Introduction to Wireshark

Fall 2019

Contents

1 Set Up

You should be familiar with Wireshark and have Wireshark set up for the dev environment that you will complete your TCP project in. If you need help with this, refer to the first Wireshark lab[1]. For this lab use the class provided VM.

1.1 Nodes

You will need to set up the starter TCP nodes we provide for this assignment. If your host machine for the VM is Windows, open a PowerShell terminal, otherwise open a terminal, then navigate into the directory that contains the Vagrantfile for the class provided VM. Make sure this VM is running, if it is not you can start/wake it with the command `vagrant up`. In the directory with the class provided Vagrantfile, clone the `ip-tcp-starter` repo with the command :

```
git clone https://github.com/brown-csci1680/ip-tcp-starter.git
```

[1]regardless of what environment you use for development, we will be grading on the class provided VM. Make sure your project is fully functional on the class provided VM.

In a terminal inside the VM navigate to the cloned repo and create the `lnx` files used to describe the mock network for your nodes by running the following commands :

1. `cd /vagrant/ip-tcp-starter/tools`
2. `./net2linx ../nets/AB.net`
3. `mv ./A.lnx ./B.lnx ../tcp_tools`

To create two TCP nodes create two terminals and in one terminal run :

1. `cd /vagrant/ip-tcp-starter/tcp_tools`
2. `./ref_tcp_node A.lnx`

and in the terminal run :

1. `cd /vagrant/ip-tcp-starter/tcp_tools`
2. `./ref_tcp_node B.lnx`

At this point we should have two TCP nodes that can talk to each other (each terminal should be ready with a REPL indicated by a single >). Keep your third terminal open to manipulate files later in the lab. We will refer to the node initiated with `A.lnx` as node A and the node initiated with `B.lnx` as node B (remember which node is which).

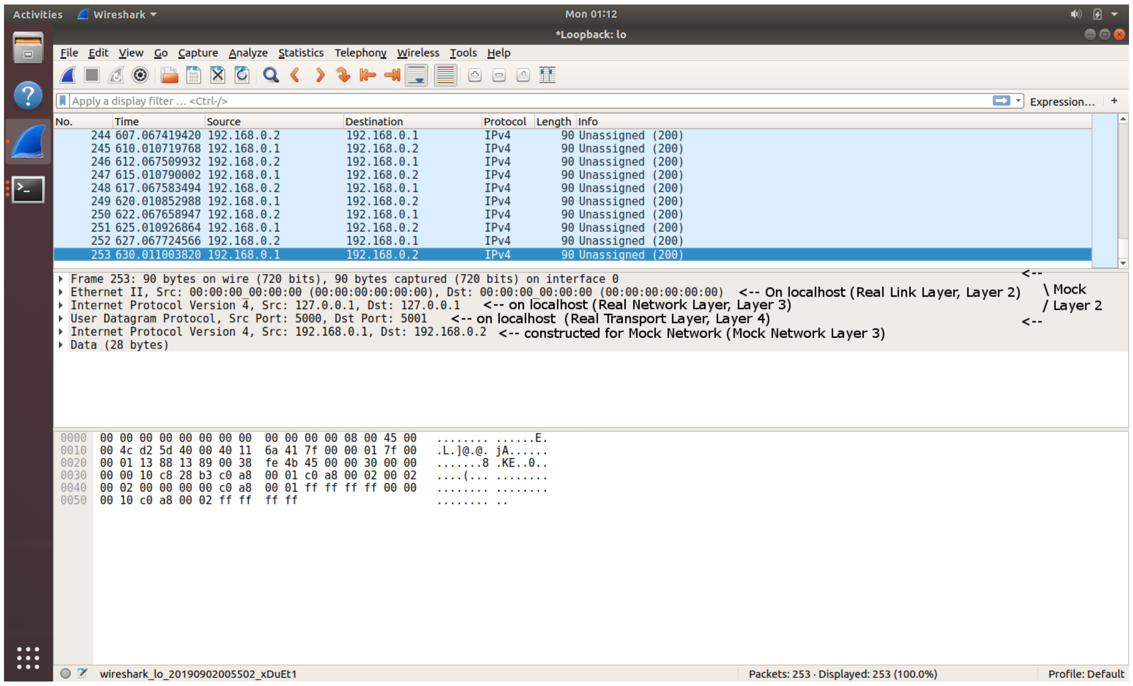
1.2 Utility Files

In your third terminal not running a node run the following commands to create two blank files (`sinkA` and `sinkB`) nodes will be able to read data into and a $50K$ file of random data that nodes will be able to send between each other (`source`) :

1. `cd /vagrant/ip-tcp-starter/tcp_tools`
2. `touch sinkA sinkB source`
3. `dd if=/dev/urandom iflag=fullblock of=source bs=75K count=1`

1.3 Encapsulation for the TCP project

Open up Wireshark, and select the `lo` (loopback) interface. You should see several UDP packets constantly popping up in the capture. These are IP RIP packets, but IP is in the network layer (layer 3) and UDP is in the transfer layer (layer 4). Wireshark will not automatically dissect our constructed layer 3 IP packet inside a layer 4 UDP packet because Wireshark does not recognize a lower layer wrapped in a higher layer. See the annotated capture below :



Activities
Wireshark
Mon 01:12
*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…

```
No.   Time          Source       Destination  Protocol Length Info
  244 607.067419420 192.168.0.2  192.168.0.1  IPv4     90 Unassigned (200)
  245 610.010719768 192.168.0.1  192.168.0.2  IPv4     90 Unassigned (200)
  246 612.067509932 192.168.0.2  192.168.0.1  IPv4     90 Unassigned (200)
  247 615.010790002 192.168.0.1  192.168.0.2  IPv4     90 Unassigned (200)
  248 617.067583494 192.168.0.2  192.168.0.1  IPv4     90 Unassigned (200)
  249 620.010852988 192.168.0.1  192.168.0.2  IPv4     90 Unassigned (200)
  250 622.067658947 192.168.0.2  192.168.0.1  IPv4     90 Unassigned (200)
  251 625.010926864 192.168.0.1  192.168.0.2  IPv4     90 Unassigned (200)
  252 627.067724566 192.168.0.2  192.168.0.1  IPv4     90 Unassigned (200)
  253 630.011003820 192.168.0.1  192.168.0.2  IPv4     90 Unassigned (200)
```

```
▸ Frame 253: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface 0                                    <--
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)  <-- On localhost (Real Link Layer, Layer 2)    \ Mock
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1    <-- on localhost (Real Network Layer, Layer 3)          / Layer 2
▸ User Datagram Protocol, Src Port: 5000, Dst Port: 5001    <-- on localhost (Real Transport Layer, Layer 4)            <--
▸ Internet Protocol Version 4, Src: 192.168.0.1, Dst: 192.168.0.2  <-- constructed for Mock Network (Mock Network Layer 3)
▸ Data (28 bytes)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 4c d2 5d 40 00 40 11  6a 41 7f 00 00 01 7f 00   .L.]@.@. jA......
0020  00 01 13 88 13 89 00 38  fe 4b 45 00 00 30 00 00   .......8 .KE..0..
0030  00 00 10 c8 28 b3 c0 a8  00 01 c0 a8 00 02 00 02   ....(... ........
0040  00 02 00 00 00 00 c0 a8  00 01 ff ff ff ff 00 00   ........ ........
0050  00 10 c0 a8 00 02 ff ff  ff ff                     ........ ..
```

wireshark_lo_20190902005502_xDuEt1
Packets: 253 · Displayed: 253 (100.0%)
Profile: Default

We are using the UDP data segment plus every layer used to transport the UDP data segment as our link Layer (layer 2) for our constructed IP packets. In our mock OSI model, our network makes sense. However, from Wireshark's perspective the entire packet of bytes looks like an Ethernet frame containing an IP packet which is holding a UDP data segment which in turn holds a payload of disorganized bytes. We can explicitly tell Wireshark to dissect the payload as an IP packet by going under "Analyze" then "Decode As" and entering which UDP ports will be sending UDP data segments containing IPv4 packets. In the "Decode as" table create two new entries (click the + button to create an entry, and the − button to delete an entry), one for UDP port 5000 and one for UDP port 5001. Decode each port as IPv4. You should now see that all the packets correctly decoded as IPv4 packets.

2 TCP basics

In this lab we will go over three basic portions of a TCP connection: three way handshake, data transfer, and connection close. You will need a TCP state diagram ready to follow how the state of each connection on each node changes state based on the TCP packets sent and received. We recommend a state diagram from the class slides, as some diagrams off the internet are missing parts/incorrectly labeled.

2.1 Handshake

No TCP packets have appeared yet in the capture because we have not yet sent any information over the loopback interface. To initiate a TCP connection first use "tcp" as your filter so we will see only the TCP packets. As you use the TCP reference node interface you can use the `help` command at any time to see the list of commands and how to use each command.

2.1.1 Listen

On node A open a port for listening with the following command :

```
a 1025
```

to listen for TCP connections on port 1025. Use the the `ls` command at each node and look at your TCP state diagram to confirm the number of connections at each node and the state of every connection is as you expect.

2.1.2 Establish

At at node B connect to node A using the following command :

```
c 192.168.0.1 1025
```

In Wireshark you should see three packets (the TCP three way handshake). For each packet, note that the `Length` column lists the total number of bytes of the entire frame "sent over the wire" however the `Len` under `Info` is the length of the payload in the TCP packet. The length of the payload will be zero for packets used in establishing a connection since they have no payload, but `Len` will be non-zero as a TCP packet is always made of some number of bytes because at the least it contains a TCP header. Also note that under `Info` Wireshark lists the Flags of the TCP packet (such as `ACK` or `SYN`) use these flags, the `ls` command, and your state diagram to confirm the number of connections and the state of each of these connections are as you expect. Make sure you can follow the TCP state diagram to understand how each connection entered the state they are currently in.

2.2 Data transfer

At node A send a short message to node B using the following command :

```
s 1 <message>
```

then read the message at node B with the following command :

```
r 0 <length of message>
```

In Wireshark you should see one data packet sent from the transmitting node with length equal to the number of bytes you sent, and one ACK packet returned by the receiving node. Inspect the message with a payload to find your message in the TCP packet.
Find the `Seq`, `Ack`, and `Win` numbers in under the `Info` column. Seq is the *relative* sequence number (important to remember when you start debugging your TCP code), Ack is the ACK number, and Win is the advertised window size. make sure you know why these fields have the values they do in each packet.
Note that the `Len` of your sent message is the number of characters you typed to send, and the `Win` number in the returned packet has decreased by `Len` (again, make sure you understand why). To

get a better idea of how `Win`, `Ack`, and `Seq` work, try sending data back and forth between the nodes with the `s` command and reading data at different points with the `r` command. Keep track of how many bytes you have sent from one node to another, how many bytes you have read at each node (sometimes read a packet after it is received, sometimes don't), and how many bytes are still in the buffer. Match these values with how `Len`, `Win`, `Ack`, and `Seq` operate.

2.3 Connection close

At node A close the connection using the command :

```
cl 1
```

In Wireshark check the packets and at each terminal check the state of your connections. Use your state diagram and make sure to understand how each connection arrived at its current state based on the packets you captured.
At Node B close the connection using the following command :

```
cl 0
```

Again check Wireshark, the state of your connections and with the help of your state diagram make sure to understand how each connection arrived at its current state.

3 Full TCP trace

Now you have seen the three basic parts of a TCP connection: three-way handshake, data transmission, and connection close, we can create a more realistic TCP capture. Start a new capture, then at node A enter the following command:

```
rf sinkA 1026
```

and at node B enter the following command :

```
sf source 192.168.0.1 1026
```

then check Wireshark, you should see the life cycle of and entire TCP connection. Note that the Window of the receiving node does not continually decrease, it stays near its full capacity of 65535 because the receiving node is continually dumping its internal buffer into the blank file (you can use `ls -l` to check that the blank file was populated). Again, with the capture and your state diagram go over how each node changed state to follow the life cycle of a TCP connection.
To understand why it is important that the listening node has a file to dump data into, start a new capture and at node A enter the following command :

```
a 1027
```

then at node B enter the following command :

```
sf source 192.168.0.1 1027
```

then check Wireshark. What is the problem? You should understand what the problem is and how it happened, but you do not need to know how zero-window probing works for this lab.

3.1 Analytic tools

Start a new capture then at node A enter the command :

```
    rf sinkA 1028
```

and at node B enter the command :

```
    sf source 192.168.0.1 1028
```

We can Analyze this capture by going under "Statistics" then "TCP stream graphs". There are a few options available for a graph, and it is advised you give each type a look, but we will start with a Steven's graph. The default type of Steven's Graph (please explore the other types as well) is time vs sequence number. You can click `Switch Direction` to look at the data transfer form node A to node B or vice-versa (look at the IP and ports at the top of that graph to see the data trasfer direction). Since sequence number is always static or increasing, so is the graph and the general shape of the graph can help you detect re-transmissions, data transfer rate, and dropped packets. Graphs will become very helpful when analyzing performance of your TCP implementation and occasionally during debugging. If you implement congestion control, Steven's graphs will help you understand exactly what your algorithm is doing.

4 Closing Tips

Note as we see in the Steven's graph, our data transmission was strictly from node B to node A. Even if we send a file from node A to node B our data transfer will still only involve one node talking at a time, however one of the great properties of TCP is that it is a full duplex protocol meaning that both nodes could be sending data to each other at the same time. Make sure to test full duplex communication in your project and check what the Wireshark capture looks like (maybe check some graphs as well).
Again, use Wireshark for your TCP assignment. Over all, use Wireshark for TCP. It can be very helpful in validating that your packets are well formed, you are correctly following the protocol, and your TCP implementation has a high data throughput. If you implement congestion control, Wireshark graphs are very helpful in analyzing the behavior of your algorithm.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS168 document by filling out the anonymous feedback form:

`https://edstem.org/us/courses/18576`.