Project 1: Mobile TCP Proxy

When a host moves to a new network and obtains a new IP address, its ongoing TCP sessions will break because TCP uses source and destination addresses (and ports) to identify packets belonging to the same session. While a number of solutions have been proposed, most of them require changes to the system. In this project we will implement a solution that does not require any changes to the operating system or applications.

We use **telnet** as the sample application since it's easy to use. The goal is to establish a telnet session, change client's IP address, and ensure the same telnet session will still work. The solution you'll implement is applicable to any TCP application; telnet is just an example.

You will be given two virtual machines. Pick one as the ServerVM, and the other ClientVM. You need to know the IP addresses of the VMs. Look for the **eth1 IP address** as part of the system information displayed when you log into the VM.

This project is broken down into three milestones, bringing you from basic socket programming to the final solution.

Milestone 1: Implement a simple TCP client-server application.

Write a TCP client and a server passing messages between them. The message format is as follows.

Length of Payload
Payload

The length field is a 32-bit integer telling how many bytes the payload contains.

Your programs (client and server) should implement the following actions.

1. User starts the server program on ServerVM

./server sport

where *sport* is the port to which the server program will bind.

1. User starts the client program on ClientVM

   ./client sip sport

   where *sip* and *sport* is the server IP (eth1) and port number. The client establishes a TCP connection with the server.

2. User types a line of text on client's terminal and hits "return". The client program should read this line of text from stdin, using function calls like getline( ), fgetc( ), or others of your choosing. Note that the last *newline* character should not be counted. The maximum text length is 1024 bytes.

3. The client program makes a message that contains the one line of text as payload and the length of the payload. Sends it to the server program through the TCP connectivity.

4. The server receives the message, print two lines to the stdout, the 1^{st} line is the length of the payload, the 2^{nd} line is the payload text. Note that the server doesn't know how many bytes to read from the byte-stream supplied by TCP; so the server should read the length first, which is a 4-byte number, then read that number of bytes as the payload.

Deliverables:

- Source files client.c and server.c, and Makefile that will produce binaries "client" and "server".

Milestone 2: Implement TCP proxy programs

Get familiar with telnet, which will be used to test your programs. Run telnet on ClientVM to log into ServerVM:

telnet sip

where *sip* is ServerVM's eth1 IP address. After successful login, you should get a regular Unix shell session.

For this milestone, you will implement a client-side proxy ("cproxy) and a server-side proxy ("sproxy) that will relay traffic between the telnet program and telnet daemon. The specific actions to implement are as follows.

ClientVM ServerVM

1. Start sproxy on ServerVM. It should listen on TCP port 6200.

   sproxy 6200

2. Start the cproxy on ClientVM to listen on TCP port 5200, and specify the sproxy:

   cproxy 5200 sip 6200

1. On ClientVM, telnet to cproxy:

   telnet localhost 5200

The above command will have telnet open a TCP connection with cproxy. After accepting the connection, cproxy should open a TCP connection to sproxy. Upon accepting this connection, sproxy should open a TCP connection to the telnet daemon on localhost (address 127.0.0.1, port 23). Now, as shown in the above figure, cproxy and sproxy connect telnet and telnet daemon via three TCP connections.

In order for telnet to work, cproxy and sproxy should relay traffic received from one side to the other side. For example, cproxy has two connected sockets. It should wait for incoming data on both sockets. If either one receives some data, cproxy should read them from the socket, and send them to the other socket. Similarly, sproxy should operate the same way. Since both proxies merely relay data between their sockets, from user's point of view, telnet should just work the same way as if it directly connects to the daemon, e.g., type a Unix command and get results displayed on the terminal. Your cproxy and sproxy should implement the relay functionality to achieve this goal.

One technical question is, how do cproxy and sproxy wait for input from two sockets at the same time? You'll need ***select( )*** for this. See http://www.cs.arizona.edu/classes/cs425/spring16/select.htm for how to use select( ).

Another required feature is the closing behavior. When the user closes the telnet session, e.g., typing "exit", "logout", or ctrl-d, cproxy should close both of its connections, and sproxy should do so too. If recv( ) returns 0 or -1, it means the socket has been closed by the peer or it has an error. In either case the socket should be closed.

To isolate problems when debugging the programs, you can run telnet---cproxy---daemon first to debug cproxy, then telnet---sproxy---daemon to debug sproxy, and finally put them together as telnet---cproxy---sproxy---daemon and test.

Deliverables:

- Source files cproxy.c and sproxy.c, and Makefile that will produce binaries "cproxy" and "sproxy".

Milestone 3: Detecting and handling address changes

Let's first establish a telnet session, change IP address, and see what happens.

Telnet from ClientVM directly to ServerVM, then start a ping command (on ServerVM):

ping localhost

This is a trivial way to generate some continuous traffic. It keeps displaying the result of each ping at one-second interval with increasing icmp_seq number. The command is executed on ServerVM, but the results are transmitted to and displayed on ClientVM via the telnet session. As long as the session is working, you should see the ping result keeps showing up and there should be no gap in icmp_seq, i.e., no data loss.

Open another terminal on ClientVM, remove the Client's eth1 address, say, 1.2.3.4

sudo ip addr del 1.2.3.4/24 dev eth1

This removes the current IP address from the eth1 interface. You can use the following command to show the interface information before and after changes:

ip addr show eth1

Once the current address is removed, the ping results in the telnet session will stop immediately. Actually the ping program is still running on ServerVM, but because the TCP connection has been severed by the address removal, the results cannot be transferred to ClientVM. Similarly, no input from the ClientVM can be transferred to ServerVM, so the telnet session appears hanging.

Add a new address to Client's eth1, say 1.2.3.5:

sudo ip addr add 1.2.3.5/24 dev eth1

In our setup, the new address must have the same first 3 numbers as the old address. The last number of the new address must be chosen between 1 and 254, excluding the numbers already taken by the ServerVM and the old ClientVM addresses.

Because the existing telnet session was established using the old address, adding a new address doesn't resume the session, which will still be hanging, not responding to any key stroke. That's the problem we want to fix.

Cleanup: to terminate the hanging telnet session, type "ctrl]" then "ctrl d". You should also kill the ping process at the ServerVM.

Our goal is to resume the telnet session after the new address has been added, and do so without data loss. The idea is to use your cproxy and sproxy, and have a protocol between these proxies to detect and handle address changes.

With cproxy and sproxy, the telnet session consists of three TCP connections. When the ClientVM's address changes, it only breaks the connection ***between*** cproxy and sproxy; the other two connections are within a host and use address 127.0.0.1 (meaning localhost), which doesn't change no matter how a host moves. Therefore, the concern is only with the cproxy---sproxy connection.

1. In order to detect the address removal, cproxy and sproxy send each other a heartbeat message every 100ms. Both proxy programs treat the missing of three consecutive heartbeats as the indication of connection loss. They should close the disconnected sockets. **Use select( ) to implement the 100ms timer.**

2. cproxy should try connecting to sproxy again. When the new address is added, such a connection request will go through and the connection between cproxy and sproxy is reestablished using **new** sockets. Application data should now flow again. In our example, ping results will show up again on ClientVM's terminal.

   To ensure the new address is available before calling connect( ), you can put the address removal command and address add command in one line separated by a semicolon (;) and run them back to back quickly.

3. From sproxy's point of view, when a new connection with cproxy is established, there are two possibilities: (i) it is a brand new telnet session, or (ii) it is the continuation of a previous session after the ClientVM has got a new address. In the former case, sproxy should close the existing connection with telnet daemon, and open a new connection with the daemon to have a new telnet session. In the latter case, sproxy should keep using the existing connection to the daemon. To differentiate these two cases, cproxy needs to send a special message to sproxy right after they are connected, telling sproxy which case it is, and sproxy handles it accordingly.

4. There should be no data loss, i.e., no gap in icmp_seq, during the entire process. Data that were sent after the address removal but before the socket closure are not delivered. Therefore, cproxy and sproxy need a mechanism similar to TCP's sequence number and acknowledge number to keep track of who has received what, and also buffer data that have been sent but have not been acknowledged. After connection is established, they first exchange each other's acknowledgement number and retransmit data if needed.

To address all these issues, you need to define a packet format for messages between cproxy and sproxy. There are three types of messages: heartbeat, session initiation, and application data. Every message should have a header and a payload. The header may identify the type of the message, the length of the payload, the sequence and acknowledgement numbers, and any other information you think useful.

Based on the message format, you need to define the protocol actions between cproxy and sproxy, i.e., how each packet is being processed, and implement them.

A successful project should make a telnet session survive host mobility (i.e., address change) using your proxy programs. Optionally you may test it with other applications such as ssh.

Deliverables

1. A project report that documents the design of the packet format and protocol actions.
2. Source files cproxy.c and sproxy.c, and Makefile that can produce binaries called "cproxy" and "sproxy".

Grading

You can work by yourself or in a group of two students. No project group should have more than two students.

Your code must be written in C and use the C socket APIs.

The project will be graded on a different pair of virtual machines. Don't hardcode anything about your VMs in the source code.

Grading is based on functionality, i.e., what works and what doesn't, **not the source code**, i.e., what has been written. For example, when a required functionality doesn't work, its credit will be deducted, regardless of whether it's caused by a trivial oversight in the code vs. a serious design flaw.

The total score of the project is 100 points with the distribution of Milestone 1 (15 pts), Milestone 2 (25 pts), Milestone 3 (60 pts).

If Milestone 1 is submitted on time and passes all tests, a bonus of 5pts will be awarded.

If Milestone 2 is submitted on time and passes all tests, a bonus of 5pts will be awarded.

Submission

Only one submission per group.

1. Name your working directory “xx-yy”, where xx and yy are the VM numbers in your assignment.

2. Make sure this directory has all the source files and the Makefile. Include a README.txt file listing the names and emails of group members. Also for Milestone 3, include the project report in PDF.

3. Create a tarball

cd xx-yy

make clean

cd ..

tar -zcf xx-yy.tgz xx-yy

4. Upload xx-yy.tgz onto D2L dropbox.

Deadlines

Milestone 1: Wednesday February 10, at 5:00pm.

Milestone 2: Wednesday February 24, at 5:00pm.

Milestone 3: Wednesday March 23, at 5:00pm.

How to edit my source files on the virtual machines?

You can ssh into the virtual machine and edit the files locally using available editors, such as nano and vim.

If you want to use an editor on your own laptop/desktop, you can use sftp or scp or a sftp/scp GUI program to transfer the file to your local machine, edit it, then transfer back to the VM.

Some editors support sftp/scp plugins so the user doesn't have to do the file transfer; the editor will do it for you. For example, vim scp://yourvm/client.c will edit file client.c in your home directory on your vm.