CS 3700 - Networks and Distributed Systems

Project 3: Simple Transport Protocol

This project is due at 11:59pm on Friday, March 13, 2020.

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 scripts to set up a vagrant image to 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 do **NOT** have to implement connection open/close etc. You may assume that the receiver is run first and will wait indefinitely, and the sender can just send the data to the receiver.

Requirements

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

- Your sending program must be named `3700send` and your receiving program must be named `3700recv`
- The `3700send` accepts data from STDIN, sending data until EOF is reached
- The `3700recv` must print out the received data to STDOUT in order and without errors
- Your sender and receiver must gracefully exit
- The sender and receiver must work together to transmit the data reliably
- The sender and receiver must print out specified debugging messages to STDERR
- 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
- 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)

You may implement any reliability algorithm(s) you choose. However, your implementation must meet certain minimums of performance. Desirable properties in any reliability algorithm 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, acknowledgments, etc.

Being said, correctness matters most: your receiver must output exactly the input to your sender. 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 reordering packets, delaying packets, and dropping packets. You should handle these errors gracefully, recover, and not crash.

Your Programs

For this project, you will submit two programs: a sending program `3700send` that accepts data on STDIN and sends it across the network, and a receiving program `3700recv` that receives data and prints it to STDOUT in-order. You must use UDP. You may not use any transport protocol libraries in your project (TCP, QUIC, etc).

Language

You can write your code in whatever language you choose, as long as your code compiles and runs on **unmodified** Khoury College Linux machines **on the command line**. Do not use libraries that are not installed by default on the Khoury College Linux machines. You may use IDEs (e.g. Eclipse) during development, but do not turn in your IDE project without a Makefile. Make sure you code has **no dependencies** on your IDE.

Starter Code

Very basic starter code in C and Python for the assignment is available at tcp-starter-c and tcp-starter-py. 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. It will only 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, you should copy down this directory into your own local directory. You can compile the code by running *make*. You can also delete any compiled code and object files by running *make clean*.

Program Specification

The command line syntax for your sending is given below. The syntax for launching your sending program must be:

```
./3700send <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 syntax for launching your receiving program must be:

```
./3700recv
```

To aid in grading and debugging, your program must print messages to STDERR.
Your sending program must print at least the following:

- "`<timestamp> [send data] start (length)`"
  When the sender sends a packet (including retransmission). Here, *timestamp* is a timestamp (down to the microsecond), *start* is the beginning offset of the data sent in the packet, and *length* is the amount of the data sent in that packet.
- "`<timestamp> [recv ack] end`"
  When `3700send` receives an acknowledgment. Here, *end* is the last offset that was acknowledged.
- "`<timestamp> [completed]`"
  Upon completion of file transfer, just before exiting.

Your receiving program must print at least the following:

- "`<timestamp> [bound] port`"
  When the receiving program has started up and bound to a port. Here, *port* is the bound port held by the receiver.
- "`<timestamp> ([recv data] start (length) status | IGNORED)`"
  Upon receiving a valid data packet. Here, *start* is the beginning offset of the data sent in the packet, *length* is the amount of the data sent in that packet, and *status* is one of "`ACCEPTED (in-order)`" or "`ACCEPTED (out-of-order)`"
- "`<timestamp> [recv corrupt packet]`"
  Upon receiving a corrupt packet.

- `"<timestamp> [completed]"`
  Upon completion of file transfer, just before exiting.

You may also print concise and readable messages of your own to indicate timeouts, etc. In the C starter code, the function *mylog(char *fmt, ...)* is provided for this purpose.

You should develop your client program on the Khoury College Vagrant image. You are welcome to develop your own Linux/OS X/Windows machines, but you are responsible for getting your code working, and your code will be graded on the Vagrant image.

Testing Your Code

In order for you to test your code over an unreliable network, we are providing scripts to setup a Vagrant image that will emulate a network that drops, reorders, duplicates, and delays your packets. These scripts are tested to work on the Vagrant image, and are known **NOT TO WORK** on Khoury machines managed by Systems. You will need to use the loopback interface in order to leverage the emulated network. In other words, you might run something like *./3700recv* in one terminal, record the port it local binds to (say, 3992), and then run *./3700send 127.0.0.1:3992* in another terminal.

The scripts mentioned below are available in the archive tcp-sim. The file should be untar'd in the same directory as your 3700send and 3700recv executables.

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

```
netsim [--bandwidth <bw-in-mbps>]
       [--latency <latency-in-ms>] [--delay <percent>]
       [--drop <percent>]
       [--reorder <percent>] [--duplicate <percent>]
```

- **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.

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
```

the simulator will configure a network with 500 Kb/s bandwidth and a latency of 100 ms, and will randomly delay 20% of your packets and drop 40%. In order to reset it so that none of your packets are disturbed, you can simply call `netsim` with no arguments. **Note that the simulator is stateful**, meaning your settings will persist across multiple sessions.

Helper Script

In order to make testing your code easier, we have also included a perl script that will launch your receiver, read the 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. This script is included in the simulator code tarball, and you can run it by executing

```
./run
```

This script also takes a couple of arguments to determine what it should do:

```
./run [--size (small|medium|large|huge)] [--printlog] [--timeout <seconds>]
```

- **size**: The size of the data to send, including 1 KB (small), 10 KB (medium), 100 KB (large), 1MB (huge). Default is small.
- **printlog**: Instructs the script to print a (sorted) log of the debug output of 3700send and 3700recv. 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 of this script include some statistics about the transfer:

```
bash$ ./run --size large
Time elapsed: 1734.921 ms
Packets sent: 140
Bytes sent: 107000
Effective goodput: 461.116
Kb/s Data match: Yes
```

where "`Data match`" indicates whether the data was transferred correctly.

Testing Script

Additionally, we have included a testing script that runs your code under a variety of network conditions. The tests provided in the testing scripts are materially the same as the ones your code will be graded against. To run the test script, simply type

```
bash$ ./test
```

This will compile your code and then test your programs on a number of inputs. If any errors are detected, the test will print out the expected and actual output.

Note: The testing script has the same restrictions as `netsim` with regards to where it can be run.

Performance Testing

20% of your grade on this project will come from performance. To help you know how you're doing, the testing script will also run a series of performance tests at the end; for each test that you successfully complete, it will report your time elapsed and bytes sent. For example, you might see

```
Performance tests
    Huge 5Mb/s, 10 ms               [DATA OK]
    13.889 sec elapsed, 1.1MB sent
```

This indicates that your code sent the correct data in 13.889 seconds, using a total of 1.1MB of data including retransmissions, overhead, etc. Using the following equations, we can calculate the minimum number of packets and according minimal time it will take a simplified model[1] of transmission to take.

```
P_th = ceil(total_data/packet_size)

P_min = 2 * P_th * (1 + drop_rate)
T_min = P_th * (1 + drop_rate) * (T_latency + packet_size/line_rate)
```

Note: delays, reorders, and duplicates do not impact this model. We will add a "fudge-factor" of 5% to drop_rate on tests that involve any of these.

We will use the following breakdown for evaluating performance when grading code. Variables can be calculates as described above, and time and packet count will be weighted equally.

Max Transfer Time	Max Packets (with non-zero errors)	Value
.5T_min	1.3P_min	141.7%
.55T_min	1.4P_min	100%
.7T_min	1.5P_min	80%
.85T_min	1.65P_min	60%
T_min	1.8P_min	40%
2T_min	2P_min	20%
∞	∞	0%

Submitting Your Project

If you have not done so already, register yourself for our grading system using the following command:

```
$ /course/cs3700sp20/bin/register-student [NUID]
```

NUID is your Northeastern ID number, including any leading zeroes.

Before turning in your project, you and your partner(s) must register your group. To register yourself in a group, execute the following script:

```
$ /course/cs3700sp20/bin/register project3 [team name]
```

This will either report back success or will give you an error message. If you have trouble registering, please contact the course staff. **You and your partner(s) must all run this script with the same [team name]**. This is how we know you are part of the same group.

To turn-in your project, you should submit your (thoroughly documented) code along with two other files:

- A Makefile that compiles your code. Your Makefile may be blank, but it must exist.
- A plain-text (no Word or PDF) README.md 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.

Your README.md, Makefile, source code, etc. should all be placed in a directory. You submit your project by running the turn-in script as follows:

```
$ /course/cs3700sp20/bin/turnin project3 [project directory]
```

"`[project directory]`" is the name of the directory with your submission. The script will print out every file that you are submitting, so make sure that it prints out all of the files you wish to submit! The turn-in script will not accept submissions that are missing a README.md or a Makefile. **Only one group member needs to submit your project.** Your group may submit as many times as you wish; only the last submission will be graded, and the time of the last submission will determine whether your assignment is late.

Double Checking Your Submission

To try and make sure that your submission is (1) complete and (2) will work with our grading scripts, we provide a simple script that checks the formatting of your submission. This script is available on the Khoury College Linux machines and can be executed using the following command:

```
/course/cs3700sp20/code/project3/project3_format_check.py [path to your project directory]
```

This script will attempt to make sure that the correct files (e.g. README.md and Makefile) are available in the given directory, that your Makefile will run without errors (or is empty), and that after running the Makefile two

programs named 3700send and 3700recv exist in the directory. The script will also try to determine if your files use Windows-style line endings (\r\n) as opposed to Unix-style line endings (\n). *If your files are Windows-encoded, you must convert them to Unix-encoding using the dos2unix utility before turning in.*

Grading

This project is worth 12% of your final grade. The grading in this project will consist of:

- 80% Program correctness: that the output from `3700recv` byte-wise matches the input to `3700send`
- 20% Performance: that you pass the performance tests (described above)

Points will be taken off for submissions that send datagrams with greater than 1500 bytes of data.

By definition, you are going to be graded on how gracefully you handle errors. Particularly, your code must never print out incorrect data. Your code will definitely see corrupted packets, 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."

You can see your grades for this course at any time by using the gradesheet program that is available on the Khoury College machines.

```
$ /course/cs3700sp20/bin/gradesheet
```

Asymptotic Assumptions for Mathematical Model

- Transmission of each packet is blocking until received
- ACK/NACKs are infinitesimal and instantaneous