Spring 2015
Computer Networks
CMPE324

Laboratory Experiment No. 7: Introduction to the Application Layer

Aims and Objectives:

- Introduce students to the application layer and socket programming.

Materials Required:

- IP routers,
- Ethernet switches,
- PCs with Ethernet adapters,
- and straight-through/crossover/rollover cables.

Change Log:

- 26-3-2015: original document – mkhonji.
- 26-3-2015: changed PC1 in 3.1.3 to PC2 – mkhonji.

1 Introduction

In this introduction, the basic functionality of the HTTP protocol is introduced, followed by an introduction to socket programming.

1.1 Hypertext Transfer Protocol (HTTP)

HTTP is an application-layer protocol. Therefore, it resides in the *payload* section of layer 4 protocols (such as TCP and UDP).

HTTP is a fairly extensive protocol. However, the basic functionality of HTTP is fairly simple. In this lab we cover simple GET requests and responses.

A HTTP GET request is presented in Figure 1. Note the following:

- This is a minimalist HTTP request. A normal HTTP request has more content (as you will observe in later sections).
- This HTTP request has a single header field that is the *request-line*.
- The request-line has a request of type GET. Note that other request types exist, however for simplicity we shall only cover GET requests.
- Note that the request line is terminated by CRLF (carriage-return followed by line-feed).
- Note that the request header is terminated by another. CRLF.

```
1 GET /path/to/resource HTTP/1.1\r\n
2 \r\n
```

Figure 1: A minimalist GET request.

An example of a minimalist HTTP response to the request above can be as presented in Figure 2. Note the following:

- This is a minimalist response. Usually, a HTTP response contains more data (as you will observe in later sections).
- The first line in the response is the *status-line* that is terminated by CRLF. This line indicates the *status code*, which is 200 in this example. As per the standard, 200 indicates that the request was successfully. This line also contains a *status phrase*, which is "*OK*" in this case. Such phrases were useful when web clients were interactive, however modern GUI HTTP clients should ignore such phrases as per the RFC7230.

```
1 HTTP/1.1 200 OK\r\n
2 \r\n
3 This is the content of the requested resource!
```

Figure 2: A minimalist HTTP response.

This lab describes the most commonly used version of HTTP to date, which is version 1.1. Further details on this can be found in RFC7230, RFC7231, RFC7233, RFC7234, RFC7235. However, note that recently (relative to the time this lab script is written), 17th of Feb 2015, IESG has approved HTTP version 2 as a Proposed Standard.

1.2 Socket Programming

In the previous labs, you were bypassing most forms of automation (except those imposed upon us by the hardware). This allowed us to craft Ethernet frames, IP packets, and TCP/UDP messages manually for experimental purposes.

However, the problem with such approach is that it's tedious, error prune and inefficient. One cannot efficiently implement the whole TCP/IP stack every time a new application is coded specially that such networking stacks are fairly broad. This can lead into code bloat, increased probability of errors/bugs as well as wasted developer time.

In this lab, we will develop applications that use TCP, IP and Ethernet without manually implementing the protocols as we did in previous labs. Thanks to *socket programming*, this can be achieved, transparently, by a few function calls.

This lab is limited to TCP. However, such concepts can be easily extended to UDP as well.

For your reference, skim throughout the manual pages *tcp(7)* and *udp(7)*. When using a POSIX-compliant operating system (such as most Linux distributions), such manual pages can be viewed by commands `man 7 tcp` and `man 7 udp` respectively.

Note: as the HTTP protocol follows a *client-server* model, we will be using terms *client* and *server* to describe TCP peers. However, note that TCP —itself— is fairly general and does not *necessarily* require following a client-server model. For example, BitTorrents is an example peer-to-peer model that uses TCP.

1.2.1 TCP Server

1. Obtain a TCP socket as presented in Figure 3. `AF_INET` refers to IPv4, `SOCK_STREAM` refers to reliable bi-directional sequenced sockets. 0 is essentially the protocol number. Since usually only TCP is the only available protocol for `SOCK_STREAM`, the protocol number is set to 0. Note that, in order to use the `socket` function, you have to include the listed header files.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Figure 3: The synopsis of the `socket` function.

2. Bind that socket to a local address and port using the function `bind` as presented in Figure 4. `socfd` is the socket obtained in the earlier step. `addr` is a pointer to a structure that contains information with regards to the local address and port number to listen to. `addrlen` specifies the size of the structure `addr` in bytes.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

Figure 4: The synopsis of the `bind` function.

3. Now that the socket is binded to a local address and port, start listening on it by calling the function `listen` as presented in Figure 5. `sockfd` is the socket binded earlier. `backlog` is the maximum number number of connections that are waiting to be accepted. Your application must ensure that the backlog queue is never full or new connections will get refused. At this stage `netstat -lp` shall show that your application is listening on the set address and port numbers.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Figure 5: The synopsis of the `listen` function.

4. Accept incoming connections from TCP clients by calling the function `accept` as presented in Figure 6. By default, this function shall *block* for until a connection from a client is established after which it returns a *connection socket* between the local server and the remotely connected TCP client. `addr2` is a memory structure that will get populated by the address of the connected peer in case your application needs this information, and `addrlen2` is the length of this memory structure. When your application does not need to analyze the client connection information, you may replace `addr2` by `NULL` and `addrlen2` by `NULL` as well.

```
#include <sys/types.h>
#include <sys/socket.h>

int conn_sockfd = accept(int sockfd, struct sockaddr *addr2,
    socklen_t *addrlen2);
```

Figure 6: The synopsis of the `accept` function.

5. Finally, once a connection socket `conn_sockfd` is obtained, you can read data from it (octets sent to you by the client) and write data to it (octets sent by you to the client). The networking stack of the kernel will then take care of the remaining networking details (i.e. Ethernet, IP and TCP headers). Quite simply, the read and write operations can be performed by the functions `read` and `write` respectively, as presented in Figures 7 and 8 respectively. Alternatively, you can use the functions `recv` and `send` when in need of setting additional arguments.

```
include <unistd.h>

ssize_t read(int conn_sockfd, void *buf, size_t count);
```

Figure 7: The synopsis of the `read` function.

```
#include <unistd.h>

ssize_t write(int conn_sockfd, const void *buf, size_t count);
```

Figure 8: The synopsis of the `write` function.

6. To terminate a connection socket `conn_sockfd`, simply use the function `close` as presented in Figure 9. At this stage, you can observe the 3-way or 4-way *FIN* handshake.

```
#include <unistd.h>

/* fd can be any socket, sockfd or conn_sockfd */
int close(int fd);
```

Figure 9: The synopsis of the `close` function.

1.2.2 TCP Client

1. Obtain a socket using the function `socket`.

2. Connect to the remote server using the function `connect` as presented in Figure 10. The TCP connection will be established against the server described in the structure `addr`. Once the connection is established, this function then returns a connection socket `conn_sockfd`.

3. Application layer data can be read and written by using functions `read` and `write` respectively as presented in Figures 7 and 8 respectively.

4. Finally, when the client finishes reading/writing the data it is supposed to, the connection can be finished by calling `close` as presented in Figure 9.

```c
#include <sys/types.h>
#include <sys/socket.h>

int conn_sockfd = connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Figure 10: The synopsis of the `connect` function.

1.2.3 How to Populate `addr`?

Figure 11 presents a code snippet that initializes the `addr`. The function `memset` is presented in Figure 12. The function `inet_addr` essentially takes the dotted-decimal IPv4 address from its input string and returns its corresponding unsigned 32bit number.

```c
/* declare the addr struct */
struct sockaddr_in addr;

/* initialize addr's content to 0 */
memset(&addr, 0, sizeof(addr));

/* set address family to IPv4 */
addr.sin_family = AF_INET;

/* set source port number to 80 */
addr.sin_port = htons(80);

/* set source IP address to "X.X.X.X" */
addr.sin_addr.s_addr = inet_addr("X.X.X.X");
```

Figure 11: A code snippet for setting `addr`.

```c
#include <string.h>

void *memset(void *s, int c, size_t n);
```

Figure 12: The synopsis of the function `memset`.

1.2.4 Error Codes

It's critical to test the returned values of the called functions to identify errors early. Below is a high-level list of functions and their respective error codes.

- `socket` — returns -1 on error.
- `bind` — returns -1 on error.
- `listen` — returns -1 on error.
- `accept` — returns -1 on error.
- `read` — returns -1 on error.

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
```

Figure 13: The synopsis of the function `addr`.

- `write` — returns -1 on error.
- `close` — returns -1 on error.
- `connect` — returns -1 on error.

2 Lab Preparation

1. Connect a PC to the routers console port using a rollover cable[1].
2. Erase the configuration of the routers[2].
3. Connect a PC to the switches console ports using rollover cables.
4. Erase the configuration of the switch[3].
5. Run Wireshark on all involved lab PCs as depicted in Figure 14.
6. Physically connect the lab as depicted in Figure 14.
7. Configure the interfaces:
   - Configure[4] `R1`'s interfaces as follows:
     - GigabitEthernet 0/0: IPv4 address 10.0.12.1, subnet mask 255.255.255.0.
     - GigabitEthernet 0/1: IPv4 address 10.0.1.1, subnet mask 255.255.255.0.
   - Configure `R2`'s interfaces as follows:
     - GigabitEthernet 0/0: IPv4 address 10.0.12.2, subnet mask 255.255.255.0.
     - GigabitEthernet 0/1: IPv4 address 10.0.2.1, subnet mask 255.255.255.0.
   - Configure[5] `PC1`'s interfaces as follows:
     - eth0: IPv4 address 10.0.1.2, subnet mask 255.255.255.0.
   - Configure `PC2`'s interfaces as follows:
     - eth0: IPv4 address 10.0.2.2, subnet mask 255.255.255.0.

[1] If using Linux: `screen /dev/ttySx` were `x` is the serial interfaces ID that is connected to the console cable. If using Windows: Use Hyperterminal or Putty to connect to `COMx` ports.

[2] `enable`, `erase startup-config`, `reload`, and make sure to answer `no` to all yes/no questions while hitting *enter* for all `confirm` prompts.

[3] `enable`, `erase startup-config`, `delete vlan.dat`, `reload`, and answer `no` to all yes/no questions while hitting *enter* for all `confirm` prompts.

[4] `enable`, `configure terminal`, `interface Gi0/0`, `ip address 10.0.0.1 255.255.255.0`.

[5] `ifconfig eth0 10.0.1.2/24`

8. Then add the IP routing tables as follows:
   - For R1: `ip route 10.0.2.0 255.255.255.0 10.0.12.2`
   - For R2: `ip route 10.0.1.0 255.255.255.0 10.0.12.1`
   - For PC1: `route add -net 0.0.0.0/0 gw 10.0.1.1`
   - For PC2: `route add -net 0.0.0.0/0 gw 10.0.2.1`
9. Test connectivity using the `ping` command to send ICMP Echo messages to PCs in other networks and receive the respective ICMP Echo-Reply messages back.

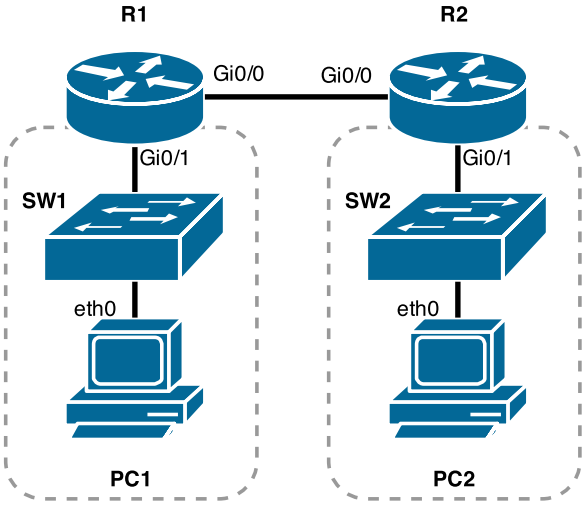

Figure 14: Physical lab topology.

3 Lab Experiments

3.1 Analyze a Reference HTTP Implementation

[40 points]

Note: When done, present your work to the lab instructor.

Throughout the steps below, *PC1* will be the web server that by which resources are published via a HTTP web server, and *PC2* will be the web client by which resources are requested and rendered via a web browser.

3.1.1 Prepare a Web Server on PC1

You are free to install any web server. However, this section presents the steps needed to setup Apache HTTPD as it is pre-installed in Kali Linux.

1. Start the web server[6].

[6]Either by using the command `service apache2 start`, or the GUI *Applications* → *Kali Linux* → *System Services* → *HTTP* → *apache2 start*

3.1.2 Publish Resources Over HTTP

Using your preferred text editor, create a few pages in your HTTP server's document root[7] as follows:

Create files `/var/www/index.html` and `/var/www/blog.html` with the contents as presented in Figures 15 and 16 respectively.

```
<html>
 <body>
  <h1>Welcome to My Homepage!<\h1>
  <p>My blog is <a href='./blog.html'>here</a>!</p>
 </body>
</html>
```

Figure 15: Index's content.

```
<html>
  <body>
    <h1>My Blog<\h1>
    <p>Work in progress. This blog will be ready soon.. not.</p>
    <p>To go back click <a href='./index.html'>here</a>.</p>
  </body>
</html>
```

Figure 16: Blog's content.

3.1.3 Access Your Published Content Over HTTP from PC2

1. From *PC2* and using a web browser, view your page. In Kali Linux, `iceweasel` is a pre-installed browser that you may use.

2. Analyze transmitted and received datagrams over the network and observe the structure of the HTTP protocol.

3. Answer the following questions after analyzing the HTTP requests and responses:

   (a) Identify when the TCP 3-way handshake, data transmission, and connection termination accor, and explain how such events are related to your interaction with the web browser.

   (b) What is the TCP destination port number of the HTTP request?

   (c) Note that you did not specify a destination port number in your web browser. Why did the web browser choose the port number?

   (d) What is the TCP source port number of the HTTP request?

   (e) How do the above port numbers in the HTTP request message (source and destination) relate to those of the HTTP response message?

[7] `/var/www/`

(f) What is the request-line of the HTTP request?

(g) What is the status-line of the HTTP response?

(h) What is the termination sequence for HTTP header fields?

(i) What is the separation sequence between HTTP fields and the message body?

3.2 Implement a Minimalist Netcat Server

[30 points]

Note: When done, present your work to the lab instructor.

Using the procedures described in the introduction section, implement a netcat server that:

- Listens on your local IP address and TCP port 80.
- When a `nc` client connects to it, it should print the data that the client sends to it to its `STDERR`.
- When a client disconnects, the server remains running and able to accept new connections from other `nc` clients.

3.3 Implement a Minimalist HTTP Server

[30 points]

Note: When done, present your work to the lab instructor.

Modify your your minimalist netcat server code such that:

- The server parses the request and attempts to identify the requested resource. The server is expected to atleast support the *GET* HTTP requests.
- Based on the requested URL, the server responds by sending the content associated with the requested URL to the requesting client.

Notes:

- As this is a minimalist experimental web server, you can ignore aspects that relate to the performance of the server. E.g. it does not have to serve requests *concurrently*.
- To simplify your programming, you are allowed to store the content of your resources in strings in your minimalist web server.
- Ensure the correct functionality of your HTTP server by accessing the resources `index.html` and `blog.html` using your favorite web browser.