CS60, Lab 5: Emulating a TCP client connection

Sergey Bratus, Spring 2017

Date due: Solutions to this lab will be due on Tuesday May 23 at 9pm. Try to finish this lab early, so that you can start on the final project!

The Task: In this lab, you will use Linux raw sockets to emulate the client side of a TCP connection. Your program will send raw TCP/IP packets to connect to a server on a given port,[1] then receive the packets sent by the server, reconstruct the stream sent by the server, and save it to a file. The stream will be sent by the server as soon as your client completes the 3-way handshake, and will contain a recorded response to an HTTP GET request, similar to that in Lab 4.

Your client should be able to work with external programs that listen on a known port. For example, it should be able to receive the file sent by `nc -l 8888 < somefile` on connection to port 8888. With a small modification of sending `GET /` followed by characters `0x0d 0x0a 0x0d 0x0a` (i.e., `"\r\n\r\n"`), it should be able to receive the default index pages served by an Apache webserver.

Your client must get the stream correctly, even in presence of random packet loss (which you will induce with a Netfilter command, same as in Lab 4).

What to implement: You should implement the following features of TCP on the client side:

- The relevant states and transitions of the TCP State Machine (see, e.g., `http://tcpipguide.com/free/t_TCPOperationalOverviewandtheTCPFiniteStateMachineF-2.htm`), namely, the Initiator Sequence parts that you left out in Lab 4.
- TCP acknowledgments for received stream and graceful connection termination (see, e.g., `http://www.tcpipguide.com/free/t_TCPConnectionTermination-2.htm`).
- The receiving part of the TCP sliding window (see, e.g., `http://www.tcpipguide.com/free/t_TCPSlidingWindowDataTransferandAcknowledgementMech-3.htm`, `http://www.tcpipguide.com/free/t_TCPSlidingWindowDataTransferandAcknowledgementMech-5.htm`). Specifically, your implementation should reject packets outside of its receiving window.

Your emulator will initiate a 3-way connection. You need not implement the Simultaneous Open or Simultaneous Close, but should be able to handle a FIN from the server correctly.

For extra credit, implement TCP Window Scaling and—for even more extra credit—implement SACKs.

Implement the following tasks.

Task 1. Implement your client for IPv4. Test your client.

Task 2. Modify your client to connect to a remote server over IPv6. Test your client: use `nc -6 ...` on your host to create a process listening for IPv6 connections, and connect to it from your VM.

Your VM's `eth0` and your host's `vboxnet0` interfaces already have autoconfigured IPv6 addresses derived from their respective MAC addresses. However, these addresses are too long to type. Since an interface can have multiple IPv6 addresses, you can set additional addresses for them that are much shorter. On Linux:

```
ip -6 addr add inet6 fe80::100 dev eth0
```

On MacOS:

```
ifconfig vboxnet0 inet6 fe80::1
```

[1] Cf. `http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentSequenceNumberSynchroniz-2.htm`)

You can then test IPv6 connectivity with these addresses by `ping6`. From the Linux VM: `ping6 -I eth0 fe80::1`, or `ping6 fe80::1%eth0` (same thing). From the host MacOS: `ping6 -I vboxnet0 fe80::100` or `ping6 fe80::100%vboxnet0`. Recall that an explicit option for the interface is required for pinging link-scope IPv6 addresses, because of potential ambiguity of automatically resolving them.

You can now test TCP IPv6 connectivity by running

```
nc -6 -l 8888 < somefile
```

on the host, and

```
nc -6 fe80::1%eth0 8888 > file
```

in the VM. The file you receive in the VM will be identical to *somefile*. Again, recall that IPv6 tools need an explicit interface specification for link-scope IPv6 addresses, hence `fe80::1%eth0` for netcat in the VM to reach the host.

Note that you will need to block RSTs from the kernel in IPv6 space for your program to work. The IPtables rule sets that apply in IPv4 and IPv6 are separate; IPv6 rules are managed by the `ip6tables` command rather that `iptables`, and are best checked with `ip6tables-save` command. Thus you will need to block RSTs *both* in IPv4 and IPv6. See Blocking RSTs below.

Task 3 (IPv6 extra credit). Find out the IPv6 address of `closet.test6.dartmouth.edu`. There is a web server listening at that address on port 80. Connect to that server over IPv6 with your client and retrieve the content served. There is a complication: the server will only respond if you connect to it with the TCP option specifying TCP MSS of exactly **1000**, otherwise you will get an ICMPv6 Prohibited packet, not the content. Note that TCP's normal response in case there's no one listening is a RST, not an ICMP* packet; if you see ICMP, you know a firewall is in the way.[2]

NOTE: Many DNS servers do not know how to serve IPv6 properly, and would time out or give you SERVFAIL when you ask them for AAAA records. Google's public servers, however, tend to handle these queries correctly. Use Google's public servers when in doubt.

NOTE: For this task, you will need external IPv6 connectivity. Installing Miredo (`apt-get install miredo`) is the easiest way to do this for your VM. Be aware that:

(a) Miredo relies on IPv4 connectivity, so test your access to the Internet with `ping` before you `ping6`; then test it with `ping6` before you do TCP. Also, check that the `miredo` process is alive and the interface `teredo` is up (i.e., shows on `ifconfig`).

(b) Miredo tends to lose the first couple of packets now and then. If a command or your program seems stuck, it may be Miredo's fault; restart your command or program.

See my notes on running Miredo.[3]

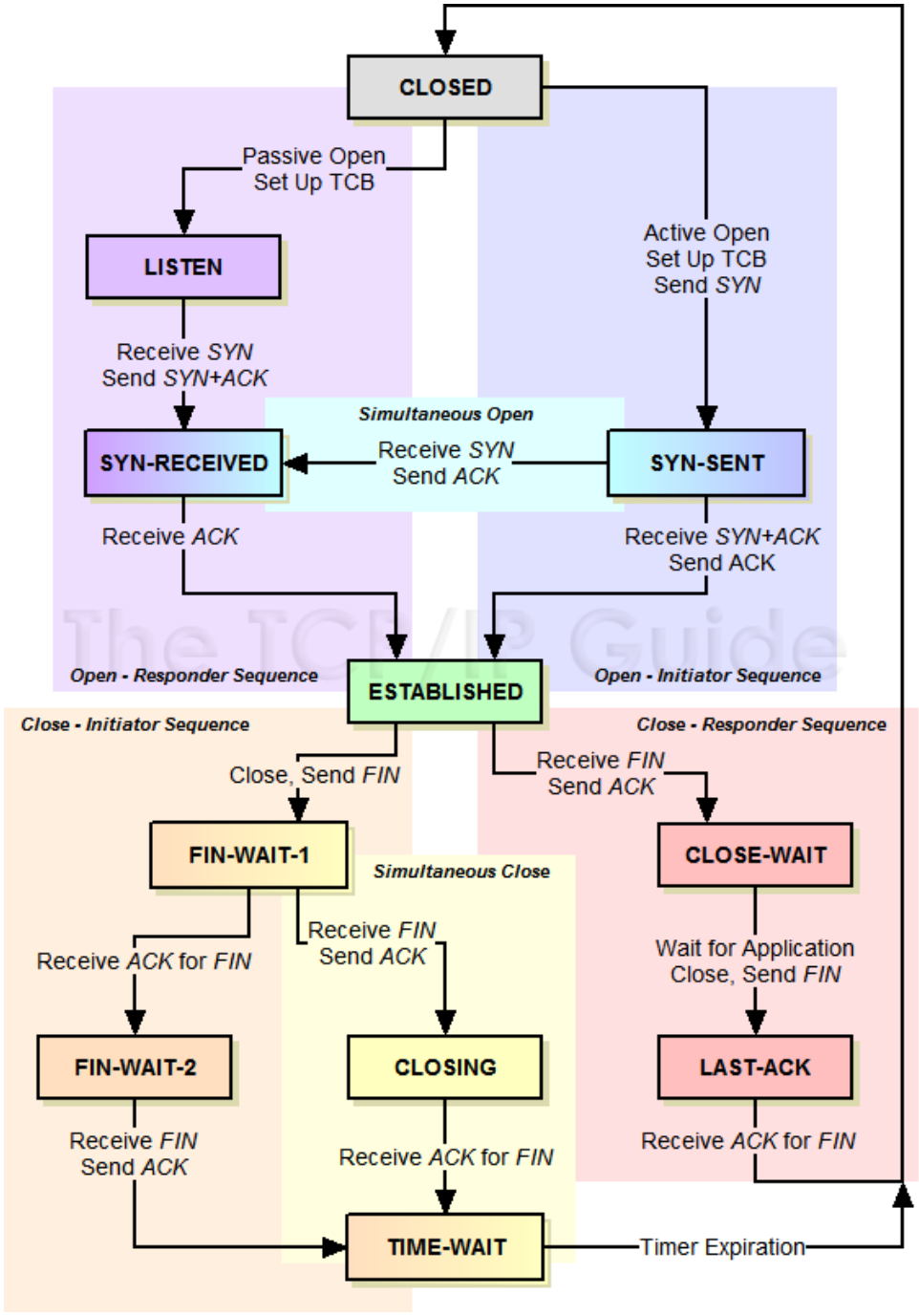
Simplifications: You may assume that any incoming connection gets the exact same response. You can use the file *http-jpg-response.txt*, which is a recorded response of an Apache web server containing an image of approximately 100Kb.[4]

Blocking RSTs: Your will need to block your VM kernel's responses to the TCP packets that, so far as it knows, do not belong to any connection it has established. Your VM kernel will consider them bogus, and will respond to each with a RST packet, which will break down your connection to the server. If any of these RSTs reach the server, it will appear to the server that you reset the connection. To block these RSTs (from just one port), use

[2] Another extra point for guessing my firewall commands to cause the above policy.

[3] `http://www.cs.dartmouth.edu/~sergey/cs60/miredo-notes.txt`

[4] If you run `nc -l 8080 < http-jpg-response.txt` on *tahoe*, and then point your browser at `http://tahoe.cs.dartmouth.edu:8080`, you should get this image displayed in your browser (provided that no one else is using port 8080). Even better: `wget -O img1.jpg http://tahoe.cs.dartmouth.edu:8080` will save this image to your disk; the correct MD5 sum for it is `e91108f934ad5ce5f2d2e245d02a9621`. Vary the host and port as needed.



The TCP Finite State Machine (FSM), Fig. 210 from `http://tcpipguide.com`

```
iptables -A OUTPUT -p tcp --dport 8080 --tcp-flags RST RST -j DROP
```

where the chosen port is port 8080. See notes from Lecture 9 for the meaning of this command.

For IPv6 in Task 2, you will need to add

```
ip6tables -A OUTPUT -p tcp --dport 8080 --tcp-flags RST RST -j DROP
```

Simulating Packet Loss: See Lab 4 for instructions.

Platform: Your solutions must compile and run in the provided Linux virtual machine (`cs60base` or `cs60mini`. You will *not* be able to develop or run them on CS Unix systems (because you don't have root privileges there). In theory, you could make it work on your MacOS, but the code developed on Darwin/MacOS would need to be substantially changed because the raw socket interfaces are different between the OSes, and behave differently. Ask me for hints on how to make it work on MacOS if you are interested.[5]

Submission: Your solutions must be submitted by checking them into the CS department LabGit system at `https://gitlab.cs.dartmouth.edu/`. In your `cs60` project, create a directory called `lab5` and work there. Don't forget to give access to your project to your section's grader and TAs (they must be able to see your code to grade it!)

Submit: Your `tcp-connector.c` for IPv4 in Task 1, `tcp6-connector.c` for IPv6 in Task 2, plus any code and logs for extra credit in Task 3 (if any). As usual, show your testing and all shell commands you used to find things out.

Your Virtual Machine: Your VM is the same that you configured for Lab 3, according to the instructions in `http://www.cs.dartmouth.edu/~sergey/cs60/lab3/vm-config/` and *vm-networking.pdf*.

Terms and conditions: You are allowed to use any external materials, printed or electronic. You are allowed to discuss problems and technical/C tricks, but the code you submit must be your own: you are not allowed to copy solutions from other students. Abide by the Honor Code; if in doubt, ask.

Late submissions: Throughout the course, you will get *two* free extensions for a late submission, each of 48 hours after a deadline. Once you've used these up, you will lose 10% points of credit for each late assignment, for each late day.

[5] Essentially, you will need to get raw packets via *libpcap* rather than raw sockets, and then block the kernel's own RST responses with PF, e.g., `block out inet proto tcp all flags R/R`.