CSE 5462: Lab 3 (Total: 100 points)

Demo in CL 112: Sep 15 (Thursday)
Electronic Code Submit Deadline: 9pm, same day

Go through the instructions on using *troll* from the project handout. Implement a file transfer program using UDP sockets, troll and a daemon (tcpd). Use MSS (number of data bytes in each packet) of 1000 bytes. In other words, headers such as TCP header and troll header are not included in 1000 bytes. Buffers/arrays in any process should not be more than 1 MSS worth of data. In order to avoid packet drops by troll, ftpc will transmit packets spaced out by a short period (say 10 ms). For example, the ftpc can *sleep* for that time after sending each 1000 bytes of data. First modify your file transfer program from Lab 2 to use SEND(), RECV() and BIND() function calls in place of send(), recv() and bind(). SEND() and RECV() will implement data transmission using UDP sockets. ACCEPT and CONNECT are null functions. When sockets are opened using SOCKET(), the ftps and ftpc processes create a socket for communicating with the tcpd running on the local machine. ftps will then block in RECV(), and ftpc will start sending data using SEND().

Note that the CAPITAL function calls are to be used only in ftpc/ftps only. In other processes you are supposed to use the standard UDP function calls. All CAPITAL functions need to have the exact same arguments as their counterparts.

SEND() and RECV() need to be compiled with ftps/ftpc. Put all capital function implementations in a separate file and compile it separately. Then link it with ftps or ftpc to create the executables. If you choose to, you can also compile the file containing the CAPITAL functions as a library.

Depending on which socket the packet comes in from in tcpd and its packet type (for packets from other processes within the machine), different parts of the code will be invoked in the client and server side. You can assume that all port numbers and IP addresses are known. It is recommended that you use the same executable "tcpd" in both sides, but it is not required.

ftpc/ftps are the application programs. tcpd is emulating the operation of the TCP module inside the OS. troll is emulating a real network.

The arguments for ftpc and ftps are same as in Lab 2. The application layer message format (4 bytes of file size, 20 bytes of filename, followed by the contents of the file) is also same as in Lab 2. Remember to use *htons, ntohs, htonl* and *ntohl* functions for IP addresses and port numbers when dealing with the sockaddr structure.

Here is a suggested way to implement it:

1. *ftpc* will read bytes from the file and use the function SEND() to send data to *ftps*. The SEND() function call will need to send these bytes to the local tcpd process.

2. Each tcpd process opens two globally defined ports, one for local communication and the other one for remote communication. The tcpd process at the client side machine waits for packets arriving on the local port and sends it off to the local troll process to deliver it to the remote tcpd process. In tcpd you can use "select" to process packets arriving on two sockets (one for external communication and one for internal communication). To distinguish different packet types, you can use a small header to identify the packet type (like say a 1 byte header). For example, tcpd can receive a packet from ftpc when SEND() is called or a request to send data from ftps when RECV() is called. The figure does not show the ports. (Later, in your project you will add various other functionalities to tcpd such as windowing, buffer management and timers in tcpd)

3. ftps executes the RECV() function and waits for data to arrive. The RECV function will send a packet to the local tcpd process to inform that it is waiting for data and the maximum amount of data that

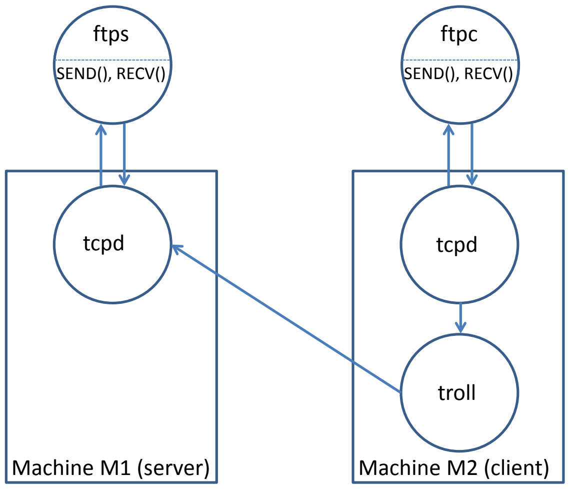

Figure 1: tcpd, troll, ftpc and ftps

is requesting to receive. When data comes in to the tcpd process at the server machine, it sends it to the right port on which ftps is waiting for data.

If you run troll with parameters to drop and garble packets, you will not receive the file correctly. If troll is running without any drops or garbling, UDP should not lose packets. You should decide on a packet format for all internal communication within the client or the server machines.

For this lab, you are not implementing any buffering in tcpd while sending or receiving.

The code must be well documented. And there should be a README file explaining the purpose of each file containing any C code in the directory. The README file should clearly list all the steps for compiling and testing your program. You can hardcode all IP addresses and port numbers, but do mention on which machine we should run your program to test them. Submit the code using the following command.

```
submit c5462aa lab3 <your directory>
```

The grading rubric is as follows:

- **Design: 60**
  ftpc 10
  ftps: 10
  SEND and RECV: 10
  tcpd client side: 10
  tcpd receiver side: 10
  Interfacing with troll:10

- **Program correctness and robustness (should work for any text file): 5**
- **Program correctness and robustness (should work for any binary file): 25**
- **Coding style (e.g., comments, indentations): 5**
- **Documentation (the README file): 5**

FAQ

Q:Can I use semaphores or threads in the program?
A: Alternate implementations are possible, but the use of semaphores and threads makes the code extremely difficult to debug. Stick to the specifications in the handout.

Q: In the description it says that "ACCEPT and CONNECT are null functions". Are these two functions something we are supposed to implement?
A: Yes, implement them just for the sake of completeness. But the implementation of these functions will be trivial as they will have nothing in it.

Q: It is mentioned that "All CAPITAL functions need to have the exact same arguments as their counterparts." Since we are implementing UDP and not TCP, do we need to hardcode the destination/ports for SEND and RECV into these functions? For example, do we need to hardcode the local port for tcpd into SEND?
A: You are implementing TCP using UDP as the underlying mechanism. SEND and RECV do not take the destination's IP address or port numbers. Assume that these are globally known constants. For example, the local port of tcpd can be a global constant.

Q: For tcpd, do we need to implement any TCP like functionality this time around or is it supposed to just receive and forward the packets?
A: just receive and forward, no TCP like functionality this time around.

Q: Are there any specific features of UDP to keep in mind?
A: Unlike TCP, UDP is not stream oriented. Think of UDP as sending chunks of data which may arrive out of order at the receiver, if they arrive at all. Think of TCP as a stream of bytes where the bytes will be delivered in-order and reliably to the application.