Project 3: TCP Blacksburg
CS4254/5565 Fall 2022

v0.1-20221025

This project has three checkpoints, each with separate due dates. Checkpoint one is due at 11:59pm on Monday, October 31. **No late submissions will be accepted on CP1.** Checkpoint 2 is due at 11.59pm on Wednesday, November 16. Checkpoint 3 is due at 11.59pm on Wednesday, December 7.

1 Description

In this project, you will design a simple transport protocol that provides reliable datagram service and that implements a TCP Reno-like congestion control algorithm. Your protocol will be responsible for ensuring data is delivered in order, without duplicates, missing data, or errors. Since the local area networks at Virginia Tech are far too reliable to be interesting, we will provide you with access to a virtual environment that will allow you to emulate an unreliable network (e.g., high latency, packet loss, reordering, etc).

This project will proceed in three checkpoints.

- **CP1: Environment Setup.** Your first checkpoint is designed to ensure you have a working environment running on your machine.
- **CP2: A Reliable Transport Protocol.** For your second checkpoint, you will implement a simple reliable transport protocol.
- **CP3: Congestion Control.** For your third checkpoint, you'll add congestion control to your reliable transport protocol and analyze its performance in a variety of conditions.

CP1 should only take a few minutes to complete, but you should start immediately on this to ensure that if you do have issues we have time to help you solve them. Note that although CP2 does not require you to implement congestion control, you will want to consider how your design decisions in CP2 will simplify or complicate your ability to implement a CCA for CP3.

Throughout this assignment, you'll write code that will transfer a file reliably between two applications (a sender and a receiver). You do **NOT** have to implement connection open/close etc., and may assume that the receiver is run first will wait indefinitely for the sender to send the data to the receiver.

2 Requirements

2.1 CP1: Getting Set Up

In CP1, you must submit a screenshot of the hostname and of a ping to 127.0.0.1 from inside your VM or container environment (Section 4) with an RTT of at least 40ms. That's it!

2.2 CP2: A Reliable Transport Protocol

You will design your own packet format and use UDP as a carrier to transmit packets. Your packet might include fields for packet type, acknowledgement number, advertised window, data, etc. This part of the assignment is entirely up to you. Your code MUST do the following:

- The sender must accept data from STDIN, sending data until EOF is reached
- The sender and receiver must work together to transmit the data reliably
- The receiver must print out the received data to STDOUT in order and without errors
- The sender and receiver must print out debugging messages to STDERR
- Your sender and receiver must gracefully exit
- Your code must be able to transfer a file with any number of packets dropped, damaged, duplicated, and delayed, and under a variety of different available bandwidths and link latencies
- Your sending program must be named **4254send** and your receiving program must be named **4254recv**
- Datagrams generated by your programs must each contain less than or equal to 1472 bytes of data per datagram (i.e., the 1500 byte Ethernet MTU - the 20 byte IP header - the 8 byte UDP header)

Your reliable transport protocol must, by definition, be *correct*. You may implement any reliability algorithm(s) you choose to achieve this. However, for full credit, your algorithm should also be:

- Fast (*timely*): Require little time to transfer a file.
- Low overhead (*efficient*): Require minimal total data volume to be exchanged over the network, given data bytes, headers, retransmissions, acknowledgements, etc.

Correctness matters most; performance is a secondary concern. We will test your code and measure these two performance metrics; better performance will result in higher credit. Remember that network-facing code should be written defensively. Your code should check the integrity of every packet received. We will test your code by (for example) reordering packets, delaying packets, duplicating packets, corrupting packets, and dropping packets; you should handle these errors gracefully, recover, and not crash. We will not test scenarios in which it is impossible for reliable transport to terminate, e.g. all packets are dropped; you can assume that given sufficient time a correct reliable transport protocol should terminate successfully.

2.2.1 Performance

10% of your CP2 grade will come from performance. Your project will be evaluated based on performance relative to a series of benchmarks that include both the time taken to successfully transmit files to the receiver and the number of total bytes sent during the transfer.

To help you know how you're doing, the testing script (Section 10) will run a series of performance tests at the end. For example, you might see something like the following:

```
Performance tests
huge 5 Mb/s, 10 ms, 0% drop, 0% duplicate 0% delay        [DATAOK]
0.401 sec elapsed, 976KB sent
Rate: 19Mb/s                                              [ OKAY ]
```

The first line presents the parameters of the current test. [DATAOK] indicates that the file was delivered successfully, without any errors. [DATAERR] would indicate that the receiver printed out a file containing errors. The following lines present the performance characteristics of the file transfer, including how long it took, how much data was sent, and the overall transmission rate. [OKAY] indicates that performance in this case was acceptable, and would receive full credit. Alternatively, you might see output like this:

```
Rate: 1Mb/s                                               [PERF50]
```

[PERFXX] indicates that the performance was at XX% of our target benchmark. In this example, performance hit 50% of the desired benchmark. In these cases, credit would be awarded proportionately to the achieved performance level. [FAIL] indicates that performance was insufficient, and would not receive any credit for performance in this test.

2.3 CP3: Reliable Transport with Congestion Control

CP3 requirements are a superset of CP2; your solution must meet all of the requirements outlined above in Section 2.2. In addition, you must also implement a congestion control algorithm (CCA).

As discussed in class, a wide design space is available for congestion control algorithms. For this assignment, you must implement a TCP Reno-like CCA that incorporates the following elements:

- The CCA must incorporate a dynamically-sized sliding window
- The CCA must operate in at least two regimes, equivalent to slow start and congestion avoidance.
- The CCA's congestion window must grow super-linearly initially, similar to TCP Reno's slow start
- The CCA's congestion window must enter congestion avoidance mode after encountering loss.
- The CCA's congestion window must respond to loss by reducing the window size.
- The CCA's congestion window must respond to non-loss by growing.

You are strongly encouraged to adopt the design decisions made by TCP Reno as a baseline, but TCP Reno isn't the only correct solution. For full credit, your CCA should also be:

- Fast (*timely*): Require little time to transfer a file.
- Low overhead (*efficient*): Require minimal total data volume to be exchanged over the network, given data bytes, headers, retransmissions, acknowledgements, etc.
- Fair: Multiple simultaneous flows using your CCA should converge to using roughly the same percentage of the link.

As before, we will test your code by (for example) reordering packets, delaying packets, duplicating packets, corrupting packets, and dropping packets; you should handle these errors gracefully, recover, and not crash. We will not test scenarios in which it is impossible for reliable transport to terminate, e.g. all packets are dropped; you can assume that given sufficient time a correct reliable transport protocol should terminate successfully.

2.3.1 CCA Evaluation

The most important part of CP3 is your *evaluation* of your solution. The dynamics of CCAs are complex; rigorous evaluation under a variety of scenarios is typically the name of the game when it comes to developing a new CCA, so that's what CP3 is about. As we discussed in class, a CCA is simply an algorithm for determining a window size. "The window size is always 1" is a valid CCA, and is what the starter code we gave you implements. Of course, that CCA probably won't perform very well, so you should consider the approaches we discussed in class as you design yours.

For CP3, consider the following network scenarios:

1. **90's Internet**: 100ms latency, 56.6kbps bandwidth, 1% loss, 50Mb queue. For full effect:
   - Listen to this during testing: https://www.youtube.com/watch?v=r6UR_3ZieE4
   - Have someone yell at you to get off the Internet so they can make a phone call
2. **User to CDN**: 15ms latency, 100Mbps bandwidth, 0.001% loss, 500Mb queue
3. **Cross-country**: 100ms latency, 1Gbps bandwidth, 0.00001% loss, 1000Mb queue
4. **Data center**: 1ms latency, 10Gbps bandwidth, 0% loss, 50Mb queue
5. **Moon**: 1500ms latency, 1Mbps bandwidth, 0% loss, 100Mb queue

Pick one scenario from above as a starting point and empirically evaluate (with graphs!) the following questions about the performance of your CCA:

1. How does varying router buffer size impact your CCA's performance?
2. How does varying link bandwidth and/or delay impact your CCA's performance? Your window size?
3. How does varying loss rate impact your CCA's performance?
4. How does your CCA perform when varying the number of sender-receiver pairs sharing the same link? Do their bandwidths converge?

In addition, compare the performance of your CCA across at least three of the above scenarios. Characterize the performance of your CCA – are there some scenarios in which your CCA performs better than others? Provide evidence support your characterization. You'll need to pick metrics and design experiments to support your claims.

2.3.2 Evaluation Writeup

Based on your experiments, prepare a short report (no more than 6 pages) describing the behavior you expected to see from your implementation and what you actually saw. Your report should clearly define how your CCA works, including key design decisions you made, as well as a thorough discussion of how your CCA stacks up in terms of correctness, efficiency, timeliness, and fairness. Did anything surprise you? Are there scenarios in which your implementation performs better or worse than others? Be sure to fully describe how you evaluated your CCA: you should describe the scenarios in which you evaluated your CCA, what you measured, and what the results of your experiments were.

Claims should be backed with quantitative evidence in the form of graphs, tables, or other statistics gathered while running experiments to evaluate your CCA. As an example, for the baseline scenario you chose, your report must include at least the following analyses:

1. A graph where the X axis is time and the Y axis is the window size of each sender.
2. A graph where the X axis is time and the Y axis is the throughput per flow. Where possible, include a horizontal line demonstrating what the "ideal" throughput would be for each sender.

Your report must not simply be a list of graphs; you will be graded on how well you synthesize and explain the results you found and how convincing and thorough your evaluation of your CCA is. You are encouraged to work with your classmates (even those outside your group) to discuss other possible experiments worth running and to share the results of your experiments in non-editable, graphical form (no raw data or code).

2.3.3 Implementation Considerations

For production software, and for research, monitoring and evaluation are critical aspects of system design: once a system is in production, it's important to be able to monitor its performance and health over time. The right monitoring design can save time and heartache down the road. For this project, it's worth spending some time developing a strategy for how you're going to implement the metering and logging necessary for generating your graphs. We'll provide a script that provides per-second throughput measurements for pairs of senders and receivers. You'll need to log your congestion window sizes on your own, and you'll also need to develop your own pipeline for processing logfiles into graphs.

Rather than printing logs to STDOUT, consider generating an automatically-named log file and writing your congestion window size each time you adjust it. You may have many log files you generate during the course of your runs, so having a clear naming convention for your logfiles will help you keep organized.

The more automated your experiments can be (from execution to log file generation to graph creation), the fewer mistakes are possible. In an ideal world, you would run an experiment and

have graphs automatically generated for you after each run. Feel free to make changes to the runner scripts we provide to plug in to your graphing libraries of choice if you'd like to do this.

2.4 Extra Credit

You can earn extra credit in a number of ways on this project.

2.4.1 Performance.

We will track the best performing implementations submitted for both CP2 and CP3. If your implementation is among the top performing *correct* implementations submitted for each checkpoint, you may be awarded extra credit proportional to your performance, not to exceed 10 points. Feel free to share your performance results with your classmates to track how you're doing!

2.4.2 Congestion Control Bells and Whistles

You should see your CP3 implementation perform better in some scenarios than others – as we discussed in class, for example, classic TCP Reno doesn't perform well on links with high BDPs.

For extra credit, choose one (or more) scenarios (either listed above, or one you create), and develop a *new* CCA that performs better than your existing one. Characterize the performance of your new CCA and show how it differs from your original implementation. Are there cases where your original implementation outperforms your new one?

3 Your Programs

For both CP2 and CP3, you will submit two programs: a sending program **4254send** that accepts data and sends it across the network, and a receiving program **4254recv** that receives data and prints it out in-order. You may not use any reliable transport protocols in your project (such as TCP); you must use UDP. You must construct the packets and acknowledgements yourself, and interpret the incoming packets yourself.

4 Project Environment

For this project, you will be using a Vagrant environment to simulate an unreliable network. We have provided both a virtual machine (intended for Windows or Linux systems) or container (intended for all OSX systems, including M1 machines) as a testing and execution environment.

We provide installation instructions for your Vagrant environment in the relevant directory of your starter code. In general, you will need to install either Virtualbox or Docker and Vagrant on your local machine (i.e., not on the rlogin cluster), following the installation instructions provided on each's website.

Included in the starter code for the project is a file named `Vagrantfile`. **Do not modify this file.** This file contains the configuration necessary for launching your Vagrant instance. After you have installed Virtualbox/Docker and Vagrant, you can navigate (via the command line) to the directory where you've unzipped the starter project directory and run the following commands to launch your Vagrant instance:

```
$ cd win_x86 # or `cd osx` for the OSX container environment

# This downloads and configures the Vagrant instance.
# It may take a while the first time.
$ vagrant up
$ vagrant ssh # connects via ssh to the Vagrant instance
```

The special folder `/cs4254` (also symlinked to `/home/cs4254`) is shared between the host and your Vagrant instance; it's the folder on your host one level above where the Vagrantfile you're using is located. If you're using an IDE, any changes you make here are automatically visible inside your Vagrant instance. In general, you should always *execute* code inside your Vagrant instance, but you can modify it on your host (using any IDE you like).

If your Vagrant instance is in a bad state, you can run `vagrant destroy` to destroy it, and then just recreate from a clean slate from the steps above. Files on your host will not be affected by deleting and recreating the instance.

5 Language

You can write your code in whatever language you choose, as long as your code compiles and runs in the Vagrant instance without additional packages beyond those installed by our provided initialization scripts. If you need packages for your language, e.g. the Go toolchain, they must be installable using a single `apt` invocation; please provide us a list of the packages you need installed in advance for approval, and be sure your Makefile installs them during the build process for your code. You may use IDEs (e.g. Eclipse) during development, but do not turn in your IDE project without a Makefile (if needed). Make sure you code has no dependencies on your IDE.

6 Starter Code

Very basic starter code in Python for the assignment is provided. You may use this code as a basis for your project, or you may work from scratch. Provided is a simple implementation that sends one packet at a time; it does not handle any packet retransmissions, delayed packets, or duplicated packets. So, it will work if the network is perfectly reliable. Moreover, if the latency is significant, the implementation will use very little of the available bandwidth. To get started, download the starter code distribution into a local directory on your machine.

7 Program Specification

The command line syntax for your sending is given below. The client program takes command line argument of the remote IP address and port number. The syntax for launching your sending program is therefore:

```
./4254send <recv_host>:<recv_port>
```

- `recv_host` (Required) The IP address of the remote host in a.b.c.d format.

- `recv_port` (Required) The UDP port of the remote host.

The sender must open a UDP socket to the given IP address on the given port. The data that the sender must transmit to the receiver must be supplied in STDIN. The sender must read in the data from STDIN and transmit it to the receiver via the UDP socket. Your sender may print out any debug information that you wish, but it must do so to STDERR.

The command line syntax for your receiving program is given below. The receiving program takes a single command line argument of the UDP port number to bind to. The syntax for launching your receiving program is therefore:

```
./4254recv <recv_port>
```

On startup, the receiver must bind to the specified UDP port and wait for datagrams from the sender. Similar to 4254send, you may add your own output messages to your receiver but they must be printed to STDERR.

The receiver program must print out the data that it receives from the sender to STDOUT. The data that it prints must be identical to the data that was supplied to the sender via STDIN. In other words, data cannot be missing, reordered, or contain any bit-level errors.

8 Testing Your Code

In order for you to test your code over an unreliable network, you will use tools in your Vagrant environment to configurably emulate a network that will drop, reorder, damage, duplicate, and delay your packets. You will use the loopback interface in order to access this emulated network. In other words, you might run something like ./4254recv 3992 in one terminal, and then run ./4254send 127.0.0.1:3992 in another terminal.

You may configure the emulated network conditions by calling the following program:

```
$ netsim [--bandwidth <bw-in-mbps>] [--latency <latency-in-ms>] [--drop <percent>]
[--reorder <percent>] [--duplicate <percent>] [--limit <buffer-in-mb>]
```

- **bandwidth**: This sets the bandwidth of the link in Mbit per second. If not specified, this is 1 Mb/s.
- **latency**: This sets the latency of the link in ms. If not specified, this value is 10 ms.
- **delay**: This sets the percent of packets the emulator should delay. If not specified, this is 0.
- **drop**: This sets the percent of packets the emulator should drop. If not specified, this is 0.
- **reorder**: This sets the percent of packets the emulator should reorder. If not specified, this is 0.
- **duplicate**: This sets the percent of packets the emulator should duplicate. If not specified, this is 0.

- **limit**: This sets the simulated router queue size in megabits (not bytes!). If not specified, this is 1000Mb.

Once you call this program, it will configure the emulator to delay/drop/reorder/duplicate all UDP and ICMP packets sent by or to you at the specified rate. For example, if you called

```
$ netsim --bandwidth 0.5 --latency 100 --delay 20 --drop 40 --limit 80Mb
```

The simulator will (1) configure a network with 500 Kb/s bandwidth and a latency of 100 ms, (2) will randomly delay 20% of your packets and drop 40%, and (3) will simulate a buffer queue on the path of 80Mb (i.e., 10MB). In order to reset it so that none of your packets are disturbed, you can simply call

```
$ netsim
```

with no arguments. Note that your configuration will not persist across reboots of the Vagrant instance, but will otherwise remain in place until changed. In addition, you may notice worse performance (e.g., higher latency) than you have configured; this is normal, particularly if your machine is under heavy load.

9 Helper Scripts

We've included two helper scripts in your Vagrant instance to make testing your code easier, in addition to the `netsim` script for configuring the emulated network.

9.1 nettest

This script will launch your sender and receiver, feed the sender input, read output from the receiver, compare the two, and print out statistics about the transfer. This script is included in the starter code distribution and is already on your path in the Vagrant instance. Note that you must run this script from the directory in which your `4254send` and `4254recv` executables are located. You can run it as follows:

```
$ nettest
```

This script takes a few arguments:

```
$ nettest [–live] [–size {small,medium,large,huge}] [–timeout TIMEOUT]
```

- **size**: The size of the data to send, including 1 KB (small), 10 KB (medium), 100 KB (large), MB (huge). Default is small.

- **live**: Instructs the script to echo the STDERR output of 4254send and 4254recv. This may add significant processing time, depending on the amount of output.
- **timeout**: The maximum number of seconds to run the sender and receiver before killing them. Defaults to 30 seconds.

The output will display useful statistics about the transfer, where "Data match" refers to whether the data was transferred correctly:

```
$ nettest --size medium
Data match: Yes
Msg Size: 10000 B
Time elapsed: 174.982 ms
Packets sent: 23
Data on Wire: 13761 B
```

9.2 congestiontest

This script will run a number of pairs of your sender and receiver, while collecting logs on your CCA's performance. Note that you must run this script from the directory in which your `4254send` and `4254recv` executables are located. You can run it as follows:

```
$ congestiontest
```

This script takes a few arguments:

```
$ congestiontest [–live] [–timeout TIMEOUT] [–config filename]
$ congestiontest [–live] [–size {small,medium,large,huge}] [–timeout TIMEOUT] [–numpairs N]
```

- **size**: The size of the data to send, including 1 KB (small), 10 KB (medium), 100 KB (large), MB (huge). Default is small. This is ignored if you're using a config file instead.
- **live**: Instructs the script to echo the STDERR output of 4254send and 4254recv. This may add significant processing time, depending on the amount of output.
- **timeout**: The maximum number of seconds to run the sender and receiver before killing them. Defaults to 30 seconds.
- **numpairs**: The number of sender-receiver pairs to launch simultaneously. Defaults to 1. Cannot be used with the config file option.
- **config**: Path to a configuration YAML file, like that provided in infra/config.yaml. The format of the YAML file is described below.

The output of this script is a CSV-formatted logfile, stored in `logs` in the directory from which you run it, that provides useful statistics on the per-second throughput of your sender.

The configuration file can include a list of objects representing send/receiver pairs. For each pair, you can specify a filesize for the sender/receiver pair to use (small, medium, large, or huge) and an *offset* in seconds to wait before starting that specific sender/receiver pair (thus allowing you to stagger the start times of sender/receiver pairs). The provided `infra/config.yaml` file implements the third experiment from Section 2.3.1.

10 Testing

We have included a basic test script that runs your code under a variety of network conditions and also checks your code's compatibility with the grading script. The testing script is also included in the stater code. If your code fails in the test script we provide, you can be assured that it will fare poorly when run under the grading script. To run the test script, simply build your code and run

```
$ testall
```

This will test your programs on a number of inputs. Note that this testing script is not exhaustive and, like the helper script, must be run from the directory in which your `4254send` and `4254recv` executables are located.

11 Submitting Your Project

Checkpoint 1: Due Monday, Oct 31 @ 11.59pm

To turn in Checkpoint 1 of your project, you should submit the following via Gradescope:

- A screenshot of a ping to 127.0.0.1 from inside your Vagrant instance with an RTT of at least 40ms.

No late submissions will be accepted for CP1.

Checkpoint 2: Due Wednesday, Nov 16 @ 11.59pm

To turn in Checkpoint 2 of your project, you should submit the following via Gradescope:

- Thoroughly documented code as a .tar.gz or .zip file. This must also include a Makefile that compiles your code, producing two appropriately named executable files. Your Makefile may be blank, but it must exist.
- A plain-text (no Word or PDF) README file. In this file, you should briefly describe your high-level approach, any challenges you faced, and an overview of how you tested your code (you can also just type this directly into the submission text box).

Checkpoint 3: Due Wednesday, Dec 7 @ 11.59pm

To turn in Checkpoint 3 of your project, you should submit the following via Gradescope:

- Thoroughly documented code as a .tar.gz or .zip file. This must also include a Makefile that compiles your code, producing two appropriately named executable files. Your Makefile may be blank, but it must exist.
- A PDF file (not Word, plain text, or any other format) no more than 6 pages in length evaluating your CCA as described in Section 2.3. *Reports exceeding 6 pages or in a non-PDF format will receive no credit.*

If you are working in a group, submit one submission per group. **If you begin working on this project with a partner, you must submit all subsequent checkpoints with that partner.** If you are working alone, you may form a group for a later milestone. For example, if you submit CP1 individually, but decide to work with a partner on CP2, you must submit CP3 with that same partner; you would not be able to work with a new partner or by yourself for CP3. Your group may submit as many times as you wish for each part; only the last submission will be graded, and the time of the last submission will determine whether your assignment is late.

12 Grading

This project is worth 200 points in the Projects category of your grade, representing 20% of your final grade. Each checkpoint will be graded as follows:

- 5 pts: CP1 - Completion
- 75 pts: CP2 - Reliable transport correctness
- 10 pts: CP2 - Performance
- 10 pts: CP2 - Style and documentation
- 15 pts: CP3 - Reliable transport correctness
- 15 pts: CP3 - Implementation of a CCA meeting the requirements outlined in Section 2.3.1
- 70 pts: CP3 - Evaluation of your CCA as outlined in Section 2.3.2

By definition, you are going to be graded on how gracefully you handle errors; your code should never print out incorrect data. Your code will definitely see delays, duplicated packets, and so forth. You should always assume that everyone is trying to break your program.

Acknowledgements

This project (particularly CP2) is based off of a project originally developed at Northeastern University.