Lab 3: Basic C Signals and Sockets; ICMP/PING, and Traceroute

Part 0 due: Sept. 24th   Part 1 due: Oct 5th   Part 2 due: Oct. 12th

Overview

The goal of this lab is to familiarize yourself with network level principals and basic socket programming by re-implementing the `ping` and `traceroute` programs.

The programming in this assignment will be in C requiring raw sockets. On most systems, accessing raw sockets requires administrative priveledge (i.e., you will require root privilege). As such, we have provided you with a virtual machine for your development where you can open raw sockets as you please as root. More info will be provided in lab.

Deliverables

Your submission should minimally include the following programs for each part of the lab:

- **Part 0**
  - `hello_signal.c`
  - `Makefile`
  - `README`
- **Part 1**
  - `ping.c`
  - `rtt_graph.png`
  - `Makefile`
  - `README`
- **Part 2**
  - `traceroute.c`
  - `rtt_graph.png`
  - `swat_net_map.*`
  - `Makefile`
  - `README`

Your `README` file should contain a short header containing your name, username, and the assignment title. The `README` should additionally contain a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your programs. Additionally, if there are any short answer questions in this lab write-up, you should provide well marked answers in the `README`, as well as indicate that you've completed any of the extra credit (so that I don't forget to grade it).

Submission Instructions

To submit part 1, use `handin43.1`. To submit part 2, use `handing43.2`. You can retrieve relevant files via `update43`

Part 0: Signals and Timing in C

This part of the lab is to be completed individually in lab. It will be graded.

You will be required to set up basic signals and signal handlers to complete your `ping` and `traceroute` program. Here you will write a sequence of simple C program that will teach you the basics of signal handling.

A signal is a operating system mechanism that enable programs to be "signaled" to take specific actions. You are probably already familiar with some UNIX signals; for example, by pressing `CTRL-C` on the terminal, you are directing the operating system (OS) to deliver a `SIGINT` signal to the running program, which usually has the effect of terminating the program. Similarly, if you press `CTRL-Z`, you are directing the OS to send a `SIGTSTP` signal, which usually has the effect of stopping a program so it can be resumed later.

In this in-class lab, we will investigate a different class of signals whose purpose is to not terminate a program, but rather instruct the program to take an action. Particularly, you will employ the `SIGALRM` signal to take periodic actions, such as ping remote host. To help you get started, I've provide a basic "Hello World" program, `hello_signal.c` which you will build upon to complete this part of the lab.

Figure 1: `hello_signal.c`

```c
1  /* hello_signal.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <signal.h>
5  #include <unistd.h>
6
7  void handler(int signum){ //signal handler
8    printf("Hello World!\n");
9    exit(1); //exit after printing
10 }
11 int main(int argc, char * argv[]){
12   signal(SIGALRM,handler); //register handler to handle SIGALRM
13   alarm(1); //Schedule a SIGALRM for 1 second
14   while(1); //busy wait for signal to be delivered
15   return 0; //never reached
16 }
```

There are two key function calls in `hello_signal.c`: `signal()` and `alarm()`. The `alarm()` system call instructs the operating system to deliver a `SIGALRM` signal after n seconds, n being 1 in this example. The `signal()` system call instructs the operating system to execute the function `handler` when the `SIGALRM` signal is delivered. Now, it is clear that the "Hello World" program sets up a signal handler for `SIGALRM`, line 12; a timing for the delivery of the `SIGALRM`, line 13; busy waits for the signal to be delivered, line 14; and once the signal is delivered, the signal handler is invoked, printing "Hello World" and exiting, line 7-9. At this point you should compile and execute `hello_signal.c` and observe the timing of the output — it is delayed by 1 second.

This style of programming with signal is based on the principals of preemptive execution. That is, the execution of the handler function *preempts* the main execution of the program. Once the signal is delivered (during the busy wait), program execution jumps to the handler function, and once the handler returns, execution jumps back to the point where the main execution was preempted (or would have, if there was not an `exit()` call). This is a very powerful (and often confusing) programming paradigm, which you will use throughout this lab.

Signal Handling Programming Problems

Program solutions to the following problems by extending `hello_signal.c`:

1. Change `hello_signal.c` such that after the handler is invoked, an additional `printf("Turing was right!\n")` occurs in `main()` before exiting. You will probably need to use a global variable and change the condition on the while loop.

2. Change `hello_signal.c` such that every second, first “Hello World!” prints from the signal handler followed by “Turing was right!” in `main()`, over and over again indefinitely. The output should look like:

```
Hello World!
Turing was right!
Hello World!
Turing was Right!
...
```

3. Program a new program `timer.c` that after exiting (via `CTRL-C`), will print out the total time the program was executing in seconds. To accomplish this task, you will need to register a second signal handler for the `SIGINT` signal, the signal that is delivered when `CTRL-C` is pressed. Conceptually, your program will request a `SIGALRM` signal to occur every second, tracking the number of alarms delivered, and when the program exits via `CTRL-C`, it will print how many alarms occurred, or the number of seconds it was executed.

Part 1: ICMP Ping Program

In this part of the lab, you and your partner will complete a `ping` program. This will require the use of ICMP (Internet Control Message Protocol) packets and the use of raw sockets. Your program will require root privileges, and so you should develop your code on the virtual machine provided or on your local machine. Additionally, you and your partner should develop a mechanism for sharing code easily, e.g., via git, svn, etc..

At the core of your `ping` program, you will craft a ICMP ECHO request packet, send it to the specified destination, and wait for a reply. You will send a packet every second, and print output based on the replies. To familiarize yourself with `ping`, try executing the built-in `ping` program that comes standard on must OS installations. For example, here is some sample output of pinging google.com on ubuntu:

Figure 2: Sample `ping` output trace

```
[aviv@myrtle] ~ >ping google.com
PING google.com (74.125.228.7) 56(84) bytes of data.
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=1 ttl=53 time=9.41 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=2 ttl=53 time=8.97 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=3 ttl=53 time=8.90 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=4 ttl=53 time=8.93 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=5 ttl=53 time=8.79 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 8.796/9.003/9.411/0.236 ms
```

Your goal is to duplicate this output as best you can. Note that after pressing `CTRL-C`, the program prints out statistics of the round trip times. To get you started, I've provided a sample program, `icmp_ex.c`, that sends a single ICMP ECHO request and receives a reply. You will alter this program so that the sending and receiving is repeated, and round trip time statistics are printed upon completion. Additionally, I have provided you with a program to calculate the Internet checkqsum. If the checksum is off, your ICMP echo may be dropped. You can find all provided programs for this lab on the course website.

Hints

- **Timing Packets**: Check out the `gettimeofday()` function, and think about embedding information in the data field of the ECHO packet, it will be echoed back at you.
- **Signals are Asynchronous**: Signals can occur at any time, and some function are not re-entrant, which means they will need to be restarted if interrupted.
- **Read the Manuals**: The man pages are your friend. In the terminal, type `man` of almost any function, and you should get some information. If not, consult the internet, but be careful not to copy code. You may find it useful to consult the manual pages for `socket`, `getaddrinfo`, `gettimeofday`, and `netinet` in particular.
- **Read the Header Files**: Many of the headers, particularly those containing structures that define a packet, are really useful to review. You can find them in `usr/include`.

- **Perform Error Checking**: C does not throw exceptions, so although your program is malfunctioning, it will still execute. Instead, take the time to perform error checking. Use `perror()` instead of manually checking the error code, but sometimes, you might want to check the error codes.

- **Network Byte order and Sequence Numbers**: Don't forget that you will need to swap the bytes where appropriate. Particularly check the sequence number.

Lab Questions/Problems

Once your program is finished, complete the following lab questions/problems:

1. Use your `ping` program it to record round trip time (RTT) statistics for the following destinations[1]. Used the average across at least 30 pings to each destination, but do not run multiple pings simultaneously. This data should be included in your submission in a space-separated file named `RTT-stats.dat` with location and RTT as the two columns. Also indicated from where on campus (via wired or wifi interface, at the science center or in your dorm, etc.) you took these measurements. It can make a difference.

   - Philadelphia: `planetlab1.cis.upenn.edu`
   - New Jersey: `planetlab1.rutgers.edu`
   - New York: `planetlab1.cs.columbia.edu`
   - Boston: `lefthand.eecs.harvard.edu`
   - DC: `planetlab1.cs.georgetown.edu`
   - Chicago: `planetlab5.cs.uiuc.edu`
   - Atlanta: `planet1.cc.gt.atl.ga.us`
   - Houston: `ricepl-4.cs.rice.edu`
   - Colorado: `planetlab2.cs.colorado.edu`
   - Montana: `pl1.cs.montana.edu`
   - Seattle: `planetlab02.cs.washington.edu`
   - Palo Alto: `pl1x1.parc.xerox.com`
   - Japan: `planetlab4.goto.info.waseda.ac.jp`
   - Korea: `netapp6.cs.kookmin.ac.kr`
   - England: `planetlab-1.imperial.ac.uk`
   - Germany: `planet1.zib.de`
   - France: `planetlab2.utt.fr`
   - Israel: `planetlab2.tau.ac.il`

2. Do you notice any patterns in the round trip times? Which destinations take the longest to reach, and which the shortest?

3. Graph the round trip time versus distance from Swarthmore, and include the graph in your submission folder. Is it always the case that places near-by have faster round trip times?

4. (**Extra Credit 5 pt**) Copy your program ping program to a new file named `fast_ping.c`. In this new program, instead of sending an ECHO request ever second, try sending it every 200 milliseconds (or 5 times a second). To complete this, you will need to use `setitimer()` instead of `alarm()`. Read the manual page on `setitimer()`.

   Rerun your RTT experiments with the faster ping interval and generate a new graph. Is there any difference? And if so, how might that difference be accounted for?

[1]All destinations are Planetlab hosts, which is a distributed collaborative experimental test bed. Over 1,000 institutions participate, and you can find the precise GPS location of each of these nodes at `https://www.planet-lab.org`

Part 2: Traceroute

With your ping program complete, in this part of the lab, you will adapt those principals to complete a traceroute program. To familiarize yourself with traceroute, here is some sample output from the Linux traceroute.

Figure 3: Sample `traceroute` output

```
[aviv@myrtle] ~ >sudo traceroute -I minus.seas.upenn.edu
traceroute to minus.seas.upenn.edu (158.130.69.89), 30 hops max, 60 byte packets
 1  spatula.cs.swarthmore.edu (130.58.68.1)  0.193 ms  0.197 ms  0.205 ms
 2  192.168.192.3 (192.168.192.3)  4.915 ms  4.921 ms  5.020 ms
 3  192.168.64.5 (192.168.64.5)  4.284 ms  4.290 ms  4.290 ms
 4  te2-2.999.ccr01.phl05.atlas.cogentco.com (38.126.144.25)  4.907 ms  4.913 ms  4.921 ms
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  minus.seas.upenn.edu (158.130.69.89)  3.737 ms  2.359 ms  7.848 ms
```

You can interpret this output as an estimation of the number of layer 3 (network layer) routers along the path to the destination, or the number of router hops on the route. In the example above, this is an ICMP based traceroute to `minus` a computer a UPenn, which is 10 hops away. Note that not all the routers along the path participate in the traceroute, which are represented with "*"; however, the routers here at Swarthmore do participate. First, `spatula` responds, which is the router at the border of the CS network; then two routers in the Swarthmore core, 192.168.192.3 and 192.168.64.5; and finally, the packet reaches Swarthmore's internet service provider, Cogentco. After that, the routers stop responding until we reach `minus`.

How does traceroute work?

A traceroute leverages the TTL (time-to-live) field in the IP header. The TTL header is designed to prevent router loops, so that packets do not just keep getting forwarded indefinitely. Instead, a router inspects the TTL field, and if the value is greater than 0, it decrements the TTL and forwards the packet on towards the destination. But, if the TTL field is 0, the packet is dropped, and the router replies to the packet originator with an ICMP `TIME_EXCEEDED` packet, essentially notifying the sender that they need to increase their TTL.

To perform a traceroute, you first set the TTL to 1 and send a ping to the destination. This packet will only reach the first router before the TTL is 0, generating an ICMP `TIME_EXCEEDED` reply which identifies the first hop. Next, you increment the TTL and repeat the process until the destination is reached. Since some routers will not send an ICMP `TIME_EXCEEDED`, you need to have a timeout before trying to ping with that TTL again; generally, there is a 3 seconds timeout and 3 attempts are made per TTL before increase the TTL value to reach the next router. You can use those settings or your own, just explain your choices in the `README`.

Setting the TTL in C

When you send a packet, the IP header is constructed for you in the `sendto()` function. To adjust the TTL manually, you need to alter the properties of the socket via the `setsockopt()`. For example, here is code that will set the default TTL to 20 for packets sent on this socket:

```
  int ttl = 20;
  setsockopt(sockfd, IPPROTO_IP, IP_TTL, &ttl, sizeof(int));
```

The first argument to `setsockopt()` is the socket file descriptor `sockfd`, which is just an integer. Next, are two flags. The first, `IPPROTO_IP`, indicates the protocol family this socket option affects, i.e., the IP header, and the second flag, `IP_TTL`, indicates exactly the option value being changed, i.e., the TTL value. Finally, the TTL value is provided by passing a pointer to the data value of the TTL, `ttl`, as well as the size of the value, which is an `int`.

Hints

- **Use What You Already Know**: There is very little difference between ping and traceroute except in the logic of sending and receiving ICMP packets. Think about taking module from ping and adopting them to traceroute.
- **Maintain State**: Traceroute requires you to maintain some state, such as waiting or sending, and how many attempts at a given TTL, and etc. Organize this information reasonably, and you'll find this assignment is much easier than you thought.
- `SIGALRM` **as the Timer**: There is no reason why you cannot continue to use `SIGALRM` as timer for timeouts waiting for a `TIME_EXCEEDED` reply.

Lab Questions/Problems

1. Describe in your own words how *your* traceroute program works. What did you try that didn't work? And, how did you address those issues?
2. Perform a traceroute to all the locations from the previous part of the lab and organize them by total hop-distance (i.e., the number of intermediary routers). Are there any hosts that you can't reach? Of those that you can reach, graph the RTT versus hops: Does there seem to be any correlation?
3. Generate a network map of the core routers on the Swarthmore network. I will provide a list of hosts to traceroute on the Swarthmore network, and do so from varied locations and connections around campus, from the wired and wireless networks. Track which core-routers are connected to other core-routers. Submit a list of core routers found as well as a visual graph of the Swarthmore network in your submission directory. We will compare it to a network map provided by ITS.
4. (**Extra Credit 15pts**) In this part of the lab, we used ICMP `ECHO_REQUEST`to perform the traceroute; however, other protocols are perfectly capable of being used for this purpose. Add in the option for your traceroute program to use UDP packets and TCP packets for a traceroute. Note, that this is a non-trivial extension. Speak with me if you'd like to attempt this.
5. (**Extra Credit 5pts**) Review the command line options for the real traceroute program. Add in options for `-f` and `-m`. Rerun the experiments to traceroute to the destinations listed in Part 1 of this lab, do these settings help reach the destinations?