CS 422 (Park) **Assignment IV** **Due: Oct. 30 (Sun.), 2011**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 11pm on the due date. You may use any number of word processing software (e.g., Framemaker, Word, LaTeX), but the final output must be in pdf format that uses standard fonts (a practical test is to check if the pdf file prints on a CS Department printer). For experiments and programming assignments that involve output to terminal, please use* `script` *to record the output and submit the output file.*

PROBLEM 1

Read chapters 15–17.

PROBLEM 2 (50 + 30 pts)

(a) The purpose of this problem is to introduce sniffing raw data from a wired link, in particular, Ethernet. The wired Ethernet link connects a pair of NICs on two PCs in **LWSN B158**. For example, `sslab01` has a dedicated link to `sslab02`, similarly for `sslab03` and `sslab04`, and so forth, with the last pair being `sslab23` and `sslab24`. "Dedicated" means that the pair of Ethernet NICs on `sslab01` and `sslab02` form a stand-alone point-to-point Ethernet LAN (i.e., no switches involved) where the two NICs are configured with private IP addresses. Use `ifconfig` to check the interfaces and their properties on the `sslab` machines. You will find that each of the `sslab` machines is dual-homed, with a separate NIC (*eth0*) configured with a public IP address that connects to the Internet through an Ethernet switch. You will run apps that send traffic between a pair of PCs connected by their dedicated Ethernet link which is sniffed by an app, `tcpdump`, that runs on both sender and receiver machines. Running `tcpdump` requires root privilege since the NIC being used for sniffing must be put into promiscuous mode. As discussed in class, *setuid* is used to make an instance of `tcpdump` run with root privilege when you (as normal user) execute it with `sudo`.

To sniff at sender or receiver PCs, execute the following command which captures Ethernet frames seen by the NIC on the dedicated Ethernet link:

```
% sudo /usr/local/etc/tcpdumpwrap-eth1 -c15 -wEX
```

The command will capture 15 packets from the point-to-point Ethernet link and save them in the file `/var/tmp/`*login-EX* where *login* is your login ID. Generate traffic on the private network by doing `ping` from sender machine to receiver machine (`ping -c count -s 10 receiver-addr`) where `count` is the number of packets transmitted. Since each PC is dual-homed, to distinguish the dedicated NIC for the point-to-point link from the general-purpose NIC used to connect to the Internet, the symbolic name "here" has been set up to refer to the private IP address of the dedicated NIC. Similarly for "there" which translates to the private IP address of the receiver PC's dedicated NIC. For example, at host `sslab01`, executing `ping there` will generate ping packets going out on its second Ethernet NIC (*eth1*)—which has private IP address 172.16.25.101—that arrive on `sslab02`'s secondary NIC which has IP address 172.16.25.102. Choose a value of `count` so that `tcpdumpwrap` captures at least 15 packets. Submit the log file in hexadecimal format. Check if the log files generated at sender and receiver PCs are identical. Save your `ping` terminal interaction using `script` and submit it along with the hexadecimal `tcpdumpwrap` log file.

Using the Ethernet header format discussed in class, decode from the hexadecimal dump what the values for the fields in the Ethernet header are. Do it manually (i.e., visual inspection), not by automated tools. Compare the Ethernet addresses that you have captured with those given by `ifconfig`. From the value of the type/length field determine whether the Ethernet frames are DIX or IEEE 802.3. Explain how you are able to distinguish between the two and what the specific value found indicates. Determine if any padding is being done in the Ethernet frame to satisfy its minimum payload requirement. In the case of DIX frames where there is no length field, how can the payload be distinguished from padding? In Ethernet's payload, locate the 20 byte IP header (`ping` implements a protocol called ICMP which uses IP) and find the values for the source and destination IP addresses which make up the last 8 bytes. Compare the values against those obtained by `ifconfig`.

(b) An anonymous `ftp` server (i.e., daemon) has been set up that listens to client requests arriving on the dedicated Ethernet NIC of a PC in LWSN B158. Execute `ftp there`. When prompted for user name enter "anonymous"; when prompted for password press the `return` key. First, execute `ls` to check the content of the server's current

directory. Second, use `get` to fetch the file `CONTENTS`. Before executing `ftp`, run `tcpdump` at the server so that all packets exchanged in the `ftp` session over the dedicated point-to-point link are captured. Submit the log file in a format of your choice. Inspect the captured packets and their payload, and try to make sense of what you find. `ftp` runs on top of TCP which, in turn, runs on top of IP. This implies that the payload of Ethernet is IP (as in the `ping` example) whose payload, in turn, is TCP (header plus TCP's own payload). Can you extract the user name or password from the sniffed data? What about the contents of the subsequent client/server interaction (`ls` and `get`)?

PROBLEM 3 (50 + 30 pts)

(a) Rewrite the remote command client/server application of Problem 3(a), Assignment II, such that client and server, running on separate PCs, use TCP sockets in place of FIFO to communicate over a network. From a programming perspective, the internals of how TCP or IP work is not needed to use it. Functionally, TCP implements the sliding window protocol discussed in class and its header specifies source and destination port numbers needed to identify the communicating processes. As noted in class, like FIFO, a socket is one of seven file types in UNIX-like operating systems (including Linux) that is created by a `socket`() system call that returns a file (or socket) descriptor. TCP is selected by using the SOCK_STREAM option. 4-byte IP addresses are used to identify a destination machine—more precisely, the NIC on a PC in LWSN B158 configured with a public IP address—and 2-byte source and destination port numbers to specify client and server processes. Details of necessary system calls, options, and header files will be discussed during the PSOs and indicated in the "TA Notes" web link. Add/modify command-line arguments of server and client apps such that on the server a port number on which the server waits for client requests can be specified, and on the client the server's IP address and port numbers are given as input. Benchmark the TCP client/server application as before, recording the terminal interaction using `script`. Perform an additional benchmark by running two clients, one from a machine in LWSN B158 and the other from a PC in LWSN B148. To do so, use `ssh` to run remote shells in separate windows on the same PC. Show the output from `script`.

(b) A variation of part (a), modify the client/server apps and their command-line arguments (IP addresses and port numbers should not be hardcoded) such that server and client can be run on a pair of PCs in LWSN B158 with dedicated Ethernet NICs that are configured with private IP addresses. The client side is straightforward in that the command-line argument needs to specify the private IP address of the server (i.e., its dedicated NIC) which will prompt the kernel to send out packets on the client PC's dedicated NIC, determined by an internal route table look-up that indicates that packets with the the server's private IP address as their destination must exit on the client's dedicated NIC. The potentially non-straightforward component is the server side implementation where it must be able to receive client requests arriving on the NIC of interest (in this instance the dedicated NIC with private IP address). This issue was alluded to in Problem 2(b), Assignment II, but here you are asked to determine how this is best handled in the Linux kernel running in LWSN B158—discuss your findings in a separate write-up—implement it, and show that it works by running appropriate benchmark tests.

PROBLEM 4 (80 pts)

Under /u/u3/park/pub/cs422, you will find `talk_client.c` and `talk_server.c`, which are simplified implementations of `talk`, a precursor of today's messaging applications. The talk messaging programs use a protocol called UDP (SOCK_DGRAM option in `socket`()) which, unlike TCP, does not provide reliable communication using sliding window. UDP is a minimalist protocol that, in essence, only provides the ability to specify source and destination port numbers which IP does not. When a UDP packet is lost, it is lost, unless recovery mechanisms such as sliding window are implemented in the application layer. From a programming technique perspective, the main feature of interest in `talk` is that you be able to see (i.e., read) what the other party is writing to you in the midst of your own write. This means that when you write—i.e., the program reads the characters you type on the keyboard—packets arriving from the other party carrying messages must be handled asynchronously and displayed on the terminal. Hence, a solution where write followed by read is repeated in an infinite loop is inadequate. To achieve asynchronous I/O, a SIGPOLL (for packet arrival events) signal handler must be written and registered with the kernel. This is the counterpart to SIGIO for FIFOs used in Problem 4, Assignment III.

The client/server talk example is incomplete in several respects. First, the syntax for turning a file descriptor into asynchronous mode uses System V UNIX (e.g., Solaris) convention which may need be modified to match the syntax suited for Linux. This is one of the OS dependent aspects of network programming that must be carefully handled when porting code across different UNIX-like platforms. Second, `talk` is by its very nature a peer-to-peer (P2P) application where there is no distinction between client and server: they are one and the same. The code given

has client and server versions which must be done away with—i.e., integrated into a single application code, call it `mtalk`. Third, the SIGPOLL handler is missing. Writing a SIGPOLL handler is straightforward but more important is an application design decision: how to display a newly arriving message on the same terminal so that what you are writing is not scrambled with what you are receiving. One old style solution—which you will not be using—uses the `curses` library to split a tty terminal in two so that what you write appears on the top half and what you read appears on the bottom. A more modern solution uses windowing platforms such as X Window System which harnesses the full power of graphical output (you can draw or write anything, anywhere). We will not use X Windows (or any number of windowing platforms) since they are part of GUI programming, not network or system programming, our focus. Instead, you will reuse what you have already learned—the FIFO command execution server which will be modified to display messages received by the SIGPOLL handler in a separate window. By "window" we mean a second shell window provided by your Linux run-time environment—no GUI programming involved—wherein you display the received messages handled by a FIFO-based server process, call it `show_received_messages`. `show_received_messages` is a helper process subservient to the talk process, `mtalk`, where `mtalk` is executed in a separate shell window. Its SIGPOLL handler—after receiving a message contained in a UDP packet—forwards it through a FIFO to `show_received_messages`. Thus from the viewpoint of FIFO IPC, `mtalk` is the client (i.e., writer to the FIFO) and `show_received_messages` is the server (i.e., reader from the FIFO) that displays messages forwarded by `mtalk` in the shell window where it runs. This way, the messages you receive will not be interleaved with the messages you type since they are output and input in separate shell windows.

Modify and complete the provided code and get it to work by solving the three problems. The convention for executing `mtalk` should be

```
% mtalk hostname port-number
```

where `hostname` and `port-number` are the coordinates of the party you wish to talk to. The port number is optional so that if it is not specified then the messaging app uses a default port number of your choice. Test your talk application on two hosts and record the interaction using `script`. In this instance, you are talking to yourself. At the start, make sure that `mtalk` and `show_received_messages` are running on both hosts before initiating actual text messaging. This is to prevent potential system crash resulting from the other party not being online when you type a message. Production code, of course, should handle such situations appropriately without causing an application crash. Finally, find another student or friend and test your P2P messaging app on two PCs, one in LWSN B148 and the other in LWSN B158. Record the output using `script` on both ends.