National University of Singapore
School of Computing

CS2105 **Assignment 2** Semester 1 AY17/18

## Submission Deadline

26 October 2017 (Thursday), 23:59pm sharp. This is a complex assignment. Please start early. Do not leave your submission to the last minute in case of an unforeseeable situation, e.g. network congestion. You can submit 99 times. We will only grade the last submission.

**2 points penalty** will be imposed on late submission (Late submission refers to submission or re-submission after the deadline).

## Group Work

This assignment should be solved individually. However, if you struggle, you are free to form a team with another student (max. team size is 2). Group submissions are subject to **2 points penalty** for each 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).

## Introduction

In this assignment, you will create a simple one-way chat[1] that sends messages and transmits files over the UDP protocol on top of an unreliable channel that may either corrupt or drop packets randomly (but will always deliver packets in order).

This programming assignment is **worth 10 points.**

## 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`.**

[1] We implement a one-way chat to keep the assignment simple.

## 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 programs to upload. **Please do not submit folders, archives, rar, or zip files**. Just submit all your java files.

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. We will not run any test cases 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. We will use a grading script.

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

  The received chat messages and files must be **identical** to the sent ones. You only need to implement one-way communication. Bob will not send messages or files to Alice (but Bob might have to send some feedback).

  The grading script **does care** what messages you print on the screen. Thus, make sure that you **remove all debug output** before you submit your solution; especially for Bob, only print

messages that were sent by Alice, nothing else. No points will be awarded if your output does not conform to the expected output.

## 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 (Please do not post to the public Internet, e.g. public repositories). Frequent backups will allow you to submit at least a partial solution that works in some scenarios described above (e.g. in a reliable channel) if you run out of time.

## 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 we find a suspicious case, we will award zero points and further disciplinary action might result.

**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 chat 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**. 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 chat messages and 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. This program is given and should not be changed.

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.

## 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 chat program. It reads from standard input line-by-line, wraps all data in UDP packets and sends them to **UnreliNet. UnreliNet** then forwards the packets to **Bob**.

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

```
java Alice <host> <unreliNetPort>
```

Substitute **<host>** with a valid host name, e.g. localhost. Substitute **<unreliNetPort>** with a valid port number, e.g. 9000.

For example,

```
java Alice localhost 9000
```

We strongly recommend to use localhost for all your development and testing. I.e., you run everything on the local computer. This minimizes interference from other network traffic.

**Alice** can send chat messages as well as files to **Bob**. To send a file after you started **Alice**, you type:

```
/send <path>
```

For example,

```
/send Alice.java
```

will send the file Alice.java in the current working directory to **Bob.** Everything that does not start with **/send** is a chat message and should be send to **Bob**. **Bob** should print out these chat messages to the screen. Please also refer to the description of Bob below.

For your convenience, the functionality to read the standard input and detect the **/send** command is already provided in the skeleton. You only have to implement the methods **sendFile** and **sendMessage.**

1. For this assignment, we will always run **UnreliNET**, **Alice** and **Bob** programs on the same host.
2. The **Alice** program should terminate only after (1) reading all input and (2) forwarding everything successfully to **Bob**. The skeleton already contains all the code to read from

standard input and differentiate between chat messages and **/send** commands.

3. After reading a line from standard input, **Alice** inspects it and send it to Bob if it is a normal chat message. For files, Alice should read the file, split it into packets and send it to Bob. The functionality to interpret the lines read from standard input is already provided in the skeleton.
4. Make sure that the file is binary equivalent (similar to assignment 0).

## Bob Class

**Bob** receives messages from **Alice** (through **UnreliNET**) and prints them on monitor.

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

```
java Bob <rcvPort>
```

For example,

```
java Bob 9001
```

listens on port 9001 and prints the messages received on standard output.

1. To solve this assignment correctly the Bob has to send feedback to the Alice (why?).
2. If you receive a message, print it to the screen using the **printMessage** function. Make sure you do not print anything else to the screen, in particular, remove all debug output before submission.
3. If you receive a file, please **store it in the current working directory as file "output".** If the file exists, you can simply overwrite it. (A real chat application should actually ask the user where to store the file, but we try to keep this assignment free from distractions. So please just store it in "output")
4. Make sure you close the file. If you leave the file open, data might not be flushed to the hard drive. Instead, it might be lost. (This is covered in CS2106 Introduction to Operating Systems)

## 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 argument 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 input is read and properly forwarded (i.e. the input stream is closed and everything in the input stream is successfully received by **Bob**). However, you may leave the Bob program running infinitely (i.e., no need for Bob to detect end of transmission and terminate).

## 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
- Some way to differentiate between binary data (files) and chat messages.

Note that each packet **Alice** sends should contain **at most 512 bytes** of application data (inclusive of user-defined header/trailer fields), or **UnreliNET** will reject it.

## 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.

## Timer and Timeout Value

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

You should use a timeout value of 100ms.

## 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.