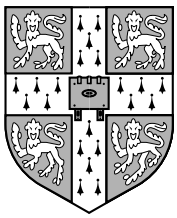
The Main Name System

Timothy John Deegan

Emmanuel College, University of Cambridge



This dissertation is submitted for the degree of
Doctor of Philosophy

April 2006

This dissertation is the author's own work and contains nothing which is the outcome of work done in collaboration with others, except where specifically indicated in the text. It is not substantially the same as any dissertation that the author has submitted or will be submitting for a degree, diploma or other qualification at any other University.

This dissertation does not exceed 60,000 words, including tables and footnotes.

The Main Name System

Timothy John Deegan

Emmanuel College, University of Cambridge

The Domain Name System (DNS) is a globally distributed nameservice for the internet. It uses delegation of areas of the namespace to spread both the administrative load and the service load. However, this delegation introduces complexity onto the operation of the service, which in turn brings opportunities for delay, failure and error.

In this dissertation, we present a *centralized* architecture for the DNS, which removes the link between administrative delegation and distribution of nameservice. By aggregating all DNS data at a few servers, it makes name lookups simpler, and removes the errors associated with inconsistency of data. This is done without requiring changes to the lookup protocol, the namespace, or the deployed client base. The cost of the simplified lookups is a more complex update mechanism; however, since nameservice updates are rarer and more tolerant of delay than lookups, this is an improvement.

We describe the current workload of the DNS, as a yardstick against which other nameservices might be measured. We then discuss how a centralized service could be built to handle the queries and updates of the current DNS, and describe and evaluate a datastructure and algorithms suitable for serving large numbers of DNS records.

Acknowledgments

This thesis started in a series of conversations with Steve Hand, Christopher Clark and Andy Warfield, and my supervisor, Jon Crowcroft. It became a real project over the summer of 2004, when I was visiting the LCN group at Kungliga Tekniska Högskolan in Stockholm. Gunnar Karlsson, Rolf Stadler and the rest of the group made me welcome there during the end of my previous project and the beginning of this one.

The Systems Research Group here at Cambridge are a fun and interesting group of people to work with, and made the experience of Ph.D. research almost pleasant. The Xen project in particular gave me some much-needed distraction.

A lot of people were generous enough to give me access to their DNS measurement data. Thanks go to RIPE, HEAnet and ISC OARC for zone-files, and to Martyn Johnson, John Kristoff and David Malone for name-server traces and logs. The various systems and networks admin teams at Cambridge and JANET were enormously helpful in planning and surviving the Adam zone-transfer probe.

Thanks to Anil Madhavapeddy for evangelizing OCaml, and for putting up with my many foolish questions about it.

Thanks to Jon, Steve, Anil, Jen Clark, Dave Richerby and Christina Goldschmidt, who read through various drafts and pointed out some of my many errors. Thanks too to the folks at OARC and HEAnet who sat through my presentations and gave useful feedback.

This work could not have been completed without the financial support of the Gates Cambridge Trust, and the angry, angry music of the 1990s.

Contents

Glossary of terms and abbreviations

AXFR A DNS request for the full contents of a zone (cf. IXFR).

Authoritative Server A DNS server responsible for serving records from an official replica of the zone (cf. Caching Resolver).

BIND Berkeley Internet Name Domain, a popular DNS implementation. `http://www.isc.org/sw/bind/`

Caching Resolver A DNS server that fetches records from other servers on behalf of its clients, and caches them for answering future queries. (cf. Authoritative Server).

ccTLD Country-code Top-Level Domain, e.g., `fr`, `se`. `http://www.iana.org/cctld/`

CDF Cumulative density function: the probability of a variate being less than or equal to a given value. CDFs shown in this document are empirical; that is, they show the proportion of all samples seen that are less than or equal to each value.

Closest Match The name in the DNS namespace which matches the most labels of a query name. Used for finding the Source of Synthesis (q.v.).

DLV DNSSEC Lookaside Validation: a method of distributing trusted DNSSEC keys.

DNSSEC DNS Security Extensions: source authentication for DNS records.

DNS The Domain Name System.

EDNS Extension Mechanisms for DNS: the extra types, flags and options, and larger UDP packet sizes added to the DNS in RFC 2671.

ENUM A mapping from E.164 telephone numbers onto domain names ending in `e164.arpa`.

gTLD Generic Top-level Domain, e.g., `com`, `info`.
`http://www.iana.org/gtld/gtld.htm`

ICANN The Internet Corporation for Assigned Names and Numbers, currently responsible for delegating top-level domains and maintaining lists of the numbers used in internet protocols.
`http://www.icann.org/`

IETF The Internet Engineering Task Force, the forum in which internet standards are defined.
`http://www.ietf.org/`

IXFR A DNS request for all changes to a zone since a specified version (cf. AXFR).

Master The primary authoritative server for a zone, where updates can be made (cf. Slave).

MyDNS An authoritative DNS server which serves records from an SQL database.

NOTIFY A DNS request intended to inform a slave server that a new version of the zone is available at the master.

NSD Name Server Daemon, a fast authoritative DNS server.
`http://www.nlnetlabs.nl/nsd/`

NSEC3 A proposed amendment to DNSSEC that will prevent DNS clients from being able to extract the contents of an entire zone using a series of negative queries.

RFCs "Requests For Comments", a series of publications of the IETF, which includes the definitions of internet protocols.
`http://www.rfc-editor.org/`

RIPE NCC The regional internet registry for Europe, the Middle East and central Asia. The RIPE NCC performs a monthly count of hostnames under some of the ccTLDs.
`http://www.ripe.net/`

RRSet The set of all RRs with the same name, class and type.

RR Resource Record. The unit of information stored in the DNS.

Slave A secondary authoritative server for a zone, which must synchronize its replica of the zone with the master's one.

SOA Start of Authority: a special record held at the top of every zone.

Source of Synthesis The wildcard record (if any) that is used to generate records for an otherwise unsuccessful query.

Stub Resolver DNS software running on an internet host to submit queries to caching resolvers.

TLD Top-level domain. A domain with only one label, e.g., `com`, `uk`.

TSIG Transaction SIGnature: an authentication scheme for DNS messages, using symmetric cryptography.

TTL Time To Live, the length of time for which an RRSet may be cached.

UFP The UNS First Protocol, a pessimistic update propagation protocol designed for name services.

UNS The Universal Name Service.

UPDATE A mechanism for making changes to single RRSets over the DNS request-response protocol.

Zone A contiguous subtree of the DNS namespace, the unit of administrative control.

1 Introduction

The DNS is a global nameservice that must satisfy many millions of requests per second, while allowing distributed, delegated administration and maintenance. In this dissertation, we discuss the performance requirements of the DNS, and argue that the robustness and performance of the DNS could be improved by moving towards a centralized architecture while maintaining the existing client interface and delegated administration.

1.1 Background: before the DNS

Before the DNS, naming information for the internet was held in the `HOSTS.TXT` file, a centrally-administered list of all known machines. This was, in a sense, a distributed database: every host would download a copy of the master file from an FTP server and use that copy for lookups locally.

As the internet grew, and organizations moved from time-sharing computers to networks of workstations, it was felt that both the administrative effort required to keep `HOSTS.TXT` up to date and the load on the FTP server would become too great. The hierarchical, distributed design of the DNS was intended to solve both of these problems [Moc87a]. The namespace was partitioned into administrative regions, and each organization was made responsible for providing redundant servers to publish its own section of the namespace.

At the time that the DNS was designed, several other networked nameservices were available. X.500 [CCITT05a] was developed at about the

same time and has some similar features (such as replication and recursive lookups) as well as features that are not available in the DNS (such as access control, customization and search functions). Grapevine [BLNS82], which had been deployed at Xerox, included a reliable asynchronous message transport, access control, and the ability to submit updates at any replica and have them propagate to the others. The Global Name Service [Lam86] offered PKI-based access control and dynamic update propagation.

The DNS was deliberately built as a simple name resolution system. Features available in other name services were not included, in the hope that a simpler service would be easier to implement and therefore be widely and quickly adopted [MD88]. Some of these features have later been added [VTRB97, VGEW00, AAL+05a].

1.2 Delegation and distribution

When the DNS namespace was broken up, each administrative organization was required to provide its own servers; the area of the namespace for which each server was responsible exactly matched the area over which the server's operators had administrative control.

As the DNS scaled to more and more domains, the tie between administrative delegation points and the distribution of the database has remained. The result is that far more nameservers are in existence than are needed for the task of publishing the database, and the benefit of having many servers is reduced by each server's publishing only the small subset of the namespace over which its owners and operators have administrative authority. There are almost 400 million hosts on the internet[†] [ISC] and approximately 1.4 million authoritative nameservers listed in the `com`, `net` and `org` zone files. However, delegation records necessary for access to any name in the DNS are published by only 114 of them[‡] and a client

[†]All statistics in this document, unless otherwise stated, are as of March 2006.

[‡]There are only 13 "root" servers listed in the DNS, but several of those servers are further distributed using BGP anycast [Abl03].

must talk to at least two servers to look up a new name (e.g., for the first page on a new website).

It is possible to decouple the distribution of DNS data from the hierarchy of authority [CMM02]. So long as the delegation of authority to publish records is not altered, the mechanism used to publish them could be replaced by any system with suitable characteristics. Specifically, the publication mechanism does not need to be distributed as a matter of principle — robustness, reachability and capacity are the main requirements of the DNS, and distribution is only necessary in so far as it helps us achieve these goals.

In this dissertation we propose the idea of re-centralizing the publication mechanism of the DNS: replacing more than a million servers with a single logically centralized database, served by a small number of well-provisioned and well-placed servers, and all but eliminating the link between domain ownership and domain publication.

The main advantage of such a scheme is that the clients' lookups are made much simpler: they can always get an answer from the first server they contact. Since these lookups are by far the most common action taken in the DNS, thousands of times more common than updates, making them fast and reliable is very important. The associated cost of making updates more complex is a bearable one, since the benefits include the elimination of the many pitfalls of managing delegations, as well as better service for the clients and more efficient use of machines and manpower.

1.3 Contribution of this dissertation

The thesis of this dissertation is that a centralized nameservice could replace the current DNS, and solve many of the current DNS's difficulties.

The contribution of this dissertation is divided into three parts:

A set of requirements for any proposed replacement for the DNS. We first discuss the architecture of the existing DNS in Chapters 2 and 3, and then Chapter 4 quantitatively describes the service provided by

the DNS, based on analysis of DNS traces and probes. We argue that any replacement architecture for the DNS must be able to meet or exceed that level of service, in addition to whatever other attractive qualities it might have.

A centralized architecture for the DNS. We argue that there is too much complexity in the DNS lookup protocol, and that this complexity should be moved into the update protocol, by removing the delegation mechanism and serving all DNS data from a few large, central servers. In Chapter 5 we propose an architecture for a centralized nameservice and discuss the advantages it would have over the current system. Chapter 6 discusses how the distribution of updates would work in the centralized service.

An implementation of a DNS server for large datasets. In Chapter 7 we describe the design and implementation of a DNS server intended to serve very large numbers of DNS records at high speeds.

Chapter 8 is a survey of related work in the area. We finish with a summary of our conclusions and some suggestions for future work.

1.4 Previously published material

Some of the ideas presented in Chapter 5 of this dissertation were published in *The Main Name System: An exercise in centralized computing,* Tim Deegan, Jon Crowcroft and Andrew Warfield, ACM SIGCOMM CCR 35(5), pp. 5–13, October 2005.

2 The Domain Name System

In this chapter we lay out the design of the current domain name system, as background for our later discussions of how it might be changed. This chapter is intended to give the technical detail needed for later chapters: as far as possible it consists of a description of the system, without any comment on it. Chapter 3 will describe some of the issues affecting the DNS and the motivation for changing it.

We do not describe every detail of the DNS. In particular, the rules for the layout and handling of the different record types are omitted; references describing them in detail are given in Appendix C.

When discussing the standards, we will not make a distinction between various IETF levels of standardization [Bra96]: officially, only RFCs 1002, 1034 and 1035 are Standards, and RFC 3596 (AAAA) is a Draft Standard. Most of the other RFCs cited here are Proposed Standards, and some are Informational or Experimental, but this does not necessarily reflect their standing in terms of acceptance or deployment.

2.1 Namespace

The DNS namespace is arranged as a tree (see Figure 2.1). Each edge of the tree has a *label* of up to 63 eight-bit characters[†]. Labels are equivalent up to ASCII case conversion, so `example` and `eXamPlE` are the same label [Eas06]. A *domain name* is a sequence of labels corresponding to

[†]Single-bit labels were also allowed [Cra99a], but they caused serious interoperability problems with older software and were not considered useful enough to retain in the standard [BDF+02].

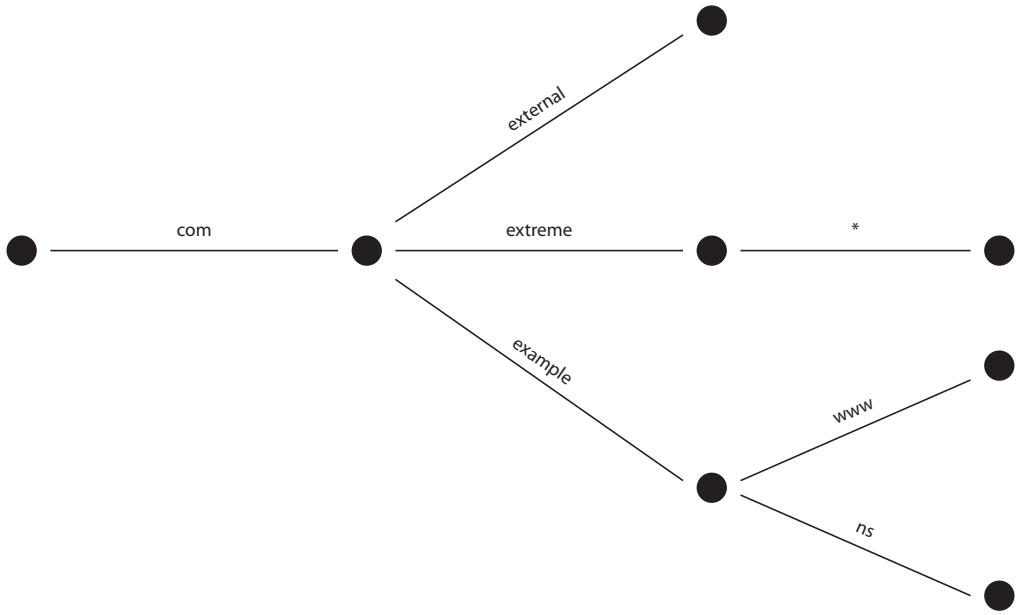


Figure 2.1: DNS namespace tree

a path through the tree from the root [Moc87a, EB97, Cra99a]. Domain names must be no more than 255 octets long, including an extra octet for each label to encode its length. For comparing and sorting names, a *canonical order* is defined, equivalent to a depth-first traversal of the namespace tree[†].

In this document, we use the normal text representation of domain names, with the labels arranged lowest-first and separated by "."; the root of the tree is represented by a single ".". For example, in Figure 2.1, `example` is a label, and `www.example.com.` is a domain name. For readability, we will omit the trailing "." of domain names.

The namespace is divided into *zones* for the purposes of administrative control, database distribution and coherence control. A zone is a contiguous subtree of the namespace, defined by *cut points*, where it is divided from the zone above (its *parent*) and any zones below it (*children*). A

[†]Names are compared using a lexicographical order by label (right-to-left), labels are compared in lexicographical order by character (left-to-right), and characters are compared by converting them to lower-case and comparing their ASCII values, with lower values coming before higher ones [AAL+05c].

zone is usually referred to by the domain name at its head. For example, the `com` zone in Figure 2.1 is a child of the root zone, and parent of the `example.com` and `external.com` zones.

Data in the DNS are arranged in *resource records* (RRs), which are identified by triples of domain name, *class* and *type*. All the RRs with a given (name, class, type) triple form an *RRSet*[†]. An RR's type specifies the layout and meaning of the data within the RR; common types of naming data include:

A An IPv4 address associated with this name.

AAAA An IPv6 address.

MX The name of a mail server which receives email for this domain, and a priority to rank it relative to other mail servers.

TXT A text string.

PTR Some other domain in the namespace (used for "reverse" lookups to map from an address back to a domain name).

The class field was designed to allow parallel namespaces to be supported, containing naming information about different networks. In practice this has not happened and the only class in use is INTERNET[‡].

Names whose first (lowest) label is $*$ are *wildcard* names: they are used to generate RRSets automatically for other names. When a query is made for a name that does not exist, if there is a matching wildcard name, the query is answered using the RRSets from the wildcard name instead. For any non-existent domain name, there is a rule for determining the single possible wildcard that can match that name: following the name from the root down through the namespace tree, stop at the node just after the last label that does exist; if there is a $*$-labelled edge leading down from that node, then the node reached by following that edge is the wildcard to use [Lew06].

[†]Like most rules in the DNS, this has an exception: the RRSIG RRs used in DNSSEC do not form RRSets, because they are conceptually a part of the RRSet they authenticate.

[‡]An exception to this is the BIND nameserver software, which will by default identify some aspects of its configuration in response to TXT queries in the CHAOS class.

2.2 Administration

The zone is the unit of administrative control in the DNS. The administrator of a zone has complete control over the namespace below the head of the zone, including the RRSets that are associated with names in the zone. By defining new cut points in the zone, the administrator can make child zones, and delegate control over them to other principals.

The root zone is currently administered by the Internet Corporation for Assigned Names and Numbers (ICANN) on behalf of the United States Department of Commerce. ICANN occasionally defines new *top-level domains* (TLDs) under the root node and delegates the corresponding zones to *registries*, which are responsible for them. The distinction is not generally made between a TLD, the zone rooted at that TLD, and the registry that administers the zone.

Most of the current TLDs are *country-code TLDs* (ccTLDs): one of these is defined for each element in the ISO 3166-1 table of two-letter country codes [ISO97] (and a few others), and the zone is typically delegated to a registry specified by the country's government [ICANN99]. The other TLDs have no geographic meaning and are usually called *generic TLDs* (gTLDs).

For historical reasons, in addition to `us`, the United States government administers `gov`, `mil` and `edu` and reserves them for U.S. government, military, and accredited educational establishments respectively. In its other capacity as the allocator of IP addresses and protocol identifiers, ICANN itself administers one other top-level domain, `arpa`, which contains infrastructure data for the internet [Hus01].

Each TLD has its own rules or guidelines for how zones are allocated below it — for example the `com` and `org` registries delegate sub-domains almost on demand, whereas the `museum` registry only delegates zones to museums and controls the names of its child zones carefully.

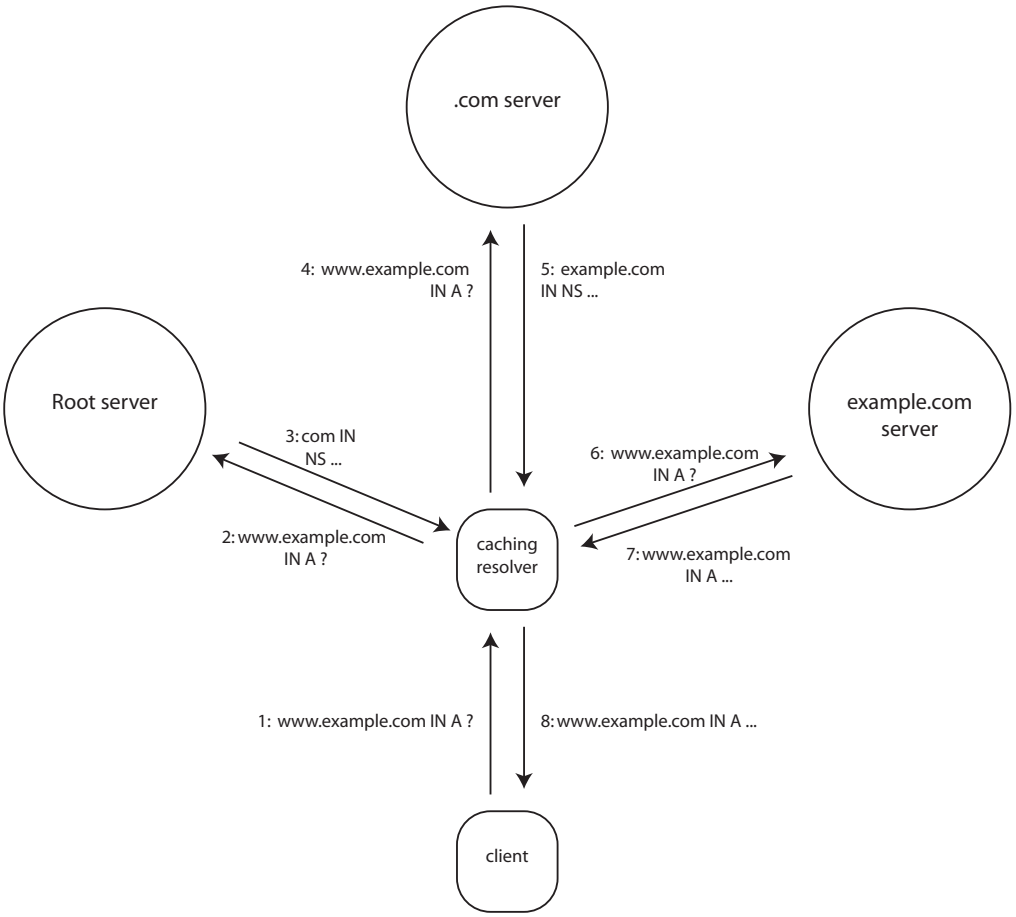


Figure 2.2: DNS lookup mechanism.

2.3 Publication and lookup

Responsibility for publishing DNS RRSets lies with the administrator of the zone in which the RRSets lie. Each zone is required to have at least two servers that answer queries about the RRSets with names in that zone; these servers are called *authoritative* for the zone. At each cut point, the child zone is authoritative for all RRSets, but the parent zone's servers also hold an extra copy of the NS RRSet, which contains the names of the child zone's authoritative servers. A list of the servers that are authoritative for the root is published by ICANN.

A *caching resolver* finds the answer to a query by starting with the root

servers and following the chain of delegations until it finds the authoritative servers for the zone the query is in. At each level it sends the query to one of the servers; if the server is authoritative it will answer the query directly, otherwise it will return the NS RRSet for the relevant child zone.

A client of the DNS typically has simple *stub resolver* software, which knows about one or more caching resolvers and sends all queries to them.

For example, a query for the A RRSet of `www.example.com` is shown in Figure 2.2:

1. A client sends a query for (`www.example.com`, INTERNET, A) to its local caching resolver, (which we will assume for simplicity has an empty cache).

2. The resolver sends the same to a server that is authoritative for the root zone. It is configured with a list of those servers as published by ICANN.

3. The `com` zone is delegated to its operator's servers, so the root server responds with an NS RRSet listing the authoritative servers for `com`.

4. The resolver sends the same query to a `com` server.

5. The `com` server responds with another NS RRSet, this time for the next cut point, which is `example.com`.

6. The resolver sends the same query to an `example.com` server.

7. This server is authoritative for `www.example.com`, so it sends back the A RRSet that it was asked for.

8. The caching resolver returns this answer to the client.

The caching resolver, as the name suggests, caches the RRSets it receives, both to save time and effort repeating queries, and so that the servers for the upper levels of the hierarchy aren't contacted for every query in a delegated zone. The time-to-live field (TTL) of each RRSet indicates to clients how long they may cache the RRSet for, and is intended to control

the trade-off between query rate and coherence: the higher the TTL, the fewer requests clients will send, but the longer it will take for changes to propagate to all caching resolvers.

In addition to the lookup protocol, the DNS standards define a protocol for synchronizing zone data between the zone's authoritative servers. One server is designated the *master* server, and the others are *slaves*. Updates to the zone are made at the master, either by directly editing its database or by sending updates to it for approval. The slaves synchronize their copies of the zone by occasionally asking the master for the the SOA RR from the head of the zone. The SOA record contains a serial number; if the serial number on the master is ahead of the one on the slave, the slave asks for a copy of the new version of the zone (using an AXFR or *zone transfer* request) or a list of the differences between the versions (using an IXFR or *incremental zone transfer* request).

This protocol is not tied to the lookup protocol; any out-of-band mechanism for keeping the authoritative servers synchronized can be used, because the synchronization is entirely between machines controlled by the zone administrator.

2.4 Wire protocol

Communication between machines in the DNS consists of exchanges of *messages*. These messages have a standard format, which is shown in Figure 2.3 [Moc87b, EB97, AAL+05a]. The fields and flags are defined below. Messages are encapsulated in UDP packets or sent in series along a TCP stream; in either case a nameserver is expected to listen for client messages on port 53. Messages sent over UDP are by default limited to 512 octets; a machine wishing to send longer messages must either use TCP or negotiate a longer payload length.

The message is made up of five sections. The first is a standard header format, which describes the type of message and various other things, including the sizes of the remaining sections. The remaining sections are variable-length [Moc87b, EB97].

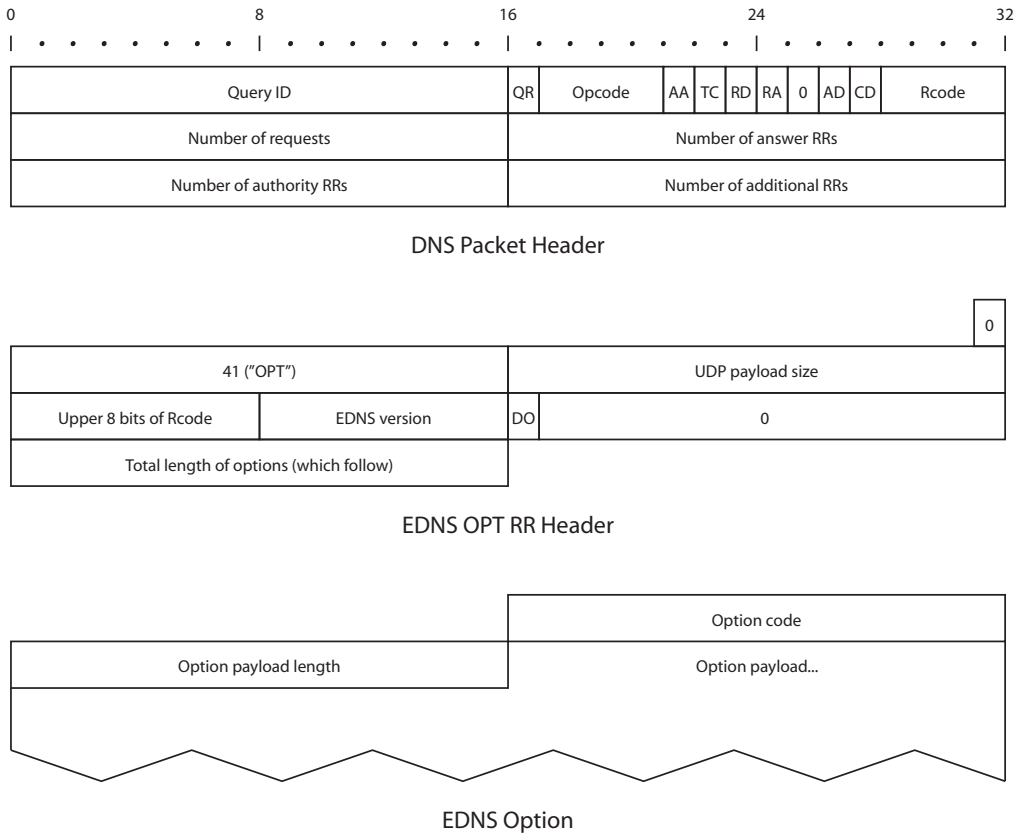


Figure 2.3: DNS message header and EDNS extensions.

- The question section describes the RRSet being queried or updated.
- The answer section contains the requested RRSets in the response, and in some cases also contains the answer to the query that a client is expected to send next.
- The authority section contains details of the zone the response came from.
- The additional records section can contain various other useful records, depending on the contents of the rest of the message.

The Extension Mechanisms for DNS (EDNS) [Vix99] define an updated version of the DNS message format, extending the Rcode, label type, and flag fields of the original DNS message, and adding a UDP payload length

field and a version number. Rather than use a new header layout, the extended fields are encapsulated in a special resource record and included in the additional records section of an old-style message. A compliant implementation must parse the additional records section of a message before it can interpret the message's header fields. The current message format (that is, the original layout, extended with the special RR shown in Figure 2.3) is known as EDNS0.

Operations: QUERY, NOTIFY and UPDATE

All protocol operations in the DNS are expressed as a request from one machine to another, and at least one matching response. The meaning of the request is defined by the opcode field, which has three possible settings.

QUERY is the basic query-response mechanism, in addition to piggybacking some other features (see below). The client sends a QUERY message to the server containing, in the question section, the domain name, type and class of the RRSet it is asking for. The server responds, echoing the question section and including the requested RRSet, if it exists, in the answer section. The server also includes some information about the zone in the authority section: if the query was successful or the name was in a delegated child zone, this is the NS RRSet of the zone head; if it was unsuccessful, it is the SOA RR. The additional records section is filled in with some more RRSets, which depend on the type of the queried RRSet — for example, if the query was for an MX RRSet, which contains the domain names of some mail servers, the server might include the IP addresses associated with those names.

NOTIFY allows a master server to indicate to a slave server that the zone contents have changed, by sending a NOTIFY request to the slave for the name of the zone head [Vix96]. This lets slaves respond more quickly to zone contents updates without having to poll more often.

UPDATE allows a client to request that a server add, change or delete an RRSet in its zone [VTRB97]. The client gives a list of preconditions, stating which RRSets must exist or not exist before the change should be accepted. The server responds using the Rcode field to indicate that the update was successful, or why it was not.

Two other opcodes, an "inverse" query (IQUERY) and a server status request (STATUS), were defined, but they are not used. IQUERY has been withdrawn [Law02] and the semantics of STATUS were never defined.

ID

The ID field is used to match requests to responses. Clients fill in the ID field with a number that identifies the request being made, and servers mirror the ID field in the response message.

Flags

There are six flag bits in the standard header and sixteen in the EDNS0 OPT record, of which the following are defined:

AA This response contains authoritative data (i.e. not cached).
TC This message was truncated to fit a UDP datagram.
RD Please perform recursive queries to answer this query.
RA This server is willing to perform recursive queries.
AD This message contains DNSSEC-authenticated records.
CD Do not perform DNSSEC validation on this request.
DO Send DNSSEC RRs in the response to this request. (Requires EDNS0.)

Response codes

Five Rcodes were defined in the original protocol: one as a catch-all for partial or complete success and four others to indicate different failure

modes [Moc87b]. Five Rcodes were added for explaining why an update failed [VTRB97]. EDNS0 added a version mismatch error [Vix99]; symmetric-key signatures add six more [VGEW00, Eas00].

2.5 Metadata and signalling

In addition to the RRSets containing naming data, the DNS has a number of record types that are used to hold metadata about the namespace or to carry signals from one machine to another, as described below.

Namespace metadata: NS, SOA, CNAME and DNAME

These four record types are used to mark the cut points between zones, and to implement aliases.

SOA (Start of Authority) records identify zones and contain metadata used for keeping replicas synchronized. Each zone has a single SOA RR, which contains the name of the zone master, a contact email address for the administrator, a serial number for the zone and four timer settings. The timers tell the slave servers how often they should refresh their copies of the zone contents, and tell caching resolvers how long they may cache negative replies for names in the zone [And98].

NS RRSets mark zone boundaries: an NS RRSet contains the names of the authoritative servers for the zone, and is served by the parent zone's servers as well as the child's. All other RRs with that name (except for DNSSEC digital signatures — see below) are served only by the child's servers. The NS RRSet should be the same regardless of which server is asked, but this must be ensured out of band.

CNAME records are used to indicate aliases: a CNAME RR for a name contains another domain name (the target), and indicates that any queries for types other than CNAME at the first name should be asked again using the target name.

DNAME records have the same effect but apply to queries for all names below the owner name in the namespace: thus a CNAME RR for `mail.example.com` containing the name `mail.isp.net` would cause a query for the A RRSet of `mail.example.com` to be reissued as a query for the A RRSet of `mail.isp.net`. A DNAME RR with the same contents, on the other hand, would have no effect on that query but would cause a query for the A RRSet of `imap.mail.example.com` to be reissued as a query for `imap.mail.isp.net` [Cra99b].

Unlike wildcard processing, which is done entirely at the server, this query reissuing is done by the caching resolver on receipt of the CNAME or DNAME RR. However, servers do need to handle these aliases: they must know to issue them in response to queries for other types. Also, if they know the answer to the reissued query that the resolver ought to send, they are expected to include it in the response with the CNAME or DNAME RR. This saves the latency of another query/response pair.

Protocol elements: AXFR, IXFR, OPT, TSIG, TKEY, ANY and NONE

There are also a number of record types for which records do not exist at all in the namespace; they are used to piggyback other kinds of protocol data on top of queries and responses.

AXFR and IXFR queries are only valid when the name being queried is the head of a zone. The server responds to an AXFR query by sending every RRSet in the zone; this allows slave servers for a zone to acquire up-to-date copies of the zone contents. An IXFR query [Oht96] is similar, but the client sends with it the SOA RR from a version of the zone that it already has access to, and the response is a list of the differences between that version of the zone and the current one.

OPT records are used to signal the use of EDNS. An OPT record is included in the additional records section of a query or response to signal that

EDNS features are available or required.

TSIG and TKEY are used for adding symmetric-key signatures to messages, and are discussed below.

ANY is a wildcard type, used in queries to ask for all records at a given name and class.

NONE is the type used in update messages to assert that a domain name has no records of any type associated with it.

There are equivalent ANY and NONE classes that serve the same purposes for the class field.

2.6 Additional RRSets in responses

Answers to DNS queries must contain extra RRs as well as the ones asked for. These depend on the type of the records being queried, as well as server configuration and the contents of the response records themselves. They usually require the server to look in more than one place in the namespace to answer a query properly. Some of the secondary lookups required in an authoritative server after a successful lookup are as follows.

1. The NS RRSet of the origin of the zone, in the authority section.
2. In a DNSSEC-secured zone, the RRSIG records that hold signatures for the other RRSets in the message, and (optionally) the DNSKEY RRSet of the zone origin.
3. If there is a CNAME RR for this name and class, the answer to a lookup of the same class and type using the target of the CNAME.
4. (Optionally) the SOA RR of the zone origin.

For an unsuccessful lookup, the server must find:

1. If the queried name is in a part of the namespace covered by a DNAME RR, the answer to a lookup of the same class and type using the appropriate substitution from the DNAME.
2. If the name is in a zone that is delegated away from this server, the NS RRSet of the closest enclosing delegation known by this server.
3. The appropriate wildcard name.
4. In a DNSSEC-secured zone, the NSEC RR that covers the gap in the zone where the name would have been, and enough others to demonstrate that there is no applicable wildcard name.
5. The SOA RR for the zone origin.

For both successful and unsuccessful queries, the server must do further lookups for some of the domain names that appeared in the contents of records in the response so far. For some types, including NS and MX, the server must extract domain names from the contents of the original reply and include any records of type A or AAAA that those names have, on the assumption that the querier is about to ask for them.

2.7 Security

The DNS was not designed with any security features beyond a belief that a record received from an authoritative server should be trusted. No additional measures were taken to secure the underlying UDP and TCP communications, or to check that the data returned from a zone's servers matched the specification given by the zone's administrators. It was assumed that such security features could easily be added later. Some of them have since been deployed, and some are still in development.

End-to-end authentication

DNSSEC [AAL+05a, AAL+05c, AAL+05b] introduces a public-key signature system for the DNS. DNSSEC signing keys are associated with zones.

A signature is calculated off-line for each RRSet in a secured zone, and recorded in an RRSIG record. The RRSIG record is stored alongside the original record and sent with it in response to queries. The client can verify the signature and be sure that the record set is as intended by the administrator, or at least by some holder of the signing key. In addition, the gaps between RRSets in the zones are described using NSEC RRs, which are also signed. This prevents an attacker from denying the existence of an RRSet that does in fact exist, because he cannot forge a signed NSEC RR that covers the correct name. The server needs to be able to find the appropriate NSEC RR when it receives a query for a non-existent name in a signed zone; this involves finding the closest preceding name (in the canonical order) that does have records.

Signing keys are arranged in a tree of trust that follows the namespace. At each cut point, the child zone's public keys are signed by the parent zone's keys[†]. The chain of signatures must eventually end with a key that is known to the client. The intention is that the root zone should be signed and its key-signing public key published, although this has not yet happened.

Hop-by-hop authentication

In order to guard against man-in-the-middle attacks on the communications between servers, a symmetric-key cryptographic *transaction signature* was introduced [VGEW00]. The sending machine can sign its communications and include the signature as a TSIG RR in the message, allowing the receiving machine to be sure that the data came from a machine that possesses the secret key.

TSIG security is more widely deployed than DNSSEC; it is often used between the authoritative servers of a zone, to protect the zone transfers between master and slaves. A different secret key is used between the master and each of its slaves, and the keys are agreed out of band.

[†]For operational reasons, each zone has a key-signing key that is intended to change infrequently, and is used to sign the keys that actually sign RRSets; see [AAL+05c] for details.

Because a key must be shared between every pair of communicating machines, it is not suitable for securing client queries unless the client base for a server is very tightly controlled. The TKEY record type and its associated processing rules [Eas00, KGG+03] allow a client and server to establish a session key for use with TSIG but they must already be able to authenticate each other in some other way. The security protocol negotiations are piggybacked on an exchange of TKEY RRs between the two ends.

Cache poisoning

Another vulnerability in the DNS arises from the way caching resolvers use cached NS RRSets to shortcut the resolution path for zones they have visited before. A malicious DNS server, as part of its response to some query, could include a spurious NS RRSet that claimed that the victim zone was delegated to servers under malicious control — for example, along with the A records for `www.malicious.com` the malicious server sends an NS RRSet for `victim.com` pointing to `ns.malicious.com`. Later, queries for names in `victim.com` will be sent to the malicious server instead of the proper servers. In this way the zone can be hijacked without having to attack the servers for the zone or any of its parents[†].

Newer caching resolvers only accept RRSets that come from servers that are authoritative for the zone in which the RRSet falls (or in some circumstances from a parent zone), to prevent this kind of poisoning. Variants on the attack can still be made by spoofing response packets from the authoritative servers, or by subverting the servers that serve the *names* of a zone's authoritative servers (rather than the authoritative servers themselves), in order to hijack the process one step earlier [RS05].

[†]A list of servers known to be issuing poisoned records is available from the Measurement Factory's DNS survey, at
`http://dns.measurement-factory.com/surveys/poisoners.html`.

2.8 Extensions

While the main use of the DNS remains the serving of a coherent database of naming information, there are some servers that do more than that. For example, load-balancing and server selection for content distribution networks is sometimes done at the DNS level: the result of a DNS query in a load-balanced zone depends on the IP address of the querier, automatically directing each client to the "best" server. The choice of server is typically based on an estimate of distance, with a lookup table of which servers are closest to each address block; it could also be based on feedback from the servers about their current load [Cis99]. Using the DNS for server selection provides poor responsiveness to change (because of clients that ignore TTLs [PAS+04]) and the use of DNS caches introduces errors into the choice of best server for each client because a user's DNS cache is not necessarily a useful estimate for that user's actual position in the network [PHL03]. However, it does usually avoid the worst choices.

The DNS is still under active development. New record types are being defined to support services as they are deployed on the internet [WS05, SLG06, NL05, Mes05, Mor05, dL05]. New record types can require more functionality at the server, as there are rules governing which extra information a server should supply with each record type. Authentication mechanisms for DNS records [AAL+05c] are being deployed slowly, but new schemes are being developed to help speed this deployment [AW06], and to allow administrators to secure their zones without revealing the entire contents of their zones [LSA05].

2.9 Summary

We have described the design and operation of the Domain Name System. In the next chapter we will consider some of the issues experienced with the DNS and motivation for making changes to it.

3 Motivation

Now that we have described the mechanisms of the DNS, we can discuss the reasons why we would want to consider changing it. In this chapter we will look at some of the issues that have been reported with the DNS, and argue that the distributed nature of the DNS lookup protocol is responsible for many of them.

Lookup latency

Resolving names in the DNS can take long enough to cause noticeable delays in interactive applications [HA00, CK00]. A resolver may need to talk to several authoritative servers to answer a single question, and each additional server adds network delays, as well as the possibility of congestion, overloaded servers, and failure. Generally, top-level domain (TLD) servers can be expected to be well placed and geographically diverse; lower-level zones are often not so lucky. This is made worse by the long timeouts in resolvers and stubs when errors are encountered [PPPW04]. In addition, high load at the caching resolvers can cause delays.

Update latency

The latency of updates in the DNS is governed by the "time to live" (TTL) field of the record being updated. For planned updates, the TTL can be reduced in advance, allowing a speedy propagation of the update in exchange for briefly increased traffic. For unplanned updates (e.g., in response to an outage or attack) the old record may remain in caches until its TTL expires.

This timeout-based caching mechanism is not a good fit with the patterns of change in the data served. Many DNS records are stable over months, but have TTLs of a few hours. At this timescale the TTL is effectively an indicator of how quickly updates should propagate through the network, but it is used by caches as if it were an indicator of how soon the record is likely to change. This generates unnecessary queries, and indicates that a "push" mechanism for updates would be more appropriate for these parts of the DNS. Jung et al. [JSBM01] show that, based on TCP connections seen in traces, 80% of the effectiveness of TTL-based caching is gained with TTLs of only 15 minutes, even though the data may be static for much longer than that.

Administrative difficulties

Even a properly-formed DNS zone can cause problems for its owners, and for other DNS users, if the delegation from the parent zone is not properly handled. The most common delegation errors are caused by a lack of communication between the zone's administrators, their ISPs, and the administrators of the parent zone. In 2004, 15% of forward zone delegations from TLDs were "lame" in at least one server [PXL+04] (i.e., a server that did not serve the zone was announced as doing so by the parent zone). Servers publishing out-of-date zone data after a zone has been re-delegated away from them also cause problems. Circular glue dependencies, which cause delays in resolving the zone from a cold cache and reduce the number of useful servers available[†], affected about 5% of zones [PXL+04]. These errors are directly connected to the way the DNS is distributed along administrative boundaries.

In addition to the risks involved in preparing the contents of a zone, the software used on nameservers is complicated and requires some expertise to configure, secure and maintain properly. This expertise is not in evidence at all nameservers[‡].

[†] See `http://cr.yp.to/djbdns/notes.html#gluelessness`

[‡] For a survey of configuration problems found on public nameservers, see `http://www.menandmice.com/6000/6000_domain_health.html`

Misconfiguration of clients and resolvers causes problems too: it is responsible for a large fraction of the load on the root servers [WF03, Wes04]. This has been a problem with the DNS for some time [MD88] and, despite efforts to educate administrators, the situation is not improving.

Vulnerability to denial of service

Redundancy is built by replication of data, but this happens multiple times per name — an organization must rely not only on its own nameservers being operational, but also on the nameservers for every level above it in the hierarchy. Again, this is caused by the distribution model.

There have been distributed denial-of-service attacks on the root and TLD servers in the past. Some of them have been successful in causing delays and losses [VSS02], although anecdotal evidence suggests that to date, human error has been much more effective than malice in causing outages at the upper levels of the DNS, and care is taken in deploying root servers that they can survive large spikes in load [BKKP00]. Lower-level zones, which typically do not have the same levels of funding and expertise available to them, are more vulnerable to attack. Many zones are served by two nameservers that are on the same LAN, or even the same machine [PXL+04], although the standard requires each zone to have redundant servers.

Lack of authentication

The current DNS relies almost entirely on IP addresses to authenticate responses: any attacker capable of intercepting traffic between a client and a server can inject false information. Mechanisms have been developed to use shared-key and public-key cryptography to provide stronger authentication of replies, but although TSIG [VGEW00] is now commonly used to protect zone transfers, DNSSEC [AAL+05a] is not being deployed quickly — indeed the details of the protocol are still being debated after

ten years of development. It is also a complex addition to an already complex protocol, so there are relatively few implementations.

Implementation difficulties

The wire protocol of the DNS is not easy to implement correctly[†]. In particular, the "compression" algorithm, where common suffixes in a packet can be merged by using pointers from one name to another, is a common cause of error. Also, as we shall see in Chapters 5 and 7, the way that the tree of the namespace is used restricts the implementation options for DNS servers. In particular, DNSSEC requires that a server be able to find the record that precedes a query name when the query name is not in the database. It would be unfair to call these architectural problems; most internet protocols have similar pitfalls and difficulties.

3.1 An observation about complexity

The DNS has two protocols, one for publishing records and one for looking them up.

The lookup mechanism of the DNS is a complex one: a caching resolver acting on behalf of a client interacts with multiple authoritative servers in order to locate the record being queried. Even an entirely successful query might travel from machine to machine six to ten times before it finally returns to the stub resolver that generated it, and will rely on the correct configuration and functioning of machines other than the client and the server holding the record. Each network hop is an opportunity for delay or failure, and each machine involved is a possible source of error, delay or failure. The complexity of this lookup architecture is responsible for some of the observed difficulties with the DNS: in particular for the latency issues associated with retry timeouts, and the vulnerability of the entire service to attacks on the root and TLD servers.

[†]Even now, DNS implementation errors are regularly reported. See, for example, CVE-2003-0432, CVE-2004-0445, CVE-2005-0034 and CVE-2006-0351.

The publication of records in the DNS is more straightforward. The zone administrator sends the record to the primary authoritative server, and the secondary servers synchronize themselves with the primary. This process involves only those machines directly responsible for serving the record, all of which are under the administrator's control; it does not rely on the configuration of other zones or their servers.

There are also differences in how the two protocols are used:

- Lookups are several thousand times more frequent than updates, and clients outnumber servers by more than two hundred to one. Typically a failed or delayed lookup will cause failure or delay in a higher-level protocol interaction, which may itself be delay-sensitive: for example, delays of more than a few seconds are considered problematic in web site downloads [BBK00].

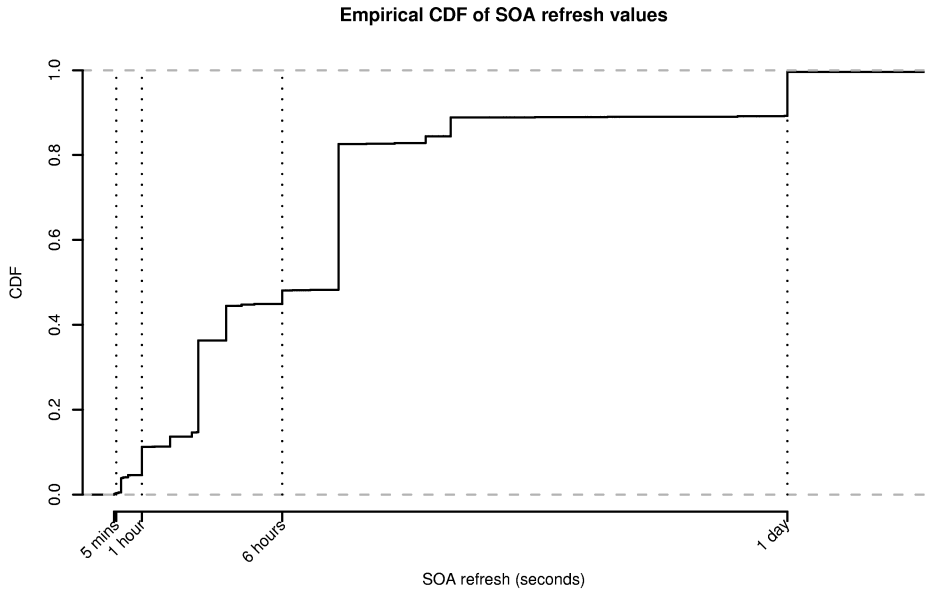


Figure 3.1: SOA refresh timers (from zones seen by RIPE and Adam)

- Updates are more tolerant of delay: typically a ten-second delay would not be noticed when updating a DNS record. Figure 3.1

shows that the vast majority of zones have refresh timers set longer than five minutes. (Although this doesn't take DNS NOTIFY into account, it is an indication of the sort of timescale that is acceptable for zone consistency. In zones that use NOTIFY, if a notification is lost between master and slave, BIND 9 waits for 15 seconds before re-sending.) In addition, DNS updates can be pre-loaded in a way that lookups cannot: some servers allow administrators to load updates into the DNS before they are needed, timestamped with when they are to come into effect.

This leads us to an observation about the architecture of the DNS. A distributed system on the scale of the DNS must have some complexity, and wherever the complexity is, there are risks of failure and delay. We would like the complexity to be in the place where the failure and delay can be best tolerated, and this is not the case in the DNS. If DNS lookups can be made simpler — even at the cost of making updates more complex — the overall behaviour of the system will be improved. In other words we believe:

> *The complexity of the DNS should be moved from the lookup protocol into the update protocol.*

The current DNS is made up of two halves: simple centralized updates and complex distributed lookups. A system using centralized lookups and distributed updates would be a better match for the requirements. Ideally, client caches should be able to send queries to *any* DNS server and have that server return the answer immediately; the hard work is done when an update is made, replicating the record at every server.

This would solve the problems of high latency, as well as many of the administrative difficulties associated with the delegation of zones between servers. The other problems discussed above are either political or related to the wire protocol, and outside the scope of this dissertation. We deliberately do not attempt to resolve any of the complex legal and political issues of zone ownership and control; and since any improved DNS architecture must be backwards-compatible with the millions of clients

deployed in the internet, we do not address the wire protocol except to require that the new service should support it.

In Chapter 5, we will discuss the design of a centralized nameservice. First, however, we will examine the size and performance of the current DNS, in order to specify the minimum requirements which any new nameservice must meet.

4 Quantifying the DNS

The first question we must ask when designing a distributed lookup system is: what data will we be serving? How is it organized, and what is the expected load of queries? In this chapter we will analyse measurements of the current DNS, and distil a concise set of requirements for our replacement system.

4.1 Sources of data

First, we will describe the various data sets which we will use. Some of these are existing data from other DNS measurement projects. Some (in particular the Adam zone files and the update probe queries) are the results of measurement probes made specifically for this project.

The Adam zone transfers

In June and July of 2004, we gathered data on the contents of the DNS by recursively requesting full copies of all zones under `com`, `net`, `org`, `info` and `coop`. This probe took about six weeks in all. We found that 23% of zones had at least one server that responded to these zone transfer (AXFR) requests, giving us 8.5 million zones.

The Adam probe used a set of Bloom filters to decide which nameservers were consistently denying zone transfers, and did not send any more queries to those servers. Altogether, the probe made 8.5 million successful zone transfer queries and 6.7 million unsuccessful ones.

We took several precautions to make it easy for server administrators to reassure themselves that the queries were not malicious.

- A web page was published giving details of what the probe was doing and why, and how to contact us about it if it seemed to be misbehaving.

- All queries came from a dedicated IP address, with a PTR record indicating that this was a DNS probe, and TXT records pointing to the project website. No other outbound connections were made from this IP address.

- A web server was run on the probe's IP address, which redirected all queries to the project website.

- People who might receive queries about it — departmental and university network administrators and the local CERT — were consulted in advance. They could then answer any queries by pointing to the website and forwarding any further questions or demands to us.

In total, not counting people who contacted administrators or CERTs and were content once they knew what the probe was doing, we received ten demands that we stop querying particular addresses or net blocks, and eight offers of assistance from administrators who had whitelisted us.

The RIPE region hostcount

Similar data for 90 European ccTLDs were provided by the RIPE NCC, from their monthly hostcount project [RIPE]. The RIPE hostcount is a well-established project, and major ISPs have been lobbied to allow zone transfers to RIPE's local collection points: for example, the collectors successfully transferred 50% of zones (a little over 10 million) from 90 ccTLDs in April 2004. Our analysis below is based on that dataset.

The ISC internet domain survey

The Internet Systems Consortium (ISC)'s quarterly internet domain survey [ISC] is a list of all the PTR records for assigned IPv4 addresses. It is gathered by recursive zone-transfers of the `in-addr.arpa` zone. Where zone transfers are not available, the zone is enumerated by querying for every address in an assigned network. It does not contain any information other than address-to-name mappings, but it does contain records from zones whose administrators refuse to allow zone transfers. The ISC probe saw 394 million PTR records in January 2006.

Query traces

We also used two traces of DNS queries taken at academic nameservers. The first was taken at an authoritative nameserver in University College, London in August 2004. It contains almost 600,000 queries seen over 4 and a half hours. The server is authoritative for various UK academic zones and reverse lookups.

The second was taken in the University of Cambridge Computer Laboratory, at a caching resolver that serves about a third of all DNS requests from a department with 250 researchers and staff. This server forwards all cache misses to a second level cache in the university's network. The trace was taken over 13 days in August 2004, and contains 2.5 million client queries. The server itself sent 1.3 million queries for records that were not cached.

Update probes

In December 2005, we probed two sets of public RRSets to see how often they changed. We sampled RRSets from four data sources: the Adam and RIPE probes, the ISC probes, and the University of Cambridge query trace. We removed those that were no longer resolvable, and then queried the remaining RRSets every four hours for a period of four weeks. We did not

cache the results, but queried all the authoritative servers for each record each time. The results of the probe are discussed below.

In contrast to the zone-transfers of the Adam probe, these were normal UDP queries, and were repeated every four hours regardless of the previous responses. We have so far received no enquiries of any kind about the query traffic generated by this probe.

4.2 Contents

Number of RRs

The Adam dataset contains 86.9 million RRs in 8.5 million zones. If the zones returned are representative of the gTLDs generally, we can estimate that there are about 371 million RRs under the gTLDs. The RIPE data contains 186 million RRs in 9.8 million zones, giving us an estimate of another 371 million RRs under the RIPE-area ccTLDs. There are also another 158 ccTLDs not covered by the RIPE count. In the output of the ISC probe, those ccTLDs appear only 81% as often as the RIPE-area ones in the contents of PTR records. Using this as a guide to their relative sizes, we can then estimate that the non-RIPE ccTLDs contain 304 million RRs. This gives us a total of 1.05 billion RRs in the summer of 2004. Multiplying by the increase in the number of hosts since then, we get an estimate of 1.45 billion for January 2006.

In January 2006 the ISC survey found that there were about 395 million IP addresses that have PTR records registered under `in-addr.arpa`.

This gives us a total estimate of about 1.8 billion RRs. That is roughly 4.6 RRs per host, or about 2–3 per internet user, depending on which estimate we take for the user population. For example, the CIA World Factbook [CIA05] estimates that there are 604 million users, but market-research firms often claim more than 900 million.

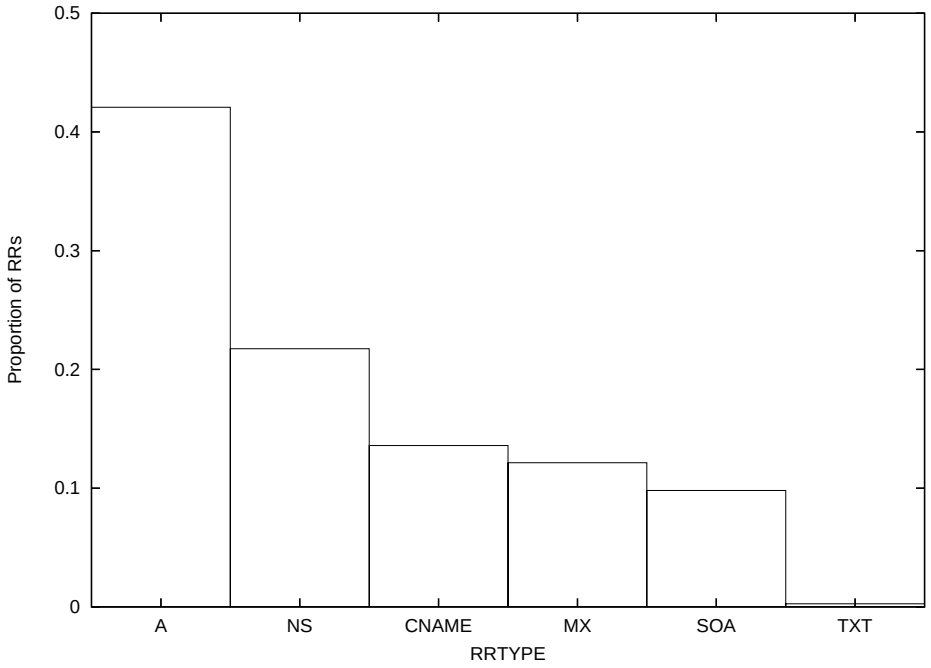


Figure 4.1: Distribution of popular record types in Adam data.

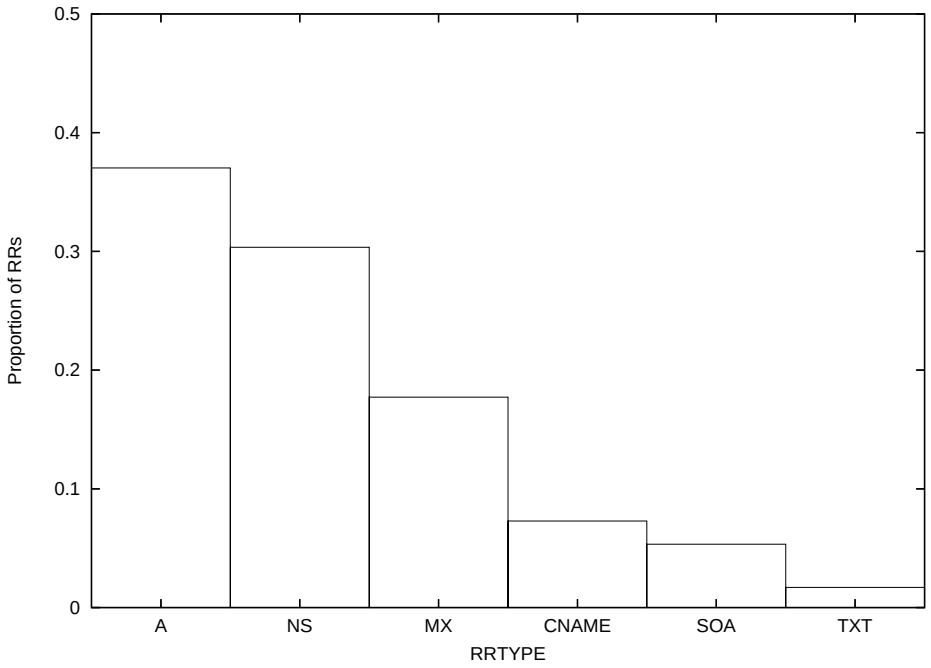


Figure 4.2: Distribution of popular record types in RIPE data.

Types of RRs

Figure 4.1 shows the proportion of each type of RR in the Adam dataset. In total, 41 record types were seen, but the five types shown represent more than 99% of the records. Figure 4.2 shows the distribution for the RIPE dataset. MX and NS RRs are markedly more popular in the RIPE dataset. This might reflect a class of more complex zones that allow zone transfers to RIPE but do not normally allow zone transfers, as well as different usage patterns under the ccTLDs.

RRSet sizes

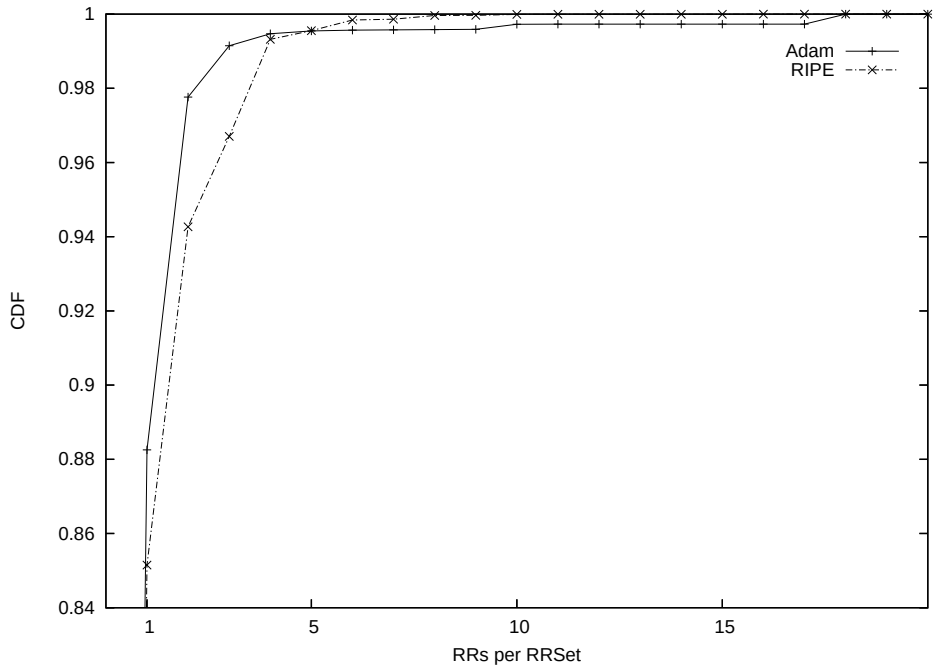


Figure 4.3: CDFs of RRSet sizes in Adam and RIPE data.

Figure 4.3 shows the empirical cumulative density functions (CDFs) of RRSet sizes in the Adam and RIPE data sets. The average RRSet size over both data sets is 1.2279 RRs (with 95% confidence within ± 0.00051).

Number of names

The Adam probes contain 45.4 million unique record-owning names, an average of 1.9 RRs per name (that is, the average number of RRs in all RRSets of any type that are indexed under that name). The RIPE probes contain 70.4 million names, an average of 5.2 RRs per name. (The RIPE data has higher number of NS and MX RRs, and larger RRSets.) The reverse records in the ISC traces contain only one record per name.

Zone sizes

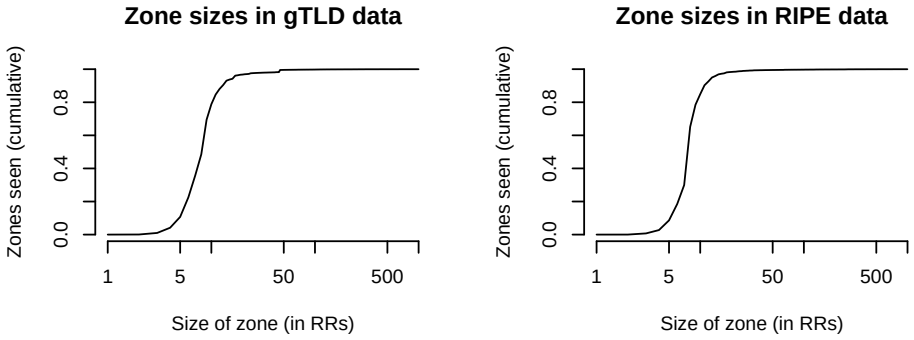


Figure 4.4: CDFs of zone sizes in Adam and RIPE data.

Figure 4.4 shows the CDFs of zone sizes in the Adam and RIPE data sets. For both data sets, most zones have between five and ten records. Since a valid delegation ought to have at least three meta-RRs (one SOA and two NS) this represents two to seven records of useful data.

TTLs

Figure 4.5 shows a cumulative distribution function of the TTLs of records in the Adam and RIPE datasets. Most TTLs are between one hour and one day. The large jump at one hour corresponds to the similar findings by Mockapetris and Dunlap, which they attribute to the way people read the standards: “Sample TTL values which mapped to an hour were

always copied; text that said the values should be a few days was ignored" [MD88]. The second jump at one day shows that perhaps the text is not being ignored as much any more.

About 1.7% of RIPE records and 0.6% of Adam records have TTLs of less than a minute. Since very short TTLs are only useful for dynamically generated records, such as those used in load-balancing schemes, they are likely to be underrepresented here. Less than 1% of records have TTLs longer than 1 week; many DNS caches will not cache records for longer than a week in any case [EB97].

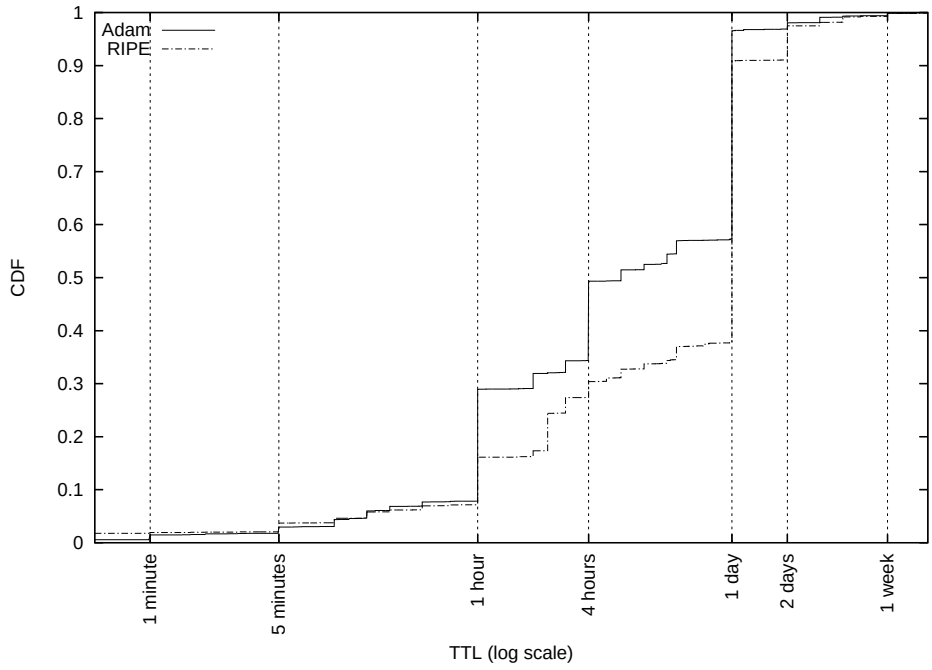


Figure 4.5: CDFs of TTL settings in Adam and RIPE data.

4.3 Updates

Previous work says very little about the rate at which RRSets change. DNS measurement work does not cover it because it is not important to the mechanism of DNS lookups (update traffic is handled privately between authorities and only TTL has any effect on lookup traffic), and because

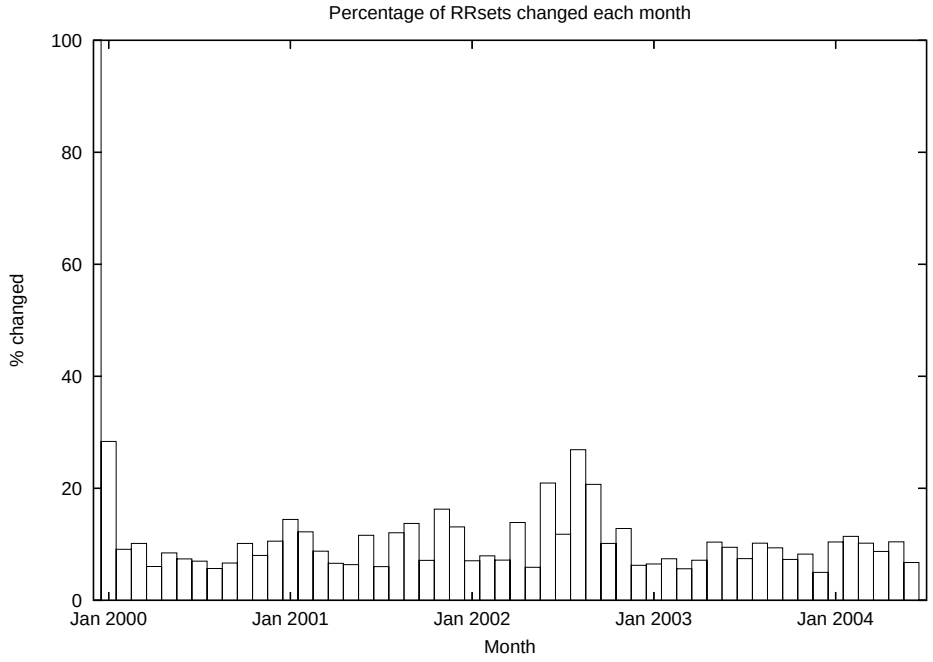


Figure 4.6: Changes to `ie` RRSets over time.

it is not easily visible in trace data. Zone transfers can be seen between servers but (at least where incremental transfers are not yet in use) do not give a good idea of how much of a zone has changed.

Cohen and Kaplan [CK01] saw 0.25–0.5% of RRSets change at least daily and fewer than 0.1% change hourly. This is equivalent to about 380 updates per second across the entire DNS. They also say that 97–98% of records did not change at all, although they do not say how long their probe sequence lasted. Their sample set was the A RRSets of a list of webservers taken from a large and busy web cache.

LaFlamme and Levine [LL04] report that the daily changes to the `com`, `net` and `org` TLD zones come to less than 500 KB per day, which is about 6600 RRSet changes per day.

Figure 4.6 shows the change history of the RRSets seen under the TLD for Ireland by the RIPE recursive-transfer tool. For each month it gives the percentage of all RRSets that were updated (including creation and deletion). It shows that the majority of records seen by the RIPE probe

stay the same over a timescale of months: in any given month we could expect 75–85% of the RRSets to be unchanged.

This graph does not include reverse zones or those dynamic zones that do not allow zone transfers: the RIPE probe sees about half of the forward zones under `ie`. Also it does not show how often RRSets change over a month, only whether they changed at all over that period.

One problem with the data given above is that it is selective: the RIPE data almost certainly excludes highly dynamic and automatically generated zones; the TLD update rate only includes whole zones being delegated or changing servers; and Cohen and others select their RRSets from the popular lookups of just one application, so do not see DNS blacklists, reverse lookups, ENUM records, or any other record types.

Update probes

In order to get a more representative view of the way that records change, we sampled records from our various data sets and probed them regularly to look for changes. We selected 1,000 RRSets from the RIPE and Adam zonefiles, the ISC reverse data and the queries seen in the University of Cambridge DNS trace. We removed those records that no longer existed, leaving us with 3,000 records. We then queried each one every four hours over four weeks.

On analysing the data, we noticed several ways in which RRSets can appear to change from probe to probe, without actually having changed at the servers.

Variations in TTL: Some zones were lame in one or more servers; that is, there were servers that were supposed to be serving the zone from authoritative data, but were not configured to do so. Of those servers, some were serving cached records obtained from the other servers at some point in the past. As a result, the TTLs of the answers seemed to vary over time, and to depend on which server was asked. Although these responses were usually marked as cached data, in

some cases they were marked as authoritative, which made them harder to diagnose.

One variant on this occurred when the zone was entirely lame, but the RRSet being queried for was the NS RRSet for the zone head. If any of the servers listed in the NS RRSet was also an open caching resolver, it could return a cached copy of the NS RRSet, which it had obtained from the parent zone.

Another version arose when following CNAME records: if the authoritative server was also a caching resolver it would return the CNAME RR and also its cached copy of the RRSet from the CNAME's target domain. Since the second RR was served from cache, its TTL would vary from query to query. We note that this does not involve any misconfiguration at the server, only the choice to allow recursive and cached data to be served from a public-facing server. (For various operational reasons this is unwise, but it is allowed by the standards, and is a common configuration[†].)

Inconsistent data: Sometimes the various servers for a zone were serving different versions of the zone. Simple queries for the RRSet would appear to change depending on which server was asked. In our probes we asked every authoritative server, so we saw the union of all data sets. However, if some servers were temporarily unreachable, this set would appear to change.

In an extreme case, we saw one ISP that had five nameservers, each serving a single record, different from the other servers, but consistent over time.

N-of-M: Some RRSets had large numbers of records, and their servers would respond with a subset of the total each time they were asked. While we suspect these changes did not reflect actual changes in configuration, we could not be sure that there wasn't some active

[†]This configuration was seen at over 75% of authoritative servers in `http://dns.measurement-factory.com/surveys/sum1.html`

load-balancing behind the changes, so have classed these RRSets as "dynamically generated".

We detected five records that were clearly dynamically generated: they changed each time the question was asked, and had low TTLs. It would be incorrect to think of this behaviour as a series of frequent updates to an RRSet. Rather, it is a special case and must be considered separately.

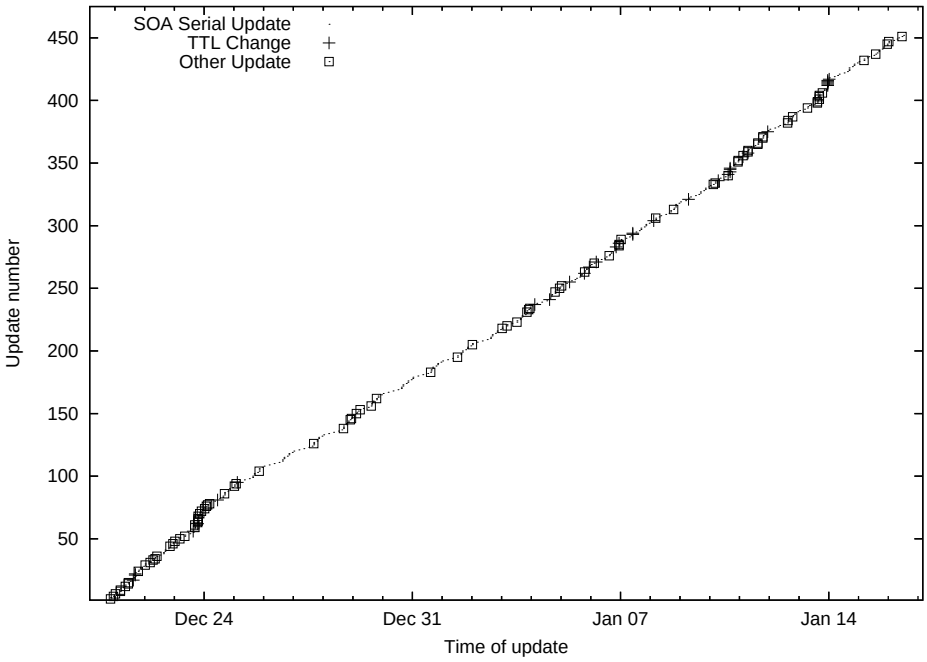


Figure 4.7: Updates detected by the probes.

Figure 4.7 shows the times at which we detected updates, with the updates broken down into those that only changed an SOA serial number, those that only changed the TTL of an RRSet, and all other updates.

Table 4.1 summarizes the results of the probes. Weighting the Adam, ISC and RIPE rates by the amount of data they represent, we get an average of 0.0063 updates per RRSet per day. This agrees roughly with Figure 4.6, and is equivalent to a rate of about 107 per second over the entire DNS.

The rate of change seen in records from the trace is not directly comparable with that seen in the other three samples, since the data sets are

Dataset	**Adam**	**ISC**	**RIPE**	**Trace**
Total RRSets	731	486	898	885
Inconsistent-TTL	8	7	0	36
Inconsistent-data	2	1	1	4
Dynamic/N-of-M	1	0	0	4
RRSets that changed	21	8	17	36
Total changes seen	262	8	126	55
Changes per RRSet per day	0.013	0.00061	0.0052	0.0023

Table 4.1: Results of the update probes.

not independent, but it is reassuring to see that they are of the same order of magnitude. Our figure is lower than the rate observed in [CK01] for the popular A records. However, that measurement did not distinguish between dynamically generated records and updates to static ones.

Figure 4.8 shows the overall distribution of changes by RRSet: most RRSets that changed did so only once over the period of the probes. All the RRsets with more then sixteen changes were SOA RRs, which changed only in their serial numbers.

4.4 Queries

Each root server handles about 8,000–11,000 queries per second [Wes04, LHFc03], so at a rough estimate the total load across the root servers is of the order of 100,000–150,000 queries per second[†]. Root servers are provisioned to handle at least three times their peak load [BKKP00]. The load at the gtld-servers, which serve `com` and `net`, is about the same.

The total load across all authoritative servers is harder to estimate; in 1992, one third of all DNS queries seen on the NSFnet were to the root servers [DOK92]. Adding the gtld-server query rate to the root-server rate (in 1992 the root server also served the gTLDs), we arrive at an estimate of 600,000–900,000 queries per second. This is an average rate of one query every 11 minutes for each host on the internet.

[†]Up to 70% of those queries are repeat queries caused by misconfigured clients, caches, firewalls and routers.

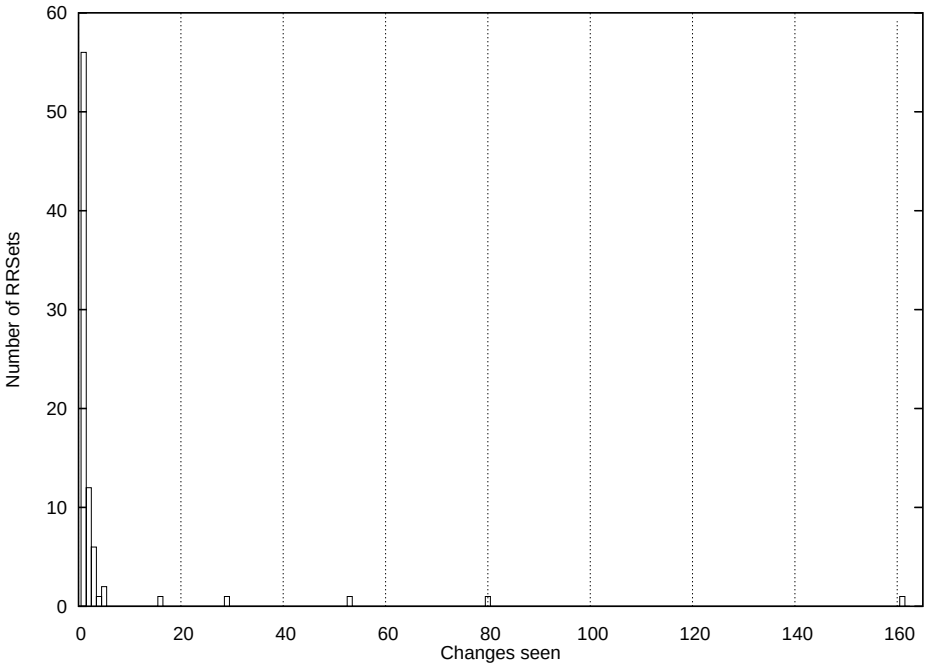


Figure 4.8: Histogram of number of changes seen per RRSet.

Name resolution at PlanetLab nodes completes in less than 100ms for approximately 85% of queries, and less than 10ms for about 75% of queries [PWPP04]. The mean lookup latency has been measured at 237ms [PWPP04] and 382ms [RS04]. The mean is dominated by the few high-latency lookups: the median is only about 40ms [CMM02, RS04]. These numbers include both the delays and benefits of local caching resolvers.

The latency of lookups to the root servers, measured over 27 days in December 2004 from a hosting centre in the UK, are summarized in Figure 4.9. For each server, we show the mean, a 95% confidence interval around the mean, and the median of the latency of queries. 0.05% of queries went unanswered, either because a UDP packet was lost or the server was overloaded; these are treated as having a latency of 5 seconds (the default timeout). The point labelled "All" gives the mean and median across all servers, as an estimate of the latency observed by a client that selects a server at random (e.g., dnscache) rather than

attempting to choose a "good" server as suggested in RFC 1034 (e.g., BIND) [Moc87a, WFBc04].

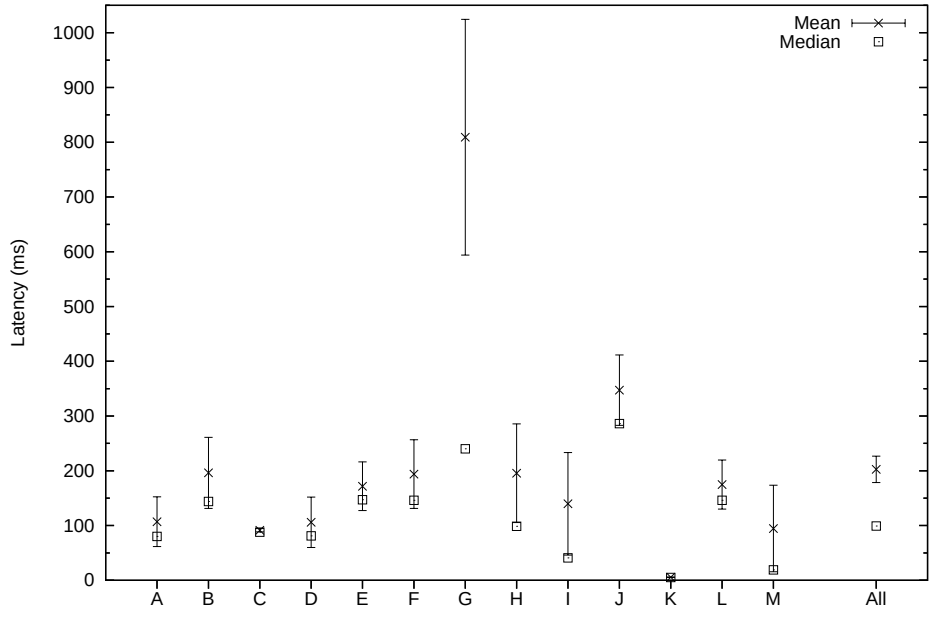


Figure 4.9: Latency of lookups to the root servers (as seen from the UK).

4.5 Availability

Because so many internet services require the DNS to be working, it is a requirement that the service should be resilient to failures in its components. It must also be resilient enough to withstand denial-of-service attacks. However, it need not be robust against attacks that would disable the underlying network or render it unusable for higher-level protocols. Park et al. [PPPW04] measured availability of DNS name resolution at 99%. The probes used in the update detection confirm this: they were able to find an answer 99.25% of the time.

It is hard to define precisely what this means as a specification for the system without a full understanding of the quantity and quality of attack traffic directed against the service, and collateral damage from attacks on

other services. Unfortunately, due to the distributed nature of the DNS, we don't have a good idea of the overall attack traffic (and even if we did, the attack patterns against a new kind of DNS would be different).

4.6 Requirements for a DNS replacement

In this section we summarize a set of minimum requirements for the DNS, as a benchmark for measuring any new naming system that is proposed as a replacement for it. Any such system should be able to meet this level of service, or have an argument for why the users of the internet will accept a degradation in name service.

Namespace. The service must support the tree-of-labels namespace of the DNS, including wildcards and aliasing, and the (name, class, type) indexing of data. (§2.1)

Datatypes. It must support the existing data types and their special handling rules. (Appendix C)

Database size. It must support a database of at least 1.8 billion resource records. (§4.2)

Administrative controls. It must provide for delegation of subtrees of the namespace to different administrative authorities, and for the current system of registry-registrar relationships. (§2.2)

Query rate. It must support an average query load of at least 900,000 queries per second and a peak load of three times that level. (§4.4)

Query latency. The mean lookup latency should be no worse than 382ms, and the median no worse than 40ms (including caching). (§4.4)

Updates. The administrator of a zone must be able to make updates to the zone contents. The service must support an average update rate of at least 0.0063 updates per RRSet per day (or equivalently, 0.0051 updates per RR per day). (§4.3)

Client protocol. There are millions of DNS clients and resolvers already deployed on the internet, using the existing lookup protocol. The service must support these clients unmodified, or come with a plan for how they will be upgraded. (§2.4)

Extensions. The service should allow extensions of the traditional role of the DNS, for example load-balancing. (§2.8)

Authentication and integrity. There must be mechanisms for validating the authenticity of records, and the integrity of communications. (§2.7)

Robustness to failure and attack. The service must be able to respond to at least 99% of queries on average. (§4.5)

4.7 The future

These requirements are only a description of the DNS today. As well as meeting them, a replacement DNS must be able to scale with the task in the future. Unfortunately, because we are not aware of any previous specification in this detail, it is hard to describe the rates of change of these requirements.

As a guide, we note that on average, since 2001, the number of hosts seen by the ISC internet domain survey has been 1.3 times the previous year's number. The number of second-level domains under the gTLDs has been on average 1.2 times the previous year's number, over the same period. This has fallen from the great expansion of the 1990s, when the average multiple was 1.8. It is well known that the equivalent ratio for the power of computers is between 1.4 and 1.5 [Tuo02].

4.8 Summary

In this chapter, we looked at various measurements of the current DNS and extracted from them a set of requirements that describe the job done

by the DNS. Next, we will address the design of a centralized nameservice which can meet those requirements, while eliminating much of the complexity of name lookups.

5 The design of a centralized DNS

In the previous chapters, we described the DNS, both in terms of its design and of the workload it currently supports. In particular, we observed that although queries are more frequent and less tolerant of error than updates, much of the complexity of the system is in the query protocol.

Here we present the design of a centralized nameservice, which uses the same wire protocol as the current DNS, but which moves much of the complexity from the lookup protocol to the update one.

5.1 Centralized architecture

Figure 5.1 shows the current DNS, and Figure 5.2 shows the architecture of a centralized DNS. The authoritative servers of the current DNS are replaced by a small network of central *nodes*, each of which contains a copy of every RRSet and a nameserver which answers DNS queries for them. A query can be sent to any server and will be replied to immediately using that server's local copy of the RRSets. The zone transfer system is replaced by a network of links between the nodes, along which updates are propagated to keep all the replicas synchronized. An update can be submitted to any server, which will propagate it to the other servers. All of the complexity of the distributed system is in the propagation of updates between servers, leaving the lookup mechanism simpler.

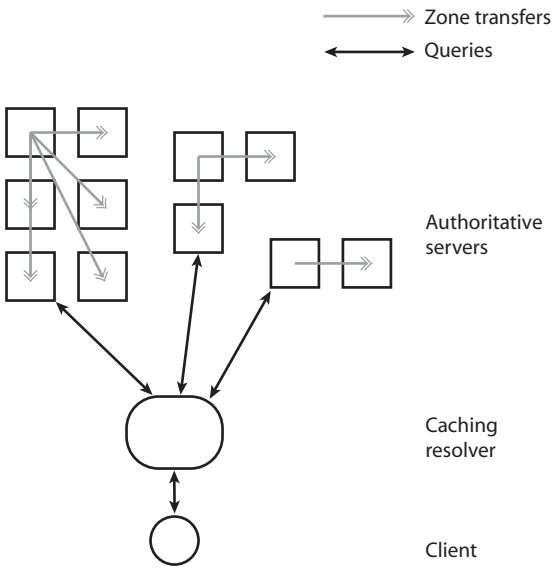


Figure 5.1: The current, distributed DNS

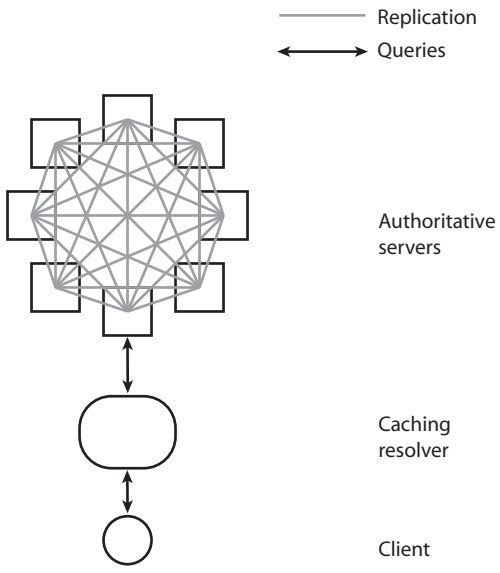


Figure 5.2: A centralized DNS

Clients

In the system described, client software does not change at all: although the mechanics of a lookup are simplified, no alteration is needed to either the wire protocol or the state-machine describing client behaviour. There are hundreds of millions of hosts already on the internet, most of which have stub resolvers in them; a system that did not work for those hosts would be unacceptable.

Clients contact the centralized servers in the same way they did before: using a configured list of known server addresses, which could be the same as the current root server addresses. For a partial deployment, the servers are advertised using NS records in the old DNS for the zones that are served from the new one. The caching resolvers of the current DNS are retained, not only because their function is still useful, but also because it would involve a lot of effort to remove them.

Zone administrators

Each zone administrator has a cryptographic certificate entitling them to make updates to records in their zone, and to issue further certificates for delegated sub-zones. Instead of being required to run DNS servers for the zone, they now publish records in the zone by sending a signed update to a centralized server. This removes a considerable technical burden from the administrator, since the software that submits updates can be much simpler than an authoritative DNS server, and does not need to run all the time.

Service nodes

Figure 5.3 shows the layout of a node in the service. It has three parts: a nameserver that answers questions from the public, a database of all the RRSets, and an update server. The update server accepts updates from the public and participates in a distributed update propagation scheme with the other nodes. We will describe them in more detail below.

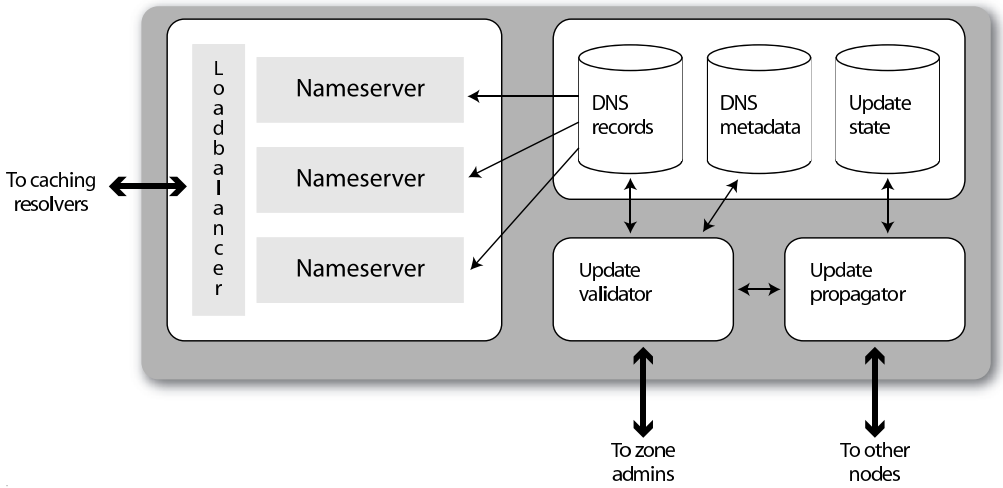


Figure 5.3: Node architecture

It is not necessary that this breakdown be reflected in the actual layout of the site. The database can be held on a single server, and it might share that server with the update mechanism. On the other hand the query load might require a cluster of DNS responders, with more machines as warm standbys for the ones in service.

Number of nodes

The service must be sufficiently distributed to ensure robustness, but no more than that. A very large number of nodes would both cost money and complicate the update mechanism, for little benefit. The more servers there are in the system, the greater the chance of one of them being isolated or of updates being delayed. In addition it must be possible to inform all the clients of the presence of the servers. Even with anycast this imposes an extra administrative overhead per node that we would like to avoid.

The deployment and management of the nodes should be similar to that of the current root servers:

- There should be about a hundred of them, to ensure reachability from all parts of the internet.

- They should be placed in well-connected places such as internet exchanges and Tier 1 ISPs, so that they can be reached quickly and easily by clients.
- They should be geographically distributed.
- They should be run by a number of competent technical bodies, independent of the namespace administration.
- They should use different software and hardware implementations of the service.

5.2 Nameservers

Each node of the service is a nameserver for the full DNS, so needs to have machines capable of holding all the records and responding to its share of the queries. A single server running the NSD nameserver can answer 45,000 queries per second, if the database fits entirely in memory. However, current nameservers are not designed to serve a database on this scale; NSD can only serve about 20 million records in 4GB of RAM. So we will have to load-balance queries across a cluster of servers.

Ideally, each server would be responsible for a particular subset of the DNS, in order to minimize the working set of each server. Unfortunately, the DNS does not lend itself particularly well to load-balancing in this way, because it requires responders to supply additional information along with the answer to a query, and that information may come from other parts of the namespace.

The naïve approach of using a hash of the queried name to choose which server to ask (similar to a distributed hash table on a local scale) is particularly unsuitable. A responder needs knowledge of the namespace surrounding the queried name as well as at that name: it must retrieve the closest enclosing zone (to supply SOA and NS RRs) and for unsuccessful queries it must search for an appropriate wildcard name, and for the nearest existing name in a DNSSEC-secured zone. So simply hashing the

query name would result in this context being repeated at every server that answers queries from a zone, as well as a number of extra hash lookups for unsuccessful queries.

Splitting the namespace into subtrees and assigning a subtree to each server solves this immediate problem. However, it requires adaptive reconfiguration at the load-balancer to make sure the load is spread evenly. Also, queries in one part of the namespace can require additional lookups in a quite different part. For example, an MX query will not only return the mail servers for a domain, but also look up each of their IP addresses, if they are available. Therefore, each server in a cluster must have the entire database available (or a mechanism for referring sub-queries for additional data to other servers). We choose to make the entire database available at every node. Each server is responsible for serving records from a subtree, and stores that subtree of the index in memory, so it can quickly perform lookups over the names it receives queries on. It also has the full contents of the DNS available on disk, with an LRU cache of the parts that are in use: this will contain the subtree for which it is responsible, as well as those parts of the rest of the DNS that are needed for additional-section processing. Responding to a query needs at most one disk read for each name involved (for example, the original name and the zone head). More popular records will not require a disk read at all, and if enough RAM is provided to hold both the index and the server's share of the RRSets, disk reads can be made rare occurrences.

Another possible approach is to pre-compute the answers to all possible successful queries, store them on disk and index them with a hash table. This would reduce the lookup time for successful queries significantly but increase the amount of space required to store the database. Unsuccessful queries would need to be handled in the same way as before, by walking the tree looking for wildcards or DNSSEC records. This method would introduce the risk of extremely expensive updates. For example, changing the IP address of a nameserver that serves many large zones would involve updating the stored response for every RRSet under all of those zones. Instead, we suggest putting a cache at the load-balancer,

which would give this sort of speedup for popular requests. Of course, the responders should be provisioned under the assumption that queries used in denial-of-service attacks will be arranged to be uncacheable.

Chapter 7 discusses the design of nameserver software capable of fulfilling this role. It presents a datastructure for serving large DNS databases, and evaluates it as well as some common alternatives.

5.3 Update service

The update service deals with requests from zone administrators to publish, update and delete their records. It verifies the signatures on the updates, and that the administrator has the authority to make the changes, using a tree of public-key cryptographic certificates. The tree of certificates follows the tree of the namespace, and matches the delegation of authority.

At each cut point, the parent zone's administrator signs a certificate containing the cut point, a validity period, and the child zone's public key. For compatibility with the old DNS, there must be appropriate NS records, but they are no longer used to redirect queries to the child zone's servers, since the records of parent and child are served from the same place.

When a zone administrator submits an update request, the update service can check that the administrator's certificate covers the correct area of the namespace, and follow the chain of certificates to verify its validity. The root certificate must be configured at every node, and other certificates can be propagated among the nodes, but the zone administrator can be expected to supply the full chain if needed. A revocation list can be published to the servers out of band.

The update service also communicates with other nodes, sending out the updates it has received from administrators, and receiving updates that were submitted elsewhere. The distributed update algorithm used between nodes will be discussed in Chapter 6.

The update service stores details of all ongoing transactions in the database, and makes the appropriate changes to the node's master copy

of the DNS when updates have been authorized.

5.4 Database

Each node needs a database, which will store its local copy of the DNS. The database needs to store at least 1.8 billion records, and their associated metadata (which will depend on the update algorithm); only the update module will be writing records, so there are no write-write concurrency issues apart from any between concurrent updates. An off-the-shelf database should be capable of filling this role.

The nameservers will need to receive updates to the records they serve but do not need to read the metadata. The authorization keys and certificates, and the details of current and past updates, are only used by the update module. The query load from the nameservers will only be affected by the rate of updates and the overall size of the database.

To make it easier to bring new nameservers online, the database might hold a copy of the DNS in a binary form that the nameservers can use directly; this can be kept up to date by the nameservers themselves.

Another possibility would be to have nameservers serve records directly from the database. However, this would require a much faster database, and the performance of nameservers that use database backends is poor (see Chapter 7).

5.5 Non-technical concerns

The political and bureaucratic mechanisms regarding administrative control of zones do not need to change at all in the new scheme. The same bodies control top-level domains, and the registry-registrar system does not need to change.

The only detail that changes is that, when a registrar assigns a zone to an administrator, the registry doesn't change the zone file. Instead, it issues a certificate for the appropriate length of time. When a delega-

tion expires, the registry deletes the child zone by submitting an update request to a node of the centralized service.

The management of a public-key infrastructure on this scale is non-trivial, but there is considerable operational experience from the operation of the X.509 certificate tree used for SSL, and best practice from that field will apply to this one [CFS+03, CCITT05b].

Some zone administrators have policy reasons for wanting to keep their zone contents secret, e.g., data-protection rules or security concerns. The centralized DNS requires that zone files be distributed only among the nodes; we do not propose that the nameservers support AXFR queries, since there is no need for them. The node operators can be constrained not to publish zone data by contractual agreements with the registries.

An important practical question is how the service would be rolled out and paid for. One possible scenario is this: a group of TLD operators co-operate to build the service, and use it first to publish their own zone files. A co-operative venture of this kind would be an opportunity to reduce running costs, and also to offer a new service to zone owners — migrating your zone to the same servers as its parent would give faster resolution, and there is already a market for highly available and well-placed nameservers [Ver05, Ult01]. Once the service is established, more customers (leaf nodes and TLDs) could be solicited. The operating costs would come from the yearly fees already paid for domain registration and DNS hosting, and the control of the system would be in the hands of the TLD operators (who are, we hope, trustworthy, since they are already in a position to do great damage to their clients).

It is usually suggested that a new DNS architecture could be incrementally deployed as a caching proxy in front of the old system's authoritative servers, with zone administrators gradually migrating to directly publishing their records in the new system [KR00, CMM02, RS04]. We reject this model on two grounds: firstly, it introduces yet *more* complexity into the system, with no guarantee that this "temporary" two-system arrangement will ever be removed. Secondly, it has no tie between the service provided and the people who pay for it. The cost of the intermedi-

ate caching layer scales with the number of clients, but the income scales with the number of zone publishers.

5.6 Discussion

The main argument for a centralized DNS is architectural: moving the complexity to where it can best be tolerated, and eliminating delay from the least delay-tolerant part of the system. But some implementation difficulties of the DNS are also solved by this scheme.

- Because the delegation mechanism is not used, none of the various ways in which it can go wrong are a problem: queries are not directed to servers that do not know the answers. Queries are answered with eventual consistency by all servers, because the synchronization between replicas is automatic. NS RRSets no longer have to be duplicated at cut points.
- Because all zones are served from the central servers, most "dual-purpose" (authority and resolver) DNS servers can be eliminated. This removes a number of security threats, and also the possibility of stale data being held on an ISP's servers after a zone has been re-delegated elsewhere.
- Having all of the DNS available at one place allows us to abandon the client-side aliases in use in the DNS (CNAME and DNAME) in favour of server-side aliases, which could be implemented more simply.
- In a central service, where all responses come from a small number of trusted servers, the opportunities for cache poisoning are greatly reduced. If there were no facility for zones to opt out of the service, then cache poisoning would be eliminated entirely.

Having a centralized DNS opens up some new opportunities for further enhancements.

- DNS measurement is easier. Having access in a simple way to measure the load on the whole DNS will be helpful for the design of future updates to the service. (Conversely, we need to ensure reasonable levels of privacy in a centralized nameservice.) Better DNS measurement can also help with measuring other parts of the internet, because DNS queries are usually caused by other higher-level actions: it is already possible to track flash-crowds and botnets using DNS statistics.

- Once the requirement for hierarchical service is removed, the requirement for a hierarchical namespace can follow. The only real requirements are some way of assigning responsibility for parts of the namespace, and a policy for access control. For example, the flat namespace required by HIP [MNJH05] is hard to implement in the current DNS but easy in a centralized system.

One drawback of the centralized approach is the reduction in flexibility:

- The domain administrator no longer chooses where the servers are. In the current DNS, a service offered only in one country might have all its nameservers in that country; in a centralized system there would be no such choice. This puts a greater emphasis on the need for caching resolvers to identify the closest server and send queries to it. This feature is not currently implemented in all resolvers [WFBc04].

- The centralized service deals only with *public* DNS service. Many organizations use the DNS protocols for their internal name service as well, serving a different set of zones within their own networks from the ones they serve publically. (We note that there is no particular requirement on them to use the DNS for this, and indeed it has been common in the past to use a different protocol, e.g. NetBIOS, for internal naming.) These organizations will have to keep their name servers (but no longer make them publically accessible) and will lose the benefit of the simple update protocol.

- Domain administrators are restricted in the amount of active content they can provide. Some domain administrators implement active load-balancing at the DNS resolution stage, by having the response depend on the IP address of the querier as well as the name and type of the query. This sort of table lookup could be supported by the centralized service. Some zones could be delegated in the old way, either if their requirements were too complex to be provided by the central servers, or as an intermediate step while new facilities were rolled out.

We believe that the relative simplicity of the design is worth the sacrifice of some flexibility. A more complicated design will be harder to implement correctly, less likely to have many different implementations, and less likely to work. In this desire for practicality we are following the authors of the original DNS [MD88].

5.7 Summary

We have presented the design of a centralized nameservice capable of replacing the DNS without changing the deployed base of clients and resolvers, or the delegation of administrative control over the namespace. By reducing the complexity of lookups it makes them faster and more reliable.

The new system must be able to meet or exceed the existing levels of service laid out in Chapter 4. In the next two chapters we will look in more detail at two aspects of the system where this will be challenging: the propagation of updates between nodes, and the construction of nameservers capable of serving the entire DNS.

6 Inter-site update propagation

The architecture proposed in Chapter 5 calls for a distributed, replicated database that allows updates and queries to be submitted at any node. Queries are always answered from a local replica of the database, and updates must be authorized locally and then propagated to the other nodes in the system.

Before looking at any particular propagation mechanisms, we will make our requirements more explicit:

- There are a modest number of nodes (about a hundred) in the distributed system.
- As far as possible, no node should be "more important" than any other node.
- Temporary disappearances are to be expected: nodes sometimes fail, and nodes sometimes cannot communicate with each other. On the other hand, the set of all nodes will be relatively static: commissioning and decommissioning of nodes will happen over long timescales (comparable to root servers). Since we also expect there to be relatively few nodes, it is possible for each node to know about all the others.
- Updates arrive into the system at an average rate of no more than a few hundred per second. They can be submitted at any node.
- Updates should be propagated quickly; generally they should reach all nodes in less than a minute. As we saw in Figure 3.1, most

zones in the current DNS are configured to check the synchronization between servers with a timer of more than five minutes. RFC 1996 suggests that when changes are made at the master server, it should notify its slaves; if it gets no immediate response it should use a 60-second default timeout before resending DNS NOTIFY messages [Vix96] (although we note that BIND 9.3.2 uses a hard-coded 15-second timer). Replicas of every object are held at every node, so updates must reach every node.

- There must be at least loose consistency between replicas. The current DNS allows the publishing of out-of-date records for a length of time defined by the expiry timer of the zone's SOA record, in addition to the TTL-based caching of records at resolvers. Figure 6.1 shows the expiry timer values seen in the RIPE and Adam data sets described in Chapter 4. About 98.5% of SOA records seen had expiry timers set to more than a day.

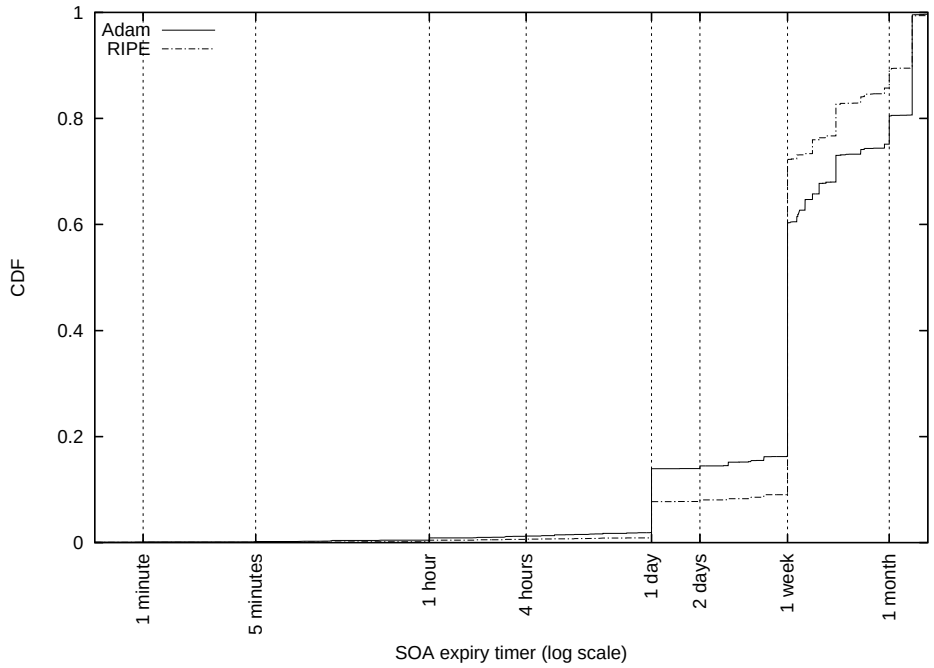


Figure 6.1: CDF of SOA expiry timers in Adam and RIPE datasets.

Some zones have very short expiry timers: for example, almost 0.1% of SOA records had the timer set to less than five minutes. This includes 4,350 records with the timer set to one second, and 1,891 with the timer set to zero. It seems likely that many of these records are configuration errors, but some zone administrators may genuinely prefer no service to five-minute-old data.

- There are few conflicting writes. This comes from the administrative structure of the DNS: each domain name has only one entity responsible for it. Shared responsibility is only possible within an organization (in the current DNS, among those with write access to the master zone file; in the new system, among different holders of the zone's signing key). Also, even within an organization, one entity tends to be responsible for each attribute of each name. Therefore we can assume that conflicts only occur in exceptional cases, and handle conflicting writes more slowly than non-conflicting ones. (There may of course be deliberately conflicting writes; the system must ensure that they do not unduly hold up *other* users' write requests. In general, since it would not be difficult to create a very high rate of updates, the system must make sure that one user's updates do not affect the service provided to other users.)

- Although writes are accepted at any node, they might not be balanced equally over all nodes. Some sort of load balancing or resource location service could be deployed to help clients find a local and unloaded node, but such systems are not always successful beyond avoiding pathological cases [JCDK01].

6.1 Using weak consistency

Because we expect few conflicting writes, and have such loose requirements on consistency, it seems appropriate to use an optimistic replication system. Saito and Shapiro [SS05] have categorized optimistic replication

schemes according to six design choices, and we will use their taxonomy here to describe a suitable mechanism.

First we define an *object* (the unit of data handled by the update system) to be an RRSet, as it is the minimum unit of data that can be described by a query or an update in the current system. Most of the operations in the DNS work at this level. Working at a finer granularity would only introduce complexity as various constraints are met — in particular all RRs in an RRSet must have the same TTL, and are covered by the same DNSSEC signature. Next we look at the six design choices.

Operations. We use *state transfer*: the only operation is the definition of a new version of the RRSet. If no previous version exists, this creates the RRSet; we use tombstones to indicate deletion of RRSets. The new version is stamped with the time of submission and the ID of the server to which it was submitted. This gives us a total order on timestamps (sorting first by submission time and then by node ID).

Number of writers. An update can be submitted at any node.

Scheduling. Operations are scheduled using a simple *syntactic* rule: operations with higher timestamps are defined to have taken place after those with lower timestamps. Because we expect few write conflicts, a complex resolution scheme for merging updates would have little benefit. Also, a "merge" operation on RRSets might create non-intuitive results, and it is not clear what action to take for unresolvable conflicts, since the original submitters of the updates would not be available to intervene manually.

Conflict resolution. Conflicts are resolved (or rather, they are ignored) using *Thomas's write rule* [JT75]: when an operation arrives at a node, its timestamp is compared with that of the last operation applied to the same RRSet. If the new operation's timestamp is more recent, it is applied. If not, it has arrived too late and is discarded. This simple scheme removes the need for a separate "commit" operation. When an RRSet is deleted, a tombstone must be kept at

each site until the delete operation has propagated to all sites. This can be done by expiring them after some suitably long interval (for example, Grapevine deleted tombstones after 14 days [BLNS82]). A node that is isolated for longer than that recovers its state by copying another node, rather than using the normal transfer of operations.

Propagation. Broadcasting all operations to all nodes of a hundred-node network would be wasteful, but we still want operations to propagate quickly though the network. Also, since not all nodes will be able to see each other at all times, operations need to be routed around failures.

A number of application-level reliable multicast schemes have come out of research into self-organising peer-to-peer systems [CJK+03]. As an example, we take SplitStream [CDK+03], a streaming multicast protocol that uses multiple, efficient trees to spread the load of multicast content distribution in a peer-to-peer network. By splitting each stream into "slices" and multicasting each slice over a different tree, and by ensuring that the set of forwarding nodes in each tree is disjoint from the others, SplitStream limits the loss incurred when a node fails. By using erasure coding at the source, it can allow several nodes to fail or disappear without losing data at the receivers. A node failure causes partial loss of a single slice (until the trees are reconfigured to cope), so if each receiver does not need all the slices, the failure is compensated for.

Consistency guarantees. The DNS does not provide read/write ordering between servers: a client that communicates with multiple servers can see updates "disappear" if the servers are not currently synchronized.

The DNS does provide *bounded divergence*, by having an expiry timer associated with each zone: if a slave server has not contacted the master after that time, it stops serving its copy of the zone. As we saw earlier, most zones have this timer set to more than a day.

The same level of bounded divergence can be provided by detecting when the update propagation is failing, and refusing to serve zones whose expiry timer is shorter than the oldest possible update that might be undetected.

If nodes that have no updates issue occasional "heartbeat" updates, it is simple to detect when a node has been disconnected from the network. If a single node is disconnected for a long time the other nodes can agree (by a pessimistic vote) that it is no longer part of the scheme and ignore it for the purposes of expiring records. This is safe so long as each node waits until it has told at least one other node about an update before acknowledging it to the client. More complex partitions of the nodes may require manual intervention.

Since there is a trade-off between how low this timer is set and how available a record will be in the centralized system, administrators who currently use a very low value of the expiry timer may need to opt out and provide their own service. We consider these special cases to be similar to other highly specialized zones, such as those performing arbitrary computation in response to queries.

Table 6.1 compares the current update propagation system (zone transfers) with what we propose for the centralized DNS.

The decision to use Thomas's write rule and state-transfer operations means that the service cannot directly implement the semantics of DNS UPDATE [VTRB97]. DNS UPDATE allows the updater to specify preconditions that must hold if the operation is to succeed, for example "The RRSet I am trying to register does not already exist." With the proposed system, two clients could make updates to the same RRSet at different servers, and both be assured that the RRSet does not (locally) exist. The conflict would not be noticed until some node receives both updates, at which time it will discard the earlier one, although the semantics of DNS UPDATE require the earlier one to succeed and the later one to fail.

Figure 6.2 shows an example of this: clients x and y are trying to use preconditions to synchronize their operations and ensure that only one of

	Old	**New**
Object	Zone	RRSet
No. of writers	Single-master	Multi-master
Operation	Operation-transfer (AXFR/IXFR)	State-transfer
Scheduling	n/a	Syntactic
Conflict resolution	n/a	Thomas's write rule
Propagation	Star topology, hybrid push/pull	Semi-structured, push (epidemic)
Consistency guarantees	Bounded divergence (by SOA timers)	Bounded divergence (by detecting partition)

Table 6.1: Properties of the old and new update propagation mechanisms

them writes to record N. Both x and y are told that their preconditions are true. When the conflict is later resolved, x's operation (which should have succeeded) is discarded, and y's (which should have failed) is kept: retrospectively, both x and y have been lied to.

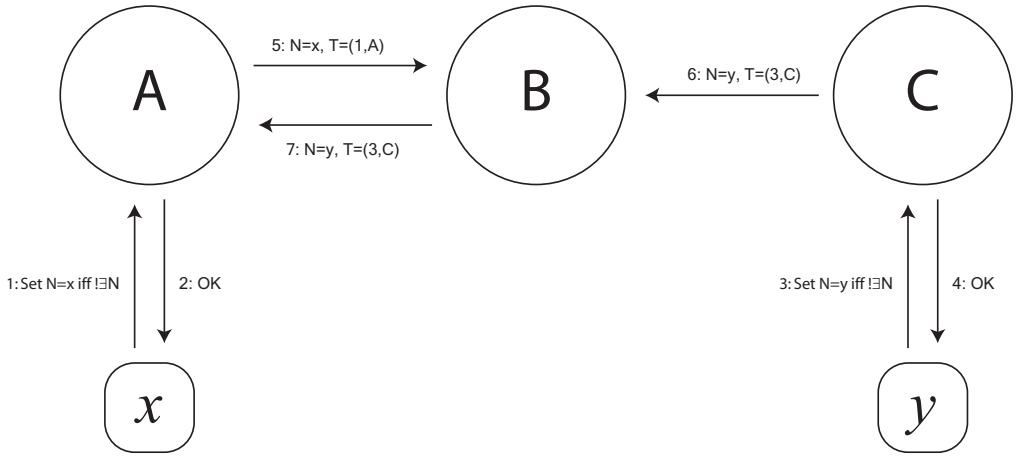


Figure 6.2: DNS UPDATE semantics are broken by state-transfer propagation.

By changing to an *operation-transfer* semantics, and propagating the precondition along with the update, the correct resolution could be enforced, but the conflict would still be detected too late — after y has

(apparently successfully) submitted his request and moved on to other things.

One way of resolving this problem is to require clients to synchronize their DNS UPDATE requests before they are submitted. This may be a reasonable requirement for zones where updates are already co-ordinated, for example when updates are sent from DHCP servers. Another way is to delegate administrative control of individual names to the entities that will use them, for example in a dynamic-DNS zone where each client has the right to update only its own record.

These two options are policy decisions, and have no effect on the implementation of the service: the first expresses policy as a voluntary code of behaviour among holders of the same capability, the other as a guideline for the entity handing out the capabilities. Since DNS UPDATE already relies on clients' good behaviour — there is nothing to stop a client simply omitting the preconditions in an update — this does not seem unreasonable.

The third option is to use pessimistic concurrency control, and not respond to a DNS UPDATE request until the operation is successfully committed and cannot be pre-empted by another update. In the next section we describe such a system.

6.2 Using strong consistency

In order to be able to tell a client with certainty that an update has succeeded, we need to provide a stronger consistency guarantee. We must know that an operation has been agreed on by the majority of nodes before it is reported as a success to the client who requested it. Operations are applied to entire zones: because the DNS UPDATE preconditions we are trying to enforce apply to zones, we cannot treat RRSets as the unit of propagation. Updates to different zones are orthogonal and may happen in parallel.

As an example of a pessimistic update propagation system we consider UFP [Ma92], an adaptation of the multi-decree Paxos protocol [Lam89]

designed specifically for use in a name service. Like Paxos, UFP uses a three-phase protocol to gather a *quorum*, or simple majority of nodes committed to an operation, before declaring it a success. Unlike Paxos, it does not require all operations to be mediated by an elected leader. Instead, under the explicit assumption that write collisions are rare, it allows any node to initiate a vote on an update operation.

In UFP, every node has a *co-ordinator*, which tries to get nodes to agree on updates, and a *participant*, which controls the local replica of the database.

The protocol executes in rounds of three phases. In the first phase, a co-ordinator gathers a quorum. The quorum members agree on the current state of the object to be changed (in our case, the serial number of the zone), and the sequence number of the current vote. In the second phase, the co-ordinator proposes a vote to the quorum. If that vote is passed by a majority of participants, the co-ordinator instructs the quorum to commit the operation to permanent storage in the third stage. The election rules for each stage are given in Appendix B; for full details and analysis see Chapter 5 of [Ma92].

In either of the first two phases, two co-ordinators may collide and one of them will fail to get a quorum; he may then time out his operation and try again from the beginning. As in Paxos, the operation being voted on in phase two may not be the one the co-ordinator wanted to vote on: if he has effectively stolen another co-ordinator's quorum in the middle of the voting process, he is required to propose the same operation as was under consideration. He may then start again with his own operation.

UFP only commits the operation at the nodes that voted for it; a secondary protocol is needed to propagate the committed operation to the other nodes in the system. This can be the same propagation method as used in the optimistic case, since operations being propagated have been approved by a majority of the nodes and are therefore known to be safe to apply.

Because updates are only ever applied by a quorum of nodes that are all working with the same version of the object, the DNS UPDATE rules

can be safely applied by the node proposing the update, and need not be checked explicitly by every node during the voting.

6.3 Discussion

The cost of weak consistency

Simply broadcasting the updates all-to-all costs $N-1$ messages per update, plus a further $N-1$ acknowledgments for reliability. Lost messages cause retransmissions on top of this.

The cost of forwarding messages over SplitStream is proportional to the number of neighbours each node has. For example, if every node is prepared to forward messages to sixteen others, and the messages are split into sixteen stripes, there will be $16N$ messages required to send a message, but each will contain $1/16$ of the original message. Using a coding rate of R (i.e. 1 byte takes R bytes to send encoded), that means a total of RN copies will be sent in $16N$ messages.

Although the number of bytes sent is the same, the number of links maintained in SplitStream is much smaller, and simulations in [CDK+03] show that the overall load on the network links between the nodes is very much lower. In addition, it deals better with unbalanced load: even if every update arrives at the same node, SplitStream arranges that each node sends only the sixteen messages per update it has been allocated, rather than one node sending all $N-1$ copies.

There is some extra cost for maintaining the SplitStream topology, but as we expect the nodes to be stable, this will be low compared to the data transmission costs.

The latency of propagation is proportional to the depth of the trees, which should not be more than two or three hops for the size of network we are describing. The hops themselves are kept short by the underlying Pastry [RD01] routing tables, which choose shorter paths where possible.

The cost of strong consistency

UFP operations on different objects can happen in parallel and, if a series of updates is being sent, the operations can be pipelined, requiring only two rounds of messages per update. The smallest number of messages that can be involved is $\lceil N/2 \rceil$ containing the update and $3\lceil N/2 \rceil$ smaller messages (votes etc.). This increases to $N-1$ copies and $3(N-1)$ small messages if all nodes are involved in the quorum. The small messages can in some cases be piggybacked on larger messages going the other way.

These messages are all either sent or received by the co-ordinating node, so the nodes do not get the load-balancing effect seen in SplitStream. There is a further cost of propagating the agreed update to the nodes that were not in the quorum. This will involve roughly one extra copy of the message per node that was not in the quorum, but can be arranged in a more balanced way. So the overall additional cost of the voting system is in three parts: the $3\lceil N/2 \rceil$ extra protocol messages, the extra latency of requiring a quorum to vote on an update before it can be distributed to all nodes, and the lack of load balancing.

If two updates collide, the cost of the second one is about tripled in the worst case, as the proposer is forwarded $\lceil N/2 \rceil$ copies of the first update and then proposes it again himself. There is some extra bandwidth cost to the non-colliding updates; this is not a problem in the optimistic case because we take no special action for colliding updates.

The latency of propagation is five hops, where each of the first four is the longest of all inter-node distances between the coordinator and the members of the quorum, followed by another delay of up to three hops while the updates are propagated to nodes not in the quorum.

In either case, there will be some additional cost in bandwidth for background checks: the servers must periodically sweep the entire database to double-check that their copies are in fact consistent.

6.4 Summary

We have discussed two distributed update schemes which would be suitable for propagating updates between nodes of the centralized nameservice. One is to optimistically multicast changes, which is cheaper; the other is to vote on them, which preserves the exact semantics of DNS UPDATE.

In the next chapter we will discuss the other challenge of such a large system: how to build a DNS server large enough and fast enough to support the load of the DNS.

7 Implementation of a high-volume DNS responder

The centralized design for a nameservice proposed in Chapter 5 calls for each node serving the DNS to have a responder large enough to handle the entire contents of the DNS and fast enough to respond to a high volume of queries. In this chapter we discuss the design of such a responder and evaluate our prototype implementation.

7.1 Index of domain names

There are three common lookup methods used in open-source DNS servers. (We do not address the datastructures used in commercial servers as the source code is not available to us.) The first, and most common, is a balanced binary search tree [MS05b] using a comparison function that implements the DNS's ordering on names. This is used in BIND, NSD and many less-popular servers. It allows support for DNSSEC's authenticated denial by allowing a failed lookup to return the closest preceding name that does exist.

The second is to store all RRSets in a hash table [MS05b], indexed by the query name and type. This allows for very fast lookups but requires some special handling for failed lookups, such as explicitly searching for the enclosing zone or wildcard. It does not support the "closest preceding name" query type needed by DNSSEC at all. This is used in many caching resolvers (which do not need to do these tree-walking lookups) and also

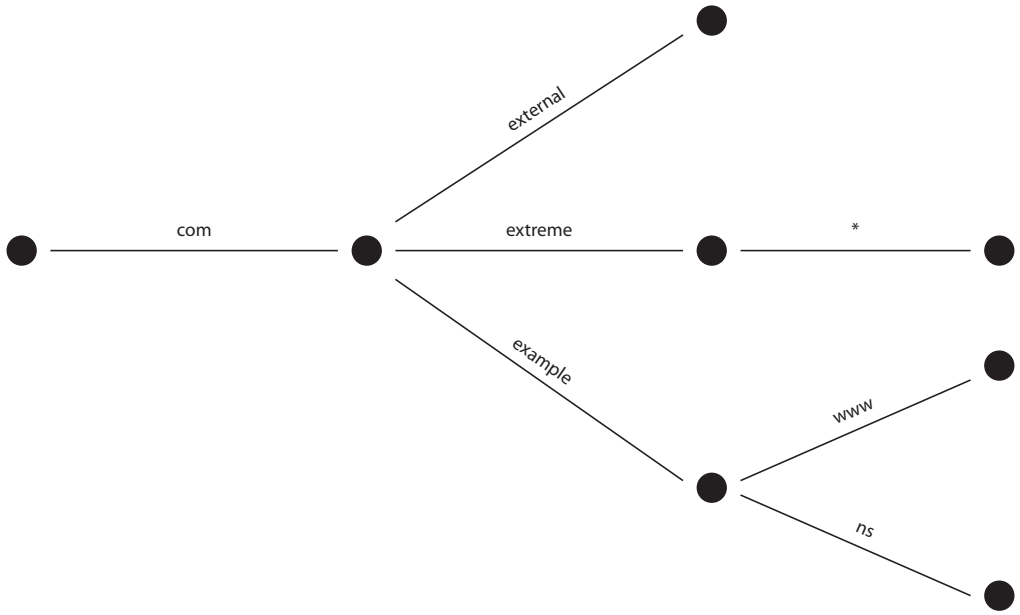


Figure 7.1: A fictional part of the DNS namespace

in the tinydns authoritative server, which does not support DNSSEC.

The third is to offload the data lookup to a general-purpose database, using a standard interface such as SQL or Berkeley DB. Using a database back-end allows administrators to integrate their other network management tools with their DNS zones without needing to export zone files. This is available in PowerDNS and BIND, although in both cases the server's built-in red-black tree gives better performance (see below).

For a DNS responder that has to handle extremely large numbers of domains, we have chosen a different data structure: a radix trie [MS05b]; specifically, a 256-way radix trie with edge information stored at each node.

Figures 7.1 and 7.2 show an example of part of the DNS namespace, and a trie holding the same names. In order to preserve the same ordering on DNS names as required for DNSSEC, we must first reorder the labels, so `www.example.com` is entered in the trie as `com|example|www|`, where `|` separates labels and is ordered before all other characters. Also, any upper-case ASCII characters are converted to lower case. With these

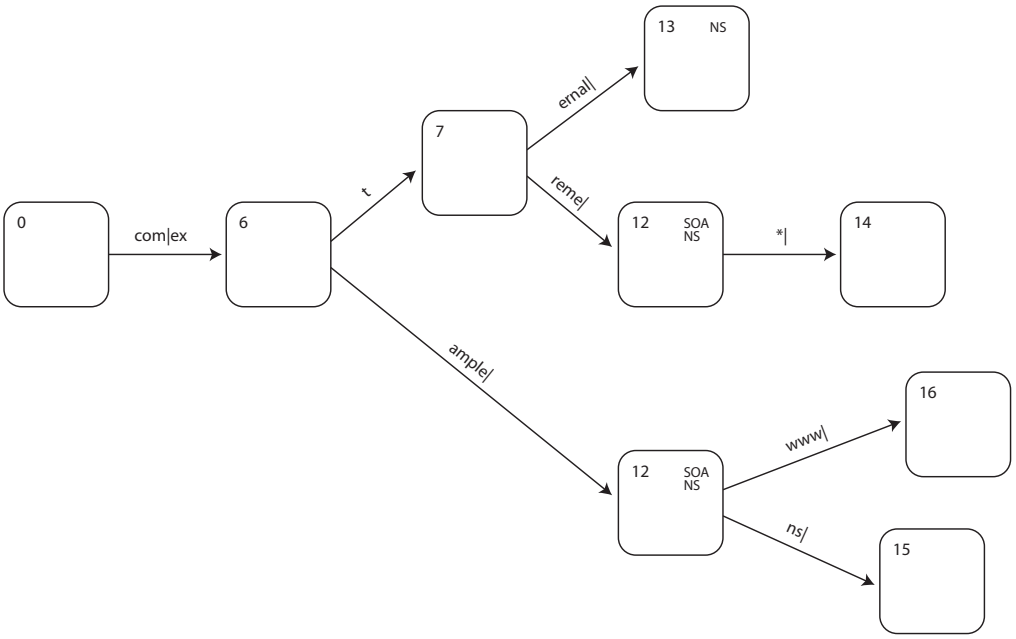


Figure 7.2: The same names arranged in a radix trie

changes, the DNSSEC ordering is the same as simple lexicographical ordering on converted names[†].

Each node of the trie is marked with the index into the key of the character that a lookup should branch on when it reaches that node. Also, each edge is marked with all the characters where there are no branches; in this way lookups can check all the characters of the key, not just the ones where there are branches in the trie. (A lookup for `com|different|mail|` would follow the edge marked `com|ex` and, by comparing the edge string to the relevant part of the key, know that there are no keys in the trie that start with `com|di`, and give up the search.)

We note that there can be nodes in the trie that have no corresponding node in the namespace (e.g., the node representing `com|ext` in Figure 7.2), and nodes in the namespace without corresponding nodes in the

[†]Bit-string labels [Cra99a] could be supported by writing them out one bit at a time, using representations for one and zero that are ordered before all characters except the separator. However, since bit-string labels are being withdrawn from the standards due to interoperability difficulties and lack of demand [BDF+02], we have not included them in the prototype.

trie (e.g., the node representing `com` in Figure 7.1).

The radix trie has the following advantages over a binary search tree.

- Like the binary tree, a failed lookup can easily return the closest preceding name. However, with a few small modifications, lookups can also return the enclosing zone or delegation point and a wildcard-location hint in a single pass through the trie. Details of this are given below.

- A balanced red-black tree with N entries needs up to $2\lceil\log_2(N)\rceil$ comparisons to complete a lookup. (For successful lookups this number will be lower if the key happens to be stored higher up in the tree; for unsuccessful lookups it must be at least $\lceil\log_2(N)\rceil$). The radix trie has a higher branching factor and so will have shorter lookup paths. This is offset by the fact that the trie cannot be reshuffled for better balance; it is only as balanced as the data being stored. This can be seen as an advantage for a large public service: imbalances in one zone's data do not affect the trie depth for other zones. Figure 7.3 shows the maximum and average depths of lookup paths through tries indexing up to 20 million lines of zonefiles (taken from the RIPE data). For scale, $\lceil\log_2(N)\rceil$ and $2\lceil\log_2(N)\rceil$ are shown.

- A binary tree requires a full comparison between the search key and the key at each node as it proceeds down the tree. A radix trie requires only a comparison of the edge data for each edge followed: each byte of the key is looked at only once. As the tree becomes larger, and therefore deeper, this makes more of a difference.

The shape of the trie follows the shape of the DNS's namespace tree. A delegation point in the DNS is represented as a sub-trie of the trie, and a DNS zone is a contiguous area of the trie. This means that we can reuse the trie structure to follow the DNS structure instead of having to store that structure separately. More interestingly, in the process of looking up a name, we pass the top of its enclosing zone. This means that by marking

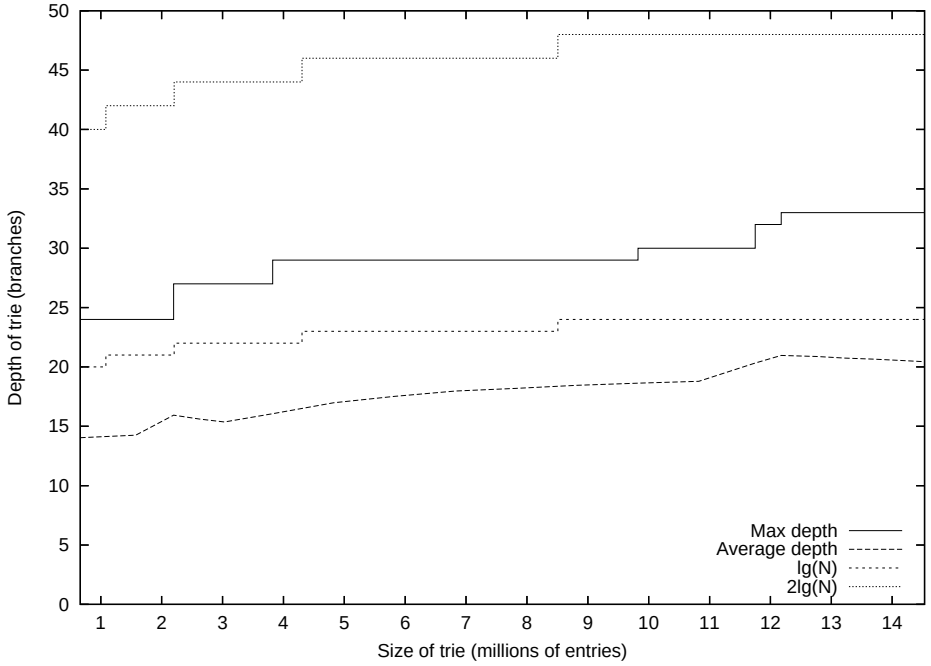


Figure 7.3: Depth of radix trie vs number of entries

nodes where SOA or NS RRSets are present, and remembering the last marked nodes we saw during a lookup, we can recover the zone head information at the same time as doing the lookup. The lookup algorithm is still very straightforward. Algorithm 1 gives a pseudo-code version of it, and a full implementation is given in Appendix A.

This algorithm returns the RRSets for the query name (if any exist) and also those for the closest enclosing zone and cut point. If the lookup fails, we can look for a suitable wildcard as well. The wildcard rules for the DNS make this more subtle than looking for zones. The server must back up along the key to the "closest encounter" (the last node that it passed in the namespace tree), and overwrite the rest of the key with $*$. This gives a key for the "source of synthesis": any RRSets found under the modified key can be used to synthesise answers to the original query[†].

[†]The source of synthesis is a domain name of the form `*.a.b.c`, and is a wildcard that matches any name of the form `x.y.z.a.b.c`, so long as none of `x.y.z.a.b.c`, `y.z.a.b.c` or `z.a.b.c` exists in the namespace. The advantage of this scheme is that it means there is only one possible wildcard that could match any given query [Lew06].

Algorithm 1: Basic lookup algorithm.

```
node := root
repeat
    last_branch := node
    if node is flagged "SOA" then last_soa := node
    if node is flagged "NS" then last_ns := node
    if node is flagged "DNAME" then
        return the DNAME RR and rewrite the query.
    node := node → child [key [node → byte ]]
    if node is null then fail
    if node → edge does not match the key then fail
until we have used all the key
```

The zone head is stored at last_soa, and the cut point for a delegated zone is at last_ns. If the lookup succeeded, the RRSets for the queried name are at node. If the lookup failed before the key was used, we report a name error (NXDomain); if it failed and all the key was used, we report a "no data" error (NoError).

Note that the closest encounter is a node in the namespace tree, which means it is at a break between labels. It does not necessarily coincide with a node in the lookup trie. The server might back up past several trie nodes, since there can be a node at each character of the key. Also, there is not guaranteed to be a trie node at the closest encounter.

For example, in the first zone fragment in Figure 7.4 a lookup of the name `trial.example.com` should be answered from the RRSets of `*.example.com`. In the second, although the tries have the same shape, a lookup for `trial.x.example.com` should fail. Algorithm 2 shows how this can be implemented in the radix trie.

In looking up `trial.example.com` in trie A of Figure 7.4, the server follows `com|example|tr`, finds that it cannot complete the lookup, and then backs up to the previous '`|`' at `com|example|`. It adds `*|` to the key at that point, and proceeds to find the wildcard. In trie B, the failing lookup of `trial.x.example.com` has passed one more '`|`' so does not back up as far. It looks for a wildcard with the key `com|example|x|*` and does not find one.

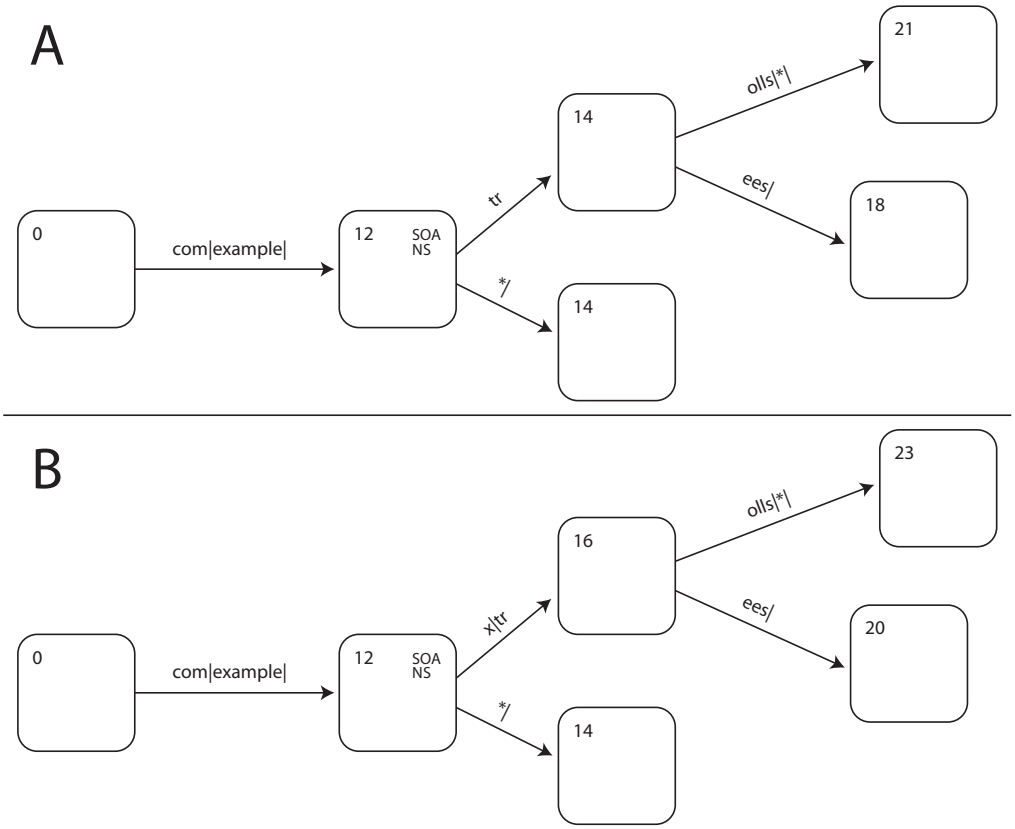


Figure 7.4: Examples of the DNS wildcard algorithm

If the zone is secured with DNSSEC, a failed lookup needs to find the previous entry in lexicographical order, where the relevant NSEC RR will be. Intuitively, we need to head up the tree until the first chance to go one branch to the left (or the top of the zone) and then go down and to the right until we meet a record: see Algorithm 3. We can precompute some of this on the first pass by remembering the last node where the branch we took was not the leftmost one. This removes the need for upward pointers in the trie.

Support for NSEC3 [LSA05] will require additional data structures as the hashed queries used for denials deliberately do not follow the tree layout of the namespace. If a query fails, the server must hash the query name and find the name with the closest preceding hashed value, which will have an NSEC3 RR covering the extent of the hashed namespace that

Algorithm 2: Wildcard lookup algorithm.

```
byte := (last_branch → byte) – length(last_branch → edge)
while key [byte] matches last_branch → edge do
    byte += 1
while key [byte] is not '|' do
    byte –= 1
length(key) := byte + 3
key [byte + 1] := '*'
key [byte + 2] := '—'
```

The key is now the key for a lookup of the source of synthesis. Re-start the search; we can start at the zone head because it must be above (or at) the closest encounter:

```
node := last_soa
repeat
    node := node → child [key [node → byte ]]
    if node is null then fail
    if node → edge does not match the key then fail
until we have used all the key
```

Algorithm 3: DNSSEC lookup algorithm.

```
node := last_branch
repeat
    char := key [node → byte ]
    if node has any children with index less than char then
        set node to the one with the largest index
        break
    if node has any records stored at it then
        return node
    if node == last_soa then
        break
    node := node → parent
while node has any children do
    Set node to whichever of its children has the largest index.
return node
```

includes the hashed query. A binary tree stored at the zone head, and mapping back from hashed names to their original owners, would suffice.

Updates

Because of the size of the database, it would be unacceptable to have to recompile the datastructure every time a change is made. The trie described above has been designed with the intention of allowing it to be updated with little effort. Zones can be split and delegated simply by adding the relevant records and flags to the trie. Adding wildcard records does not require any extra housekeeping in the trie. Because we use a trie instead of a balanced tree, there is never any need to re-balance the structure.

Names can be deleted entirely from the trie by removing at most two nodes (see Algorithm 4).

Algorithm 4: Name deletion algorithm

Find the node corresponding to the name, using Algorithm 1.

if node *has more than one child* **then**
  return
if node *has one child* **then**
  Prepend node → edge to child → edge
  Update parent → child [] to point to child instead of node.
  return
else
  Delete node from parent → child []
  if parent *now has only one child, and no data of its own* **then**
    Recursively delete parent. (This is guaranteed not to recurse again, since parent has one child.)

7.2 Implementation

The data structures and algorithms above were implemented twice: once in C and once in Objective Caml (OCaml).

The first version is an optimized, memory-efficient C implementation. It consists of just the data structure and algorithm; for the evaluation below, the rest of the DNS server (network service, packet marshalling, additional-records rules, zonefile parsing, etc.) was taken from NSD 2.3.1.

The OCaml version is intended as a reference implementation, giving the details that are skipped over in the pseudocode above, and is much more readable than the C one. It is the core of our implementation of a full authoritative DNS server, including a zonefile parser and the logic for assembling full responses from queries, written entirely in OCaml. Code for marshalling and unmarshalling DNS packets was provided by Anil Madhavapeddy, as part of his work on high-performance type-safe internet applications [MS05a]. The entire OCaml server will be released as open-source code in the near future. The OCaml implementation of the radix trie is included as Appendix A.

Since we plan to keep the entire index trie in memory, we need to have an efficient way of storing the nodes. Figure 7.5 is a histogram of the fanout of nodes in a trie populated with about 14 million domain names taken from the RIPE probe data. Most of the nodes have relatively low fanout; less than 1% of nodes have more than eleven children. The highest fanout seen in this trie is 52. In the C implementation, we embed the edge strings and child tables directly in the nodes. Nodes with fewer then eight children have a simple array of edges; larger ones have a two-level lookup table. Because all of the data for a node is held in one place, we can use small offsets within the nodes rather than pointers, and the nodes have good locality of reference.

In the OCaml implementation, child tables are represented as arrays of pointers which are automatically grown to fit the number of children. OCaml does not allow the programmer the same control over data layout; indeed the garbage collector is allowed to move objects around at run-time, and makes heavy use of pointers.

In order to reduce the memory footprint at run-time, the OCaml implementation uses hash consing [Fil00] to merge identical strings and string lists. There is only one copy of any character string in the database

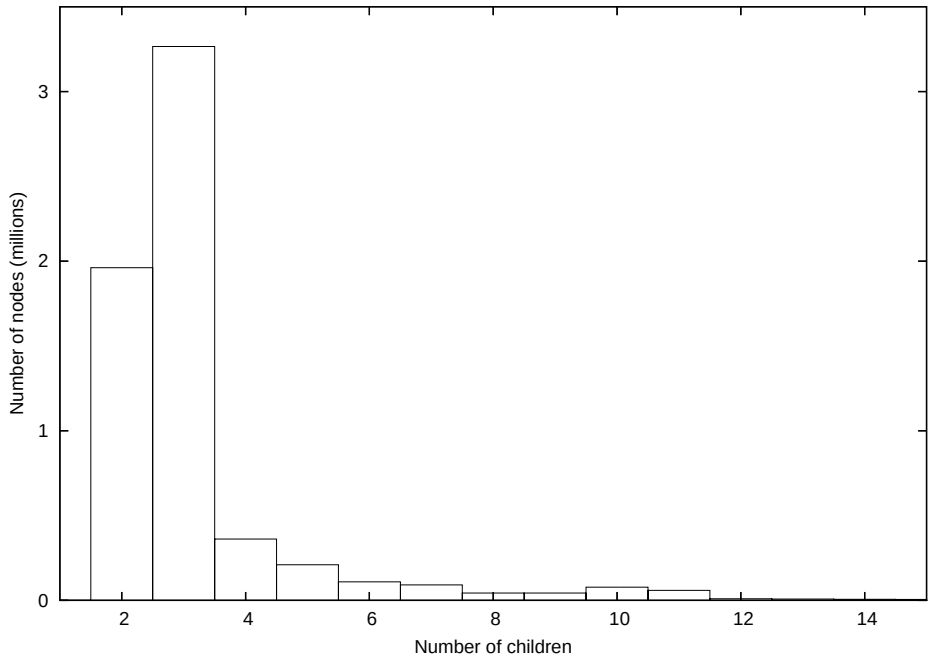


Figure 7.5: Histogram of node fanout in a populated radix trie

at a time: this means that for example, the string “com” is not repeated for every domain name in the database. This scheme actually requires more memory at zone loading time, because it uses a hash table to ensure uniqueness, but the hash table can be paged out once loading is finished, since it is only needed when updates are being made.

7.3 Evaluation

Figures 7.6 and 7.7 show the performance of the trie-based server against version 2.3.1 of NSD and version 9.2.3 of BIND for different query sets. In both figures the server was loaded with RRs taken from the RIPE dataset of `uk`, and run on a dual-processor 2.4GHz AMD Opteron with 4GB of RAM, running Fedora Core 3 with a Linux 2.6.9 kernel. The servers were constrained to use only one processor, while the queryperf program ran in parallel on the other CPU to test how quickly the server could answer queries. Queryperf ships with the BIND nameserver and is designed to

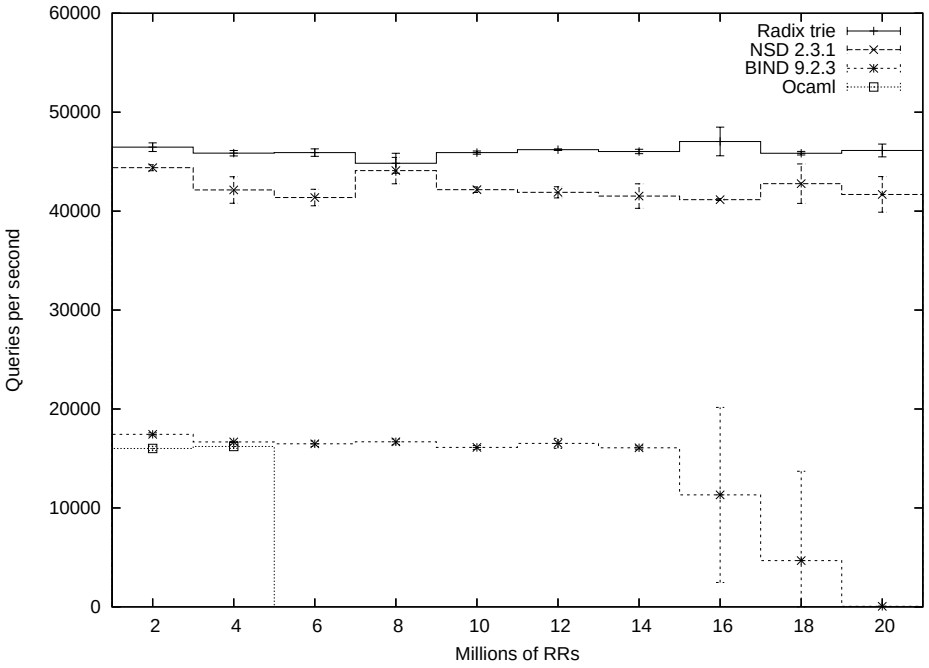


Figure 7.6: Performance vs database size: 2.5 million successful queries

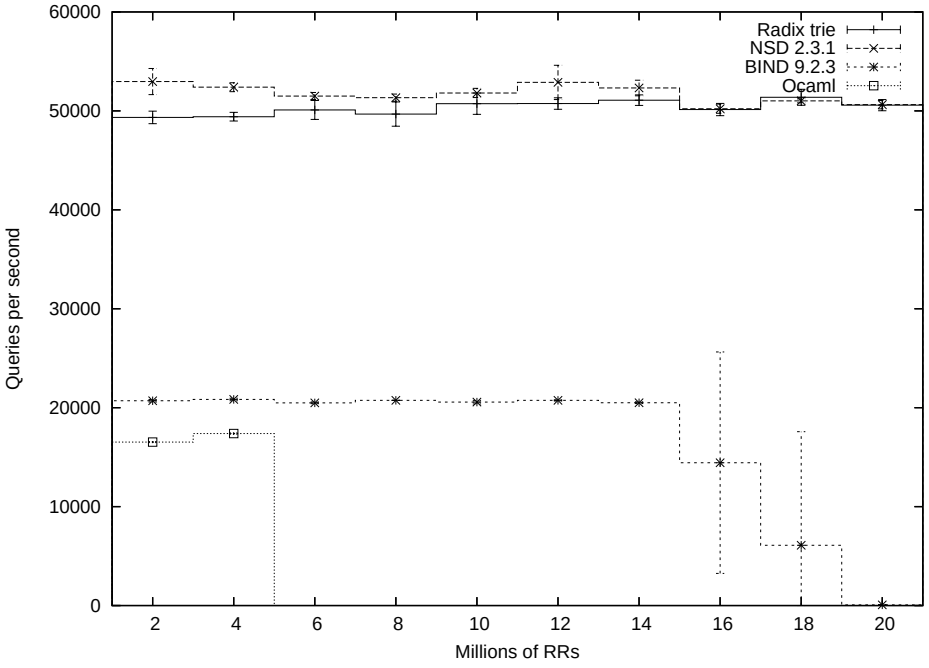


Figure 7.7: Performance vs database size: queries from UCL trace

measure nameserver performance by sending queries as quickly as they are answered. In all cases, the server was warmed up with a pass through the query set, and then queryperf was run for five tests of five minutes. The error bars are at 95% confidence intervals around the mean. Results from testing over a LAN were almost identical to those given here.

In Figure 7.6 the queries used were 2.5 million unique RRSets from the same data that was being loaded onto the server — that is, for every point above 2 on the horizontal axis, all queries were for data that was present at the server. The trie server is about 8% faster overall for these queries. NSD spends 22% of its time doing red-black tree lookups (55% in the kernel, 23% on everything else), so the speedup of the trie itself over the red-black tree is significant.

Figure 7.7 shows the performance for the set of all unique queries in the UCL trace file described in Chapter 4. In this case there were only 525,324 queries, and 76% of them were not present in the database.

The trie server is slower than NSD for these queries, because of the way it handles negative searches. It is using the algorithms described above, while NSD uses some optimizations based on the assumption that it will serve static data. Since the namespace tree will not change, various hints about where wildcard and DNSSEC records will be found can be precomputed at zone loading time and stored in the database. The radix trie, on the other hand, has the ability to change parts of the namespace without recomputing the entire index.

In both of these graphs, we show the performance of the OCaml server as well. Unsurprisingly, it is nowhere near as fast as the optimized C of NSD. Also, because OCaml makes heavy use of (aligned) pointers, its memory footprint on the 64-bit test machine is very large; loading data-sets larger than 4 million records was not practical on a 4GB machine with 8GB of swap.

We have also included BIND 9 for comparison, as BIND is the most popular nameserving software in use[†]. BIND is very much slower than

[†]According to various online surveys, e.g.: `http://mydns.bboy.net/survey/` or `http://dns.measurement-factory.com/surveys/200504-full-version-table.html`

NSD, and at about 14 million records the data set becomes too large for it to handle. BIND is faster than the OCaml server, although for smaller datasets it is slightly slower.

We also measured BIND 9 using a PostgreSQL 8.1.3 database backend. The database was loaded with 2 million records, but BIND could not serve all of them because it uses a connection to the database per zone and quickly runs out of available connections. Therefore we used queries for only the 27,481 records that BIND would serve. Over five runs of five minutes, the server answered an average of 23 queries per second ($\pm$0.55).

Since the BIND plugin is clearly not intended for reliable production use (indeed some effort is required even to compile it) we also evaluated a dedicated SQL frontend server, MyDNS, which claims to be "very fast and memory-efficient". MyDNS gets most of its speed by caching both SQL queries and entire DNS responses. With caching disabled its performance is even worse than BIND's SQL frontend, taking almost a tenth of a second per query. With caching enabled, it answered 15,665 queries per second from a warm cache ($\pm$505), which is worse than BIND's red-black tree.

Figure 7.8 shows the performance of the servers when loaded with 60 million records from the RIPE survey, and queried about different subsets of those records. The queries are contiguous subsets of the loaded data, shuffled randomly and then sent as queries to the server. Once again, the error bars are at 95% confidence after five runs. Both the original and modified versions of NSD can serve up to 20 million records on this machine before the working set becomes larger than memory and the server starts to thrash. The trie version is faster, as expected, and the benefit increases from about 9% to about 16% as the load increases. BIND was not included in this test as 60 million records is far more than it can handle on our test server.

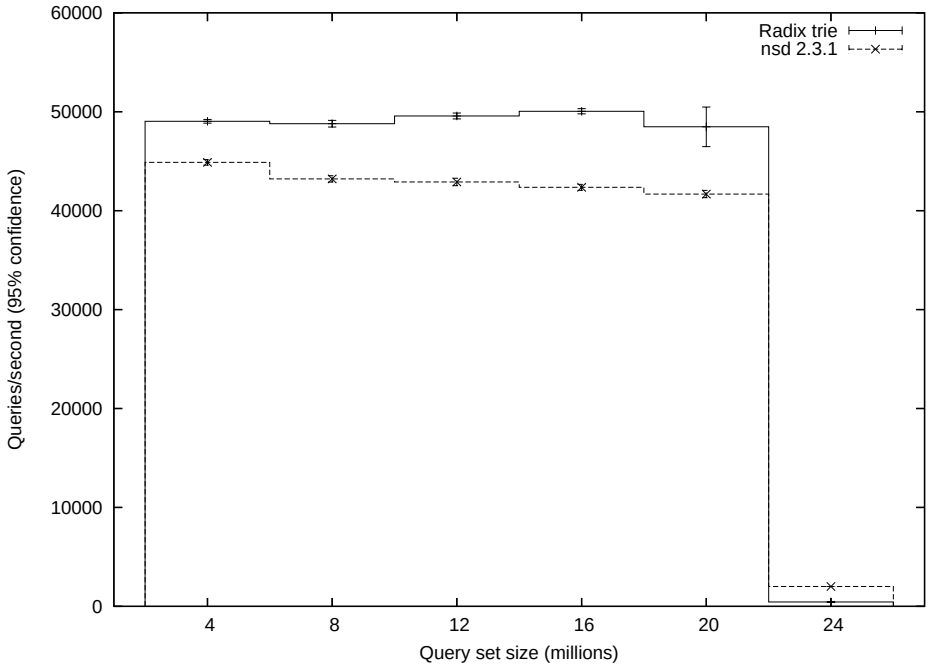


Figure 7.8: Performance vs size of query set

7.4 Discussion

In Chapters 4 and 5 we laid down some requirements for our nameservers. Let us revisit them now.

Database size 1.8 billion resource records would take up about 360GB of storage. If we want to be able to answer queries from RAM, this means we need to load-balance, for example across six machines with 64GB of RAM each

Query rate The peak query rate of 2.7 million queries per second is spread across all the nodes of the service, though not necessarily evenly. If each server can handle 50,000 queries per second, then the six servers per node we already suggested can handle 300,000 queries per second. Then the peak load can be handled with only nine nodes in service.

Cost Although such a cluster would be expensive to build and to run, the requirement is only that it be no more expensive than the current system, which contains more than 1.4 million servers. Even if many of those servers are shared with other tasks that is still a considerable investment. The difficulty is one of economics, namely of collecting the money in one place.

Over the last three chapters, we have described how a centralized nameservice could be built to replace the DNS. In the next chapter, we will discuss related work in the area, before summing up in Chapter 9.

8 Related work

There have been a number of projects suggesting changes to the DNS architecture, as well as the ongoing work of developing new features for the existing DNS and of measuring its behaviour. We summarize some of that work here.

8.1 DNS standards development

The IETF working groups on DNS Extensions (dnsext) and DNS Operations (dnsop) continue to develop and standardize new features for the DNS, and to issue best-practice documents on operational matters.

DNSSEC [AAL+05a, AAL+05c, AAL+05b] is the proposed authentication mechanism for RRSets. It is a public-key infrastructure where each zone has an associated signing key that is trusted to sign RRSets in that zone. That key is signed by another key-signing key for the zone, which in turn is signed by the parent zone's key. A resolver wishing to authenticate RRSets it receives can follow the chain of signatures until it reaches a key it knows and trusts.

In theory, this trust anchor will eventually be a widely published key-signing key for the root zone but, for now, other methods are needed to distribute trusted keys to resolvers. One such mechanism is DLV [AW06, Vix05], which allows the distribution of DNSSEC keys from a point other than the root of the DNS, allowing early rollout without having to solve the thorny problems of root key maintenance until there is more operational experience of DNSSEC. Operational procedures for key mainten-

ance and rollover in signed DNS zones are being developed in the IETF and other fora, such as the RIPE DNS working group.

It is likely that yet another version of DNSSEC will be needed before wide adoption: the current mechanism of signing the "gaps" between RRSets (to allow authenticated denials without online signing) allows resolvers to enumerate all RRSets in a zone, and this is unacceptable to many zone administrators. NSEC3 is a new mechanism being developed to avoid this problem by hashing all the valid query names first and signing the gaps in the *hashed* zone [LSA05].

The other direction in DNS standards development is defining new record types for new applications of the DNS. This is often tied to the development of new protocols that require some nameservice element (e.g., IPv6), but also includes extensions to how naming data are used in existing protocols. For example, there are currently various schemes that attempt to reduce unwanted email by blacklisting misbehaving servers or whitelisting "good" servers[†], or by using DNS records to indicate which servers are allowed to emit emails from particular domains [MS05c] (this is roughly analogous to MX records, which indicate the servers that may receive mail for each domain).

Most of this work is orthogonal to the sort of architectural changes proposed in Chapter 5. A new record type with special handling rules would require some changes to a DNS responder, but would not affect the mechanisms for submission and distribution of records within the centralized DNS. Any DNSSEC scheme that allows off-line zone signing will be compatible with a centralized scheme.

8.2 DNS surveys and measurement

Traffic analysis

Quite a lot is known about the traffic at the root servers; in particular it is well known that a lot of it is unnecessary. Danzig et al. [DOK92]

[†]An index of many of these DNS-based lists is at `http://rbls.org/` .

argued in 1992 that the proportion of packets on backbone links that were DNS lookups was higher than it ought to be, even ignoring the effects of caching. They analysed traces taken at `a.isi.edu` and showed that 95% of the traffic at that root server could have been avoided by better-written software and smarter retry rules. More recently, Wessels and Fomenkov [WF03] have analysed traces taken at the F-root servers and shown many of the configuration errors in clients and resolvers that cause unnecessary traffic at the root servers. Most of the errors uncovered in those papers are in resolvers, although some are in hosts that should not be sending traffic to the root servers at all.

Wessels et al. [WFBc04] analysed DNS traffic in several academic networks and in laboratory experiments to describe the behaviour of different implementations of DNS caching resolvers, and the effect that the implementation decisions have on traffic to root and TLD servers.

Zone contents

The RIPE NCC have performed a monthly host-count of the RIPE-area ccTLDs since October 1990 [RIPE]. Their collector machines recursively zone-transfer all zones under the relevant TLDs, and the collected zones are analysed to provide statistics about DNS usage. Despite having a long-standing project and actively requesting zone transfer access from ISPs, they typically collect transfers of only half of the zones they see, because server administrators commonly deny zone transfers as a security measure.

ISC have host count data going back to 1981 [ISC]. In 1987 they started performing recursive zone transfers starting at the TLDs. In 1998, because server admins were refusing zone transfers, they switched to walking the `in-addr.arpa` zone of mappings from IP addresses to names. Their current probe walks any allocated netblocks for which it cannot get zone transfers. They produce a listing of `PTR` records in `in-addr.arpa` every quarter.

Our Adam survey described in Chapter 4 used the same method as the

RIPE count probe, and as the old ISC probes, to gather 23.4% of the zones under the most popular gTLDs.

Edge measurement

Liston et al. [LSZ02] looked up a list of about 15,000 RRSets from 75 different locations on the internet, and compared what they saw. They concluded that the best improvement in DNS performance would come from reducing the latency of lookups to servers other than the root and TLDs, although they also found that TLD server placement meant that lookup latency depended on the client's geographical location. These findings support the thesis of this dissertation: removing the large number of lower-level servers would increase the performance of the DNS.

As part of the CoDNS project, Park et al. [PPPW04] performed a survey of query latencies at caching resolvers around PlanetLab [PACR02]. They found that there were significant delays in the caches, and that the delays were independent of the queries. Their diagnosis is that DNS caching resolvers are usually on machines that perform other duties (mail server, web server etc.) and those other duties occasionally starve the DNS cache of resources. This leads to internal delays that occur in bursts and have nothing to to with the rest of the DNS architecture. They suggest that these delays have probably introduced noise into other DNS surveys. Their response, the CoDNS network, is described below.

Pang et al. [PHA+04] measured the reachability of caching resolvers, which they identified from the logs of Akamai servers, and of authoritative servers, which they identified from web cache logs. Over two weeks, they found that generally, availability was high: 95% of caches and 95% of authorities were reachable more than 95% of the time. Also, availability was slightly positively correlated with the load on the server — that is, both busy caches and popular authorities were more likely to have high availability.

Bugs and misconfigurations

Broido et al. [BNc03] give an interesting analysis of erroneous updates to the reverse mappings of RFC 1918 IP addresses. These used to arrive at the root servers but are now mostly diverted to dedicated servers in AS112[†]. By using spectroscopy on the inter-arrival times of these update packets they find that the major cause is unhelpful default settings in Microsoft Windows 2000 and XP DNS software.

Wessels [Wes04] identifies a set of common errors in network, host and resolver configuration, which are responsible for useless DNS traffic at the root servers. He even provides a tool to help network administrators detect these problems on their networks.

Pappas et al. [PXL+04] present a survey of configuration errors in the delegation of zones, based on about 52,000 zones. They found that 15% of the delegations inspected were lame in at least one server, 45% had all their nameservers in the same /24 subnet, and 77% had all their servers in the same AS. These errors would be eliminated from the DNS by centralizing it and removing the large number of authoritative servers.

Ramasubramanian and Sirer [RS05] raise the issue of transitive trust in zone delegations. The security of a zone's service depends not only on that zone's nameservers, but on the servers that are authoritative for the *names* of those nameservers, and on the servers that are authoritative for *their* names, and so forth. They calculate that, on average, a zone can be hijacked by compromising 2.5 servers in this tree of trust, which need not necessarily be controlled by, or even known to the zone's administrator. For 30% of zones, this vulnerable set is made entirely of servers that are running out-of-date and known-vulnerable software. Again, removing the delegation of service from the DNS would solve this problem.

[†] `http://public.as112.net/`.

8.3 New architectures

A number of projects and proposals have been published in the last few years for major architectural changes to the DNS, with the aim of solving the problems discussed in Chapter 3.

Replication

Kangasharju and Ross [KR00] advocate replacing per-zone secondary authoritative nameservers with a set of replica servers, each holding a full copy of the DNS. Each server would be responsible for managing updates to a subset of the zones in the DNS: it would pull copies of its zones from their primary servers and propagate them to the other replicas via multicast IP or satellite channels.

They calculate the probable size of the DNS by multiplying the number of hosts in the ISC host count (56 million in 1999) by an estimate of the amount of DNS data each host should need, and conclude that 10 GB should be enough (small enough to be held on a single hard drive). Using this estimate they calculate that a 2Mbit/s global multicast channel would allow each record to change 11 times per day. They do not discuss any details of how the service would be implemented.

Although this proposal is similar to what we describe in this dissertation, it has a number of differences. Kangasharju and Ross want to keep the existing set of primary authoritative servers; their new system is a replacement for secondary servers and caches. They replace the zone-transfer system with a more widely distributed multicast or satellite channel. This leaves all the problems of wide distribution and lame delegations in place. They do not present any details of implementation or evaluate their design against real DNS data.

LaFlamme and Levine [LL04] suggest using a similar scheme, but only for the root and top-level-domain zones. These zones would be replicated widely to mitigate the effects of denial-of-service attacks on the root and TLD servers. They measured the changes in the `com`, `net` and `org` zones

over a month in 2004 and found that the daily updates came to less than 500KB. Querying the NS RRSets of 2,266 second-level domains every four hours for six months, they found that 70% of them had not changed over that time, and 93% of them had at least one NS RR the same. They conclude that it would be reasonable for caching resolvers to fall back on a replica of the TLD zones if queries to the TLD servers timed out. They describe a distribution mechanism for TLD updates implemented by publishing the daily differences to a USENET group.

Handley and Greenhalgh [HG05] suggest another similar scheme, using a custom peer-to-peer content distribution network to push the TLD zone contents. This scheme is also intended as a fallback mechanism to be used after normal DNS queries fail.

Malone [Mal04] considers widespread replication of the root zone as a means of lightening the load on the root servers. He suggests that each caching resolver should act as a secondary server for the root zone. Resolvers would then send one SOA query to a root server every six hours and transfer only the changes that are made, instead of sending one query per TLD every four days. In an experiment using some academic caching resolvers (serving about 1000 users) it was found that the traffic to and from the root servers was about the same using zone transfers as it had been before, although it involved fewer packets. Malone's experiment used AXFR; it seems likely that his results would be more encouraging using incremental transfers. Unfortunately, the scheme requires the cooperation of resolver administrators throughout the internet, and so would only reduce the quantity of well-behaved traffic at the root servers; but the traffic at the root servers is mostly caused by ill-behaved resolvers [WF03], which would be unaffected.

Peer-to-peer lookup schemes

The DNS could be served from a peer-to-peer (p2p) overlay network. It is an attractive target application for p2p systems, since it is a large and useful distributed lookup system, and widely distributed lookups are what

p2p overlays do well. Also, the robustness and scalability of p2p networks are desirable properties in a nameservice. The data, and the load of serving it, could be spread across many relatively lightweight servers, and new servers could easily be added to increase capacity.

In 2002, Cox et al. [CMM02] evaluated using the DHash distributed hash table (DHT) to provide DNS-style nameservice. They simulated a nameservice based on DHash/Chord, using lookups taken from traces of academic networks, and concluded that the DHT-based service would be more fault-tolerant and better load-balanced (assuming all nodes to be equally capable of serving data). However, they saw a marked increase in lookup latency over the native DNS, since each record is held on a small number of nodes and queries must be routed through the overlay network, travelling $\mathcal{O}(\log n)$ hops in a n-server network.

The CoDoNS project [RS04] aims to address the latency issue by replicating popular (i.e. frequently-read) records more widely through the DHT, thus reducing the *average* latency of lookups. Less popular data are left with longer read times. (Also, the latency improvement is only felt by those clients whose lookup patterns match the global popularity distribution [PMTZ06].) Update latency is also higher for more popular records, because the extra replicas must be reached. CoDoNS has been implemented and deployed on PlanetLab; in [RS04] a 75-server network was measured serving 47,230 records at a rate of 6.5 queries per second divided uniformly among the servers (i.e., 0.087 queries per second per server). The mean latency was 106ms and the median 2ms, and the network used 12.2KB/s per node including all queries and update traffic. Equivalent numbers for a dedicated 75-server "legacy" DNS testbed serving the same query rate were not supplied. Instead, the numbers were compared to the measured latency of the same queries submitted to the public DNS, which had mean 382ms and median 39ms (including one extra mandatory network hop, since the clients were co-located with CoDoNS nodes but not with their caching resolvers). The PlanetLab deployment of CoDoNS is currently available for public use as a caching resolver, relying on the normal DNS to answer queries that are not in the system.

While membership of the CoDoNS overlay is restricted to "official" servers, avoiding the question of peers' trustworthiness, Awerbuch and Scheideler [AS04a, AS04b] suggest some schemes whereby a nameservice could be made based on a peer-to-peer network that is open to all comers. They describe two mechanisms by which such a network could be made immune to arbitrary adversarial peers joining the system, provided that the adversaries are "in a sufficient minority".

These peer-to-peer systems do not address the central issue of the complexity of lookups — rather, they replace one complex lookup mechanism with another one. Even with aggressive caching, many queries will need to be redirected from server to server; our centralized system, by contrast, answers every query at the first server.

In addition, although the peer-to-peer networks are proposed as replacements for the authoritative servers, the implementations described have been of caching resolvers. The authors do not suggest how the DNSSEC and wildcard algorithms would be efficiently implemented in a hash-based lookup scheme. (We observe that it might be possible to implement DNSSEC using an order-preserving hash function [FCDH90] instead of a uniform one, but that would introduce another trade-off into the system: uniformity of query load across the nodes is traded for uniformity of database size.) Additional-section processing would also require lookups to be redirected between nodes before the answer is returned.

Pappas et al. [PMTZ06] compare the recursive hierarchical lookups of the DNS with flat DHT-based lookups. They show that the hierarchy in the DNS allows it to outperform DHTs because DNS clients can use cached routing data from one lookup to help in other lookups. They also show that although DHTs are less vulnerable to orchestrated attacks than the DNS is, they are more vulnerable to random node failure. They conclude that a DHT could achieve the current DNS's level of performance and failure-resistance only at a higher cost, and suggest that the extra effort would be better spent in hardening the existing DNS.

New caching mechanisms

Jung et al. [JSBM01, JBB03] argue that the DNS's TTL-based caching is limited by the Zipf-like popularity distribution of records. Using trace-driven simulation of DNS caches, they show that eliminating caching entirely for A RRSets would increase DNS traffic by only a factor of four, and that 80–90% of the cache hit rate of the DNS would be achieved with TTLs of about 15 minutes.

Cohen and Kaplan [CK01] suggest allowing caches to keep stale records, and provide them in response to lookups while re-fetching them in the background. This would eliminate the lookup delay for queries where stale answers are in the cache. Their example is a unified web cache and DNS resolver, where an incorrect speculation can be caught before its effects are seen by the client. The cache would proceed with re-fetching the RRSet and at the same time attempt to fetch the web page using the old RRSet, only forwarding it to the client once the refetch completed and it was known that the RRSet had not changed. Using trace-based simulation they show that using stale DNS records has a 98–99% success rate for this kind of lookup.

CoDNS [PPPW04] responds to delays and faults in caching resolvers by replacing the stub resolver library with one that times out more quickly in the face of failure, and that can route queries to caching resolvers other than the ones configured at its local site. Each CoDNS client is part of the CoDeeN peer-to-peer content distribution network, and is given the ability to send DNS queries to its neighbour nodes in CoDeeN, which will pass on the queries to their local caches. If a node's local caching resolvers do not respond to a query within 200ms, it starts using remote caches in parallel with the local ones. This smooths out temporary delays in caches by timing out more quickly and works around long-term issues with caches by providing a wider pool of caches to work with. In tests on PlanetLab, CoDNS clients routed 18.9% of their queries to remote caches and, for 34.6% of those queries, the latency of going to a remote server was lower than that of waiting for the local query to finish [PWPP04].

Building on top of the DNS

The papers discussed above address the implementation difficulties of the DNS. There are also proposed naming systems that tackle its semantic shortcomings by building other naming services on top of it. In particular these proposals address naming services for higher-level protocols, where the DNS's hierarchical namespace and implied one-to-one name-to-host mapping are unhelpful.

The Web Services community has a series of naming schemes starting with the DNS, and adding Uniform Resource Identifiers (URIs) [BLFM05], then a Web Services Definition Language (WSDL)[CGM+04], which maps from the name of a service to the URIs that provide it. This is followed by a Web Services Addressing Specification [Box04], which defines a service in terms of both URIs and WSDL specifications.

The Resource Locator Service [EF01] proposed to identify objects in the internet using versioned opaque identifiers, with a DHT-like lookup layer that would provide objects with mobility and permanent names. Semantic Free Referencing [WBS04] is similar: URLs would be replaced by signed opaque identifiers; lookups would involve resolving the semantic-free name to a tuple of lower-level naming information (for example, a URL) and then resolving names from that tuple in the DNS. Again the motivation is that these opaque identifiers would be permanently attached to web objects, independent of the objects' locations. The argument is that that this would remove the tendency for URLs to become out of date as web publishers reorganize or move their sites. Also, since the identifiers have no inherent meaning, there would be no need for the political difficulties involved in DNS delegation disputes (leaving only the usual difficulties of a public-key infrastructure for verifying the signatures).

Balakrishnan et al. [BLR+04] take this idea further and suggest a three-layered naming scheme, with Service Identifiers at the top level, attached to internet services. Each Service Identifier would resolve to a set of Endpoint Identifiers, indicating the locations where the service is available and possibly also gateways through which queries to those en-

dpoints should be routed. Endpoint Identifiers would in turn resolve to network addresses. The main difference between this and the Web Services naming hierarchy mentioned above is that the Service and Endpoint Identifiers would be opaque, drawn from flat namespaces and resolved using peer-to-peer lookups.

The Host Identity Protocol [MNJH05] is another cryptographic flat-namespace architecture, intended to give mobility to IP endpoints rather than objects.

Replacing the DNS entirely

Building a new nameservice from scratch would allow a lot of freedom to design a service that is suited to the needs of the DNS's current or proposed future users. Unfortunately, the DNS is too far entwined in the software at every point on the internet to hope to extract it now, so any plausible strategy will have to interoperate cleanly with the clients of the current DNS or, at least, leave the DNS in place for those who cannot use the new scheme.

The Universal Name Service [Ma92] is a general naming service for distributed systems. It uses a two-tier system, where each directory is controlled by a small pool of "first class" servers which accept updates, and also served from a larger number of "second class" servers whose replicas are read-only, and may be out of date. The UNS uses a hierarchical naming scheme, where each node is assigned an opaque, globally unique identifier, rather like a filesystem inode number. These opaque identifiers are used to allow reorganization of the namespace tree, including recursively encapsulating subspaces within each other. The UFP protocol discussed in Chapter 6 was designed for synchronizing updates between first-class replicas in UNS.

Active Names [VDAA99] extend the ideas of active networks into naming: each name lookup would involve following a chain of programs in various resolvers around the network. Each program would perform arbitrary computations on the request to transform it into another request

to be forwarded to another server. The programs themselves would be identified by their own (also active) names, to allow them to be loaded dynamically into the servers. There would have to be a small set of "bootstrap" programs that would be present in every server to help them load the other ones.

8.4 Summary

We have described the DNS development work that is going on in the IETF, some measurement projects, and some of the other designs for improved DNS architectures. Much of it is orthogonal to our centralized design, except that a centralized architecture would make it easier to roll out changes across the whole DNS.

Both replicating the DNS and removing the current hierarchy of servers have been suggested independently, but not as part of a single design and not with the aim of centralizing the service. Since there has been no quantitative description of the task performed by the current DNS, previous architectural proposals have not been evaluated against it. We note in particular that the difficulties of implementing DNSSEC and additional-section processing over a distributed hash table have not been addressed.

In the next chapter, we will conclude the dissertation with a brief discussion of the changing circumstances that have made a centralized nameservice into a plausible idea once again.

9 Conclusion

In this dissertation, we have argued that a centralized nameservice is capable of replacing the current DNS, as well as solving many of its difficulties.

We described the operation of the current DNS, and gave a set of requirements that any replacement must meet. We argued that many flaws in the DNS come from the misplacement of complexity in the current system, and proposed an architecture for a centralized DNS that would change this without requiring the clients of the DNS to change. We discussed how updates would be handled in this new system, and how a large server might be built that could serve the entire DNS.

In conclusion, we believe that this centralized DNS is capable of handling the load on the current DNS with lower latency, fewer configuration errors, and less risk of failure.

9.1 Changing tradeoffs

The size and shape of any distributed system depend on a number of tradeoffs that change over time. The architecture of the system must change with them — for example, the world-wide web is distributed for load balance and administrative reasons but, after five years, it developed a series of centralized index sites.

When the DNS was designed and deployed, some of the tradeoffs were as follows.

- Manpower was more important than bandwidth or latency: the task

of approving updates to the database had to be distributed. This has not changed; indeed it would be ridiculous today to imagine a scheme where one administrator managed the entire namespace.

- Processing capacity was more important than bandwidth or latency: the task of serving the records was best spread across many systems. This tradeoff has changed as processor power has become cheaper. The task of serving DNS records has not become any more difficult, nor have the zones grown in size, so the number of zones that a single server can serve has grown over time. This makes it possible once again to build a server that knows the entire DNS.

- The latency of a few extra hops was not important considering the timescales of processes involved. This has also changed. As bandwidth and processing power increase, making the other parts of an internet transaction faster, the latency of a DNS lookup is becoming more noticeable [HA00, CK00].

Because the tradeoffs have changed, the architectural choices that are possible have changed with them, and a central nameservice is once again a reasonable choice. Of course, in the future, these tradeoffs will change again.

- As the number of computers and people attached to the internet grows, the number of names in the DNS will increase. Assuming that the average rate of change stays constant, this may drive the update rate so high that it overwhelms the update propagation mechanism. We may need to abandon our design choice that every node knows every record, and fall back to something more like Grapevine [BLNS82]: every zone is replicated only at a subset of servers, but all servers hold the index of which zones are where.

- Likewise, as the number of internet-connected machines increases, with ubiquitous computing and smart appliances becoming more

popular, the load of queries may become too much for a small number of nodes to handle. In this case we may have to abandon our design decision that every node accepts updates, and fall back to a two-tier service more like UNS: a second class of servers hangs off the primary servers, serving all the records but only accepting queries from the core set of primary servers.

In either of these cases, we still believe that the current widely distributed system would be a worse choice.

9.2 Future directions

There are, of course, a number of questions that have arisen during the writing of this dissertation, that we have been unable to address. We list some of them here, as possible starting points for future work.

Changes to caches and clients

In this dissertation, we have restricted ourselves to changes to the authoritative servers that do not require any change in the clients or caching resolvers. However, there might be some benefit to be had from relaxing this requirement. What features could be enabled by small changes to the caching resolvers? What useful metadata could be added to responses to help caches in the new centralized system? Would it be possible to implement a smarter, more explicit version of the DNS-based load-balancing tricks that are in use today?

We know from [CK01] and [JBB03] that the TTL-based caching scheme is not optimal, and from [PPPW04] that timers and retry algorithms in caches cause noticeable delays. Is there a better caching scheme that could be incrementally deployed in the DNS? (And, if changes are to be made at every DNS-aware machine in the internet, would it be better to abandon the DNS altogether and start again with a clean slate?)

Other approaches

There have been a number of architectural changes proposed for the DNS in recent years [KR00, CMM02, LL04, HG05, RS04, AS04b]. It would be interesting to evaluate them against the quantitative requirements we listed in Chapter 4.

Data structures

The data structure described in Chapter 7 is efficient for lookups but needs locking to allow safe concurrent updates. Because lookups can backtrack in the trie, either reads must be disabled while updates are made, or some sort of double-buffering is needed to allow updates to be made to one copy while lookups are done on the other. Can we use techniques from lock-free data structures to allow safe concurrent updates to the trie?

Protocol development

Even if a system such as we have described is never built, are there parts of it that can be useful to DNS protocol development? The principle that complexity in the DNS should be in the updates rather than the queries is one that could help in the design of new protocol extensions.

Also, some intermediate stage between today's wide distribution and our centralized model could be useful. The main advantages of the scheme are only achieved when parent and child zones are aggregated onto the same server, which suggests that per-TLD centralization would be a possible middle ground.

Other protocols

The argument about placement of complexity in a distributed system can be applied to other internet protocols, even if the particular solution we suggest is not appropriate. Not all protocols have the relative uniformity of deployment that the DNS has, and not all have the option of separating service from administrative control. HTTP, for example, is used for a

wide range of services far exceeding its original intended use. NFS or RSS are more promising protocols for investigation. They are services which are still used mostly for their original purposes, and where there is some flexibility in the way the workload is shared between systems. Unsurprisingly, they are also both areas for which peer-to-peer alternatives have been suggested.

Measurement

There is already a lot of DNS measurement work going on. However, although we know a lot about caches and root servers, we know relatively little about the authoritative servers lower down the delegation tree. How much work are they doing? How is the load spread among them? What sort of attacks are being launched against them, and with how much success?

9.3 Summary

We will end by repeating the three points that we made in the introduction, and have discussed in the preceding chapters:

- The complexity of the DNS should be moved from the lookup protocol into the update protocol, because there are many lookups and comparatively few updates.
- A centralized nameservice of about a hundred nodes, each having a full copy of the namespace, would be an effective way of achieving this.
- A nameserver node capable of holding the entire DNS can be built using the data structure and algorithms we have described.

A Implementation in Objective Caml

The following OCaml functions implement the data structure and associated algorithms described in Chapter 7. The C version is about five times the size, and a great deal less readable.

Support for DNSSEC takes up approximately one sixth of the code. In particular, the differing attitudes of the wildcard and the DNSSEC algorithms to internal “empty non-terminals” causes some duplication of effort.

```
(*
   dnstrie.ml – 256-way radix trie for DNS lookups
   Copyright (c) 2005-2006 Tim Deegan <tjd@phlegethon.org>
*)

open Dnsrr;;

(*
   Non-standard behaviour:
    – We don't support '\000' as a character in labels (because
      it has a special meaning in the internal radix trie keys).
    – We don't support RFC2673 bitstring labels. Could be done but
      they're not worth the bother: nobody uses them.
*)


type key = string;;                    (* Type of a radix-trie key *)
exception BadDomainName of string;; (* Malformed input to canon2key *)

(* Convert a canonical [ "www"; "example"; "com" ] name to a key.
   N.B. Requires that the input is already lower-case! *)
let canon2key string_list =
  let labelize s =
    if String.contains s '\000' then
      raise (BadDomainName "contains null character");
    if String.length s = 0 then
      raise (BadDomainName "zero-length label");
    if String.length s > 63 then
      raise (BadDomainName ("label too long: " ^ s));
    s
  in List.fold_left (fun s l → (labelize l) ^ "\000" ^ s) "" string_list
```

```
(* A "compressed" 256-way radix trie, with edge information provided
   explicitly in the trie nodes rather than with forward pointers.

   For more details, read:
   Robert Sedgewick, "Algorithms", Addison Welsey, 1988.
   Mehta and Sahni, eds., "Handbook of Data Structures and Applications",
     Chapman and Hall, 2004. *)

module CTab = Hashtbl.Make (struct                                    40
  type t = char
  let equal a b = (a == b)
  let hash a = Hashtbl.hash a
end)

type nodeflags = Nothing | Records | ZoneHead | SecureZoneHead | Delegation
and dnstrie = {
    mutable data: dnsnode option;    (* RRSets etc. *)
    mutable edge: string;            (* Upward edge label *)
    mutable byte: int;               (* Byte of key to branch on *)      50
    mutable children: dnstrie array; (* Downward edges *)
    mutable ch_key: string;          (* Characters tagging each edge *)
    mutable least_child: char;       (* Smallest key of any child *)
    mutable flags: nodeflags;        (* Kinds of records held here *)
  }

let bad_node = { data = None; edge = ""; byte = -1;
                 children = [||]; ch_key = ""; least_child = '\255';
                 flags = Nothing; }
                                                                      60
exception TrieCorrupt                (* Missing data from a soa/cut node *)
```

```
(* Utility for trie ops: compare the remaining bytes of key with the
   inbound edge to this trie node *)
let cmp_edge node key =
  let edgelen = String.length node.edge in
  let offset = node.byte − edgelen in
  let keylen = String.length key − offset in
  let cmplen = min edgelen keylen in
  let cmpres = String.compare (String.sub node.edge 0 cmplen)
                              (String.sub key offset cmplen) in
  if cmpres = 0 then begin
    if keylen >= edgelen then
      'Match       (* The key matches and reaches the end of the edge *)
    else
      'OutOfKey    (* The key matches but runs out before the edge finishes *)
  end
  else if cmpres < 0 then
    'ConflictGT  (* The key deviates from the edge, and is "greater" than it *)
  else (* cmpres > 0 *)
    'ConflictLT  (* The key deviates from the edge *)

(* Utility for trie ops: number of matching characters *)
let get_match_length trie key =
  let rec match_r edge key n off =
    try
      if edge.[n] = key.[n + off] then match_r edge key (n + 1) off
      else n
    with
      Invalid_argument _ → n
  in
  let offset = trie.byte − (String.length trie.edge) in
  match_r trie.edge key 0 offset
```

```
(* Child table management: arrays*)
let new_child_table () = Array.copy [| |]

let children_iter f node =
  let ch_count = Array.length node.children in
  assert (ch_count == String.length node.ch_key);
  for i = 0 to (ch_count − 1) do                                    100
    f node.ch_key.[i] node.children.(i)
  done

let child_lookup char node =
  let ch_count = Array.length node.children in
  assert (ch_count == String.length node.ch_key);
  let rec lfn i =
    if (i >= ch_count) then None
    else if node.ch_key.[i] = char then Some node.children.(i)
    else lfn (i + 1)                                               110
  in lfn 0

let child_update node char child =
  let ch_count = Array.length node.children in
  assert (ch_count == String.length node.ch_key);
  let rec ufn i =
    if (i >= ch_count) then true
    else if node.ch_key.[i] = char then
      begin node.children.(i) ← child; false end
    else ufn (i + 1)                                               120
  in if ufn 0 then
    begin
      node.children ← Array.append node.children (Array.make 1 child);
      node.ch_key ← (node.ch_key ^ (String.make 1 char))
    end
```

```
let child_delete node char =
  let ch_count = Array.length node.children in
  assert (ch_count == String.length node.ch_key);
  try
    let i = String.index node.ch_key char in                          130
    node.ch_key ← ((String.sub node.ch_key 0 i)
                   ^ (String.sub node.ch_key (i+1) (ch_count − (i+1))));
    node.children ← (Array.append
                       (Array.sub node.children 0 i)
                       (Array.sub node.children (i+1) (ch_count − (i+1))));
  with Not_found → ()

let child_lookup_less_than char node =
  let ch_count = Array.length node.children in
  assert (ch_count == String.length node.ch_key);                     140
  let rv = ref None in
  let rc = ref '\000' in
  let ifn c n =
    if c >= !rc && c < char then begin rc := c; rv := Some n end
  in children_iter ifn node;
  !rv

let child_count node =
  Array.length node.children
                                                                      150
let only_child node =
  assert (Array.length node.children = 1);
  node.children.(0)
```

```
(* Assert that this value exists: not every trie node needs to hold data,
   but some do, and this allows us to discard the "option" when we know
   it's safe *)
let not_optional = function
    Some x → x
  | None → raise TrieCorrupt

(* Make a new, empty trie *)
let new_trie () =
  let rec n = { data = None;
                edge = "";
                byte = 0;
                children = [| |];
                ch_key = "";
                least_child = '\255';
                flags = Nothing;
              } in n

(* Simple lookup function: just walk the trie *)
let rec simple_lookup key node =
  if not (cmp_edge node key = `Match) then None
  else if ((String.length key) = node.byte) then node.data
  else match (child_lookup key.[node.byte] node) with
    None → None
  | Some child → simple_lookup key child
```

```
(* DNS lookup function *)
let lookup key trie =
  (* Variables we keep track of all the way down *)
  let last_soa = ref bad_node in      (* Last time we saw a zone head *)
  let last_cut = ref trie in          (* Last time we saw a cut point *)
  let last_lt = ref trie in           (* Last time we saw something < key *)
  let last_rr = ref trie in           (* Last time we saw any data *)
  let secured = ref false in          (* Does current zone use DNSSEC? *)

  (* DNS lookup function, part 4: DNSSEC authenticated denial *)
  let lookup_nsec key =
    (* Follow the trie down to the right as far as we can *)
    let rec find_largest node =
      let lc = ref '\000' in
      let child = ref node in
      let ifn c node = if c >= !lc then begin lc := c; child := node end in
      children_iter ifn node;
      if !child = node then node
      else find_largest !child
    in
    let ch = key.[!last_lt.byte] in
    (* last_lt is the last chance we had to go to the left: could be for a
       number of reasons *)
    if not (cmp_edge !last_lt key = `Match) then
      find_largest !last_lt           (* All of last_lt is < key *)
    else if ch <= !last_lt.least_child then
      !last_lt                        (* Only last_lt itself is < key *)
    else (* Need to find the largest child less than key *)
      let lc = ref !last_lt.least_child in
      let ln = ref !last_lt in
      let ifn c node =
        if c >= !lc && c < ch then begin lc := c; ln := node end in
      children_iter ifn !last_lt;
      find_largest !ln
  in
```

```
(* DNS lookup function, part 3: wildcards. *)
let lookup_wildcard key last_branch =

  (* Find the offset of the last '\000' in the key *)
  let rec previous_break key n =                                    220
    if n < 0 then -1 else match key.[n] with
      '\000' → n
    | _ → previous_break key (n - 1)
  in

  (* Find the offset of the "Closest Encounter" in the key *)
  let closest_encounter key last_branch =
    let first_bad_byte = (last_branch.byte - String.length last_branch.edge)
        + get_match_length last_branch key in
    previous_break key first_bad_byte                               230
  in

  (* Lookup, tracking only the "last less-than" node *)
  let rec wc_lookup_r key node =
    match (cmp_edge node key) with
      'ConflictGT → last_lt := node; None
    | 'ConflictLT | 'OutOfKey → None
    | 'Match →
        if ((String.length key) = node.byte)
        then node.data                                              240
        else begin
          if not (node.data = None) || node.least_child < key.[node.byte]
          then last_lt := node;
          match (child_lookup key.[node.byte] node) with
            None → None
          | Some child → simple_lookup key child
        end
  in

  (* Find the source of synthesis, and look it up *)                 250
```

```
  let byte = (closest_encounter key last_branch) + 1 in
  let ss_key = String.sub key 0 byte ^ "*\000" in
  (ss_key, wc_lookup_r ss_key !last_rr)
in


(* DNS lookup function, part 2a: gather NSECs and wildcards *)
let lookup_failed key last_branch =
  if (!last_cut.byte > !last_soa.byte)
  then 'Delegated (!secured, not_optional !last_cut.data)
  else begin
    if !secured then
      let nsec1 = lookup_nsec key in
      match lookup_wildcard key last_branch with
        (_, Some dnsnode) → 'WildcardNSEC (dnsnode,
                                           not_optional !last_soa.data,
                                           not_optional nsec1.data)
      | (ss_key, None) → let nsec2 = lookup_nsec ss_key in
        'NXDomainNSEC (not_optional !last_soa.data,
                       not_optional nsec1.data, not_optional nsec2.data)
    else
      (* No DNSSEC: simple answers. *)
      match lookup_wildcard key last_branch with
        (k, Some dnsnode) → 'Wildcard (dnsnode, not_optional !last_soa.data)
      | (k, None) → 'NXDomain (not_optional !last_soa.data)
  end
in


(* DNS lookup function, part 2b: gather NSEC for NoError *)
let lookup_noerror key =
  if (!last_cut.byte > !last_soa.byte)
  then 'Delegated (!secured, not_optional !last_cut.data)
  else begin
    if !secured then
```

```
      (* Look for the NSEC RR that covers this name. RFC 4035 says we
         might need another one to cover possible wildcards, but since this
         name "exists", the "Next Domain Name" field of the NSEC RR will
         be a child of this name, proving that it can't match any wildcard *)
      let nsec = lookup_nsec key in                                    290
      'NoErrorNSEC (not_optional !last_soa.data, not_optional nsec.data)
    else
      'NoError (not_optional !last_soa.data)
  end
in


(* DNS lookup function, part 1: finds the node that holds any data
   stored with this key, and tracks last_* for use in DNSSEC and wildcard
   lookups later. *)                                                   300
let rec lookup_r key node last_branch =
  match cmp_edge node key with
    'ConflictLT → lookup_failed key last_branch
  | 'ConflictGT → last_lt := node; lookup_failed key last_branch
  | 'OutOfKey → lookup_noerror key
  | 'Match →
      begin
        begin
          match node.flags with
            Nothing → ()                                               310
          | Records → last_rr := node
          | ZoneHead → last_rr := node;
              last_soa := node; last_cut := node; secured := false
          | SecureZoneHead → last_rr := node;
              last_soa := node; last_cut := node; secured := true
          | Delegation → last_rr := node; last_cut := node
        end;
        if ((String.length key) = node.byte) then
          match node.data with
            Some answer →                                              320
```

```
                if (!last_cut.byte > !last_soa.byte)
                then 'Delegated (!secured, not_optional !last_cut.data)
                else 'Found (!secured, answer, not_optional !last_soa.data)
            | None → lookup_noerror key
          else begin
            if not (node.data = None) || node.least_child < key.[node.byte]
            then last_lt := node;
            match (child_lookup key.[node.byte] node) with
              None → lookup_failed key node
            | Some child →                                              330
                begin
                  assert (child.byte > node.byte);
                  lookup_r key child node
                end
          end
        end
in

lookup_r key trie trie
```

```
(* Return the data mapped from this key, making new data if there is
   none there yet. *)
let rec lookup_or_insert key trie ?(parent = trie) factory =
  let get_data_or_call_factory node =
    match node.data with
      Some d → d
    | None →
        let d = factory () in
        node.data ← Some d;
        assert (node.flags = Nothing);
        node.flags ← Records;
        d
  in
  if not (cmp_edge trie key = 'Match) then begin
    (* Need to break this edge into two pieces *)
    let match_len = get_match_length trie key in
    let rest_len = String.length trie.edge − match_len in
    let branch = { data = None;
                   edge = String.sub trie.edge 0 match_len;
                   byte = trie.byte − rest_len;
                   children = [| |];
                   ch_key = "";
                   least_child = trie.edge.[match_len];
                   flags = Nothing;
                 } in
    (* Order of the next two lines is important *)
    child_update parent branch.edge.[0] branch;
    trie.edge ← String.sub trie.edge match_len rest_len;
    child_update branch trie.edge.[0] trie;
    (* Don't need to update parent.least_child *)
    lookup_or_insert key branch ~parent:parent factory
  end else begin
    let length_left = (String.length key) − trie.byte in
    if length_left = 0 then
      get_data_or_call_factory trie
```

```
  else begin
    match (child_lookup key.[trie.byte] trie) with
      Some child → begin
        lookup_or_insert key child ~parent:trie factory
      end
    | None → let child = { data = None;                                    380
                           edge = String.sub key trie.byte length_left;
                           byte = String.length key;
                           children = [| |];
                           ch_key = "";
                           least_child = '\255';
                           flags = Nothing;
                         } in
      assert (child.edge.[0] = key.[trie.byte]);
      child_update trie child.edge.[0] child;
      trie.least_child ← min trie.least_child key.[trie.byte];             390
      get_data_or_call_factory child
  end
end
```

```
(* Sort out flags for a key's node; call after inserting significant RRs *)
let rec fix_flags key node =
  let soa = ref false in
  let ns = ref false in
  let dnskey = ref false in

  let rec set_flags node rrset =                                    400
    match rrset.rdata with
      SOA _ → soa := true
    | NS _    → ns := true
    | DNSKEY _ → dnskey := true
    | _        → ()
  in

  if not (cmp_edge node key = 'Match) then ()
  else if ((String.length key) = node.byte) then begin
    match node.data with                                            410
      None → node.flags ← Nothing
    | Some dnsnode → List.iter (set_flags node) dnsnode.rrsets;
        if !soa then
          if !dnskey then
            node.flags ← SecureZoneHead
          else
            node.flags ← ZoneHead
        else if !ns then
          node.flags ← Delegation
        else                                                        420
          node.flags ← Records
  end else match (child_lookup key.[node.byte] node) with
    None → ()
  | Some child → fix_flags key child
```

```
(* Delete all the data associated with a key *)
let rec delete key ?(gparent = None) ?(parent = None) node =
  if not (cmp_edge node key = 'Match) then ()
  else if ((String.length key) != node.byte) then
    begin
      match (child_lookup key.[node.byte] node) with                    430
        None → ()
      | Some child → delete key ~gparent:parent ~parent:(Some node) child
    end

  else let collapse_node p n =
    let c = only_child n in
    c.edge ← (n.edge ^ c.edge);
    child_update p c.edge.[0] c

  in begin                                                              440
    node.data ← None;
    match parent with None → () | Some p →
      match child_count node with
        1 → collapse_node p node
      | 0 →
          begin
            child_delete p node.edge.[0];
            if ((p.data = None) && (child_count p = 1)) then
              match gparent with None → () | Some gp → collapse_node gp p
          end                                                           450
      | _ → ()
  end
```

B UNS First Protocol

For reference, we give here a summary of the workings of the UNS First Protocol, the pessimistic protocol used in Chapter 6. Full details and analysis of the protocol are published in Chaoying Ma's Ph.D. dissertation [Ma92].

Each participant P_i maintains a triple (Psid, Nsid, PrevR) describing the state of operations. *Psid* is the identifier of the last synchronization in which the the participant voted (if any), and *PrevR* is the update for which it voted. *Nsid* is the identifier of the last synchronization in which the participant has promised to participate (but not yet necessarily voted).

A *synchronization* is initiated by a co-ordinator C, and proceeds in three phases:

Phase 1: gathering a quorum

C makes a new synchronization ID *sid* and sends it to every participant P_i, along with the version number of the object's current state.

Each P_i does one of three things, depending on the version number of its local replica of the object and the version number it received from C:

- If P_i's version is newer, it tells C about it. C then abandons the update until it has obtained the new version of the object.
- If P_i's version is older, it cannot participate in the synchronization. Instead, it pulls the new version of the object from another node.
- If the versions are the same, P_i compares its Nsid to *sid*. If *sid* is newer, then it sets Nsid to *sid* and enters the quorum; otherwise it

declines to enter the quorum. In either case, it sends its Psid and PrevR to C.

If enough P_i agree to enter a quorum, C looks at all the Psid/PrevR pairs it received from them. If all of them are empty it proceeds to Phase 2 with the update it wants to propagate. If any of them is not empty, it chooses the pair with the highest value of Psid and proceeds to Phase 2 using that pair's PrevR update. (It must then start again at Phase 1 to propagate its own update.)

Phase 2: voting

C sends *sid* and the proposed update to every P_i in the quorum it gathered in Phase 1. Each P_i compares *sid* with its local triple: if it is equal to Nsid (and greater than Psid), it sets Psid to *sid* and PrevR to the update proposed. In any case, it sends Psid and PrevR to C.

C again looks at the Psid/PrevR pairs it receives from the P_i. If it receives enough messages with Psid equal to *sid* to form a quorum, the vote has passed. If not, C must start again at Phase 1.

Phase 3: commit

C sends a commit order for the proposed update to every P_i in the quorum that voted for it.

Each of those P_i applies the update to its local replica of the object, updates its version number, and clears its (Psid, Nsid, PrevR) triple, ready for the next synchronization.

C DNS record types

The definitions of DNS data types are surprisingly widely scattered: there is no single standards document listing them all, and some are not described in IETF documents at all. We have tried to gather a list of references for the types that are established.

Table C.1 lists the RFCs that define the layout and handling rules of record types. Where types have been redefined or updated we try to give the most recent reference. Two type numbers are multiply assigned: the numbers 32 and 33 were assigned in RFC 1002 to NB and NBSTAT, but were later also assigned to NIMLOC and SRV.

There are ten types that are assigned numbers but not (yet) defined in any RFC: SPF [WS05], TA [Wei04b], EID [Pat96], NIMLOC [Pat96], ATMA [ATM96], SINK [Eas99], UINFO, UID, GID, and UNSPEC[†]. In addition, the WINS and WINSR types [GE99] are not even on the IANA list of assigned numbers; they were defined with provisional "private use" numbers (65281 and 65282), and WINS/65281 records are still in use.

New types are constantly under development. Currently proposed types include DHCID [SLG06], HIP [NL05], NFS4ID [Mes05], SLOC [dL05], SDDA [Mor05] and NSEC3 [LSA05].

RFC 3597 [Gus03] describes handling rules to be followed when resolvers encounter records of unknown type.

[†]These last four types were implemented in older versions of BIND but never described in RFCs, so were removed after version 4. BIND 4 came with an `ex` script to make UINFO, UID and GID RRs out of a UNIX password file; UNSPEC was an unformatted binary type used to contain arbitrary data in formats BIND did not understand.

RFC 1002 [NIE87]	NB NBSTAT
RFC 1035 [Moc87b]	A NS MD MF CNAME SOA MB MG MR NULL WKS PTR HINFO MINFO MX TXT AXFR MAILB MAILA ANY
RFC 1183 [EMUM90]	RP AFSDB X25 ISDN RT
RFC 1348 [Man92]	NSAP-PTR (obsoleted by RFC 1706.)
RFC 1706 [MC94]	NSAP
RFC 1712 [FSPB94]	GPOS
RFC 1876 [DVGD96]	LOC
RFC 1995 [Oht96]	IXFR
RFC 2163 [All98]	PX
RFC 2230 [Atk97]	KX
RFC 2538 [EG99]	CERT
RFC 2671 [Vix99]	OPT
RFC 2672 [Cra99b]	DNAME
RFC 2782 [GVE00]	SRV
RFC 2874 [CH00]	A6
RFC 2930 [Eas00]	TKEY
RFC 3123 [Koc01]	APL
RFC 3402 [Mea02]	NAPTR
RFC 3596 [THKS03]	AAAA
RFC 3645 [KGG+03]	TSIG
RFC 3755 [Wei04a]	SIG KEY NXT (obsoleted by RFC 4034)
RFC 4025 [Ric05]	IPSECKEY
RFC 4034 [AAL+05c]	RRSIG DNSKEY NSEC DS
RFC 4255 [SG06]	SSHFP
RFC 4431 [AW06]	DLV

Table C.1: RFCs that define DNS record types.

Bibliography

[AAL+05a] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, March 2005. 2, 11, 18, 25, 89

[AAL+05b] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, March 2005. 18, 89

[AAL+05c] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, March 2005. 6, 18, 19, 21, 89, 128

[Abl03] J. Abley. Hierarchical anycast for global service distribution. Technical Note ISC-TN-2003-1, Internet Systems Consortium, 2003. 2

[All98] C. Allocchio. Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM). RFC 2163, January 1998. 128

[And98] M. Andrews. Negative Caching of DNS Queries (DNS NCACHE). RFC 2308, March 1998. 15

[AS04a] B. Awerbuch and C. Scheideler. Group spreading: A protocol for provably secure distributed name service. In *Proc. 31st ICALP*, volume 3142 of *LNCS*, pp. 183–195, July 2004. 97

[AS04b] B. Awerbuch and C. Scheideler. Robust distributed name service. In *Proc. 3rd IPTPS*, volume 3279 of *LNCS*, pp. 237–249. Springer-Verlag, February 2004. 97, 106

[Atk97] R. Atkinson. Key Exchange Delegation Record for the DNS. RFC 2230, November 1997. 128

[ATM96] ATM Forum Technical Committee. *ATM Name System Specification version 1.0*. ATM Forum, November 1996. `ftp://ftp.atmforum.com/pub/approved-specs/af-saa-0069.000.pdf`. 127

[AW06] M. Andrews and S. Weiler. The DNSSEC Lookaside Validation (DLV) DNS Resource Record. RFC 4431, February 2006. 21, 89, 128

[BBK00] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proc. 9th WWW*, May 2000. 27

[BDF+02] R. Bush, A. Durand, B. Fink, O. Gudmundsson, and T. Hain. Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System (DNS). RFC 3363, August 2002. 5, 75

[BKKP00] R. Bush, D. Karrenberg, M. Kosters, and R. Plzak. Root Name Server Operational Requirements. RFC 2870, June 2000. 25, 43

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005. 99

[BLNS82] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: an exercise in distributed computing. *Commun. ACM*, 25(4):260–274, April 1982. 2, 65, 104

[BLR+04] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *Proc. ACM SIGCOMM 2004*, pp. 343–352, August 2004. 99

[BNc03] A. Broido, E. Nemeth, and k. claffy. Spectroscopy of DNS update traffic. In *Proc. ACM SIGMETRICS 2003*, pp. 320–321, June 2003. 93

[Box04] D. Box, editor. *Web Services Addressing*. W3C, August 2004. `http://www.w3.org/Submission/ws-addressing/`. 99

[Bra96] S. Bradner. The Internet Standards Process – Revision 3. RFC 2026, October 1996. 5

[CCITT05a] Open systems interconnection — the directory. ITU–T Recommendation X.500–X.586, International Telecommunication Union, August 2005. 1

[CCITT05b] Open systems interconnection — the directory: Public-key and attribute certificate frameworks. ITU–T Recommendation X.509, International Telecommunication Union, August 2005. 57

[CDK+03] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proc. 19th ACM SOSP*, pp. 298–313, October 2003. 65, 70

[CFS+03] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu. Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework. RFC 3647, November 2003. 57

[CGM+04] R. Chinnici, M. Gudgin, J-J. Moreau, J. Schlimmer, and S. Weerawarana. *Web Services Definition Language (WSDL) Version 2.0*. W3C, 2004. `http://www.w3c.org/TR/wsdl20`. 99

[CH00] M. Crawford and C. Huitema. DNS Extensions to Support IPv6 Address Aggregation and Renumbering. RFC 2874, July 2000. 128

[CIA05] Central Intelligence Agency. *The World Factbook*. 2005. `http://www.cia.gov/cia/publications/factbook/`. 34

[Cis99] Cisco Systems, Inc. *Cisco DistributedDirector*. 1999. White Paper. `http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/dd_wp.pdf`. 21

[CJK+03] M. Castro, M. B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlay networks. In *Proc. 22nd IEEE INFOCOM*, volume 2, pp. 1510–1520, March 2003. 65

[CK00] E. Cohen and H. Kaplan. Prefetching the means for document transfer: A new approach for reducing web latency. In *Proc. 19th IEEE INFOCOM*, volume 2, pp. 854–863, March 2000. 23, 104

[CK01] E. Cohen and H. Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. In *Proc. SAINT 2001*, pp. 85–94, January 2001. 39, 43, 98, 105

[CMM02] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a peer-to-peer lookup service. In *Proc. 1st IPTPS*, volume 2429 of *LNCS*, pp. 155–165, March 2002. 3, 44, 57, 96, 106

[Cra99a] M. Crawford. Binary Labels in the Domain Name System. RFC 2673, August 1999. 5, 6, 75

[Cra99b] M. Crawford. Non-Terminal DNS Name Redirection. RFC 2672, August 1999. 16, 128

[DCW05] T. Deegan, J. Crowcroft, and A. Warfield. The Main Name System: An exercise in centralized computing. *Comput. Commun. Rev.*, 35(5):5–13, October 2005.

[dL05] C. de Launois. Synthetic location information in the domain name system. Internet Draft draft-de-launois-dnsext-sloc-rr-00, July 2005. Expired January 2006. `http://www2.info.ucl.ac.be/people/delaunoi/ietf/draft-de-launois-dnsext-sloc-rr-00.txt`. 21, 127

[DOK92] P. B. Danzig, K. Obrackza, and A. Kumar. An analysis of wide-area name server traffic: A study of the internet domain name system. In *Proc. ACM SIGCOMM 1992*, pp. 281–292, 1992. 43, 90

[DVGD96] C. Davis, P. Vixie, T. Goodwin, and I. Dickinson. A Means for Expressing Location Information in the Domain Name System. RFC 1876, January 1996. 128

[Eas99] D. Eastlake 3rd. The kitchen sink DNS resource record. Internet Draft draft-ietf-dnsind-kitchen-sink-02, September 1999. Expired March 2000. 127

[Eas00] D. Eastlake 3rd. Secret Key Establishment for DNS (TKEY RR). RFC 2930, September 2000. 15, 20, 128

[Eas06] D. Eastlake 3rd. Domain Name System (DNS) Case Insensitivity Clarification. RFC 4343, January 2006. 5

[EB97] R. Elz and R. Bush. Clarifications to the DNS Specification. RFC 2181, July 1997. 6, 11, 38

[EF01] M. P. Evans and S. M. Furnell. The resource locator service: fixing a flaw in the web. *Computer Networks*, 37(3):307–330, November 2001. 99

[EG99] D. Eastlake 3rd and O. Gudmundsson. Storing Certificates in the Domain Name System (DNS). RFC 2538, March 1999. 128

[EMUM90] C. Everhart, L. Mamakos, R. Ullmann, and P. V. Mockapetris. New DNS RR Definitions. RFC 1183, October 1990. 128

[FCDH90] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. Order preserving minimal perfect hash functions and information retrieval. In *Proc. 13th ACM SIGIR*, pp. 279–311, September 1990. 97

[Fil00] J-C. Filliâtre. Hash consing in an ML framework. Research report 1368, LRI, Université Paris-Sud, September 2000. 82

[FSPB94] C. Farrell, M. Schulze, S. Pleitner, and D. Baldoni. DNS Encoding of Geographical Location. RFC 1712, November 1994. 128

[GE99] J. Gilroy and L. Esibov. WINS lookup by DNS server (WINS-Lookup). Internet Draft draft-levone-dns-wins-lookup-00, October 1999. Expired April 2000. 127

[Gus03] A. Gustafsson. Handling of Unknown DNS Resource Record (RR) Types. RFC 3597, September 2003. 127

[GVE00] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, February 2000. 128

[HA00] M. Habib and M. Abrams. Analysis of sources of latency in downloading web pages. In *Proc. 5th WebNet*, October 2000. 23, 104

[HG05] M. Handley and A. Greenhalgh. The case for pushing DNS. In *Proc. 4th HotNets*, November 2005. 95, 106

[Hus01] G. Huston. Management Guidelines & Operational Requirements for the Address and Routing Parameter Area Domain ("arpa"). RFC 3172, September 2001. 8

[ICANN99] Internet domain name system structure and delegation (ccTLD administration and delegation). Internet Coordination Policy document ICP1, ICANN, May 1999. `http://www.icann.org/icp/icp-1.htm`. 8

[ISC] Internet Systems Consortium. *Internet Domain Survey*. `http://www.isc.org/ops/ds/`. 2, 33, 91

[ISO97] Codes for the representation of names of countries and their subdivisions — Part 1: Country codes. ISO Standard 3166-1, 5th edition, International Organization for Standardization, 1997. 8

[JBB03] J. Jung, A. W. Berger, and H, Balakrishnan. Modeling TTL-based internet caches. In *Proc. 22nd IEEE INFOCOM*, volume 1, pp. 417–426, March 2003. 98, 105

[JCDK01] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. *Comput. Commun.*, 24(2):202–206, February 2001. 63

[JSBM01] J. Jung, E. Sit, H. Balakrishnan, and R. T. Morris. DNS performance and the effectiveness of caching. In *Proc. 1st IMW*, pp. 55–67, November 2001. 24, 98

[JT75] P. Johnson and R. Thomas. Maintenance of duplicate databases. RFC 677, January 1975. 64

[KGG+03] S. Kwan, P. Garg, J. Gilroy, L. Esibov, J. Westhead, and R. Hall. Generic Security Service Algorithm for Secret Key Transaction Authentication for DNS (GSS-TSIG). RFC 3645, October 2003. 20, 128

[Koc01] P. Koch. A DNS RR Type for Lists of Address Prefixes (APL RR). RFC 3123, June 2001. 128

[KR00] J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proc. 19th IEEE INFOCOM*, volume 2, pp. 660–669, March 2000. 57, 94, 106

[Lam86] B. W. Lampson. Designing a global name service. In *Proc. 5th PODC*, pp. 1–10, August 1986. 2

[Lam89] L. Lamport. The part-time parliament. Digital SRC Research report 49, Digital Equipment Corporation, September 1989. 68

[Law02] D. Lawrence. Obsoleting IQUERY. RFC 3425, November 2002. 14

[Lew06] E. Lewis. The role of wildcards in the domain name system. Internet Draft draft-ietf-dnsext-wcard-clarify-11, March 2006. Work in progress. Expires September 13, 2006. 7, 77

[LHFc03] T. Lee, B. Huffaker, M. Fomenkov, and k. claffy. On the problem of optimization of DNS root servers' placement. In *Proc. 4th PAM*, April 2003. 43

[LL04] D. M. LaFlamme and B. N. Levine. DNS with ubiquitous replication to survive attacks and improve performance. Technical Report UM-CS-2004-039, University of Massachusetts Amherst, June 2004. 39, 94, 106

[LSA05] B. Laurie, G. Sisson, and R. Arends. DNSSEC hash authenticated denial of existence. Internet Draft draft-ietf-dnsext-nsec3-03, August 2005. Work in progress. Expires April 26, 2006. 21, 79, 90, 127

[LSZ02] R. Liston, S. Srinivasan, and E. Zegura. Diversity in DNS performance measures. In *Proc. 2nd IMW*, pp. 19–31, November 2002. 92

[Ma92] C. Ma. *Designing a universal name service*. Ph.D. dissertation, University of Cambridge, 1992. Also available as Technical Report UCAM-CL-TR-270.
`http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-270.html`
68, 69, 100, 125

[Mal04] D. Malone. The root of the matter: hints or slaves. In *Proc. 2nd IMC*, pp. 15–20, October 2004. 95

[Man92] B. Manning. DNS NSAP RRs. RFC 1348, July 1992. 128

[MC94] B. Manning and R. Colella. DNS NSAP Resource Records. RFC 1706, October 1994. 128

[MD88] P. V. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Proc. ACM SIGCOMM 1988*, pp. 123–133, August 1988. 2, 25, 38, 60

[Mea02] M. Mealling. Dynamic Delegation Discovery System (DDDS) Part Two: The Algorithm. RFC 3402, October 2002. 128

[Mes05] R. Mesta. A DNS RR for NFSv4 ID domains. Internet Draft draft-ietf-nfsv4-dns-rr-00, October 2005. Work in progress. Expires April 15, 2006. 21, 127

[MNJH05] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host identity protocol. Internet Draft draft-ietf-hip-base-04, October 2005. Work in progress. Expires April 27, 2006. 59, 100

[Moc87a] P. V. Mockapetris. Domain names — concepts and facilities. RFC 1034, November 1987. 1, 6, 45

[Moc87b] P. V. Mockapetris. Domain names — implementation and specification. RFC 1035, November 1987. 11, 15, 128

[Mor05] T. Moreau. The SEP DNSKEY direct authenticator DNS resource record (SDDA-RR). Internet Draft draft-moreau-dnsext-sdda-rr-01, December 2005. Work in progress. Expires June 2006. 21, 127

[MS05a] A. Madhavapeddy and D. Scott. On the challenge of delivering high-performance, dependable, model-checked internet servers. In *Proc. 1st HotDep*, June 2005. 82

[MS05b] D. P. Mehta and S. Sahni, editors. *Handbook of Data Structures and Applications*. Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton, 2005. 73, 74

[MS05c] Microsoft Corporation. *Sender ID Overview*. 2005. White Paper. `http://www.microsoft.com/senderid`. 90

[NIE87] NetBIOS Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, and End-to-End Services Task Force. Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications. RFC 1002, March 1987. 128

[NL05] P. Nikander and J. Laganier. Host identity protocol (HIP) domain name system (DNS) extensions. Internet Draft draft-ietf-hip-dns-04, December 2005. Work in progress. Expires June 19, 2006. 21, 127

[Oht96] M. Ohta. Incremental Zone Transfer in DNS. RFC 1995, August 1996. 16, 128

[PACR02] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proc. 1st HotNets*, pp. 59–64, October 2002. `http://www.planet-lab.org/`. 92

[PAS+04] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan. On the responsiveness of DNS-based network control. In *Proc. 2nd IMC*, pp. 21–26, October 2004. 21

[Pat96] M. A. Patton. DNS resource records for nimrod routing architecture. Internet Draft draft-ietf-nimrod-dns-02, November 1996. Expired May 1997. `http://www.ir.bbn.com/projects/nimrod/draft-ietf-nimrod-dns-02.txt`. 127

[PHA+04] J. Pang, J. Hendricks, A. Akella, B. Maggs, R. De Prisco, and S. Seshan. Availability, usage, and deployment characteristics of the domain name system. In *Proc. 2nd IMC*, pp. 1–14, October 2004. 92

[PHL03] J. Pan, Y. T. Hou, and B. Li. An overview of DNS-based server selections in content distribution networks. *Computer Networks*, 43:695–711, December 2003. 21

[PMTZ06] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A comparative study of the DNS design with DHT-based alternatives. To appear in *Proc. 25th IEEE INFOCOM*, April 2006. 96, 97

[PPPW04] K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proc. 6th OSDI*, pp. 199–214, December 2004. 23, 45, 92, 98, 105

[PWPP04] K. Park, Z. Wang, V. S. Pai, and L. Peterson. CoDNS: Masking DNS delays via cooperative lookups. Technical Report TR-690-04, Princeton University Computer Science Department, February 2004. 44, 98

[PXL+04] V. Pappas, Z. Xu, S. Lu, D. Massey, A. Terzis, and L. Zhang. Impact of configuration errors on DNS robustness. In *Proc. ACM SIGCOMM 2004*, pp. 319–330, August 2004. 24, 25, 93

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*, volume 2218 of *LNCS*, pp. 329–350, November 2001. 70

[Ric05] M. Richardson. A Method for Storing IPsec Keying Material in DNS. RFC 4025, March 2005. 128

[RIPE] RIPE NCC. *The RIPE Region Hostcount*. `http://ripe.net/info/stats/hostcount/`. 32, 91

[RM92] Rage Against the Machine. *Rage Against the Machine*. Epic Records, March 1992.

[RM99] Rage Against the Machine. *The Battle of Los Angeles*. Epic Records, February 1999.

[RMK+96] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, February 1996.

[RS04] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *Proc. ACM SIGCOMM 2004*, pp. 331–342, August 2004. 44, 57, 96, 106

[RS05] V. Ramasubramanian and E. G. Sirer. Perils of transitive trust in the domain name system. In *Proc. 3rd IMC*, October 2005. 20, 93

[SG06] J. Schlyter and W. Griffin. Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. RFC 4255, January 2006. 128

[SLG06] M. Stapp, T. Lemon, and A. Gustafsson. A DNS RR for encoding DHCP information (DHCID RR). Internet Draft draft-ietf-dnsext-dhcid-rr-13, September 2006. Work in progress. Expires September 23, 2006. 21, 127

[SS05] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005. 63

[THKS03] S. Thomson, C. Huitema, V. Ksinant, and M. Souissi. DNS Extensions to Support IP Version 6. RFC 3596, October 2003. 128

[Tuo02] I. Tuomi. The Lives and Death of Moore's Law. *First Monday*, 7(11), November 2002. 47

[Ult01] UltraDNS Corporation. *SecondaryDNS.com*. `http://www.secondarydns.com/overview/tech.html`. 57

[VDAA99] A. Vahdat, M. Dahlin, T. Anderson, and A. Agarwal. Active names: Flexible location and transport of wide-area resources. In *Proc. 2nd USITS*, October 1999. 100

[Ver05] Verisign, Inc. *Ensuring your company's online presence*. June 2005. White paper. `http://www.verisign.com/Resources/`. 57

[VGEW00] P. Vixie, O. Gudmundsson, D. Eastlake 3rd, and B. Wellington. Secret Key Transaction Authentication for DNS (TSIG). RFC 2845, May 2000. 2, 15, 19, 25

[Vix96] P. Vixie. A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY). RFC 1996, August 1996. 13, 62

[Vix99] P. Vixie. Extension Mechanisms for DNS (EDNS0). RFC 2671, August 1999. 12, 15, 128

[Vix05] P. Vixie. Preventing child neglect in DNSSECbis using lookaside validation (DLV). *IEICE Trans. Commun.*, E88-B(4):1326–1330, April 2005. 89

[VSS02] P. Vixie, G. Sneeringer, and M. Schleifer. *Events of 21–Oct–2002*. `http://d.root-servers.org/october21.txt`. 25

[VTRB97] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136, April 1997. 2, 14, 15, 66

[WBS04] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from DNS. In *Proc. 1st NSDI*, pp. 225–238, March 2004. 99

[Wei04a] S. Weiler. Legacy Resolver Compatibility for Delegation Signer (DS). RFC 3755, May 2004. 128

[Wei04b] S. Weiler. Deploying DNSSEC without a signed root. INI 1999-19, Carnegie Mellon University, April 2004. 127

[Wes04] D. Wessels. Is your caching resolver polluting the internet? In *Proc. ACM SIGCOMM NetTs*, pp. 271–276, August 2004. 25, 43, 93

[WF03] D. Wessels and M. Fomenkov. Wow, that's a lot of packets. In *Proc. 4th PAM*, April 2003. 25, 91, 95

[WFBc04] D. Wessels, M. Fomenkov, N. Brownlee, and k. claffy. Measurements and laboratory simulations of the upper DNS hierarchy. In *Proc. 5th PAM*, pp. 147–157, April 2004. 45, 59, 91

[WS05] M. Wong and W. Schlitt. Sender policy framework (SPF) for authorizing use of domains in e-mail, version 1. Internet Draft draft-schlitt-spf-classic-02, June 2005. To be published as an Experimental RFC. 21, 127