COMP/ELEC 429/556 Project 2: Reliable File Transfer Protocol

Assigned: Thu, 28 September

Due: 11:55pm, Tue, 24 October

1 Description

The objective of this project is to give you hands-on experience with designing an end-to-end protocol, and to engage you in performance driven design and evaluation. In particular, you will design a simple transport protocol that provides reliable file transfer. Your protocol will be responsible for ensuring data is delivered in order, without duplicates, missing data, or errors. Since the local area networks at Rice are far too reliable, we will provide you with access to a machine that will emulate an unreliable network.

2 Requirements

You will write code that will transfer a file reliably from a sender host to a receiver host. You may assume that the receiver program is run first and will wait indefinitely for the sender program.

For this project, you **must not** use `SOCK_STREAM` sockets. You must design your own packet format and use `SOCK_DGRAM` sockets (which use the `UDP` unreliable datagram protocol) to transmit and receive packets. Your packet might include fields for packet type, acknowledgment number, advertised window, data, etc. This part of the assignment is entirely up to you. Your code MUST:

- Transfer the file directory name reliably.
- Transfer the file name reliably.
- Transfer the file contents reliably.
- The receiver must write the contents it receives into a file in the specified directory and name the file using the specified file name and adding the `.recv` file name suffix.
- Your sender and receiver programs must gracefully exit with an exit code of 0.
- Your code must be able to transfer a file with any number of packets dropped, damaged, duplicated, reordered, and delayed.

You may implement any reliability algorithm(s) you choose. We will test your code for correctness and measure the file transfer speed; better performance will result in a higher score. A total of 20 points are allocated to performance – 10 points for performance under no unreliable network behavior, and 10 points for performance under some specific unreliable network behavior.

Specifically, your solution's performance is graded relatively to the performance of other submissions. The score is assigned in an inverse-$\log_2$ style. Say the fastest solution receives 10 points with time t1, then any submission within 2*t1 will also receive 10 points. Then the best submission with performance t2 > 2*t1 will receive 9 pts. Similarly, any submission within 2*t2 will also receive 9 pts. We keep working on this calculation for all available performance results.

Remember that network-facing code should be written defensively. Your code should check the integrity of every packet received. We will test your code by corrupting packets, reordering packets, delaying packets, duplicating packets, and dropping packets; your programs should handle these errors gracefully, recover, and *not* crash.

2.1 File size

Students enrolled in COMP/ELEC 429 may assume that the file we ask your program to send will not be larger than 1 MB. This is to reduce the file I/O complexity that you have to cope with. For example, your program may read the entire file into memory before it commences transmission. If you choose to and successfully support large file sizes (up to 30 MB) using only (or approximately) 1 MB of memory at both sender and receiver, we will reward a 15 point extra credit to your submission. Be sure to indicate in the `README` file if you are seeking this extra credit.

The command `top` provides a straightforward way to check the amount of memory your programs are using. Don't stress over 0.5 MB vs 1.5 MB. The point is, if you are transferring a 30 MB file, and you are only using 1.5 MB, you are clearly handling the transfer in an intelligent manner and not just consuming memory linearly with the file size. The exact numeric value of the memory used isn't important, and that's why we emphasize "approximately 1 MB".

Students enrolled in COMP/ELEC 556 must ensure your program can handle large file sizes (up to 30 MB) using only (or approximately) 1 MB of memory.

3 Your programs

For this project, you will develop two programs: a `sendfile` program that sends the file across the network, and a `recvfile` program that receives the file and stores it to local disk. You may assume `recvfile` is run first and will wait indefinitely for the `sendfile` program. You may assume `recvfile` only receives one file from one sender during its lifetime. You must use C or C++. You must *not* use any transport protocol libraries in your project such as `TCP`. You must construct the packets and acknowledgments yourself, and interpret the incoming packets yourself. `sendfile` must use one `SOCK_DGRAM` socket for sending and receiving all packets. The `UDP` port number associated with this socket must remain the same during a file transfer. Likewise, `recvfile` must use one `SOCK_DGRAM` socket for sending and receiving all packets. The `UDP` port number associated with this socket must remain the same during a file transfer. Use `sendto()` and `recvfrom()` system calls for sending and receiving packets. See the manual pages `man sendto` and `man recvfrom` on CLEAR for details.

The command line syntax for your `sendfile` program is given below:

`sendfile -r <recv_host>:<recv_port> -f <subdir>/<filename>`

- `<recv_host>` (Required) The `IP` address of the remote host in `a.b.c.d` format.
- `<recv_port>` (Required) The `UDP` port of the remote host.
- `<subdir>` (Required) The local subdirectory where the file is located.
- `<filename>` (Required) The name of the file to be sent.

To aid in grading and debugging, your `sendfile` program should print out messages to the console:

- When `sendfile` sends a packet (including retransmission), it should print the following:
  `[send data] start (length)`
  where start is the beginning offset (in byte) of the file sent in the packet, and length (in byte) is the amount of the file sent in that packet.
- You may also print some messages of your own to indicate receiving acknowledgment, timeout, etc, depending on your design, but make it concise and readable. Note that too much printing could degrade performance.

The syntax for launching your `recvfile` program is:
`recvfile -p <recv_port>`

- `<recv_port>` (Required) The `UDP` port to listen on. `recvfile`'s `SOCK_DGRAM` socket must be assigned this port number using `bind()`. On CLEAR, the valid values to use (i.e. not blocked by the firewall) are 18000-18200 inclusive.

If `sendfile` sends a file named `x` in subdirectory `y`, `recvfile` should store the content in the subdirectory `y` in a file called `x.recv`. In other words, from the sender's view, `y` is a subdirectory of where the `sendfile` binary is run from. `recvfile` should store the received file in subdirectory `y` which is a subdirectory of where the `recvfile` binary is run from. You must use the correct directory name and file name with the added `.recv` file name suffix for the received file.

To aid in grading and debugging, your `recvfile` program should print out messages to the console:

- When `recvfile` receives a valid data packet, it should print:
  `[recv data] start (length) status`
  where status is one of `ACCEPTED(in-order)`, `ACCEPTED(out-of-order)`, or `IGNORED`.
- If a corrupt packet arrives, it should print
  `[recv corrupt packet]`
- Similar to `sendfile`, you may add your own output messages. Note that too much printing could degrade performance.

Both `sendfile` and the `recvfile` should print out the message `[completed]` after the completion of the file transfer, and then exit with an exit code 0.

4 Testing your code

In order for you to test your code over an unreliable network, we have set up a machine that emulates a network that may drop, reorder, damage, duplicate, and delay your packets. You must run `sendfile` on this machine, and `recvfile` on CLEAR.

The machine is `cai.cs.rice.edu`. An account on this machine has been created for you, using your NetID as the username. The initial password will be given out separately on Canvas. **Remember to change your password to protect the privacy of your work**. See `man passwd` for details. You will be able to `ssh` to the machine, `scp` files to the machine, and run your code on it. Again, you must run `recvfile` on CLEAR, and `sendfile` on `cai.cs.rice.edu`. If you have trouble with using `cai.cs.rice.edu`, please email the instructor.

You should develop your programs on the CLEAR machines and `cai.cs.rice.edu`. You are welcome to use your own Linux/OS X/Windows machines, but your code *must* work when graded on the CLEAR machines and `cai.cs.rice.edu`.

You may configure the emulated network conditions by calling the following program on `cai.cs.rice.edu`:

```
/usr/bin/netsim [--delay <percent>] [--drop <percent>]
                [--reorder <percent>] [--mangle <percent>]
                [--duplicate <percent>]
```

- `delay` This sets the percent of packets the emulator should delay. If not specified, this percentage is 0.
- `drop` This sets the percent of packets the emulator should drop. If not specified, this percentage is 0.
- `reorder` This sets the percent of packets the emulator should reorder. If not specified, this percentage is 0.
- `mangle` This sets the percent of packets the emulator should introduce errors into. If not specified, this percentage is 0.
- `duplicate` This sets the percent of packets the emulator should duplicate. If not specified, this percentage is 0.

Once you call this program, it will configure the emulator to delay/drop/reorder/mangle/duplicate *all* `SOCK_DGRAM` (i.e. `UDP`) packets sent by you at the specified rate. Additionally, it also affects `ICMP` packets; thus, you can use `ping` to observe the effects of `netsim` if you are interested. For example, if you called:

```
/usr/bin/netsim --delay 20 --drop 40
```

the emulator will randomly delay 20% of your packets and drop 40%. In order to reset it so that none of your packets are disturbed, you can simply call:

```
/usr/bin/netsim
```

with no arguments. Note that the configuration is done on a per-user-account, rather than per-project-group, basis. The emulator is also stateful, meaning your settings will persist across multiple login sessions (unless the machine or the `netsim` emulator is restarted). **IMPORTANT: Whenever testing your code, please run the `netsim` command on `cai.cs.rice.edu` first. Even if you don't specify any specific rule, just run `netsim` without parameters. The purpose is to provide a configuration for `netsim` for your user ID. If you don't run `netsim` once, `netsim` will not work for your user ID.**

5 Submitting your project

You may form a group of up to 3 students to work on this project. You should upload a `.tar` file containing your work to Canvas by 11:55pm on the due date. One submission per group is enough. This `.tar` file should unpack to have the following structure:

```
project2-{lastname1}-{lastname2}-{lastname3}/
                                README
                                Makefile
                                sendfile.{c, cc, h}
                                recvfile.{c, cc, h}
                                .... (other files)
```

You should include a `Makefile` which will compile your code. The `README` should contain the names of the students in your group, have a brief description and explanation of the packet formats, protocols, and algorithms you use, a list of properties/features of your design that you think is good, as well as examples of how to run your code, and anything else you want the TAs to know.

6 Advice

- Start by getting your code working with no `netsim` manipulations. You can check whether your code correctly transmits the contents of the file by using the `diff` and `md5sum` programs in Linux to compare the received file against the sent file.
- Test your code with each type of `netsim` manipulation separately and then in combination. Note that you will likely have to introduce multiple reliability mechanisms (checksums, timeouts, retransmits, etc.) in order to handle all of the possible errors.
- *Get started early!!*

7 Acknowledgment

We thank Professor Alan Mislove at Northeastern University for his generous help. This project is largely based on a project in Professor Mislove's course at Northeastern. Coincidentally, Professor Mislove is an alumnus of Rice.