Project 2: Reliable Data Transfer

Project Description

In this project, you will build a simple reliable transport protocol **on top of UDP**. Your RDT implementation must provide in order, reliable delivery of UDP datagrams in the presence of events like packet loss, delay, corruption, duplication, and reordering.

Language

You will write your code in Python, and test it on Linux machines from command line. Do not use libraries that are not installed by default in Linux. Similarly, your code must compile and run on the command line. You may use IDEs (e.g. Eclipse) during development. Make sure you code has no dependencies on your IDE.

RDT Protocol

RDT sends data in the format of a header, followed by a chunk of data.

RDT has four header types: `START`, `END`, `DATA`, `ACK` and `END_ACK` all following the same format:

```
PacketHeader:
  int type;      // 0: START; 1: END; 2: DATA; 3: ACK
  int seq_num;   // Described below
  int length;    // Length of data; 0 for ACK, START and END packets
  int checksum;  // 32-bit CRC
```

To initiate a connection, sender starts with a `START` message along with a random seq_num value, and wait for an `ACK` for this `START` message. After sending the `START` message, additional packets in the same connection are sent using the `DATA` message type, adjusting seq_num appropriately. After everything has been transferred, the connection should be terminated with sender sending an `END` message, and waiting for the corresponding `END_ACK` for this message.

The ACK seq_num values for `START` and `END` messages should both be set to whatever the seq_num values are that were sent by sender.

Packet Size

An important limitation is the maximum size of your packets. The UDP protocol has an 8 byte header, and the IP protocol underneath it has a header of 20 bytes. Because we will be using Ethernet networks, which have a maximum frame size of 1500 bytes, this leaves 1472 bytes for your entire packet structure (including both the header and the chunk of data).

The appropriate packet size and payload size have been defined in *rdt.py*.

Code Organization

Overall, there are four components in this assignment:

Part 1: *util.py*

Class `UnreliableSocket` and `PacketHeader` are defined in this file. Class `UnreliableSocket` defines an unreliable socket that 'inherits' several functions and the unreliable features of UDP sockets. The 'inherited' functions include `bind()`, `recvfrom()`, `sendto()` and `close()`. Function `recvfrom()` simulates packet loss, packet delay and packet corruption scenarios. You don't need to modify any of these functions in your implementation.

Class `PacketHeader` specifies the format of RDT packets. There are also some additional auxiliary functions. `compute_checksum()` calculates the crc32 checksum value and `verify_packet()` verifies the integrity of the received segments. Both functions will be needed when encapsulating and processing RDT packets.

You do **NOT** need to make any changes in this file.

Part 2: *rdt.py*

Class `RDTSocket` is defined in this file. This class inherits all the methods and properties from class `UnreliableSocket`. There are five key functions you need to implement:

- `accept()`: Similar to the accept function defined in TCP protocol, it is invoked by the receiver to establish 'connections' with the sender. The return value is the address (tuple of IP and port number) of the sender. You do not need to create a new connection socket like TCP does.

- `connect()`: Similar to the connect function defined in TCP, it is invoked by the sender to initiate connection request with the receiver.
- `send()`: This function is invoked by the sender to transmit data to the receiver. This function should split the input data into appropriately sized chunks of data, and append a checksum to each packet. *seq_num* should increment by one for each additional segment in a connection. Please use the `compute_checksum()` function provided in *util.py* for the 32-bit CRC checksum.
- `recv()`: This function is invoked by the receiver to receive data from the sender. This function should reassemble the chunks and pass the message back to the application process. Use the `verify_packet()` function provided in *util.py* to check the integrity of the segments.
- `close()`: This function is invoked by the sender to terminate the connection between the sender and the receiver.

When implementing `send()` and `recv()`, you **MUST** use the `sendto()` and `recvfrom()` functions defined in *util.py*.

You will implement reliable transport using a sliding window mechanism. The size of the window (`window_size`) will be specified in the command line. The sender must accept cumulative `ACK` packets from the receiver.

After transferring the entire message, the sender will send an `END` message to mark the end of connection.

The sender must ensure reliable data transfer under the following network conditions:

- Packet loss;
- Reordering of ACK messages;
- Duplication of any amount for any packet;
- Delay in the arrivals of ACKs.

To handle cases where `ACK` packets are lost, you should implement a 500 ms retransmission timer to automatically retransmit segments that were not acknowledged. Whenever the window moves forward (i.e., some ACK(s) are received and some new packets are sent out), you reset the timer. If after 500 ms the window still has not advanced, you retransmit all segments in the window because they are all never acknowledged (not just the missing segment).

The receiver needs to handle only one sender at a time and should ignore `START` messages while in the middle of an existing connection. It should also calculate the checksum value for the data in each segment it receives with its header information. If the calculated checksum value does not match the `checksum` provided in the header, it should drop the packet (i.e. not send an `ACK` back to the sender).

For each segment received, it sends a cumulative `ACK` with the `seq_num` it expects to receive next. If it expects a packet of sequence number `N`, the following two scenarios may occur:

1. If it receives a packet with `seq_num` not equal to `N`, it will send back an `ACK` with `seq_num=N`. Note that this is slightly different from the Go-Back-N (GBN) mechanism discussed in class. GBN totally discards out-of-order packets, while here the receiver buffers out-of-order packets.
2. If it receives a packet with `seq_num=N`, it will check for the highest sequence number (say `M`) of the in-order packets it has already received and send `ACK` with `seq_num=M+1`.

If the next expected `seq_num` is `N`, the receiver will drop all packets with `seq_num` greater than or equal to `N + window_size` to maintain a `window_size` window.

Part 3: *sender.py*/*receiver.py*

The sender and receiver application process are programmed with the socket APIs implemented in Part 2. The sender transmits a text file, `alice.txt`, to the receiver. The receiver will write the received data to file `download.txt`.

The sender should be invoked as follows:

```
python sender.py [Receiver IP] [Receiver Port] [Window Size]
```

- `Receiver IP`: The IP address of the host that the receiver is running on.
- `Receiver Port`: The port number on which the receiver is listening.
- `Window Size`: Maximum number of in-flight segments.

The receiver should be invoked as follows:

```
python receiver.py [Receiver Port] [Window Size]
```

- `Receiver Port`: The port number on which `receiver` is listening for data.
- `Window Size`: Receiver window size.

You do **NOT** need to make any changes in these files. To test your code, you could compare `download.txt` with `alice.txt` with the following commands:

```
diff download.txt alice.txt
```

If nothing is printed, then your code is correct.

Tips

- You could assume the communication between the sender and receiver is half-duplex, which means that only the sender is allowed to send data to the receiver.
- To implement timer, you could calculate the time elapsed between the current time (by invoking `time.time()` in Python) and a previous timestamp. If this value exceeds 500 ms, then a timeout event happens.
- You do not need to consider efficiency when implementing this protocol. Thus, there is no need to implement concurrency in your code.

REMINDERS

- All your source code should be placed in a directory. Submit your code to Canvas in a zip file by running command in terminal:

```
zip -r [you-case-id]-proj-2.zip [project directory]
```

- If your code could not **run**, you will not get credits.
- All student code will be scanned by plagiarism detection software to ensure that students are not copying code from the Internet or each other. We will randomly check student's code to make sure that it works correctly.
- Document your code (by inserting comments in your code)
- **DUE: 11:59 pm, Wednesday, April 28th**.