# CS 536 Fall 2021

# Lab 3: Traffic Monitoring and Simple File Server [190 pts]

# Due: 10/18/2021 (Mon.), 11:59 PM

## Objective

The first objective of this lab is to monitor Ethernet LAN traffic by capturing and analyzing Ethernet frames generated by the ping application from lab2. The second objective is to build a simple file server by modifying the remote command server of lab2.

---

## Reading

Read chapter 3 from Peterson & Davie (textbook).

---

## Problem 1 [70 pts]

### 1.1 System set-up, traffic generation, and capture

In this problem, you will sniff Ethernet frames on an Ethernet interface veth0 on one of the pod machines in LWSN B148. When sniffing Ethernet frames, you are putting the interface in promiscuous mode which requires superuser privilege to do so. On a pod machine, run

```
% sudo /usr/local/etc/tcpdumpwrap-veth0 -c 12 -w - > testlogfile
```

which will capture 12 Ethernet frames and save them into testlogfile. Enter your password when prompted. tcpdumpwrap-veth0 is a wrapper of tcpdump that allows sudo execution. Check the man page of tcpdump for available options. To generate traffic arriving on veth0, use the ping app from Problem 1, lab2, with server mypingsrv running on a pod machine bound to IP address 192.168.1.1. The client, mypingcli, is executed from the same machine using

```
% veth 'mypingcli 192.168.1.2 192.168.1.1 srv-port'
```

where veth, similar to our remote-command execution app from Problem 2, lab2, executes mypingcli at a machine with IP address 192.168.1.2. Thus the client transmits/receives packets on interface 192.168.1.2, and the server transmits/receives traffic through interface 192.168.1.1. 192.168.1.1 is a private IP address that is not routable on the global IP Internet which has been configured for veth0. 192.168.1.2 is the IP address at the opposite end of veth0 as if the two interfaces were connected by a point-to-point Ethernet link.

For security reasons, we cannot perform sniffing on eth0 which is the interface through which the lab machines, a shared resource, are connected to the Internet. Therefore we use dummy/virtual interfaces in Linux that allows veth0 to be configured as a separate Ethernet interface -- albeit virtual, not physical -- with private IP address 192.168.1.1 that can reach 192.168.1.2, and vice versa. Thus performing veth at 192.168.1.2 on a pod machine does not execute mypingcli on a different physical machine equipped with an Ethernet interface veth0 with IP address 192.168.1.2. Instead, both server mypingsrv and client mypingcli run on the same physical machine, and packet forwarding is handled virtually by Linux as if 192.168.1.2 were a physical Ethernet interface on a separate machine. For our Ethernet frame sniffing and inspection exercise, this will suffice.

### 1.2 Traffic analysis

Use parameters for mypingcli from Problem 1, lab2, so that at least 12 Ethernet frames are generated by the ping client/server app which will be captured by tcpdumpwrap-veth0. After doing so, analyze testlogfile using wireshark or tcpdump (tcpdump is also an analysis tool). Wireshark (/usr/bin/wireshark-gtk), the postcursor of ethereal, is a popular graphical tool for analyzing (as well as capturing) traffic logs in pcap format. Use wireshark or tcpdump to inspect the 12 captured Ethernet frames. Using the MAC address associated with 192.168.1.1 (perform ifconfig -a on a pod machine) and the MAC address associated with 192.168.1.2 (perform veth 'ifconfig -a' on the same pod machine), identify the relevant Ethernet frames whose payload are IPv4 packets that, in turn, contain UDP packets generated by the ping app, as their payload. Check the type field of the captured Ethernet frames to confirm that they are DIX frames.

The first 20 bytes of Ethernet payload comprise IP header and the next 8 bytes the UDP header. The last 8 bytes of the IP header specify the source IP address and the destination IP address. Check their values against the IP addresses used by the ping app. The first four bytes of the UDP header specify the source and destination ports. Check that they match the port numbers used by the ping app. Inspect the remaining bytes of the captured Ethernet frames which comprise the application layer payload that mypingcli and mypingsrv sent using sendto(). For example, the client request is 5 bytes long comprised of a 4-byte sequence number and 1-byte control field. Wireshark/tcpdump will provide output where Ethernet header fields (MAC addresses and type) are decoded. Inspect the captured raw data in hexadecimal form to match the IPv4 addresses and UDP port numbers. Use wireshark/tcpdump as a confirmation tool. Do the same when analyzing the application layer payload carried by the Ethernet frames. Discuss your findings in lab3.pdf.

*Note: To run tcpdump, please use the command, % tcpdump -r - < testlogfile, instead of, % tcpdump -r testlogfile, which will trigger an "access denied" error. As noted in class, you may also run the command-line version of wireshark, /usr/bin/tshark, instead of tcpdump. To run wireshark, you will have to be physically at a lab machine. If you prefer, you may install wireshark on a Windows, MacOS, or Linux machine, copy testlogfile to the machine and run wireshark to inspect captured frames.*

---

## Problem 2 [120 pts]

Modify Problem 2, lab2, so that the remote command server becomes a file server. The client, myftpc, is executed with command-line arguments

```
% myftpc server-IP server-port filename secret-key blocksize
```

that specify server coordinates, name of the file to fetch, a secret key for authentication, and a blocksize (in unit of bytes) that is used to write the content of the transferred file. We will restrict file names to be less than or equal to 8 ASCII characters which must be lower- or upper-case alphabet. The secret key is an integer in the interval [0, 65535]. The client's request consists of a 2 byte unsigned integer which encodes the secret key, and a filename whose length is between 1-8 bytes. The client rejects command-line input that does not meet the length specifications. The client takes a timestamp before transmitting its request to the server, and a second timestamp after the last byte of the file has been received and written to disk. The client outputs to stdout the size of the file (in unit of bytes), completion time (second timestamp minus first timestamp), and throughput (file size divided by completion time).

The server, myftps, is executed with command-line arguments

```
% myftps server-IP server-port secret-key blocksize
```

which specify the IP address and port number it should bind to, a secret key used to authenticate client requests, and a blocksize (in unit of bytes) that is used to read the content of the requested file. The server should not read more than 10 bytes from the client request to prevent buffer overflow. If the secret key communicated as the first two bytes of a request does not match the secret key provided in its command-line arguments, the request is ignored. The same goes if the filename fails to meet the naming convention. These checks are performed in addition to source IP address filtering carried out in Problem 2, lab2. The server will continue to toss a coin and ignore a request if it comes up heads. If a client request passes the preceding checks, the server process verifies that the requested file exists in the current working directory. If not, the request is ignored by closing the connection. Otherwise, a child is forked that reads the file content using read() in unit of blocksize bytes and calls write() to transmit the bytes to the client. The child closes the connection after the last byte has been transmitted.

Note that overall file transfer client/server performance is influenced by the particulars of both network I/O and disk I/O. In general, for file servers without specialized kernel support a rule of thumb is to perform disk I/O to read/write files in unit of a file systems block size. Try blocksize values 512, 1024, 2048, 4096 for small (tens of KB) and large (tens of MB) files to evaluate performance. Larger blocksize values reduce the number system calls which improves disk I/O performance. In the case of write operations to network sockets, considering the payload size of the underlying LAN technologies to prevent fragmentation (we will discuss it under IP internetworking) can improve performance. In this problem, set the count argument of write() and read() to sockets to blocksize. When testing your file server app, make sure to verify that the requested file has been transferred correctly, for example, by using file checksums. Compare file transfer performance when server and client are located on two machines in the same lab (e.g., LWSN B148) or across two different labs (LWSN B148 and HAAS G050). Discuss your findings in lab3.pdf. Implement your file server app in a modular fashion and provide a Makefile in v1/. Include README under v1/ that specifies the files and functions of your code, and a brief description of their roles. Henceforth lack of modularity, annotation, and clarity of code will incur 5% point penalty. Make sure to remove large files created for testing after tests are completed.

---

## Bonus problem [15 pts]

One of the performance factors ignored in Problem 2 is the influence of network file systems which carry out network I/O combined with caching to enable access to mounted file systems that may be remotely located. A method that reduces the effect is to run client and server processes so that they access files on local file systems of our lab machines, in particular, /tmp. For a large file and the blocksize that yields the best performance in Problem 2, carry out experiments across two lab machines and determine if there is a performance difference. Discuss your finding in lab3.pdf. Remove files in /tmp after tests are completed.

---

## Turn-in Instructions

*Electronic turn-in instructions:*

We will use turnin to manage lab assignment submissions. Place lab3.pdf under lab3/. Go to the parent directory of the directory lab3/ and type the command

```
turnin -v -c cs536 -p lab3 lab3
```

This lab is an individual effort. Please note the assignment submission policy specified on the course home page.

---

Back to the CS 536 web page