UNIVERSITY COLLEGE LONDON
DEPARTMENT OF COMPUTER SCIENCE
Dr Kyle Jamieson and Dr Brad Karp

3035/GZ01 Networked Systems
Individual Coursework 2: Ben's Local DNS Server
Distributed: 3rd November 2011; Due: 09:00 24th November 2011

Introduction

Having just learned about the Domain Name System (DNS), GZ01 student Ben Bitdiddle's interest is piqued. In his spare time, Ben runs a small ISP, and of course, he wants to offer his customers a reliable DNS service. Suspicious of recent cache poisoning security exploits in the DNS server implementation he currently runs, he decides to write his own server in Python. With some of the profits from his ISP business, Ben hires you and a fellow GZ01 student, Alyssa P. Hacker, to help.

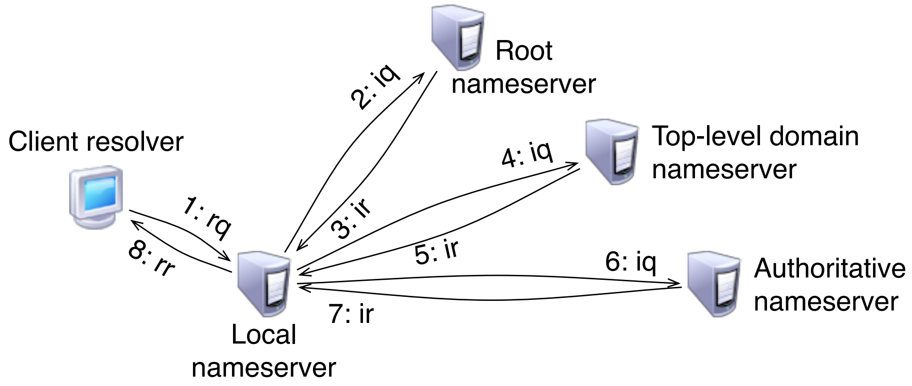

FIGURE 1: A recursive DNS lookup is made up of many iterative lookups, each initiated by the local nameserver.

Recall the overall structure of the DNS from lecture: the local name server is placed close to the pool of clients that it serves, and normally only accepts requests from that pool of clients. When a program running on the client needs to resolve a hostname, a piece of code running on the client called the *resolver* contacts the local nameserver with a *recursive query* for that name (labelled *rq*) in Figure 1 below. This step and the following steps are numbered in consecutive order in the figure. Recall that a recursive query is named as such because it results in the local nameserver making a number of other queries (called *iterative*) on behalf of the recursive query. The local nameserver next makes a series of iterative queries (labelled *iq*) to various other nameservers, which may include a *root* nameserver (authoritative for `.`), a *top-level domain* (TLD) nameserver (authoritative for top-level domains, such as `.edu.`, `.com.`, etc.), and other authoritative nameservers for the domains and subdomains listed in the original recursive query. This sequence of iterative requests and responses, labelled 2–7 in the figure below, may be reduced by information present in the local nameserver's cache. Finally, once the local name server has resolved recursive query 1, it replies to the client's resolver with the *recursive response* labelled *rr* in the figure. The resolver then returns the response to the application running on the client.

This is the heart of the DNS nameserver functionality. You can think of the local nameserver as the "workhorse" of the DNS, because it is the one required to implement the recursive lookup algorithm. In fact, most TLD and root nameservers will not honour requests for recursive lookups—they are simply too busy.

This assignment is in two parts. In Part 1, we will understand how recursive queries work in practice by using command line tools and our knowledge about the DNS. In Part 2, we will implement the recursive query functionality in Ben's DNS server, testing against answers provided by the department's local DNS server. Note that the second part is substantially more challenging than the first—be sure to leave yourself enough time to finish the entire lab!

Part 1: Manual Recursive Queries

Let's begin by getting a deeper understanding of how DNS queries work in practice. We'll be using `dig`, a command-line UNIX tool that instructs the client's resolver to issues DNS queries to one of the local nameservers listed in `/etc/resolv.conf`, usually `128.16.6.8` (`haig.cs.ucl.ac.uk`) on department systems. `dig` then parses the resulting DNS reply, printing it clearly on the console. For example,

```
$ dig @haig.cs.ucl.ac.uk sipb.mit.edu
```

Results in output similar to the following:

```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 54529
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
sipb.mit.edu. IN A

;; ANSWER SECTION:
sipb.mit.edu. 18517 IN A 18.181.0.29

;; AUTHORITY SECTION:
mit.edu. 14006 IN NS BITSY.mit.edu.
mit.edu. 14006 IN NS STRAWB.mit.edu.
mit.edu. 14006 IN NS W20NS.mit.edu.

;; ADDITIONAL SECTION:
BITSY.mit.edu. 165011 IN A 18.72.0.3
STRAWB.mit.edu. 165011 IN A 18.71.0.151
W20NS.mit.edu. 165011 IN A 18.70.0.160

;; SERVER: 128.16.6.8#53(128.16.6.8)
```

From the metadata at the end of the output, we see that `dig` has received an answer from `haig` (`128.16.6.8`). First, the header section tells us firstly that the query was successful (`status: NOERROR`), listing the query's unique identifier, or UID: (`id: 54529`). Second, the header's one-bit flags tell us that this is a response (flag `qr` is set), that `dig` requested a recursive lookup (flag `rd`, for recursion desired, is set), and that `haig` honored the request for a *recursive* lookup (flag `ra`, for recursion available, is set). Finally, the header tells us to expect one question entry, one answer entry, three authority entries, and three additional entries in the body of the reply. The question entry (`sipb.mit.edu. IN A`) echoes the question entry that `dig` sent `haig`, querying the Internet (class `IN`) address (`A`) of `sipb.mit.edu`. The answer entry tells us that `sipb.mit.edu`'s IP address is `18.181.0.29`. The authority section lists three nameservers authoratative for `mit.edu.`, `BITSY.mit.edu.`, `STRAWB.mit.edu.`, and `W20NS.mit.edu.` The additional section contains three "glue" records specifying the IP addresses of the three authoritative nameservers listed in the authority section.

A key point to note here is that since `dig` requested recursion, `haig` has done the hard part for us. Let's now see how the process works from `haig`'s perspective by making the sequence of iterative queries that `haig` would, using the `+norecurse` flag.

First, let's find out the names of the root DNS servers:

```
$ dig @haig.cs.ucl.ac.uk . +norecurse
```

dig tells returns the following start-of-authority (SOA) record:

```
;; AUTHORITY SECTION:
.                        3966    IN      SOA     A.ROOT-SERVERS.NET.
           NSTLD.VERISIGN-GRS.COM. 2009081100 1800 900 604800 86400
```

The SOA tells us that a root nameserver is at A.ROOT-SERVERS.NET and is administered by nstld@verisign-grs.com. As we learned in lecture, there are many root nameservers. Let's ask haig for the address of one. We expect an answer here because local nameservers are usually configured with the (well-known) addresses of the root nameservers.

```
$ dig @haig.cs.ucl.ac.uk A.ROOT-SERVERS.NET. +norecurse
```

haig responds (in part):

```
;; ANSWER SECTION:
A.ROOT-SERVERS.NET.     595112  IN      A       198.41.0.4
```

Supposing that our address caches are empty, let's now use dig to make the same series of (non-recursive) requests that haig would make given a query for sipb.mit.edu. with the rd bit set. We start our hypothetical recursive query at A.ROOT-SERVERS.NET (198.41.0.4):

```
$ dig @198.41.0.4 sipb.mit.edu. +norecurse
```

A.ROOT-SERVERS.NET responds (in part):

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 63951
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 7, ADDITIONAL: 8

;; QUESTION SECTION:
;sipb.mit.edu. IN A

;; AUTHORITY SECTION:
edu. 172800 IN NS L.GTLD-SERVERS.NET.
edu. 172800 IN NS E.GTLD-SERVERS.NET.
edu. 172800 IN NS G.GTLD-SERVERS.NET.
edu. 172800 IN NS A.GTLD-SERVERS.NET.
edu. 172800 IN NS C.GTLD-SERVERS.NET.
edu. 172800 IN NS F.GTLD-SERVERS.NET.
edu. 172800 IN NS D.GTLD-SERVERS.NET.

;; ADDITIONAL SECTION:
A.GTLD-SERVERS.NET. 172800 IN A 192.5.6.30
A.GTLD-SERVERS.NET. 172800 IN AAAA 2001:503:a83e::2:30
C.GTLD-SERVERS.NET. 172800 IN A 192.26.92.30
D.GTLD-SERVERS.NET. 172800 IN A 192.31.80.30
E.GTLD-SERVERS.NET. 172800 IN A 192.12.94.30
F.GTLD-SERVERS.NET. 172800 IN A 192.35.51.30
G.GTLD-SERVERS.NET. 172800 IN A 192.42.93.30
L.GTLD-SERVERS.NET. 172800 IN A 192.41.162.30
```

The first thing to notice in the above is that we haven't received a direct answer to our query. A.ROOT-SERVERS.NET has instead observed that our query is for a host in the edu. top-level domain, and referred us to one of several top-level domain nameservers. It's even been kind enough to provide us with the aforementioned glue records that tell us where to continue our query. In the following "warmup exercises," you will continue this recursive query in a similar fashion.

Warmup exercises

1. Using dig with the +norecurse flag always set, continue the recursive query for sipb.mit.edu. on your machine. Turn in the full list of command lines that you issue as well as the resulting dig output. In this and the following exercises, you may elide irrelevant lines in the DIG output. [1 mark]

2. Again, using dig with the +norecurse flag always set, emulate recursive queries in the same manner for the following hostnames. Be sure to start at a root DNS server for each.
   sonic.cs.ucl.ac.uk.
   mistlab.lcs.mit.edu.
   www.microsoft.com.
   [1 mark]

3. For each of the three hostnames listed above, what about the name resolution process differed from the name resolution process for sipb.mit.edu.? [1 mark]

4. In the case of sonic.cs.ucl.ac.uk., why might two correct DNS implementations return differing results to two different runs of dig? [1 mark]

[Total marks for Part 1: 4]

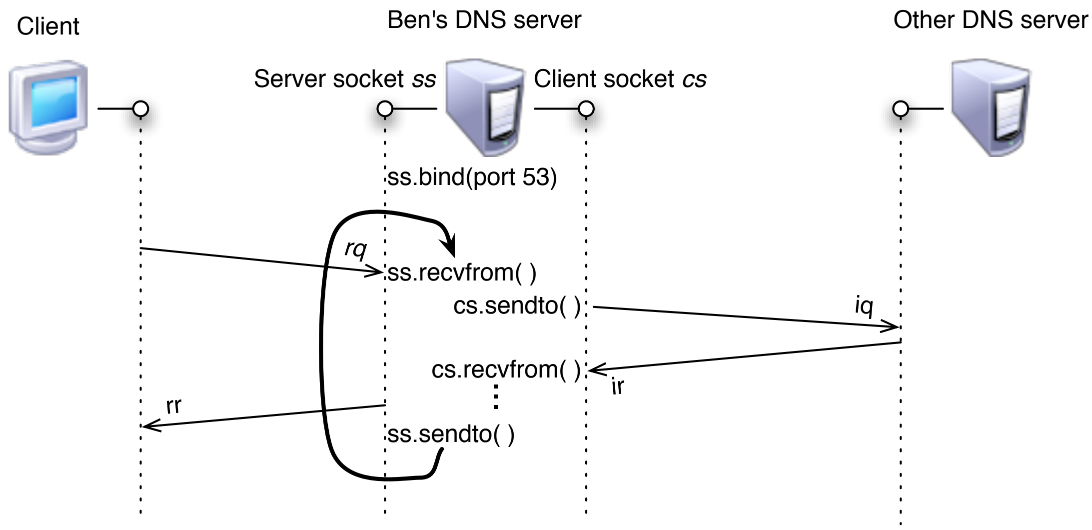

FIGURE 2: Communication flow from client to local nameserver and local nameserver to other DNS servers during one client's request to its local nameserver.

Part 2: Building a Simple Local Nameserver

Now that you are familiar with the process of making recursive queries by hand, you and Alyssa are ready to help Ben build his local nameserver. Ben decides that the nameserver will be a simple "single-threaded" server model that serves one client at a time, competing service of the current client before starting service of the next client.[1] The nameserver we will build will be a "recursive" nameserver, capable of answering the same recursive queries you resolved by hand in Part 1, and will be *caching*, retaining the results of prior queries until they are invalid, using cached data instead of iterative queries whenever possible.

Recall from lecture that normal DNS communication uses UDP, the unreliable datagram delivery service of the Internet, and that applications send and receive UDP datagrams using a software interface called a *socket*. As shown in the picture below, Ben's server will utilise two sockets: `ss`, a server socket bound to UDP port 53 (the well-known DNS port), and `cs`, a client socket that the server uses to query other servers as a client of those servers. These sockets are shown in the figure below, along with a time diagram of how communication proceeds in this single-threaded server. First, Ben's local nameserver uses the `bind` call to bind `ss` to UDP port 53. Then, serially, for each incoming client query, the server calls `recvfrom` on the server socket to wait for and receive the recursive query *rq* as shown in the figure. Next, the server makes zero or more iterative queries *iq* using `sendto` and `recvfrom` on the client socket, while processing the results. Finally, the server calls `sendto` on the server socket to send its response to the client.

The preceding steps execute in a continuous loop, as indicated in the figure, with `ss.recvfrom` blocking until the next incoming request.

Getting started

All programming for this coursework must be done under Linux on the department's lab machines. We have ensured that the code we give you to build upon works correctly on these lab machines. Note that these machines are accessible over the Internet, so you may work on the coursework either from home (using ssh to login to the machine at *machinename*`.cs.ucl.ac.uk`, or by logging into the machine in person, in lab.

[1] You will learn about more complex ways to design servers in a later NCS class.

The Linux lab machines are those with the following hostnames:

```
auerbach calder eyck fulla goya hals judd kubin lowry munch
nattier opie pollock quarton rubens shitao valdes whistler
zorach aquinas calvini carey choukri cocteau collins cowper
dahl dumas tanizaki okri steinbeck proust mahfouz kerouac
golding grass akutagawa alcott austen sands kahiga achebe
alkali bellow faulkner flaubert heine hesse bronte swift
eco hardy delhi darjeeling lucknow patna
```

To get started, copy the skeleton files to within your home directory:

```
$ cp ~jamieson/gz01_cw2_distribution.tgz .
$ gunzip gz01_cw2_distribution.tgz
$ tar xvpf gz01_cw2_distribution.tar
```

This will create a directory called `gz01_cw2` which we will refer to as the lab distribution directory.

Source code walkthrough

The main nameserver is located in file `ncsdns.py`, in which Alyssa has coded up the single-threaded structure described above. Your task is to implement the recursive lookup algorithm described above, as well as a caching functionality so that your server doesn't make any unnecessary iterative queries.

The first problem you encounter is the need to parse and construct the various DNS packet payload formats. In short order, Alyssa codes up the following functionality to perform much of the necessary work. She documents her code and places it in the HTML pages in the `html` subdirectory of the lab distribution directory.

> **Action:** To understand how you will be parsing and constructing DNS payloads, view each class documentation in your web browser by opening the HTML pages in the `html` subdirectory of the lab distribution as they are discussed below.

In package `gz01.inetlib`, Alyssa provides you with two helper classes. `DomainName` is a class that represents DNS domain names and contains logic to parse and construct packet payload representations. `InetAddr` provides the same functionality for an IPv4 address.

Parsing and constructing DNS packets

The Python `recvfrom()` library function yields packet payloads as **strings of binary-valued data**, while the python `sendto()` library function expects packet payloads in the same format. We now describe how you may handle this requirement.

Every DNS packet payload is made up of a header, followed by zero or more *question entries*, followed by zero or more *resource records*. For more information about DNS packet formats, see the class lecture notes and the references at the end of this assignment.

Domain names in responses from production DNS servers are often compressed according to RFC1035. Rather than requiring you to implement the string compression and decompression algorithms, package `gz01.inetlib` contains a `DomainName` class that implements the decompression algorithm for you. The tricky part, however, is the semantics of how long a domain name is. Suppose your server receives a packet with a domain name inside whose compressed binary representation is 10 bytes long, but whose string length is 13 bytes long. When you construct a `DomainName` object `d` from that compressed binary representation, should `len(d)` evaluate to 10 or 13? The approach we have chosen is to have `len(d)` evaluate to its compressed length, in this case 10, but any copies of `d` that are made by the Python call `copy(d)` get uncompressed and hence have a length of 13. This issue comes up in resource record handling; see below.

The following Python classes in package `gz01.dnslib` parse and construct DNS protocol payloads using the classes in `gz01.inetlib` discussed above.[2]

- `Header` represents a DNS protocol payload header; its constructor creates a new `Header` object from user-specified arguments, while its `fromData` method returns a new `Header` object from user-supplied binary string data.
- `QE` represents the question entry that is usually the first entry after the header. Like `Header`, `QE` has a constructor that creates a `QE` from individual arguments, and a `fromData` method that returns a `QE` object from binary string data.
- `RR` is an abstract class that represents a generic resource record, and contains logic common to the various types of resource records. All resource records contain a domain name, a time-to-live, and a field specifying the length of the data they contain.

  `RR`'s `fromData` method returns a pair `(rr, length)` where `rr` is one of the classes below derived from `RR`, which represent resource records of various types, and `length` is the length in bytes of the resource record returned, in the binary string representation supplied to the user's call to `fromData`. Any `DomainName` objects that `RR` or its derivatives contain are copies of the originals, and so have a length commensurate with their string representation length. See above for a discussion of string representation `DomainName` length versus binary representation `DomainName` length. Thus, as a user of the `RR` classes, you will need to refer to the second element of the pair that `fromData` returns in order to find out how many bytes a given `RR` takes up in the binary string you give `fromData`.

  The classes derived from `RR` are:

  – `RR_A` represents an "address" resource record, which contains an IPv4 address.
  – `RR_CNAME` represents a canonical name address record, which contains a canonical hostname (a `DomainName`).
  – `RR_NS` represents a nameserver resource record, which contains the domain name of another nameserver (a `DomainName`).
  – `RR_SOA` represents a start-of-authority resource record, which contains information about a domain.

Debugging and logging

Alyssa provides a `hexdump` function in package `gz01.util` to assist you in debugging the contents of packets received or sent from or to other DNS servers. When called with one argument whose value is a packed binary string containing any data, `hexdump` will list the data in human-readable hexadecimal and ASCII form, in two columns, with byte indices in another column to the left. The hexadecimal display is one byte (two hexadecimal characters) at a time. By judicious use of `hexdump` on each packet that you send or receive on the network, combined with cross-referencing the DNS specifications, you can resolve many issues.

A sample invocation of `hexdump` in the context of the provided code is:

```
while 1:
  (data, address,) = ss.recvfrom(512) # DNS limits UDP msgs to 512 bytes
  if not data:
    log.error("client provided no data")
    continue
```

[2] Note that Python packages reside in subdirectories corresponding to the package name, so the directory path to Alyssa's DNS data format handling code is `gz01/dnslib` in the lab distribution directory.

```
  else:
    print "Query received from client is:\n", hexdump(data)
    queryheader = Header.fromData(data)
    print "Query header received from client is:\n", hexdump(queryheader.pack())
```

The resulting output visible on the console when the server runs, answering a query for `www.cs.ucl.ac.uk` is:

```
Query received from client is:
0000   38 8E 01 00 00 01 00 00 00 00 00 00 03 77 77 77    8............www
0010   02 63 73 03 75 63 6C 02 61 63 02 75 6B 00 00 01    .cs.ucl.ac.uk...
0020   00 01                                              ..

Query header received from client is:
0000   38 8E 01 00 00 01 00 00 00 00 00 00               8...........
```

> **Action:** To familiarize yourself with DNS payloads (and thus prepare for debugging the results when you begin to construct your own), reconcile the above hexdump with the DNS packet format, understanding where each field's value is in the hexdump.

Alyssa has also provided a logging infrastructure to assist in the debugging of your server. Package `gz01.util` sets up logging to the console and file `ncsdns.log`, rotating that log file and its predecessors to files `ncsdns.log.1`, `ncsdns.log.2`, `ncsdns.log.3`, etc., all in the directory from which you run your server.[3] Then to debug your server, make calls from `ncsdns.py` to the `logger` object with debugging messages you find useful. Each logging call takes two arguments: an importance level, and a string to output to the log. The importance levels and the corresponding call to make in your code are as follows:

Level	Numeric value	How to call
CRITICAL	50	`logger.critical(string)`
ERROR	40	`logger.error(string)`
WARNING	30	`logger.warning(string)`
INFO	20	`logger.info(string)`
DEBUG	10	`logger.debug(string)`
DEBUG1	9	`logger.log(DEBUG1, string)`
DEBUG2	8	`logger.log(DEBUG2, string)`

Then in file `gz01/util.py` you can configure calls to `setLevel` as directed by the comments in order to alter the verbosity level of the debugging output directed to either the console or the log file. When you select a given verbosity level x you will then see only logging calls tagged with a level whose numeric value is x or greater.

Caching

One of the requirements in the next section is that your server must answer queries from an internal *cache* whenever possible, instead of querying another nameserver. Since each resource record that your nameserver receives as a result of making a query on another nameserver contains a *time to live* field (as described in lecture), upon receiving each query, your nameserver can deduce which resource records are still valid, and use the valid records to answer the query.

Requirements

To earn points on this coursework, your nameserver must pass various tests generated by our testing suite. To help you pass the testing suite, we provide you with a test driver, described in the next section, which runs exactly the same tests as our testing suite.

[3]Note that the rotation happens when `ncsdns` exits, so to read the most recent log, check `ncsdns.log.1`.

In addition, you must return answers according to the following requirements, for **every query** that your nameserver responds to. **Failure to meet any of these requirements for *any* of the queries we make to your nameserver when we evaluate it will result in a loss of one point per unmet requirement.**

1. Your nameserver must run with unchanged versions of our library files.
2. Your nameserver must answer queries without entering an infinite loop and hanging, or terminating.
3. Upon answering a query, your nameserver must wait for a subsequent query, and answer it in turn.
4. Your nameserver must follow all canonical name `CNAME` aliases for a query and return the correct address records for the canonical host name in the query.
5. Your nameserver must return authority records of nameservers in the most highly-qualified subdomain of the canonical host name.
6. Your nameserver must return the correct glue records for nameservers it mentions in the authority section of its reply.
7. When possible, your nameserver must use the cache to answer a given query, without generating any network traffic.
8. Your nameserver must cope with not receiving a reply to requests it sends, for any reason (failure of *other* DNS servers, network failures, timeouts due to network congestion, for example).

Negative requirements. Your nameserver does not need to implement the following functions that production DNS servers provide:

1. Your nameserver does not need to respond to queries containing multiple questions (multiple hostnames to resolve, for example).
2. Your nameserver does not need to respond to queries of type other than `A` (address).
3. You are not required to return time-to-live fields that take into account network propagation delay.

Testing your nameserver

We have installed Python 2.6 for both Intel 32-bit and 64-bit architectures on the lab machines. To make sure you use the correct architecture, always use the `python-wrapper` program instead of the `python` command when you work on this assignment.

To run and test the server, use the following command:

```
$ ./python-wrapper ./ncsdns.py
```

The server responds:

```
./ncsdns.py: listening on port 40229
```

indicating that it is listening on port 40229.

To workaround the need for "root" access to the machine we are using, our server socket `ss` listens on a high-numbered "ephemeral" port instead of port 53. The server prints the port it is listening on to the standard output when it starts, as above.

To pass the test suite, your nameserver *must not* print anything on the console (standard output) *before* the port message.

For testing purposes, you should send `dig` queries to the port your server is listening on using the `-p [port number]` option. In a different window, use `dig` to query your local nameserver:

```
$ dig @127.0.0.1 -p [port number] [query hostname]
```

To test the server in this example, you would type:

```
$ dig @127.0.0.1 -p 40229 sipb.mit.edu.
```

`dig` should time out, not having received a response from your server.

Once your server responds to queries, use `dig` as in the first part of the warmup as follows:

```
$ dig @haig.cs.ucl.ac.uk [query hostname]
```

In this way you can compare the responses your server returns with the responses `haig` (or another local nameserver in the department) returns; they should be nearly identical. Try testing using the following host names, referring to the log as you test for insights as to what the possible bugs in your server are.

```
haig.cs.ucl.ac.uk.
sipb.mit.edu.
lxr.linux.no.
www.caltech.edu.
www.microsoft.com.
www.yahoo.com.
```

Once your name server starts to respond more or less correctly to these `dig` queries, run the `test-dns.py` script (located in your lab distribution directory to exhaustively test your server:

```
$ ./python-wrapper ./test-dns.py ./ncsdns.py
```

These are the tests that we will use to mark your server. For each of the above hostnames, `test-dns.py` runs one or more of the following sub-tests:

1. *Canonical hostname*: follow any CNAME records that map the queried hostname (or recursively, another CNAME) to a target "canonical" hostname. Compare the resulting canonical hostname returned by the test nameserver with the canonical hostname returned by the department's local nameserver. The test fails if they differ.

2. *Address resolution*: given the canonical hostname from the previous step, compare the IP addresses in A records returned by the test nameserver with the corresponding set returned by the department's local nameserver. The test fails if they differ.

3. *Auth domains*: compare the authoritative domains listed in the test nameserver reply's "AUTHORITY" section with those returned in the corresponding section by the department's local nameserver. The test fails if they differ.

4. *Auth domain ns*: compare the nameservers listed in the test nameserver reply's "AUTHORITY" section with those returned in the corresponding section by the department's local nameserver. The test fails if they differ.

5. *Glue records*: compare the nameservers listed in the test nameserver reply's "ADDITIONAL" section with those returned in the corresponding section by the department's local nameserver. The test fails if they differ.

6. *Glue rec addrs*: compare the IP addresses listed in the glue records of the test nameserver reply's "ADDITIONAL" section with those returned in the corresponding section by the department's local nameserver. The test fails if they differ.

Note that some of the above tests are disabled for some domains that use DNS load balancing to return different results in round-robin order or DNS-based content distribution networks that change results based on client identity. The test for a given hostname passes if all of the enabled subtests pass, as shown in this example run for a fully-functioning nameserver:

```
Testing ./ncsdns.py:
server is listening on port 47781
Begin test sipb.mit.edu.: canonical hostname...ok (sipb.mit.edu.).
  address resolution...ok.  auth domains...ok.  auth domain ns (mit.edu.)...ok.
  glue records...ok.  glue rec addrs (bitsy.mit.edu.)...ok.
  glue rec addrs (w20ns.mit.edu.)...ok.  glue rec addrs (strawb.mit.edu.)...ok.
PASS.
```

After the above hosts are tested, `test-dns.py` begins monitoring your server using the `strace` command to determine which other hosts your server queries, if any. `test-dns.py` then runs through the above tests again, verifying first that your server executes them correctly again, and second, that your server answers queries from the cache whenever possible. The cache test fails if your server queries any other nameserver while answering these queries.

The `test-dns.py` script creates the following log files to assist you if and when tests fail:

- `ncsdns.stdout` the "standard out" of your DNS server as it runs in the testing script.
- `ncsdns.stderr` the "standard error" of your DNS server as it runs in the testing script.
- `baseline.log`: the output of each `dig` query as it runs on the "baseline" server—CS's departmental `dns` nameserver.
- `test.log`: the output of each `dig` query as it runs on your `dns` server, the "test" server.

Action: when tests fail is to examine the latter two files, `test.log` and `baseline.log`, to compare the answers that the test and baseline `dns` servers returned, respectively.

[Total marks for Part 2: 8]

Hand in instructions

You will use the **handin** program on the UNIX CS machines to submit your work. Note that the module name you should use to hand in your work is `gz01` regardless of whether you are registered for GZ01 or 3035. You may run handin multiple times; we will evaluate the last submission you make.

To use the `handin` program:

1. `cd` to the directory containing the files to be submitted.
2. Run the `handin` program.
   - Module name: `gz01`
   - Coursework name: `cw2`

Submission instructions for Part 1

Hand in Part 1 of this coursework by cutting and pasting your work into a plain ASCII text file named `cw2-part1.txt` and submitting only this text file using `handin`. In accordance with class policy, please state how many late days you would like to take in this file.

Submission instructions for Part 2

Hand in only the Python source code necessary to run your DNS server, excluding library files, which should be left unchanged per the requirements above (usually this is just the file `ncsdns.py`). To meet size restrictions, remove all precompiled Python intermediate files (those files ending in `*.pyc` in your Coursework 2 directory tree).

Your work in Part 2 will be tested by our automated testing scripts, which exercise the same functionality of your DNS server as `test-dns.py`.

[Total marks for this coursework: 12]

The teaching staff is not responsible for coursework submissions not made in accordance with the above instructions—in particular, we do not accept coursework submissions via email.

References

The following references on the Domain Name System may be useful in completing this assignment:

1. RFC1034: Domain names—Concepts and facilities (web)
2. RFC1035: Domain names—Implementation and specification (web)
3. RFC1536: Common DNS implementation errors and suggested fixes (web)
4. Python documentation
   `http://docs.python.org`
   `http://docs.python.org/tutorial/index.html`
5. The Python library reference is available at:
   `http://docs.python.org/library/index.html`

Mailing lists

Please monitor your Moodle-registered email address during the period between now and the due date for the coursework. Any announcements (e.g., helpful tips on how to work around unexpected problems encountered by others) will be sent to the class via email.

Academic honesty

This coursework is an individual coursework. You are to complete it alone; you may not consult with other students in the course (or with other people who are not taking the course) about what algorithm to use or how to write your code. You may not look at other students' code (from this year or past years), nor show them yours. You are free to read reference materials found on the Internet (and any other reference materials). You may of course use the code we have given you. All other code you submit must be written by you alone. Copying of code from student to student is a serious infraction; it will result in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities (normally resulting in exclusion from all further examinations at UCL). The course staff use extremely accurate plagiarism detection software to compare code submitted by all students and identify instances of copying of code; this software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!