant of the domain name at which authority is delegated. These forward references are a problem because there is no reasonable method to guarantee that the A type records for the delegatee are available unless they can somehow be attached to the NS records.

For example, suppose the ARPA zone delegates authority at MIT.ARPA, and states that the name server is on AI.MIT.ARPA. If a resolver gets the NS record but not the A type record for AI.MIT.ARPA, it might try to ask the MIT name server for the address of AI.MIT.ARPA.

The solution is to allow type A records that are outside of the zone of authority to be copied with the zone. While these records won't be found in a search for the A type record itself, they can be protected by the zone refreshing system, and will be passed back whenever the name server passes back a referral to the corresponding NS record. If a query is received for the A record, the name server will pass back a referral to the name server with the A record in the additional section, rather than answer section.

The only exception to the use of master files is a small amount of data stored in boot files. Boot file data is used by name servers to provide enough resource records to allow zones to be imported from foreign servers (e.g. the address of the server), and to establish the name and address of root servers. Boot file records establish the initial contents of the cache tree, and hence can be overridden by later loads of authoritative data.

The data in a master file first becomes available to users of the domain name system when it is loaded by the corresponding name server. By definition, data from a master file is authoritative.

Other name servers which wish to be authoritative for a particular zone do so by transferring a copy of the zone from the name server which holds the master copy using a virtual circuit. These copies include parameters which specify the conditions under which the data in the copy is authoritative. In the most common case, the conditions specify a refresh interval and policies to be followed when the refresh operation cannot be performed.

A name server may acquire multiple zones from different name servers and master files, but the name server must maintain each zone separately from others and from non-authoritative data.

When the refresh interval for a particular zone copy expires, the name server holding the copy must consult the name server that holds the master copy. If the data in the zone has not changed, the master name server instructs the copy name server to reset the refresh interval. If the data has changed, the master passes a new copy of the zone and its associated conditions to the copy name server. Following either of these transactions, the copy name server begins a new refresh interval.

Copy name servers must also deal with error conditions under which they are unable to communicate with the name server that holds the master copy of a particular zone. The policies that a copy name server uses are determined by other parameters in the conditions distributed with every copy. The conditions include a retry interval and a maximum holding time. When a copy name server is

unable to establish communications with a master or is unable to complete the refresh transaction, it must retry the refresh operation at the rate specified by the retry interval. This retry interval will usually be substantially shorter than the refresh interval. Retries continue until the maximum holding time is reached. At that time the copy name server must assume that its copy of the data for the zone in question is no longer authoritative.

Queries must be processed while maintenance operations are in progress because a zone transfer can take a long time. However, to avoid problems caused by access to partial databases, the maintenance operations create new copies of data rather than directly modifying the old copies. When the new copy is complete, the maintenance process locks out queries for a short time using the main lock, and switches pointers to replace the old data with the new. After the pointers are swapped, the maintenance process unlocks the main lock and reclaims the storage used by the old copy.

Name server data structures and top level logic

The name server must multiplex its attention between multiple activities. For example, a name server should be able to answer queries while it is also performing refresh activities for a particular zone. While it is possible to design a name server that devotes a separate process to each query and refresh activity in progress, the model described in this memo is based on the assumption that there is a single process performing all maintenance operations, and one or more processes devoted to handling queries. The model also assumes the existence of shared memory for several control structures, the domain database, locks, etc.

The model name server uses the following files and shared data structures:

1. A configuration file that describes the master and boot files which the name server should load and the zones that the name server should attempt to load from foreign name servers. This file establishes the initial contents of the status table.

2. Domain data files that contain master and boot data to be loaded.

3. A status table that is derived from the configuration file. Each entry in this table describes a source of data. Each entry has a zone number. The zone number is zero for

non-authoritative sources; authoritative sources are assigned separate non-zero numbers.

4. The shared database that holds the domain data. This database is assumed to be organized in some sort of tree structure paralleling the domain name space, with a list of resource records attached to each node and leaf in the tree. The elements of the resource record list need not contain the exact data present in the corresponding output format, but must contain data sufficient to create the output format; for example, these records need not contain the domain name that is associated with the resource because that name can be derived from the tree structure. Each resource record also internal data that the name server uses to organize its data.

5. Inversion data structures that allow the name server to process inverse queries and completion queries. Although many structures could be used, the implementation described in this memo supposes that there is one array for every inversion that the name server can handle. Each array contains a list of pointers to resource records such that the order of the inverted quantities is sorted.

6. The main and cache queue locks

7. The cache queue

The maintenance process begins by loading the status table from the configuration file. It then periodically checks each entry, to see if its refresh interval has elapsed. If not, it goes on to the next entry. If so, it performs different operations depending on the entry:

If the entry is for zone 0, or the cache tree, the maintenance process checks to see if additions or deletions are required. Additions are acquired from the cache queue using the cache queue lock. Deletions are detected using TTL checks. If any changes are required, the maintenance process recalculates inversion data structures and then alters the cache tree under the protection of the main lock. Whenever the maintenance process modifies the cache tree, it resets the refresh interval to the minimum of the contained TTLs and the desired time interval for cache additions.

If the entry is not zone 0, and the entry refers to a local file, the maintenance process checks to see if the file has been modified since its last load. If so the file is reloaded using the procedures specified under "Name server file

loading". The refresh interval is reset to that specified in the SOA record if the file is a master file.

If the entry is for a remote master file, the maintenance process checks for a new version using the procedure described in "Names server remote zone transfer".

Name server file loading

Master files are kept in text form for ease of editing by system maintainers. These files are not exchanged by name servers; name servers use the standard message format when transferring zones.

Organizations that want to have a domain, but do not want to run a name server, can use these files to supply a domain definition to another organization that will run a name server for them. For example, if organization X wants a domain but not a name server, it can find another organization, Y, that has a name server and is willing to provide service for X. Organization X defines domain X via the master file format and ships a copy of the master file to organization Y via mail, FTP, or some other method. A system administrator at Y configures Y's name server to read in X's file and hence support the X domain. X can maintain the master file using a text editor and send new versions to Y for installation.

These files have a simple line-oriented format, with one RR per line. Fields are separated by any combination of blanks and tab characters. Tabs are treated the same as spaces; in the following discussion the term "blank" means either a tab or a blank. A line can be either blank (and ignored), a RR, or a $INCLUDE line.

If a RR line starts with a domain name, that domain name is used to specify the location in the domain space for the record, i.e. the owner. If a RR line starts with a blank, it is loaded into the location specified by the most recent location specifier.

The location specifiers are assumed to be relative to some origin that is provided by the user of a file unless the location specifier contains the root label. This provides a convenient shorthand notation, and can also be used to prevent errors in master files from propagating into other zones. This feature is particularly useful for master files imported from other sites.

An include line begins with $INCLUDE, starting at the first line position, and is followed by a local file name and an optional offset modifier. The filename follows the appropriate local conventions. The offset is one or more labels that are added to the offset in use for the file that contained the $INCLUDE. If the offset is omitted, the included file is loaded using the

offset of the file that contained the $INCLUDE command. For example, a file being loaded at offset ARPA might contain the following lines:

```
$INCLUDE <subsys>isi.data ISI
$INCLUDE <subsys>addresses.data
```

The first line would be interpreted to direct loading of the file <subsys>isi.data at offset ISI.ARPA. The second line would be interpreted as a request to load data at offset ARPA.

Note that $INCLUDE commands do not cause data to be loaded into a different zone or tree; they are simply ways to allow data for a given zone to be organized in separate files. For example, mailbox data might be kept separately from host data using this mechanism.

Resource records are entered as a sequence of fields corresponding to the owner name, TTL, CLASS, TYPE and RDATA components. (Note that this order is different from the order used in examples and the order used in the actual RRs; the given order allows easier parsing and defaulting.)

The owner name is derived from the location specifier.

The TTL field is optional, and is expressed as a decimal number. If omitted TTL defaults to zero.

The CLASS field is also optional; if omitted the CLASS defaults to the most recent value of the CLASS field in a previous RR.

The RDATA fields depend on the CLASS and TYPE of the RR. In general, the fields that make up RDATA are expressed as decimal numbers or as domain names. Some exceptions exist, and are documented in the RDATA definitions in Appendicies 2 and 3 of this memo.

Because CLASS and TYPE fields don’t contain any common identifiers, and because CLASS and TYPE fields are never decimal numbers, the parse is always unique.

Because these files are text files several special encodings are necessary to allow arbitrary data to be loaded. In particular:

. A free standing dot is used to refer to the current domain name.

@ A free standing @ is used to denote the current origin.

.. Two free standing dots represent the null domain name of the root.

\X where X is any character other than a digit (0-9), is used to quote that character so that its special meaning does not apply. For example, "\." can be used to place a dot character in a label.

\DDD where each D is a digit is the octet corresponding to the decimal number described by DDD. The resulting octet is assumed to be text and is not checked for special meaning.

( ) Parentheses are used to group data that crosses a line boundary. In effect, line terminations are not recognized within parentheses.

; Semicolon is used to start a comment; the remainder of the line is ignored.

Name server file loading example

A name server for F.ISI.ARPA , serving as an authority for the ARPA and ISI.ARPA domains, might use a boot file and two master files. The boot file initializes some non-authoritative data, and would be loaded without an origin:

```
..              9999999 IN      NS      B.ISI.ARPA
                9999999 CS      NS      UDEL.CSNET
B.ISI.ARPA      9999999 IN      A       10.3.0.52
UDEL.CSNET      9999999 CS      A       302-555-0000
```

This file loads non-authoritative data which provides the identities and addresses of root name servers. The first line contains a NS RR which is loaded at the root; the second line starts with a blank, and is loaded at the most recent location specifier, in this case the root; the third and fourth lines load RRs at B.ISI.ARPA and UDEL.CSNET, respectively. The timeouts are set to high values (9999999) to prevent this data from being discarded due to timeout.

The first master file loads authoritative data for the ARPA domain. This file is designed to be loaded with an origin of ARPA, which allows the location specifiers to omit the trailing .ARPA labels.

```
    @   IN  SOA     F.ISI.ARPA       Action.E.ISI.ARPA (
                                     20     ; SERIAL
                                     3600   ; REFRESH
                                     600    ; RETRY
                                     3600000; EXPIRE
                                     60)    ; MINIMUM
            NS      F.ISI.ARPA ; F.ISI.ARPA is a name server for ARPA
            NS      A.ISI.ARPA ; A.ISI.ARPA is a name server for ARPA
    MIT     NS      AI.MIT.ARPA; delegation to MIT name server
    ISI     NS      F.ISI.ARPA ; delegation to ISI name server

    UDEL    MD      UDEL.ARPA
            A       10.0.0.96
    NBS     MD      NBS.ARPA
            A       10.0.0.19
    DTI     MD      DTI.ARPA
            A       10.0.0.12

    AI.MIT  A       10.2.0.6
    F.ISI   A       10.2.0.52
```

The first group of lines contains the SOA record and its parameters, and identifies name servers for this zone and for delegated zones. The Action.E.ISI.ARPA field is a mailbox specification for the responsible person for the zone, and is the domain name encoding of the mail destination Action@E.ISI.ARPA. The second group specifies data for domain names within this zone. The last group has forward references for name server address resolution for AI.MIT.ARPA and F.ISI.ARPA. This data is not technically within the zone, and will only be used for additional record resolution for NS records used in referrals. However, this data is protected by the zone timeouts in the SOA, so it will persist as long as the NS references persist.

The second master file defines the ISI.ARPA environment, and is loaded with an origin of ISI.ARPA:

```
    @   IN  SOA     F.ISI.ARPA      Action\.ISI.E.ISI.ARPA (
                                     20     ; SERIAL
                                     7200   ; REFRESH
                                     600    ; RETRY
                                     3600000; EXPIRE
                                     60)    ; MINIMUM
            NS      F.ISI.ARPA ; F.ISI.ARPA is a name server
    A       A       10.1.0.32
            MD      A.ISI.ARPA
            MF      F.ISI.ARPA
    B       A       10.3.0.52
            MD      B.ISI.ARPA
```

```
          MF      F.ISI.ARPA
    F     A       10.2.0.52
          MD      F.ISI.ARPA
          MF      A.ISI.ARPA
    $INCLUDE <SUBSYS>ISI-MAILBOXES.TXT
```

Where the file <SUBSYS>ISI-MAILBOXES.TXT is:

```
    MOE     MB      F.ISI.ARPA
    LARRY   MB      A.ISI.ARPA
    CURLEY  MB      B.ISI.ARPA
    STOOGES MB      B.ISI.ARPA
            MG      MOE.ISI.ARPA
            MG      LARRY.ISI.ARPA
            MG      CURLEY.ISI.ARPA
```

Note the use of the \ character in the SOA RR to specify the responsible person mailbox "Action.ISI@E.ISI.ARPA".

Name server remote zone transfer

When a name server needs to make an initial copy of a zone or test to see if a existing zone copy should be refreshed, it begins by attempting to open a virtual circuit to the foreign name server.

If this open attempt fails, and this was an initial load attempt, it schedules a retry and exits. If this was a refresh operation, the name server tests the status table to see if the maximum holding time derived from the SOA EXPIRE field has elapsed. If not, the name server schedules a retry. If the maximum holding time has expired, the name server invalidates the zone in the status table, and scans all resource records tagged with this zone number. For each record it decrements TTL fields by the length of time since the data was last refreshed. If the new TTL value is negative, the record is deleted. If the TTL value is still positive, it moves the RR to the cache tree and schedules a retry.

If the open attempt succeeds, the name server sends a query to the foreign name server in which QTYPE=SOA, QCLASS is set according to the status table information from the configuration file, and QNAME is set to the domain name of the zone of interest.

The foreign name server will return either a SOA record indicating that it has the zone or an error. If an error is detected, the virtual circuit is closed, and the failure is treated in the same way as if the open attempt failed.

If the SOA record is returned and this was a refresh, rather than an initial load of the zone, the name server compares the SERIAL

field in the new SOA record with the SERIAL field in the SOA record of the existing zone copy. If these values match, the zone has not been updated since the last copy and hence there is no reason to recopy the zone. In this case the name server resets the times in the existing SOA record and closes the virtual circuit to complete the operation.

If this is initial load, or the SERIAL fields were different, the name server requests a copy of the zone by sending the foreign name server an AXFR query which specifies the zone by its QCLASS and QNAME fields.

When the foreign name server receives the AXFR request, it sends each node from the zone to the requestor in a separate message. It begins with the node that contains the SOA record, walks the tree in breadth-first order, and completes the transfer by resending the node containing the SOA record.

Several error conditions are possible:

If the AXFR request cannot be matched to a SOA, the foreign name server will return a single message in response that does not contain the AXFR request. (The normal SOA query preceding the AXFR is designed to avoid this condition, but it is still possible.)

The foreign name server can detect an internal error or detect some other condition (e.g. system going down, out of resources, etc.) that forces the transfer to be aborted. If so, it sends a message with the "Server failure" condition set. If the AXFR can be immediately retried with some chance of success, it leaves the virtual open; otherwise it initiates a close.

If the foreign name server doesn’t wish to perform the operation for policy reasons (i.e. the system administrator wishes to forbid zone copies), the foreign server returns a "Refused" condition.

The requestor receives these records and builds a new tree. This tree is not yet in the status table, so its data are not used to process queries. The old copy of the zone, if any, may be used to satisfy request while the transfer is in progress.

When the requestor receives the second copy of the SOA node, it compares the SERIAL field in the first copy of the SOA against the SERIAL field in the last copy of the SOA record. If these don’t match, the foreign server updated its zone while the transfer was in progress. In this case the requestor repeats the AXFR request to acquire the newer version.

If the AXFR transfer eventually succeeds, the name server closes the virtual circuit and and creates new versions of inversion data structures for this zone. When this operation is complete, the name server acquires the main lock in write mode and then replaces any old copy of the zone and inversion data structures with new ones. The name server then releases the main lock, and can reclaim the storage used by the old copy.

If an error occurs during the AXFR transfer, the name server can copy any partial information into its cache tree if it wishes, although it will not normally do so if the zone transfer was a refresh rather than an initial load.

RESOLVER ALGORITHMS

Operations

Resolvers have a great deal of latitude in the semantics they allow in user calls. For example, a resolver might support different user calls that specify whether the returned information must be from and authoritative name server or not. Resolvers are also responsible for enforcement of any local restrictions on access, etc.

In any case, the resolver will transform the user query into a number of shared database accesses and queries to remote name servers. When a user requests a resource associated with a particular domain name, the resolver will execute the following steps:

1. The resolver first checks the local shared database, if any, for the desired information. If found, it checks the applicable timeout. If the timeout check succeeds, the information is used to satisfy the user request. If not, the resolver goes to step 2.

2. In this step, the resolver consults the shared database for the name server that most closely matches the domain name in the user query. Multiple redundant name servers may be found. The resolver goes to step 3.

3. In this step the resolver chooses one of the available name servers and sends off a query. If the query fails, it tries another name server. If all fail, an error indication is returned to the user. If a reply is received the resolver adds the returned RRs to its database and goes to step 4.

4. In this step, the resolver interprets the reply. If the reply contains the desired information, the resolver returns the information to the user. The the reply indicates that the domain name in the user query doesn’t exist, then the resolver returns an error to the user. If the reply contains a transient name server failure, the resolver can either wait and retry the query or go back to step 3 and try a different name server. If the reply doesn’t contain the desired information, but does contain a pointer to a closer name server, the resolver returns to step 2, where the closer name servers will be queried.

Several modifications to this algorithm are possible. A resolver may not support a local cache and instead only cache information during the course of a single user request, discarding it upon

completion. The resolver may also find that a datagram reply was truncated, and open a virtual circuit so that the complete reply can be recovered.

Inverse and completion queries must be treated in an environment-sensitive manner, because the domain system doesn't provide a method for guaranteeing that it can locate the correct information. The typical choice will be to configure a resolver to use a particular set of known name servers for inverse queries.

DOMAIN SUPPORT FOR MAIL

Introduction

Mail service is a particularly sensitive issue for users of the domain system because of the lack of a consistent system for naming mailboxes and even hosts, and the need to support continued operation of existing services. This section discusses an evolutionary approach for adding consistent domain name support for mail.

The crucial issue is deciding on the types of binding to be supported. Most mail systems specify a mail destination with a two part construct such as X@Y. The left hand side, X, is an string, often a user or account, and Y is a string, often a host. This section refers to the part on the left, i.e. X, as the local part, and refers to the part on the right, i.e. Y, as the global part.

Most existing mail systems route mail based on the global part; a mailer with mail to deliver to X@Y will decide on the host to be contacted using only Y. We refer to this type of binding as "agent binding".

> For example, mail addressed to Mockapetris@ISIF is delivered to host USC-ISIF (USC-ISIF is the official name for the host specified by nickname ISIF).

More sophisticated mail systems use both the local and global parts, i.e. both X and Y to determine which host should receive the mail. These more sophisticated systems usually separate the binding of the destination to the host from the actual delivery. This allows the global part to be a generic name rather than constraining it to a single host. We refer to this type of binding as "mailbox binding".

> For example, mail addressed to Mockapetris@ISI might be bound to host F.ISI.ARPA, and subsequently delivered to that host, while mail for Cohen@ISI might be bound to host B.ISI.ARPA.

The domain support for mail consists of two levels of support, corresponding to these two binding models.

> The first level, agent binding, is compatible with existing ARPA Internet mail procedures and uses maps a global part onto one or more hosts that will accept the mail. This type of binding uses the MAILA QTYPE.
>
> The second level, mailbox binding, offers extended services

that map a local part and a global part onto one or more sets of data via the MAILB QTYPE. The sets of data include hosts that will accept the mail, mailing list members (mail groups), and mailboxes for reporting errors or requests to change a mail group.

The domain system encodes the global part of a mail destination as a domain name and uses dots in the global part to separate labels in the encoded domain name. The domain system encodes the local part of a mail destination as a single label, and any dots in this part are simply copied into the label. The domain system forms a complete mail destination as the local label concatenated to the domain string for the global part. We call this a mailbox.

For example, the mailbox Mockapetris@F.ISI.ARPA has a global domain name of three labels, F.ISI.ARPA. The domain name encoding for the whole mailbox is Mockapetris.F.ISI.ARPA. The mailbox Mockapetris.cad@F.ISI.ARPA has the same domain name for the global part and a 4 label domain name for the mailbox of Mockapetris\.cad.F.ISI.ARPA (the \ is not stored in the label, its merely used to denote the "quoted" dot).

It is anticipated that the Internet system will adopt agent binding as part of the initial implementation of the domain system, and that mailbox binding will eventually become the preferred style as organizations convert their mail systems to the new style. To facilitate this approach, the domain information for these two binding styles is organized to allow a requestor to determine which types of support are available, and the information is kept in two disjoint classes.

Agent binding

In agent binding, a mail system uses the global part of the mail destination as a domain name, with dots denoting structure. The domain name is resolved using a MAILA query which return MF and MD RRs to specify the domain name of the appropriate host to receive the mail. MD (Mail delivery) RRs specify hosts that are expected to have the mailbox in question; MF (Mail forwarding) RRs specify hosts that are expected to be intermediaries willing to accept the mail for eventual forwarding. The hosts are hints, rather than definite answers, since the query is made without the full mail destination specification.

For example, mail for MOCKAPETRIS@F.ISI.ARPA would result in a query with QTYPE=MAILA and QNAME=F.ISI.ARPA, which might return two RRs:

```
F.ISI.ARPA MD IN F.ISI.ARPA
F.ISI.ARPA MF IN A.ISI.ARPA
```

The mailer would interpret these to mean that the mail agent on F.ISI.ARPA should be able to deliver the mail directly, but that A.ISI.ARPA is willing to accept the mail for probable forwarding.

Using this system, an organization could implement a system that uses organization names for global parts, rather than the usual host names, but all mail for the organization would be routed the same, regardless of its local part. Hence and organization with many hosts would expect to see many forwarding operations.

Mailbox binding

In mailbox binding, the mailer uses the entire mail destination specification to construct a domain name. The encoded domain name for the mailbox is used as the QNAME field in a QTYPE=MAILB query.

Several outcomes are possible for this query:

1. The query can return a name error indicating that the mailbox does not exist as a domain name.

   In the long term this would indicate that the specified mailbox doesn't exist. However, until the use of mailbox binding is universal, this error condition should be interpreted to mean that the organization identified by the global part does not support mailbox binding. The appropriate procedure is to revert to agent binding at this point.

2. The query can return a Mail Rename (MR) RR.

   The MR RR carries new mailbox specification in its RDATA field. The mailer should replace the old mailbox with the new one and retry the operation.

3. The query can return a MB RR.

   The MB RR carries a domain name for a host in its RDATA field. The mailer should deliver the message to that host via whatever protocol is applicable, e.g. SMTP.

4. The query can return one or more Mail Group (MG) RRs.

   This condition means that the mailbox was actually a mailing list or mail group, rather than a single mailbox. Each MG RR has a RDATA field that identifies a mailbox that is a member of

the group. The mailer should deliver a copy of the message to each member.

5. The query can return a MB RR as well as one or more MG RRs.

   This condition means the the mailbox was actually a mailing list. The mailer can either deliver the message to the host specified by the MB RR, which will in turn do the delivery to all members, or the mailer can use the MG RRs to do the expansion itself.

In any of these cases, the response may include a Mail Information (MINFO) RR. This RR is usually associated with a mail group, but is legal with a MB. The MINFO RR identifies two mailboxes. One of these identifies a responsible person for the original mailbox name. This mailbox should be used for requests to be added to a mail group, etc. The second mailbox name in the MINFO RR identifies a mailbox that should receive error messages for mail failures. This is particularly appropriate for mailing lists when errors in member names should be reported to a person other than the one who sends a message to the list. New fields may be added to this RR in the future.

Appendix 1 - Domain Name Syntax Specification

The preferred syntax of domain names is given by the following BNF rules. Adherence to this syntax will result in fewer problems with many applications that use domain names (e.g., mail, TELNET). Note that some applications use domain names containing binary information and hence do not follow this syntax.

```
<domain> ::=  <subdomain> | " "

<subdomain> ::=  <label> | <subdomain> "." <label>

<label> ::= <letter> [ [ <ldh-str> ] <let-dig> ]

<ldh-str> ::= <let-dig-hyp> | <let-dig-hyp> <ldh-str>

<let-dig-hyp> ::= <let-dig> | "-"

<let-dig> ::= <letter> | <digit>

<letter> ::= any one of the 52 alphabetic characters A through Z
in upper case and a through z in lower case

<digit> ::= any one of the ten digits 0 through 9
```

Note that while upper and lower case letters are allowed in domain names no significance is attached to the case. That is, two names with the same spelling but different case are to be treated as if identical.

The labels must follow the rules for ARPANET host names. They must start with a letter, end with a letter or digit, and have as interior characters only letters, digits, and hyphen. There are also some restrictions on the length. Labels must be 63 characters or less.

For example, the following strings identify hosts in the ARPA Internet:

```
F.ISI.ARPA     LINKABIT-DCN5.ARPA     UCL-TAC.ARPA
```

Appendix 2 - Field formats and encodings

```
+-----------------------------------------------+
|                                               |
|             *****  WARNING  *****             |
|                                               |
|  The following formats are preliminary and    |
| are included for purposes of explanation only.|
| In particular, new RR types will be added,    |
| and the size, position, and encoding of       |
| fields are subject to change.                 |
|                                               |
+-----------------------------------------------+
```

TYPE values

TYPE fields are used in resource records. Note that these types are not the same as the QTYPE fields used in queries, although the functions are often similar.

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

QTYPE values

QTYPE fields appear in the question part of a query. They include the values of TYPE with the following additions:

AXFR	252	A request for a transfer of an entire zone of authority
MAILB	253	A request for mailbox-related records (MB, MG or MR)
MAILA	254	A request for mail agent RRs (MD and MF)
*	255	A request for all records

CLASS values

CLASS fields appear in resource records

CLASS value	meaning	
IN	1	the ARPA Internet
CS	2	the computer science network (CSNET)

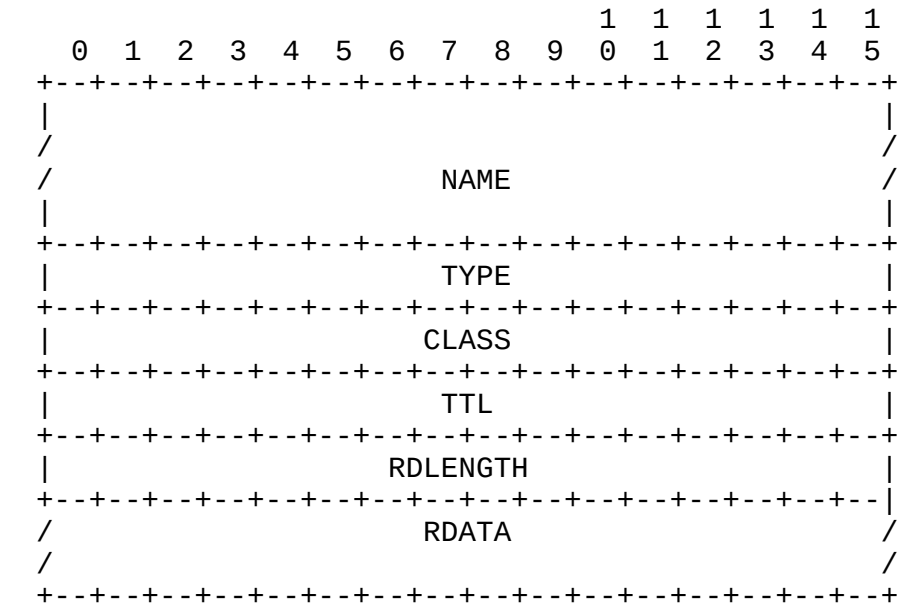
QCLASS values

QCLASS fields appear in the question section of a query. They include the values of CLASS with the following additions:

*	255 any class

Standard resource record formats

All RRs have the same top level format shown below:



where:

NAME - a compressed domain name to which this resource record pertains.

TYPE - two octets containing one of the RR type codes defined in Appendix 2. This field specifies the meaning of the data in the RDATA field.

CLASS - two octets which specifies the class of the data in the RDATA field.

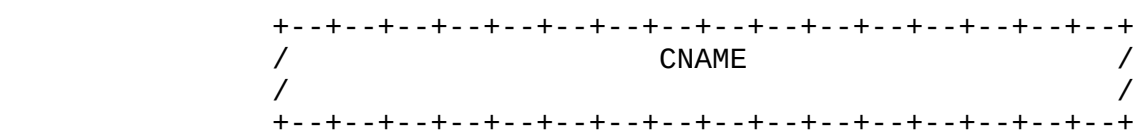
TTL - a 16 bit signed integer that specifies the time interval that the resource record may be cached before the source of the information should again be consulted. Zero values are interpreted to mean that the RR can only be used for the transaction in progress, and should not be cached. For example, SOA records are always distributed with a zero TTL to prohibit caching. Zero values can also be used for extremely volatile data.

RDLENGTH- an unsigned 16 bit integer that specifies the length in octets of the RDATA field.

RDATA - a variable length string of octets that describes the resource. The format of this information varies according to the TYPE and CLASS of the resource record.

The format of the RDATA field is standard for all classes for the RR types NS, MD, MF, CNAME, SOA, MB, MG, MR, PTR, HINFO, MINFO and NULL. These formats are shown below together with the appropriate additional section RR processing.

CNAME RDATA format

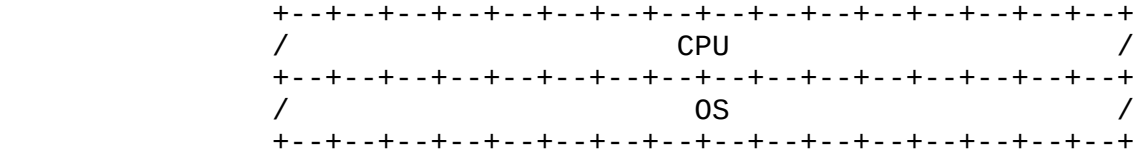


where:

CNAME - A compressed domain name which specifies that the domain name of the RR is an alias for a canonical name specified by CNAME.

CNAME records cause no additional section processing. The RDATA section of a CNAME line in a master file is a standard printed domain name.

HINFO RDATA format



where:

CPU - A character string which specifies the CPU type. The character string is represented as a single octet length followed by that number of characters. The following standard strings are defined:.

PDP-11/70	C/30	C/70	VAX-11/780
H-316	H-516	DEC-2060	DEC-1090T
ALTO	IBM-PC	IBM-PC/XT	PERQ
IBM-360/67	IBM-370/145		

OS - A character string which specifies the operating system type. The character string is represented as a single octet

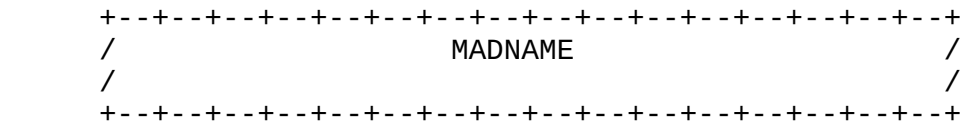
length followed by that number of characters. The following standard types are defined:.

ASP	AUGUST	BKY	CCP
DOS/360	ELF	EPOS	EXEC-8
GCOS	GPOS	ITS	INTERCOM
KRONOS	MCP	MOS	MPX-RT
MULTICS	MVT	NOS	NOS/BE
OS/MVS	OS/MVT	RIG	RSX11
RSX11M	RT11	SCOPE	SIGNAL
SINTRAN	TENEX	TOPS10	TOPS20
TSS	UNIX	VM/370	VM/CMS
VMS	WAITS		

HINFO records cause no additional section processing.

HINFO records are used to acquire general information about a host. The main use is for protocols such as FTP that can use special procedures when talking between machines or operating systems of the same type.

MB RDATA format

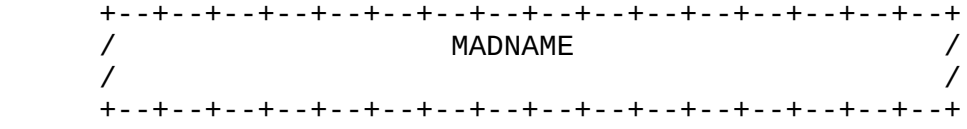


where:

MADNAME - A compressed domain name which specifies a host which has the specified mailbox.

MB records cause additional section processing which looks up an A type record corresponding to MADNAME. The RDATA section of a MB line in a master file is a standard printed domain name.

MD RDATA format



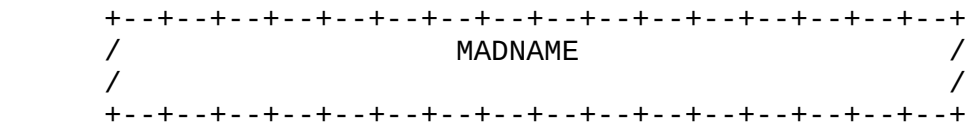
where:

MADNAME - A compressed domain name which specifies a host which

has a mail agent for the domain which should be able to deliver mail for the domain.

MD records cause additional section processing which looks up an A type record corresponding to MADNAME. The RDATA section of a MD line in a master file is a standard printed domain name.

MF RDATA format

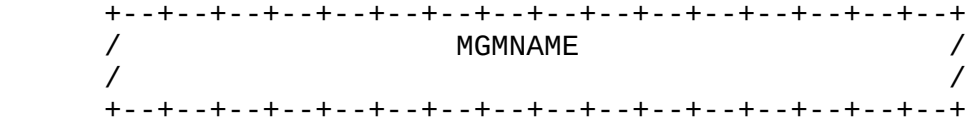


where:

MADNAME - A compressed domain name which specifies a host which has a mail agent for the domain which will accept mail for forwarding to the domain.

MF records cause additional section processing which looks up an A type record corresponding to MADNAME. The RDATA section of a MF line in a master file is a standard printed domain name.

MG RDATA format

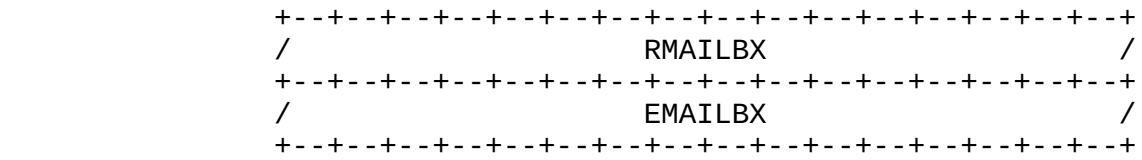


where:

MGMNAME - A compressed domain name which specifies a mailbox which is a member of the mail group specified by the domain name.

MF records cause no additional section processing. The RDATA section of a MF line in a master file is a standard printed domain name.

MINFO RDATA format



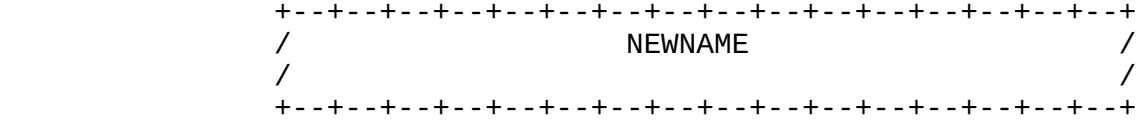
where:

RMAILBX - A compressed domain name which specifies a mailbox which is responsible for the mailing list or mailbox. If this domain name names the root, the owner of the MINFO RR is responsible for itself. Note that many existing mailing lists use a mailbox X-request for the RMAILBX field of mailing list X, e.g. Msgroup-request for Msgroup. This field provides a more general mechanism.

EMAILBX - A compressed domain name which specifies a mailbox which is to receive error messages related to the mailing list or mailbox specified by the owner of the MINFO RR (similar to the ERRORS-TO: field which has been proposed). If this domain name names the root, errors should be returned to the sender of the message.

MINFO records cause no additional section processing. Although these records can be associated with a simple mailbox, they are usually used with a mailing list. The MINFO section of a MF line in a master file is a standard printed domain name.

MR RDATA format

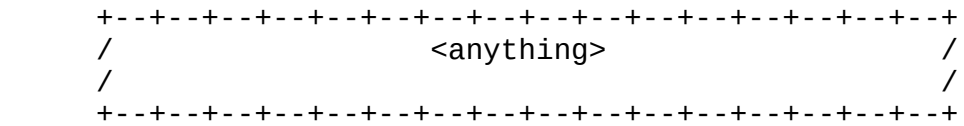


where:

NEWNAME - A compressed domain name which specifies a mailbox which is the proper rename of the specified mailbox.

MR records cause no additional section processing. The RDATA section of a MR line in a master file is a standard printed domain name.

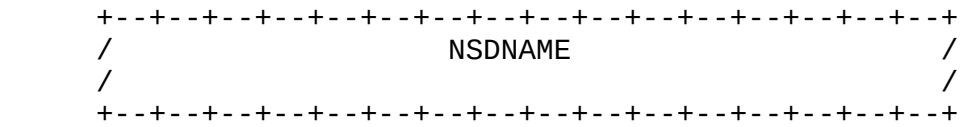
NULL RDATA format



Anything at all may be in the RDATA field so long as it is 65535 octets or less.

NULL records cause no additional section processing. NULL RRs are not allowed in master files.

NS RDATA format

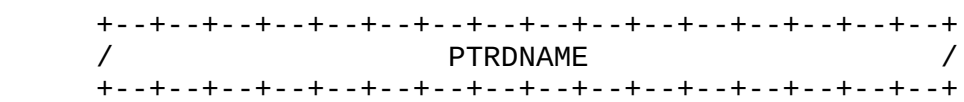


where:

NSDNAME - A compressed domain name which specifies a host which has a name server for the domain.

NS records cause both the usual additional section processing to locate a type A record, and a special search of the zone in which they reside. The RDATA section of a NS line in a master file is a standard printed domain name.

PTR RDATA format

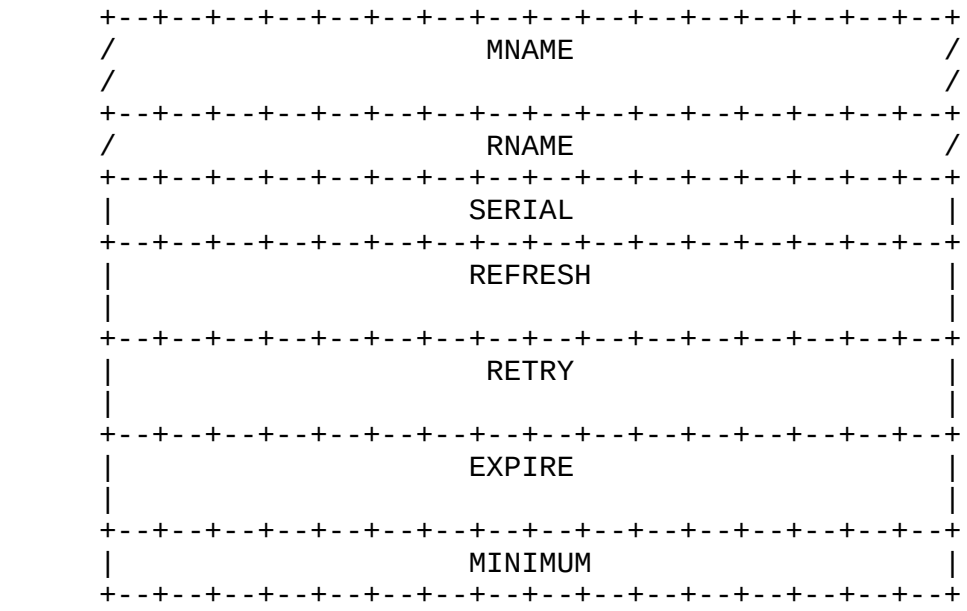


where:

PTRDNAME - A compressed domain name which points to some location in the domain name space.

PTR records cause no additional section processing. These RRs are used in special domains to point to some other location in the domain space. These records are simple data, and don't imply any special processing similar to that performed by CNAME, which identifies aliases. Appendix 3 discusses the use of these records in the ARPA Internet address domain.

SOA RDATA format



where:

MNAME - The domain name of the name server that was the original source of data for this zone.

RNAME - A domain name which specifies the mailbox of the person responsible for this zone.

SERIAL - The unsigned 16 bit version number of the of the original copy of the zone. This value wraps and should be compared using sequence space arithmetic.

REFRESH - The unsigned 32 bit time interval before the zone should be refreshed.

RETRY - The unsigned 32 bit time interval that should elapse before a failed refresh should be retried.

EXPIRE - A 32 bit time value that specifies the upper limit on the time interval that can elapse before the zone is no longer authoritative.

MINIMUM - The unsigned 16 bit minimum TTL field that should be exported with any RR from this zone (other than the SOA itself).

SOA records cause no additional section processing. The RDATA

section of a SOA line in a master file is a standard printed domain name for MNAME, a standard X@Y mailbox specification for RNAME, and decimal numbers for the remaining parameters.

All times are in units of seconds.

Most of these fields are pertinent only for name server maintenance operations. However, MINIMUM is used in all query operations that retrieve RRs from a zone. Whenever a RR is sent in a response to a query, the TTL field is set to the maximum of the TTL field from the RR and the MINIMUM field in the appropriate SOA. Thus MINIMUM is a lower bound on the TTL field for all RRs in a zone. RRs in a zone are never discarded due to timeout unless the whole zone is deleted. This prevents partial copies of zones.

Appendix 3 - Internet specific field formats and operations

Message transport

The Internet supports name server access using TCP [10] on server port 53 (decimal) as well as datagram access using UDP [11] on UDP port 53 (decimal). Messages sent over TCP virtual circuits are preceded by an unsigned 16 bit length field which describes the length of the message, excluding the length field itself.

```
+-----------------------------------------------+
|                                               |
|             *****  WARNING  *****             |
|                                               |
|  The following formats are preliminary and    |
| are included for purposes of explanation only.|
| In particular, new RR types will be added,    |
| and the size, position, and encoding of       |
| fields are subject to change.                 |
|                                               |
+-----------------------------------------------+
```

A RDATA format

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ADDRESS                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

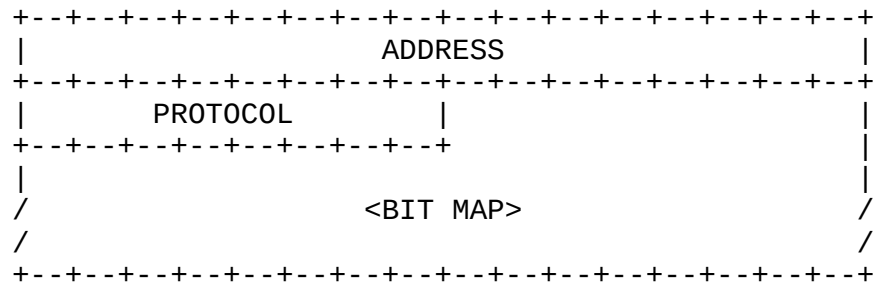
where:

ADDRESS - A 32 bit ARPA internet address

Hosts that have multiple ARPA Internet addresses will have multiple A records.

A records cause no additional section processing. The RDATA section of an A line in a master file is an Internet address expressed as four decimal numbers separated by dots without any imbedded spaces (e.g., "10.2.0.52" or "192.0.5.6").

WKS RDATA format



where:

ADDRESS - An 32 bit ARPA Internet address

PROTOCOL - An 8 bit IP protocol number

<BIT MAP> - A variable length bit map. The bit map must be a multiple of 8 bits long.

The WKS record is used to describe the well known services supported by a particular protocol on a particular internet address. The PROTOCOL field specifies an IP protocol number, and the bit map has one bit per port of the specified protocol. The first bit corresponds to port 0, the second to port 1, etc. If less than 256 bits are present, the remainder are assumed to be zero. The appropriate values for ports and protocols are specified in [13].

For example, if PROTOCOL=TCP (6), the 26th bit corresponds to TCP port 25 (SMTP). If this bit is set, a SMTP server should be listening on TCP port 25; if zero, SMTP service is not supported on the specified address.

The anticipated use of WKS RRs is to provide availability information for servers for TCP and UDP. If a server supports both TCP and UDP, or has multiple Internet addresses, then multiple WKS RRs are used.

WKS RRs cause no additional section processing. The RDATA section of a WKS record consists of a decimal protocol number followed by mnemonic identifiers which specify bits to be set to 1.

IN-ADDR special domain

The ARPA internet uses a special domain to support gateway location and ARPA Internet address to host mapping. The intent of this domain is to allow queries to locate all gateways on a

particular network in the ARPA Internet, and also to provide a guaranteed method to perform host address to host name mapping.

Note that both of these services are similar to functions that could be performed by inverse queries; the difference is that this part of the domain name space is structured according to address, and hence can guarantee that the appropriate data can be located without an exhaustive search of the domain space. It is anticipated that the special tree will be used by ARPA Internet resolvers for all gateway location services, but that address to name resolution will be performed by first trying the inverse query on the local name server database followed by a query in the special space if the inverse query fails.

The domain is a top level domain called IN-ADDR whose substructure follows the ARPA Internet addressing structure.

Domain names in the IN-ADDR domain are defined to have up to four labels in addition to the IN-ADDR label. Each label is a character string which expresses a decimal value in the range 0-255 (with leading zeros omitted except in the case of a zero octet which is represented by a single zero). These labels correspond to the 4 octets of an ARPA Internet address.

Host addresses are represented by domain names that have all four labels specified. Thus data for ARPA Internet address 10.2.0.52 is located at domain name 52.0.2.10.IN-ADDR. The reversal, though awkward to read, allows zones to follow the natural grouping of hosts within networks. For example, 10.IN-ADDR can be a zone containing data for the ARPANET, while 26.IN-ADDR can be a separate zone for MILNET. Address nodes are used to hold pointers to primary host names in the normal domain space.

Network addresses correspond to some of the non-terminal nodes in the IN-ADDR tree, since ARPA Internet network numbers are either 1, 2, or 3 octets. Network nodes are used to hold pointers to primary host names (which happen to be gateways) in the normal domain space. Since a gateway is, by definition, on more than one network, it will typically have two or more network nodes that point at the gateway. Gateways will also have host level pointers at their fully qualified addresses.

Both the gateway pointers at network nodes and the normal host pointers at full address nodes use the PTR RR to point back to the primary domain names of the corresponding hosts.

For example, part of the IN-ADDR domain will contain information about the ISI to MILNET and MIT gateways, and hosts F.ISI.ARPA and MULTICS.MIT.ARPA. Assuming that ISI gateway has addresses

10.2.0.22 and 26.0.0.103, and a name MILNET-GW.ISI.ARPA, and the MIT gateway has addresses 10.0.0.77 and 18.10.0.4 and a name GW.MIT.ARPA, the domain database would contain:

```
10.IN-ADDR           PTR  IN MILNET-GW.ISI.ARPA
10.IN-ADDR           PTR  IN GW.MIT.ARPA
18.IN-ADDR           PTR  IN GW.MIT.ARPA
26.IN-ADDR           PTR  IN MILNET-GW.ISI.ARPA
22.0.2.10.IN-ADDR    PTR  IN MILNET-GW.ISI.ARPA
103.0.0.26.IN-ADDR   PTR  IN MILNET-GW.ISI.ARPA
77.0.0.10.IN-ADDR    PTR  IN GW.MIT.ARPA
4.0.10.18.IN-ADDR    PTR  IN GW.MIT.ARPA
52.0.2.10.IN-ADDR    PTR  IN F.ISI.ARPA
6.0.0.10.IN-ADDR     PTR  IN MULTICS.MIT.ARPA
```

Thus a program which wanted to locate gateways on net 10 would originate a query of the form QTYPE=PTR, QCLASS=IN, QNAME=10.IN-ADDR. It would receive two RRs in response:

```
10.IN-ADDR           PTR  IN MILNET-GW.ISI.ARPA
10.IN-ADDR           PTR  IN GW.MIT.ARPA
```

The program could then originate QTYPE=A, QCLASS=IN queries for MILNET-GW.ISI.ARPA and GW.MIT.ARPA to discover the ARPA Internet addresses of these gateways.

A resolver which wanted to find the host name corresponding to ARPA Internet host address 10.0.0.6 might first try an inverse query on the local name server, but find that this information wasn't available. It could then try a query of the form QTYPE=PTR, QCLASS=IN, QNAME=6.0.0.10.IN-ADDR, and would receive:

```
6.0.0.10.IN-ADDR     PTR  IN MULTICS.MIT.ARPA
```

Several cautions apply to the use of these services:

- Since the IN-ADDR special domain and the normal domain for a particular host or gateway will be in different zones, the possibility exists that that the data may be inconsistent.

- Gateways will often have two names in separate domains, only one of which can be primary.

- Systems that use the domain database to initialize their routing tables must start with enough gateway information to guarantee that they can access the appropriate name server.

- The gateway data only reflects the existence of a gateway in a

manner equivalent to the current HOSTS.TXT file. It doesn't replace the dynamic availability information from GGP or EGP.

REFERENCES and BIBLIOGRAPHY

[1] E. Feinler, K. Harrenstien, Z. Su, and V. White, "DOD Internet Host Table Specification", RFC 810, Network Information Center, SRI International, March 1982.

[2] J. Postel, "Computer Mail Meeting Notes", RFC 805, USC/Information Sciences Institute, February 1982.

[3] Z. Su, and J. Postel, "The Domain Naming Convention for Internet User Applications", RFC 819, Network Information Center, SRI International, August 1982.

[4] Z. Su, "A Distributed System for Internet Name Service", RFC 830, Network Information Center, SRI International, October 1982.

[5] K. Harrenstien, and V. White, "NICNAME/WHOIS", RFC 812, Network Information Center, SRI International, March 1982.

[6] M. Solomon, L. Landweber, and D. Neuhengen, "The CSNET Name Server", Computer Networks, vol 6, nr 3, July 1982.

[7] K. Harrenstien, "NAME/FINGER", RFC 742, Network Information Center, SRI International, December 1977.

[8] J. Postel, "Internet Name Server", IEN 116, USC/Information Sciences Institute, August 1979.

[9] K. Harrenstien, V. White, and E. Feinler, "Hostnames Server", RFC 811, Network Information Center, SRI International, March 1982.

[10] J. Postel, "Transmission Control Protocol", RFC 793, USC/Information Sciences Institute, September 1981.

[11] J. Postel, "User Datagram Protocol", RFC 768, USC/Information Sciences Institute, August 1980.

[12] J. Postel, "Simple Mail Transfer Protocol", RFC 821, USC/Information Sciences Institute, August 1980.

[13] J. Reynolds, and J. Postel, "Assigned Numbers", RFC 870, USC/Information Sciences Institute, October 1983.

[14] P. Mockapetris, "Domain names - Concepts and Facilities," RFC 882, USC/Information Sciences Institute, November 1983.

INDEX