Assignment 3: Reliable Transport

Due: April 10th, 2022 at 11:59 PM

Special notes:

- This is a group based assignment and grouping info is the same as assignment 2
- You really want to read through the entire assignment document first. Latter parts might give you a hint on what you should do in former ones.

Table of contents

Overview

In this project, you will build a simple reliable transport protocol, WTP, **on top of UDP**. Your WTP implementation must provide inorder, 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 (`wSender`) and a receiver (`wReceiver`) that follows the following WTP specification.

WTP Specification

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

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

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

To initiate a connection, `wSender` starts with a `START` message along with a random `seqNum` 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 `seqNum` appropriately(see below). After everything has been transferred, the connection should be terminated with `wSender` sending an `END` message with the same `seqNumq` as the `START` message, and waiting for the corresponding ACK for this message.

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

`wSender` will use **0** as the initial sequence number for data packets in that connection. Furthermore, `wReceiver` sends back cumulative `ACK` packets (described in more details below).

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).

Learning Outcomes

After completing this programming assignment, students should be able to:

- Explain the mechanisms required to reliably transfer data
- Describe how different sliding window protocols work

Clarifications

- Your program will only be tested with one wSender and one wReceiver for all parts.

Part 1: Implement `wSender`

`wSender` should read an input file and transmit it to a specified receiver using UDP sockets following the WTP protocol. It should split the input file into appropriately sized chunks of data, and append a `checksum` to each packet. `seqNum` should increment by one for each additional packet in a connection. Please use the 32-bit CRC header we provide in the `starter_files` directory, 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. `wSender` must accept cumulative `ACK` packets from `wReceiver` .

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

`wSender` 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.

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 500ms the window still has not advanced, you retransmit all packets in the window because they are all never acknowledged.

Running `wSender`

`wSender` should be invoked as follows:

```
./wSender <receiver-IP> <receiver-port> <window-size> <input-file> <log>
```

- `receiver-IP` The IP address of the host that `wReceiver` is running on.
- `receiver-port` The port number on which `wReceiver` is listening.
- `window-size` Maximum number of outstanding packets.
- `input-file` Path to the file that has to be transferred. It can be a text as well as a binary file (e.g., image or video).

- `log` The file path to which you should log the messages as described below.

Example: `./wSender 10.0.0.1 8888 10 input.in log.txt`

Note: for simplicity, arguments will appear exactly as shown above during testing and grading. Error handling with the arguments is not explicitly tested but is highly recommended. At least printing the correct usage if something went wrong is worthwhile.

Logging

`wSender` should create a log of its activity. After sending or receiving each packet, it should append the following line to the log (i.e., everything except the `data` of the `packet` structure described earlier):

```
<type> <seqNum> <length> <checksum>
```

Part 2: Implement `wReceiver`

`wReceiver` needs to handle only one `wSender` at a time and should ignore `START` messages while in the middle of an existing connection. It must receive and store the file sent by the sender on disk completely and correctly; i.e., if it received a video file, we should be able to play it! The stored file should be named `FILE-i.out`, where `i=0` for the file from the first connection, `i=1` for the second, and so on.

`wReceiver` 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 `seqNum` 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 `seqNum` not equal to `N`, it will send back an `ACK` with `seqNum=N`.
2. If it receives a packet with `seqNum=N`, it will check for the highest sequence number (say `M`) of the inorder packets it has already received and send `ACK` with `seqNum=M+1`.

If the next expected `seqNum` is `N`, `wReceiver` will drop all packets with `seqNum` greater than or equal to `N + window-size` to maintain a `window-size` window.

`wReceiver` should also log every single packet it sends and receives using the same format as the `wSender` log.

Put the programs written in parts 1 and 2 of this assignment into a folder called `WTP-base`.

Running `wReceiver`

`wReceiver` should be invoked as follows:

```
./wReceiver <port-num> <window-size> <output-dir> <log>
```

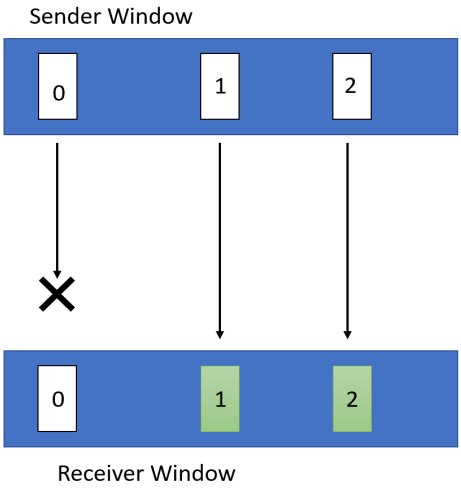
- `port-num` The port number on which `wReceiver` is listening for data.
- `window-size` Maximum number of outstanding packets.
- `output-dir` The directory that the `wReceiver` will store the output files, i.e the `FILE-i.out` files.
- `log` The file path to which you should log the messages as described above.

Example: `./wReceiver 8888 2 /tmp log.txt`

Note: for simplicity, arguments will appear exactly as shown above during testing and grading. Error handling with the arguments is not explicitly tested but is highly recommended. At least printing the correct usage if something went wrong is worthwhile.

Part 3: Optimizations (Bonus part)

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

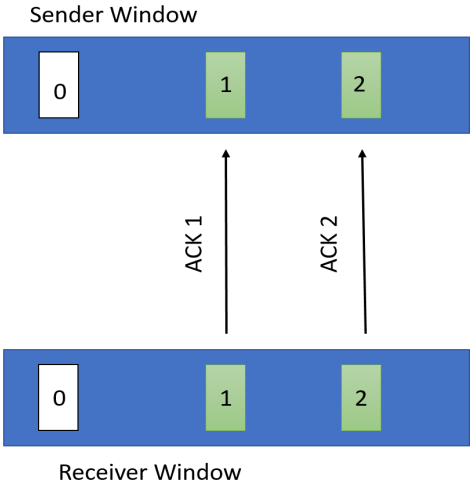


In this case `wReceiver` would send back two ACKs both with the sequence number set to 0 (as this is the next packet it is expecting). This will result in a timeout in `wSender` and a retransmission of packets 0, 1 and 2. However, since `wReceiver` has already received and buffered packets 1 and 2. Thus,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

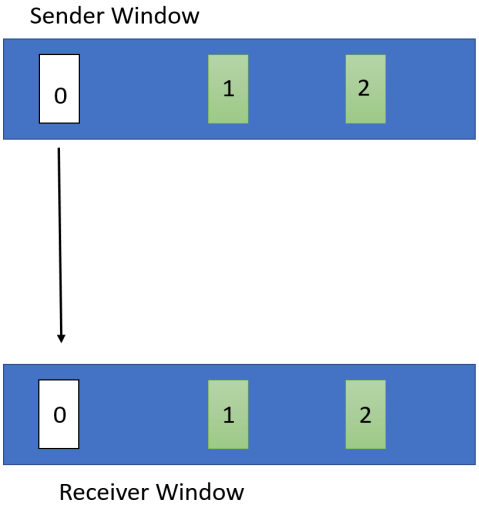
In order to account for situations like this, you will be modifying your `wReceiver` and `wSender` accordingly (save these different versions of the program in a folder called `WTP-opt`):

- `wReceiver` will not send cumulative ACKs anymore; instead, it will send back an ACK with `seqNum` set to whatever it was in the data packet (i.e., if a sender sends a data packet with `seqNum` set to 2, `wReceiver` will also send back an ACK with `seqNum` set to 2). It should still drop all packets with `seqNum` greater than or equal to `N + window_size`, where `N` is the next expected `seqNum`.
- `wSender` must maintain information about all the ACKs it has received in its current window and maintain an individual timer for each packet. So, for example, packet 0 having a timeout would not necessarily result in a retransmission of packets 1 and 2.

For a more concrete example, here is how your improved `wSender` and `wReceiver` should behave for the case described at the beginning of this section:



`wReceiver` individually ACKs both packet 1 and 2.



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

The command line parameters passed to these new `wSender` and `wReceiver` are the same as the previous two sections.

Important Notes

- It is up to you how you choose to read from and write to files, but you may find the `std::ifstream.read()` and `std::ofstream.write()` or `mmap` functions particularly helpful.
- Please closely follow updates on Piazza. All further clarifications will be posted on Piazza via pinned Instructor Notes. We recommend you follow these notes to receive updates in time.
- You MUST NOT use TCP sockets (autograder knows if you do).
- Another good resource for UDP socket programming is Beej's Guide to Network Programming Using Internet Sockets.

Tips

- Man page is your friend, when you are not sure about how to send/receive packets or libc function definition and usages in general, e.g. `man socket`.
- If you only use a single thread, polling on a socket might be helpful for receiving packets and timer implementation, see `MSG_DONTWAIT` in `recvfrom` or `recvmsg`.

Submission Instructions

Submission is done via the autograder. Submission policy will be announced when the autograder is released, which we anticipate being around halfway through the assignment.

To submit:

1. Pack your code into a tarball, e.g. `tar acvf p3.tar.gz p3`. You can decompress it via `tar axvf p3.tar.gz`
2. Go to autograder. You can specify the tarball you want us to grade.
3. Press submit. Your results will show up on that page once grading is finished.

Your assigned repository must contain:

- `Makefile`(s) to compile both executables with one single `make` command
- The source code for `wSender` and `wReceiver` from parts 1 and 2: all source files should be in a folder called `WTP-base`.
    - Subdirectories for source code within `WTP-base` are perfectly fine, as long as the executables are present after running `make` in `WTP-base`

- The source code for `wSender` and `wReceiver` from part 3: all source files should be in a folder called `WTP-opt`. Subdirectories are fine here too.

Example final structure of repository:

```
$ tree -I "<ignore-pattern>" ./p3/
./p3/
├── README.md
├── WTP-base
│   ├── Makefile <- supports "make clean" and "make"
│   ├── wReceiver <- Binary executable present after running "make"
│   ├── ** source c or cpp files **
│   └── wSender <- Binary executable present after running "make"
├── WTP-opt
│   ├── Makefile <- supports "make clean" and "make"
│   ├── wReceiver <- Binary executable present after running "make"
│   ├── ** source c or cpp files **
│   └── wSender <- Binary executable present after running "make"
└── starter_files
    ├── PacketHeader.h
    └── crc32.h
```

Autograder

The autograder will be released roughly halfway through the assignment. You are encouraged to design tests by yourselves to fully test the functionality of sending and receiving complete and unaltered files. You should *NEVER* rely on the autograder to debug your code, instead you might want to try gdb e.g. `gdb --args ./wReceiver 8888 2 /tmp log.txt`. Clarifications on the autograder will be added in this section:

Our autograder runs the following versions of gcc/g++, please make sure your code is compatible.

```
$ gcc --version
gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ g++ --version
g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Acknowledgements

This programming assignment is based on University of Michigan's Assignment 3 from EECS 489: Computer Networks and UC Berkeley's Project 2 from EE 122: Introduction to Communication Networks.