Project 2: Dynamic DNS, Part A

October 28, 2008

Due Nov 12 @ 11:59:59 pm

EE122: Introduction to Communication Networks (Fall 2008)
Department of Electrical Engineering and Computer Science
College of Engineering
University of California, Berkeley

1 Overview

In this project, you will implement a DNS server system. The project has two parts:

- Part A: Dynamic Domain Name System (DNS) server and client.
- Part B: Peer-to-peer gossiping with the other DNS servers to learn DNS records.

This document describes Part A.

Your DNS server will implement a limited version of the real DNS protocol. Your server will interoperate with the live Internet DNS system. For example:

- You will be able to query your server with standard commands like nslookup and dig.
- We will give you your own publicly-accessible subdomain to be served by your server. Within this subdomain you will be able to create your own names that anyone on the Internet can use.
- If you’d like, and if you have your own server on which to run it, you can even purchase your own domain name and use your server to serve it. (Surprisingly, ee122madeallmymedianscometrue.com is unclaimed as of this writing.)

Standard old DNS servers like named on Unix learn their name-to-IP address mappings by reading a configuration file. Your server, however, will be much, much slicker. Specifically, it will accept mapping messages over the network from a dynamic DNS client. You can run the client on your laptop, so when you move around from one wireless network to another and thus change IP addresses, your laptop will always be available at the same DNS name. This is similar to services such as DynDNS.com.

Key concepts in Part A of the project include:

- Working with a real protocol.
- Use of UDP.
- Use of soft state.
2 Command line interfaces

Server
Your server should take one command line argument: a port to which to bind, accepting incoming UDP DNS queries. For example we should be able to run it on port 6001 like this:

```
./server 6001
```

It should then keep running, waiting for and replying to queries, until it is killed manually. Your server doesn’t need to print anything out to the console, although this will help you in debugging.

Recall that the DNS protocol expects servers to run on port 53. But only root can bind to ports 1023 or less. So to run on the instructional machines, you’ll have to use a port number of 1024 or greater.

It goes without saying that your server should not crash regardless of what it receives over the network, and should not have memory leaks.

Client
The dynamic DNS client should accept four arguments: a server IP, server port, a domain name, a corresponding IP address, and a TTL. For example:

```
./client 128.32.48.169 6001 mylaptop.fakedomain.com 128.2.1.2 120
```

The client should then contact the server at the given IP (here 128.32.48.169) and port (6001) using the protocols described in Section 3 below. It should tell the server to install the given mapping (mylaptop.fakedomain.com → 128.2.1.2) for the given amount of time (120 seconds).

If this all works, you should be able to query the server within the next two minutes...

```
nslookup mylaptop.fakedomain.com 128.32.48.169 -port=6001
```

...and get a reply showing the IP address 128.2.1.2.

3 Network protocol and semantics

Having specified the interface between humans and your programs in the previous section, we now specify the interface between the programs themselves across the network.

Your server will accept and reply to DNS Question messages on UDP. The client will install mappings in the server by sending a DNS Response message to the server. Hence, it all comes down to the DNS protocol. Next we give a detailed description of the DNS protocol, followed by how it should be used by your client and server.

The DNS protocol
A DNS datagram is formatted as follows:

- **[16 bits] IDENTIFICATION.** Set by the client when it sends a query, the server should keep this the same in its reply, allowing the client to match up the response with the request.

- **[16 bits] PARAMETER.** An assortment of bits annotating the query or response, indicating error conditions, and so on. As a simplification of the protocol, you don’t need to deal with most of these. Handle the bits as follows:
  - Bit 0 is set to 0 in queries, and 1 in responses. When your server gets a name lookup query, this bit will be set to 0; its reply should set the bit to 1. To install a name mapping your dynamic client should send a message to the server with this bit set to 1.
– Bit 5 is set to 1 if the message is a response and the answer is authoritative; otherwise the bit is set to 0. Your server should act as the authoritative name server for all names installed in it by your dynamic client.
– All other bits in the parameter field should be ignored in messages you receive, and should be set to 0 in messages that you send.

• **[16 bits × 4] Number of questions, Number of answers, Number of authority, Number of additional** (in that order). These integers specify the number of entries in the corresponding four sections which follow. As a simplification of the protocol, you may assume that the Number of questions and Number of answers are are either 0 or 1; and the Number of authority and Number of additional are always 0.

• **[Variable length] Question section.** This section contains zero or more questions, the number of which is specified above. You’ll need to deal only with zero or one questions. Each question has the following format:

  – **[Variable length] Domain name.** This is the name that the querier wishes to be translated to an IP address. It is not just a sequence of characters; instead it uses a representation that can compresses the size if there are many names similar names mentioned in this datagram. Good news: we will give you code to translate from this format to a standard string, and vice versa. If you want, you can also implement this yourself.

  – **[16 bits] Type.** This integer can take any one of dozens of values specifying the type of query, including some corresponding to types we discussed in lecture like A, CNAME, and MX. As a simplification of the protocol, your server can assume that the query type is always A, in which case this field has the value 1 (not 4 as previously advertised).

  – **[16 bits] Class.** This integer describes the type of address requested in the reply. For the purposes of this project, this value should always be set to 1, which corresponds to the “Internet class”. (Although several other classes are defined, they are not used in practice.)

• **[Variable length] Answer section.** This section contains zero or more answers, the number of which is specified above. You’ll need to deal only with zero or one answers. Each answer has the following format:

  – **[Variable length] Domain name.** This is the name which is being mapped to an IP address. The format is the same as in the Question section.

  – **[16 bits] Type.** As in the Question section.

  – **[16 bits] Class.** As in the Question section.

  – **[32 bits] Time to live.** This integer specifies the number of remaining seconds before the mapping should expire.

  – **[16 bits] Resource data length.** An integer specifying the number of bytes in the Resource data field.

  – **[Variable length] Resource data.** The contents of this field depends on the record’s type, but for the A records which your server and client will deal with, it is a (4-byte) IP address.

• **[Variable length] Authority section.** Responses may contain zero or more resource records in this section, but your server may assume that there are always zero entries here.

• **[Variable length] Additional information section.** Responses may contain zero or more resource records in this section, but your server may assume that there are always zero entries here.
Using the protocol: client

- When your client is run, it should generate a DNS Response datagram containing a Type A record in the ANSWER section, mapping the specified name to IP address with the given TTL. It should send this datagram to the server.

- Since UDP is unreliable, your client should check to make sure the server got the message. It should send a query to the server for the name that it just installed, and wait for a reply. If the reply is received within 3 seconds and matches the name → IP mapping that the client originally sent, then the client has succeeded and should exit.

- If after 3 seconds the client has received no response or an incorrect response, it should retry the entire process described above (send Response, send Query, wait for answer) up to a maximum of 3 times total. After the third unsuccessful round—that is, a total of up to 9 seconds—your client should exit and print the following error message:

  Error: could not install mapping in server

Using the protocol: server

- When your server receives a Response message with name $n$, IP address $i$, and TTL $t$, it should install the mapping $n \rightarrow i$ in its database. If a mapping for name $n$ existed previously, it should be removed or overwritten. After $t$ seconds, the new mapping should be considered expired and can be removed from the database.

- When your server receives a Query message for a name, it looks for a match of the name in its local database. Name matching is case insensitive but otherwise the names must match exactly.
  - If it finds a match which is not expired, it should send a Response message back to the querier containing (1) the original query in the QUESTION section, and (2) a type A record with the name → IP mapping and remaining TTL in the ANSWER section.
  - Otherwise, it should send a Response message back containing the original query in the QUESTION section (and nothing in the ANSWER section).

- Whenever your server receives a problematic message, it should simply ignore it, without sending any message back. This includes receipt of malformed messages, and messages which do not conform to the simplifications of the protocol described above.

- Your server should be robust to malicious activity. No message sent to your server, regardless of how long or badly formatted it is, should cause your server to crash. The memory used by your server should not grow indefinitely due to storing expired entries or due to memory leaks.

4 Suggestions

Getting started

We suggest the following as a possible implementation plan:

1. Write the server code to bind to the UDP port and read incoming DNS messages. Print them out so that you can tell it looks sensible. Test with nslookup or dig.

2. Write the server code to send DNS answer messages.

3. Write the client to install mappings.

Beej’s guide, which we mentioned in Project 1, also includes information on UDP (datagram) socket programming:

http://www.beej.us/guide/bgnet/
Real domain names

This entire project can be completed using only “fake” DNS names. But if you’d like, you can also use your server and client with real publicly-accessible DNS names.

- **Buy your own name.** Numerous companies online will sell you a domain name for in the neighborhood of $10 per year. The company will let you enter your DNS server IP address, which will then get installed in the Internet’s TLD servers. The only complication is that the servers must be running on port 53. So if you want to buy your own domain name and serve it with your server, you’ll need to run your server on a machine on which you have root access.

- **Use our names.** We will provide a subdomain which you will be able to serve from your DNS server. This will circumvent the need for you to purchase the domain, and also the need for you to run on port 53: our server will recursively query yours for names in your subdomain. We will release specific instructions for using these names at a later date.

5 Submission and grading

Your submission should be a directory containing:

- The source code of your project (no executables).
- A Makefile, so that running `make` compiles the server executable (`server`) and a client executable (`client`).
- A brief README explaining the design of your server and client.

Submit your project on the instructional machines using the command `submit proj2checkpoint`.

The submitted project needs to compile and run on the UNIX instructional machines. Both the client and server should be written in C or C++.

The grade breakdown will be as follows:

- 55%: server passes our test cases.
- 5%: server doesn’t have memory leaks (Valgrind is a good tool to check for leaks).
- 40%: client passes our test cases.