Reliable transmission in the transport layer

Venkat Arun

1 Introduction

In this assignment, you will create a custom transport layer program that provides reliable data delivery to higher layers. It consists of a sender and a receiver connected over a network link. The sender will take a large arbitrarily sized message and send it to the receiver after breaking it down into packets that are at most 1500 bytes long. On the way, packets may get delayed and lost, or arrive at the receiver in a different order from which they were sent. The receiver reconstructs the message from these packets. In addition, for each packet received, it sends a packet acknowledging the receipt of all the data packets that have been received so far. The sender uses these acknowledgments to guess which packets have been lost and retransmits them.

In addition, it is risky to send a lot of packets at once. First, the network may not be able to handle the load and drop/delay the packets. Second, the receiver may not have enough memory to store all the incoming data and might have to discard some of it. This is solved by having a "receive window" that limits the maximum number of bytes that can be "in flight" at any given time. A byte is in-flight if it has been sent, but not yet acknowledged or assumed to be lost. That is, number of packets in-flight = number of packets sent - number of packets acknowledged - number of packets assumed to be lost. In this assignment, we will treat the receive window as a constant. In a real transport stack, it varies with time as follows.

1. The receiver allocates a block of memory in which to store the received data. This is called the receive buffer. Data is added to it when packets arrive and removed when it delivers bytes to the application. Once data has been given to the application, it is the application's responsibility to maintain it if necessary. The transport layer is free to delete those bytes. In a real transport protocol like TCP, the receiver continually updates the amount of free memory in its receive buffer through a dedicated field in the acknowledgment packet. This process is described in the "Flow control" paragraph of section 5.2.4 in the book.

2. To ensure that we do not overload the network and create excessive packet loss and delay, a second algorithm continually adjusts the receive window, except in this case it is called the "congestion window". This algorithm is called a "congestion control algorithm (CCA)" and typically runs on the

sender. CCAs *guess* the available network bandwidth, accounting for not only the network capacity, but also any other flows that may be sharing the same capacity. This is a hard problem because, in order to do so, it must contend with the large diversity of links on modern networks and the fact that it can never explicitly communicate with the other flows sharing the bandwidth. Since it is also a critical part of the infrastructure of the internet, CCAs have been the subject of intense research for forty years. Your instructor has, perhaps foolishly, spent a large fraction of his career trying to solve, or at least understand, the problem. The basics of congestion control are discussed in chapter 6 of the book.

2 Implementation Strategy

In this assignment, we will use the User Datagram Protocol (UDP) to transmit packets. UDP is a lightweight transport protocol that augments the IP layer with send and receive ports (plus a checksum and length which will not concern us in this assignment). These ports tell the operating system to which program it should deliver the packets. The skeleton code to set up UDP sockets is already provided in the github repository: `https://github.com/venkatarun95/networks-assignment-transport`. It is possible to directly create IP packets, but it is painful and requires special permissions from the OS and a mechanism, like the port numbers, to decide which application to which we should deliver the data.

TCP assigns each byte a sequence number represented as a 32-bit integer. This allows for ~ 4 billion separate sequence numbers which allows it to assign a unique number to ~ 4 giga bytes of data. Thus it has mechanisms for what to do when it wraps around (plus some security reasons). While practically important, this mechanism is complicated and not very intellectually interesting. Thus, in this assignment, we will ignore this issue and assume that sequence numbers can be arbitrarily large and nobody is trying to attack us. For us, the first byte will start with a sequence number of zero and the rest increment from there.

TCP packets have a standardized header format where the role of each bit in the first 40-80 bytes that constitute the header is well defined. You should take a look at what it looks like. However, in this assignment, we will serialize information using json to keep debugging and serializing/deserializing simple.

Data packets contain the following fields: Sequence number of the first byte, sequence number of the last byte + 1, the data

Acknowledgement packets contain 1) the sequence numbers that were received just now and 2) a single list of *ranges* of sequence numbers that have been received. They tell the sender which bytes have been received. We use ranges because, in the common case, bytes are delivered in sequence and thus can be represented with a very small number of ranges. If there is no loss or packet reordering, a single range suffices. We illustrate this with an example:

- If packets containing data with sequence numbers 0-100, 100-200, 500-600, and 300-400, 700-800 have been received, the receiver will send the list `[0-200, 300-400, 500-600, 700-800]`

- Now, if packets with sequence numbers 200-300 and 400-500 are received, the receiver sends a list with just a single range: `[0-600, 700-800]`

Tangent: The TCP header only includes space for up to three ranges. These are called SACKs (Selective ACKnowledgments) This is because bits were scarce when the internet was first designed.[1] Things have changed in the 21st century, and a new transport protocol is becoming popular: QUIC. Like our toy transport layer, QUIC also operates on top of UDP and allows for arbitrarily many ranges.[2] It is rare for protocol-level changes to occur on the internet, since many different programs and hardware have to support it in order for it to work. QUIC is an exception because it was developed and promoted by Google, who controlled both ends of the transport layer: the browser (through Google Chrome) and the servers of their web services. After Google's initial push, it has now been standardized by the IETF and adopted by many people.

Unlike TCP, we do not implement a handshake since we assume the receive window is constant and pre-allocated and that the sequence numbers start from 0. To terminate the connection, the sender sends a special packet. The starter code handles this

Your assignment is to fill in the places in the starter code marked as `TODO` and implement the logic that decides what sequence numbers to transmit next and what ranges to acknowledge. This logic is surprisingly tricky to get right because there are a few corner cases to consider. So be careful! Note: as in the previous assignment, it is ok to use inefficient datastructures and algorithms if it makes coding simpler (within reason).

Bonus: In the main component of the assignment, you may assume that packets are dropped, but never reordered. You will get 20% bonus points if your code can also handle packet reordering.

In addition to what you write, you should understand the starter code as well. It has some ideas that might come in handy for you in the future.

3 Running the code

In the end, you will have a program that can transmit bytes over an unreliable network. To run it, you should first generate a file to send. You can pick any pure-text file you like, or you can generate one of the desired size using the command:

```
python3 generate_bogus_text.py 1000000 >test_file.txt
```

[1] This is also why we decided to only include 32 bits for the IP addresses because, obviously, the internet will never have more than 4 billion devices right? To this date, the new version of IP, called IPv6, has not been fully deployed. It was standardized very recently... in 1998

[2] QUIC also assigns unique numbers to each packet it sends. This makes life easier. For instance, we can determine exactly which packet prompted the ack and accurately measure round trip times. We adopt this approach.

The input specifies the length of the file to generate. We recommend using at least 1 million. The next step is to start the receiver so that it can listen for incoming connections using the following command:

```
python3 transport.py --ip localhost --port 7000 receiver
```

Note: you can use any IP address and port you like. The above choices are useful for testing on your local machine. Now, you can run the server to transmit the test file you just generated (here, `test_file.txt`) as follows:

```
python3 transport.py --ip localhost --port 7000 --sendfile test_file.txt sender
```

We are providing two ways to test your code: 1) a built-in emulator that randomly drops/reorders packets before sending them and 2) the Mahimahi network emulator. For this assignment, option 1 suffices. Mahimahi will be necessary for the next one.

To use the built-in emulator, just add the option `--simloss 0.1` to simulate a 10% packet loss. You should test your code with a range of losses including for example, 0, 0.01, 0.1, 0.9, 0.99. A good transport protocol should offer reliability no matter what.

Mahimahi is a network emulator that emulates many different types of network links without needing access to multiple computers. Mahimahi only works on Linux, which means that you will have to either use a local Linux installation or a VM. Alternately, you can use a machine from cloud lab as you learned in the first assignment. You can learn how to use it here: mahimahi.mit.edu. For this assignment, you only want a shell that can create packet losses. Thus, you should run:

```
mm-loss uplink 0.1
```

This will create a *new* shell environment. The prompt on your shell will change to indicate this. Any further commands you run will run *inside* the shell. This way, any packets leaving the shell will get dropped. You should run the receiver in a separate terminal *outside* the shell and the sender *inside* it. To make the receiver listen to all packets at all interfaces, use the IP address `0.0.0.0`. To the sender, give the IP address stored in the environment variable `MAHIMAHI_BASE_IP`[3].

Gradescope Uploads: after you write your codes in transport.py, upload that file into the Gradescope. Your code should have complete Receiver and Sender classes.

[3] To figure out what the value is, you can run `$echo MAHIMAHI_BASE_IP`. Usually, it is `100.64.0.1`