National University of Singapore
School of Computing

CS2105 **Assignment 2** Semester 2 AY17/18

Submission Deadline

26 March 2018 (Monday), 1pm sharp. **2 marks penalty** will be imposed on late submission (late submission refers to submission or re-submission after the deadline).

This assignment is complex. Please start early and submit early. You can submit 99 times and only the last submission will be graded. We strongly advice against submitting only a few minutes before the deadline.

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.**

Group Work

This assignment should be solved individually. However, if you struggle, you are free to form a team with another student (i.e. maximum team size is 2) and solve the assignment together. Group submission is subject to **2 marks penalty** for each member student.

Under no circumstances should you solve it in a group and then submit it as an individual solution. This is considered plagiarism. Please also refer to the detailed explanations below and the skeleton code for more information about plagiarism. Group work need special care during submission (see below).

Writing Your Programs

You are free to write your programs on any platform/IDE.

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

Program Submission

Please submit all files to **CodeCrunch**: https://codecrunch.comp.nus.edu.sg.

You can submit multiple Java files to **CodeCrunch** simultaneously by pressing the <Ctrl> key when choosing the programs to upload. **Please do not submit folders, archives, rar, or zip files**. Just submit your java programs. There is no need to submit UnreliNET.java.

You can create additional Java files for helper classes but make sure to submit them all. However, if possible try to use inner classes to keep the number of submitted files small.

Note that we use **CodeCrunch** only for program submission. No test cases will run on **CodeCrunch**. Hence, you may ignore the feedback from **CodeCrunch** regarding the quality of your programs.

Grading

We will grade your programs according to their correctness using a grading script. There is no manual grading and no partial mark for each of the following grading rubrics.

- [2 points] Programs compile on **sunfire** without error; program execution follows specified Java commands exactly (see sections below). In addition, you submitted Java programs only. The files **are not zipped, tarred, or hidden** somewhere in a folder and have the correct (file and class) names.
- [1 point] Programs can successfully send a small file (a few KB) from Alice to Bob in a perfectly reliable channel (i.e. no error at all).
- [1 point] Programs can successfully send a large file (< 4GB) from Alice to Bob in a perfectly reliable channel (i.e. no error at all).
- [2 points] Programs can successfully send a (small or large) file from Alice to Bob in the presence of both data packet corruption and ACK/NAK packet corruption.
- [2 points] Programs can successfully send a (small or large) file from Alice to Bob in the presence of data packet lost and ACK/NAK packet loss.
- [1 point] Programs can successfully send a (small or large) file from Alice to Bob in the presence of both packet corruption and packet loss.
- [1 point] Programs pass time test (see Section UnreliNET Class).

The 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. As in assignment 0 exercise 3, please use the **cmp** command to check binary equivalence.

A Word of Advice

This assignment is complex and time-consuming. We suggest you start writing programs early, incrementally and modularly. For example, deal with error-free transmission first, then data packet corruption, ACK packet corruption, etc. Test your programs frequently and make backup copies before every major change (however, do not post your code to the public Internet, e.g. public repositories). Frequent backups will allow you to submit at least a partially correct solution that passes some test cases, if not all.

Question & Answer

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

Plagiarism Warning

You are free to discuss this assignment with your friends. However, 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.

Do not post your solution in any public domain on the Internet or share it with friends even after this semester.

Overall Architecture

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

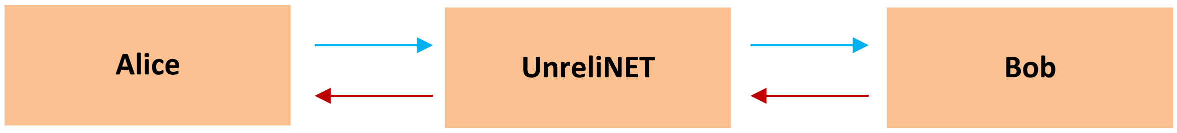

Figure 1: UnreliNet Simulates Unreliable Network

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

The **`UnreliNET`** program is complete and given. Your task in this assignment is to develop the **`Alice`** and **`Bob`** programs so that Bob will receive files successfully in the presence of packet corruption and packet loss.

`UnreliNET` Class

The **`UnreliNET`** program simulates an unreliable channel that may corrupt or lose packets with a certain probability, as well as adding a random amount of delay to each packet to simulate network delay. <u>This program is given and should not be changed.</u>

To run **`UnreliNET`** on **`sunfire`**, type then following command:

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

For example:

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

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

To pass the last grading rubric (time test), your program should finish transmitting the provided test input file (around 1 MB) within **180 seconds**, with all 4 packet lost/corruption parameters set to 0.1.

Packet Error Rate

The **UnreliNET** program randomly corrupts or loses data packets and ACK/NAK packets according to the specified parameters P_DATA_CORRUPT, P_DATA_LOSS, P_ACK_CORRUPT, and P_ACK_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 transmission).

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

Alice Class

The **Alice** program is a very simple one-way file transfer program. It opens a given file and sends its content as a sequence of packets to **UnreliNet**, which will then forward the packets to **Bob**.

To run **Alice** on **sunfire**, type the following command:

```
java Alice <path/filename> <unreliNetPort> <rcvFileName>
```

For example,

```
java Alice ../test/cs2105.zip 9000 notes.zip
```

sends the file **cs2105.zip** from directory (relative path) **../test** to **UnreliNet** running in the same host at port 9000. **UnreliNet** will then forward the file to **Bob** to be stored as **notes.zip**.

Notes:

1. You should always run **UnreliNET**, **Alice** and **Bob** programs on the same host.
2. **Alice** should terminate after (1) reading all data from the file and (2) forwarding it successfully to **Bob**.
3. Use a **Maximum Segment Size of 1024 bytes** or your packets may be rejected by the **UnreliNET** program.
4. **<rcvFileName>** will be no longer than 20 characters.
5. Do not hard code the target file name “notes.zip” in the Bob program. During testing we will use different files.
6. Again, be very careful about binary equivalence.

Bob Class

The **`Bob`** program receives packets from **`Alice`** (through **`UnreliNET`**) and stores the received file in the current working directory, with a filename specified by **`Alice`**.

To run **`Bob`** on **`sunfire`**, type command:

```
java Bob <rcvPort>
```

For example,

```
java Bob 9001
```

listens on port 9001 and stores the received bytes into a file whose name is given by **`Alice`**. **`Bob`** does not have to terminate, i.e. it can run infinitely and does not need to detect end of the transmission and terminate. (This simplifies the implementation. Otherwise you would have to implement a full TCP teardown which is quite hard) You can assume that after **`Alice`** is terminated, **`Bob`** will not be reused for another communication attempt.

However, even though Bob does not have to terminate, Bob has to make sure that all the received data is saved to the hard drive. A good way to achieve this is to flush buffers and close the file stream properly once the last data is successfully received.

Running All Three Programs

You should first launch **`Bob`**, followed by **`UnreliNET`** in the second window. Finally, launch **`Alice`** in a third window to start data transmission. Please note that Alice and Bob take the ports (**`<unreliNetPort>`** and **`<rcvPort>`**, respectively) as command-line arguments as described above. Please always test your programs on **`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 Alice and Bob programs should not communicate with each other directly – all traffic has to go through the **`UnreliNET`** program. **Alice should terminate** once all inputs are read and properly forwarded. However, you may leave the Bob program running infinitely (i.e. no need for Bob to detect end of transmission and terminate).

You may also test your program using the following command:

```
bash test.sh <unreliNetPort> <rcvPort>
```

For example,

```
bash test.sh 9000 9001
```

If you run into issues running the test script, you may need to run the following first:

```
chmod u=rwx test.sh timeout
```

Tips and Hints

1. 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 the application layer. The following header/trailer fields might be needed (but you might also implement different ones):

- Sequence number
- Checksum

Note that each packet **`Alice`** sends should contain **at most 1024 bytes** of application data (inclusive of user-defined header/trailer fields), or **`UnreliNET`** will reject it.

2. Computing Checksum

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

Please refer to Assignment 0 Exercise 2 on how to compute checksum using Java **`CRC32`** class.

3. Timer and Timeout Value

Alice may have to run a timer for unacknowledged packet. You are suggested to use the **`setSoTimeout()`** method of Java **`DatagramSocket`** class and set the timeout value to 100ms.

4. Reading/Writing Values to Header/Trailer Fields

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

You may use **`ByteBuffer`** class from the **`java.nio`** package to form a packet containing various header/trailer and application message.