CS/ECE 438: Computer Networks **Spring 2026**

Machine Problem 3

Handed Out: April 2, 2026 *Due: Apr 26, 2026 11:59pm CST*

TA: Nishant Sheikh

Updated April 15, 2026: `reliable_receiver` cmdline args: removed `num_bytes`

1 Introduction

This machine problem tests your understanding of reliable packet transfer. You will use UDP to create your own implementations of reliable transport: one similar to TCP, and one open-ended. Your implementations must be able to tolerate packet drops, reordering, allow other concurrent connections a fair chance, and must not be overly nice to other connections (i.e. should not give up the entire bandwidth to other connections).

1.1 Functions

In `sender_main.c`, the following functions are declared:

```
void reliably_transfer_TL(char *hostname, unsigned short int port, char *filename, unsigned long long int num_bytes)
```

```
void reliably_transfer_OE(char *hostname, unsigned short int port, char *filename, unsigned long long int num_bytes)
```

These functions should transfer the first `num_bytes` bytes of the file at `filename` to the receiver at `hostname:port` correctly and efficiently, even if the network drops or reorders packets.

In `receiver_main.c`, the following functions are declared:

```
void reliably_receive_TL(unsigned short int port, char *filename)
```

```
void reliably_receive_OE(unsigned short int port, char *filename)
```

These are counterparts to the functions in `sender_main.c`. They should write their output to the file at `filename`.

1.2 Interface

Your executable names must be `reliable_sender` and `reliable_receiver`. They must be called as follows:

```
reliable_sender variant hostname port filename num_bytes
```

```
reliable_receiver variant port filename ~~num_bytes~~
```

The `variant` argument is a string — either `"TL"` or `"OE"` — which indicates the variant of `reliably_transfer` and `reliably_receive` to run.

2 What is expected in this MP?

Your job is to implement the `reliably_transfer_{TL, OE}` and `reliably_receive_{TL, OE}` functions.

1. The TCP-like variant, in `reliably_transfer_TL()`, must approximately match the behavior of TCP.
2. For the open-ended variant, in `reliably_transfer_OE()`, get creative! You may choose to implement additional features on top of your TCP-like variant (such as Go-Back-N, Selective ACK, or Fast Retransmit), or you can try something completely different.
3. For both variants, you must write a description of the algorithm and your thought process in `mp3/README.md`. After the deadline, we may check your descriptions to make sure they match your implementation.

   IMPORTANT: You may not use AI to write these descriptions.

Please make sure you read the requirements for each configuration carefully.

2.1 TCP-like variant

Your job is to implement `reliably_transfer_TL()` and `reliably_receive_TL()` functions, with the following requirements:

1. The data written to disk by the receiver must be exactly what the sender was given.
2. Two instances of your protocol competing with each other must converge to roughly fairly sharing the link (same throughputs $\pm 10\%$), within 100 RTTs. The two instances might not be started at the exact same time.
3. Your protocol must be TCP-friendly: an instance of TCP competing with your protocol must get on average 70% as much throughput as your flow.
4. Your protocol must not be overly nice to TCP: an instance of your protocol competing with TCP must get on average at least 70% as much throughput as the TCP flow.
5. All of the above should hold in the presence of any amount of dropped packets. All flows, including the TCP flows, will see the same rate of drops. The network will not introduce bit errors.
6. Your protocol must, in steady state (averaged over 10 seconds), utilize at least 70% of bandwidth when there is no competing traffic, and packets are not artificially dropped or reordered.
7. You cannot use TCP in any way. Use `SOCK_DGRAM` (UDP), not `SOCK_STREAM`.

2.2 Open-ended variant

Your job is to implement `reliably_transfer_OE()` and `reliably_receive_OE()` functions, with the following requirements:

1. The data written to disk by the receiver must be exactly what the sender was given.
2. Two instances of your protocol competing with each other must converge to roughly fairly sharing the link (same throughputs $\pm 10\%$), within 100 RTTs. The two instances might not be started at the exact same time.
3. All of the above should hold in the presence of any amount of dropped packets. All flows will see the same rate of drops. The network will not introduce bit errors.
4. Your protocol must, in steady state (averaged over 10 seconds), utilize at least 70% of bandwidth when there is no competing traffic, and packets are not artificially dropped or reordered.
5. You cannot use TCP in any way. Use `SOCK_DGRAM` (UDP), not `SOCK_STREAM`.

3 VM/Docker Setup - Simulating Network Conditions

When running your code locally, the network performance will be ridiculously good, so you'll need to limit it. The autograder uses `tc` for this purpose. If your network interface inside the container or VM is `eth0`, then run the following command from inside the container/VM to delete existing `tc` rules:

```
sudo tc qdisc del dev eth0 root 2>/dev/null
```

Now, you can use `tc` to adjust the network conditions. Below, we show you how to create a 20Mbit, 20ms RTT link where every packet sent has a 5% drop chance. Simply omit the `loss n%` part to get a channel without artificial drops:

```
tc qdisc add dev eth0 root handle 1:0 netem delay 20ms loss 5%
```

```
tc qdisc add dev eth0 parent 1:1 handle 10:  tbf rate 20Mbit burst 10mb latency 1ms
```

Add `sudo` to run these commands in VMs, if needed.

You can just run these commands just on the sender. If you want to run them on the receiver as well, be aware that the effects will stack, and adjust accordingly.

3.1 Debugging

Please do not fall into the trap of debugging via autograder. If you submit a new version every time you make some change that might help pass an extra test, you are going to waste a lot of time waiting for results. Instead, only submit when you have made major progress or have definitively figured out what you were previously doing wrong. If you aren't genuinely surprised that your most recent submission didn't increase your score, you are submitting too often!

For testing your code locally, we recommend using `tcpdump` or with `wireshark` to test network statistics.

The below command allows you to capture all the packets transmitted from the local system to the remote host at the given IP address into the file `capture.pcap`. You can then analyze these files using Wireshark:

```
tcpdump -i eth0 host <host IP address> -w capture.pcap
```

4 Submission

Similar to prior MPs, get the starter code from the release repository:

```
git fetch release
git merge release/main -m ‘‘Merging release repository (MP3)’’
```

4.1 Submission instructions

Submission instructions are the same as in MP2.

1. You are expected to work in groups of 2 on this MP. The partner with the lexicographically earlier NetID should submit the group registration form via Microsoft Forms (one submission per group; see the Campuswire announcement for details). If you wish to work alone, you must first request an exception on Campuswire and receive approval from the course staff before proceeding.
2. We will grade the repository of the partner with the lexicographically earlier NetID. Both partners must work out of that single repository to avoid confusion. If both partners submit separately, only the score from the lexicographically earlier NetID's repository will count.
3. Your final grade will be based on your **last** submission.
4. You may form different groups for different MPs, but group changes within the same MP are not permitted after registration unless you explicitly request an exception on Campuswire and receive approval from the instructors.

4.2 Grading

1. When the autograder is released, we will provide a score breakdown. Note that if compilation fails, no tests will be run, resulting in a score of 0 for the MP.
2. The autograder will test both your TCP-like and open-ended variants for the requirements specified in this document.

   Detailed per-test feedback including pass/fail results and your total autograder score will be available in the `_grades` branch of your GitHub repository after each submission.
3. Tests generally take **10-15 minutes**, and there may be a queue of students. Near the deadline, the queue may be an hour (or more) long. Please plan accordingly!
4. There will be hidden tests run after the MP deadline. These tests will not be available in the autograder. Please make sure you create your own test cases for a wide variety of workloads and network conditions!
5. You must use UDP for this MP. Use of TCP in any way for this MP will result in a zero.

4.3 Implementation advice

1. The MTU on the test network is 1500 bytes, so payloads up to 1472 bytes (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.

2. Your executables must use the interface specified in this document. Be careful about the executable and output file names. A single run of `make` (with no arguments) inside your MP3 directory should build both binaries.

3. Be sure 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.

4. Input files on the grader are READ-ONLY. Do not use the `rb+` mode to read them; the + adds write permission. (In general, you shouldn't use `rb+` unless you need it).

5. Input files on the grader are general binary data, NOT text.

4.4 Common Pitfalls to Avoid

1. For Windows users, we suggest avoiding the use of WSL. Because of the `tc` commands, it becomes more difficult to test your programs within WSL, even with Docker running inside WSL. Course staff also will not provide support with WSL troubleshooting.

2. With VirtualBox, if you haven't created a second VM yet, you may encounter an issue with identical IP addresses when you copy the first VM. Resolve this with running `dhclient -v` in the second VM.

3. If your program terminates too early, check if `fread()` returns early because of a flawed conditional statement.

4. When testing TCP friendliness, try reading from `/dev/zero` into `/dev/null` so the program doesn't crash from a large `numBytes` input.

5. Remember to clear the `tc` rules to reset your testing environment.

6. If the results return "infinite loop detected" for a test, it's likely your program isn't sending data fast enough. Try adjusting your congestion window size.

Please refer to the MP2 instructions for other notes.