Project 4: Reliable Transport Protocol

Build something like TCP on top of UDP

This project is due at 11:59pm on Monday, March 11, 2024.

Description

You will design a simple transport protocol that provides reliable datagram service. Your protocol will be responsible for ensuring data is delivered in order, without duplicates, missing data, or errors. Since the local area networks at Northeastern are far too reliable to be interesting, we will provide you with access to a virtual machine that will emulate an unreliable network.

For the assignment, you will write code that will transfer a file reliably between two nodes (a sender and a receiver). You may assume that the receiver is run first and will wait indefinitely, and the sender can just send the data to the receiver.

Your Programs

For this project, you will submit two programs: a sending program `4700send` that accepts data and sends it across the network, and a receiving program `4700recv` that receives data and prints it out in-order. You must use UDP as the transport protocol. You may not use any other transport protocol libraries in your project (such as TCP). You must construct the packets and acknowledgements yourself, and interpret the incoming packets yourself.

Requirements

You have to design your own packet format and use UDP as the transport layer protocol. Your packet might include fields for packet type, acknowledgement number, advertised window, data, etc. This part of the assignment is entirely up to you. There are several things your code **must** do:

- 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 specified 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, duplicated, and delayed, and under a variety of different available bandwidths and link latencies
- Your sending program must be named `4700send` and your receiving program must be named `4700recv`
- You must also submit a `Makefile` that builds your client program; if you do not need to do so, you must submit a `Makefile` with a blank target (e.g., `all:`)
- Datagrams generated by your programs must each contain less than or equal to 1500 bytes of data. Any datagrams sent with more data that that will be dropped.

You may implement any reliability algorithm(s) you choose. However, more sophisticated algorithms (i.e., those that perform better) will be given higher credit. For example, some desirable properties include (but are not limited to):

- Fast: Require little time to transfer a file.
- Low overhead: Require low data volume to be exchanged over the network, including data bytes, headers, retransmissions, acknowledgements, etc.

Regardless, 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

Language

You can write your code in whatever language you choose, as long as your code compiles and runs on Gradescope. Do not use libraries that are disallowed for this project. Similarly, your code must compile and run on the command line. You may use IDEs (e.g., Eclipse) during development, but do not turn in your project without a Makefile. Make sure you code has **no dependencies** on your IDE. We provide starter code in Python; you are welcome to use this, but if you decide use another language, you will need to port the starter code to your language yourself.

Testing Environment

The goal of this project is to get you to develop a TCP-like, reliable transport protocol. As such, we need a way for you to test your code on an unreliable network, i.e., one that is slow, drops packets, reorders and duplicates packets, etc. Unfortunately, real networks are far too reliable for this purpose. As such, we will provide a simulator that is pre-configured to simulate networks with lossy, slow, unreliable network links. This simulator works by interposing itself on your connection, and it will emulate properties of real networks, including delay, router buffers, and various forms of network unreliability.

Starter code

Very basic starter code for the assignment in python is available on the Khoury GitHub server. Provided is a simple implementation of a sender/receiver that implements a simple stop-and-wait protocol. You may use this code as a basis for your project if you wish. To get started, you should create a copy of the starter code on your local machine (or on the Khoury Linux machines if you wish):

```
git clone https://github.khoury.northeastern.edu/cs3700/transport-starter-code.git
```

Simulator

As noted above, rather than testing your router on a real network, we will test it in a simulator. The simulator takes care of reading in the configuration file, launching your sender and receiver, forwarding messages between them, simulating the network, feeding in data to your sender, and verifying that the your receiver prints out the right data. The simulator comes with a suite of configuration files in the directory `configs/` that define situations your code must be able to handle. These are the same configuration files that we will use when evaluating everyone's code (in addition to a few additional configurations we do not reveal). Note that you do not need to parse the configuration files; the simulator is responsible for this functionality.

The simulator is implemented in a script named `run` that is available as part of the starter code.

At no point during this project should you modify `run`. Furthermore, when we grade your code, we will use the original versions of `run` and the configurations in `configs/`, not your (possibly modified) versions.

The simulator accepts the following command line spec:

```
$ ./run <config-file>
```

The simulator will read in the config file, start up your sender and receiver with the appropriate arguments, and then feed your sender data. The simulator will record the output your receiver prints out, and will compare that output to what is expected. The simulator will tell you whether your router passed the test case, printing out a summary of the error if your code did not do what was expected.

Program Specification

```
$ ./4700send <recv_host> <recv_port>
```

- recv_host (Required) The domain name (e.g., "foo.com") or IP address (e.g., "1.2.3.4") of the remote host
- recv_port (Required) The UDP port of the remote host.

The sender must open a UDP socket and send packets 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 program may assume that the `4700recv` program is running on the remote host/port (i.e., you do not need to worry about connection setup).

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.

```
$ ./4700recv
```

On startup, the receiver must bind to a UDP port, and the *first thing* that it prints out to `STDERR` must be the message:

```
Bound to port <port>
```

where `<port>` is the local UDP port number it chose. It will then wait for datagrams from the sender. Similar to 4700send, 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, and *only* the data 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.

NOTE: Your `4700send` program must exit once it has determined the data has been successfully received by the sender. This is the signal that the simulator uses to tell that the test is over. Your `4700recv` program should *not* exit of its own volition, even after it believes it has received all of the data; the simulator will kill it automatically once the `4700send` program exits.

Helper Script

In order to make testing your code easier, we have included a script in the starter code that will launch your receiver, grab its port number, launch your sender, feed the sender input, read the output from the receiver, compare the two, and print out statistics about the transfer. You can run it by executing

```
$ ./run <config-file>
```

The script will print out logs from both your sender and receiver, and at the end, will print out a summary of how your code performed. If you see a message

```
Success!  Data was transmitted correctly.
```

that means your code ran correctly. If you see a message indicating the data did not match, the script will print out what it expected and what it received. Additionally, if your code did not complete in the alloted time, you will see an

error stating as much.
If your correctly transmitted the data, you will see some brief statistics about how well your code performed:

```
Stats: 3.7528 total time, 11736 bytes/16 packets sent (11616/8 sender -> receiver, 120/8 receiver -> ser
```

The above means your code completed the transfer in 3.7528 seconds, and that it sent a total of 11,736 bytes across 16 packets. Obviously, completing tests more quickly and sending fewer bytes/packets is more desirable.

Testing Script

Additionally, we have included a basic test script that runs your code under a variety of configurations and also checks your code's compatibility with the grading script. 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 type

```
$ ./test
```

This test your programs on a number of inputs. If any errors are detected, you can manually run that particular configuration to get more details.

Performance Testing

A portion of your grade on this project will come from performance. This includes the time it takes for sender to fully transmit files to the receiver, and the number of bytes sent during the transmission. The performance tests will be provided in the starter code, and will be labeled in Gradescope. Your project will be graded against a series of benchmarks that we have established, and we will post a leaderboard in Gradescope so you can see how you are doing relative to your peers.

Grading

The grading in this project will consist of

Item	Percentage of Grade
Program correctness	70%
Performance	15%
Style and documentation	15%

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 experience delays, duplicated packets, and so forth. You should always assume that everyone is trying to break your program. To paraphrase John F. Woods, "Always code as if the [the remote machine you're communicating with] will be a violent psychopath who knows where you live."

Implementation strategy

To successfully complete this project, your code will need to handle a variety of different errors (lost packets, duplicate packets, corrupted packets, delayed packets, etc) and will also have to be adaptable to different network environments (high and low bandwidth, high and low latency, high and low jitter). Thus, it is imperative that you break down the project into manageable chunks and slowly work towards completing the project.

You'll notice that this implementation does not pass all of these tests, as it sends too slowly. You'll need to modify the code to have a larger window (more than one packet in the network at once). Having a window of two packets should be sufficient to pass the level 1 tests (note that if you have a window bigger that two packets you might start experiencing packet drops and not pass the test).

2. In the level 2 tests, the network will occasionally duplicate packets and acks, making it hard to tell if a packet is new or not. You will need to implement sequence numbers in order to detect that packets are duplicates of previous ones, and then avoid printing them out. You should then pass the level 2 tests (as above, if you have a window bigger that two packets you might start experiencing packet drops and not pass the test).

3. In the level 3 tests, the network's delay is no longer constant, and the receiver may receive packets in a different order than you sent them from the sender. As a result, your need to implement a "window" at the receiver, as it may receive a packet but not be able to print it out. You will also need to support a sender-side window of at least four packets. You should then pass the level 3 tests.

4. In the level 4 tests, the network becomes even more unreliable, and will start dropping packets. You will need to implement packet loss detection at the sender, as well as the ability to retransmit packets that you detect have been lost. For this, you can use a rule of thumb that the round-trip-time (RTT) of the network is 1 second. You should then pass the level 4 tests.

5. In the level 5 tests, the network becomes yet more unreliable, and will start randomly corrupting your packets. You wil need to implement a way to detecting these corruptions, and dropping any packets that arrive that are not correct (i.e., treat them as if they were lost). You will need to do this at both the sender and receiver, as both data and ack packets can be corrupted. You should then pass the level 5 tests.

6. In the level 6 tests, we test your code in networks with a variety of different latencies. You will need to implement the ability to estimate the round-trip-time (RTT) at the sender and adjust your packet timeouts appropriately (i.e., to twice the RTT). You should then pass the level 6 tests.

7. In the level 7 tests, we test your code in networks with a variety of different available bandwidths. You will need to implement a mechanism to adjust your window size at your sender based on how you observe packets being delivered (both growing and shrinking the window as needed). You should then pass the level 7 tests.

8. Finally, the level 8 tests are stress tests that create networks that have a number of different faults to test whether all the mechanisms of your code work together. If you've implemented everything above correctly, you should pass these tests automatically.

Submission

To submit your work, you should submit your (thoroughly documented) code along with a plain-text (no Word or PDF) `README.md` file. In this file, you should describe your high-level approach, the challenges you faced, a list of properties/features of your design that you think is good, and an overview of how you tested your code.

You should submit your milestone on Gradescope to the Project 4 project. Be sure to indicate who your teammate is, otherwise, they will not get any credit!

← Project 3: BGP Router

Project 5: Web Crawler →

GENERAL

Syllabus

Schedule

PROJECTS

Project 1: Socket Basics

Project 3: BGP Router

Project 4: Reliable Transport Protocol

Project 5: Web Crawler

Project 6: Content Delivery Network