Machine Problem 3

Reliable Transmission & Congestion Control

Due: Thursday, April 30th, 11:59pm

In this assignment, you will use UDP to implement a transport protocol with properties similar to TCP. You will implement two commands: a sender command (named `reliable_sender`) and a receiver command (named `reliable_receiver`). Your implementation should transfer a given file (with binary data, not text) from the sender to the receiver correctly, efficiently and reliably even when the network drops and reorders some of your UDP packets.

1. Environment Setup

1.1 Your Group GitLab Repository

You will use `git` to maintain your code submission. In this machine problem, you will use the same repository that you used for MP2:

`https://gitlab.engr.illinois.edu/cs438-sp2020/mp2-mp3/group<group_num>`

Use your UIUC login credentials to access this repository. Both you and your teammate (if a group of two) have access to this repository. We will grade your programs that are pushed to this repository. Feel free to use this repository to maintain your code during development. **For this machine problem, you must maintain your code in the *mp3* folder.**

1.2 The Auto-grader Ubuntu Machine

Here, we describe the system environment where the auto-grader runs to compile and test your code.

Like in previous MPs, you must make sure that your submitted code can be compiled (by `make`) and run on an Ubuntu 18.04.4 LTS Desktop machine. The auto-grader runs such an OS version in a virtual machine. Use `apt install` command to install necessary development tools such as `make`, `gcc`, `g++`, `clang` to ensure that you use the latest compiler versions (that the auto-grader uses). You are recommended to set up a virtual machine with the same OS version.

In this MP, two Ubuntu VMs are used to test the file transmission. One of the VMs is used as the file sender and the other as the file receiver – let's call them the Sender VM and the Receiver VM, respectively. These VMs are running in the same host machine and their networks are connected together (if you use VirtualBox, you can use "`Host-only Adapter`" to bridge two VMs together). The MTU on the test network is 1500 (you can check the value with the command `ifconfig`).

2. Test Environment

When testing the file transmission in the VMs introduced above, the network performance will be unrealistically good (and the same goes for testing on the localhost), so the test environment has to be restrained by using the `tc` command (you only need to run this command in the Sender VM). An example is given as follows:

Assuming the Sender VM's network interface that connects to the Receiver VM is `eth0`, run the following commands in the Sender VM:

```
sudo tc qdisc del dev eth0 root 2>/dev/null
sudo tc qdisc add dev eth0 root handle 1:0 netem delay 20ms loss 5%
sudo tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 100Mbit burst
     40mb latency 25ms
```

The first command will delete any existing tc rules.
The second and the third commands will give you a 100 Mbit, ~20 ms RTT link where every packet sent has a 5% chance to get dropped (remove "`loss 5%`" part if you want to test your programs without artificial drops.) Note that it's also possible to specify the chance of reordering by adding "`reorder 25%`" after the packet loss rate argument.

3. The Assignment – The Sender and Receiver Commands

3.1 The Basics

Your code should compile to two executables named `reliable_sender` and `reliable_receiver` that support the following usage:

```
./reliable_sender <rcv_hostname> <rcv_port> <file_path_to_read> <bytes_to_send>
./reliable_receiver <rcv_port> <file_path_to_write>
```

The `<rcv_hostname>` argument is the IP address where the file should be transferred.
The `<rcv_port>` argument is the UDP port that the receiver listens.
The `<file_path_to_read>` argument is the path for the **binary file** to be transferred.
The `<bytes_to_send>` argument indicates the number of bytes from the specified file to be sent to the receiver.
The `<file_path_to_write>` argument is the file path to store the received data.

For example:

First, on the Receiver VM, run:

```
./reliable_receiver 2020 /path/to/rcv_file
```

Then, on the Sender VM, run:

```
./reliable_sender 192.168.4.38 2020 /path/to/readonly_file.dat 1000
```

This allows the sender command to connect to the receiver hosted at `192.168.4.38` with an UDP port `2020` and transfer the first 1000 bytes in the file named `readonly_file.dat` to the receiver. Upon receiving the data (1000 bytes in this case), the receiver stores it in the file named `rcv_file.`

3.2 Transmission Function Templates

You are free to write your own functions to implement the commands required in this MP. You can also start with the files we provide: a file named `sender_main.c` and a file named `receiver_main.c` which declare the sender's and receiver's functions. They are available at:

https://courses.engr.illinois.edu/cs438/sp2020/mp/mp3-supplement.zip

(1) The Sender's 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 and reorders some of your packets.

(2) The Receiver's Function:

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

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

3.3 Implementation Requirements

Your job is to implement the sender and receiver's functions/commands, 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. (Hint: The auto-grader uses the command `iperf` to create the competing TCP 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 $\leq$ 100 Mbps connection and an unspecified RTT -- meaning you'll need to estimate the RTT for timeout purposes, like real TCP.
- The sender and receiver's commands/programs must end once the file transmission is done.

3.4 Extra Hints

- 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 use `sendto()` to send larger packets and the UDP socket library will handle the fragmentation/reassembly for you. It's up to you to reason out the benefits and drawbacks of using large UDP packets in various settings.
- Be sure that you have a clean design for implementing the send/receive buffers. Trying to figure out which part of the data to resend won't be fun if your sender's window buffer doesn't have a nice clean interface.
- Input files on the auto-grader are READ-ONLY. In your program, please open the files with read mode only. These files contain binary data, NOT text.

4. What to Submit

You must submit (i.e., commit and push) your code to the private GitLab repository provided to you in Section 1.1. We will only grade your programs that are pushed to the `master` branch in that repository. In your repository, it must (at least) contain the following files:

- `/mp3/readme.txt` # writes down who did what in your group
- `/mp3/Makefile` # the makefile
- `/mp3/<your_source_files>`

If you work as a group of two, please write down each person's responsibility in this MP.

You must create the makefile so that execution of the command

```
make
```

compiles and generates the executable files named `reliable_sender` and `reliable_receiver`, in the same *`mp3`* folder and they should be able to run with arguments as specified in Section 3.1.

Finally, running

```
make clean
```

should delete all executable files and any temporary files that the Makefile or your program creates.

You may commit and push your code as many times as you like before the deadline. It's in fact a good practice to commit your code whenever there is a change that is worth noted. We will use the last commit you made for the repository when we grade your MP.

Your program cannot be graded if it has not been pushed to the repository. Fail to commit and push your code on time will result in your MP being considered late. It is your responsibility to ensure all your work is committed and pushed to your repository by the due date.

5. Grading

The grading will be determined by 10 test cases:

1. 10%: Send a single byte in 3 seconds on a 100Mb, normal link with no competition.
2. 10%: Send 50KB (50,000) in 6 seconds on a 100Mb, normal link with no competition.
3. 10%: Send 65MB (65,000,000) in 20 seconds on a 100Mb, normal link with no competition.
4. 10%: Send 1MB (1,000,000) in 10 seconds on a 100Mb, 5% loss link with no competition.
5. 10%: Send 1MB (1,000,000) in 10 seconds on a 100Mb, 25% loss link with no competition.
6. 10%: Quick convergence: two instances of your protocol should share a link roughly evenly (without taking too long to converge), on a 40Mb normal link.
7. 10%: TCP friendliness: a TCP instance and an instance of your protocol should share a link roughly evenly, on a 40Mb normal link.
8. 10%: TCP friendliness: a TCP instance and an instance of your protocol should share a link roughly evenly, on a 100Mb, 1% loss link.
9. 10%: Send 65MB (65,000,000) in 40 seconds on a 40Mb normal link, competing with a TCP instance.
10. 10%: Send 10KB (10,000) in 10 seconds on a 100Mb, 25% loss, large chance of reordering link with no competition.

Because routing is nondeterministic, your final grade on this MP will be calculated by running the AG 3 times after the deadline and averaging the top 2 scores.

6. Important Notes

1) You must use C or C++.
2) Your code must be runnable in 64-bit Ubuntu 18.04.4 LTS Desktop machines. This is the environment in which the auto-grader runs. You program must fulfill the given assignment requirements to get said scores.
3) If you need to use a library for data structures, you MUST get the approval of the course staff. Additionally, you MUST acknowledge the source in a README in *mp3* folder. However, algorithms MUST be your own.
4) The auto-grader will test every group's current version every night. The TAs will try their best to make sure this is the case, but bugs happen. We will make an announcement of its starting date.
   - Do a git pull the next morning to see your score in *mp3/result.txt*.
   - The score you are given before the deadline is for your reference and has no effect on your final grade.
   - The late penalty is applied to the last git commit you make for anything in the folder *mp3*. If you want to make any change in *mp3* that does not change your code, please wait until the third day after the deadline to make it.
5) Do not make your private repository public. You will be held partially responsible for any resultant plagiarism.
6) You must work alone or with your teammate (for MP3). Your code must be your own in your group. You can discuss very general concepts with other groups, but if you find yourself looking at a screen/whiteboard full of pseudo code (let alone real code), you are going too far.
   - Refer to the class slides and official student handbook for academic integrity policy. In summary, the standard for guilt is "more probable than nor probable", and penalties range from warning to recommending suspension/expulsion, based entirely on the instructor's impression of the situation.
   - The Grainger College of Engineering has some guidelines for penalties that we think are reasonable, but we reserve the right to ignore them when appropriate.

7. (Optional) Competition

We are going to roll out a performance contest for MP3. This does not affect your grade, but the professor will give out prizes at the end of the semester! We will be using test cases **3, 4, and 5** for the contest.

The webpage linked below contains the rankings, and it uses the team name specified in `teamname.txt` under your mp3 folder to represent your group (you create it). You are free to change the team name anytime (in fact, you may want to make a unique team name to differentiate your results from others.) The rankings are updated after each AG run.

`https://courses.engr.illinois.edu/cs438/sp2020/mp/competition/`

The AG will start on April 18th (Saturday) and runs once every night until the deadline. The AG will pull your code at exactly 6:00pm (of course, it will be delayed if someone's code breaks the AG). The AG test results are usually pushed back to you in a few hours if everything runs as expected. We will make adjustment for the AG schedule if needed.

If you want to change your group (e.g., break up with your partner, form a new group), please make sure you follow the steps below:

(1) Both you and your partner have to agree on the group change.
(2) Have one of you in this group post a private post on Piazza. State your request (e.g., splitting the group, forming a new group) and include both netIDs.
(3) We'll get back to you as soon as possible. If approved, new group ID(s) and new repository(s) will be assigned.

Good luck and have fun!