COSC 465: Computer Networking
Lab 1: C++ warmup: a DNS cache
Due: 8 February 2012, before next lab

1 Overview

The first lab is designed as a warm-up exercise for C++, and as an introduction to using some parts of the standard sockets networking API.

The Domain Name System (DNS) is one of the most important components of the internet. It is, in essence, a distributed database that maps IP addresses to their corresponding "fully-qualified domain names" (FQDNs), and vice versa [1]. The DNS makes it possible for people to use easily-remembered names like `cs.colgate.edu` rather than having to memorize an IP address like `149.43.80.13`. As we'll learn soon, almost all internet applications and lower-layer protocols *only* work with IP addresses. Thus, the DNS is queried very often in order to provide a translation between human-centric domain names, and computer-centric IP addresses.

In order to reduce request load to the DNS and improve performance, the responses to DNS queries are typically *cached*. Caching is used extensively in the DNS, including intermediate DNS servers and inside end-host operating systems. Applications also sometimes perform DNS caching, and that's where this lab fits. We're going to create a C++ class that mimics some aspects of a standard DNS cache.

For this lab, you should each submit your own code, but you are welcome to work with one another. Please make a comment in your code who you worked with.

2 Detailed Description

Your class should be named `DnsCache` and must provide the following public methods:

`bool lookup(const std::string &s, struct in_addr &ip);`	Look up the IP address corresponding to the domain name referred to by the string `s`. The IP address should be stored in the reference variable `ip`. The method should return `true` on success and `false` otherwise. Note that there may be multiple IP addresses that are mapped from the same host name. Your cache only needs to consider *one* of them. You should also only consider IPv4 addresses (not IPv6; see below for some notes on this issue).
`void invalidate(const struct in_addr &ip);`	Invalidate (remove) any cached entries that are related to the IP address `ip`. The method shouldn't return anything.
`void invalidate(const std::string &s);`	Invalidate (remove) any cached entries that are related to the domain name referred to by `s`. The method shouldn't return anything.
`double average_lookup();`	Return a double containing the average measured time to perform any type of DNS lookup, in seconds. The returned value should be computed over all lookup requests, and consider both entries that are found in cache, and entries that must be obtained through a lookup to the DNS. (It should also include time taken for both "failed" and good lookups, too.) If no lookups have been done and this method is called, it can just return 0.
`void dump_cache();`	Print the entire contents of your cache to standard output. You can choose any reasonable format for the output.

[1] The DNS actually has quite a bit more information than mappings between IP addresses and names. We'll look at DNS in detail toward the end of the semester.

For any lookup, your class should first consult an internal cache, which may be structured in any way you choose (some suggestions are listed below). If the lookup can be satisfied from the cache, no "external" request to the DNS needs to be made. If your `DnsCache` does not have the appropriate information cached, you'll need to perform a network DNS query.

Put your class definition in a file named `dnscache.h` and your class implementation in `dnscache.cc`. A `SConstruct` file and a test program that uses your `DnsCache` class (a file named `lab01.cc`) are posted on Moodle.

2.1 Queries to the DNS

To make a network query to the DNS to look up the IP address corresponding to a fully-qualified domain name, you can use the `getaddrinfo` call. An example of this call is given below.

The function `getaddrinfo` requires the use of two C structs: `struct sockaddr_in` and `struct in_addr`. The `struct sockaddr_in` is used to specify the endpoint address for a *socket*, which is the primary endpoint abstraction used in the sockets API. It consists of three fields: `sin_family`, `sin_port`, and `sin_addr`. For this lab, you can always set `sin_family` to `AF_INET`, and you should always set `sin_port` to 0. (In later labs, `sin_port` will almost never be set to 0.) The `sin_addr` field of `struct sockaddr_in` is itself of type `struct in_addr`, and `struct in_addr` is used to specify a specific IP version 4 address. This structure has just one field, `s_addr`, and is simply a 32-bit unsigned integer.

You may also find the following two utility functions useful: `inet_pton` can be used to convert a "presentation" formatted IP address to a number, and `inet_ntop` can be used to do the reverse. These functions basically allow one to easily convert the C string `"192.168.1.1"` to a 32-bit unsigned integer that represents the equivalent IP address (using `inet_pton`), and to convert a 32-bit unsigned integer IP address to a C (NULL-terminated) string (`inet_ntop`).

An example of performing a DNS query to obtain an IPv4 address corresponding to a particular host name is as follows:

```
// Note: the following code uses "hints" to getaddrinfo to specifically
// ask for just IPv4 addresses.  It also only grabs the first IP address
// returned, and ignores any other additional addresses returned.

std::string hostname = "cs.colgate.edu";
struct in_addr ipaddr;
struct addrinfo *result = 0;       // ptr to lookup results
struct addrinfo hints;             // tell getnameinfo a bit about what we're looking for
memset(&hints, 0, sizeof(hints));  // zero out the hints structure
hints.ai_family = AF_INET;         // we're only interested in IPv4 addrs
                                   // (FYI: AF_INET6 is for IPv6)

int error = getaddrinfo(hostname.c_str(), 0, &hints, &result);
if (error == 0) {
    // use the first IP address obtained from the function call

    struct sockaddr_in *sin = (struct sockaddr_in*)(result->ai_addr);

    // put IP address in our ipaddr struct
    memcpy(&ipaddr, &sin->sin_addr, sizeof(ipaddr));

    // free up the memory consumed by DNS results
    freeaddrinfo(result);

    char ipstr[32];
    inet_ntop(AF_INET, &ipaddr, ipstr, 32);
    std::cout << hostname << " has IP address " << ipstr << std::endl;
} else {
    std::cout << "Epic fail in DNS lookup." << std::endl;
}
```

(Note: you'll need to look up the `man` pages for these calls to ensure that you're including the correct header files, and to read up on all the ins and outs of these functions.)

2.2 Cache structure

Internally, your class should store any domain name to IP address mappings that you obtain from the DNS. When the `lookup` method is called, you should consult your cache to see whether the request can be satisfied without consulting the DNS (and calling `getaddrinfo`). Basically, you should avoid making calls out to DNS if you already have the information in your cache.

You can structure your cache any way you want, but here are a few suggestions:

1. At one end of the spectrum, you could create a simple C array-based hash table. You could hash the domain names to a particular array location. At each array location, you could keep a linked list of structures that each stores name and IP address information that hashes to that particular location.

2. You could utilize the STL `std::vector` class, that provides a simple array-like interface. You could create a struct (or class!) that contains a domain name/IP address pair, and store these objects in the vector. You'd have to perform linear search to check whether an item is in your cache or not. Potentially slow, but this is perhaps the easiest approach of the three suggestions here, and would get your feet wet using a (fairly simple) STL structure.

3. You could utilize the STL `std::map` class, which provides a key-value type store. Maps are little more cumbersome to use than simple STL vectors, but with a little extra work, this route would provide a fairly natural lookup mechanism, and much faster than a vector.

4. Some other type of structure of your choosing. Anything goes, as long as you correctly cache domain name / IP address mappings. For example, you could use a combination of hashing that uses a simple C-based, but where each array location refers to an STL vector of all the name/IP address pairs that hash to that location.

For documentation and some examples on using the STL classes, see `http://en.cppreference.com/w/cpp` (among other web references).

2.3 Other useful system calls

For computing and returning the average time taken to perform a lookup, you'll need some way to get the current OS time, at a fairly fine granularity. The standard `gettimeofday` system call is probably the best one to use. For example:

```
struct timeval begin, end, diff;
gettimeofday(&begin, 0);

// do some stuff

gettimeofday(&end, 0);

// Note: timersub is a macro defined in <sys/time.h>
timersub(&end, &begin, &diff);
```

3 Submission

Just submit your `DnsCache.cc` and `DnsCache.h` files to Moodle.

Note: If you've got gobs of spare time on your hands, there are 2 bonus points available for the fastest cache implementation. I'll use a test program that works your cache over a little more intensely than the supplied `lab01.cc` file. Caveat: if there are any ties (within 10 microseconds) for the fastest implementation, no bonus points will be awarded.