Checkpoint 4: interoperating in the world

Due: Tuesday, February 20, 3 p.m.

0 Collaboration Policy

Collaboration Policy: Same as checkpoint 0. Please fully disclose any collaborators or any gray areas in your writeup—disclosure is the best policy.

1 Overview

By this point in the class, you've implemented the Transmission Control Protocol in an almost fully standards-compliant manner. TCP implementations are arguably the world's single most popular computer program, found in billions of devices. Most implementations use a different strategy from yours, but because all TCP implementations share a common **language**, they are all interoperable—every TCP implementation can be a peer with any other. This checkpoint is about testing *your* TCP implementation in the real world and **measuring** the long-term statistics of a particular Internet path.

If your TCP implementation is written properly, you may not have to write any code for this checkpoint. But despite our efforts, it is possible for bugs to escape our unit tests. If you find issues, you will probably want to debug with `wireshark` and fix any bugs. We would welcome (and reward!) any test cases you can contribute for the TCP modules to catch bugs that weren't caught by the existing unit tests.

2 Getting started

To get started:

1. Make sure you have committed all your solutions to Checkpoint 3. Please don't modify any files outside the top level of the `src` directory, or `webget.cc`. You may have trouble merging the Checkpoint 4 starter code otherwise.
2. While inside the repository for the lab assignments, run `git fetch --all` to retrieve the most recent version of the lab assignment.
3. Download the starter code for Checkpoint 4 by running `git merge origin/check4-startercode`. (If you have renamed the "origin" remote to be something else, you might need to use a different name here, e.g. `git merge upstream/check4-startercode`.)
4. Make sure your build system is properly set up: `cmake -S . -B buil d`
5. Compile the source code: `cmake --build build`

6. Reminder: please make frequent **small commits** in your local Git repository as you work. If you need help to make sure you're doing this right, please ask a classmate or the teaching staff for help. You can use the `git log` command to see your Git history.

3 Using your TCP implementation in the real world

Remember back in checkpoint 0 when you wrote `webget.cc`? You used a `TCPSocket`, a C++ class that models the Linux kernel's TCP implementation. Can you swap this out for *your* TCP implementation and still have a working Web getter?

1. Run `cmake --build build --target check_webget` to confirm that your `webget` still works. (If not, debug this before moving on.)

2. Make sure that your TCP implementation never sends debugging information to `cout`. This will get confused with the actual output of any program that uses TCP. It's fine to debug to `cerr`, but not `cout`.

3. After merging the checkpoint 4 starter code, go back to your `apps/webget.cc` file and change the `TCPSocket` to a `CS144TCPSocket`. This is a C++ class that wraps your TCP implementation to behave like a kernel-provided socket.

4. Run a script to give your TCP implementation permission to send raw Internet datagrams on the network:
   `./scripts/tun.sh start 144`. (N.B.: you will need to do this every time you reboot your VM.)

5. Now, rerun `cmake --build build --target check_webget`. If the test passes, congratulations! You have created a TCP implementation that successfully interoperates with arbitrary computers on the Internet.

6. If the test doesn't pass... try running the `webget` program by hand:
   `./build/apps/webget cs144.keithw.org /nph-hasher/xyzzy`

   If all goes well, the output will look like this:

```
$ ./build/apps/webget cs144.keithw.org /nph-hasher/xyzzy
DEBUG: minnow connecting to 104.196.238.229:80...
DEBUG: minnow successfully connected to 104.196.238.229:80.
HTTP/1.1 200 OK
Content-type: text/plain

7SmXqWkrLKzVBCEalbSPqBcvs11Pw263K7x4Wv3JckI
DEBUG: minnow waiting for clean shutdown... DEBUG: minnow inbound stream from 104.196.238.229:80 finished cleanly.
DEBUG: minnow outbound stream to 104.196.238.229:80 finished (0 seqnos still in flight).
DEBUG: minnow outbound stream to 104.196.238.229:80 has been fully acknowledged.
DEBUG: minnow TCP connection finished cleanly.
done.
```

   If not, time to debug with wireshark.

7. For debugging, you will also probably find the `./build/apps/tcp_ipv4` program helpful. This is a program that works like `telnet`, but uses your TCP implementation. In one terminal, run `./build/apps/tcp_native -l 0 9090` (this is the server, using Linux's TCP implementation). In another terminal, run `./build/apps/tcp_ipv4 169.254.144.1 9090` (this is the client, using your TCP implementation). If all goes well, the two peers will be able to communicate with each other until **both** peers end their outbound streams (by typing Control-D).

4 Collecting data

1. Choose a remote host on the Internet that has a RTT (from your VM) of at least 100 milliseconds. Some possibilities:
   - `www.cs.ox.ac.uk` (Oxford University CS department webserver, United Kingdom)
   - `162.105.253.58` (Computer Center of Peking University, China)
   - `www.canterbury.ac.nz` (University of Canterbury webserver, New Zealand)
   - `41.186.255.86` (MTN Rwanda)
   - *preferred:* an original choice with an RTT of at least 100 ms from you
2. Use the `mtr` or `traceroute` commands to trace the route between your VM and this host.
3. Run a `ping` **for at least two hours** to collect data on this Internet path. Use a command like `ping -D -n -i 0.2 hostname | tee data.txt` to save the data in the "data.txt" file. (The `-D` argument makes ping record the timestamp of every line, and `-i 0.2` makes it send one "echo request" ICMP message every 0.2 seconds. The `-n` argument makes it skip trying to use DNS to reverse-lookup the replying IP address to a hostname.)
4. Note: A default-sized ping every 0.2 seconds is fine, but please do not flood anybody with traffic faster than this.

5 Analyzing data

If you sent five pings per second for two hours, you will have sent approximately 36,000 echo requests ($= 5 \times 3600 \times 2$), of which we expect the vast majority to have received a reply in the ping output. Using the programming language and graphing tools of your choice, please prepare a report (in PDF format) that contains at least the following information:

1. What was the overall delivery rate over the entire interval? In other words: how many echo replies were received, divided by how many echo requests were sent? (Note: ping on GNU/Linux doesn't print any message about echo replies that are **not** received. You'll have to identify missing replies by looking for missing sequence numbers.)

2. What was the longest consecutive string of successful pings (all replied-to in a row)?
3. What was the longest burst of losses (all **not** replied-to in a row)?
4. How independent or correlated is the event of "packet loss" over time? In other words:
   - Given that echo request #N received a reply, what is the probability that echo request #(N+1) was also successfully replied-to?
   - Given that echo request #N did **not** receive a reply, what is the probability that echo request #(N+1) was successfully replied-to?
   - How do these figures (the **conditional** delivery rates) compare with the overall "unconditional" packet delivery rate in the first question? How independent or bursty were the losses?
5. What was the minimum RTT seen over the entire interval? (This is probably a reasonable approximation of the true MinRTT...)
6. What was the maximum RTT seen over the entire interval?
7. Make a graph of the RTT as a function of time. Label the x-axis with the actual time of day (covering the 2+-hour period), and the y-axis should be the number of milliseconds of RTT.
8. Make a histogram or Cumulative Distribution Function of the distribution of RTTs observed. What rough shape is the distribution?
9. make a graph of the correlation between "RTT of ping #N" and "RTT of ping #N+1". The x-axis should be the number of milliseconds from the first RTT, and the y-axis should be the number of milliseconds from the second RTT. How correlated is the RTT over time?
10. What are your conclusions from the data? Did the network path behave the way you were expecting? What (if anything) surprised you from looking at the graphs and summary statistics?

Please submit your report as a PDF via Gradescope.