Machine Problem 3

Reliable & Congestion Control

Due: Friday, Apr 27, 11:59pm

Please read all sections of this document before you begin coding.

In this MP, you will implement a transport protocol with properties similar to TCP.

You have been provided with a file called **`sender_main.c`**, which declares the function

`void reliablyTransfer(char* hostname, unsigned short int hostUDPport, char* filename, unsigned long long int bytesToTransfer)`

This function should transfer the first **`bytesToTransfer`** bytes of **`filename`** to the receiver at **`hostname:hostUDPport`** correctly and efficiently, even if the network drops or reorders some of your packets.

You also have **`receiver_main.c`**, which declares the function

`void reliablyReceive(unsigned short int myUDPport, char* destinationFile)`

This function is **`reliablyTransfer`**'s counterpart and should write what it receives to a file called **`destinationFile`**.

Your job is to implement these two functions, with the following requirements:

- The data written to disk by the receiver must be exactly what the sender was given.
- Two instances of your protocol competing with each other must converge to roughly fair sharing the link (same throughputs +/- 10%) within 100 RTTs. The two instances might not be started at the exact same time.
- Your protocol must be somewhat TCP friendly: an instance of TCP competing with you must get on average at least half as much throughput as your flow.

- An instance of your protocol competing with TCP must get on average at least half as much throughput as the TCP flow.
- All of the above should hold in the presence of any amount of dropped or reordered packets. All flows, including the TCP flows, will see the same rate of drops/reorderings. The network will not introduce bit errors.
- Your protocol must get reasonably good performance when there is no competing traffic and no packets are intentionally dropped or reordered. Aim for at least 33 Mbps on a 100 Mbps link.
- You cannot use TCP in any way. Use **`SOCK_DGRAM`** **(UDP)**, not **`SOCK_STREAM`** **(TCP)**. The test environment has a ≤ 100 Mbps connection, and an unspecified RTT – meaning you'll need to estimate the RTT for timeout purposes, like real TCP.

Appendix – NOTES: Same as the notes of the previous assignments, plus:

- The MTU on the test network is 1500, so up to 1472 bytes payload (IPv4 header is 20 bytes, UDP is 8 bytes) won't get fragmented. You can **`sendto()`** larger packets and the sockets library's UDP will handle fragmentation/reassembly for you. It's up to you to reason out the benefits and drawbacks of using large UDP packets in various settings.
- You can use the provided main files to be sure that your program has the right interface. However, feel free to write your own main files.
- **Executable names**: **`reliable_sender`** and **`reliable_receiver`**. A single run of "make" (with no arguments) inside your MP3 directly should build both binaries.
- Be sure that you have a clean design for implementing the send/receive buffers. Trying to figure out where to get the data to resend an old packet won't be fun if your send window's buffer doesn't have a nice clean interface.
- Input files on the autograder are **READ-ONLY**. In your program, please open file with read mode only.
- Input files on the autograder are general binary data, NOT text.

- We will test your program against the following scenarios:
    - Send a single byte in a few seconds on a normal link with no competition.
    - Send 50KB in 6 seconds on a normal link with no competition.
    - Send a big file.
    - Handle 5% loss with slightly-better-than-rock-bottom throughput.
    - Handle 25% loss – but throughput only needs to be 0.1MB/s on a 100Mbps link.
    - Achieve a certain healthy fraction of ideal throughput on a normal link with no competition.
    - Two instances of your protocol can share a link roughly evenly (without taking too long to converge).
    - TCP friendliness.

Appendix – Testing Environment

You can use the same basic test environment described in MP1. However, the network performance will be unrealistically good (same goes for testing on localhost), so you need to limit its performance. The autograder uses **tc**.

If your network interface inside the VM is eth0, run (from inside the VM) the following commands:

1. `sudo tc qdisc del dev eth0 root 2>/dev/null` [this delete any existing tc rules]
2. `sudo tc qdisc add dev eth0 root handle 1:0 netem delay 20ms loss 5%`
3. `sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 100Mbit burst 40mb latency 25ms`

will give you a 100 Mbit, ~20 ms RTT link where every packet sent has a 5% chance to get dropped. Simply omit the loss n% part to get a channel without artificial drops. You can run these commands on just the sender; running them on the receiver as well won't make much of a difference, although you'll get a ~40 ms RTT if you don't adjust the delay to account for the fact that it gets applied twice.