

Submission Deadline

19 March 2016 (Saturday), 6pm sharp. 3 points penalty will be imposed on late submission (Late submission refers to submission or re-submission after the deadline for whatever reason).

Introduction

In this assignment, you will transfer a file over UDP protocol on top of an unreliable channel that may either corrupt or drop packets randomly (but always deliver packets in order).

This programming assignment is worth 10 marks and shall be completed **individually**.

Writing Your Programs

You are free to write your programs on any platform/IDE that you are familiar with.

However, you are responsible to ensure that your programs run properly on **sunfire server** because **we will test and grade your programs on sunfire**.

Program Submission

Please submit your programs to **CodeCrunch**: <https://codecrunch.comp.nus.edu.sg>.

You may submit both the sender and receiver programs to **CodeCrunch** simultaneously by pressing the <Ctrl> key when choosing programs to upload. Note that **CodeCrunch** is just used for program submission and no test case has been mounted on it. Hence you may ignore the feedback from CodeCrunch regarding the quality of your programs (e.g., can improve).

Grading

Your programs will be graded according to their correctness using a grading script:

- **[2 points]** Programs are compilable on **sunfire**, program execution follows specified **Java** commands (see sections below).
- **[1 point]** Programs can successfully transfer a file from sender to receiver when channel is perfectly reliable (i.e. no error at all).

- **[1 point]** Programs can successfully transfer a file from sender to receiver in the presence of data packet corruption.
- **[1 point]** Programs can successfully transfer a file from sender to receiver in the presence of ACK/NAK packet corruption.
- **[1 point]** Programs can successfully transfer a file from sender to receiver in the presence of data packet loss.
- **[1 point]** Programs can successfully transfer a file from sender to receiver in the presence of ACK/NAK packet loss.
- **[1 point]** Programs can successfully transfer a file from sender to receiver in the presence of both packet corruption and packet loss.

To conclude a successful file transfer, received file must have identical content as the sent one (use command `cmp` to check it on `sunfire`). Your program should work for both text and binary files, and for both small files and large files (a few MBs).

Grading script doesn't care what messages your programs print on the screen. It just checks if the received file is exactly the same as the sent one in respective test cases.

- **[2 points]** (Who runs faster?) Time taken for your programs to transfer an enormous file (more than 10MB) in the presence of both packet corruption and packet loss will be noted down. **The fastest batch of student programs will receive 2 marks and slower programs that are still faster than average will receive 1 mark.** 😊

This competition aims to stretch the potential of those who pursue 'A's in CS2105. You may apply the knowledge gained from this or other courses as appropriate. However, since the objective of this assignment is to design a fast network transmission protocol, you are NOT allowed to play tricks (e.g., data compression) to gain unfair advantage over your peers. Please consult the teaching team if you are unsure if certain techniques can be used or not.

A Word of Advice

This assignment is complex and time-consuming. You are suggested to write programs incrementally and modularly. For example, deal with data packet corruption first, then ACK packet corruption, then data packet loss, etc. Test your programs after every single major change. Take note that partial credit will be awarded even if your programs don't meet all listed requirements.

Plagiarism Warning

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

Overall Architecture

There are three programs in this assignment, **FileSender**, **UnreliNET** and **FileReceiver**. Their relationship is illustrated in Figure 1 below. The **FileSender** and **FileReceiver** programs implement a file transfer application over UDP protocol. The **UnreliNET** program simulates the transmission channel that transmits packets unreliably and randomly corrupts or loses packets. However, for simplicity, you can assume that this channel always delivers packets in order.



Figure 1: UnreliNet Simulates Unreliable Network

The **UnreliNET** program acts as a proxy between **FileSender** and **FileReceiver**. Instead of sending packets directly to **FileReceiver**, **FileSender** sends all packets to **UnreliNET**. **UnreliNET** may introduce bit errors to packets or lose packets randomly. It then forwards packets (if not lost) to **FileReceiver**. When receiving feedback packets from **FileReceiver**, **UnreliNET** may also corrupt them or lose them with certain probability before relaying them to **FileSender**.

The **UnreliNET** program is complete and given. Your task in this assignment is to develop the **FileSender** and **FileReceiver** programs so that a file can be successfully transferred from sender to receiver in the presence of possible packet corruption and packet loss. The received file should be identical to the file sent. You may need to implement techniques learnt in lecture, including sequence number, acknowledgement, timeout and retransmission, to ensure that packets are correctly delivered.

FileSender Class

The **FileSender** program is basically a file uploader that opens a given file and sends its content as a sequence of packets to **UnreliNet**. **UnreliNet** would then corrupt/lose packets with certain probability before relaying them to the **FileReceiver** program.

To run **FileSender** on **sunfire**, type command:

```
java      FileSender      <path/filename>      <unreliNetIP>
<unreliNetPort> <rcvFileName>
```

For example:

```
java FileSender ../test/cny.mp3 localhost 9000 gxfc.mp3
```

sends the file **cny.mp3** from directory (relative path) **../test** to **UnreliNet** running in the same host at port 9000. **UnreliNet** will then pass the file to your **FileReceiver** program to be stored as **gxfc.mp3**.

(Note: Windows system uses a different file separator '\', e.g., ..\test\cny.mp3)

You may assume that during testing, your sender program will be supplied with the correct path and filename (no more than 100 bytes long). No input validation is needed.

UnreliNET Class

The **UnreliNET** program simulates an unreliable channel that may corrupt or lose packets with a certain probability. This program is given and shouldn't be changed.

To run **UnreliNET** on **sunfire**, type command:

```
java      UnreliNET      <P_DATA_CORRUPT>      <P_ACK_CORRUPT>
<P_DATA_LOSS>  <P_ACK_LOSS>  <unreliNetPort>  <rcvHostName>
<rcvPort>
```

For example:

```
java UnreliNET 0.3 0.2 0.1 0.05 9000 localhost 9001
```

listens on port 9000 and forwards all received data packets to **FileReceiver** running on the same host at port 9001, with 30% chance of packet corruption and 10% chance of packet loss. The **UnreliNET** program also forwards ACK/NAK packets to **FileSender**, with 20% packet corruption rate and 5% packet loss rate.

Packet Corruption Probability

The **UnreliNET** program randomly corrupts or loses data packets and ACK/NAK packets according to the specified parameters P_DATA_CORRUPT, P_ACK_CORRUPT, P_ACK_LOSS and P_DATA_LOSS. You can set these values to anything in the range [0, 0.3] during testing (setting a too large corruption/loss rate may result in a very slow file transmission).

If you have trouble getting your code to work, it might be advisable to set them to 0 first for debugging purposes.

FileReceiver Class

The **FileReceiver** program receives a file from **FileSender** (through **UnreliNET**) and saves it in the same directory as the **FileReceiver** program, with a filename specified by **FileSender**.

To run **FileReceiver** on **sunfire**, type command:

```
java FileReceiver <rcvPort>
```

For example:

```
java FileReceiver 9001
```

listens on port 9001 and dumps the bytes received into a file whose name is given by sender.

Running All Three Programs

You should first launch **FileReceiver**, followed by **UnreliNET** in the second window. Finally, launch **FileSender** in a third window to start data transmission. All programs should receive user input from command-line argument only. Please always test your programs in **localhost** to avoid the interference of network traffic on your programs.

The **UnreliNET** program simulates unreliable communication network and runs infinitely. Once launched, you may reuse it in consecutive tests. To manually terminate it, press <Ctrl> + c. The **UnreliNET** program prints out information that is helpful in debugging your programs. However, we will disable such outputs in speed test.

The sender and receiver programs shouldn't communicate with each other directly – all traffic should be forwarded by the **UnreliNET** program. The sender program should terminate properly (e.g., no exception, no infinite loop) once transmission finishes. However, you may leave the receiver program running infinitely (i.e., no need for receiver to detect end of transmission and terminate, but DO remember to close the file output stream at the end).

Self-defined Header/Trailer Fields at Application Layer

UDP transmission is unreliable. To detect packet corruption or packet loss, you may need to implement reliability checking and recovery mechanisms at application layer. The following header/trailer fields are suggested though you may have your own design:

- Sequence number
- Checksum

Note that each packet **FileSender** sends should contain at most 1000 bytes of application data (inclusive of self-defined header/trailer fields), or **UnreliNET** program will reject it.

Computing Checksum

To detect bit errors, **FileSender** should compute checksum for every outgoing packet and embed it in the packet. **FileReceiver** needs to re-compute checksum to verify the integrity of a received packet.

Please refer to Assignment 0 Exercise 3 on how to compute checksum using Java **CRC32** class.

Timer and Timeout Value

Sender may have to maintain a timer for unacknowledged packet. You are suggested to use the **setSoTimeout()** method of Java **Socket** class.

You shouldn't set a timeout value that is larger than 200ms, or your program might be too slow in transmitting data and thus get killed by the grading script.

Reading/Writing Values to Header/Trailer Fields

The number of header/trailer fields and the sequence of their appearance in a packet is the agreement between sender and receiver (i.e. an application layer protocol designed by you).

As discussed in tutorial 3, to give value to a header field of a 4-byte integer, you may consider **ByteBuffer** class from **java.nio** package. An example is shown below.

At sender side:

```
int length = 1000;
// allocate a 4-byte array to store converted integer
byte[] pktLen = ByteBuffer.allocate(4).putInt(length).array();
// copy content of pktLen to the beginning of output buffer of pkt
System.arraycopy(pktLen, 0, buffer, 0, pktLen.length);
```

At receiver side:

```
// extract the first 4 bytes of a packet as the integer 'length'
ByteBuffer wrapper = ByteBuffer.wrap(pkt.getData(), 0, 4);
int length = wrapper.getInt();
```

Question & Answer

If you have any doubts on this assignment, please post your questions on IVLE forum or consult the teaching team. We are not supposed to debug programs for you. However, we may help to clarify misconceptions or give you necessary directions.