Lab 1.1 : Host Resolution

Getting Started – General Instructions

This is the first assignment, where you will resolve a host name into an IPv4 address. It serves as an short (and hopefully easy) introduction to network programming using the BSD socket API in C/C++.

You should work on this assignment in groups of two people. Try to find a lab partner that is approximately on the same skill level as you are with respect to programming. Start working on this as soon as you are able.

The assignment includes some source code, which you will extend. If you want to write the program from scratch you may of course do so (but use the `getaddrinfo()` method as described below). Either way, in the very end a reference solution is included. Primarily, you should try to complete the lab without looking at the reference solution. If you get stuck somewhere, you can obviously refer to the solution – but avoid just copying it![1]

The lab mainly targets Linux machines, although the code ought to work on MacOS X machines without modifications. If you have a Windows machine, or you want to use one of the lab computers, there are two options available: 1) Use a virtual Linux environment with VirtualBox, that is already installed in the lab computers. You will have to download a virtual image of your choice (e.g. the one we provide in the lab pages) 2) Remotely connect to a remote Linux server. Refer to the appendices for some additional information.

You don't have to hand in anything at the end of this lab, but **please check the Lab 1 page on Canvas and fill in the *Quiz*!**

To get the source code for the included programs, use either the provided link or grab the embedded file. (In e.g. some versions of Acrobat Reader, right-click on the text-file icon at the beginning of a section and select *"Save Embedded File to Disk"* to get the source code from the PDF file.)

Part I Resolving a host by name

For our first program we set the modest goal of turning a host/domain name, such as *xkcd.com* or *remote5.student.chalmers.se*, into an IP address. So, when we're finished, we want to be able to do the following with our program:

```
$ ./lab1-resolve-host xkcd.com
Resolving 'xkcd.com' from 'mieli':
IPv4: 107.6.106.82
```

In the output, we can see the dotted-notation IPv4 address printed as a string. While this is useful for humans, in the program's memory we use a different representation. The standard API provides several structures (C-**`struct`**s) that we have to use for this purpose.

► I.a A first look at the code

Grab the code from Canvas. Compile and run it. You should see something similar to the following:

```
$ ./lab1-resolve-host xkcd.com
Resolving 'xkcd.com' from 'mieli':
```

As you can see, the critical part (the IPv4 address) is missing. We'll be adding that in a moment.

[1] At least type the code by yourself. That way it will have passed through your brain at least once!

What you can see, however, is the name of your computer. (In the examples the name is 'mieli', which is the name of the computer I was logged in on when writing this document. Obviously your output may be different, depending on the name of your computer!) Look in the code – there name of the computer is retrieved using a function called `gethostname()`. If you're not already familiar with this function, check its *man-page*[2] (to view a man-page from the terminal, type `man foo`, where you replace `foo` with the keyword you're interested in):

```
$ man gethostname

GETHOSTNAME(2)              Linux Programmer's Manual            GETHOSTNAME(2)



NAME
       gethostname, sethostname - get/set hostname

SYNOPSIS
       #include <unistd.h>

       int gethostname(char *name, size_t len);

...
```

The man-page (or other equivalent documentation) will generally tell you

- Where the function is declared (here: `<unistd.h>`)
- How to invoke the function (arguments, return type/value, ...)
- What the function does (see section *DESCRIPTION*)
- What errors can occur

We'll have to be especially careful with the final item: ignoring potential errors is never a good idea! It's even worse in network programming, since improperly handled errors may in the end introduce bugs that might compromise security and might give evil people access to our machine remotely.

Handling errors properly is difficult. In the programs presented here, we'll generally print an error message and then exit. This is, of course, not always a good idea. Imagine a server handling multiple clients. We don't want to shut down the whole server just because a single client had an network error – this would deny service to all the other clients!

Exercise I.a.1 On my machine, the man-page for `gethostname()` lists four errors:

(a) `EFAULT` name is an invalid address
(b) `EINVAL` len is negative or, for **sethostname()**, len is larger than the maximum allowed size.
(c) `ENAMETOOLONG` (glibc **gethostname()**) len is smaller than the actual size. (Before version 2.1, glibc uses `EINVAL` for this case.)
(d) `EPERM` For **sethostname()**, the caller did not have the `CAP_SYS_ADMIN` capability.

Which errors are relevant for our call to `gethostname()`? Discuss the implications of each with your partner.

Examine the remaining code. Check the man-pages for any functions that you encounter and are unfamiliar with.

[2] Or do a Google-search.

Resolving a host name

Fire up you favorite text editor[3] and grab hold of your coffee: we're going to do some coding.

...

I'm sorry if I got your hopes up, but there's a few things we should look at before starting hacking away at stuff. First, let's look at the facilities that are provided by the network API / standard library – after all, resolving host names is a fairly common task.

Basically, the two most common choices are

- `gethostbyname()`
- `getaddrinfo()`

Both of these are available on Linux, MacOS and Windows (and probably most other semi-competent OSes) through the respective BSD networking implementations and standard libraries. The former, `gethostbyname()`, should however be considered deprecated and you should no longer use it in any new software that you write (assuming `getaddrinfo()` is available).

Therefore, you *should* use `getaddrinfo()` for this exercise. It's a bit more complicated, but has more capabilities too (for instance, we can get it to return IPv6 addresses with relative ease).

> Also, `getaddrinfo()` is thread-safe and re-entrant, since memory is allocated for the result of each call of `getaddrinfo()`. This is also why you should call `freeaddrinfo()` – otherwise each call to `getaddrinfo()` will allocate additional memory, and you'll have a memory leak.
> The `gethostbyname()` method, on the other hand, returns its result in a static `hostent` structure to which it returns a pointer. After each call to `gethostbyname()` you **must** extract the data from the returned pointer and store it elsewhere: the next call to `gethostbyname()` **will overwrite the data in the `hostent` structure**!

Read the man-page/documentation for `getaddrinfo()`. Depending on the version of your man-page, it might even contain some examples!

Representing addresses

As you've seen by now, `getaddrinfo()` uses the `addrinfo` struct to store and represent addresses. Unfortunately, this is not the whole story. Specifically, we need to look closer at the `ai_addr` member of type **`sockaddr`**.

The **`sockaddr`** structure by itself is not terribly interesting. It might look something like the following:

```
// From 'man bind'
struct sockaddr {
   sa_family_t sa_family;
   char        sa_data[14];
}
```

The only interesting field is the `sa_family`, which tells us the address family of this instance of **`sockaddr`**.

Our main interest for now is the `AF_INET` address family, which represents IPv4 addresses. There are others, such as `AF_INET6`, `AF_UNIX`, and so on. Each address family has a corresponding socket address structure:

- `AF_INET` has **`sockaddr_in`**
- `AF_INET6` has **`sockaddr_in6`**
- `AF_UNIX` has **`sockaddr_un`**

[3] Which, of course, should be vi(m). </joke>

- ...

Right now, let's focus on the `AF_INET`/**`sockaddr_in`** variant. In the man page for IPv4 ('`man 7 ip`'), the structure is documented:

```
// From 'man 7 ip'
struct sockaddr_in {
   sa_family_t    sin_family; /* address family: AF_INET */
   in_port_t      sin_port;   /* port in network byte order */
   struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
   uint32_t       s_addr;     /* address in network byte order */
};
```

Here we find two fields that are of interest: `sin_port` and `sin_addr`, which are the port and the IPv4 address, respectively.

So, when we get a **`sockaddr`**`*` pointer (without the `_in`!) that has the `sa_family` member set to `AF_INET`, we can treat that pointer as a **`sockaddr_in`**`*` pointer instead, and with that we'll get access to the actual port and IPv4 address.

Example:

```
sockaddr* sockAddr = /* get pointer to sockaddr from e.g. addrinfo struct */;

assert( AF_INET == sockAddr.sa_family );
sockaddr_in* inAddr = (sockaddr_in*)sockAddr; // cast to sockaddr_in*

// Now we can get at the port and address data!
int port = inAddr->sin_port;
uint32_t ipNumber = inAdd->sin_addr.s_addr;
// ...
```

Basically, we assume that the `sockAddr` above points to a **`sockaddr_in`** structure. This might be safe, for instance, because we got the pointer via `getaddrinfo()` where we specified `ai_family = AF_INET` in the `hints` argument. Still, since it is an assumption that we make, it is a good idea to check that it holds true. Here, **`assert`**`()` is used to do this: **`assert`**`()` checks the result of the argument, and if the argument evaluates to **`false`**, **`assert`**`()` prints an message and aborts execution.
You should not use **`assert`**`()` as a general error handling mechanism. Rather, only use it to check that your assumptions hold.

▶ I.b Putting it into code

Ok, now you're ready to write some code. Here are some suggestions on how you could use `getaddrinfo()` in the program:

- the `service` argument (the second argument), you can set to `NULL` – we're not interested in any specific port or service.
- in the `hints` structure, you'll need to specify `AF_INET` as the address family. Additionally, you can set the socket type to `SOCK_STREAM` and the protocol to `IPPROTO_TCP` from `<arpa/inet.h>`. This way, you won't get "duplicate" IPv4 addresses in the results simply because several socket types and protocols are available.
- set the remaining elements to zero, which indicates that you don't care about those.
- check the return value for errors, and use `gai_strerror()` to convert the returned error code to a human-readable string.

Also, to make the code compile, you'll probably have to add a few includes. Again, check the man-page for these.

Next, you'll want to print the results. First, `getaddrinfo()` sometimes returns several results in a linked list. You should print all of them. To convert an **`in_addr`** to a human-readable string, you can use `inet_ntop()` (check the man-page!).

Finally, don't forget to free the data produced by `getaddrinfo()` with `freeaddrinfo()`.

Tests

Resolve a few host names. Make sure that your program produces the correct results. Use e.g. the `host` command for this:

```
$ host chalmers.se
chalmers.se has address 129.16.71.10
...
```

Can you find host names that return several associated IP addresses? (Try `google.com`! Or alternatively, `multi.it12.x.stas.is`, which is configured to return two nonsensical IPv4 addresses.).

Exercise I.b.1 Can you think of any reason(s) why a host name may have several associated IP addresses? Discuss with your lab partner.

Also try to resolve an invalid host name. Hopefully your program handles that and prints an appropriate error message[4]. For example:

```
$ ./lab1-resolve foobar
Resolving 'foobar' from 'mieli':
Error in getaddrinfo(): Name or service not known
```

You may look at the reference solution that you can find in the lab 1 page at Canvas. If you want to, you can compare it to your own solution.

Part II IPv6 addresses (optional)

So far we've limited ourselves to IPv4 addresses. As you might have heard, IPv6 is gaining some traction slowly. In this part, you'll extend the above program to also look for IPv6 addresses:

```
$ ./lab1-resolve-host xkcd.com
Resolving 'xkcd.com' from 'mieli':
IPv4: 107.6.106.82
IPv6: 2001:4830:120:1::2
```

▶ II.a Hackin' it

As shown in the above example, we want to print both IPv4 and IPv6 addresses. A fairly trivial solution would be to use two consecutive calls to `getaddrinfo()`, each with a different address family specified in the `hints` agrument.

Can you do it with a single call to `getaddrinfo()`? (Hint: look at `AF_UNSPEC`!)

Tests

Make sure that you test your program. The following domains had IPv6 addresses at the time of writing:

- google.com (both IPv4 and IPv6 addresses)

[4] If you're on a weird ISP, this can be tricky. The ISPs DNS servers might return a valid address for non-existent domains; this address usually then points to a portal page of sorts. Changing DNS servers to public ones, like the ones provided by Google, might be an option in that case.

- ipv6.google.com (IPv6 only)
- ipv6.it12.x.stas.is (nonsensical IPv6 only)

The `host` utility should also print IPv6 addresses. Verify that your program's output matches the results from the `host` utility.

Exercise II.a.1 Discuss the following scenario with your partner. You have a machine that has a public IPv6 address only. You want to read the latest XKCD webcomic on *xkcd.com*. You know that the machine serving the XKCD webpage has a public IPv6 address, but you do not know the actual address. One way or another, you'll therefore do a DNS lookup.

Your machine's `/etc/resolv.conf` lists two name servers with an IPv4 address each.

Do you expect to be able to read the latest XKCD on that machine?

Solution

You may look at the reference solution that you can find in the lab 1 page at Canvas. If you want to, you can compare it to your own solution.

A Compilation Instructions

Build the programs using the following command:

```
g++ -Wall -g -o program source.cpp
```

This command will produce an executable binary from `source.cpp`. The binary will be named *program*, as specified with the `-o` option (`o` as in *output*).

The flag `-Wall` enables all Warnings. This is usually a good idea as warnings will help you find errors in your program early. The `-g` option instructs the compiler to generate debugging information, which can help you if you need to debug the program using *gdb* or similar. (Note: `g++` is a C++ compiler.)

B Instructions for accessing the remote machines at Chalmers

There are two central machines accessible to students via *ssh* at Chalmers:

- `remote11.chalmers.se`
- `remote22.chalmers.se`

Login requires a valid CID (username) and password. These machines run Linux, so you should be able to complete this lab there.

Most GNU/Linux and Mac OS X installations include a SSH client by default. For example, to login to the *remote1* machine, issue the following command in a terminal:

```
ssh CID@remote11.chalmers.se
```

You should replace `CID` with your actual CID in the example above!

Some additional examples, including using SCP to transfer files between different machines, are available at

http://write.flossmanuals.net/command-line/ssh/

Windows users can use the *Putty* SSH client, freely available at

http://www.chiark.greenend.org.uk/~sgtatham/putty/

Windows users may want to use *WinSCP* to transfer files. It is freely available at SourceForge:

http://sourceforge.net/projects/winscp/