CS 342: Computer Networks Lab

(July-November, 2022)

Assignment - 3: Reliable Transport

Submission deadline: 11:55 PM on Monday, 26th September 2022 [Hard deadline].

Viva-voce Date & Time: Lab session on Monday, 10th October 2022 (ML-1: 9:00 am to 11:55 am)

This assignment is a programming assignment where you need to implement Reliable Data Transfer of Transport Layer using C programming language. The assignment will be solved in groups where each group is comprised of 3-4 members. The **group membership information will be informed separately**. The applications' description and requirement specification are given in this document.

Instructions:

1. Each group needs to implement the required functions and make one single submission on Moodle/MSTeams (whichever is used by the Instructor). Only one member from a group needs to make the submission.
2. Writing and testing of codes need to be performed on a Unix/Linux-based computer (i.e. the operating system must be from the Unix or GNU/Linux family).
3. Submit the **set of source code files** of the application as a zipped file on Moodle/MSTeams (maximum file size is 10 MB) by the deadline. The **ZIP file's name should be the same as your group number**, for example, "Group_4.zip", or "Group_4.rar", or "Group_4.tar.gz".
4. The assignment will be evaluated offline/through viva-voce during your lab session where you will need to explain your source codes and execute them before the evaluator (evaluation schedule and TA allocation will be notified in due time).
5. **Write your own source codes and do not copy from any source. Plagiarism detection tool will be used and any detection of unfair means will be penalised by awarding NEGATIVE marks (equal to the maximum marks for the assignment).**

In this assignment, you have to implement a **reliable sliding window transport layer** on top of the user datagram protocol (UDP).

Your implementation should have the following features:

- Handle packet drops
- Handle packet corruption
- Allow multiple packets to be outstanding
- Handle packet reordering
- Detect any single-bit errors in packets

For this assignment, **you will be provided with the necessary library files** (*rlib.h* and *rlib.c*) and you need to implement some data structures and functions in ***reliable.c***. **But, don't modify library files (*rlib.h* and *rlib.c*).**

You will implement both the sender and receiver component of a transport layer. The sender will read a stream of data in, break it into fixed-sized packets suitable for UDP transport, prepend a control header to the data, and write this packet to the receiver. The receiver will read these packets, and write the corresponding data, in order, to a reliable stream.

Understanding the code

Packet format

There are two types of packets - Data packets and Acknowledgement (Ack) packets. Ack packets are 8 bytes, while Data packets vary from 12 to 512 bytes. These data structures are defined in *rlib.h* library file.

```
struct packet {
  uint16_t cksum;          /* Ack and Data */
  uint16_t len;            /* Ack and Data */
  uint32_t ackno;          /* Ack and Data */
  uint32_t seqno;          /* Data only */
  char data[500];          /* Data only, Not always 500 bytes, can be less */
};

typedef struct packet packet_t;
```

Every Data packet contains a 32-bit sequence number and 0 or more bytes of payload. Both Data and Ack packets contain the following fields. The length, seqno, and ackno fields are always in big-endian order (meaning you will have to use htonl/htons to write those fields and ntohl/ntohs to read them).

- **cksum**: 16-bit IP checksum (you can set the cksum field to 0 and use the *cksum(const void *, int)* function on a packet to compute the value of the checksum that should be in there). Note that you shouldn't call htons on the checksum value produced by the *cksum* function -- it is already in network byte order.
- **len**: 16-bit total length of the packet. This will be 8 for Ack packets, and 12 + payload-size for data packets (since 12 bytes are used for the header). An end-of-file condition is transmitted to the other side of a connection by a data packet containing 0 bytes of payload, and hence a len of 12. 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**: 32-bit cumulative acknowledgment number. 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. Note that the ackno is the sequence number you are waiting for, that you have not received yet. The first sequence number in any connection is 1, so if you have not received any packets yet, you should set the ackno field to 1.

The following fields only exist in a data packet:

- **seqno**: Each packet transmitted in a stream of data must be numbered with a seqno. The first packet in a stream has seqno 1. Note that in TCP, sequence numbers indicate bytes, whereas by contrast this protocol just numbers packets. That means that once a packet is transmitted, it cannot be merged with another packet for retransmission. This should simplify your implementation.
- **data**: Contains (len - 12) bytes of payload data for the application.

reliable_state data structure: The **reliable_state** data structure is defined in ***reliable.c*** and typedefed as **rel_t** in rlib.c. This should store the details about the connection. You have to add any other data field in this data structure if it is required for your implementation. All other data structures present in rlib.h and rlib.c are already defined completely.

```
struct reliable_state {
  rel_t *next;      /* Linked list for traversing all connections */
  rel_t **prev;

  conn_t *c;      /* This is the connection object */

  /* Add your own data fields below this */
};
```

Functions and data structures you need to implement:

The reliable transport protocol connects the standard input and output of the sender and receiver processes together. You are provided with a library (*rlib.h* and *rlib.c*).

Your task is to implement the following six functions:

- **rel_create():** This function creates a **rel_t** object. This function is directly called by the library function. All the information corresponding to a connection should be initialized here.

- **rel_destroy():** A rel_t is deallocated by rel_destroy(). The library will call rel_destroy when it receives an ICMP port unreachable. When **all** of the following scenarios occurs, you should also call rel_destroy().
    - Read an EOF from the other side (i.e., a Data packet with 0 bytes payload field).

    - Read an EOF or error from your input (conn_input returned -1).
    - All packets you sent have been acknowledged.

  At least one side should also wait around for twice the maximum segment lifetime in case the last ack it sent got lost.

- **rel_recvpkt():** When a packet is received, the library will call *rel_recvpkt()* and provide you with the rel_t object.
- **rel_read():** To get the data that you must transmit to the receiver, call conn_input(). conn_input() reads from standard input. If no data is available, conn_input will return 0. At that point, the library will call rel_read once data is available again, so that you can once again call conn_input. (Do not loop calling conn_input if it returns -1, simply return and wait for the library to invoke rel_read!)
- **rel_output():** To output data you have received in decoded UDP packets, call *conn_output*. This function outputs data to STDOUT. You may find the function *conn_bufspace* useful--it tells you how much space is available for use by *conn_output*. If you try to write more than this, *conn_output* may return that it has accepted fewer bytes than you have asked for. You must flow control the sender by not acknowledging packets if there is no buffer space available for *conn_output*. The library calls rel_output() when output has drained, at which point you can call conn_bufspace to see how much buffer space you have and send out more Acks to get more data from the remote side.
- **rel_timer():** The function rel_timer is called periodically, currently at a rate 1/5 of the retransmission interval. You can use this timer to inspect packets and retransmit packets that have not been acknowledged. Do not retransmit every packet every time the timer is fired! You must keep track of which packets need to be retransmitted.
- You don’t need to implement rel_demux() function for this assignment.

Requirements

Your transport layer must support the following. [Corresponding marks are also mentioned in brackets.]

a. The sequence number of the first packet in a new connection is always 1. [2]
b. You will ensure reliable transport by having the acknowledge messages received from the receiver, so that the client may resend dropped or corrupted packets. [2+2]
c. Receiver should be able to detect error in received packets. It can also detect error in packet format (in header length). [2+2]
d. Acknowledgements should be cumulative rather than selective. You acknowledge the next sequence number you are expecting to receive, which is 1 more than the largest in-order sequence number you have received. You don't have to handle sequence number overflowing and wrapping in the lifetime of a connection. [2+2]
e. You must support a window size larger than 1. The window size is supplied by the -w command-line option, which will show up as the *window* field in the config_common data structure passed to the rel_create function you implement. [2]
f. Your sender and receiver should ensure that data is printed in the correct order, even if the network layer reordered packets. Your receiver should buffer as many packets as the sender may send concurrently. In other words, the sender window size (SWS) should equal the receiver window size (RWS), and both should be the same as the *window* field in the config_common structure. [3+2]
g. The sender should resend a packet if the receiver does not acknowledge it within an appropriate time period. You can send packet(s) whenever a sent packet has gone unacknowledged for the timeout period. The timeout period in milliseconds is supplied to you by the *timeout* field of the config_common structure. The default is 2000 msec, but you may change this with the -t command-line option. [3+2]
h. You can retry packets infinitely many times, and should make sure you retry at least 5 times, after which if you want, the sender can terminate the connection with an error. You can call rel_destroy to destroy the state associated with a connection when you give up on retransmitting. [2]
i. Connection closing should be implemented properly as mentioned in rel_destroy. [2]

Note: For debugging printfs, you should use the Standard Error *fprintf* (*stderr*, ...) and not print on standard output. This is because standard output is being used for the actual program output and it will be confusing for the grader as well as the tester. You can use **print_pkt**() function, defined in the library, to print all the packets sent and received.

Instructions for code compilation and execution

Download and untar the **reliable.tar.gz**. (This will be provided with the assignment)

Run the command "**make**" to build the *reliable* program.

To execute the program, use the following commands with your own arguments.

See the examples below:

In Terminal 1 / machine 1:

```
./reliable -w 5 6666 localhost:5555
```

where -w option is to set window size. This command also binds this process to port no. 6666, i.e. the process is listening on UDP port 6666, and be connected with localhost (i.e. other client) at port no. 5555

In Terminal 2 / machine 2:

```
./reliable -w 5 5555 localhost:6666
```

where -w option is to set window size. This command also binds this process to port no. 5555, i.e. the process is listening on UDP port 5555, and be connected with localhost (i.e. other client) at port no. 6666

Acknowledgement

This lab assignment is adapted from one of the assignments used in CS 144 at Stanford University. We thank to the course instructor and all staffs at Stanford University who prepared this assignment.