Taughannock Networks

The Design of the Domain Name System

June 2012

Abstract

The Domain Name Systems is a globally distributed realtime database. We look at aspects of its design, and how that affects what kinds of data can be stored in it, and what usage patterns are more or less successful.

Introduction

Over the past 30 years the Domain Name System has become an integral part of the operation of the Internet. Due to its ubiquity and good performance, many new applications over the years have used the DNS to publish information. But as the DNS and its applications have grown farther from its original use in publishing information about Internet hosts, questions have arisen about what applications are appropriate for publication in the DNS, and how one should design an application to work well with the DNS.

The DNS basically is a client-server system that provides a map from a tree-structured name-space into arrays of typed records. That is, a client sends a name and a record type to a server which has authoritative data for that name, and the DNS server returns a set of records. The set may be empty (often known as NOERROR from the name of the status code), or it may be nonexistent (NXDOMAIN) which is semantically different.

The names are a sequence of string labels representing a path from the root of the DNS to a node. Although the DNS protocol is almost 8-bit clean (other than case folding the 26 upper and lower case letters), by convention all labels are composed of 7-bit ASCII printing characters. Names are conventionally written with a dot separating the labels, although a dot is a valid character in a label.

The records returned have types known as RRTYPEs. Each RRTYPE has a well-defined format for the fields in the record. Common RRTYPEs are A, which includes an IPv4 address, and MX, which includes the name of host that handles mail for a domain and a number that gives the relative priority of multiple hosts handling mail for a domain. Although it is possible to make an "any" query, because of the way that DNS caches work, the only reliable way to determine if a particular kind of record is present at a name is to make a query for that record type.

Taughannock Networks provides strategic consulting, design, and prototype implementation on e-mail and Internet infrastructure. +1 607 330 5711 or info@taugh.com

Delegation and caches

The DNS divdes up its work using *delegation* and *caches*. A DNS server can delegate the tree at and below any node, known as a *zone*, to one or more other servers. In response to a query for a name in the delegated tree, the delegating server returns a list of the servers to which the tree is delegated. The client then resends the query to one of those servers. Chains of three or four levels of delegation are common. Each zone can be (and typically is) managed separately from the zones above it and below it.

Rather than send queries directly to the authoritative servers, most DNS clients instead query a local *cache* which queries the authoritative servers on the client's behalf. This has two practical advantages. One is that the cache handles all of the delegation following, as well as some other name indirection due to CNAME and DNAME records, allowing a much simpler query library in the client. The other is that the cache remembers and reuses the results of previous queries, notably including queries that provided delegation information. DNS caches are believed to be highly effective, although actual data on their effectiveness is surprisingly sparse. Hence applications with high query rates and few repeated queries may cause problems for DNS caches, both because of the amount of DNS traffic caused directly, and because the results will replace other entries in caches, causing more DNS traffic to re-fetch results that would otherwise have been available in the cache.

Provisioning systems

The set of records in each authoritative server is provided via some kind of *provisioning system*. The original model in RFCs 1034 and 1035, the 1987 documents that are the basic definition of the DNS, assumed that the data for each zone would be hand edited into a text file known as a *master file* which is then loaded into one of the authoritative servers, known as the master for the zone. The rest of the zone's servers, known as slaves, use AXFR queries to copy over the zone, and recopy whenever the zone changes. A more recent modified version of AXFR called IXFR allows slave servers to copy just the changes to modified zones.

Although there are still plenty of zones that work this way, it doesn't scale very well for servers with very large zones, or that serve many different zones. Since the details of server provisioning, including the distinction between master and slave servers, are not apparent to DNS clients, many servers are provisioned in other ways, such as storing the zone data in a data base, using a web-based front end to manage data base or text zone data, or in some cases such as rbldnsd (the server used for most DNS blacklists and whitelists) to store the zone data in an entirely different form and create the DNS responses on the fly.

Although provisioning systems have not gotten a lot of attention from the DNS community, they are a key part of the DNS system. Most notably, provisioning systems are the main reason that it is difficult to deploy new DNS RRTYPEs, since the provisioning systems tend to have specific support for each kind of record, and need to be upgraded to handle each new record type. Limitations of provisioning systems are also a major reason that non-ASCII domain names are encoded as ASCII A-labels rather than putting the UTF-8 U-labels directly in the DNS.

DNS names

The most important limitation of the DNS, compared to other databases, is that it only does exact match lookups. That is, with a few minor exceptions, the name in the query has to match the name of the desired records exactly. One exception is folding of upper and lower case characters, which has little effect, the other is DNS wildcards.

Wildcards have always been part of the DNS, but the details of their definition have been confusing. The definition was clarified by RFC 4592 in 2006.

Wildcards provide a very constrained form of pattern matching, telling the server to synthesize records for all nodes below a specified node that don't have explicit records. That is, if there is a DNS record named `*.foo.example`, it will match requests for `something.foo.example`, so long as there aren't any records with that explicit name. A single star as the leftmost component of a domain name is the only form of wildcard; stars anywhere in a name are are just normal characters.

In practice, this turns out to be of limited use, with typical applications being web servers that catch any variation of their name, e.g. http://anything.example.com, and mail systems that give each user a separate domain, e.g., mailbox@anyuser.example.com. Wildcards do not work with prefixed names, such as _attribute.*.example.com, nor are they useful to handle ranges of queries except in some very stylized cases.

Some applications have proposed sequences of multiple queries to simulate range queries. For example, DNS blacklists (DNSBLs) map IP addresses into DNS names using a modified version of the mapping used for reverse DNS. If a DNSBL is called `dnsbl.example`, the entry for the IP adresss 12.34.56.78 would be `78.56.34.12.dnsbl.example`.

When a DNSBL wants to list a range of IP addresses, it needs conceptually to include a record for each name corresponding to an IP address in the range. For DNS servers that use traditional master files, since each component in the name represents eight bits of the IP ad-

dress, this involves breaking an IPv4 range into a minimal covering set of blocks on eight bit boundaries, adding wildcards for each block, and an individual entry for each individual address not in a larger block.

Some people have suggested approaches to try to optimize range listings by querying for prefixes of a desired address, e.g., if the address is 1.2.3.4 and the name is `4.3.2.1.dnsbl.example`, query for `2.1.dnsbl.example` to see of any of the 1.2.xx.xx range of addresses are listed. According to DNS rules, the query should return NXDOMAIN if there are no entries in the range, or NODATA if there are some. While this technique might work, it is quite fragile, due to DNS servers that don't correctly distinguish between NXDOMAIN and NODATA responses, currently including the most popular DNSBL server rbldnsd. Also, at this point there is no evidence that the probes really would save queries or cache entries compared to just querying for each address as needed. In principle, a DNS cache could synthesize its own NXDOMAIN responses for names below existing NXDOMAIN (anything in `*.2.1.dnsbl.example` here), but again, it's fragile, and as far as I know, no widely used DNS cache does that other than as an experiment.

As a general rule, a successful DNS application makes one query, or at most a small bounded number of queries for each application call. Note that the issue of DNS range queries is separate from that of application ranges. Most notably, the NAPTR RRTYPE, defined in RFC 3403, used to find servers for things like telephone numbers, includes a string which is interpreted by applications as a regular expression to be matched against a source string to find a domain for a subsequent lookup. While one can debate the wisdom of the rather complicated application design of which NAPTR is a part, it does not involve any pattern matching in the DNS. The NAPTR lookup algorithm makes a small set of specific DNS queries which the DNS handles without difficulty. It does involve potential provisioning problems, since regular expressions include a lot of special characters and escape sequences, something that few other RRTYPEs include and whose handling by provisioning software may not be well debugged.

DNS delegation

The DNS gains considerable administrative flexibility from its delegation structure. Each zone cut, the place in the DNS name tree where one set of DNS servers hands off to another, offers the option to delegate the administration of a part of the DNS at the delegation point. But for the delegation to work well, the delegation structure has to match the name structure.

In particular, a zone cut can only happen between name components, not within them. Assume, for example, the `example.com` organization wanted to delegate names starting with A to one group, and

names starting with B to another group, so they delegate `able.example.com`, `acme.example.com`, and `adept.example.com` to the first group, and `baker.example.com`, `beaver.example.com`, and `bingo.example.com` to the second group.

The DNS doesn't make that easy, since all of those names are part of the same zone. It would be possible to handle it by custom programming in the provisioning system that manages the `example.com` zone to manage who can update what names, but it would be a lot easier (from a DNS point of view) to add an extra name level and change the names to `able.a.example.com`, `acme.a.example.com`, `adept.a.example.com`, `baker.b.example.com`, `beaver.b.example.com`, and `bingo.b.example.com`. Then the organization can delegate `a.example.com` to the first group and `b.example.com` to the second group, and let them each manage its own zone. This assumes, of course, that it's acceptable for the applications to use the names with the extra components.

Reverse DNS delegation

The reverse DNS (rDNS) for IPv4, which allows you to look up a domain name corresponding to an IP address, is the part of the DNS where this problem has occurred most often. Originally, blocks of IP address space were allocated on 8, 16, and 24 bit boundaries. The structure of the rDNS matches these boundaries since it uses a name component for each eight-bit octet of the address. That is, if an organization were allocated the IP address range 12.34.xx.xx, they'd also be delegated the rDNS for `34.12.in-addr.arpa`.

As soon as CIDR came along, allowing IP address space allocation at arbitrary bit boundaries, rDNS name delegation became a mess. For ranges larger than /24, delegation is straightforward but tedious. For a /22, the address space holder is separately delegated each of the four /24's in the /22, e.g., for 12.34.56/22 the holder would get the rDNS for 12.34.56.x, 12.34.57.x, 12.34.58.x, and 12.34.59.x, which are `56.34.12.in-addr.arpa` through `59.34.12.in-addr.arpa`.

But there's no practical way to delegate the rDNS for ranges smaller than a /24 using the normal DNS delegation technique. (In theory one could add a zone cut for each individual rDNS name, making each its own zone, but that would be hundreds of tiny zones, something that DNS servers don't handle well.) Since the names in the rDNS happen to be known in advance, one per IP address, and the number of names in a /24 is only 256, people have kludged around it with groups of CNAMEs, one per allocated address, aliased to names with an extra component to allow normal DNS delegation to work. For example, if someone were allocated the /27 range 12.34.56.64 through 12.34.56.95, create these CNAME aliases:

64.56.34.12.in-addr.arpa CNAME 64.64-95.56.34.12.in-addr.arpa
...
95.56.34.12.in-addr.arpa CNAME 95.64-95.56.34.12.in-addr.arpa

Then do one zone delegation of `64-95.56.34.12.in-addr.arpa`. This hack was described in RFC 2137 in 1998, so it's well established, but it's still pretty ugly.

IPv6 avoided this particular problem by noting that the smallest plausible allocation boundaries for the huge IPv6 addresses are four-bit groups, and making IPv6 rDNS names have a separate component for each four-bit nibble, so it's always possible to make a zone cut that matches an IP space allocation. The moral here is to think hard about what you might want delegate in the future when designing your name space.

Conversely, it is also possible to over-decompose DNS names. DNSBLs use the same multi-component name structure as rDNS, but gain little benefit from it other than the range wildcards discussed above. Since each DNSBL is invariably managed by a single organization, in the common case that DNSBLs are served using DNS servers that don't use wildcards, there's no benefit to decomposing the name into multiple components. It would work just as well to use names like `127-0-0-2.dnsbl.example`, or for IPv6, a 32 character hex number like `20010470lf0711260000537061 6d6d79.dnsbl.example`.

Applications should not assume that any particular name boundary is a delegation point, in particular, that the boundary between a top-level and second-level domain is one. Different domains have different policies, and they are under no obligation to tell you what those policies are. While the management of `com` and `google.com` are different, `uk` and `co.uk` are the same, while the third level `google.co.uk` is someone else. It's also unwise to assume that if a TLD has delegated some names at the second or third level, that all of their delegations are at the same level. For example, `ca`, `qc.ca`, and `montreal.qc.ca` are all run by CIRA, but `google.ca` is not.

In the cases above, there's a zone cut at the place where the management changes, but zone cuts do not necessarily represent administrative boundaries either. In many cases a single organization will divide up its own namespace into multiple zones for its own administrative convenience.

There are places like publicsuffix.org that try to keep lists of delegation points for TLDs, but they are only as accurate as the data people contribute to them, which is of highly variable completeness.

Global data consistency

Many databases go to great effort to present a globally consistent view of the data they control, since the alternative is to lose credit card charges and double-book airline seats.

The DNS has never tried to to that. The data is roughly consistent, but not perfectly so.

Multiple servers and caches

Most zones have multiple DNS servers. The servers usually do not update their copies of the zone data at the same time, so one server may have slightly newer data than another. RFC 1034 describes the method that many zones use to keep in sync, with one server as the master, and the rest as slaves use AXFR to copy over updated versions of the zone as needed. The zone's SOA record contains some time values to tell the slaves how often to check. Although in theory one could crank the refresh interval down to one second, in practice refresh intervals are an hour or more. Hence it is quite common for the authoritative servers to be slightly out of sync.

Furthermore, DNS caches remember previous queries, up to the TTL provided by the server when the query was answered. If the data changes, a cache will not re-query until the TTL expires. Although it's possible for a server to set the TTL to zero, meaning not to cache the data, typical TTLs are from minutes to days. Adding to the excitement, caches do not always honor the requested TTL, sometimes applying minimum or maximum retention times.

As a result, "flash cuts" where the DNS changes and all clients can immediately see the new data don't work. Instead, DNS changes need to be done in ways that allow for the old and new data to coexist for a while. For example, when changing the IP address of a server, rather than trying to crank the TTL down to zero and forcing all the zone's servers to update at once, it's a lot easier if one can run the service in parallel on the old and new IP addresses long enough for all of the servers to update on their normal schedule, and for cached entries to expire.

Deliberately inconsistent DNS

Some DNS servers deliberately return different answers to different clients. The primary reasons are for "split horizon" DNS and for load sharing.

Split horizon means that clients on different networks get different answers. Most often it's that clients within the organization's own network get a larger set of names than the rest of the world does, or

names resolve to addresses on the internal network while external clients get addresses on a firewall system. I've also used split horizon to deal with broken clients or caches that send high volume streams of bogus queries, sending them delegations to name servers on non-existent networks.

Load sharing in its simplest form involves rotating through a set of records in responses, so that clients are spread across a set of mirror web or mail servers. In more sophisticated forms, the DNS server tries to guess where the client is, and tries to return the address of a server topologically or geographically close to the client.

Split horizon DNS is somewhat defensible as a legitimate DNS configuration, since the responses to each client are consistent, and for the usual inside/outside split, organizations should know what addresses are on their own network, with appropriate router configurations to keep out forged outside traffic purporting to be from their own network.

Load sharing shouldn't hurt anything (much) so long as the server is prepared for its guesses about the client's location to be completely wrong. As an obvious example, people all over the world use Google's public DNS cache. At this point, DNS-based load sharing is probably a necessary evil, but given the ever more convoluted topology of the Internet, it is a poor idea to design new applications that depend on it.

Large data in the DNS

Most DNS queries are made via UDP, a single packet for query and a single packet for the response, with the packet size traditionally limited to 512 bytes. This limits the payload of the returned records in a response packet to about 400 bytes, after allowing for the overhead in a DNS response. Many clients and caches (about half in my experience) support the EDNS0 extension, which lets the client specify the maximum packet size, usually 4096. The length fields in DNS records are 16 bits, so the absolute limit on a packet or a record is 64K bytes. The DNS spec says that if a response is too big to fit in a UDP packet, the server sends a partial response with the truncation bit set, which tells the client to retry the query over TCP.

Until a few years ago, it was rare to see a DNS response that didn't fit in a 512 byte packet other than for AXFR and IXFR. Now, as DNSSEC is starting to become more widespread, larger packets are becoming more common, since DNSSEC adds a great deal of signature material to every response and requires EDNS0.

Although it is possible in principle to put large chunks of data into the DNS, up to 64K, the bigger a response is, the less likely it is to be returned reliably. Some DNS servers still don't support TCP, far too many firewalls don't allow TCP DNS traffic, a few broken firewalls

won't pass DNS packets bigger than 512 bytes, and DNS caches are not tuned to cache large data well. I've seen the occasional proposal to store chunks of XML in the DNS, but they generally seem to be from people who want to see XML everywhere.

Since a TCP query requires a UDP query and response, which includes the truncation flag, and a subsequent TCP session, a more sensible way to handle large data is to put a pointer to the data such as a URI in the DNS which can be returned via UDP, and then have the application use another scheme such as HTTP to retrieve the large data. That's about the same amount of net traffic (a UDP round trip followed by a query and response via TCP), but HTTP servers and HTTP caches are designed to handle large data that DNS servers and caches aren't.

Overloaded record types

All the records in the DNS are strongly typed. Each record includes an RRTYPE, a small number, which defines both the format of the record and what the record means. It is possible and common to have different record types with the same format, but different meanings. For example, the NS (name server), CNAME (canonical name), and PTR (name pointer) records all contain a single domain name, but their semantics are quite different.

The original plan was that as people came up with new kinds of data to store store in the DNS, they'd define and add new RRTYPEs. But for a variety of good and bad reasons, many DNS applications have reused existing RRTYPEs rather than registering new ones. Until it becomes a lot easier to add new types to DNS servers and provisioning systems, this seems unlikely to change.

When an RRTYPE is reused, there are two ways to tell the new use from the original use, by name and by value. The most common technique is to put the RRs with reused types at names where they are unlikely to collide with the original usage. For example, DNSBLs reuse A records, but they are in their own branches of the namespace where host names never occur, so in theory there should be no collision. In practice, collisions occur all the time, when the domain name of an abandoned DNSBL is re-registered by someone else who parks it with a wildcard A record, thereby causing the abandoned DNSBL to appear to list everything. Well-written DNSBL client code can defend against this by making the recommended checks in RFC 5782 intended to detect inappropriate wildcards, but it remains a problem.

A variant of this approach is to put reused records at prefixed names. The data for an attribute of `example.com` is stored at `_attribute.example.com`, for a suitable attribute. This approach has been reasonably successful in DKIM and VBR, putting TXT records at prefixed names, where the contents have a format defined by the

DKIM or VBR application. But they are still subject to collisions due to unwise wildcards, and suffer from the related name problem I address below.

The other way to manage reused records is by value, most often by putting a specified string at the beginning of a reused TXT record. For example, DKIM records usually start with "v=DKIM1" and SPF records with "v=spf1". This also works reasonably well as a way to deal with inappropriate wildcards, and allows multiple applications to coexist at the same name, but at the cost of extra application logic to check the records and ignore the ones without the string the application expects. It also scales poorly. A request for the TXT records at a name always returns all of the TXT records, so the responses will be cluttered with unrelated records as more applications add them. (The design of the DNS anticipated this problem, which is why requests normally ask for a particular record type rather than for all records at a name.)

Related names are not related

A poorly appreciated aspect of the DNS is that there is no inherent relationship between similar looking names. That is, to a person the names `example.com` and `www.example.com` may look obviously related, but to the DNS, they're just two different names. Names do exist in a tree, and as we saw above, if a name exists in a tree, all names above it (closer to the root) exist, even if there is no data at some of those names. Conversely, if a name does not exist, all possible names below it don't exist either. But if two names both exist, the DNS has no good way to describe the relationship between them.

The way to make two names "the same" in the DNS is to use a CNAME, for a single name, or a DNAME, for all the names below a name. When a DNS cache receives a CNAME result in response to a non-CNAME query, it will repeat the query for the target of the CNAME, and return both the CNAME and the target record in the response to the original request. A DNAME is sort of a generalized CNAME, and says that the names in the tree below one name are aliases for the tree below another name. That is, if `a` is a DNAME for `b`, then `x.a` is an alias for `x.b`. DNAMEs have been around since 1999, but for the benefit of caches that still don't understand them, when a server returns a DNAME answer, it also synthesizes suitable CNAMEs as needed when replying for a name below the DNAME.

CNAMEs have been around since the dawn of the DNS, but they can only partially make one name a synonym for another. A CNAME only provides an alias for its own name, and not for any prefixed name. So if `foo.example` is a CNAME pointing to `bar.example`, queries for `_prefix.foo.example` won't work unless there are additional CNAMEs to point to every prefix defined for `bar.example`. This

makes the provisioning considerably more difficult, since it means every time you add or change a prefixed name record, you have to know everyone who's pointing a CNAME at your name, and tell them to add or change the prefix, exactly the sort of error-prone manual processing that computers were supposed to avoid.

When a DNS request is satisfied via CNAMEs, the client can see the chain of CNAMEs. So when a DNS client or cache looks up a prefixed name and finds nothing, it could, in principle, make a query for the base name, see if there's a CNAME, and if so make a prefixed query for the target, in this case `_prefix.bar.example`. Prefixed names have existed for 15 years (since the invention of SRV records in RFC 2052) and nobody to my knowledge has done anything special about CNAMEs and prefixes, so it seems a little late to start now. DNAMEs alias the entire subtree below a name, which deals with the prefixed names, but doesn't alias the name itself, limiting its utility for aliasing specific names and their variants.

Names outside the DNS

In the early years of the DNS, domain names were typically resolved to A records which were used to identify a host running a service. With the notable exception of e-mail, once the host was identified, the name no longer mattered. Another way to look at it is that for traditional services like telnet and FTP, servers don't know or care what their names are. If you want to give a Telnet server a new name, just point a new A or CNAME record at it, and it'll work.

E-mail was and is different. While a mail server doesn't need to know what its name is, it definitely needs to know for what domains it handles mail, i.e., what MX records should be pointing at it. This means that CNAMEs pointing at a name with an MX generally won't do what you'd want.

In the early World Wide Web, servers didn't need to know their names, since each logical server had a separate IP address. As the Web grew rapidly, the virtual server feature was added to conserve IP addresses. It allows many names to resolve to the same IP address, with the Web server returning different content for different names.

What mail and the Web have in common here is that even after all of the DNS queries are done, and the names are all resolved to IP addresses, and a TCP session is active to the Web or mail server, the domain names appear in the data stream. In some cases, particularly mail, this seems to be unavoidable, but it makes the provisioning much more difficult, as the DNS and the application configuration need to be kept in sync.

Prefixed names (which we mentioned above) can present problems when a server has multiple names. The key observation is that CNAMEs work for servers that don't know their names, and prefixes work for servers that do. We had an argument in the Anti-Spam Research Group a while ago about possible ways to tell a user mail program where to send spam button complaints. One possibility was to put the info in a record at a prefix of the POP or IMAP server name like `_report.server.example`. I didn't think that would work very well because POP and IMAP servers don't know their names, and it is common in hosted mail systems to use CNAMEs for the server names. A server can have thousands of CNAMEs pointing at it, all of which would need an extra manually configured CNAME for each prefix in use.

On the other hand, nobody ever points a CNAME at a name with an MX because it wouldn't work without configuring the mail server to know about the CNAME, and once it's that complicated, everyone adds another MX instead. Since there aren't any CNAMEs used to route mail via MXes, prefixed names for DKIM work just fine in practice.

Web servers are sort of a middle case. People use lots of CNAMEs to point to web servers, but in IPv4 at least, they use them in very stylized ways since most web servers use virtual names so again the DNS and the server have to be coordinated.

To the extent applications can avoid this need to coordinate DNS and non-DNS configuration, and have the results of the DNS queries completely identify the desired service, they will work much more smoothly with the DNS. Ways to do this are to have adequate IP addresses (there's no need for virtual web servers in IPv6), to use SRV so that different services can be on different ports, or perhaps to use new RRTYPEs that include a token that the client can present to the server to tell which DNS record led it to the server.

The DNS technical community has been wrestling with the question of how to make two DNS trees "the same" for quite a while. ICANN now has a process to issue non-ASCII top-level domains in addition to the traditional two letter codes, such as .РФ in addition to .RU. In many cases, the new domain is supposed to be the same as the existing one, for some version of "same". After going through the arguments several times, I came to the conclusion–which I think was shared by most of the participants–that nobody adequately understands what that sameness would mean technically. Whatever it might turn out to be, it's unlikely that the DNS could provide it, even with some improved version of CNAME or DNAME.

Since it's now possible to have domain names in any script, the number of nominally equivalent domain trees is likely to increase, and if the equivalence is supposed to be component by component, the number of nominally equivalent names will become huge. (That is, if `A` and `B` are supposed to be equivalent to each other, and `C` and `D` to each other, then `A.C`, `A.D`, `B.C`, and `B.D` are all the same.) So what could we do in the DNS to make the configuration easier?

With no changes to the DNS at all, applications could handle CNAME or DNAME aliases. The result from a DNS lookup that's redirected via CNAMEs includes all the CNAMEs in the chain, so it would be easy enough for an application getting a request for a domain it doesn't recognize to do a suitable DNS lookup (MX for mail, A or AAAA for web, etc.) and see if the unknown name is a CNAME for one it does know. This has been possible for decades, but to my knowledge nobody's ever done it, suggesting that whatever problem it solves isn't worth solving. Beyond the programming effort, this also presents a security issue in that anyone can now point a CNAME at your server and make it handle a domain you might not have wanted to handle.

The most plausible solution so far to that problem is a new CLONE record that is sort of the inverse of CNAME, in which the primary name lists all of the other equivalent names, and the name server arranges to handle all of them. DNS lookups for any of the equivalent names also provide the primary name, and the DNSSEC signature is adequate to prove that the equivalence is authorized by the primary.

This would require new coordination whenever one server delegates a subtree to another, in that the second server would have to inherit all of the cloned names that apply to the name delegated from the first server, but it's not hard to imagine ways that they could coordinate automatically. So far this is all hypothetical, and there's still a lot of work to be done getting DNSSEC more widely deployed before anything like CLONE is likely to happen, but it's one of the few ways in which the DNS has any chance of interesting extensions.

Conclusion

The DNS is now over 25 years old. Considering its age, and how large it has become, it works remarkably well. (If you'd told people in 1987 that the `.COM` zone would have 100,000,000 entries, they'd have laughed at you.) But it still has design issues to be solved for the Internet of the 21st century.

Revision date: 2012/06/17 16:32:55