

CHAPTER 21:

NAMING

There are two systems of naming that are very widely encountered.

The Domain Naming Service always known familiarly as *DNS*, is the glue that ties the Internet together, as far as applications are concerned. Most users just see it in the familiar "dot com" form as sort of a backwards syntax for names, perhaps most useful as a mnemonic device.

Much more importantly for the operation of the Internet, and for network programming in particular, is that it guides us in many ways toward the proper interconnection of resources, by telling us what resources are available and where they are located.

It can be very accurately modelled as a means for building an ordinary function that maps from a name and a resource type to a list of resources that are associated with that name, by a map whose values are universal across an entire network such as the Internet. Thus no matter where you ask the question from and no matter who you ask the answer will be the same.

Thus DNS is intended to provide naming in situations where you need to be able to identify a resource from anywhere on the Internet. In that sense it can be thought of as most appropriate for naming network-related resources such as locations where a service or names that should be used to request resources.

The Lightweight Directory Access Protocol familiarly known as LDAP, provides answers to very specific queries made to very specific servers. The "names" that a client requests information about are actually strings of attribute values and their settings, and the server is typically providing a directory service for a particular organization.

Thus you might ask an organization if it has a phone number for an individual named "doug" who belongs to a department named "computer science". You might think that such a query should also have the same answer wherever it is asked, and that will be so provided you contact the appropriate server. But another organization and thus another server might have another "doug" in another "computer science" department, and the answer then would almost surely be another phone number.

Thus LDAP is intended for resources related specifically to the structure of a particular organization, and its queries are answered by servers managed specifically by units of that organization. It is becoming relatively common to use LDAP for email addresses of individuals who can be identified in some other manner, and for administratively assigned entities such as public keys of machines or individuals.

It is not always clear how to decide whether a particular resource is better named through DNS or through LDAP.

DOMAIN NAME SYSTEM

In particular in the *Peer-Provider-Protocol* framework DNS is almost always embedded in any addressing scheme defined by a protocol and implemented by a provider for establishing a connection to a peer. Thus to specify that you wish to communicate using standard mail protocols you direct your mail to `doug@kisadvice.com`, where the last part of that name is in fact the DNS name of a node on the world-wide Internet. And to specify that you wish to communicate with the web-server at a particular node you specify the universal resource locator `http://kisadvice.com/`, where the DNS name specified is again that of a node on the Internet, this time specifically stating that the HTTP protocol should be used.

The basic specifications for DNS are given in *RFC-1034*, which shows the desired behavior of the protocol, and *RFC-1035*, which shows the attributes of records and packets.

What does DNS do?

DNS finds resources from a name.

The DNS is a general system for discovering various kinds of resources that have a domain name associated with them. One of those kinds of resources is an Internet address, but there are others that we might also need to know.

You can learn not only the address of a node in a domain, but also the names and addresses of nameservers for the domain, who to send email to if there are problems with the domain, and actual names associated with addresses (through treating the address as a kind of "reverse name").

Having a name with hierarchical structure allows the authority for and administration of changes in this association of names with resources to be centralized in whatever way works the best for your problem.

DNS provides authority for name-resource maps

An authority for a DNS record is any server that is designated by the administrator of the domain it pertains to as having authoritative information. This typically means that the server is under the control of that administrator, or perhaps managed by others under authoritative supervision.

But how do you know that a server is authoritative?

The authority is derived hierarchically, based on successive suffixes of the name.

Thus authority for *kisadvice.com* is granted by the *.com* nameservers, and authority for *.com* is granted from the *root-nameservers*, which are authorities for the root name ".".

A name is a series of suffixes where the divider is a "." character, beginning with an empty suffix that represents the "root" name. There are root name servers which are authoritative for that domain. At the moment there are 13 of them and they run under the supervision of the Internet management.

The distributed implementation of this map consists of:

- DNS servers which when queried provide a reply showing any records they manage regarding the name in the query
- DNS resolvers that query servers that might have the answer or resolvers that might be more likely to find the proper record from a server. It may behave recursively, or iteratively by telling the requestor the name of a resolver more likely to have an answer

What the root name servers know is authoritative name servers for the various top level generic domains such as com, edu, net, and the top level country domains. Each of these top level servers in turn knows authoritative name servers for the various second level domains underneath such as *microsoft.com*, *marquette.edu*, *dougharris.net*.

Each authoritative server for a second level domain knows the authoritative server for any subdomains of it. In fact the way a subdomain is created is that a server for the initial domain delegates authority to a server (perhaps itself) for the subdomain.

DNS predicts the need for other resources

A DNS resolver not only answers your query, it usually gives additional information as well. In particular the reply from such a resolver will consist of the original query (in case your software forgot!), one or more answers to the question posed (if any are available), records showing where the authority came from for the answer, and additional records that answer queries the resolver-thought you might be asking next about the answers it has returned so far.

In the examples show earlier I actually left out most of the reply, which consisted of *authority* and *additional* records in addition to the *answer* records shown.

It is fairly typical, by the way, that if some resource is mentioned in the authority section, a record, called "*glue*", will be provided in the additional section. This is in fact part of the behavior standards for DNS.

Glue records are sometimes misused, by a malicious DNS server (yes, they exist, there is no formal authority that constantly polices nameservers). The misuse depends upon improper administration of the resolver on some other machine.. The malefactor places a glue record giving the address of a machine they control, as the address of some important nameserver. The cache has now been "poisoned", and if this resolver (improperly) returns this additional record into an answer record returned in a reply, the querying machine now will be directed toward the imposter machine.

Here is a sample reply to a query (made by a program we will examine later in this chapter). The query asked a DNS server to give what information it has regarding Internet address records for the domain name *kisadvice.com* (this is the web site for this book, and the server *ns4.netpik.com* is one of the servers that is officially registered as a server for that site).

```
009 Provider: dns Server <134>
010 id1: QUERY R:AA:-T:RDRA: OK: queries 1 ans 1 auth 2 addl 2
011 Answer 1
012 Name:kisadvice.com. Type:A Class:IN TTL:0y-0w-0d-6h-0m-0s Length:4
013 209.53.126.245
014
```

Listings such as the previous one will be used throughout this chapter to illustrate what we say. This listing was made using the DumpPacket software from Chapter 6, together with parsing programs for the DNS layer which are discussed in this chapter.

They will always show the provider as dns Server on the first line, meaning that the UDP layer identified this as something coming from port 53 on its machine, which is the port at which a DNS server normally operateis. The second line will have various information we need not consider right now, but if you look toward the right of the second line (line 10 above) it shows queries 1 ans 1 auth 2 addl 2, which means that there was 1 query sent to the nameserver, and it returned 1 answer, along with 2 authorities for that answer and 2 additional records that it thought would be helpful.

The third line (line 11 above) will usually show Answer 1, if there were answer records. Once in a while there will be no answer, and any authority records that are sent are other nameservers that should be consulted instead of the one that returned this reply.

The next line is the actual answer. *Name* is the domainname that you required about (*kisadvice.com* in the previous example), and *Type* is the type of record that was asked for. Class will always be **IN** (Internet), **TTL** is the time-to-live of this record, which tells how long you can keep it in a cache, and **Length** is the length of the encoded record, which due to name compression may end up shorter than its value after decoding. In the previous case the record queried for was an **A** record, and such a record is an Internet address, and always has length 4.

Listing 21-1: reply to a query about kisadvice.com

This reply marked itself as authoritative (the *AA* flag on line 10), and gave an answer record (lines 11-13) which shows that the address is **209.53.126.245**. It also gave additional information, which is not included here.

What can you ask?

You can ask for any kind of record that has been stored in the server. There is a set of standard types of record that are constantly requested, but a server can be set up to serve other types that are not part of the standard.

<i>value</i>	<i>abbreviation</i>	<i>purpose</i>
1	<i>A</i>	an Internet address record
2	<i>NS</i>	a nameserver record
5	<i>CNAME</i>	a canonical name record
6	<i>SOA</i>	a start-of-authority record
12	<i>PTR</i>	a pointer record
15	<i>MX</i>	a mail exchanger record

Table 22: DNS record types

nameserver (*NS*) records

telling which nameservers have the DNS records for the argument name.

Here are the two nameserver records returned when I queried for the NS records of `kisadvice.com`. As you can see from line 13, there were also 2 additional records, which I did not show here. They gave the address records for the 2 nameservers listed.

```
012 Provider: dns Server <118>
013 id1: QUERY R:AA:-T:RDRA: OK: queries 1 ans 2 auth 0 addl 2
014 Answer 1
015 Name:kisadvice.com. Type:NS Class:IN TTL:0y-0w-0d-6h-0m-0s Length:16
016 ns4.netpik.com.
017 Answer 2
018 Name:kisadvice.com. Type:NS Class:IN TTL:0y-0w-0d-6h-0m-0s Length:15
019 ns2.inetwave.com.
```

Listing 21-1: NS records for kisadvice.com

On lines 14-16 the server tells me of the nameserver *ns4.netpik.com*, and on lines 17-19 it tells of the nameserver *ns2.inetwave.com*.

This NS query and reply, is the formal basis for much of the security of the Internet. If you want to connect to "*somestore.com*" and spend your money, you would like to be sure that you are not instead getting "*hijack.who.knows*" instead. So as we will see in more detail your machine can follow a chain of authority to be sure it is given the correct address.

Of course I recommend also checking that they have the correct sign on the door when you get there! And perhaps doing some additional checking with the sales folks there, and maybe your bank.

address (A) records

showing the Internet address[es] corresponding to the argument name Contrary to the thoughts of some, there may be many address records corresponding to the same name, and they may represent different interfaces on a single machine, or physically different machines.

Here are the 6 address records provided when I queried for the *A* records of *microsoft.com*. Again, line 13 shows that in addition to the 6 answers, there were also 6 authority records and 6 additional records, showing the addresses of the machines involved.

```
012 Provider: dns Server <363>
013 id1: QUERY R:-A:-T:RDRA: OK: queries 1 ans 6 auth 6 addl 6
014 Answer 1
015 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
016 207.46.230.220
017 Answer 2
018 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
019 207.46.230.219
020 Answer 3
021 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
022 207.46.230.218
023 Answer 4
024 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
025 207.46.197.113
026 Answer 5
027 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
028 207.46.197.102
029 Answer 6
```

030 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
031 207.46.197.100

Listing 21-2: A records for microsoft.com

In lines 14-32 the reply gives 6 address records. This makes sure you understand that there may not be a single address for a particular name, and, as we shall see later, there may not be a single name for an address. The relation is literally many-to-many.

Notice in lines 15,18,21,24,27, and 30 the TTL (time-to-live) value of 32 minutes and 11 seconds. This means that the nameservers could be changed and within that time every machine on the net would be notified!

This *A* query is by far the most common kind of query that is made. You might think "since the correspondence of names and addresses is pretty much one-one, why don't we just keep track of this locally, and not continue to have to ask". And the answer is that the name "*www.busysite.com*" might frequently be moved from one address to another, for maintenance or for other reasons, and in fact asking for an address for that name might give you one of many different servers, which are being "rotated" in order to distribute the load.

pointer (*PTR*) records

using a name corresponding to the argument name, but specially constructed as a "reverse name" or "pointer name". It is just the dotted-decimal representation of the IP address, reversed, with the suffix *in-addr.arpa*. That particular suffix is reserved by the DNS system for use in constructing names with meaning particular to the system.

The following query asks, in the DNS way, for the name corresponding to the internet address *134.48.4.129*. The "reverse name" is *129.4.48.134.in-addr.arpa*.

```
011 Provider: dns Server <153>
012 id1: QUERY R:AA:-T:RDRA: OK: queries 1 ans 1 auth 3 addl 1
013 Answer 1
014 Name:129.4.48.134.in-addr.arpa. Type:PTR Class:IN TTL:0y-0w-1d-1h-0m-0s Length:22
015 spectral.mscs.mu.edu.
016
```

Listing 21-3: PTR records for 129.4.48.134.in-addr.arpa

The reply (line 15) shows that this machine has the name *spectral.mscs.mu.edu*. There were also three authority records, giving the nameservers which are authoritative for this name, and one additional record giving the *A* records for *spectral*, which should include *134.48.4.129*.

Servers which receive mail often perform a *PTR*-query for the address from which mail is being received. You can often see the result in the headers if you are brave enough to look, and have the patience to parse what they say. When a client machine connects to a mail server it provides a name (with syntax *HELO some.maybe.phony.name*). The server will have been given the Internet address the connection came from, and it looks this up via a *PTR* query to get that as the "official" name. Thus although you can connect to a mail server and tell it your machine is "*whitehouse.gov*", using *PTR* records it can find out that your machine is really "*no.where.net*".

canonical (*CNAME*) names

which are domain names that the argument name is a nickname for. This is sometimes misunderstood: the *CNAME* record gives a canonical name, which is a real domain name and will usually have an *A* record, while the argument is an "alias" which for convenience may be used in some settings in place of the canonical name.

If a name has a *CNAME* record for it, then according to the standard there should be no other record for the name. In other words you should expect to obtain any other information needed by finding the canonical name and obtaining records for it.

Names such as "*www.some.where*" are often aliases, and the canonical name may be needed, especially in official logs and the like. Registering the name as an alias means that it can be moved at will to another server.

Here is the reply to my request for the canonical name of *www.microsoft.com*, which shows that its canonical name is *www.microsoft.akadns.net*.

```
012 Provider: dns Server <319>
013 id1: QUERY R:-A:-T:RDRA: OK: queries 1 ans 1 auth 6 addl 6
014 Answer 1
015 Name:www.microsoft.com. Type:CNAME Class:IN TTL:0y-0w-0d-1h-57m-12s Length:26
016 www.microsoft.akadns.net.
017
```

Listing 21-4: querying the canonical name of www.microsoft.com

Notice that in line 13 no less than 6 authorities are cited for the single answer record given. These six authorities are of course the 6 nameservers that *Microsoft* lists for the domain *microsoft.com*.

mail exchanger (MX) records,

whose value is the domain name of a mail exchanger host that should handle mail intended for the argument name, preceded by a *precedence* value that indicates the order in which the exchangers should be tried..

Often other systems are not formally connected to the Internet, or connected through a firewall that deliberately does not allow any machine on the inside to accept mail directly from the outside. The manager of the domain space for such machines can designate another machine as the one which should receive mail. It will probably then forward it by other means to the proper recipient, or very frequently will hold it for them to access in some other manner.

Here is a reply to a query for the mail servers that handle mail for one of my email addresses, *doug@wi.rr.com*. It gives the satisfying authoritative answer that there are 8 machines ready to accept email to this address (and only one person to read the mail :-). The machines that should receive the mail are marked as *Exchanger*, and the *Preference* value shows the order in which they should be contacted. In this case that order does not matter, any one will do

```
011 Provider: dns Server <493>
012 id1: QUERY R:AA:-T:RDRA: OK: queries 1 ans 8 auth 4 addl 12
013 Answer 1
014 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:21
015 Preference: 50 Exchanger: lamx01.mgw.rr.com.
016 Answer 2
017 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:11
018 Preference: 25 Exchanger: kcmx01.mgw.rr.com.
019 Answer 3
020 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:11
021 Preference: 50 Exchanger: flmx01.mgw.rr.com.
022 Answer 4
023 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:12
024 Preference: 50 Exchanger: nycmx01.mgw.rr.com.
025 Answer 5
026 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:11
027 Preference: 50 Exchanger: ncmx01.mgw.rr.com.
028 Answer 6
029 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:11
030 Preference: 50 Exchanger: vamx01.mgw.rr.com.
031 Answer 7
```

032 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:11
033 Preference: 50 Exchanger: tnmx01.mgw.rr.com.
034 Answer 8
035 Name:wi.rr.com. Type:MX Class:IN TTL:0y-0w-0d-1h-0m-0s Length:11
036 Preference: 50 Exchanger: nymx01.mgw.rr.com.
037

Listing 21-5: .the mail exchangers for wi.rr.com

Notice in line 12 that it gives 4 authorities for these 8 answers, and that there are 12 additional records. We will shortly see that those 12 additional records are in fact the *A* records for the 8 mail server machines and the 4 authority machines. DNS knows what you would ask it next, if it did not provide these additional records!

example use of DNS with email

DNS is used in many ways in email, http, and every other kind of exchange at the level of application protocols. You seldom need to invoke it directly, but understanding what information it provides, and in particular what information it actually defines, is vital to tracking what is happening. And since you typically only track what is happening when it is not to your liking, a good understanding of DNS, developed before you really need it, is a good addition to the knowledge that will keep you calm when everyone else is not!

The following capture of an email message from *doug@studsys.mscs.mu.edu* (a university account) to *doug@wi.rr.com* (my personal account) shows that one of the exchangers above, (*kcmx01.mgw.rr.com*) actually received the mail from *studsys*, and held it until it was transferred (through *Outlook Express*) to my personal machine. Line 3 shows that the mail server recorded both the name of the sending machine, and the address from which the mail came. They agreed in this case, in that the *PTR*-lookup of the address gave the same name as the one the client machine was using.

```
001 Received: from kcmx01.mgw.rr.com ([24.94.163.190]) by mail8.wi.rr.com with Microsoft
SMTPSVC(5.5.1877.537.53);
002 Sun, 10 Mar 2002 13:12:58 -0600
003 Received: from mscs.mu.edu (studsys.mscs.mu.edu [134.48.4.15])
004 by kcmx01.mgw.rr.com (8.11.4/8.11.3) with SMTP id g2AJCHI12281
005 for <doug@wi.rr.com>; Sun, 10 Mar 2002 14:12:17 -0500 (EST)
006 Received: (qmail 26762 invoked by uid 199); 10 Mar 2002 19:07:59 -0000
007 Date: Sun, 10 Mar 2002 13:07:59 -0600
008 From: John Douglas Harris <doug@studsys.mscs.mu.edu>
009 To: doug@wi.rr.com
010 Subject: Test from studsys
011 Message-ID: <20020310130759.A26743@studsys.mscs.mu.edu>
012 Mime-Version: 1.0
```

013 Content-Type: text/plain; charset=us-ascii
014 Content-Disposition: inline
015 User-Agent: Mutt/1.2.5i
016 Return-Path: doug@mscs.mu.edu
017
018 Just some mail from doug@studsys.mscs.mu.edu to doug@wi.rr.com, to get some header examples for the net-working book.

Listing 21-6: mail from studsys.mscs.mu.edu to wi.rr.com.

Notice (lines 3-4) that the mail was actually sent by *studsys* to the mail exchanger of lowest preference (which is the one that should be tried first). The scenario is that studsys asked for *MX* records for *wi.rr.com* before it actually sent the mail, and followed the DNS protocol for interpreting such records, which is given in *RFC974*.

administrative (SOA) information

is kept by each authoritative server for the domain, and tells an email address of an individual responsible for managing the DNS records, and some information about the default TTL for its records.

This record is used if you find that someone at a site is disrupting your machines, through sending disruptive mail or worse. The victim can look up the *SOA* record and find the email, at least, of the administrator of the site. Well, its a place to begin!

010 Provider: dns Server <173>
011 id1: QUERY R:-A:-T:RDRA: OK: queries 1 ans 1 auth 2 addl 2
012 Answer 1
013 Name:kisadvice.com. Type:SOA Class:IN TTL:0y-0w-0d-5h-15m-56s Length:57
014 RName: eadlersp@topchoice.com
015 MName: ns4.netpik.com.
016 serial: 9982801
017 refresh: 21600
018 retry: 3600
019 expire: 604800
020 minimum: 21600
021
022 Authority 1
023 Name:kisadvice.com. Type:NS Class:IN TTL:0y-0w-1d-23h-15m-56s Length:2
024 ns4.netpik.com.
025 Authority 2
026 Name:kisadvice.com. Type:NS Class:IN TTL:0y-0w-1d-23h-15m-56s Length:15
027 NS2.INETWAVE.com.
028
029 Additional 1
030 Name:ns4.netpik.com. Type:A Class:IN TTL:0y-0w-1d-23h-58m-52s Length:4
031 209.52.182.122

032 *Additional 2*
033 *Name:NS2.INETWAVE.com. Type:A Class:IN TTL:0y-0w-1d-23h-58m-52s Length:4*
034 *209.52.182.72*
035

Listing 21-7: start of authority record for kisadvice.com

A very important piece of information given in the SOA record is the serial number (line 16 in the listing above). It can be compared with a previous serial number to see if the records have been updated since they were last retrieved. Many administrators just use the current date in some form for this number

.Line 14 tells you if you are unhappy with a DNS-matter related to kisadvice.com you can send email to the address given there. Please don't do it, contact me using *doug@kisadvice.com* or *doug@wi.rr.com* instead!

Just for completeness this listing shows everything that was received in the packet: the answer record, the two authority records showing the official nameservers that provided the answer (which should be the same answer for each if the nameservers are properly administered), and the address records for the two nameservers.

Why does DNS do it?

So why do we need both names and addresses? It seems at first glance that DNS names are equivalent to addresses, and are just there because people don't like to deal with numbers (and will like to even less when version 6 of the IP protocol arrives). As the examples above showed, however, *names* tend to be used to describe a service that might be provided, and *addresses* are a location at which that service currently is being provided.

They are different hierarchies -JDH

Here is the full information returned from a query asking the server *uwm.edu* about *A* records for the domain name *kisadvice.com*, which shows the same answer as the other server gave, along with 2 *authority* records, one showing the name of an official nameserver for that domain, and 2 *additional* records, giving the address records for the domain names in the authority reply.

010 *Provider: dns Server <144>*
011 *id1: QUERY R:AA:-T:RDRA: OK: queries 1 ans 1 auth 2 addl 2*
012 *Answer 1*
013 *Name:kisadvice.com. Type:A Class:IN TTL:0y-0w-0d-6h-0m-0s Length:4*
014 *209.53.126.245*
015
016 *Authority 1*

017 Name:kisadvice.com. Type:NS Class:IN TTL:0y-0w-2d-0h-0m-0s Length:15
018 NS2.INETWAVE.com.
019 Authority 2
020 Name:kisadvice.com. Type:NS Class:IN TTL:0y-0w-2d-0h-0m-0s Length:13
021 NS4.NETPIK.com.
022
023 Additional 1
024 Name:NS2.INETWAVE.com. Type:A Class:IN TTL:0y-0w-1d-18h-32m-9s Length:4
025 209.52.182.72
026 Additional 2
027 Name:NS4.NETPIK.com. Type:A Class:IN TTL:0y-0w-1d-8h-52m-10s Length:4
028 209.52.182.122
029

Listing 21-8: full reply to an A query for kisadvice.com

This reply shows 1 answer (since this particular name has only a single address), and 2 authorities (which are the two nameservers registered with the **.com** nameservers as authoritative for the domain name **kisadvice.com**, and 2 additional records, which are the "glue" records, that is, the addresses of the two nameservers given in the authority section.

Here is the listing of the (imaginatively-named) root nameservers. I obtained this listing by asking one of them, **g.root-servers.net** (whose name I happened to know) to give me the authoritative servers for ".".

001 /192.168.1.2:1750 querying g.root-servers.net:53:512
002 Received a packet of length 436 in a buffer of length 512
003 id1: QUERY R:AA:-T:RD-A: OK: queries 1 answers 13 authorities 0 additional 13
004 Answer 1
005 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:20
006 A.ROOT-SERVERS.NET.
007 Answer 2
008 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
009 H.ROOT-SERVERS.NET.
010 Answer 3
011 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
012 C.ROOT-SERVERS.NET.
013 Answer 4
014 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
015 G.ROOT-SERVERS.NET.
016 Answer 5
017 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
018 F.ROOT-SERVERS.NET.
019 Answer 6
020 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
021 B.ROOT-SERVERS.NET.

022 Answer 7
023 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
024 J.ROOT-SERVERS.NET.
025 Answer 8
026 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
027 K.ROOT-SERVERS.NET.
028 Answer 9
029 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
030 L.ROOT-SERVERS.NET.
031 Answer 10
032 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
033 M.ROOT-SERVERS.NET.
034 Answer 11
035 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
036 I.ROOT-SERVERS.NET.
037 Answer 12
038 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
039 E.ROOT-SERVERS.NET.
040 Answer 13
041 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
042 D.ROOT-SERVERS.NET.
043
044
045 Additional 1
046 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:20
047 A.ROOT-SERVERS.NET.
048 Additional 2
049 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
050 H.ROOT-SERVERS.NET.
051 Additional 3
052 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
053 C.ROOT-SERVERS.NET.
054 Additional 4
055 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
056 G.ROOT-SERVERS.NET.
057 Additional 5
058 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
059 F.ROOT-SERVERS.NET.
060 Additional 6
061 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
062 B.ROOT-SERVERS.NET.
063 Additional 7
064 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
065 J.ROOT-SERVERS.NET.
066 Additional 8
067 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
068 K.ROOT-SERVERS.NET.
069 Additional 9
070 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
071 L.ROOT-SERVERS.NET.
072 Additional 10
073 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
074 M.ROOT-SERVERS.NET.
075 Additional 11

```
076 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
077 I.ROOT-SERVERS.NET.
078   Additional 12
079 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
080 E.ROOT-SERVERS.NET.
081   Additional 13
082 Name: Type:NS Class:IN TTL:0y-0w-6d-0h-0m-0s Length:4
083 D.ROOT-SERVERS.NET.
084
085
```

Listing 21-9: the root name-servers

Notice the response is authoritative, since it came from one of the root-nameservers. It has 13 answers, and 13 additional records.

The reply is marked (line 3) as authoritative by the *AA* flag, which means the server which sent it claimed it to be authoritative. That, however, is not what makes it authoritative.

The reason I know the response is actually authoritative, is because I believe that the reply is from one of the official root nameservers. This list is very publically available, and the server I contacted is one that list.

top-level nameservers

The operational requirements for root namservers are given in *RFC2870*, which is also *BCP 40*. It discusses security issues, time-to-live for records, and general administrative matters.

You can obtain a file showing the root nameservers and those for other top-level domains, such as *com*, *net*, and *org*. from <ftp://rs.internic.net/domain/root.zone.gz>.

The IANA (Internet Assigned Numbers Authority) keeps a list of information about country code top level domains (familiarily called ccTLDs) on a website at <http://www.iana.org/cctld/cctld.htm>.

How is the delegation done?

This delegation is done precisely by establishing the *NS* records that we mentioned above. So a root server for com knows of every “dot-com” in the world, in the sense that it can give you an *NS* record which shows an authoritative server for that dot-com.

And an authoritative server itself maintains an *SOA* (start of authority) record for any domain it is authoritative for.

When a domain is registered this is done with the immediate parent domain: meaning that a second-level domain in particular is registered with an authority for its parent. Thus my web-site *kisadvice.com* was registered with an authority for *.com*, and *dougharris.net* was registered with an authority for *.net*: as it happens, both were the same organization. To register you provide information about the administrator, and give the name of at least two nameservers for the domain. These servers may be in the registered domain. In fact, to many who have studied the problem it is best if they themselves have names in that domain. Thus *marquette.edu* might have servers *dns1.marquette.edu* and *dns2.marquette.edu*, as well as perhaps servers that belong to other domains. Each such server was registered officially with the parent *edu* domain.

So the collection of names over which any name server is authoritative is always a “cone” of names with a particular suffix, with “sub-cones” removed for any further delegation to a server for a subdomain.

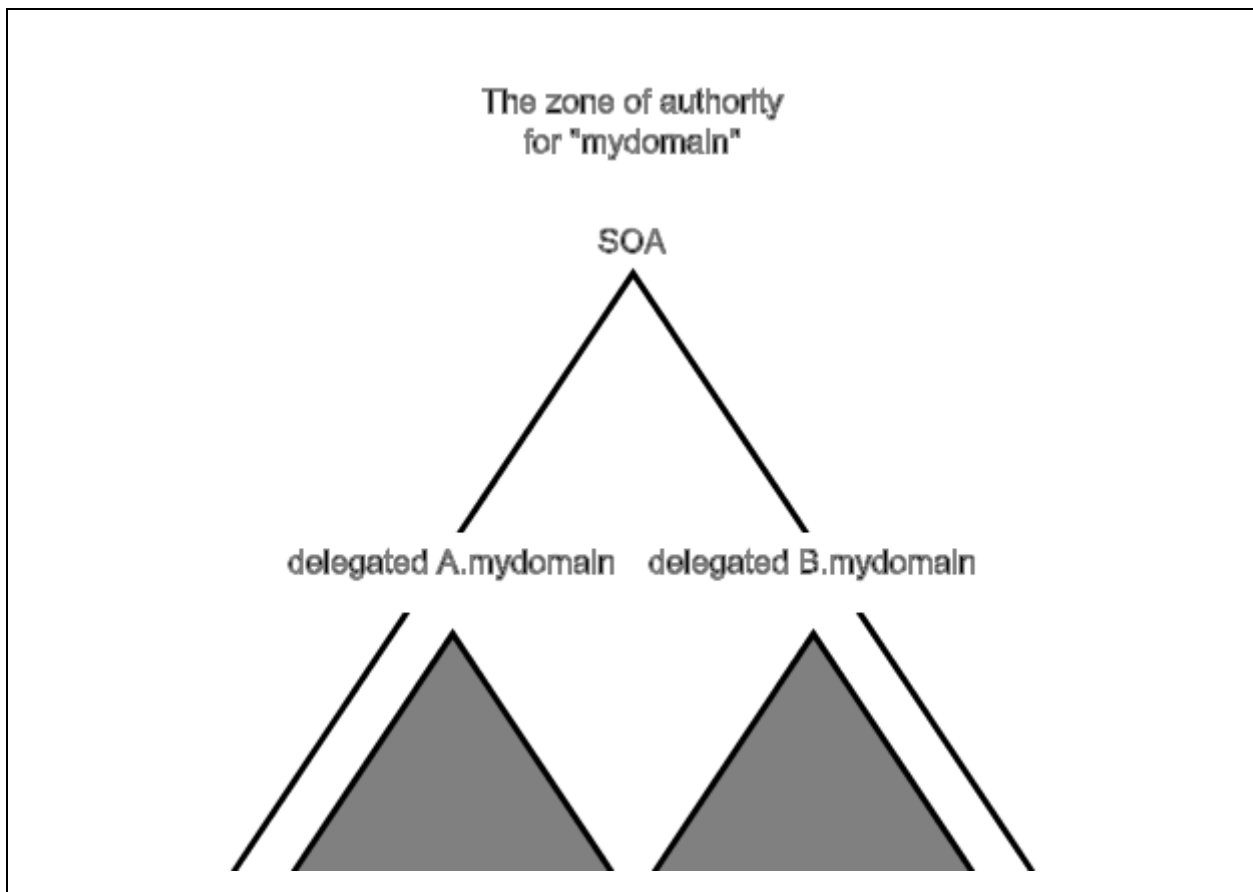


Figure 21-10: DNS delegation

A full name-service-client will build its own chain of authorities for any name that it learns about. It will start at the root name servers, get servers for the top-level name, from them get servers for the second-level name, and so on until it finds a server whose zone of authority includes precisely the name it is looking for.

To avoid keeping constantly busy tracing these chains of authority, the client builds a “cache” of names, and once it has found a name it refers to the cache in future.

How long does the delegation last?

Of course nothing lasts forever, certainly not the assignment of resources to names, and thus periodically the client should obtain a new record for each name in its cache.

How often? Well, amongst the information that is returned in a resource record for a name, is a little (not so little, 32 bits wide!) number called the *TTL*, or *time-to-live*. This is the maximum length of time that it should remain in a cache. When the client sees that this value has gone to 0 it tosses the record and obtains a new one, and of course with intelligent management it may be able in most cases to assure that the record is replaced before it expires.

In the examples shown previously, for example in Listing 6-1, line 12, a *TTL* value is given for each record, in the form *0y-0w-0d-6h-0m-0s*, which (surprised?) means this particular record is good for 6 hours.

This chain of authority is in many ways the basis for any security that is found in the Internet, and should be understood, and honored, by everyone. And of course things are so designed in general that even though some will try to defeat it, properly managed clients and servers can be reasonably secure. Observe the term properly-managed, and realize that many things do not fall under that category in any industry, certainly ours!

There is work afoot to add additional means of security to DNS records, and we may discuss some of them more fully later on. For now you understand that authoritative servers are the only sources for authoritative records, and you know how a server is designated to be authoritative.

One term related to DNS management is amusing, and instructive. It can happen (I even know organizations well in which it has happened :-), that a parent server designates another as authoritative, but that other server somehow did not get the word, and is not maintaining an SOA record for the domain it is supposedly authoritative for. This is called a "lame delegation", lame literally in the sense that it does not fully work! The administrators of the parent domain and the child domain need to get together and resolve this right away!

How does DNS work?

Protocol

The behavior required by the protocol is described in *RFC-1034*. A DNS client connects, and then sends a packet which contains one or more queries, each one asking for all resource-records of a specified type in a specified class.

The class will almost always be the *IN* or *Internet* class (we will not consider any other class in this book), and the record type will be *SOA|A|PTR|NS|CNAME|MX*. The *SRV* record type is also used by Microsoft in recent implementations, and other types may occur.

The server potentially sends back those resource records that it thinks might be suitable, collected into three categories.

Answer records are the resource records that the server has, of the types you requested.

Authority records are those that show where the authority resides for the answers that have been given.

Additional records are other information that the server implementor thought would be useful. For example if an answer record gives the name of a machine, an additional record might give the address of the same machine.

Now that you know the job that must be done, lets take a look at the specifics of what must be done and shortly see how to get Java to do it.

the packets

The packet and record structure are given in *RFC-1035*. We will consider only the records that are commonly used. Some other types are defined and now consider obsolete, or are seldom used. There are other types that have been defined later and are frequently seen. The structure of a resource record describes what it represents when stored, not necessarily the way it is actually stored, while the packet structure describes literally the byte, and ordering of bytes, that is actually seen "crossing the wire".

data types

There are several types of data involved.

A domain name segment consists of not more than 63 bytes representing characters. The overwhelming majority of the characters used are *7-bit ASCII*, but the specification does not restrict the bytes used.

A domain name consists of domain name segments, separated by "." characters, with total length of no more than 255 bytes.

A resource type is represented by a 16-bit unsigned integer, and also by the short strings we have previously been using. The types we have used so far are:

A resource record contains data about a specific type of DNS data, and has the following structure:

<i>field</i>	<i>data type</i>
name	a domain name
type	an unsigned 16-bit integer
class	an unsigned 16-bit integer
ttl	an unsigned 32-bit integer
length	an unsigned 16-bit integer
data	a byte array of the specified length.

Table 22: DNS resource record

The structure of a DNS protocol packet is as follows:

<i>section</i>	<i>purpose</i>
header	id, counts, flags
query	what you want to know
answer	the information you requested
authority	who defines the information
additional	other things you might want to know

Table 23: DNS packet

each with a specific format as follows.

<i>name</i>	<i>value</i>	<i>value set by</i>
id	unsigned 16-bit id	client
query reply	1 bit, value 0 for query, 1 for reply	client/server
opcode	4-bit , value 0 for all our packets	

Table 24: values in a DNS header

<i>name</i>	<i>value</i>	<i>value set by</i>
authoritative?	1 bit: value 1 for authoritative	server
truncated?	1 bit, value 1 if truncated	server
recursion-desired?	1 bit, value 1 if recursion is desired	client
recursion-available?	1 bit, value 1 if recursion is available	server
reserved	3 bits, value 0	
response code	See the following	server
question count	unsigned 16-bit integer	client
answer count	unsigned 16-bit integer	server
authority count	unsigned 16-bit integer	server
additional count	unsigned 16-bit integer	server

Table 24: values in a DNS header

<i>code</i>	<i>meaning</i>
0	no error, your reply is valid
1	format error, the server did not understand the query
2	server had a problem
3	the name does not exist
4	server does not support the request
5	server refuses
*	other values reserved

Table 25: result codes in a reply

name	the domain name that started it all
------	-------------------------------------

Table 26: the structure of a query record

type	type of record desired
class	value 1, representing the "Internet" class

Table 26: the structure of a query record

Some details of the protocol.

Now that you agree that DNS is surely powerful, and carries a lot of information back and forth, lets get to some engineering details. Since the proper operation of the Internet depends to a substantial degree on the proper operation of the DNS system we need to keep its impact on that operation as small as possible.

UDP is a great protocol to use, since it carries only the most basic information, that is, port numbers! There is a checksum that might tell us if something went wrong, and that's about it. DNS servers are very busy, and do not have time to carry on a conversation. Especially they are so busy managing the connection of names with resources, they don't have time left over to manage the connection of clients with queries, and with connections. UDP is great for this, since it does not require the maintenance of any sort of connection.

A size problem Packets must be able to be sent, and will be sent, over every possible kind of network. Some of those networks have very small limits on the size of the packets that can be sent, although to follow proper Internet standards at this level a user must be able to send a packet of 576 bytes at the link level. Now the link headers and IP header take up some space ("original"-Ethernet takes 14, and IP "without-options" takes 20, and some line protocols have taken much more than Ethernet. The UDP header itself takes only 8 bytes.

A solution The folks who designed DNS decided that they had to fit all of their information into a data payload of 512 bytes at the DNS level: meaning that is the largest amount of data it can give to UDP, and the largest amount it will receive.

You see, if there was more data required, there would be additional packet size overhead, and management overhead on both ends, to keep track of the ordering of the packets, deciding what goes in which packet, and so on.

So it was decided, and is still part of the consideration in using DNS to carry any new type of information, that every attempt would be made to fit things in to 512 bytes.

Truncation A safety valve was installed, as well. The principles of Internet engineering have always been (Thank God!) "keep it simple, stupid", so they give you one little bit you may have noticed above called the "truncated" bit, and will set it if they really needed to send more information.

You can then attempt to get the information again, using the TCP protocol, from a server that is willing and capable of doing this job. Part of the standard says that servers "should" be willing to help (so "should" we all in many crises that befall our neighbors - but are we?).

Since it sounds as if dealing with truncation and truncated packets might be a problem, perhaps there is a way to use some ingenuity and additional effort to "keep our packets small".

Compression In typical Internet engineering fashion, lets consider the nature of the data that DNS must handle. Its not very hard, it consists of names!

And what is the nature of the names?

- They are broken up into segments, reflecting the hierarchical structure of their authority.
- Not only that, but one might (correctly) imagine, that many of the names in a typical reply might share the same suffix, after perhaps an initial segment of two that distinguishes them. And of course many of the names might be the same, as you will see if you look back at the samples shown earlier.

So maybe we can "play some tricks" and devise a way to indicate that a name used as part of one answer, is the same name, or shares a prefix with, another name in the same reply packet.

That word "packet" is the key to thinking about this problem and solution. There is no way to "compress" information about names if there is only one name, and it won't work very well if there are only a few. But if there are a "lot" of names, it may save enough space to make it possible to send a "lot" of names.

If the preceding statement sounded circular it was!

- How big are the segments? The people that determined the specifications for DNS set them at 63 bytes or less: you might see why in a moment.
- To represent a dotted name we can just run across it from the left, using a single byte to show the length of the segment (no problem, it can be represented in a byte). We clearly don't need to send the "dot". And we can represent the end of the name by a "length byte" of 0.

There's no compression yet.

But what if in parsing across you see that the remainder of the name points to a suffix that you have already seen (or to be fair perhaps one that you will later need to encode - we look at the packet as a whole)?

- If this occurs then why not just use a "pointer" to the suffix already represented? This will need to cover at least the 512 byte length of the packet to be sent, so will need at least 9 bytes.
- Since lengths of segments are less than 64 bytes they can be represented in 6 bits.

- That leaves two more bits to use to indicate that compression is afoot, and we can let that signal our name decompressor (we don't need to signal the compressor, we ARE the compressor) to grab the next byte as well, stick it together with the 6 bits that now are not needed for a length, and make the whole 14 bits an offset to the segment already shown.
- This is something that is done to for the packet as a whole, for the purpose of sending it “over the wire”, and specifically to try to make the whole thing fit into the fixed 512-byte maximum size of the datagram.

The point of all of this is that in working with DNS it is common for names to share the same suffix, and so once a suffix has been entered it can be used over and over at the cost of only 2 bytes.

And this lets us see why this representation can only be done in a packet, and must be created just as the packet is sent, and removed once the packet is received. The pieces of the packet will go into individual resource records at the client, typically to be cached, and so the pointer structure no longer has any meaning! In other words you cannot just store the packet as it arrives, without doing the decompression.

Here's another place we need to do a little Internet engineering thinking. You could keep the packet in compressed form, and try to "pop" names out when they are needed. But a real client is likely to want to "cache" the information it receives, and it may receive information about a name from many different sources, in many different compressed packets. So just in order to recognize that we have seen a name before, we had probably better decompress it.

Of course you can now use your ingenuity to devise a compression scheme for the cache!

Analysis of an actual dump

At the risk of appearing to be crawling through a garbage dump (-:-) lets look at an actual dump and follow it. Once you have done one you can do them all, and of course can teach a computer to do it so you never have to do it again!

This is a fragment captured with DumpPacket from the DNS query shown earlier asking for A records related to *microsoft.com*

```
001 00000280                00 01 81 80 .....5.5.s~.....
002 00000290 00 01 00 06 00 06 00 06 09 6d 69 63 72 6f 73 6f .....microso
003 000002a0 66 74 03 63 6f 6d 00 00 01 00 01 c0 0c 00 01 00 ft.com.....
004 000002b0 01 00 00 09 40 00 04 cf 2e c5 66 c0 0c 00 01 00 ....@....f....
005 000002c0 01 00 00 09 40 00 04 cf 2e c5 64 c0 0c 00 01 00 ....@....d....
```

```

006 000002d0 01 00 00 09 40 00 04 cf 2e e6 dc c0 0c 00 01 00 ....@.....
007 000002e0 01 00 00 09 40 00 04 cf 2e e6 db c0 0c 00 01 00 ....@.....
008 000002f0 01 00 00 09 40 00 04 cf 2e e6 da c0 0c 00 01 00 ....@.....
009 00000300 01 00 00 09 40 00 04 cf 2e c5 71 c0 0c 00 02 00 ....@.....q....
010 00000310 01 00 02 1c bd 00 12 04 44 4e 53 33 02 55 4b 04 .....DNS3.UK.
011 00000320 4d 53 46 54 03 4e 45 54 00 c0 0c 00 02 00 01 00 MSFT.NET.....
012 00000330 02 1c bd 00 0a 04 44 4e 53 31 02 54 4b c0 93 c0 .....DNS1.TK...
013 00000340 0c 00 02 00 01 00 02 1c bd 00 0a 04 44 4e 53 31 .....DNS1
014 00000350 02 43 50 c0 93 c0 0c 00 02 00 01 00 02 1c bd 00 .CP.....
015 00000360 0a 04 44 4e 53 31 02 53 4a c0 93 c0 0c 00 02 00 ..DNS1.SJ.....
016 00000370 01 00 02 1c bd 00 0a 04 44 4e 53 31 02 44 43 c0 .....DNS1.DC.
017 00000380 93 c0 0c 00 02 00 01 00 02 1c bd 00 0a 04 44 4e .....DN
018 00000390 53 33 02 4a 50 c0 93 c0 8b 00 01 00 01 00 02 1c S3.JP.....
019 000003a0 5b 00 04 d5 c7 90 97 c0 a9 00 01 00 01 00 02 1c [.....
020 000003b0 5b 00 04 cf 2e e8 25 c0 bf 00 01 00 01 00 00 06 [....%.
021 000003c0 69 00 04 cf 2e 8a 14 c0 d5 00 01 00 01 00 00 06 i.....
022 000003d0 69 00 04 cf 2e 61 0b c0 eb 00 01 00 01 00 00 06 i...a.....
023 000003e0 69 00 04 cf 44 80 97 c1 01 00 01 00 01 00 00 06 i...D.....
024 000003f0 69 00 04 cf 2e 48 7b 88 i...H{.O.<c..*

```

Listing 21-1: Hex dump of A-query capture

And this is a portion of the interpreted output, showing in particular the names that were found:

```

000 // File: a.microsoft.com.cap.out
001 Provider: dns Server <363>
002 id1: QUERY R:-A:-T:RDRA: OK: queries 1 ans 6 auth 6 addl 6
003 Query 1
004 microsoft.com. type 1 class 1
005
006 Answer 2
007 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
008 207.46.230.220
009 Answer 3
010 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
011 207.46.230.219
012 Answer 4
013 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
014 207.46.230.218
015 Answer 5
016 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
017 207.46.197.113
018 Answer 6
019 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4
020 207.46.197.102
021 Answer 7
022 Name:microsoft.com. Type:A Class:IN TTL:0y-0w-0d-0h-32m-11s Length:4

```

```

023 207.46.197.100
024
025 Authority 1
026 Name:microsoft.com. Type:NS Class:IN TTL:0y-0w-1d-2h-14m-32s Length:18
027 DNS1.CP.MSFT.NET.
028 Authority 2
029 Name:microsoft.com. Type:NS Class:IN TTL:0y-0w-1d-2h-14m-32s Length:10
030 DNS1.SJ.MSFT.NET.
031 Authority 3
032 Name:microsoft.com. Type:NS Class:IN TTL:0y-0w-1d-2h-14m-32s Length:10
033 DNS1.DC.MSFT.NET.
034 Authority 4
035 Name:microsoft.com. Type:NS Class:IN TTL:0y-0w-1d-2h-14m-32s Length:10
036 DNS3.JP.MSFT.NET.
037 Authority 5
038 Name:microsoft.com. Type:NS Class:IN TTL:0y-0w-1d-2h-14m-32s Length:10
039 DNS3.UK.MSFT.NET.
040 Authority 6
041 Name:microsoft.com. Type:NS Class:IN TTL:0y-0w-1d-2h-14m-32s Length:10
042 DNS1.TK.MSFT.NET.
043
044 Additional 1
045 Name:DNS1.CP.MSFT.NET. Type:A Class:IN TTL:0y-0w-0d-0h-15m-27s Length:4
046 207.46.138.20
047 Additional 2
048 Name:DNS1.SJ.MSFT.NET. Type:A Class:IN TTL:0y-0w-0d-0h-15m-27s Length:4
049 207.46.97.11
050 Additional 3
051 Name:DNS1.DC.MSFT.NET. Type:A Class:IN TTL:0y-0w-0d-0h-15m-27s Length:4
052 207.68.128.151
053 Additional 4
054 Name:DNS3.JP.MSFT.NET. Type:A Class:IN TTL:0y-0w-0d-0h-15m-27s Length:4
055 207.46.72.123
056 Additional 5
057 Name:DNS3.UK.MSFT.NET. Type:A Class:IN TTL:0y-0w-1d-2h-12m-33s Length:4
058 213.199.144.151
059 Additional 6
060 Name:DNS1.TK.MSFT.NET. Type:A Class:IN TTL:0y-0w-1d-2h-12m-33s Length:4
061 207.46.232.37

```

Listing 21-2: dump of an A-query capture

And here is a table giving the correspondence between the hex dump and the capture output. There were 25 names found in the output, and the number is given in the first column of the table, and the actual name in the second column. The third column shows the offset in the hex dump where this name was represented, and the fourth column shows the algorithm that was used to parse the name from the file. The "plain" algorithm means it was found as length-prefixed strings of ascii-7 characters. The "@c00c" notation means it was found using the given number as an off-

set from the beginning of the capture: thus since the packet captured begins at offset 028c in this hex dump, the offset tells us to look at 028c+000c (the initial c hex character is just the bits 11 that begin a pair of bytes that represents an offset). At that location you will find the name *microsoft.com*, which turns out, not surprisingly, to be the most common name in the packet.

The notation 2 plain +@c093 means, as you might suspect, that after two segments that are encoded plain, you encountered the pair c093 of bytes, which tells us to look at 028c+0093, which turns out to be offset 31f, which begins the plain-encoded suffix *msft.net*, the most common suffix after com in the capture.

the number	the name	name begins at	how to parse it
01	<i>microsoft.com</i>	298	plain
02	<i>microsoft.com</i>	2ab	@c00c
03	<i>microsoft.com</i>	2bb	@c00c
04	<i>microsoft.com</i>	2cb	@c00c
05	<i>microsoft.com</i>	2db	@c00c
06	<i>microsoft.com</i>	2eb	@c00c
07	<i>microsoft.com</i>	2fb	@c00c
08	<i>microsoft.com</i>	30b	@c00c
09	<i>dns1.cp.msft.net</i>	317	plain
10	<i>microsoft.com</i>	329	@c00c
11	<i>dns1.sj.msft.net</i>	335	2 plain+@c093
12	<i>microsoft.com</i>	33f	@c00c
13	<i>dns1.dc.msft.net</i>	34b	2 plain+@c093
14	<i>microsoft.com</i>	355	@c00c
15	<i>dns3.jp.msft.net</i>	361	2 plain+@c093
16	<i>microsoft.com</i>	36b	@c00c
17	<i>dns3.uk.msft.net</i>	377	2 plain+@c093
18	<i>microsoft.com</i>	381	@c00c
19	<i>dns1.tk.msft.net</i>	38d	2 plain+@c093
20	<i>dns1.cp.msft.net</i>	397	@c08b

Table 22: names found in a DNS reply

the number	the name	name begins at	how to parse it
21	<i>dns1.sj.msft.net</i>	3a7	@c0a9
22	<i>dns1.dc.msft.net</i>	3b7	@c0bf
23	<i>dns1.jp.msft.net</i>	3c7	@c0d5
24	<i>dns3.uk.msft.net</i>	3d7	@c0eb
25	<i>dns1.tk.msft.net</i>	3e7	@c101

Table 22: names found in a DNS reply

A little calculation might help you appreciate the savings that compression can bring. There were 25 names seen, and if each had been represented in its standard dotted form, the names would have consumed 361 bytes.

With compression there were 2 names stored completely in plain fashion, and 5 names needed an initial plain prefix of two segments, and could then continue with a pointer. The total number of bytes needed for this was 119 bytes.

So the compressed names required less than 1/3 of the space that plain names required. A savings of 242 bytes may not seem like much (in these days of Gigabit Ethernet), but remember, you have very little idea or control over where your name service request may need to go, and keeping it well within the packet size limits of any network in the world means you never have to worry.

How Java might do it

Java has several ways of handling DNS information.

The basic **java.net.InetAddress** class has minimal functionality built in, for dealing with A records and some PTR records.

With jdk1.4 a reasonably full DNS service is provided using the JNDI interface.

There are several packages available in the open source community that handle various aspects of DNS service in Java.

Recalling however that our goal in this chapter (and in this book) is to learn as much as we can about DNS as a protocol, and recalling the author's belief that a good way to learn is to try to coordinate "what you get" with "what you see", the remainder of this section will show code for parts of the name service provider that produced the packets we have been observing.

A reason for doing this, other than helping to build our understanding regarding exactly what DNS does, is that seeing how we might handle DNS packets may give some insights into problems you need to confront in which there are not solutions ready on the shelf.

Dealing with datatypes

Almost all networked or distributed programming involves moving data back and forth, formatted and parsed in accordance with grammars that "only a (grand)parent could love". One very important concept to understand, and become very comfortable with, is that you must think about where information is coming from and going to. And this analysis applies even more to the movement of data on your own machine. And the ways in which data is represented "outside" and "inside" will vary.

If you believe in the promise of Java as a platform, you realize that as far as Java itself is concerned data and its representation looks alike on every machine, and you don't need to worry about it when it is "in flight". However, the real value of Java is that it can be used as a "glue" to data represented in any sort of way on any sort of machine, and to do that it is going to have to help you do arcane data conversions.

So this problem has two aspects: it will give an example of such conversions, and it will provide something useful. Yes, you can use some of this code if you need DNS yourself in some way that your current software does not provide, and yes, you may need to do exactly that with DNS or a successor as the importance of naming of all sorts continues to grow.

Pulling packets apart

First lets see how to "put a packet together" and how to "take a packet apart". Since we are dealing with networking almost surely there is a socket around with its **InputStream** and **OutputStream**. The package **java.io** has available almost exactly what we need now in its **DataInputStream** and **DataOutputStream**, which are simple wrappers that know how to read and write various data types.

So to write a packet, we merely use the appropriate write methods. There is a slight complication here, in that some of the writes that we need are of 32-bit unsigned quantities, something that Java does not officially support at present.

If you are reading the chapters in order (do people?) then you have already seen the solution. In Chapter-Java we extended those classes to handle larger unsigned quantities as well as some other useful types, and the **PacketInputStream** is exactly what we will need (which may explain its name).

The one trick the current code will not show how to do, is how to write a compressed packet. The "art of name compression" is required only on the server side, and we are focussing on clients. If

you are looking for a fun problem (for some types of mind) consider whether you would rather "code the long names and use pieces for the short names", "code the short names and build the long ones from them", or "code things as you encounter them and use their suffixes when you can". From looking at real binary output from real nameservers, I can only attest that they haven't made up their minds on this problem at all. Maybe they're too busy.

Using the `PacketInputStream`

The client needs to do decompression, and thinking through how this might be done should be part of thinking through the general processing of a received packet.

Begin with a `PacketInputStream` which is connected to the beginning of the packet. The packet structure defined in *RFC-1035* lends itself very nicely to just "chewing off bytes" as they are needed, and this is exactly what a `DataInputStream` is good at doing, a property which its extension inherits.

When we encounter a name, we can chew off its bytes very simply as well.

We read a count byte, and if it is less than 64 we just grab the next `byte[count]` array using the `readFully(..)` method of the stream. Be sure and add a "." if building a String representation of the name.

This processing is shown in lines XXX-YYY below.

If it is greater than 191, that is, its first two bits are 1, then chop off those leading two bits, move the stream temporarily to the offset from the original beginning of the stream, and read the name as usual.

If the number is between 64 and 191 inclusive, someone is pulling your leg, what you received was not a legitimate RFC1035 DNS packet. You figure out what you want to do, but one thing you should not do is continue to process the packet!

Something was slipped in just now. There doesn't seem to be any automatic "move the stream" facility in `java.io.DataInputStream`. There is marking, and reset, but the way they behave will depend upon what is going on in the streams underneath.

`PacketInputStream` knows about this. It took the whole packet in initially (which is why it is called a "source") as an array of bytes in its constructor. It then combined a `ByteArrayInputStream` and a `DataInputStream`, and its own extensions, and built in a way to move to another place in the stream and return.

You should read the fuller discussion in Chapter-Java, if you want to follow how this was done.

Returning to uncompressing the compressed name, after following the pointer, just recursively read a name again. When done return to where you were and continue pulling the packet apart. All of this is shown in lines XXX-YYY.

It is helpful to realize that once you have gone to a compressed segment you are done with the name, as far as processing at the current location is concerned.

So now we know how to read names that have been compressed!

Since you asked, lets just think for a moment about the problem of the 32-bit unsigned integers. The simplest thing to do in Java is just deal with them internally as longs, but we cannot just cast to a long, since that would preserve their sign. We just follow the lead of the **DataInputStream** class, reading 8 consecutive bytes from the stream, and putting each byte in turn into place as part of the bits of a long.

So now we can take a packet apart! This will be "standard operating procedure" for all other protocol handling.]

Here is the code for **RFC1035.java**, which does most of the parsing:

```
000 // Program: RFC1035.java
001 package net.dougharris.dns;
002
003 import net.dougharris.utility.PacketInputStream;
004 import net.dougharris.utility.DumpHex;
005 import java.io.DataOutputStream;
006 import java.io.DataInputStream;
007 import java.io.ByteArrayOutputStream;
008 import java.io.ByteArrayInputStream;
009 import java.io.IOException;
010 import java.util.ArrayList;
011 import java.util.StringTokenizer;
012 import java.util.Iterator;
013 import java.util.Map;
014 import java.util.HashMap;
015
016 public class RFC1035{
017     static protected HashMap typeNames = new HashMap();
018     static{
019         typeNames.put(new Integer(1), "A");
020         typeNames.put(new Integer(2), "NS");
021         typeNames.put(new Integer(5), "CNAME");
022         typeNames.put(new Integer(6), "SOA");
023         typeNames.put(new Integer(11), "WKS");
024         typeNames.put(new Integer(12), "PTR");
025         typeNames.put(new Integer(13), "HINFO");
026         typeNames.put(new Integer(15), "MX");
027         typeNames.put(new Integer(16), "TXT");
```

```

028 }
029 private PacketInputStream inputStream = null;
030 protected QueryRecord[] queryArray = new QueryRecord[0];
031 protected ResourceRecord[] ansArray = new ResourceRecord[0];
032 protected ResourceRecord[] autArray = new ResourceRecord[0];
033 protected ResourceRecord[] addArray = new ResourceRecord[0];
034 StringBuffer reportBuffer=new StringBuffer();
035 short id;
036 boolean qr;
037 int opcode;
038 boolean aa;
039 boolean tc;
040 boolean rd;
041 boolean ra;
042 int rcode;
043 short flags;
044 short queryCount;
045 short answerCount;
046 short authorityCount;
047 short additionalCount;
048
049 public RFC1035(byte[] b){
050     this(b, 0, b.length);
051 }
052
053 public RFC1035(byte[] b, int offset, int length){
054     try{
055         this.inputStream = new PacketInputStream(b, offset, length);
056     }catch(Exception x){
057         System.exit(2);
058     }
059 }
060
061 public ResourceRecord[] getAnswers(){
062     return ansArray;
063 }
064
065 public ResourceRecord[] getAuthorities(){
066     return autArray;
067 }
068
069 public ResourceRecord[] getAdditional(){
070     return addArray;
071 }
072
073 public static byte[] createQueryPacket(String name, String type, boolean recurse){
074     int intType;
075     try{
076         intType = Integer.parseInt(type);
077     } catch(NumberFormatException x){
078         intType=RFC1035.getTypeNumber(type);
079         if (intType == -1){
080             System.err.println("No type "+type);
081             System.exit(2);

```

```

082     }
083     }
084     return createQueryPacket(intType, name, recurse);
085 }
086
087 public static byte[] createQueryPacket
088 (int type, String name, boolean recurse){
089     ByteArrayOutputStream ob = new ByteArrayOutputStream();
090     DataOutputStream o = new DataOutputStream(ob);
091     try{
092         String n;
093         o.writeShort(1); // Id
094         if (recurse){
095             o.writeShort(256); // Recursion Desired
096         } else {
097             o.writeShort(0); // No Recursion Desired
098         }
099         o.writeShort(1); // Questions
100         o.writeShort(0); // Answers
101         o.writeShort(0); // Authority
102         o.writeShort(0); // Additional
103         StringTokenizer k = new StringTokenizer(name, ".");
104         while(k.hasMoreTokens()){
105             n=k.nextToken();
106             o.writeByte(n.length());
107             for(int j=0;j<n.length();j++){
108                 o.writeByte(n.charAt(j));
109             }
110         }
111         o.writeByte(0); // Root name
112         o.writeShort(type);
113         o.writeShort(1); // class IN
114         for (int j=o.size();j<512;j++){
115             o.writeByte(0);
116         }
117         o.close();
118     }catch(IOException x){
119         System.exit(69);
120     }
121     return ob.toByteArray();
122 }
123
124 public RFC1035 parse(){
125     ArrayList recordList = new ArrayList();
126     String name;
127     int type;
128     int rClass;
129     try{
130         id=inputStream.readShort();
131         flags=inputStream.readShort();
132         queryCount=inputStream.readShort();
133         answerCount=inputStream.readShort();
134         authorityCount=inputStream.readShort();
135         additionalCount=inputStream.readShort();

```

```

136   for (int j=1;j<=queryCount;j++){
137       recordList.add(readQueryRecord(inputStream));
138   }
139   recordList.toArray(queryArray=new QueryRecord[recordList.size()]);
140   recordList.clear();
141
142   reportBuffer.append(answerCount+" answer records--\n");
143   for (int j=1;j<=answerCount;j++){
144       recordList.add(readResourceRecord(inputStream));
145   }
146   recordList.toArray(ansArray=new ResourceRecord[recordList.size()]);
147   recordList.clear();
148
149   reportBuffer.append(authorityCount+" authority records--\n");
150   for (int j=1;j<=authorityCount;j++){
151       recordList.add(readResourceRecord(inputStream));
152   }
153   recordList.toArray(autArray=new ResourceRecord[recordList.size()]);
154   recordList.clear();
155
156   reportBuffer.append(additionalCount+" additional records--\n");
157   for (int j=1;j<=additionalCount;j++){
158       recordList.add(readResourceRecord(inputStream));
159   }
160   recordList.toArray(addArray=new ResourceRecord[recordList.size()]);
161   recordList.clear();
162 }catch(Exception x){
163     System.out.println(x.getClass().getName()+" says "+x.getMessage());
164 }finally{
165     return this;
166 }
167 }
168
169 private final static String[] classNames = new String[]{"", "IN"};
170
171 public static String getRclassString(int cls){
172     return classNames[cls];
173 }
174
175 public static String getTypeString(int type){
176     String t = (String)typeNameNames.get(new Integer(type));
177     if (t==null){
178         System.exit(69);
179     }
180     return t;
181 }
182
183 public static ResourceRecord getTypeInstance(int type){
184     String className="net.dougharris.dns."
185     +getTypeString(type)+"ResourceRecord";
186     ResourceRecord instance=null;
187     try{
188         instance = (ResourceRecord)Class.forName(className).newInstance();
189     } catch(ClassNotFoundException x){

```

```

190 System.err.println(x.getClass().getName()+":"+x.getMessage());
191 } catch(InstantiationException x){
192 System.err.println(x.getClass().getName()+":"+x.getMessage());
193 } catch(IllegalAccessException x){
194 System.err.println(x.getClass().getName()+":"+x.getMessage());
195 }
196 return instance;
197 }
198
199 public static int getTypeNumber(String typeName){
200 int reply = -1;
201 typeName=typeName.toUpperCase();
202 Map.Entry entry;
203 String entryName;
204 Integer entryNumber;
205 Iterator j = typeNames.entrySet().iterator();
206 while (j.hasNext()){
207 entry=(Map.Entry)j.next();
208 entryName = (String)(entry.getValue());
209 entryNumber= (Integer)(entry.getKey());
210 if (entryName.equals(typeName)){
211 reply = entryNumber.intValue();
212 }
213 }
214 return reply;
215 }
216
217 QueryRecord readQueryRecord(PacketInputStream i)
218 throws IOException{
219 QueryRecord result=new QueryRecord();
220 String name = decompressRFC1035Name(i);
221 int type = i.readUnsignedShort();
222 int rClass = i.readUnsignedShort();
223 result.setName(name);
224 result.setType(type);
225 result.setRClass(rClass);
226 return result;
227 }
228
229 ResourceRecord readResourceRecord(PacketInputStream i)
230 throws IOException{
231 ResourceRecord result = null;
232 try{
233 String name = decompressRFC1035Name(i);
234 int type = i.readUnsignedShort();
235 int rClass = i.readUnsignedShort();
236 int ttl = i.readInt();
237 int length = i.readUnsignedShort();
238 result = getTypeInstance(type);
239 result.setName(name);
240 result.setType(type);
241 result.setRClass(rClass);
242 result.setTTL(ttl);
243 result.setLength(length);

```

```

244  /*
245   Now switch on the type number and fill the packets
246  */
247  //byte[] b = readData(length);
248  //result.setData(b);
249  switch(type){
250   case 2: /* NS */
251   ((NSResourceRecord)result).setNSName(decompressRFC1035Name(i));
252   break;
253
254   case 5: /* CNAME */
255   ((CNAMEResourceRecord)result).setCNAME(decompressRFC1035Name(i));
256   break;
257
258   case 6: /* SOA */
259   SOAResourceRecord rr = (SOAResourceRecord)result;
260   rr.setMname(decompressRFC1035Name(i));
261   rr.setRname(decompressRFC1035Name(i));
262   rr.setSerial(i.readUnsignedInt());
263   rr.setRefresh(i.readUnsignedInt());
264   rr.setRetry(i.readUnsignedInt());
265   rr.setExpire(i.readUnsignedInt());
266   rr.setMinimum(i.readUnsignedInt());
267   break;
268
269   case 12: /* PTR */
270   ((PTRResourceRecord)result).setPTRName(decompressRFC1035Name(i));
271   break;
272
273   case 15: /* MX */
274   ((MXResourceRecord)result).setPreference(i.readUnsignedShort());
275   ((MXResourceRecord)result).setExchanger(decompressRFC1035Name(i));
276   break;
277
278   case 1: /* A */
279   default:
280   byte[] b = readData(i,length);
281   result.setData(b);
282   break;
283   }
284   } catch(Exception x){
285   System.out.println(x.getClass().getName()+" says "+x.getMessage());
286   } finally {
287   return result;
288   }
289   }
290
291   public byte[] readData(PacketInputStream i, int n) throws IOException{
292   byte[] data=new byte[n];
293   i.readFully(data);
294   return data;
295   }
296
297   String decompressRFC1035Name(PacketInputStream i)

```

```

298 throws IOException{
299 String name=new String();
300 try{
301 int n=0;
302 while(true){
303     try{
304         n=i.readUnsignedByte();
305     } catch(Exception x){
306         System.out.println(x.getClass().getName()+" says "+x.getMessage());
307         break;
308     }
309     if (n==0){
310         break;
311     }
312     if (n<64){
313         name+=plainSegment(i,n);
314         continue;
315     } else if (n>191){
316         n=((n&63)<<8)+i.readUnsignedByte();
317         name+=compressedSegment(i,n);
318         break;
319     } else {
320         throw new IOException("bad encoding of name with n= "+n);
321     }
322 }
323 }catch(Exception x){
324     System.out.println(x.getClass().getName()+" says "+x.getMessage());
325     System.exit(69);
326 } finally {
327     return name;
328 }
329 }
330
331 String plainSegment(PacketInputStream i,int n)
332 throws IOException{
333 byte[] b = new byte[n];
334 i.readFully(b);
335 return new String(b,"ASCII7")+". ";
336 }
337
338 String compressedSegment(PacketInputStream i,int n)
339 throws IOException{
340 String s="";
341 try{
342 long where=i.getPosition();
343 i.setPosition(n);
344 s=decompressRFC1035Name(i);
345 i.setPosition(where);
346 }catch(Exception x){
347     System.out.println(x.getClass().getName()+" at "+n);
348     System.exit(69);
349 } finally {
350 return s;
351 }

```

```

352 }
353
354 public String showFlags(short flags){
355     StringBuffer b = new StringBuffer();
356     qr = (0==(flags&(short)0x8000)>>15);
357     opcode = (flags&(short)0x7800)>>11;
358     aa = (0!=(flags&(short)0x0400)>>10));
359     tc = (0!=(flags&(short)0x0200)>>9));
360     rd = (0!=(flags&(short)0x0100)>>8));
361     ra = (0!=(flags&(short)0x0080)>>7));
362     rcode = (flags&(short)0x000f);
363     return b.toString();
364 }
365 public String toString(){
366     StringBuffer b = new StringBuffer();
367     int n;
368     b.append("id"+id);
369     b.append(": ");
370     if (opcode==0){
371         b.append("QUERY");
372     }
373     b.append(" "+showFlags(flags));
374     b.append(qr?"Q":"R");
375     b.append(":");
376     b.append(aa?"AA":"-A");
377     b.append(":");
378     b.append(tc?"TC":"-T");
379     b.append(":");
380     b.append(rd?"RD":"-D");
381     b.append(ra?"RA":"-A");
382     b.append(": ");
383     switch(rcode){
384     case(0):
385         b.append("OK");
386         break;
387     case(1):
388         b.append("FormatErr");
389         break;
390     case(2):
391         b.append("SrvFail");
392         break;
393     case(3):
394         if(aa){
395             b.append("NxName");
396         }
397         break;
398     case(4):
399         b.append("NotImpl");
400         break;
401     case(5):
402         b.append("WillNot");
403         break;
404     }
405     b.append(":");

```

```

406 b.append(" queries "+queryCount);
407 b.append(" ");
408 b.append(" ans "+answerCount);
409 b.append(" ");
410 b.append(" auth "+authorityCount);
411 b.append(" ");
412 b.append(" addl "+additionalCount);
413 b.append("\n");
414
415 n=0;
416 for (int j=0;j<queryCount;j++){
417     b.append(" Query "+(++n));
418     b.append("\n");
419     b.append(queryArray[j].toString());
420     b.append("\n");
421 }
422 if (queryCount>0){
423     b.append("\n");
424 }
425 for (int j=0;j<answerCount;j++){
426     b.append(" Answer "+(++n));
427     b.append("\n");
428     b.append(ansArray[j].toString());
429 }
430 if (answerCount>0){
431     b.append("\n");
432 }
433 n=0;
434 for (int j=0;j<authorityCount;j++){
435     b.append(" Authority "+(++n));
436     b.append("\n");
437     b.append(autArray[j].toString());
438 }
439 if (authorityCount>0){
440     b.append("\n");
441 }
442 n=0;
443 for (int j=0;j<additionalCount;j++){
444     b.append(" Additional "+(++n));
445     b.append("\n");
446     b.append(addArray[j].toString());
447 }
448 if (additionalCount >0){
449     b.append("\n");
450 }
451
452 return b.toString();
453 }
454 }

```

Listing 21-1: RFC1035.java, the main parsing program

```

000 // Program: MXResourceRecord.java
001 package net.dougharris.dns;
002
003 public class MXResourceRecord extends ResourceRecord{
004     private int preference;
005     private String exchanger;
006
007     public void setPreference(int preference){
008         this.preference=preference;
009     }
010
011     public int getPreference(){
012         return preference;
013     }
014
015     public void setExchanger(String name){
016         this.exchanger=name;
017     }
018
019     public String getExchanger(){
020         return exchanger;
021     }
022
023     public String dataToString(){
024         StringBuffer b = new StringBuffer();
025         b.append("Preference: ");
026         b.append(""+getPreference());
027         b.append(" Exchanger: ");
028         b.append(getExchanger());
029         b.append("\n");
030         return b.toString();
031     }
032 }

```

Listing 21-2: MXRecord.java, a sample record parser

Going further into DNS

There are a number of standards related to DNS, and that number will grow, and the existing standards will mature, as the number of networks and hosts increases, and mainly as the number grows of ways in which a simple name to resource mapping service can help.

There are frequent predictions that some other protocol and related service will replace DNS. Put these in the same place where you keep predictions that vehicles will soon replace feet for transportation. May be!

The following table merely lists some of the RFCs related to DNS, and their current status. Since many of them deal with record types, I have listed that information also when it applies.

Table 22:

<i>RFC</i>	<i>Status</i>	<i>Purpose</i>	<i>record type</i>
974		mail exchanger	MX
1034		concepts and facilities	
1035		implementation and specification	
1535		security problems	
1537		common configuration errors	
1876		location information	LOC
1886		IPv6	AAAA
1995		incremental zone transfer	IFXR
2136			UPDATE
2181		clarifications	
2219		aliases for services	
2308		negative caching	
2345		domain names and company names	
2535		security extensions	
2541		operational security	
2606		reserved domain names	
2782			SRV
2870		root server operation	
2901		how to get an address	
2929	BCP 42	IANA considerations	
2931		record signatures	SIG
3007	STD-track		

Lightweight Directory Access Protocol

Sure enough if you are not in on the facts already there is (was?) a directory access protocol which is the heavyweight ancestor of this one. You don't want to know about it, or if you do you will not learn any more from this book.

This protocol runs on top of TCP on port 389. It is a very traditional client-server protocol as far as how you make connections, and once you are connected you can make a series of queries and receive a series of replies.