C H A P T E R 1 5

Some example TCP/IP clients

In this chapter, we examine the implementation of some simple TCP clients that interact with existing Internet protocols. Specifically, we will look at a finger client (RFC 1288) and a DNS client (RFC 1035), the equivalent of the nslookup command. These clients can serve as the basis for implementations of other Internet protocols that possess similar structures.

15.1 Internet protocols

An important prerequisite for the implementation of an Internet protocol is a correct and current specification. In the case of the protocols that we shall look at, the specifications exist in the form of Internet requests for comments (RFCs) and standards (STDs): public documents that describe information of relevance to the Internet community. Although in many cases there are sample client implementations available, it is important to use the actual specification as a basis for an implementation. The Internet RFCs and Standards are available for public download from the location ftp://ds.internic.net/. The specification documents for various common Internet protocols are listed in table 15.1.

Table 15.1 Internet protocol specification documents (from STD 1)

Protocol	Name	RFC	STD
—	Internet Official Protocol Standards	1920	1
—	Assigned Numbers	1700	2
—	Host Requirements - Communications	1122	3
—	Host Requirements - Applications	1123	3
IP	Internet Protocol	791	5
—	IP Subnet Extension	950	5
—	IP Broadcast Datagrams	919	5
—	IP Broadcast Datagrams with Subnets	922	5
ICMP	Internet Control Message Protocol	792	5
IGMP	Internet Group Multicast Protocol	1112	5
UDP	User Datagram Protocol	768	6
TCP	Transmission Control Protocol	793	7
TELNET	Telnet Protocol	854, 855	8
FTP	File Transfer Protocol	959	9
SMTP	Simple Mail Transfer Protocol	821	10
SMTP-SIZE	SMTP Service Ext for Message Size	1870	10
SMTP-EXT	SMTP Service Extensions	1869	10
MAIL	Format of Electronic Mail Messages	822	11
CONTENT	Content Type Header Field	1049	11
NTPV2	Network Time Protocol (Version 2)	1119	12

Table 15.1 Internet protocol specification documents (from STD 1)

Protocol	Name	RFC	STD
DOMAIN	Domain Name System	1034, 1035	13
DNS-MX	Mail Routing and the Domain System	974	14
SNMP	Simple Network Management Protocol	1157	15
SMI	Structure of Management Information	1155	16
Concise-MIB	Concise MIB Definitions	1212	16
MIB-II	Management Information Base-II	1213	17
NETBIOS	NetBIOS Service Protocols	1001, 1002	19
ECHO	Echo Protocol	862	20
DISCARD	Discard Protocol	863	21
CHARGEN	Character Generator Protocol	864	22
QUOTE	Quote of the Day Protocol	865	23
USERS	Active Users Protocol	866	24
DAYTIME	Daytime Protocol	867	25
TIME	Time Server Protocol	868	26
TFTP	Trivial File Transfer Protocol	1350	33
TP-TCP	ISO Transport Service on top of the TCP	1006	35
ETHER-MIB	Ethernet MIB	1643	50
PPP	Point-to-Point Protocol (PPP)	1661	51
PPP-HDLC	PPP in HDLC Framing	1662	51
IP-SMDS	IP Datagrams over the SMDS Service	1209	52
FINGER	Finger User Information Protocol	1288	-

15.2 A finger client

A finger client is a comparatively simple tool that lists the users logged into a remote machine and displays detailed information about a specific user on the machine (figure 15.1).

To use the client, you specify a host to which to connect and an optional username. The typical syntax is `finger @host` to list the users on the machine `host`, or `finger username@host` to find out information about the specified user, `username`.

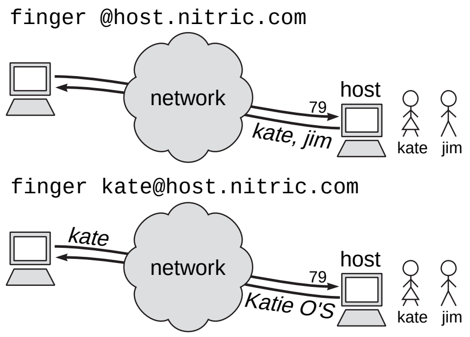

Figure 15.1 The finger protocol

There are two further details relevant to the protocol: Finger supports an optional verbose flag (`-l` in this implementation), which can be used to obtain more detailed information. It also supports a finger forwarding service whereby a finger request can be forwarded from one machine to another. The command `finger username@hostA@hostB` connects to the machine `hostB` which in turn forwards the finger request to machine `hostA` (figure 15.2).

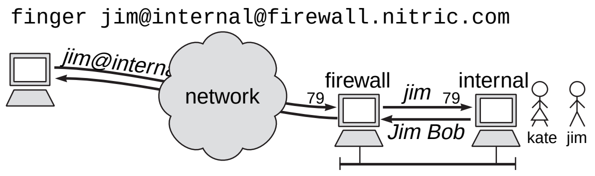

Figure 15.2 Finger forwarding

15.2.1 Protocol specification

The finger protocol is specified in RFC 1288. The specification states that the client machine should make a TCP connection to port 79 of the remote machine and transmit a finger query in 8-bit ASCII; the host will respond with an ASCII result. For this purpose, we'll use the ISO Latin 1 character set.

The client's query (table 15.2) is basically taken from RFC 1288.

Table 15.2 Client's query

{Q1}	::=[{U} | {W} | {W}{S}{U}] {C}
{Q2}	::=[{W}{S}] [{U}] {H}{C}
{U}	::=username
{H}	::=@*hostname* | @*hostname* {H}
{W}	::=/W
{S}	::=<SP> | <SP> {S}
{C}	::=<CRLF>

This type of specification is typical of the RFCs and should be interpreted as follows: {Q1} is the symbol Q1, technically called a nonterminal: every nonterminal is assigned a definition that states what values it can hold. In the case of the finger protocol, there are two types of queries: Q1 and Q2. The format of a Q1 query is defined in the first line; the format of a Q2 query is defined in the second line.

Each definition has the form {LHS} ::= RHS, which states that nonterminal on the left hand side can hold the value indicated on the right hand side.

The right hand side of a definition consists of a sequence of nonterminals and terminals that define the values that the left-hand-side symbol may hold. A *terminal* is a final value, such as a username, that will appear in an actual instance of the symbol.

A vertical bar (|) represents OR, meaning that a symbol can hold either the value on the left side of the bar or the right side. Square brackets ([]) enclose optional parts of a symbol's definition. Angle brackets (<>) enclose terminal values such as space or carriage return.

In the finger specification above, the symbols *U*, *W*, and *C* represent exactly the value on the right side of their definitions. The symbol *S* can represent either a space or a space followed by another symbol *S*. This other *S* can represent again, by definition, either a space or a space followed by another symbol *S*. In other words, the symbol *S* represents a sequence of one or more spaces. Similarly, the symbol *H* represents a sequence of one or more @*hostname* values, such as `@hostA@hostB`.

Symbol Q1 represents all or part of a `/W` header followed by whitespace, a username, and a newline sequence. Symbol Q2 represents all or part of a `/W` header followed by whitespace, a username, a sequence of @*hostname* values, and a newline sequence. Expanding the various possibilities, table 15.3 lists the possible rewritings of Q1 and Q2, or the valid types of query that a finger client may make.

Table 15.3

Definition	Symbols	Value	e.g.
Q1	{C}	<CRLF>	\r\n
Q1	{U}{C}	*username* <CRLF>	merlin\r\n
Q1	{W}{C}	`/W` <CRLF>	/W\r\n
Q1	{W}{S}{U}{C}	`/W` <SP>+ *username* <CRLF>	/W jim\r\n
Q2	{H}{C}	@*hostname*+ <CRLF>	@x.com@y.com@z.com\r\n
Q2	{U}{H}{C}	*username* @*hostname*+ <CRLF>	ego@nitric.com\r\n
Q2	{W}{H}{C}	`/W` <SP>+ @*hostname*+ <CRLF>	/W @int@gw.x.com\r\n
Q2	{W}{S}{U}{H}{C}	`/W` <SP>+ *username* @*host*+ <CRLF>	/W ego@nitric.com\r\n

Spacing here is just for legibility; the <SP> terminal represents actual space characters. A plus sign (+) indicates that the adjacent value may occur one or more times.

Note that these queries are not what the user types on the command line. They are the queries that the finger client sends to the remote machine. Queries of type Q1 are used to obtain information about users on the target machine; Q2 queries are forwarded from the target machine to another machine. Finger forwarding can be used to pass finger requests through a firewall gateway; however, it is usually considered a security risk and so disabled. Multiple levels of forwarding are permitted.

15.2.2 *Command syntax*

The syntax for the finger client that we develop is fairly simple:

```
java Finger [-l] [<username>][@[<hostname>{@<hostname>}][:<port>]]
```

The `-l` flag indicates that a verbose query should be made; the body of the finger request follows, consisting of an optional username, a sequence of host name specifications, and an optional port number. If no host is specified, the local host is queried. If more than one host name is specified, a forwarded finger query is sent to the last listed host. If no port is specified, the default port 79 is used.

```
java Finger -l
```

This produces a verbose listing of all users on the local machine.

```
java Finger merlin@sadhbh.nitric.com:8079
```

This requests information about user `merlin` on host `sadhbh.nitric.com`, using the nonstandard port number 8079.

```
java Finger -l merlin@sadhbh.nitric.com@fw.nitric.com
```

This requests verbose information about user `merlin` on host sadhbh.nitric.com; this request is sent to host fw.nitric.com, which forwards the query to host `sadhbh`. If `sadhbh` is an internal machine and `fw` is a firewall that supports finger forwarding, this allows external users to obtain potentially sensitive information about internal users, which is why it is usually disabled.

15.2.3 *Class Finger*

The finger client is implemented by a single class, `Finger`. The class is instantiated with two parameters: the actual finger request and a verbosity flag. Methods are provided to execute the finger request and to display the resulting information.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Finger {
  public static final int DEFAULT_PORT = 79;

  // public Finger (String request, boolean verbose) throws IOException ...
  // public Finger (String query, String host, int port, boolean verbose)
       throws IOException ...
  // public Reader finger () throws IOException ...
  // public static void display (Reader reader, Writer writer)
       throws IOException ...
  // public static void main (String[] args) throws IOException ...
}
```

The `Finger` class is designed to be both a reusable class and a standalone application. The first constructor accepts a `String` request, `request`, and verbosity flag, `verbose`. The request should have the form [<*username*>][@[<*hostname*>]{@<*hostname*>}[:<*port*>]]; this is automatically parsed into its component parts. An alternative constructor allows the parts to be specified separately. The `finger()` method connects to the remote host, sends the query, then returns a `Reader` from which the result can be read. A helper method, `display()`, displays the result of a finger query to a specified `Writer`. The `main()` method allows the `Finger` class to be run as a standalone application.

```
  protected boolean verbose;
  protected int port;
  protected String host, query;

  public Finger (String request, boolean verbose) throws IOException {
    this.verbose = verbose;
    int at = request.lastIndexOf ('@');
    if (at == -1) {
      query = request;
      host = InetAddress.getLocalHost ().getHostName ();
      port = DEFAULT_PORT;
    } else {
      query = request.substring (0, at);
      int colon = request.indexOf (':', at + 1);
      if (colon == -1) {
        host = request.substring (at + 1);
        port = DEFAULT_PORT;
      } else {
        host = request.substring (at + 1, colon);
        port = Integer.parseInt (request.substring (colon + 1));
      }
      if (host.equals (""))
        host = InetAddress.getLocalHost ().getHostName ();
    }
  }
```

In this constructor, we keep a copy of the flag `verbose` and parse the request `String` into the `port`, `host`, and `query` variables. If no host was specified, `localhost` is used. If no port was specified, the default is used.

To parse the request, we separate the actual user and forwarding information from the host specification (which follows the last occurence of `@`). We then separate the host specification into a host name and port number. If no port was specified, we use the default; otherwise, we parse the specified value. If no host was specified, we obtain the local host name from class `InetAddress`.

```
  public Finger (String query, String host, int port, boolean verbose)
      throws IOException {
    this.query = query;
    this.host = host.equals ("") ?
      InetAddress.getLocalHost ().getHostName () : host;
    this.port = (port == -1) ? DEFAULT_PORT : port;
    this.verbose = verbose;
  }
```

This constructor simply allows the various parts of the request to be specified separately. We use the same defaults as before if a host or port are omitted.

```
  public Reader finger () throws IOException {
    Socket socket = new Socket (host, port);
    OutputStream out = socket.getOutputStream ();
    OutputStreamWriter writer = new OutputStreamWriter (out, "latin1");
    if (verbose)
      writer.write ("/W");
    if (verbose && (query.length () > 0))
      writer.write (" ");
    writer.write (query);
    writer.write ("\r\n");
    writer.flush ();
    return new InputStreamReader (socket.getInputStream (), "latin1");
  }
```

This method connects to the remote host and writes the finger request, returning a `Reader` from which the result can be read. We first create an `OutputStreamWriter` using ISO Latin 1 character encoding. If the verbose flag was specified, then we write out `/W`, and optionally a space. We then write the request and line terminator, and flush the output stream. We don't need to add a `BufferedWriter` because `OutputStreamWriter` provides sufficient internal buffering.

We actually specify a character encoding here rather than using the default because the default may not be appropriate for the protocol. RFC 1288 specifies an international ASCII character set, of which ISO Latin 1 is the most prevalent.

```
  public static void display (Reader reader, Writer writer)
      throws IOException {
    PrintWriter printWriter = new PrintWriter (writer);
    BufferedReader bufferedReader = new BufferedReader (reader);
    String line;
    while ((line = bufferedReader.readLine ()) != null)
      printWriter.println (line);
    reader.close ();
  }
```

This method reads a finger response from the `Reader reader` and prints it to the `Writer writer`. We create a `BufferedReader` using ISO Latin 1 character encoding and then proceed to read and print lines of text.

```
  public static void main (String[] args) throws IOException {
    if (((args.length == 2) && !args[0].equals ("-l")) || (args.length > 2))
      throw new IllegalArgumentException
        ("Syntax: Finger [-l] [<username>][{@<hostname>}[:<port>]]");

    boolean verbose = (args.length > 0) && args[0].equals ("-l");
    String query = (args.length > (verbose ? 1 : 0)) ?
      args[args.length - 1] : "";

    Finger finger = new Finger (query, verbose);
    Reader result = finger.finger ();
    Writer console = new FileWriter (FileDescriptor.out);
    display (result, console);
    console.flush ();
  }
```

This method allows the `Finger` class to be executed as a standalone application. We verify that the parameters are valid, throwing an explanatory exception if not. We set the flag `verbose` if `-l` is specified, and we set `query` to be any remaining argument or an empty `String` if no further arguments are supplied. We then create a `Finger` object with these values and call its `finger()` method; if an exception occurs here, we simply pass it on. Otherwise, we create a new `Writer`, `console`, and print and flush the finger information to it.

15.2.4 Using it

The `Finger` class can either be used as part of a separate program or executed as a standalone application. Examples of standalone use follow:

```
java Finger
Login     Name              TTY  Idle   When         Bldg.     Phone
merlin    Merlin Hughes     *s0  159d   Sun 22:53
id        Rev. Blue         s6   4:13   Mon 09:00
```

This just fingers the users connected to the local machine.

```
java Finger -l
Login name: merlin      (messages off)  In real life: Merlin Hughes
Directory: /org/merlin/home/merlin      Shell: /bin/rc
On since Oct 30 22:53:17 on pty/ttys0 from alpha0
159 days Idle Time
Plan:
9 from outer space
"The good Christian should beware of mathematicians and all those
who make empty prophecies.     The danger already exists that the
```

```
mathematicians have made a covenant with the devil to darken  the
spirit and to confine man in the bonds of hell."   - St Augustine

Login name: id                          In real life: Rev. Blue
Directory: /com/disinfo/home/id         Shell: /bin/sh
On since Apr  7 09:00:45 on pty/ttys6 from ultra1
4 hours 14 minutes Idle Time
No Plan.
```

Here, we request verbose information about these users.

```
java Finger merlin@sadhbh.nitric.com@fw.nitric.com
Remote finger not allowed: merlin@sadhbh.nitric.com
```

We finally try finger forwarding: our request is refused. For further details, chapter 19 provides an example of using this class in a different manner by integrating it with the URL framework.

15.3 A DNS client

The DNS is a globally distributed database, specified in RFC 1035, that maintains the mapping from Internet host names to IP addresses, in addition to various other data including the reverse mapping. Some details of the DNS system can be found in the networking introduction. However, for the purposes of this example it is sufficient to know that a client can request the IP address of a given host name from a machine called a name server, and it will return the requested information. Behind the scenes, the name server may require contact with various other machines; however, this process is mostly transparent to the client.

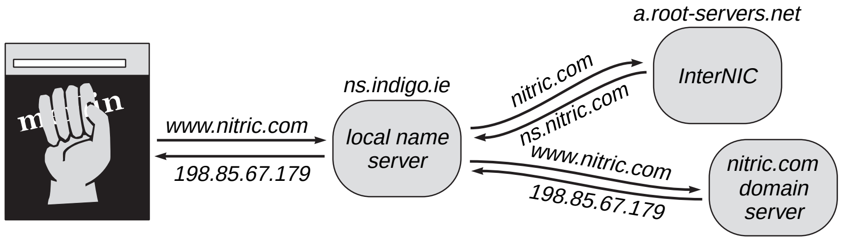

Figure 15.3 The domain name system

Most organizations and all ISPs have at least one name server to serve their connected machines. To determine a local name server that you can use with this example, either examine your own machine's DNS configuration or try similar names for your own domain or ISP as `ns.inch.com` or `ns1.aol.com`, or whatever is appropriate. A common convention is that the name server is the first machine on a subnet (e.g., host 198.85.67.193 on subnet 198.85.67.192).

The client that we develop here allows you to query a name server for information about a given host name or domain name. This can be used to determine a machine's IP address, to determine the mail exchanger for a domain, and so forth. The client framework is fairly generic and extensible and is intended to demonstrate the implementation of a more complex protocol than the finger client. We use a custom stream class as well as various other helper classes in the implementation.

To use the client, you must specify a name server to query and a host name that you wish to look up. The client connects to the name server, requests information about the name, and prints the result. There are many DNS options that are supported by the framework but, for brevity, not accessible through the simple command-line interface.

15.3.1 Protocol specification

The protocol specification for DNS is fairly complex so we shall only provide a brief and incomplete overview. For full details, consult RFC 1035.

Essentially, a client constructs a query and sends it to a name server over either TCP or UDP. The query includes some flags and a host name or domain name in which the client is interested.

The server will then respond with a series of resource records (RRs) which are basically answers to the particular question, along with additional related information. Each resource record includes one piece of information, such as the IP address of a particular host or the name of an authoritative name server for the host.

The query that is sent from a client to a name server and the response that is returned from the name server have the same form (figure 15.4).

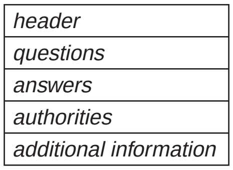

Figure 15.4 DNS queries and responses

The header specifies some flags and information about the request or response. The questions block includes one or more questions. The answers block includes zero or more answers to those questions. The authorities block includes zero or more authoritative name servers for the requested name, and the additional information block includes zero or more additional pieces of information.

Header The same header is used for queries and responses, and has the following format (figure 15.5):

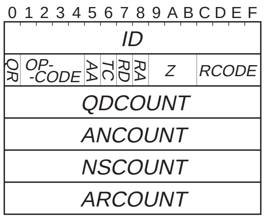

Figure 15.5 DNS headers

ID is a 16-bit identifier that the client assigns to the request; the server will include this in its response. All data is written in network byte order (high-byte first).

The next 16 bits are the query/response flags: *QR* should be 0 for a query and 1 for a response; *OPCODE* indicates the type of query, usually 0. *AA* is set in a response if it is an authoritative answer; *TC* means that the response is truncated; *RD* is set in a query if recursion is desired (the nameserver should forward the query to other nameservers); *RA* is set in a response if recursion is available from the server. The *Z* bits are reserved, and *RCODE* is a response code: An *RCODE* of zero means there was no error; otherwise, some error was encountered.

The next four fields indicate the number of questions, answers, authoritative answers, and additional resource records following the header. Each is an unsigned 16-bit value. For a query, there will usually be one question and no other resource records. For a response, the question will be returned, followed by various resource records that answer the question and provide extra information.

Questions Each question has the following format (figure 15.6):

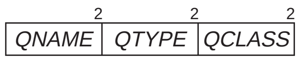

Figure 15.6 DNS questions

QNAME is the domain name or host name with which the question is concerned. The name is split into its component labels (such as *www*, *nitric*, and *com*) and then each label is written as a single-byte length followed by the label's characters. The maximum label length is 63 bytes; a complete host or domain name is terminated by a zero byte. *QTYPE* is an unsigned 16-bit value that indicates the type of resource record desired; usually 255, meaning that all information is desired, or 1, meaning that an IP address for the name is desired. *QCLASS* is an unsigned 16-bit value that indicates the network class with which this question is concerned; usually 1, the Internet.

Resource records Each resource record has the following format (figure 15.7):

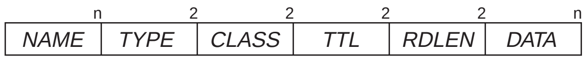

Figure 15.7 DNS resource records

NAME is the domain name or host name to which the resource record applies. *TYPE* is an unsigned `short` indicating the type of the resource record; *CLASS* is an unsigned `short` indicating the class of the resource record; *TTL* is an unsigned `int` indicating the number of seconds for which the resource record is valid. These fields are followed by *RDLEN*, an unsigned `short` indicating the length of the following data and *DATA*, which are data specific to the particular type of resource record.

Address records (A, type 1) contain a 4-byte data part, which consists of just the IP address of the corresponding host. Mail exchanger records (MX, type 15) contain a variable-length data part consisting of a 16-bit preference value followed by a domain name.

For details of the format of the remaining resource records, consult the RFCs. For more details of the format and meaning of the various fields that we have mentioned, look at the following implementation. We include rudimentary comments about the various values that some of the fields can hold.

15.3.2 *Implementation*

The implementation that we provide uses a number of classes, including:

DNS This class contains various constants that are used by the DNS-related classes, including the default DNS port number and the values of the various DNS request and response codes.

DNSQuery This class represents a DNS query. It includes details of the host name being queried and the type of query, and provides methods to allow this query to be transmitted and a response to be received.

DNSRR This class represents a DNS resource record, which is the encapsulation of a piece of DNS information. The response to a DNS query consists of a series of resource records; each RR has a type and some accompanying information. After a `DNSQuery` has been transmitted and a response received, the returned information can be extracted from the `DNSQuery` as a sequence of `DNSRR`s.

`DNSRR` is actually an abstract superclass for the different types of possible resource records; we will document some of these, but for brevity's sake, the remainder will only be available on the book's accompanying Web site.

record.Address This class in the `record` package represents an address record, which is a DNS response that gives the IP address of a particular host name.

record.MailExchanger This class represents a mail exchanger record, which is a DNS response that gives the name and priority of the machine to which mail for a particular domain should go.

DNSInputStream This class is an `InputStream` that provides helper methods to decode the typical data that are returned in a DNS response.

NSLookup This is a command-line nslookup client that uses these DNS classes to perform DNS resolution.

In essence, the `NSLookup` command constructs a `DNSQuery` for the host name and information in which it is interested. It then transmits this query to a name server and receives a response. The `DNSQuery` class then uses a `DNSInputStream` to decode this response into a series of `DNSRR` records. The `NSLookup` class can then extract and process these responses using the general `DNSRR` methods and the resource-record-specific `DNSRR` subclass methods.

15.3.3 *Command syntax*

The syntax for the nslookup client that we develop is as follows:

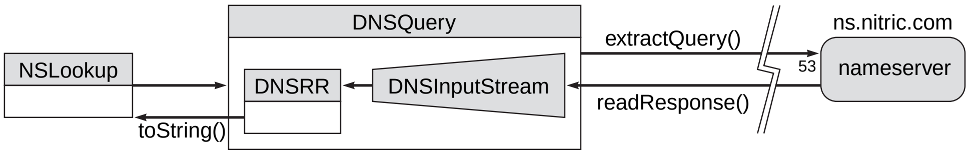

Figure 15.8 The DNS framework

```
java NSLookup <hostname>[@<nameserver>]
```

Host name is the name of a host or domain name about which we wish to find information. This must be a fully qualified domain name such as www.att.com or crypto.org; a partial name such as www is not valid. The name server is the optional name of a name server to query; if none is specified, a default name ns is used. Usually you must use the name of your local name server here, or if you wish to query another name server, then supply its name or address.

15.3.4 Class DNS

This class contains various DNS-related constants and some helper methods.

```
public final class DNS {
}
```

This class simply provides constants so we don't need to subclass anything.

```
  public static final int
    DEFAULT_PORT = 53;
```

The default TCP and UDP port for DNS requests is number 53.

```
  public static final int
    TYPE_A     = 1,   // address
    TYPE_NS    = 2,   // nameserver
    TYPE_MD    = 3,   // mail domain
    TYPE_MF    = 4,   // mail forwarder
    TYPE_CNAME = 5,   // canonical name
    TYPE_SOA   = 6,   // start of authority
    TYPE_MB    = 7,   // mail box
    TYPE_MG    = 8,   // mail group
    TYPE_MR    = 9,   // mail rename
    TYPE_NULL  = 10,  // null
    TYPE_WKS   = 11,  // well-known services
    TYPE_PTR   = 12,  // pointer
    TYPE_HINFO = 13,  // host info
    TYPE_MINFO = 14,  // mail info
    TYPE_MX    = 15,  // mail exchanger
    TYPE_TXT   = 16,  // text
    TYPE_AXFR  = 252, // zone transfer request
    TYPE_MAILB = 253, // mailbox request
    TYPE_MAILA = 254, // mail agent request
    TYPE_ANY   = 255; // request any
```

Every DNS request includes a value that indicates the *type* of record desired; it can be any one of these values, including TYPE_ANY which indicates that all relevant resource records are wanted. Each resource record in the response includes a value that indicates its type; it is common to receive additional types of record than those requested. The last four types are for requests only; a response will only be of type TYPE_A through TYPE_TXT.

```
  public static final int
    CLASS_IN = 1,     // internet
    CLASS_CS = 2,     // csnet
    CLASS_CH = 3,     // chaos
    CLASS_HS = 4,     // hesiod
    CLASS_ANY = 255;  // request any
```

The domain naming system was designed to cater to different types of networks other than just the Internet. Therefore, requests and responses must also state the *class* in which they are interested or to which they correspond. CLASS_IN is most common, although CLASS_ANY may be used for a request.

```
  public static final int
    SHIFT_QUERY = 15,
    SHIFT_OPCODE = 11,
    SHIFT_AUTHORITATIVE = 10,
    SHIFT_TRUNCATED = 9,
    SHIFT_RECURSE_PLEASE = 8,
    SHIFT_RECURSE_AVAILABLE = 7,
    SHIFT_RESERVED = 4,
    SHIFT_RESPONSE_CODE = 0;
```

These values indicate the bit locations of pieces of information in the *flag* field of a DNS header.

```
  public static final int
    OPCODE_QUERY = 0,
    OPCODE_IQUERY = 1,
    OPCODE_STATUS = 2;
```

These are the various values that the opcode part of the header flags may take.

```
  private static final String[] typeNames = {
    "Address", "NameServer", "MailDomain", "MailForwarder",
    "CanonicalName", "StartOfAuthority", "MailBox", "MailGroup",
    "MailRename", "Null", "WellKnownServices", "Pointer",
    "HostInfo", "MailInfo", "MailExchanger", "Text"
  };
  public static String typeName (int type) {
    return ((type >= 1) && (type <= 16)) ? typeNames[type - 1] : "Unknown";
  }
```

This method returns a textual representation of the specified resource record type, `type`, or `Unknown` if it is not a valid type.

```
  private static final String[] codeNames = {
    "Format error", "Server failure", "Name not known",
    "Not implemented", "Refused"
  };

  public static String codeName (int code) {
    return ((code >= 1) && (code <= 5)) ?
      codeNames[code - 1] : "Unknown error";
  }
```

This method returns a textual representation of the specified error code, `code`, or `Unknown error` if it is not a known error code.

15.3.5 Class DNSQuery

This class represents a DNS query. It is constructed with a host name about which information is desired, a query type, and a query class. It then provides support for transmission to a remote host and decoding the response.

```
import java.io.*;
import java.util.*;

public class DNSQuery {
  // public DNSQuery (String host, int type, int clas) ...
  // public String getQueryHost () ...
  // public int getQueryType () ...
  // public int getQueryClass () ...
  // public int getQueryID () ...
  // public byte[] extractQuery () ...
  // public void receiveResponse (byte[] data, int length)
       throws IOException ...
  // public boolean isAuthoritative () ...
  // public boolean isTruncated () ...
  // public boolean isRecursive () ...
  // public Enumeration getAnswers () ...
  // public Enumeration getAuthorities () ...
  // public Enumeration getAdditional () ...
}
```

The DNS query information is specified in the constructor. The query can be converted into a byte-array for transmission to a name server with the `extractQuery()` method, and then the response can be decoded with the `receiveResponse()` method.

After a response has been decoded, it can be queried through the remaining methods of this class.

```
private String queryHost;
private int queryType, queryClass, queryID;
private static int globalID;

public DNSQuery (String host, int type, int clas) {
  StringTokenizer labels = new StringTokenizer (host, ".");
  while (labels.hasMoreTokens ())
    if (labels.nextToken ().length () > 63)
      throw new IllegalArgumentException ("Invalid hostname: " + host);
  queryHost = host;
  queryType = type;
  queryClass = clas;
  synchronized (getClass ()) {
    queryID = (++ globalID) % 65536;
  }
}
```

In the constructor, we verify that the host name is valid (each element must be shorter than 64 characters) and then store the supplied query information in the variables `queryHost`, `queryType`, and `queryClass`. We also allocate a VM-unique identifier, `queryID`, that is used to identify the DNS query.

```
public String getQueryHost () {
  return queryHost;
}
```

This method returns the host about which this `DNSQuery` is seeking information.

```
public int getQueryType () {
  return queryType;
}
```

This method returns the resource record type in which this `DNSQuery` is interested.

```
public int getQueryClass () {
  return queryClass;
}
```

This method returns the resource class in which this `DNSQuery` is interested.

```
public int getQueryID () {
  return queryID;
}
```

This method returns the identifier that was allocated to this `DNSQuery`.

```
public byte[] extractQuery () {
    ByteArrayOutputStream byteArrayOut = new ByteArrayOutputStream ();
    DataOutputStream dataOut = new DataOutputStream (byteArrayOut);
    try {
      dataOut.writeShort (queryID);
      dataOut.writeShort ((0 << DNS.SHIFT_QUERY) |
        (DNS.OPCODE_QUERY << DNS.SHIFT_OPCODE) |
        (1 << DNS.SHIFT_RECURSE_PLEASE));
      dataOut.writeShort (1); // # queries
      dataOut.writeShort (0); // # answers
      dataOut.writeShort (0); // # authorities
      dataOut.writeShort (0); // # additional
      StringTokenizer labels = new StringTokenizer (queryHost, ".");
      while (labels.hasMoreTokens ()) {
        String label = labels.nextToken ();
        dataOut.writeByte (label.length ());
        dataOut.writeBytes (label);
      }
      dataOut.writeByte (0);
      dataOut.writeShort (queryType);
      dataOut.writeShort (queryClass);
    } catch (IOException ignored) {
    }
    return byteArrayOut.toByteArray ();
  }
```

This method returns this DNS query encoded as a byte array in the correct format for transmission to a name erver. We create the query array by attaching a DataOutputStream, dataOut, to a ByteArrayOutputStream, byteArrayOut, and writing the query using the standard DataOutputStream methods.

We first write the query identifier, followed by the flags and the number of queries, answers, authorities, and additional resource records in this query. We next write the query, which consists of the host name, the query type, and the query class. We encode the host name by breaking it into its component parts (such as www, nitric, com) and then writing each part preceded by a byte that indicates the length of the part. We terminate the host name with a zero byte.

For this purpose, we could have created a DNSOutputStream that subclasses ByteArrayOutputStream and adds these data-writing methods. However, we would only use the stream once so the additional class would not serve much purpose.

Writing to a ByteArrayOutputStream will not throw an IOException, so we add a dummy exception handler and simply return the contents of byteArrayOut.

```
  private Vector answers = new Vector ();
  private Vector authorities = new Vector ();
  private Vector additional = new Vector ();

  public void receiveResponse (byte[] data, int length) throws IOException {
    DNSInputStream dnsIn = new DNSInputStream (data, 0, length);
    int id = dnsIn.readShort ();
    if (id != queryID)
      throw new IOException ("ID does not match request");
    int flags = dnsIn.readShort ();
    decodeFlags (flags);
    int numQueries = dnsIn.readShort ();
    int numAnswers = dnsIn.readShort ();
    int numAuthorities = dnsIn.readShort ();
    int numAdditional = dnsIn.readShort ();

    while (numQueries -- > 0) { // discard questions
      String queryName = dnsIn.readDomainName ();
      int queryType = dnsIn.readShort ();
      int queryClass = dnsIn.readShort ();
    }
    try {
      while (numAnswers -- > 0)
        answers.addElement (dnsIn.readRR ());
      while (numAuthorities -- > 0)
        authorities.addElement (dnsIn.readRR ());
      while (numAdditional -- > 0)
        additional.addElement (dnsIn.readRR ());
  }   catch (EOFException ex) {
      if (!truncated)
        throw ex;
    }
  }

  // protected void decodeFlags (int flags) throws IOException ...
```

This method decodes a DNS response, which must be in the form of a byte array. We use byte arrays rather than streams because DNS requests can be transported over either UDP or TCP; abstracting to byte arrays allows us to use either transport mechanism.

We create a `DNSInputStream` from the response array and read data from this stream. The header of the response has the same format as the request: We read the response ID and verify that it matches our own, throwing an appropriate `IOException` if not. We then read the response flags and the number of queries, answers, authorities, and additional resource records that will follow.

We loop through, reading and discarding the query data; this is exactly what we send out in the initial query. We could verify that this matches the query that we sent out, but that is probably unnecessary.

We finally read the resource records that were returned in response to our request. We use the `readRR()` method to decode these from the stream, inserting them into the appropriate `Vector`: `answers`, `authorities`, or `additional`. If the response was truncated and an `EOFException` occurs, then we simply ignore it and return with as much data as we have managed to decode.

```
  private boolean authoritative, truncated, recursive;

  protected void decodeFlags (int flags) throws IOException {
    boolean isResponse = ((flags >> DNS.SHIFT_QUERY) & 1) != 0;
    if (!isResponse)
      throw new IOException ("Response flag not set");
    int opcode = (flags >> DNS.SHIFT_OPCODE) & 15;
    // could check opcode
    authoritative = ((flags >> DNS.SHIFT_AUTHORITATIVE) & 1) != 0;
    truncated = ((flags >> DNS.SHIFT_TRUNCATED) & 1) != 0;
    boolean recurseRequest = ((flags >> DNS.SHIFT_RECURSE_PLEASE) & 1) != 0;
    // could check recurse request
    recursive = ((flags >> DNS.SHIFT_RECURSE_AVAILABLE) & 1) != 0;
    int code = (flags >> DNS.SHIFT_RESPONSE_CODE) & 15;
    if (code != 0)
      throw new IOException (DNS.codeName (code) + " (" + code + ")");
  }
```

This method decodes the specified flags, `flags`. We verify that the response flag is set. We could check other fields such as opcode and recurse request, but this is probably unnecessary. We extract whether this is an authoritative response, whether it is truncated, and whether recursion was available. We also check the response code; if this is non-zero, an error was encountered so we throw an appropriate exception.

```
  public boolean isAuthoritative () {
    return authoritative;
  }
```

This method queries whether the response is authoritative, which means that it came from a name server that holds authoritative data for the requested domain.

```
  public boolean isTruncated () {
    return truncated;
  }
```

This method queries whether the response was truncated; this will happen if there was too much data in the response to fit into the transport limit. UDP responses are limited to just 512 bytes; any more data will simply be discarded.

```
  public boolean isRecursive () {
    return recursive;
  }
```

This method returns whether the response was the result of a recursive query; this means that the name server had to forward the query to another name server in order to determine the information.

```
  public Enumeration getAnswers () {
    return answers.elements ();
  }
```

This method returns an `Enumeration` of the resource records from the answers section of the DNS response. These are the direct answers to the specified request.

```
  public Enumeration getAuthorities () {
    return authorities.elements ();
  }
```

This method returns an `Enumeration` of the resource records from the authorities section of the DNS response. This is a list of the authoritative name servers for the specified request. If the original question was not answered, one of these should be asked instead.

```
  public Enumeration getAdditional () {
    return additional.elements ();
  }
```

This method returns an `Enumeration` of the resource records from the additional information section of the DNS response. This is a list of additional information, such as the addresses of the listed authoritative name servers.

15.3.6 Class DNSInputStream

This class is a `ByteArrayInputStream` that provides special methods related to reading data from a DNS response. We directly subclass `ByteArrayInputStream` and add these methods rather than implementing a stream filter because some features of the DNS format require essentially random access to data that have previously been read.

```
import java.io.*;
import java.util.*;

public class DNSInputStream extends ByteArrayInputStream {
  // public DNSInputStream (byte[] data, int off, int len) ...
  // public int readByte () throws IOException ...
  // public int readShort () throws IOException ...
  // public long readInt () throws IOException ...
  // public String readString () throws IOException ...
  // public String readDomainName () throws IOException ...
  // public DNSRR readRR () throws IOException ...
}
```

We extend `ByteArrayInputStream` and add methods that allow us to read the various DNS datatypes including unsigned `bytes`, `shorts`, `ints`, `Strings`, domain names, and resource records.

```
  protected DataInputStream dataIn;
  public DNSInputStream (byte[] data, int off, int len) {
    super (data, off, len);
    dataIn = new DataInputStream (this);
  }
```

Our constructor calls the superconstructor to read from the specified `len`-byte subarray of `data`, starting from offset `off`. We also attach a `DataInputStream`, `dataIn`, to `this`, to provide easy read access through the basic methods that provides.

```
  public int readByte () throws IOException {
    return dataIn.readUnsignedByte ();
  }
```

This method reads an unsigned `byte` from this stream.

```
   public int readShort () throws IOException {
    return dataIn.readUnsignedShort ();
  }
```

This method reads an unsigned `short` from this stream.

```
  public long readInt () throws IOException {
    return dataIn.readInt () & 0xffffffffL;
  }
```

This method reads an unsigned `int` from this stream, returning the resulting `long`.

```
  public String readString () throws IOException {
    int len = readByte ();
    if (len == 0) {
      return "";
    } else {
      byte[] buffer = new byte[len];
      dataIn.readFully (buffer);
      return new String (buffer, "latin1");
    }
  }
```

This method reads a `String`, encoded as we wrote the parts of a domain name in the `DNSQuery` class: we read a byte, `len`, which is the length of the `String`, and then read the indicated number of bytes, returning the result as a `String`.

```
  public String readDomainName () throws IOException {
    if (pos >= count)
      throw new EOFException ("EOF reading domain name");
    if ((buf[pos] & 0xc0) == 0) {
      String label = readString ();
      if (label.length () > 0) {
        String tail = readDomainName ();
```

```
          if (tail.length () > 0)
            label = label + '.' + tail;
        }
        return label;
      } else {
        if ((buf[pos] & 0xc0) != 0xc0)
          throw new IOException ("Invalid domain name compression offset");
        int offset = readShort () & 0x3fff;
        DNSInputStream dnsIn =
          new DNSInputStream (buf, offset, buf.length - offset);
        return dnsIn.readDomainName ();
      }
    }
```

This method reads a domain name. As we saw, when writing a domain name in the `DNSQuery` class, the name is simply split into its component parts and then each is written, preceded by its length, followed ultimately by a zero-byte.

For the purposes of space efficiency in the response, however, a special compression option is supported: if any length-byte has either of its top 2 bits set, then it and the next byte form a pointer to part of a domain name somewhere else in the response. This way, for example, www.nitric.com and ns.nitric.com can be encoded in a single response as www, nitric, and com, and ns followed by a pointer to the earlier occurrence of nitric.com.

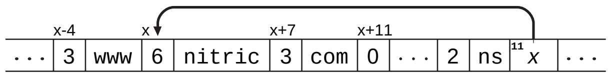

Figure 15.9 DNS name compression

In the `readDomainName()` method, we examine the next `length` byte: if both the top bits are clear, it is a normally encoded string; otherwise it is a back reference pointer. If it is a normal string, we can use `readString()` to read it, followed by `readDomainName()` for the rest of the name.

Otherwise, we read the encoded pointer by reading a `short` and discarding the top 2 bits. We can then construct a temporary new `DNSInputStream` that reads from the specified offset, and call `readDomainName()` on this new stream to decode the rest of the domain name stored there. This decompression process may be recursive if the remainder of the domain name itself contains a pointer reference.

```
  public DNSRR readRR () throws IOException {
    String rrName = readDomainName ();
    int rrType = readShort ();
    int rrClass = readShort ();
    long rrTTL = readInt ();
    int rrDataLen = readShort ();
    DNSInputStream rrDNSIn = new DNSInputStream (buf, pos, rrDataLen);
    pos += rrDataLen;
    try {
      String myName = getClass ().getName ();
      int periodIndex = myName.lastIndexOf ('.');
      String myPackage = myName.substring (0, 1 + periodIndex);
      Class theClass = Class.forName
        (myPackage + "record." + DNS.typeName (rrType));
      DNSRR rr = (DNSRR) theClass.newInstance ();
       rr.init (rrName, rrType, rrClass, rrTTL, rrDNSIn);
       return rr;
    } catch (ClassNotFoundException ex) {
      throw new IOException ("Unknown DNSRR (type " +
        DNS.typeName (rrType) + " (" + rrType + "))");
    } catch (IllegalAccessException ex) {
      throw new IOException ("Access error creating DNSRR (type " +
        DNS.typeName (rrType) + ')');
    } catch (InstantiationException ex) {
      throw new IOException ("Instantiation error creating DNSRR " +
        "(type " + DNS.typeName (rrType) + ')');
    }
  }
```

This method reads a resource record. The resource record consists of a domain name, a resource record type and class, a time to live, and a length followed by the resource record-specific data. We read the initial data and then extract a substream for the resource record data. This substream reads from just the resource record data part of this stream.

We finally construct a `DNSRR` and initialize it with the data that we have read. Rather than combining all resource records into a single `DNSRR` class, we use a generic `DNSRR` superclass and then provide a subclass for each type of resource record. The name of the subclass is constructed by joining `"record."` and the name of the resource record type. We use `Class.forName().newInstance()` to create an instance of this subclass, using the type name returned by the `typeName()` method of class DNS. We then initialize this resource record with the data that we read and the substream that we created. We rethrow any exceptions that arise during this process, or else return the resulting `DNSRR` instance.

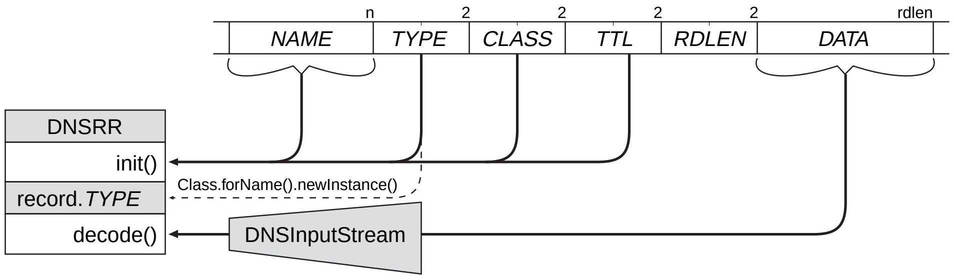

Figure 15.10 Reading a resource record

15.3.7 Class DNSRR

This is the generic superclass for all resource record classes. This superclass provides the methods that are supported by all resource records, including methods to query the standard resource record data. Subclasses may add whatever extra methods are appropriate for the data that they represent.

```
import java.io.*;

public abstract class DNSRR {
  // public String getRRName () ...
  // public int getRRType () ...
  // public int getRRClass () ...
  // public long getRRTTL () ...
  // public boolean isValid () ...
}
```

The `DNSRR` superclass provides methods to query the basic resource record data that are found at the start of every resource record. We declare this class abstract because it must be extended by a subclass that is appropriate for a particular resource record type.

```
  private String rrName;
  private int rrType, rrClass;
  private long rrTTL, rrCreated;

  void init (String name, int type, int clas, long ttl, DNSInputStream dnsIn)
      throws IOException {
    rrName = name;
    rrType = type;
    rrClass = clas;
    rrTTL = ttl;
    rrCreated = System.currentTimeMillis ();
    decode (dnsIn);
  }

  // protected abstract void decode (DNSInputStream dnsIn)
       throws IOException ...
```

This method is called by a `DNSInputStream` to initialize the `DNSRR` with the basic resource record data. We keep local copies of this information and the current time, and then call the `decode()` method to decode the supplied stream, which is a `DNSInput-Stream` attached to just the data that is specific to this resource record.

```
  protected abstract void decode (DNSInputStream dnsIn) throws IOException;
```

A subclass must implement this method to decode the specified `DNSInputStream`, `dnsIn`, and extract any resource record-specific data.

```
  public String getRRName () {
    return rrName;
  }
```

This method returns the name of the host or domain to which this resource record applies.

```
  public int getRRType () {
    return rrType;
  }
```

This method returns the type of this resource record; this will be one of the types defined in the `DNS` class.

```
  public int getRRClass () {
    return rrClass;
  }
```

This method returns the class of this resource record; this will be one of the classes defined in the `DNS` class.

```
  public long getRRTTL () {
    return rrTTL;
  }
```

This method returns the time to live (TTL) of this resource record, which is the number of seconds for which it is valid. After this time has expired, another query should be made to determine accurate and fresh information. A TTL of zero means that this resource record is only valid for the current transaction.

```
  public boolean isValid () {
    return rrTTL * 1000 > System.currentTimeMillis () - rrCreated;
  }
```

This method returns whether this resource record is still valid; that is to say, whether its time to live has not yet expired. This does not cater to a TTL of zero; the caller must be prepared to handle that case manually.

15.3.8 Class record.Address

This class represents a DNS address record, which is a resource record that specifies the IP address of a particular host.

```
package record;
import java.io.*;
import java.net.*;

public class Address extends DNSRR {
  // public byte[] getAddress () ...
  // public InetAddress getInetAddress () ...
  // public String toString () ...
}
```

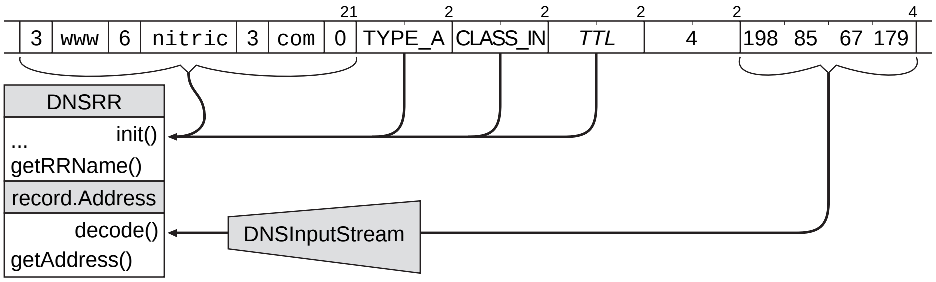

Figure 15.11 Class record.Address

We extend the `DNSRR` class and add a method `getAddress()` that returns the 4-byte IP address represented by this resource record. We also add a `getInetAddress()` method that returns this value as an `InetAddress` object and a `toString()` method that returns a useful `String` representation of this class.

```
  private int[] ipAddress = new int[4];

  protected void decode (DNSInputStream dnsIn) throws IOException {
    for (int i = 0; i < 4; ++ i)
      ipAddress[i] = dnsIn.readByte ();
  }
```

The `decode()` method reads 4 bytes from the `DNSInputStream` `dnsIn` and stores these in the array `ipAddress`.

```
  public byte[] getAddress () {
    byte[] ip = new byte[4];
    for (int j = 0; j < 4; ++ j)
      ip[j] = (byte) ipAddress[j];
    return ip;
  }
```

This method creates and returns a new 4-entry byte array containing the IP address from `ipAddress`.

```
  public InetAddress getInetAddress () throws UnknownHostException {
    return InetAddress.getByName (toByteString ());
  }

  // private String toByteString () ...
```

This method returns a new `InetAddress`, created from the result of the `toByteString()` method.

```
  private String toByteString () {
    return ipAddress[0] + "." + ipAddress[1] + "." +
      ipAddress[2] + "." + ipAddress[3];
  }
```

This method returns the IP address as a sequence of 4 period-separated numbers, for example, *12.34.56.78*.

```
  public String toString () {
    return getRRName () + "\tinternet address = " + toByteString ();
  }
```

This method returns a `String` representation of this resource record, including the target host, as returned by `getRRName()` and its IP address, as returned by `toByteString()`.

15.3.9 *Class record.MailExchanger*

This class represents a DNS mail exchanger record. These records indicate the host that should receive mail for a particular domain. Each record includes a host name that should receive mail and a preference that indicates in what order the mail exchangers should be tried. Domains typically have several mail exchangers that are tried in sequence; should one go down then another can receive mail until the primary exchanger is reanimated.

```
package record;
import java.io.*;

public class MailExchanger extends DNSRR {
  // public String getMX () ...
  // public int getPreference () ...
  // public String toString () ...
}
```

We extend `DNSRR` and add `getMX()`, `getPreference()`, and `toString()` methods that return the mail exchanger and its preference, and a useful `String` representation of this resource record.

```
  private int preference;
  private String mx;

  protected void decode (DNSInputStream dnsIn) throws IOException {
    preference = dnsIn.readShort ();
    mx = dnsIn.readDomainName ();
  }
```

In the `decode()` method, we read the preference as an unsigned `short` and the mail exchanger's domain name, which is encoded in the usual domain name encoding.

```
  public String getMX () {
    return mx;
  }
```

This method returns the mail exchanger's host name.

```
  public int getPreference () {
    return preference;
  }
```

This method returns the mail exchanger's preference.

```
  public String toString () {
    return getRRName () + "\tpreference = " + preference +
      ", mail exchanger = "+ mx;
  }
```

This method returns a `String` representation of this resource record.

This is the last resource record implementation that we will look at here; all others follow essentially the same pattern, reading data with the methods of `DNSInputStream` and then supplying methods to query this information. Implementations of the other fourteen types are available from the accompanying online site.

15.3.10 Class NSLookup

This class implements a simple DNS resolver client, similar in nature to the `nslookup` command. To run this example, specify a host name and optional name server. It will construct a `DNSQuery`, send it to the name server, receive a response, and print the resulting information.

This client uses TCP/IP and automatically requests all Internet-class resource records for the specified host name or domain name.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class NSLookup {
  // public static void main (String args[]) ...
  // public static void sendQuery (DNSQuery query, Socket socket)
       throws IOException ...
  // public static void getResponse (DNSQuery query, Socket socket)
       throws IOException ...
  // public static void printRRs (DNSQuery query) ...
}
```

We provide a `main()` method that decodes the arguments; constructs a `DNSQuery`; opens a `Socket` to the name server; and then uses the `sendQuery()`, `getResponse()`, and `printResult()` methods to transmit the query, receive the response, and print the resulting resource records.

```
  public static void main (String[] args) {
    if (args.length != 1)
      throw new IllegalArgumentException
        ("Syntax: NSLookup <hostname>[@<nameserver>]");

    int atIdx = args[0].indexOf ("@");
    String nameServer = (atIdx > -1) ? args[0].substring (atIdx + 1) : "ns";
    String hostName = (atIdx > -1) ? args[0].substring (0, atIdx) : args[0];

    System.out.println ("Nameserver: " + nameServer);
    System.out.println ("Request: " + hostName);

    DNSQuery query = new DNSQuery (hostName, DNS.TYPE_ANY, DNS.CLASS_IN);

    try {
      Socket socket = new Socket (nameServer, DNS.DEFAULT_PORT);
      socket.setSoTimeout (10000);
      sendQuery (query, socket);
      getResponse (query, socket);
      socket.close ();

      printRRs (query);
    } catch (IOException ex) {
      System.out.println (ex);
    }
  }
```

In the `main()` method, we first verify that an argument has been supplied, throwing an appropriate exception if not. We then decode the argument into its component parts: If just a host name was specified, we assume a default name server called *ns.* Otherwise, we split the argument into a host name, `hostName`, and name server, `nameServer`.

We construct a `DNSQuery`, `query`, for the specified host name, requesting *any* resource records in the `Internet` class. We open a `Socket` to the name server on the default DNS port, transmit the query and then await a response. If the name server does not exist or is not operational, an exception will be thrown. Sometimes a name server will simply not respond at all, so we set a 20-second receive timeout. Note that the connection attempt, when we create the `Socket`, may in some cases take a long time to fail if there is a network problem.

We finally call upon the `printRRs()` method to display the returned resource records.

```
  public static void sendQuery (DNSQuery query, Socket socket)
      throws IOException {
```

```
    BufferedOutputStream bufferedOut =
      new BufferedOutputStream (socket.getOutputStream ());
    DataOutputStream dataOut = new DataOutputStream (bufferedOut);
    byte[] data = query.extractQuery ();
    dataOut.writeShort (data.length);
    dataOut.write (data);
    dataOut.flush ();
  }
```

This method sends the specified `DNSQuery`, `query`, out the specified `Socket`, `socket`. We create a buffered `DataOutputStream`, `dataOut`, attached to the `Socket`'s output stream. We extract the query into a byte array using its `extractQuery()` method; we write its length to the output stream using the `writeShort()` method and then write the entire query before flushing the output stream. When sending a DNS query over TCP, it is necessary to precede the query by its length; this is not necessary over UDP because UDP packets automatically contain the payload length.

```
  public static void getResponse (DNSQuery query, Socket socket)
      throws IOException {
    InputStream bufferedIn =
      new BufferedInputStream (socket.getInputStream ());
    DataInputStream dataIn = new DataInputStream (bufferedIn);
    int responseLength = dataIn.readUnsignedShort ();
    byte[] data = new byte[responseLength];
    dataIn.readFully (data);
    query.receiveResponse (data, responseLength);
  }
```

This method reads a response from the specified `Socket`, `socket`, into the specified `DNSQuery`, `query`. We create a buffered `DataInputStream`, `dataIn`, attached to the `Socket`'s input stream. We read an unsigned `short`, which indicates the length of the following response, and then read the entire response using the `readFully()` method. We then call `query`'s `receiveResponse()` method to decode the received data.

```
  public static void printRRs (DNSQuery query) {
    Enumeration answers = query.getAnswers ();
    if (answers.hasMoreElements ())
      System.out.println (query.isAuthoritative () ?
        "\nAuthoritative answer:\n" :
        "\nNon-authoritative answer:\n");
    while (answers.hasMoreElements ())
      System.out.println (answers.nextElement ());

    Enumeration authorities = query.getAuthorities ();
    if (authorities.hasMoreElements ())
      System.out.println ("\nAuthoritative answers can be found from:\n");
    while (authorities.hasMoreElements ())
      System.out.println (authorities.nextElement ());
    Enumeration additional = query.getAdditional ();
    if (additional.hasMoreElements ())
      System.out.println ("\nAdditional information:\n");
    while (additional.hasMoreElements ())
      System.out.println (additional.nextElement ());
  }
```

This method prints out the responses received for our query. We extract all the direct answers into the `Enumeration answers`; we print whether the result is authoritative or not, and then print each answer. We then print out all the authorities returned with this request, followed by any additional resource records that were returned.

This example is, perforce, simple. A more complex implementation would automatically use some of the resource record-specific methods of the returned data in order to provide increased function. For example, if no answers were returned to a request except for a list of authoritative name servers, we could automatically query those listed name servers. Similarly, we could automatically obtain additional information about returned canonical name records and so forth.

15.3.11 Using it

The `NSLookup` class is most easily demonstrated when used as a stand-alone application.

```
java NSLookup nitric.com
Nameserver: ns
Request: nitric.com

Authoritative answer:

prominence.com  start of authority
        origin = ns.nitric.com
        mail address = hostmaster.nitric.com
        serial = 1062
        refresh = 14400
        retry = 3600
        expire = 604800
        minimum TTL = 86400
nitric.com  nameserver = ns.nitric.com
nitric.com  internet address = 198.85.67.179
nitric.com  preference = 1000, mail exchanger = mail2.catalogue.com
nitric.com  preference = 10, mail exchanger = mailman.nitric.com
nitric.com  preference = 100, mail exchanger = mail.catalogue.com

Authoritative answers can be found from:

nitric.com  nameserver = ns.nitric.com

Additional information:

mailman.nitric.com      internet address = 198.85.67.179
mail.catalogue.com      internet address = 198.85.68.40
```

This looks up information about the domain nitric.com using our local name server, ns.

```
java nslookup.NSLookup internic.net@ns.isi.edu
Nameserver: ns.isi.edu
Request: internic.net

Authoritative answer:

internic.net    start of authority
        origin = ops.internic.net
        mail address = markk.internic.net
        serial = 970506000
        refresh = 3600
        retry = 3600
        expire = 432000
        minimum TTL = 86400
internic.net    nameserver = rs0.internic.net
internic.net    nameserver = ds0.internic.net
internic.net    nameserver = noc.cerf.net
internic.net    nameserver = ns.isi.edu
internic.net    text = WP-DAP://c=US@o=NSFNET@ou=InterNIC
internic.net    text = WP-SMTP-EXPN-Finger://internic.net
internic.net    preference = 100, mail exchanger = rs.internic.net
internic.net    internet address = 198.41.0.8
internic.net    internet address = 198.41.0.9
internic.net    internet address = 198.41.0.6
internic.net    internet address = 198.41.0.5

Authoritative answers can be found from:

internic.net    nameserver = rs0.internic.net
internic.net    nameserver = ds0.internic.net
internic.net    nameserver = noc.cerf.net
internic.net    nameserver = ns.isi.edu

Additional information:

rs0.internic.net        internet address = 198.41.0.5
ds0.internic.net        internet address = 198.49.45.10
noc.cerf.net    internet address = 192.153.156.22
ns.isi.edu      internet address = 128.9.128.127
rs.internic.net internet address = 198.41.0.8
rs.internic.net internet address = 198.41.0.9
rs.internic.net internet address = 198.41.0.12
rs.internic.net internet address = 198.41.0.5
rs.internic.net internet address = 198.41.0.6
rs.internic.net internet address = 198.41.0.7
```

This looks up information about the domain internic.net, querying the specified name server ns.isi.edu.

For a further example of the usage of these classes, chapter 21 demonstrates a UDP version of this command.

15.4 Wrapping up

In this chapter, we have looked at the implementations of two TCP clients. The first, finger, we implemented with a single class that performed all necessary function for the protocol. For the sake of reusability, we did not inline all code as `static` methods called by `main()` but instead provided a `Finger` object that could be constructed with desired parameters and then printed to any desired `Writer`. Because we provide this reusable interface, we will be able to use this class again in our discussion of the URL classes.

The DNS client, on the other hand, was implemented with many supporting classes. We could have implemented it in a single monolithic class, but the implementation that we chose has the advantage that it can be used by many different pieces of code. We demonstrated transport of DNS requests over TCP. In a later chapter, we will reuse these classes to provide DNS transport over UDP. Similarly, it is possible to make use of the DNS classes in a SMTP (mail) client. We can make a DNS request to determine the mail exchanger for a particular domain and can then connect appropriately. When mail is sent to the user x@nitric.com, a DNS request is made to determine the mail exchanger for the domain nitric.com; a TCP connection is then made to the SMTP port of the indicated machine. Because we have a generic reusable framework in place, we can simply use the classes of this example to query records of type `DNSRR_MX` and determine the domain information that we require (figure 15.12).

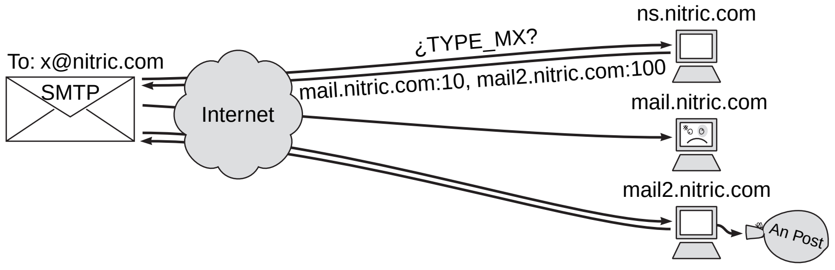

Figure 15.12 MX records

The trade-off between a simple but nonflexible approach (one class with `static` methods) and a generalized but slightly more voluminous approach is fairly common. The appropriate use of the powerful language mechanisms to provide a generalized solution is a better choice in almost all cases. Our use of a `DNSInputStream` class is much more elegant than explicitly dissecting an array of bytes in the main class, although the latter approach requires less initial design.

Following this introduction to TCP network clients we will now look at some servers. The next chapters cover the server API with some fairly simple examples, followed by an implementation of a real-world protocol: HTTP.