CS 536 Fall 2019

Lab 3: File Servers, Adaptive RTT Estimation, and Traffic Monitoring [250 pts]

Due: 10/16/2019 (Wed), 11:59 PM

Objective

The objectives of this lab are to implement a performance-oriented file server using the concurrent client/server framework of lab2/lab1, improve the stop-and-wait file server app of lab2 by endowing it with adaptive RTT estimation capability, and perform traffic monitoring to inspect header and payload of Ethernet frames.

Problem 1 [120 pts]

1.1 App interface

Modify the remote command server of Problem 2, lab2, to implement a networked file server. The client, myfetchfile, is executed using

% myfetchfile filename srv-ip srv-port

where filename is a string specifying the name of a regular file in the file system of a file server. For simplicity, filename should contain alphanumeric characters only (no space or special characters). For example, mytestfile is a valid file name. my\ testfile is not. The client creates a new file under /tmp of the same file name appended by the machine name and user name. For example, /tmp/mytestfileescher02joe123 if the client is executed on escher02 and user name is joe123. If a file of the same file name already exists, myfetchfile overwrites the existing file. /tmp is a local file system mounted on the disk of the lab machine where myfetchfile is executed. Your home directory will reside on a remote file system of a server machine that is mounted on the local file system of every lab machine. NFS (Network File System), a legacy networked file server, manages the remote file system using network programming whose influence we aim to minimize. srv-ip and srv-port are the IP address (dotted decimal) and port number of the file server. The client lets the operating system choose a default IP address and unused port number. The client performs a read() system call using the buffer size, MAXBUFSZM, set to 1000000 (bytes) using #define in the third argument of read() (i.e., count). However many bytes are returned by a call to read() is the number of bytes written using write().

The server, myfetchfiled, is executed using

% myfetchfiled blocksize srv-port

where blocksize is the number of bytes (attempted) to be read by a single read() system call. As with the client, the server maintains a buffer of size MAXBUFSZM and blocksize, if it exceeds MAXBUFSZM, is set to MAXBUFSZM. The number of bytes to be written by a single write() system call is the number of bytes read from the preceding read() system call. srv-port is the port to be used to listen for client requests. The server accepts connection requests from any of its IP configured network interfaces.

1.2 Client/server interaction and message format

Run the server first using a port number of your choice. If the port number is busy, try a different one. The server after performing bind() and listen(), blocks on accept(). When a connection request is received, the server forks a child process and lets the child handle the client's request. The parent goes back to blocking on accept(). The client, myfetchfile, transmits the characters of filename in its command-line argument using write() ending with '\0' to indicate the end of file name to the server. The client then waits for a control message from the server. The child process at the server, upon receiving the filename from the client, checks whether it exists or not. If it does not exist (or is otherwise inaccessible), a byte containing ASCII character '0' is returned. If the file exists but its size is 0 (file is empty), a byte containing character '1' is sent to the client. Otherwise a byte containing ASCII character '2' is returned to the client. After sending the control message, the child process of the server performs read() using blocksize as size argument followed by write() in a loop. After the last byte has been transmitted, the child process closes its end of the connection and terminates.

If the child closing the connection using close() causes irregularities (e.g., read() at the client returns 0 without returning all the bytes from write() system calls at the server), then try the following remedies: (a) call shutdown(sd, SHUT_WR) before calling close(sd) at the child process, (b) call shutdown(sd, SHUT_WR) followed by read(), and if read() returns 0 (i.e., client has closed its connection), then call close() at the server, (c) set a timer in the child process at the server (several seconds) and call close() when the timer expires, (d) terminate child process without calling close(). None of the above (and other methods not listed) are guaranteed to work in our version of Linux nor any other operating system where TCP is implemented. This is a flaw of the TCP specification and its varied kernel implementation whose roots we will discuss in the context of TCP connection set-up and termination.

The client, upon receiving control message '0' from the server prints a suitable message to stdout and terminates. Upon receiving '1', a suitable message indicating that the file is empty is output, and the client terminates. Upon receiving '2', myfetchfile calls gettimeofday() to take a start time stamp. Subsequently, the client performs repeated blocking read() system calls until it finds that the server has closed its side of the connection which is taken to mean that file transmission has ended. After writing the last byte of the received file, the client calls gettimeofday() to take an end time stamp. The difference with the start time stamp we will view as completion time. The number of bytes transferred divided by completion time is the average speed (bps) of file transfer. Before terminating, myfetchfile prints to stdout completion time (msec), speed (bps), and file size (bytes). Unlike Problem 2, lab2, the server does not randomly drop client requests and the client does not time out after 2 seconds to retransmit requests.

1.3 Testing

Test your file server application on our lab machines with the server running on a pod machines and clients running on escher machines. Put your code in v1/.

(i) *Basic function*. Set blocksize to 1472. Create a dummy file (or use an existing file) under /tmp of your server machine where myfetchfiled will be run. Its size, when testing with a single client, should require completion time in the 3 second range. Perform the experiments three times to get a sense of how stable the completion time and throughput numbers are. Although the number of tranferred bytes output by the client should give a reasonable indication that the bytes of the file were reliably transferred, double-check using diff that the received file has indeed the same content as the sent file. Verify that your app works with empty files. Discuss your findings in Lab3Answers.pdf (place under v1/).

(ii) *Multiple clients*. Run the server with 2, 6, 10 clients, and check what happens to their performance. Discuss your findings in Lab3Answers.pdf.

(iii) *Block size*. Run the single client set-up of (i) by varying blocksize. Use values 500, 1000, 5000, 10000 bytes. Is there an optimal blocksize value? Discuss your findings in Lab3Answers.pdf.

Problem 2 [80 pts]

2.1 Adaptive RTT estimation

Modify the stop-and-wait file transfer application of Problem 3, lab2, so that timeout is not fixed by adaptively adjusted based on measured RTT. For every data packet, gettimeofday() is called before sending the packet to record a start time. Upon receiving an ACK for the data packet, gettimeofday() is called to record an end time. Their difference (in microseconds) is taken as the new RTT estimate, newRTT. Assuming a current (i.e., old) RTT estimate curRTT is given, an updated curRTT estimate is computed by taking a weighted average

curRTT = A * curRTT + (1 - A) * newRTT

where 0 < A < 1 is a weighting parameter. Use #define to set A, call it RTT_weight, to 0.7. Thus more weight is assigned to the current RTT estimate so that fluctuations inherent in new RTT estimates is gradually incorporated. Unlike Problem 3, lab2, interprete the timeout command-line argument of myftpd as an initial RTT estimate (i.e., initial value of curRTT), not as the actual timeout used by the server to retransmit data packets. Set timeout as 1.2 * curRTT.

2.2 Matching problem

When performing RTT estimation, care has to be taken to consider the impact of missing ACK packets and outlier RTT values. If an ACK is missing or delayed, the event is detected at the server by raising of the SIGALRM signal. In the case of missing ACK, no update to curRTT is made. In the case of delayed ACK, newRTT needs to be computed since the network system may be experiencing increased end-to-end delay that should be incorporated into curRTT. To do so, the delayed ACK must be matched to its corresponding data packet since the latter has been retransmitted. That is, the "delayed" ACK may actually be the ACK transmitted by the client in response to the retransmitted data packet. In which case, the start time (from gettimeofday()) of the retransmitted data packet should be used to calculate newRTT. Otherwise the start time of the preceding data packet (which could itself be a retransmission in case of consecutive ACK drops) should be used to calculate newRTT. Propose your own method for handling the issue and describe it in Lab3Answers.pdf. Implement your solution accordingly.

2.3 Testing

Rerun test case (iii) ("jumping the gun") of Problem 3, lab2, for type B (server and client run on pod and escher machines) with initial RTT (i.e., curRTT) set through the timeout command-line argument of myftpd as 5000 microseconds. Define a debug parameter, #define RTTPRINT, in header file myftpd.h so that when the value of RTTPRINT is 1 new calculations of newRTT, curRTT (old), and curRTT (updated) are output to stdout. If RTTPRINT is set to 0, tracking changes to estimated RTT is disabled. Rerun the above test case with dropwhen set to 10. Discuss your findings in Lab3Answers.pdf. Deposit your code in v2/.

Problem 3 [50 pts]

3.1 System set-up

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 10 -w - > mylogfile

which will capture 10 Ethernet frames and save them into mylogfile. tcpdumpwrap-veth0 is a wrapper of tcpdump to allow sudo execution. You may capture more frames as necessary (see 3.2), and use additional options to affect how Ethernet frames are captured. Check the man page of tcpdump for available options. To generate traffic arriving on veth0, use the stop-and-wait file transfer app of Problem 3, lab2, with client myftp running on a pod machine bound to IP address 192.168.1.1. The server, myftpd, is executed on the same machine using

% veth 'myftpd filesize blocksize timeout 192.168.1.1 cli-port'

where veth, similar to our remote-command execution app (i.e., Problem 1, lab2) remote executes myftpd at a machine with IP address 192.168.1.2. Thus the server transmits packets from interface 192.168.1.2, and the client receives traffic through interface 192.168.1.1. 192.168.1.1 is a private IP address (it is not an address that is routable on the global 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 reachable over the IP Internet. Therefore we use dummy/virtual interfaces in Linux that allows veth0 to be configured as a separate (but virtual) Ethernet interface 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 myftpd on a different physical machine equipped with an Ethernet interface veth0 with IP address 192.168.1.2. Instead, both sender myftpd and receiver myftp 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 packet sniffing and inspection exercise, this will suffice. Monitoring WiFi traffic in the wild can be done in the Bonus Problem.

3.2 Traffic capture and analysis

Use a small filesize and blocksize 30 so that at least 5 data packets and 5 ack packets are captured by tcpdumpwrap-veth0. After capturing 10 Ethernet frames in mylogfile analyze their content 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 (wireshark is also a capture tool) captured network traffic in pcap format. Use wireshark or tcpdump to inspect the 10 (or more) 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 (i.e., generated by the stop-and-wait file transfer app) Ethernet frames by matching the source and destination MAC addresses of the captured Ethernet frames and content of the type field. The latter, for Ethernet II (i.e., DIX) frames, should identify IPv4 (0x0800) as the payload type of the Ethernet frame.

Ignoring the 20 bytes of payload occupied by IP header and 8 bytes by UDP header, identify the 1-byte application layer header containing the sequence number of our file transfer protocol. The rest of the Ethernet payload should contain 30 bytes of the ASCII character '3' if it is a data frame. In the case of an ACK packet, the stop-and-wait file transfer protocol inserts a single byte containing the sequence number. Given the minimize Ethernet frame size (and associated minimum payload size) requirement we discussed in class, determine if padding was applied to Ethernet's payload. Lastly, in the stream of captured Ethernet frames, identify the last two stop-and-wait application packets containing sequence number 2 which signals end of file transmission. Discuss your findings in Lab3Answers.pdf.

Although wireshark (or tcpdump) will provide output where Ethernet header fields (MAC addresses and type) are decoded, inspect the captured raw data in hexadecimal form to identify the bytes comprising source/destination MAC addresses, type field, 1-byte application layer sequence number, and 30-byte application layer payload (for data packets). In your write-up show the hexadecimal output (you can copy the raw data or use a screen capture) and your interpretation of the captured data.

Bonus Problem [20 pts]

Use a host with IEEE 802.11 WLAN interface running wireshark to sniff 802.11 frames. Wireshark is ported to most computing platforms including Windows, Linux, MacOS, and UNIX operating systems. The most important variable is whether the WLAN interface hardware and kernel driver support monitor mode. Many do, some don't. You can try to find relevant documentation for a specific hardware interface and kernel driver. The simplest method is to install wireshark (or other sniffing software) and check that it works. In the case of wireshark, if working on your own system you will need to install the software with administrator/root privilege. The same goes when running the software.

To sniff 802.11 frames, two methods are possible. In the first method, a WLAN interface is put in promiscuous mode (as in Ethernet frame sniffing in Problem 3) which translates the captured 802.11 frames into 802.3 Ethernet frames discarding all the 802.11 specific information. That is, captured 802.11 frames will appear as Ethernet frames. This is the default mode in wireshark which can be verified by selecting Capture -> Options -> and turning on the Promiscuous flag for the WLAN interface. The second method allows for true WLAN sniffing by forwarding detailed information of captured 802.11 frames that can be inspected through wireshark's interface. To do so, select Capture -> Options -> and turn on the Monitor flag (in addition to the Promiscuous flag). If you have a Windows/Linux/MacOS/UNIX laptop or PC with a WiFi interface that supports monitor mode, install wireshark as noted above (takes a couple of minutes) and you are ready to capture WiFi traffic. If you do not have access to such a device, I will leave the MacBook Air used in the Bonus Problem of lab1 with Tinghan so that you can use it to capture WiFi traffic during his PSO or office hours.

When performing capture, do it in HAAS or LWSN by first opening a terminal window and performing ping to one of the lab machines. Use ifconfig -a (on Windows you can use a command window or inspect the address via the control panel) to determine the MAC and IP addresses of the laptop. First, run in promiscuous mode only and check if you can detect the ping packets (ping uses IPv4 with special management payload called ICMP which will show up in the protocol field of wireshark) and select a captured frame and inspect its MAC and IP addresses, type field, and any other relevant information. You may even run the stop-and-wait file transfer app of Problem 3, lab2, while performing traffic monitoring but that is not necessary. Discuss your findings in Lab3Answers.pdf contrasting against your findings in Problem 3.

In the second capture experiment, put the device in monitor mode as noted above and capture 802.11 frames, focusing on Beacon frames. The structure of 802.11 frames is far more complex than Ethernet frames. Inspect 5 different Beacon frames (i.e., broadcast from different access points) and inspect relevant information such as SSID, signal strength of the captured frame, source and destination MAC addresses, carrier frequency/channel used, PHY layer coding (e.g., OFDM), amount of FEC (forward error correction) applied to the frame, and using time stamps estimate their Beacon periods. This will reveal basic information that is exposed when engaging in WiFi communication. Discuss your findings in Lab3Answers.pdf.

The Bonus Problem may be done as group (up to three students). If this is the case, please specify with whom you have collaborated in Lab3Answers.pdf. The write-up should be in your own words.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Go to the parent directory of the directory lab3/ where you deposited the submissions 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]