Computer Networking, Fall 2021

Assignment 2: A Simple Reliable Transport Protocol

2021.10.25

1 Introduction

In this assignment, you will build a simple reliable transport protocol, **RTP**, on top of UDP. Your RTP implementation must provide in order, reliable delivery of UDP datagrams in the presence of events like packet loss, delay, corruption, duplication, and reordering. There are a variety of ways to ensure a message is reliably delivered from a sender to a receiver. You are to implement a `sender` and a `receiver` that follows the following RTP specification.

2 RTP Specification

RTP sends data in the format of a header, followed by a chunk of data. RTP has four header types: $START$, END, $DATA$, and ACK, all following the same format:

```
typedef struct RTP_header {
    uint8_t  type;  // 0: START; 1: END; 2: DATA; 3: ACK
    uint16_t length;  // Length of data; 0 for ACK, START and END packets
    uint32_t seq_num;
    uint32_t checksum;  // 32-bit CRC
} rtp_header_t;
```

Establish connection. 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.

Data transmission. After establishing the connection, additional packets in the same connection are sent using the $DATA$ message type, adjusting `seq_num` appropriately. `sender` will use 0 as the initial sequence number for data packets in that connection.

Terminate connection. After everything has been transferred, the connection should be terminated with `sender` sending an END message, and waiting for the corresponding ACK for this message.

ACK for START & END. 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).

3 Assignment components

3.1 Part 1: Implement `sender`

`sender` should read an input message and transmit it to a specified receiver using UDP sockets following the RTP protocol. It needs to split the input message into appropriately sized chunks of data, and append a checksum to each packet. `seq_num` should increment by one for each additional packet in a connection. Please use the 32-bit CRC header we provide in *src/util.c*, in order to add a checksum to your packet.

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

After transferring the entire message, you should send an END packet to mark the end of connection.

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

- Loss of arbitrary levels
- Reordering of ACK messages
- Duplication of any amount for any packet
- Delay in the arrivals of ACKs
- Packet corruption

To handle cases where ACK packets are lost, you should implement a **500 milliseconds retransmission timer** to automatically retransmit packets that were never 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 packets in the window because they are all never acknowledged.

3.1.1 Running `sender`

`sender` should be invoked as follows:

```
./sender [Receiver IP] [Receiver Port] [Window Size] [Message]
```

- Receiver IP: The IP address of the host that `receiver` is running on.
- Receiver Port: The port number on which `receiver` is listening.
- Window Size: Maximum number of outstanding packets.
- Message: The message to be transferred. It can be a text string as well as a filename. `sender` will try to open the file with filename [Message] at first. If the file exists, it sends the content of this file to `receiver`; otherwise, it directly sends the [Message] content.

3.2 Part 2: Implement `receiver`

`receiver` needs to handle only one `sender` at a time and should ignore $START$ messages while in the middle of an existing connection. It must receive and store the message sent by the sender on disk completely and correctly.

`receiver` should also calculate the checksum value for the data in each packet it receives using the header mentioned in Part 1. 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 packet 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 `receiver` buffers out-of-order packets. The mechanism here is more efficient than GBN.

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

3.2.1 Running `receiver`.

`receiver` should be invoked as follows:

```
./receiver [Receiver Port] [Window Size] [File Name]
```

- Receiver Port: The port number on which `receiver` is listening for data.
- Window Size: Maximum number of outstanding packets.
- File Name: The name of the file used to store received messages.

3.3 Part 3: Optimizations

For this part of the assignment, you will be making a few modifications to the programs written in the previous two parts. Consider how the programs written in the previous parts would behave for the following case in Figure 1 where there is a window of size 3:

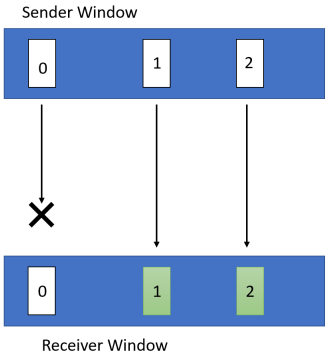

Figure 1: Inefficient transfer of data.

In this case `receiver` would send back two ACKs both with the sequence number set to 0 (as this is the next packet it is expecting). This will result in a timeout in `sender` and a retransmission of packets 0, 1 and 2. However, since `receiver` has already received and buffered packets 1 and 2. Thus, there is an unnecessary retransmission of these packets.

In order to account for situations like this, you will be modifying your `receiver` and `sender` accordingly:

- `receiver` will not send cumulative ACKs anymore; instead, it will send back an ACK with `seq_num` set to whatever it was in the data packet (i.e., if a sender sends a data packet with `seq_num` set to 2, `receiver` will also send back an ACK with `seq_num` set to 2). It should still drop all packets with `seq_num` greater than or equal to $N+$ `window_size`, where N is the next expected `seq_num`.

- `sender` must maintain information about all the ACKs it has received in its current window. In this way, packet 0 having a timeout would not necessarily result in a retransmission of packets 1 and 2.

For a more concrete example, Figure 2 and Figure 3 show how your improved sender (`opt_sender`) and receiver (`opt_receiver`) should behave for the case described at the beginning of this part.

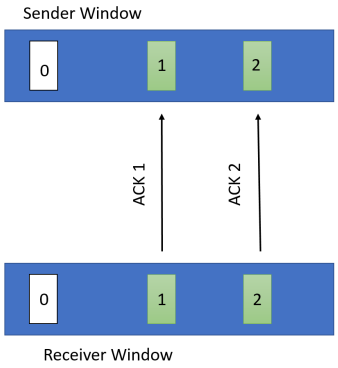

Figure 2: Only ACK necessary data.

`opt_receiver` individually ACKs both packet 1 and 2 in Figure 2.

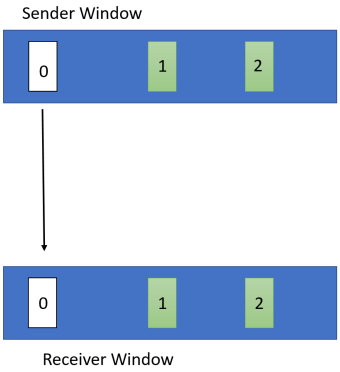

Figure 3: Only send unACKd data.

`opt_sender` receives these ACKs and denotes in its buffer that packets 1 and 2 have been received in Figure 3. Then, it waits for the 500 ms timeout and only retransmits packet 0 again.

You need to save your optimized program in *src/opt_sender.c* and *src/opt_receiver.c*). Note that you should update the CMakeList.txt file to add the targets of `opt_sender` and `opt_receiver`. The command line parameters passed to `opt_sender` and `opt_receiver` are the same as the previous two parts.

4 Important Notes

4.1 Development and test environment

This assignment needs to be developed and tested under the Linux system. **We strongly recommend that you use ubuntu 18.04 (i.e. the final testing environment of TA).**

4.2 Code template

We provide two code templates, which are placed in the *assignment2-rtp/templates/basic* and *assignment2-rtp/templates/recommended* directories. Both templates demonstrate the same basic function, i.e., how to encapsulate and transmit the RTP protocol atop UDP. You can complete this assignment on the basis of either template. The main difference between the two templates is that under the recommended template, you need to encapsulate and implement the RTP protocol in a POSIX-like API. We define the POSIX-like API of RTP in *rtp.h* and you need to provide your implementations in *rtp.c*. We detail the differences between the semantics of the RTP API and the standard POSIX API that you need to pay attention to in the source file.

Note that no matter which template you choose, you need to copy the files under your chosen template directory to the *src* directory.

4.3 Compilation

The code templates we provide are managed using the **cmake**. The recommended compilation approach is as follows. (Before you try to compile the template, you should copy the files in your chosen template to the *src* directory.)

```
cd assignment2-rtp
mkdir build
cd build
cmake ..
make
```

All the compiled executable files (including test cases) and intermediate files of cmake will be put in this build directory. Of course, you can compile a specified executable file at a time, e.g. run *make sender* to compile `sender`. You only need to re-run the cmake command to update the Makefile every time you modify the CMakeList.txt file.

5 Testing

We provide two executable files (`broken_sender` and `broken_receiver`) in *assignment2-rtp/test* directory. `broken_sender` and `broken_recevier` follows the specification of

RTP protocol in Part 1 & 2. Both files can simulate some different types of network failures during transmission. You can test your implementation using the two test cases. In order to simplify testing, you can run the two processes on the same host (using the local ip 127.0.0.1 for `receiver`). We detail the use of these two test cases and the types of failures that can be simulated in §5.1 and §5.2. You need to implement your own test case to test your code in Part 3.

Note that the test cases we provide only simulate the most simple network failure. The test cases used by TA for grading will be more complicated (multiple types of failures appear multiple times). You are encouraged to customize your test case to cover more failures. You can test your implementation against your classmates' customized `broken_sender` and `broken_receiver`. Of course, you should exchange the compiled executable files, NOT SOURCE CODES.

5.1 `broken_receiver`

To test your `sender`, you can run your `sender` with `broken_receiver` in two terminals respectively as in the following example.

Terminal 1:

```
cd assignment2-rtp/build
./broken_receiver 8076 10 ../data/recv.txt [Error Code]
```

Here, the first three arguments are the same as `receiver` in §3.2.1. The value of Error Code argument for `broken_receiver` can be as follows.

- If error code is 1, `broken_receiver` will drop one packet randomly during transmission (without ACK).
- If error code is 2, `broken_receiver` will exchange the order of two ACK packets.
- If error code is 3, `broken_receiver` will select one received packet and send its ACK twice.

Terminal 2:

```
cd assignment2-rtp/build
./sender 127.0.0.1 8076 10 ../data/test.txt
```

We provide a test text file (RFC-793 of TCP protocol) in the *assignment2-rtp/data* directory. You can use *diff* command to compare the raw file and the file saved by `broken_receiver`. If the function of `sender` is correct, the two files should be the same.

```
cd assignment2-rtp/data
diff test.txt recv.txt
```

5.2 `broken_sender`

To test your `receiver`, you should run `broken_sender` with your `receiver` together.

Terminal 1:

```
./receiver 8076 10 ../data/recv.txt
```

Terminal 2:

```
./broken_sender 127.0.0.1 8076 10 ../data/test.txt [Error Code]
```

Similarly, `broken_sender` will send the test file to your `receiver`. Here, the Error Code argument of `broken_sender` indicates different types of network failure, which can be as follows.

- If error code is 1, `broken_sender` will drop one packet randomly during transmission.
- If error code is 2, `broken_sender` will select one packet and send it twice.
- If error code is 3, `broken_sender` will send one packet with wrong checksum.

6 Grading

6.1 Part 1 & 2 (70%)

If your implementation (`sender` and `receiver` in Part 1 and 2) can pass one complicated test case that mixes all the failure types (described below), you can directly get all the scores of this part. Otherwise, your implementation will get corresponding scores based on the passed test cases.

For your `sender`:

- Normal case without network failures (10%)
- Loss of arbitrary amount and types of ACK messages (5%)
- Reordering of ACK messages (5%)
- Duplication of ACK messages (5%)
- Delay in the arrivals of ACK messages (5%)
- Connection failure (receive incorrect ACK for $START$ message) (5%)

For your `receiver`:

- Normal case without network failures (10%)

- Loss of arbitrary levels (5%)
- Reordering of $DATA$ messages (5%)
- Duplication of any amount for any packet (5%)
- Packet corruption (bad checksum) (5%)
- Wrong connection (receive incorrect $START$ message) (5%)

6.2 Part 3 (30%)

For Part 3, if your `opt_sender` and `opt_receiver` can pass two normal case without network failures respectively, you can get 10% for each one. If your implementations can pass the test cases with mixed failures above respectively, you can get 5% for each one.

7 Submission Guidelines

You need to submit your assignment by compressing the *assignment2-rtp* directory, which includes at least the following files:

- CMakeLists.txt
- src/sender.c
- src/opt_sender.c
- src/receiver.c
- src/opt_receiver.c
- src/util.h
- src/util.c
- src/rtp.h
- (optional) src/rtp.c

You may submit additional files that are needed. Your programs must be implemented in C. Please refer to the course website for the submission instructions.
Deadline: November 28 23:59:59

8 Recommended references

- Beej's Guide to Network Programming
- Linux C Programming
- The Missing Semester of Your CS Education

9 Acknowledgements

This programming assignment is based on JHU's Assignment 2 from EN.601.414/614: Computer Networks.