Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Spring Term 2019

Computer Networks
Project 1: Reliable Transport

Assigned on: **Monday, 18 March 2019**
Due by: **Monday, 15 April 2019 at 12:00 (noon)**

1 Introduction

In this project, your task is to implement a reliable sliding window transport layer on top of the User Datagram Protocol (UDP). Please check the accompanying slides for details about the reliable sliding window transport layer.

In this lab, you are provided with a library (rlib.h and rlib.c), and you have to implement some functions and data structures for which skeletons are provided (in reliable.c). You will probably find it useful to look through rlib.h, as several useful helper functions have been provided.

Additionally, a buffer implementation is provided (buffer.h and buffer.c) which you can optionally use. If you decide not to use it, overwrite the content of reliable.c with the content of reliable_blank_skeleton.c. reliable.c is the only file you should have to edit to succesfully complete this assignment.

In general, your implementation should:

- Handle packet drops
- Handle packet corruption
- Provide trivial flow control
- Provide a stream abstraction
- Allow multiple packets to be outstanding at any time (using a limit given to your program as a run-time parameter, via the -w option)
- Handle packet reordering
- Detect any single-bit errors in packets

You will implement the client and server component of a transport layer. The client reads a stream of data (from STDIN), breaks it into fixed-sized packets suitable for UDP transport, prepends a control header to the data, and sends each packet to the server. The server reads these packets and writes the corresponding data, in order, to a reliable stream (STDOUT). Figure 1 presents a high-level overview of the system.

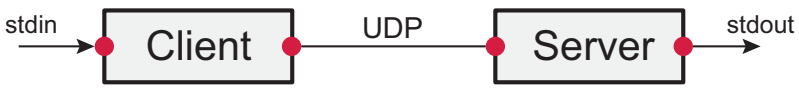

Figure 1: Overview of the reliable transport protocol

2 Requirements

Your transport layer must support the following:

- Each side's output should be identical to the other side's input, regardless of a lossy, congested, or corrupting network layer. You will ensure reliable transport by having the recipient acknowledge packets received from the sender; the sender will detect missing acknowledgements and resend the dropped or corrupted packets.
- You should handle connection tear down properly. When you read an EOF, you should send a zero-length payload (12-byte packet) to the other side to indicate the end of file condition. When you receive a zero-length payload and have written the contents of all previous packets (i.e., have written all output data with `conn_output`), you should send an EOF to your output by calling `conn_output` with a `len` of 0.
- You should support arbitrary window sizes. 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 that you have to implement.
- Your server and client should ensure that data is written in the correct order, even if the network layer reorders packets. Your receiver should buffer as many packets as the client may send concurrently. In other words, the sender window size (SWS) should equal the receiver window size (RWS).
- The sender should resend a packet if the receiver does not acknowledge it within an appropriate time period. You need not implement any back-off like TCP, and can instead merely 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.
- Acknowledgements should be cumulative rather than selective. Like TCP, 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 do not have to handle sequence number overflowing and wrapping in the lifetime of a connection.
- You can retry packets infinitely many times, and should make sure you retry at least five times, after which, if you want, the client 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.
- 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 become confusing if the two ouputs are mixed.

3 Implementation Details

3.1 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. Ack packets are 8 bytes, while Data packets vary from 12 to 512 bytes. The packet format is defined in rlib.h:

```
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 as well as 0 or more bytes of payload. The length, seqno, and ackno fields are always in network byte order (meaning you will have to use htonl/htons to write those fields and ntohl/ntohs to read them). Both Data and Ack packets contain the following fields:

- **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 should not 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. 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.

To limit the number of packets, a sender should not send more than one unacknowledged Data frame with less than the maximum number of bytes (500).

3.2 Implementations Details

In this project, you are provided with a library (rlib.h/rlib.c). The following details the six functions that you will be implementing for this project.

- **rel_create**: The `reliable_state` structure encapsulates the state of each connection. The structure is typedefed to `rel_t` in rlib.h, but the contents of the structure are defined in reliable.c, where you should add more fields as needed to keep your per-connection state. A `rel_t` is created by the `rel_create` function. The library will call `rel_create` directly for you.
- **rel_destroy**: A `rel_t` is deallocated by `rel_destroy()`. The library will call `rel_destroy` when it receives an ICMP port unreachable (signifying that the other end of the connection has died). You should also call `rel_destroy` when all of the following hold:
  - You have read an EOF from the other side (i.e., a Data packet of len 12, where the payload field is 0 bytes).
  - You have read an EOF or error from your input (`conn_input` returned -1).

  - All packets you have sent have been acknowledged.
  - You have written all output data with `conn_output`.

  Note that to be correct, at least one side should also wait around for twice the maximum segment lifetime in case the last ack it sent got lost, the way TCP uses the FIN_WAIT state, but this is not required.

- **rel_recvpkt**: When a packet is received, the library will call `rel_recvpkt` and supply you with the `rel_t`.
- **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 again available again, so that you can once again call `conn_input`. Do not loop calling `conn_input` if it returns 0; simply return and wait for the library to invoke `rel_read`! Do not accept more into your send buffer than the sliding window permits; call again `rel_read` when you were able to move the sending sliding window in `rel_recvpkt` in order to fill up the send buffer again.
- **rel_output**: To output data you have received in decoded UDP packets, call `conn_output`. `conn_output` 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 gave it. You must flow-control the sender by not acknowledging packets if there is no buffer space available for `conn_output`. You should schedule the calling of `rel_output`.
- **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 when.

4 Task

Your task is to implement the six functions (`rel_create`, `rel_destroy`, `rel_recvpkt`, `rel_read`, `rel_output`, `rel_timer`) described in Section 3.2.

Download and untar the project package from the course website. The six functions you need to implement are all in the file `reliable.c`. This is the only file you need to modify for the assignment. You should be able to run the command `make` to build the reliable program.

When you are done with the project, two instances of `reliable` should be able to talk to one another. An example of the working program is given here.

On one shell, run:

```
ethz:~/test/reliable> ./reliable 6666 -w 5 localhost:5555
[listening on UDP port 6666]
Hello. From port 6666 to port 5555
```

On another shell, run:

```
ethz:~/test/reliable> ./reliable 5555 -w 5 localhost:6666
[listening on UDP port 5555]
Hello. From port 6666 to port 5555
```

Now anything typed on one shell will show up on the other shell.

The value specified for the -w argument is stored in the `window` field of the `config_common` data structure. You should access it as `cc->window` in the `rel_create` function, and store the value somewhere in the `reliable_state` structure so you have access to it in other functions.

For debugging purposes, you may find it useful to run `./reliable` with -d command-line option. This option will print all the packets your implementation sends and receives.

5 Testing

There is also a tester program called `tester`, which is the same program we will use to assess your project. `tester` is provided as an x86_64 linux binary. Hence, for testing, please use any x86_64 linux-based operating systems. If you do not have access to such operating system, please use the virtual machine from the link below. The username is 'student' and the password is 'networketh6'. Link to the virtual machine: `https://ndal.ethz.ch/external/blank-ubuntu-networks.zip`

Run the `tester` giving it your `./reliable` program as an argument (e.g., ./tester -w 2 ./reliable). By default the `tester` program will run all tests, and set a window size of one. The following options may be useful to you:

- *-w N*: sets the window size to *N*. This also passes the -w option to your `reliable` program
- *-v*: shows the stderr output of your `reliable` program.
- *-T N*: runs test number *N* instead of running all of them. This is useful for debugging one particular test. There are 14 tests.
- *--gdb*: spawns a copy of your `reliable` program, prints a copy of your `reliable` program, and waits for you to press return. This is useful if you want to attach to the process in the gdb debugger (using the command "attach PID").

You can check whether your code indeed sends out the correct window size (e.g., 5) by executing:

```
python3 window_test.py 5 ./reliable ./reliable_no_answer
```

6 Assessment

We will assess your project using the provided `tester` on the virtual machine provided in the course website. **Make sure your code compiles and runs on the virtual machine supplied on the course website**.

We will run the `tester` with several different window sizes and different seeds (e.g., ./tester –seed 38935952 -w 2 ./reliable). The final normal score is averaged over all. The maximum normal score is 14. It is mandatory that your implementation implements the sliding window. We will test that your implementation does implement a sliding window, which is expressed in the window score. The maximum window score is 6. Together, the normal score and window score can sum up to 20.

7 Submission

To submit the project, you must do the following things:

- Run the command `make submit`, this should create a file called reliable.tar.gz
- Name the file as following: reliable_[netid].tar.gz
- You will be assigned a Git repository to submit your solution.

8 Acknowledgements

This Project has been adapted from Stanford's CS144 Introduction to Computer Networking Labs.

9 F.A.Q.

Can we assume that the UDP packet length received is correct?
No. As stated above, you must examine the length field, and should not assume that UDP packet you receive is the correct length. The network might truncate or pad packets.

How are `rel_output`, `conn_output`, and `conn_bufspace` related?

- `conn_output`: outputs data to STDOUT. Call this function with received data.
- `conn_bufspace`: returns the space available for use with conn_output. Calls to `conn_output` have limited space, as there is an underlying buffer. If you call `conn_output` with more data than it can handle, `conn_output` may return that it has accepted fewer bytes. In order to avoid passing in too much data, call `conn_bufspace` to find out how much space is available.
- `rel_output`: Once the data passed in to `conn_output` is all sent, the library will call `rel_output` in order to continue processing any more data available.

Should Acks be piggybacked on top of outgoing data packets?
Piggybacking Acks is preferable but not required.

Assume packets 1-5 are received, and I output packets 1-3, but I do not have space to output 4 and 5 (even though I have them buffered). What should I do?
Only Ack packet 6 once you have space for packets 4 and 5. This helps to rate limit the sender.

Does the `tester` restart a new version of the `./reliable` binary for each test?
No. We recycle a running instance of your program and put in new calls to `rel_create`. So if you don't handle tear down correctly, you can have one test affecting another test.

Do we retransmit each packet individually or always just retransmit all unacknowledged packets?
Either is fine for correctness, but the preferred action is to retransmit just one packet (rather than a whole window).

Do we have to handle multiple connections at the same time?
No, your code can work for only one sender and only one receiver, hence, connection demultiplexing is unnecessary.

How do we obtain the current time?

```
// Now in milliseconds
struct timeval now;
gettimeofday(&now, NULL);
long now_ms = now.tv_sec * 1000 + now.tv_usec / 1000;
```

What about byte ordering?
The length, seqno, and ackno fields of a packet are always in big-endian order (also called network byte order). Your machine might use little-endian. The functions `ntohl()`, `ntohs()`, `htonl()` and `htons()` might come in handy.

When running the test on my own machine I get an error that loading the shared library libgmp failed.
We encourage you to use the Virtual Machine image we provide in order to save time from such errors and because we will test your code on the same machine. To solve the problem you have to create a symlink: sudo ln -s /usr/lib/x86_64-linux-gnu/libgmp.so.10 /usr/lib/libgmp.so.3. Please ensure that in the end your code compiles on the Virtual Machine.