Programming Assignment 2: Reliable Transport over UDP
CSEE 4119, Spring 2019
Due: May 4, 2019

In this assignment, you will build your own reliable transport protocol, over UDP. Your protocol must provide in-order, reliable delivery of UDP datagrams, and must do so in the presence of packet loss, delay, corruption, duplication, and re-ordering.

To reduce your workload, we will provide you with the receiver and you will implement the sender side. Since you probably cannot modify the OS on your machine, we have included a simulated channel in the receiver side that drops, delays, and corrupts packets, and effectively simulates an unreliable channel. We will also provide several useful functions that you can use in your implementation.

1. Protocol Description

Our simple protocol has four packet types: *start*, *end*, *data*, and *ack*. **All packets** follow the general format below.

Packet specification:

checksum	seqnum	flag	optional	data

- checksum: (4 bytes): 32-bit checksum
- seqnum: (4 bytes): sequence number
- flag (1 byte): start packet: 0, data packet: 1, end packet: 2, special packet: 3, ack packet: 4
- optional (1 byte): optional field if needed for performance improvement
- data: at most (MAX_SIZE - 10) bytes

To initiate a connection, send a *start* packet. The receiver will use the sequence number provided in the start packet as the initial sequence number for all packets in that connection. After sending the start packet, send additional packets in the same connection using the data packet type, adjusting the sequence number appropriately. The last data in a connection should be transmitted with the end packet type. If the file to transmit only requires one packet, that packet is sent with a special packet flag.

A single packet will not exceed MAX_SIZE bytes. Because of the constraints on the total size of packets in UDP and layers below, MAX_SIZE cannot be more than ~1400 bytes. You should set MAX_SIZE to 500 bytes in this assignment.

Receiver Description:

We will provide you the receiver Receiver.py for you. The receiver responds to data packets with *cumulative acknowledgements*. Upon receiving a packet of type start, data, special, or end, the receiver generates an ack packet with the sequence number it expects to receive next, which is the lowest sequence number not yet received in order.

Receiver has a maximum window size WND_SIZE which is set to 10 packets in this assignment. This means if the next expected packet is N, the receiver will not accept out-of-order packets with sequence number greater than N+ WND_SIZE.

The receiver has a default timeout of 5 seconds, this means it will automatically close any connections for which it does not receive packets for that duration.

To simulate an unreliable channel, we have used a function channel(packet), defined in utils.py, that receives the received packet, and two constants, packet drop probability PROB_LOSS and packet corruption probability PROB_CORR, and drops or corrupts the packet according to those probabilities. It further delays all the packets by some random time in DELAY_RANGE currently set to interval (80, 120) msecs. The output of channel function is either a packet (corrupted or uncorrupted) or *None* (in the case of packet drop). The output of this function has been considered as what is actually received by the receiver to emulate an unreliable channel.

The receiver does not send back any ack if the packet is dropped (output of channel is *None*), otherwise it sends an ack packet.

The receiver is invoked with the following command:

python Receiver.py <output file> <receiver address> <receiver port>

Output file is the name of the file where the receiver writes data, and receiver address could be simply localhost and receiver port is the UDP port number used by the receiver.

Sender specification

Your sender should read an input file and transmit it to a specified receiver using UDP sockets. It should split the input file into appropriately sized chunks of data, specify an initial sequence number for the connection, and append a checksum to each packet.

Function for generating packet checksums will be provided for you (see utils.py), also functions for making packet, and extracting fields from a packet. In utils.py, we have also provided you with read_file function that divides the file into chunks of size chunk_size. For example you can use f = read_file(filename, chunk_size), and f.next() to return the next chunk.

Your sender must implement a reliable transport algorithm using a sliding window mechanism. Your sender must be able to accept ack packets from the receiver. Any ack packets with an invalid checksum should be ignored. You can place sequence number 0 in the start packet.

Your sender should provide reliable delivery of file under the following network conditions:

- Loss: arbitrary levels
- Corruption: arbitrary types and frequency.
- Re-ordering: packets may receive in any order, and
- Delay: packets may be delayed (here randomly in DELAY_RANGE).

Your sender should be invoked with the following command:

python Sender.py <input file> <receiver address> <receiver port>

- The sender should implement a 500ms timeout interval to automatically retransmit packets that were never acknowledged (potentially due to ack packets being lost). We do not expect you to use an adaptive timeout (though this is one of the bonus options). Further the sender should perform fast retransmission, i.e., upon the receipt of three duplicate acks, it should transmit the corresponding packet without waiting for timeout.
- Sender should only retransmit the oldest unacknowledged packet (rather than naively retransmitting the last N packets).
- Your sender should support a window size W of at most WND_SIZE packets (we do not expect adaptive window size but this is one of the bonus options).
- Your sender should roughly meet or exceed the performance (in both time and number of packets required to complete the file transfer) of a properly implemented sender.
- Your sender should be able to handle arbitrary files (i.e., it should be able to send an image file just as easily as a text file).
- Any ack packets received with an invalid checksum should be ignored.
- Your sender MUST NOT produce console output during normal execution. Python exception messages are ok, but try to avoid having your program crash in the first place.
- We will evaluate your sender and receiver on correctness, time of completion for a file transfer, and number of packets sent (and re-sent). Transfer completion time and number of packets used in a transfer will be measured against our own reference implementation of a sliding-window based sender. Note that a “Stop-And-Go” sender (i.e., without pipelining) will not have adequate performance.

2. Bonus Points:

Congestion Control (10 points): When packet loss, corruption, delay, and reordering are minimal, a large window size will obtain higher performance. A large window on a lossy channel, however, will lead to a large amount of overhead due to retransmissions. Modify your sender to dynamically adjust its window size based on a variant of TCP congestion control mechanism.

Accounting for variable round-trip times (5 points): Modify your sender to determine the round trip time between the sender and the receiver, and adjust the timeout interval appropriately through a similar mechanism to TCP.

Bidirectional transfer (5 points): While our protocol as defined only supports unidirectional data transfer, it could be modified to allow bi-directional transfer (i.e., both ends of the connection could send and receive simultaneously). Implement this functionality by modifying both the sender and receiver as necessary.

Be sure to provide a description of your updated protocol in your README.txt file.

3. Downloads

The utils.py and Receiver.py are on the course webpage.

4. What to hand In

Submit a zip file using the format <UNI>. zip (e.g. jg3465.zip) to PA2 in courseworks. Your zip file must include:

- Python file Server.py (basic sender)
- README.txt file: any general instructions about running your code. If you choose to do bonus parts, you can additionally include:
- Sender2.py
- Receiver2.py

Notes

- Your implementation should work on any system that runs the appropriate version of Python 2.7.x.
- Make sure your code is well commented and readable or you will be taken 5-10 points.
- If your code doesn't compile, we will call you to have a look at it and fix it. However, it will result in a deduction of 20% of your total points. We will use Python 2.7.x to test your code.
- The basic implementation has at most 100 points.
- You are permitted and encouraged to help each other through Piazza. However, you may not share source code. Refrain from getting any code off the Internet. Please read the academic integrity policy in the first lecture.
- To streamline the process, we have dedicated three TAs solely to this project:
    - Georgia Essig
    - Brayn Fridman
    - Christos Tsanikidis

  These TAs will be answering your questions about PA2 and can help during their office hours with this project.