Connection Establishment

Networked Systems (H) 2020-2021 – Laboratory Exercise 2
Dr Colin Perkins, School of Computing Science, University of Glasgow

1 Introduction

The laboratory exercises for Networked Systems (H) will introduce you to network programming in C using the Berkeley Sockets API, and help you understand the operation and structure of the network. The exercises will help you practice C programming, building on the Systems Programming (H) course, and introduce network programming in C. Other exercises will illustrate key points in the operation of the network. The laboratory exercises are intended to complement the material covered in the lectures. Some expand on the lectures to give you broader experience in a particular subject. Others exercises cover material, such as network programming in C, that's better taught by doing than by lecturing.

There are a mixture of formative and summative exercises. The formative exercises will give you practice in programming networked systems in C, and allow you to gain further experience in C programming. They are not assessed. The two summative exercises will focus on TCP behaviour and congestion control, and on understanding the network topology. They are assessed and together are worth a total of 20% of the marks for this course.

This exercise explores the performance of TCP connection establishment, to complement the material in Lecture 2. **This is a formative exercise, and is not assessed.**

2 Formative Exercise 2: Client-server Connection Establishment

A TCP connection starts with a three-way handshake between client and server (SYN→SYN+ACK→ACK), before it starts to send data packets. As discussed in Lecture 2, the network round-trip time (RTT) can be a significant performance factor in connection establishment, and in the performance of client-server applications. This exercise explores the timing and performance of TCP connection establishment and data transfer.

2.1 Basic HTTP Client

Make a copy of the `hello_client.c` program you wrote in Laboratory Exercise 1, and call it `http_connect.c`. Modify this program as follows:

- The `http_connect.c` program should connect to port 80 of the server, rather than port 5000 (or whatever port you chose previously). Port 80 is the port used by HTTP web servers.
- The `http_connect.c` program should send the following text to the server, in place of the simple "Hello, World!" message that was sent before:

  ```
  GET / HTTP/1.1\r\nHost: %s\r\nConnection: close\r\n\r\n
  ```

  The `%s` should be replaced by the name of the server to which the program is making a connection, and `\r` and `\n` are the C escape characters representing a carriage return and newline (hint: use `sprintf()` to print this into a string). This is a minimal HTTP/1.1 request to fetch the main page from a web server.

- After sending the data, your `http_connect.c` program should enter a loop where it reads and prints the data received from the server. This should continue until there is no more data to read, and the server closes the connection. A `recv()` call on the socket file descriptor representing the connection will return `0` when the connection is closed. Be careful to print the data without buffer overflow (hint: make sure to add a terminating '`\0`' byte to the data retrieved by `recv()` before passing it to any function that expects a C string).

Test your program by using it to retrieve, and print, the main pages from several web sites. The response returned by a web server should comprise two parts. First will be a header, starting `HTTP/1.1 200 OK` for a successful request, and continuing with several header lines. This will be followed by a blank line, then the HTML content of the page.

As an example, if your program is working correctly, then running "`./http_connect www.gla.ac.uk`" will print something like the following output:

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 19 Jan 2020 22:48:26 GMT
Server: Apache
Location: https://www.gla.ac.uk/
Content-Length: 230
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://www.gla.ac.uk/">here</a>.</p>
</body></html>
```

The header `HTTP/1.1 301 Moved Permanently` is a redirect header, signalling that the page has moved. The other header lines, ending in the `Content-Type:` line, provide further information about the page. The `Location:` header is a machine readable description of where the page has moved to, in this case to the HTTPS version of the site. The headers are followed by a blank line, then the HTML page. In this case, the HTML page is a placeholder document that says the page has moved.

Also try using your `http_connect.c` program to retrieve a longer web page, to make sure it works with pages that need more than one call to the `recv()` function to retrieve. Running "`./http_connect neverssl.com`" will retrieve a non-obfuscated static HTML site. If retrieved correctly the content will end with a `</html>` tag on a line of its own.

2.2 Timing TCP Connection Establishment

Once your `http_client.c` program is successfully retrieving the HTML source of web pages, modify it to record the time immediately before, and immediately after, it makes the `connect()` call. The `clock_gettime()` system call can be used to record the time. Store both values in variables, and print them after the `connect()` call has completed:

```
#include <time.h>
...

struct timespec before_connect;
struct timespec  after_connect;
...

clock_gettime(CLOCK_MONOTONIC, &before_connect);
if (connect(fd, ai->ai_addr, ai->ai_addrlen) == -1) {
  close(fd);
  continue;       // Try the next address
}
clock_gettime(CLOCK_MONOTONIC, &after_connect);
printf("before: %ld.%09ld\n", (long) before_connect.tv_sec, before_connect.tv_nsec);
printf(" after: %ld.%09ld\n", (long)  after_connect.tv_sec,  after_connect.tv_nsec);
```

The times are in seconds, and nanoseconds, since 1 January 1970 (as of January 2021, it is just over 1,610,000,000 seconds since 1 January 1970). Read the documentation for the `clock_gettime()` function for details.

Run your program for several websites, located in different parts of the world, to see how long the `connect()` function takes to complete. That is, subtract the *before* time from the *after* time, to find the round-trip time for the connection establishment. How do the times you get compare to the round-trip time examples shown in the lecture? Can you explain any differences?

2.3 Timing TCP Downloads

Modify your `http_connect.c` program to record the time, using the `clock_gettime()` function, immediately before it sends the request to the server to retrieve the web page using the `send()` call, and immediately after each call to `recv()`. Remove the calls that print data returned from the server (since printing the data will take long enough to disrupt the results) and instead print the values retrieved from the `clock_gettime()` calls.

How long does it take to send the request and retrieve the first part of the response? That is, what is the difference between the timestamp that is returned after the first `recv()` call, and that returned just before the `send()` call. How does it compare to the time taken to connect?

How does the time taken for each subsequent `recv()` call compare to the time taken for the first `recv()` call? That is, if you subtract the time recorded after the first `recv()` from the time recorded after second `recv()` call, how do they differ from the times between the first `recv()` and the `send()`? Similarly, how does the time taken for subsequent pairs of `recv()` calls compare. Can you explain any differences you see?

2.4 Comparing TCP Connection Timing

Many DNS names resolve to more than one IP address, to allow for load balancing across different servers, and to allow services to be available via both IPv4 and IPv6. The `getaddrinfo()` call performs a DNS lookup to find the IP addresses that a given DNS name resolves to.

Modify your `http_connect.c` program, so that rather than looping through the set of IP addresses returned by the `getaddrinfo()` call and sending the HTTP request to only the *first* address that successfully connects, it rather connects to *each* address of the server in turn, measuring the connection latency and HTTP performance for each. That is, rather than break out of the `for` loop iterating through the list of IP addresses of the server on a successful connection, instead connect to each address, measure how long the connection and download took, close the connection, then continue to try to connect to the next available address for the server.

Do you observe differences in timing when connecting to the different IP addresses of a server? That is, for those addresses that you can connect to, are some faster (i.e., have a lower RTT or download the content faster) than others? For those where the connection fails, how does the time taken to fail to connect compare to the time taken to connect? Are there any obvious differences in performance between IPv4 and IPv6 connections?

3 Discussion

This is a formative exercise. It is not assessed, and you do not need to submit your code or results. The goals are (i) to give further practise in C programming; and (ii) to complement the material covered in Lecture 2. Discuss the behaviour you see with the lecturer or lab demonstrators, to ensure you understand the timing characteristics of TCP connections, and the impact of the round-trip time on TCP performance.

- + -