CMSC 417 Computer Networks **Spring 2016**

Programming Assignment 4

Assigned: April 8 *Due: April 22, 11:59:59 PM.*

1 Introduction

For this assignment, you will design and implement a reliable file transfer protocol atop UDP, similar to the reliability provided by TCP. To do so, implement a file sender that sends a file split across many UDP packets and a file receiver that receives the packets and reconstructs the file. Much as in Project 0, the connection will be flaky, and may occasionally drop, reorder, or even corrupt packets. Despite this, your implementation must successfully reconstruct the file.

2 Protocol

We have very few requirements for your protocol, because we want you to design your own based on the networking concepts presented in class and online. The only requirements are:

- Your protocol must transmit the file over UDP.
- All transmissions between the sender and receiver must occur on the ports specified. No sending over secondary ports to avoid the gremlins (see Section 4).
- Each packet must be no more than 10,000 bytes in size.
- You may not send data unnecessarily. Occasionally is fine, and we don't intend to be super harsh on this, but sending each packet several times to preemptively handle loss is not sufficient to pass this assignment.
- Your sender and receiver must somehow communicate that file transmission is complete and it is okay to terminate. This may be harder than it sounds.[1]

Beyond this, you are left to your own creativity.

3 Deliverable

You will deliver a receiver program that is executed as:

```
bash runReceiver.sh <port> <destination>
```

Where `<port>` is the port that the receiver will listen on and `<destination>` is the location to which the receiver should write the received file. If destination already exists, overwrite it.

You will also deliver a sender program that is executed as:

```
bash runSender.sh <port> <file>
```

[1] `https://en.wikipedia.org/wiki/Two_Generals%27_Problem`

Where `<port>` is the port on `localhost` that the sender should transmit to and `<file>` is the path to the file that the sender should transmit. You may assume that the file exists, is readable, and is under 500MB in size. We will also always start the receiver before executing the sender.

We will not grade the output of either program. However, writing too much to `stdout` will cause the scripts that we provide you to hang (when the buffer is full). This won't affect our tests, but will likely bite you if you're not careful. Both the sender and receiver must terminate upon successful file transfer.

As in Project 3, you are allowed to write your program in any language that you'd like. Again, we will be installing support for python, ruby, java, scala, C and C++ on the VM. If you feel that you need additional language support and the VM doesn't support it, please let us know and we may (but also may not) accommodate your request. Make sure to test that your code compiles and runs in the VM using `make` and `bash runSender.sh` and `bash runReceiver.sh` so that our tests will also be able to compile and run it.

4 Gremlins

To simulate adverse network conditions, we will place gremlins between your sender and receiver. The gremlins are known to occasionally drop packets, reorder packets, or corrupt packets. Your submission must be able to reliably transmit the file despite this in order to receive full points on the project, however, partial credit will be awarded if you only handle some of these conditions.

We provide an example gremlins script to you, but we do not guarantee that this is the script we will use in testing.

In order to insert gremlins between your sender and receiver, we will execute:

```
ruby bin/gremlins.rb <port A> <port B>
bash runReceiver.sh <port B> <destination>
bash runSender.sh <port A> <file>
```

The gremlins script will read messages sent to either `<port A>` or `<port B>`, alter or drop them as necessary, and then echo them to the other port. For convenience, we have provided you with the `runWithGremlins.rb` ruby script that automatically executes the gremlins, receiver, and sender.

5 Additional Requirements

- Please clone your starter repository from: `http://ter.ps/417p4`
- It may be useful to randomly generate large files to test your implementation on. GitHub has strict repository size limits, so be careful not to commit these. We have provided the `testFiles` directory that you can place these in to have git ignore them.
- The sender and receiver must each use less than 100MB of RAM at any given time, for a total of 200MB between the two. It should be possible to stay well under this limit.
- The receiver must not read the file from disk, and must only receive the file's contents from the sender.
- You're not required to implement any algorithms related to congestion control.