ECE 4450 Thursday, 03/24/16

Lab 2: TCP, UDP and IP (Due: Friday, 04/15/16, 5pm)

Dr. Kevin Tang Handout 21

Acknowledgement: This lab is based upon the "TCP", "UDP", "IP", "ICMP" Wireshark Labs, and the programming assignment in the Chapter 4 of the textbook, © 2005–2012, J.F. Kurose and K.W. Ross, and its use is limited to this course only.

Lab 2 consists of two parts: a Wireshark lab assignment (required) and a programming assignment (optional). In Wireshark lab part, you will use the Wireshark software to explore several aspects of the TCP, UDP and IP protocol. The programming assignment asks you to implement a distributed asynchronous distance vector routing algorithm for a four-node network. Please hand in the hard copy of your lab report in the homework drop box including all your answers to the Wireshark lab assignment and programming assignment (optional).

1 Wireshark Lab (100 points)

1.1 TCP

In this section, we will investigate the behavior of the celebrated TCP protocol by analyzing a trace of TCP segments sent and received in transferring a 150 KB file from your computer to a remote server. We will study the use of sequence and acknowledgement numbers for providing reliable data transfer. We will also briefly look at TCP connection setup and we will investigate the performance (throughput and round-trip time) of the TCP connection between your computer and the server. Before continuing, you will probably want to review Section 3.5 in the text.

In the following, you will first access a Web page that allows you to enter the name of a file stored on your computer, and then transfer the file to a Web server using the HTTP POST method (see Section 2.2.3 in the text). We are using the POST method rather than the GET method as we would like to transfer a large amount of data from your computer to another computer. Do the following:

- Start up your Web browser. Go to
`http://gaia.cs.umass.edu/wiresharklabs/alice.txt`
and retrieve an ASCII copy of *Alice in Wonderland*. Store this file somewhere on your computer.
- Next go to
`http://gaia.cs.umass.edu/wireshark-labs/TCP-wireshark-file1.html`
and you will see a form for uploading a file.

- Use the `Browse` button on this page to select the file you just downloaded. Do *not* press the `Upload alice.txt file` button at this time.
- Start up the Wireshark packet sniffer.
- Returning to your browser, press the `Upload alice.txt file` button now. Once the file has been uploaded, a short congratulation message will be displayed in your browser window.
- Stop Wireshark packet capture.
- Type `http` (in lower case) into the display filter at the top of the Wireshark window and then press `Enter`. Right-click the HTTP POST request in the packet list and choose `Follow > TCP Stream`.
- Close the prompted window. Now the TCP stream between your computer and `gaia.cs.umass.edu` server carrying the POST message is filtered out.

At the beginning of the packet list, you should see the initial three-way handshake containing a SYN and a SYNACK message. Some TCP segments in the middle are marked by `[TCP segment of a reassembled PDU]`, which indicates that they contain data that belongs to an upper layer protocol message (HTTP here). You should also see TCP ACK segments being returned from `gaia.cs.umass.edu` to your computer.

In the Wireshark capture, you may find some TCP segments whose length is much larger than the usual MTU (maximum transmission unit) of the Ethernet. This is caused by the TCP segmentation offload in the modern NIC (network interface card). With this feature, some TCP segmentation and reassembly work is done by the hardware instead of the operating system to reduce the CPU overhead, and thus a large TCP segment seen by Wireshark is a result of several actual TCP segments combined by the hardware in the NIC.

Answer the following questions (40 points):

1. What is the TCP port number used by your computer and `gaia.cs.umass.edu` server?
2. What is the sequence number of the SYN segment that is used to initiate the TCP connection between the client computer and `gaia.cs.umass.edu`? Which fields in the segment identify the segment as a SYN segment?
3. What is the sequence number of the SYNACK segment sent by `gaia.cs.umass.edu` to the client computer in reply to the SYN? What is the value of `Acknowledgement` field in the SYNACK segment? How did `gaia.cs.umass.edu` determine that value? Which fields in the segment identify the segment as a SYNACK segment?
4. What is the sequence number of the TCP segment containing the HTTP POST command? Hint: In order to find the POST command, you need to dig into the content of packets displayed at the bottom of the Wireshark window, looking for a segment containing the characters `POST` within its `Data` field.
5. What is the minimum amount of available buffer space advertised at the receiver for the entire trace? Does the lack of receiver buffer space ever throttle the sender?

Hint: The receive window size is followed by `Win` in the packet summary. It is also given in the `Calculated window size` line of the packet details. The original receive window size field in the TCP header is not scaled by window scaling factor.

6. Estimate the RTT between your computer and `gaia.cs.umass.edu` by calculating the time difference between SYN segment and SYNACK segment.
7. What is the average throughput for the TCP connection? Explain how you calculated this value.

1.2 UDP

In this section, we will take a quick look at the UDP transport protocol. We will observe the UDP traffic sent from some background service (for example, SNMP described in Chapter 9) in your operating system. You may want to review Section 3.3 in the text for this part. Now do the following:

- Start up the Wireshark packet sniffer.
- Wait for some amount of time, then stop Wireshark packet capture.
- Type `udp` (in lower case) into the display filter at the top of the Wireshark window and then press `Enter`.
- Click an arbitrary packet listed in the Wireshark window.

Answer the following questions (15 points):

8. By consulting the displayed information in the packet content area of the Wireshark window, determine the length (in bytes) of each of the UDP header fields.
9. The value in the `Length` field of the UDP header is the length of what? Verify your claim with your captured UDP packet.
10. What is the maximum number of bytes that can be included in a UDP payload? What is the largest possible source port number? Hint: Look at your answer to the above two questions.

1.3 IP

In this section, we will examine the IP protocol in details by using the `ping` tool to send IP datagram with different lengths. Recall that the `ping` program is a simple tool that allows anyone (for example, a network administrator) to verify if a host is live or not. The `ping` program in the source host sends a packet to the target IP address; if the target is live, the target host responds by sending a packet back to the source host. Both of these packets are ICMP packets. Before beginning this part, you will probably want to review Section 4.4 in text. Do the following:

- Start up the Wireshark packet sniffer.

- Open a command prompt window. If you are on Windows, execute the following command:

  ```
  ping -n 3 -l 100 gaia.cs.umass.edu
  ```

  If you are on Linux or Mac OS X, execute the following command:

  ```
  ping -c 3 -s 100 gaia.cs.umass.edu
  ```

  Your computer will send 3 ping messages to `gaia.cs.umass.edu` with 100 bytes payload.
- Repeat the previous step but using 3000 instead of 100.
- Stop Wireshark tracing.
- Type `ip.dst==128.119.245.12` (in lower case) into the display filter at the top of the Wireshark window and then press `Enter`. All packets sent to the destination `gaia.cs.umass.edu` will be filtered out. Here `128.119.245.12` is the IP address of `gaia.cs.umass.edu`, as you may have already known from the previous lab.

Answer the following question (30 points):

11. Select the first ICMP echo request message sent from your computer. Within the IP header, what is the value in the upper-layer protocol field? Why does *not* this packet have source and destination port numbers?
12. For the above packet, how many bytes are in the IP header? How many bytes are in the payload of the IP datagram? Explain how you determined these numbers.
13. Has the `Identification` field in the IP header changed for the first three echo request messages? Explain the reason.
14. Examine the first fragment of any fragmented IP datagram. What information in the IP header indicates that the datagram has been fragmented? What information in the IP header indicates whether this is the first fragment versus a later fragment? How long is this IP datagram?
15. Examine the second fragment of the fragmented IP datagram. What information in the IP header indicates that this is not the first datagram fragment? Are there more fragments? How can you tell?

1.4 ICMP

You have already got a small taste of ICMP protocol in the last section. In this section, we will further investigate the ICMP message format using the `traceroute` program. Before continuing, you may want to review the Section 1.4.3 and Section 4.4.3 of the text.

Recall that `traceroute` operates by first sending one or more datagrams with the TTL field in the IP header set to 1; it then sends a series of one or more datagrams towards the same destination with a TTL value of 2; and so on. A router must decrement the TTL in each received datagram by 1 (actually, RFC 791 says that the router must decrement the TTL by *at least* one). If the TTL reaches 0, the router returns an ICMP TTL-exceeded message to the sending host. As a result of this behavior, a datagram with a TTL of 1 will cause the router one hop away from the sender to send an ICMP TTL-exceeded message back to the sender; the datagram sent with a TTL of 2 will cause the router two hops away to send an ICMP message back to the sender; and so on. In this manner, the host executing `traceroute` can detect the path between itself and the destination. Now do the following:

- Start up the Wireshark packet sniffer.
- Open a command prompt window. If you are on Windows, execute the following command:

  ```
  tracert gaia.cs.umass.edu
  ```

  If you are on Linux or Mac OS X, execute the following command:

  ```
  sudo traceroute -I gaia.cs.umass.edu
  ```

  Here the `-I` option forces `traceroute` to use ICMP instead of UDP.
- Wait until the `traceroute` program terminates. Make sure that the destination `gaia.cs.umass.edu` has been reached successfully. Then stop Wireshark tracing.
- Type `icmp` (in lower case) into the display filter at the top of the Wireshark window and then press `Enter`.

In your trace, you should be able to see the series of ICMP echo request sent by your computer and the ICMP TTL-exceeded messages returned to your computer by the intermediate routers.

Answer the following questions (15 points):

16. Examine the *last* echo request message sent from your computer. What are the ICMP type and code numbers? What other fields does the ICMP header have?
17. Examine the *last* ICMP message returned to your computer. Compare the ICMP header of this message with the ICMP header you examined in the previous question.
18. Examine any ICMP TTL-exceeded message. What are the ICMP type and code numbers? What information is included in the ICMP message payload?

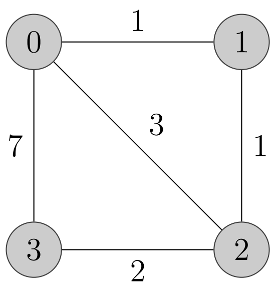

Figure 1: Network topology and link costs. Each link in the network is bi-directional.

2 Programming Assignment (40 points)

2.1 The Basic Assignment (20 points)

In this part, you will write a distributed set of procedures that implement a distributed asynchronous distance-vector routing algorithm for the network shown in Figure 1. Your task is to calculate the distance between every pair of nodes and figure out the entries in the forwarding table of each node. Before doing this part, make sure that you have understood Section 4.5.2 of the text.

The distance-vector algorithm requires neighbors to exchange their distance vectors through the link. Instead of running the code on a real network, we provide you a simple network emulator written in C. To send the distance vector of a node to its neighbor, you can call the routine `tolayer2(struct rtpkt packet)`, where `rtpkt` is the following structure:

```
struct rtpkt
{
    int sourceid;       /* id of node sending this pkt */
    int destid;         /* id of router to which pkt being sent
                        (must be an immediate neighbor) */
    int mincost[4];     /* min cost to node 0 ... 3 */
};
```

Note that `tolayer2` is passed a structure, not a pointer to a structure. Then the parameter `packet` will be delivered to the specific destination in-order and without loss. Only directly-connected nodes can communicate. The delay between is sender and receiver is variable (and unknown).

The four files `node0.c`–`node3.c` contain the skeleton code for the corresponding nodes. For node 0, you will complete the following routines defined in file `node0.c`:

- `void rtinit0()`: This routine will be called once at the beginning of the emulation. This is the place for you to initialize the distance-vector algorithm.
- `void rtfinalize0()`: This routine will be called when the emulation terminates. At this time, there is no more information being exchanged between neighbors. This is the place for you to output the calculated results.
- `void rtupdate0(struct rtpkt *rcvdpkt)`: This routine will be called by the emulator when node 0 receives a routing packet that was sent to it by one of its directly connected neighbors. The parameter `*rcvdpkt` is a pointer to the packet that was received.

You also need to complete the similar routines defined for nodes 1, 2 and 3. The file `routing.c` is the code for the emulator, and you should not modify the content in this file except for the value of `TESTCASE` (see next part). You are *not* allowed to declare any global variables that are visible outside of a given C file (e.g., any variables you define in `node0.c` may only be accessed inside `node0.c`). This is to force you to abide by the coding conventions that you would have to adopt if you were really running the procedures in four distinct machines. Calling the `tolayer2` routine is the only way to exchange information between nodes.

At the end of the emulation, you should output the distances between every two nodes. You should also output the forwarding table of every node, i.e., what is the next hop of a packet to a given destination address. Hint: When calling the routine `tolayer2`, the emulator will automatically output a line including the source, destination, and the content of the sent packet, which is very helpful for your debugging.

Since the code makes no use of specific operating system features, this assignment can be completed on any machine with C compiler. For example, if you are using `gcc`, run the following command to compile your code:

```
gcc -o routing node0.c node1.c node2.c node3.c routing.c
```

2.2 Handling the Link Cost Change (20 points)

In this part, you will make your algorithm adaptive to the link cost change that may happen during your calculation. You are to write two new procedures, `rtlinkhandler0` and `rtlinkhandler1`, which will be called by the emulator when the cost of the link between node 0 and node 1 changes. The routines will be passed the name of the neighboring node on the other side of the link whose cost has changed (in the parameter `linkid`), and the new cost of the link (in the parameter `newcost`).

We provide you two additional test cases including link cost change. To test your program, first change the value of constant `TESTCASE` in line 3 of file `routing.c` from 0 to 1. After that, the cost of the link will change from 1 to 20 at time 10000 during the emulation. If you change the value of `TESTCASE` to 2, the link cost will change from 1 to 20 at time 10000 and then change from 20 to 3 at time 20000.

What to hand in: In your lab report, you should include your code in `node0.c`–`node3.c` and the output of your program for the three test cases (i.e., with `TESTCASE` equals 0, 1, and 2).