National University of Singapore
School of Computing

CS2105 **Assignment 2** Semester 1 AY16/17

Submission Deadline

14 October 2016 (Friday), 1pm sharp. Do not leave your submission to the last minute in case of unforeseeable situation, e.g. network congestion. You have 99 chances of submissions and only the last submission will be graded.

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

Introduction

In this assignment, you will create a simple one-way chat that sends messages 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 9 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`** 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 programs to upload. Please do not submit folders or zip 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

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

- **[3 points]** Programs are compilable on **`sunfire`**; 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 are also named correctly.
- **[2 points]** Programs can successfully send chat messages from sender to receiver in a perfectly reliable channel (i.e. no error at all).
- **[2 points]** Programs can successfully send messages from sender to receiver in the presence of both data packet corruption and ACK/NAK packet corruption.
- **[2 points]** Programs can successfully send messages from sender to receiver in the presence of both packet corruption and packet loss.

  The received chat messages must be identical to the sent ones. You only need to implement one-way communication. Thus, the sender will read messages from standard input and send them via UDP to the receiver. The receiver will read the messages and print them to standard output. The receiver will not send messages to the sender (but receiver may have feedback for sender).

  The grading script **does care** what messages your programs print on the screen. Thus, make sure that you **remove all debug output** before you submit your solution and print only messages that you received, nothing else. No point 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 to write 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). This will allow you to submit a partial solution that works in some scenarios described above (e.g. in a reliable channel) if you run out of time.

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, zero marks will be awarded and further disciplinary action might result.

You are not allowed to post your solutions in any public domain on the Internet.

Overall Architecture

There are three programs in this assignment, **Sender**, **UnreliNET** and **Receiver**. Their relationship is illustrated in Figure 1 below. The **Sender** and **Receiver** 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 **Sender** and **Receiver**. Instead of sending packets directly to **Receiver**, **Sender** sends all packets to **UnreliNET**. **UnreliNET** may introduce bit errors to packets or lose packets randomly. It then forwards packets (if not lost) to **Receiver**. When receiving feedback packets from **Receiver**, **UnreliNET** may also corrupt them or lose them with certain probability before relaying them to **Sender**.

The **UnreliNET** program is complete and given. Your task in this assignment is to develop the **Sender** and **Receiver** programs so that receiver will receive chat messages successfully in the presence of packet corruption and packet loss.

Sender Class

The **Sender** program is a very simple one-way chat program. It reads messages from standard input line-by-line, wraps them in UDP packets and sends them to **UnreliNet. UnreliNet** then forwards the packets to **Receiver**.

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

```
java Sender <unreliNetPort>
```

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

Notes:

1. For this assignment, we will always run **UnreliNET**, **Sender** and **Receiver** programs on the same host.
2. After reading a line of chat message from standard input, the **Sender** program should send it to receiver immediately.
3. The **Sender** program should terminate only after (1) reading all input and (2) forwarding it successfully to the **Receiver**.

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 **Receiver** 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 **Sender**, 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.

Receiver Class

The **Receiver** program receives messages from **Sender** (through **UnreliNET**) and prints them on monitor.

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

```
java Receiver <rcvPort>
```

For example,

```
java Receiver 9001
```

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

Notes:

1. To solve this assignment correctly the receiver has to send feedback to the sender (why?).

Running All Three Programs

You should first launch **`Receiver`**, followed by **`UnreliNET`** in the second window. Finally, launch **`Sender`** in a third window to start data transmission. Please note that Sender and Receiver 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 sender and receiver programs should not communicate with each other directly – all traffic has to go through the **`UnreliNET`** program. The sender program 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 **`Receiver`**). However, you may leave the receiver program running infinitely (i.e., no need for receiver 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 application layer. The following header/trailer fields are suggested though you may have your own design:

- Sequence number
- Checksum

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

Computing Checksum

To detect bit errors, **`Sender`** should compute checksum for every outgoing packet and embed it in the packet. **`Receiver`** 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

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

You should use a timeout value of 500ms.

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 sender and receiver (i.e. an application layer protocol designed by you).

As will be discussed in the week 8 tutorial, you may use **`ByteBuffer`** class from the **`java.nio`** package to form a packet containing various header/trailer and application message.

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.