National University of Singapore
School of Computing
CS2105: Introduction to Computer Networks
Semester 1, 2015/2016

Assignment 2
Reliable File Transfer Protocol

Release date: 21 September 2015
Due: 16 October 2015, 23:59

Overview

In this assignment, you will be writing the sending and receiving transport-level code for implementing a reliable file transfer protocol over UDP. The underlying channel in unreliable and may corrupt, drop or even re-order the packets are random. This assignment should be fun since your implementation will differ very little from what would be required in a real-world situation.

A Word of Advice

This assignment is potentially complex and time consuming. We advise you write your program incrementally and modularly. For example, start by dealing with packet corruption, then packet loss and so one. You should test your program after every major change and it will be useful to use a version control system like Git or Mercurial.

Plagiarism Warning

You are free to discuss this assignment with your friends.But, ultimately, you should write your own code. We employ a zero-tolerance policy against plagiarism. If a suspicious case is found, student would be asked to explain his/her code to the evaluator in person. Confirmed breach may result in zero mark for this assignment and further disciplinary action from the school.

Architecture

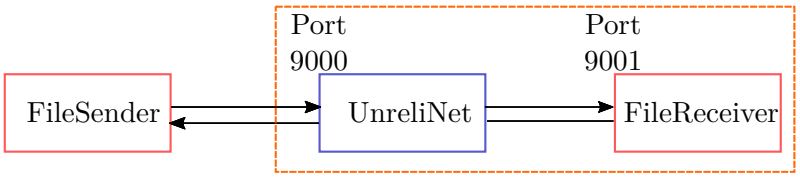

Figure 1: UnreliNET simulates an unreliable network.

There are three programs in this assignment: `FileSender`, `UnreliNET` and `FileReceiver` The interactions of the programs is illustrated in Figure 1. The `FileSender` and `FileReceiver` programs implement a reliable file transfer protocol over UDP. The `UnreliNET` program acts as a

proxy between `FileSender` and `FileReceiver`. Instead of the `FileSender` sending the packets directly to the `FileReceiver`, it sends all packets to `UnreliNET`. `UnreliNET` simulates an unreliable channel by randomly discarding packets, corrupting packets and even reordering the packets before forwarding them to the `FileReceiver`. Similarly, packets from the `FileReceiver` are also subject to the same unreliable conditions when forwarded to the `FileSender`.

The `UnreliNET` program is complete and given to you. Your task is to write the `FileSender` and `FileReceiver` programs so that a file can be successfully transferred in over an unreliable channel. The received file should be exactly identical to the one being sent. You will have to employ techniques taught in the lecture or from the textbook to ensure correct and reliable delivery.

Note: Protocols to handle out-of-order packets is not covered in the syllabus. Bonus marks will be given if your protocol is able to cater for this.

The `FileSender`

The `FileSender` program sends a given file to `FileReceiver` via `UnreliNET`. `UnreliNET` may then lose, corrupt or reorder the packets with a given probability before relaying them to the `FileReceiver` program.

The `FileSender` program will take in 4 input arguments: the host name, port number, source file, and the destination file name. For example:

```
$ java FileSender localhost 9000 ../large.mp4 big.mp4
```

sends the file `../large.mp4` to `UnreliNET` running on localhost on port 9000. `UnreliNET` will then relay the file to `FileReceiver` to be stored as `big.mp4`. The `FileSender` program will quit once the file has been successfully transfered.

You may assume that during testing, `FileSender` will be supplied with a valid path and filename, and that the pathname will not exceed 255 characters. No input validation is needed. You will also need to design a way to send the filename over to the `FileReceiver` for storing.

Note: Windows OS uses a different file separator '`\`', e.g., `..\large.mp4`

`UnreliNET.class`

We simulate an unreliable channel by relaying packets through the `UnreliNET` program. `UnreliNET` will only relay packets that contains at most 1,000 bytes of application data.

The compiled classfile of `UnreliNET` is provided. To run `UnreliNET`, the command is:

```
java UnreliNET <loss> <corrupt> <reorder> <unrelinet port> <recv port>
```

where

- `loss` is the probability of losing a packet.
- `corrupt` is the probability of a packet being corrupted.
- `reorder` is the probability that a packet will be delayed and re-ordered.
- `unrelinet port` is the port that `UnreliNET` is to listen on.
- `recv port` is the port on localhost that the `FileReceiver` is listening on.

For example, the `UnreliNET` command for the setup shown in Figure 1 could be:

```
$ java UnreliNET 0.1 0.2 0.3 9000 9001
```

where on average 10% of the packets will be lost, 20% of the packets will be corrupted, and 30% of the packets will be delayed and potentially delivered out-of-order.

We recommending setting these values between the range [0.0, 3.0] for testing. Setting too large a value may result in very slow file transmission. If you have trouble getting your code to work, you might want to set them to 0 first for debugging.

You may also store the output of `UnreliNET` to a file for analysis as so:

```
$ java UnreliNET 0.1 0.2 0.3 9000 9001 > output.txt
```

FileReceiver.class

The `FileReceiver` program receives a file from `FileSender` (via `UnreliNET`) and saves it in the same directory it is running in, with the filename specified by the `FileSender` program. We will terminate `FileReceiver` with `<Ctrl>+C` once `FileSender` terminates, so you should ensure that the file is correctly written to disk at this point in time.

The `FileReceiver` takes in one argument, the port number of which to listen on, e.g.,

```
$ java FileReceiver 9001
```

will being listening on port 9001.

Additional Pointers

Testing

Although `UnreliNET` is designed to accept traffic over a network, you should run all three programs on localhost to avoid interference from network traffic.

You should first run `FileReceiver` and `UnreliNET` before running `FileSender`. When testing on `sunfire`, you can open three SSH windows, one for each program. `UnreliNET` runs indefinitely and so you have to press `<Ctrl>+C` to terminate it.

You may use the included `SimpleUDPSender` and `SimpleUDPReceiver` to test your set up. The `SimpleUDPSender` simply sends a given number of packets which the `SimpleUDPReceiver` displays the packet number it receives. You can use the code as a reference for your own programs.

To check if both files are identical, you can use `md5sum` to compute the digest of both files, e.g., `$ md5sum doge.jpg`. The digest for both files will be identical if their contents are identical.

Timer and Timeout Values

We do not recommend that you use a timeout value larger than 200 ms. Using a larger value might cause your program to be slow in sending data and exceed the allowable time limit. For simple alternating-bit protocols, you could potentially use a 1 ms timeout when testing on localhost as the RTT is below 1 ms.

Computing Checksum

`UnreliNET` will corrupt the payload of the UDP packet. Thus, you should use a checksum to verify the integrity of the received packets. You can use the same method as in Assignment 0 for computing a checksum.

Self-defined Header/Trailer Fields

As part of your protocol, you will have to implement certain header/trailer fields into the packets, such as sequence number and checksum. The number of fields and the structure in a packet is agreed upon between the sender and the receiver. That is an application layer protocol that you design.

The `ByteBuffer` class from the `java.nio` package makes it easy to insert and retrieve primitive data into a byte array. You can refer to `SimpleUDPSender` and `SimpleUDPReceiver` on how a `ByteBuffer` can be used. This is just a suggestion and you do not to use this class if you have an alternative solution.

Submission

You will submit your program on IVLE. Zip your files into a single zip file and name it "`a2-<student number>.zip`" and upload to the Assignment 2 folder. For example, if your student number is A0123456X, then you should name your file "`a2-a0123456x.zip`".

It is your responsibility to ensure that your file is successfully uploaded into IVLE before the dateline. You should refresh the IVLE workbin and confirm that your file there.

The dateline for submission is at 16 October 2015, 23:59 hrs. Late submissions will have an additional penalty of 25% per day.

Submitting in Java

Your zip file should include two java files named `FileSender.java` and `FileReceiver.java`, which contains the classes `FileSender` and `FileReceiver` respectively. You may include other necessary java files in the zip file.

We will unzip the contents of your zip file and compile your submission using `javac *.java` and run each program as described above.

Submitting in Python

You may write your program in Python. Your zip file should contain the files `FileSender.py` and `FileReceiver.py`. We will run each program as described above.

Submitting in C++

You may write your program in C++. Your zip file should contain the files `FileSender.cpp` and `FileReceiver.cpp`. We will compile all .cpp files with `for i in *.cpp; do g++ $i -o ${i/.cpp}; done`. We will run each program as described above.

Details and Grading

The score for this assignment is 13 marks. If you obtain above 13 marks, half of the extra marks can be passed-on to other assignments.

There will be a **time limit of 60 seconds** for every 1 MB transferred.

Basic criteria:

- Your program can be compiled on `sunfire`. **(1 mark)**
- Your programs can successfully transfer a file from sender to receiver over a perfectly reliable channel, i.e., no error is introduced. **(2 marks)**
- Your program can successfully transfer a file from sender to receiver in the presence of packet corruption. **(2 marks)**
- Your program can successfully transfer a file from sender to receiver in the presence of packet loss. **(2 marks)** Your program can successfully transfer a file from sender to receiver in the presence of both packet corruption and packet loss. **(2 marks)**

Advanced criteria:

- Your program can successfully transfer a file from sender to receiver in the presence of packets being reordered. This is not covered in the syllabus so you will have to do your own research. **(2 marks)**
- Your program can successfully transfer a file from sender to receiver in the presence of packet corruption, packet loss and reordering. **(2 marks)**
- **Speed Test.** It is not difficult to ensure reliable delivery using the simple alternate bit protocols we discussed in lecture. However, they have very poor utilization. Thus, for this test, you will need to implement pipelining to increase the speed of the transfer.

  We will rank you according to the time your program takes to transfer a large file (more than 50 MB) in the presence of 2% packet corruption and 2% packet loss.

  - Rank 1–30 students. **(4 marks)**
  - Rank 31–60 students. **(3 marks)**
  - Rank 61-90 students. **(2 marks)**
  - Rank 91 and beyond, whose timing is under 60 seconds. **(1 mark)**

  Note: Printing output to the terminal can cause additional delay. You may disable the output of `UnreliNET` by redirecting it to `/dev/null`, e.g., "`java UnreliNET 0.02 0.02 0 8000 8001 > /dev/null`".