Project 1

20 points

Write a sender program and a receiver program, where the sender accepts a file as a command line parameter (any binary file on your hard disk), breaks it into smaller chunks (assume at least 12 chunks for your file), and sends it to the receiver using UDP.

The receiver will concatenate the pieces it receives and will store it to a file on its end. Note that you would have at least 12 datagrams sent by the sender to the receiver.

The sender and receiver will write out a detailed log of what they are doing to the console.

- for each datagram sent, the sender will write the [packet#]-[start byte offset]-[end byte offset].
- for each datagram received, the receiver will write the [packet#]-[start byte offset]-[end byte offset]

Note that you must:

- use the UDP protocol for this project.
- transmit a binary file (image or sound file)

Upload the completed Java code to your dropbox.

Project 2 – Stop and Wait

80 points

This project provides an all-inclusive implementation experience of major topics in Computer Networks, including protocol design and implementation.

The User Datagram Protocol (UDP) provides point-to-point, unreliable datagram service between a pair of hosts. It does not provide any reliability or sequencing guarantees – i.e. packets may arrive late, may not arrive at all, or arrive corrupted.

Your project is to implement a reliable and sequenced message transport protocol on top of this unreliable UDP (stop and wait). Your protocol will ensure reliable, end-to-end delivery of messages in the face of packet loss and packet corruption.

In order to simulate a lossy network on a single computer, you will implement a proxy that randomly drops, or corrupts packets.

Whereas TCP allows fully bidirectional communication, your implementation will be asymmetric. Each endpoint will play the role of a "sender" and a "receiver". Data packets will only flow from the sender to the receiver, while ACKs will only flow in the "reverse" direction from the receiver back to the sender.

(Project inspired by my project assignment at UMN and more recently by Stefan Savage, UC San Diego.)

Implementation Notes:

You will implement the sender and receiver components of a transport layer. The sender reads a stream of data (from a file), breaks it into fixed-sized packets suitable for UDP transport, prepends a control header to the data, and sends each packet to the receiver. The receiver reads these packets and writes the corresponding data, in order, to a reliable stream (a file).

A high-level overview of the system:

Input File → Sender → data datagrams travel over UDP → Receiver → Output File

← ACK datagrams travel over UDP ←

The Receiver should exactly reproduce the Sender's input file's content in its output file, regardless of a **lossy**, or **corrupting** network layer.

- You will ensure reliable transport by having the Receiver acknowledge packets received from the Sender; the Sender will detect missing acknowledgements and resend the dropped or corrupted datagrams after a timeout period. The default timeout period is 2000ms, but you may change this with the `-t` command-line option.
- The Receiver will only use positive ACKs for any datagrams that it receives.
- Acknowledgement datagrams do not need to be acknowledged by the Sender.
- Your Receiver should ensure that data are written in the correct order.
- For stop-and-wait, set the window size at both the sender and receiver to 1.
- File data will be binary.
- Because we are doing this between two processes on a single machine, latency is near 0, and no network faults can occur. As a result, you should introduce errors to test the correctness of your implementation. The Sender and Receiver programs must both accept options to force packets to be lost or corrupted at both processes.
- **User Interface**
  Design a simple UI (command line, or optionally GUI).
  (`java edu.metrostate.Sender -s 100 -t 30000 -d 0.25 receiver_ip_addr receiver_port`
  `java edu.metrostate.Receiver -d 0.5 receiver_ip_addr receiver_port`)
  The user may specify:
  - For the Sender:

Deleted: , and will preserve message ordering in the face of arbitrary latencies due to multiple paths taken by packets.

Deleted: delays,

Deleted: ←

Deleted: , congested

        - size of packet using the `-s` command line argument
        - timeout interval using the `-t` command line argument
        - the IP address and port at which the receiver is listening
    - For the Receiver
        - the IP address and port at which the Receiver should listen
    - For the Sender and Receiver:
        - percentage of datagrams to corrupt, or drop using the `-d` argument

Deleted: delay,

- You will start your Sender and Receiver in two separate JVMs.
    - Both programs should run outside of your IDE - do not launch them from Eclipse
    - Both windows should be open side by side, so you can watch the output from each
    - Windows should have reasonable contrast and font size so text is easily readable (use black text on a white background, and a medium to large font)
    - Your programs should present enough information to demonstrate the protocol in action. This is important – I will not take your word for what is happening unless there is independent corroboration via stdout messaging.
    - The information provided in the **sender's window** should include (in fixed width columns):
        - For each datagram attempted to be sent:
            - Static text:
                - If first time datagram is being sent: `[SENDing]:`
                - If resending a datagram: `[ReSend.]:`
            - sequence number of datagram
              `integer, [0,(size of file)/(size of packet)]`
            - byte sequence carried in datagram:
              `[<start byte offset>:<end byte offset>]`
            - datagram sent time in milliseconds
            - datagram condition depending on random error `[SENT|DROP|ERRR]`
        - For each ACK received
            - Static text `[AckRcvd]:`
            - Sequence number of datagram that was ACKed
            - Static text:
                - if duplicate ACK received: `[DuplAck]`
                - if corrupted ACK is received: `[ErrAck.]`
        - For each timeout event
            - Static text `[TimeOut]:`
            - Sequence number of datagram that timed out
        - Representative Examples (not intended to be complete):
            - first time sending of a datagram
              `SENDing 1 0:10 100234456 SENT`
            - resending a datagram that has timed out
              `ReSend. 1 0:10 122234456 SENT`
            - first time sending of a datagram, but datagram was dropped and never sent
              `SENDing 1 0:10 100234456 DROP`
            - first time sending of a datagram, but datagram was corrupted by the network
              `SENDing 1 0:10 100234456 ERRR`
            - ACK received for frame 1, and window will move
              `AckRcvd 1 MoveWnd`
            - ACK received for frame 1, but had an error, so window will not move
              `AckRcvd 1 ErrAck.`
            - Duplicate ACK received for frame 1:
              `AckRcvd 1 DuplAck`
            - Timeout occurred for datagram 1:
              `TimeOut 1`
    - The information provided in the **receiver's window** should include (fixed width output), for each datagram received:
        - Static text

Deleted: <#>if ACK will move window: `[MoveWnd]`¶

- if first time datagram is being received: `[RECV]`
- if duplicate datagram received: `[DUPL]`

- datagram received time in milliseconds
- Sequence number of the received data datagram
- Condition of data Datagram
  - if datagram is corrupt: `[CRPT]`
  - if datagram is received out of sequence: `[!Seq]`
  - if datagram is good: `[RECV]`
- For each ACK datagram, decision for the ACK: `[DROP|SENT|ERR]`
  Use the same rules as the Sender.
- Representative Examples (not intended to be complete):
  - Datagram 1 was received successfully
    `RECV 100234456 1 RECV`
  - Datagram 1 was received a second time (duplicate)
    `DUPL 111234456 1 !Seq`
  - Datagram 1 was received but with an error
    `RECV 100234456 1 CRPT`
  - ACK for Datagram 1 was created and sent successfully
    `SENDing ACK 1 100234456 SENT`
  - ACK for Datagram 1 was created but was dropped by the network
    `SENDing ACK 1 100234456 DROP`
  - ACK for Datagram 1 was created but was corrupted by the network
    `SENDing ACK 1 100234456 ERR`

It is important that your console output explains the behavior of your software program as it reacts to packets. If the output does not match the examples, I will interpret your program's behavior based on what I do see.

Implementation Details

Packet Types and Fields

There are two kinds of packets, Data packets and Ack-only packets. You can tell the type of a packet by its length.

```
public class Packet {
 short cksum; //16-bit 2-byte
 short len; //16-bit 2-byte
 int ackno; //32-bit 4-byte
 int seqno ;   //32-bit 4-byte Data packet Only
 byte data[500]; //0-500 bytes. Data packet only. Variable
}
```

- **cksum**: 2 byte IP checksum. Use 0 for good, 1 for bad. Not expecting you to compute the checksum.
- **len**: 2 byte total length of the packet.
  - For Ack packets this is 8: 2 for cksum, 2 for len, and 4 for ACK no
  - For data packets, this is 12 + payload size: 2 for `cksum`, 2 for `len`, 4 for `ackno`, 4 for `seqno`, and as many bytes are there in `data[]`

    Note: You must examine the length field, and should not assume that the UDP packet you receive is the correct length. The network might truncate or pad packets.
- **ackno**: 4-byte cumulative acknowledgment number.
  `ackno` is the sequence number you are waiting for, that you have not received yet – it is the equivalent of Next Frame Expected.
  This says that the sender of a packet has received all packets with sequence numbers earlier than `ackno`, and is waiting for the packet with a `seqno` of `ackno`.
  The first sequence number in any connection is 1, so if you have not received any packets yet, you should set `ackno` to 1.

Formatted: Font: Bold

The following fields will not exist in an ACK packet:

- **seqno**: Each packet transmitted in a stream of data must be numbered with a `seqno`. The first packet in a stream has a `seqno` of 1. This protocol numbers packets.
- **data**: Contains (`len - 12`) bytes of payload data for the application.
  To conserve packets, a sender should not send more than one unacknowledged Data frame with less than the maximum number of bytes (500)

Use a datagram with an empty `data[]` to indicate the end of your stream.

Demonstration

You will be asked to demonstrate your project to the class. This is a critical component of your project, as your grade will depend on your ability to explain your implementation. All members of a team should be present at the demonstration - no exceptions! I will pick the team member who will drive the demonstration.

Project

All project source code must be checked into D2L by 6pm the day they are due.

Implement the above requirements using the Stop and Wait protocol.

Grading criteria:

- Working Implementation: 100%
  - Deliver a binary file successfully to the receiver process (split the input file into chunks, use the correct header format, handle binary files, reassemble the file at the destination.) 30%
  - Introduce and handle errors at the receiver (DROP, CRPT) 35%
  - introduce and handle errors at the sender (DROP, CRPT) 35%
- You must demonstrate your project to me in order for points to be awarded.

If you are confused about a requirement, please let me know as soon as possible.

Ideally you have been doing this early enough in the semester to ensure that you get a timely response. If it is too late to reach me, please go ahead and make a reasonable assumption.

A "reasonable assumption" is one where a majority out of 3 random peers in class (I get to pick) would agree with your assumption. Any assumptions that you make must be recorded in a technical paper that you submit along with your project.