Project 2: Designing and Optimizing a Reliable Data Transmission Protocol

Project Goals:

In this project, you will design, implement and optimize reliable data transfer protocols. A basic requirement of any reliable transfer protocol is ***correctness***, i.e., although the network itself may drop and reorder packets, application data such as a file should be transferred in order and without loss. However, an equally important consideration is ***performance***.

Consider a file of size ***F*** bytes transferred over a network with a bandwidth ***B*** Megabits/second. Owing to the need to wait for acknowledgments (ACKs), and if necessary retransmit data, it is unlikely any protocol can reliably transfer data at a rate of ***B*** Megabits/second. We refer to the ratio of the useful data transferred (file size ***F***) to the total transfer time as the ***goodput*** achieved by the protocol. Further, a reliable transfer protocol incurs ***overheads,*** which we define as ***T - F,*** where ***T*** is the total bytes transmitted by the sender or receiver over the network inclusive of ACKs, retransmitted packets, and header fields.

In this project, ***your goal is to design and implement a reliable transfer protocol that can achieve high goodput, and low overhead under diverse network conditions***. You will compare your protocol with a baseline scheme for comparison, which you will also implement. While we provide hints, the specific optimizations that you will implement in your protocol is up to you. While correctness of all implementations is a minimal requirement, the project will be primarily evaluated on (i) the kinds of optimizations you implement; (ii) the performance that you achieve; and (iii) a documentation of the performance of the schemes under different network conditions, along with a clear understanding of the design trade-offs. You will document the above information in a report, which is a mandatory requirement for the project.

Protocols that you will implement

You will implement the following protocols:

- **Baseline Scheme: Stop and Go:** In this protocol, the sender sends a packet and waits for an ACK before sending the next packet. If the corresponding ACK is not received within a predefined timeout period then the packet is retransmitted. This protocol results in a low header overhead but significantly lower goodput.
- **Your own optimized protocol:** You are free to design or optimize your protocol as you choose, but view a view point of achieving high goodput under diverse conditions, and low overheads.

We next discuss a strategy that could serve as a starting point. Here, the sender maintains a sliding window of packets to be transmitted to the receiver, moving the window forward as it receives ACKs. The sender transmits W packets to the receiver, where W is the window size. The receiver uses a cumulative ACK scheme. For example, consider that when transmitting a window of 5 packets, packet 3 is dropped but packets 1, 2, 4, 5 are successfully delivered. The receiver sends an ACK for packet 2 multiple times (when each of packets 2, 4, and 5 are received). Under this strategy, the sender will eventually timeout on packet 3, and retransmit packets 3, 4, and 5.

However, the above should only serve as a starting point, and we expect the best projects would include many other optimizations. Some hints to consider:

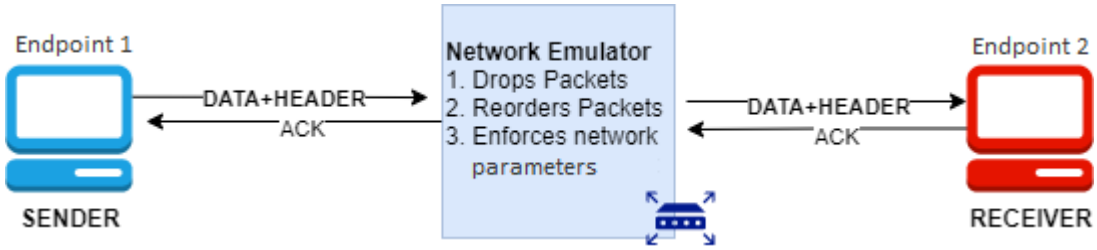
- Does a sender need to wait until a timeout to detect packet loss, or are there cases it can retransmit earlier?
- Are there disadvantages of a cumulative ACK scheme, and how could one do better (hint: selective ACKs)?
- What is a good window size to use and how would this depend on network settings such as the bandwidth and delay?
- What is a good choice of timeout? (hint: ideally, the timeout should match the round-trip time of the network, but it is desirable to leave some slack for delay variability. The amount of slack may need to be experimentally tuned and reported).

Starter Code

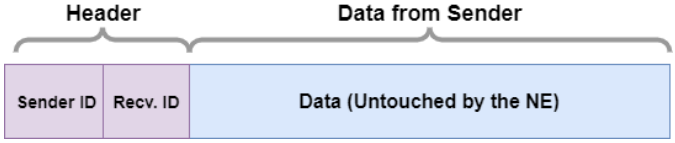
We provide code for (i) a network emulator which emulates network conditions based on settings specified in a configuration file; and (ii) a module monitor.py which interfaces with the network emulator, and must be included in your sender and receiver code. The starter code is located in the Student Code/example directory.

Network Emulator (NE)

The NE emulates an unreliable data transfer layer which can cause **data loss, and reorder packets.** (See figure 1). The NE is also responsible for enforcing various parameters such as the propagation delay and bandwidth of the network (see Network Parameters for a detailed list). The parameters are set using the configuration file (.ini format).



(a)



(b)

Figure 1: shows (a) the emulator setup and functions; (b) the packet format required by the NE for transmission.

Network Parameters

- PROP_DELAY or Propagation Delay is the time it takes for the packet to travel from the sender to the receiver packet. This is separate from the transmission delay (due to Link Bandwidth). This field is in seconds.
- MAX_PACKET_SIZE is the maximum payload/data transferred per packet. In figure 1(b) (Header + Data) should be less than or equal to the MAX_PACKET_SIZE. Larger packet sizes are not accepted and dropped.
- LINK_BANDWIDTH is the link bandwidth at which the packets are pushed onto the channel (in this case simulated by the NE). Transmission delay is the length of the packet divided by the link bandwidth. This field is in Bytes per second.
- MAX_PACKETS_QUEUED gives the upper limit on the number of outstanding packets that can be queued in the NE. Any new packets are dropped. This value is set using the bandwidth-delay product of the network.
- DROP_MODE sets the pattern used to generate losses by the NE. DROP_MODE=1 results in packets being dropped randomly (the probability of drop is set based on RANDOM_DROP_PROBABILITY). DROP_MODE=2 is a setting where packets are dropped with a probability based on the queue length in a router buffer that the NE emulates. . For this project, only DROP_MODE=1 will be used.
- RANDOM_DROP_PROBABILITY sets the probability at which packets are dropped by the NE in DROP_MODE=1. If this field is set to 0 then no drops occur. For (0, 1], the drops occur at the probability value specified.
- REORDER_PROBABILITY sets the probability at which packets are reordered by the NE. If this field is set to 0 then no reorders occur. For (0, 1], the reorders occur at the probability value specified. The reordered packets are inserted back into the internal NE queue to be transmitted later.

Note: A packet is transmitted by the NE between the endpoints solely based on the destination id given. The network emulator doesn't check or modify the packet content in any way. The packet can contain any number of header fields and in any format (there is no need to comply with TCP's header format). The only constraint is the MAX_PACKET_SIZE parameter mentioned above.

Configuration File

The network parameters are specified in a configuration file (config.ini), which can be tweaked to emulate a real-world scenario. The network emulator reads the configuration file (config.ini) and sets the various parameters.
The configuration file contains the host and port information for all endpoints (we will use the term endpoint to refer generically to the sender and receiver) involved. All parameters of the configuration file should be set before launching the NE or the endpoints. The NE reads the host/port information to track senders and forwards messages to the correct receivers.
The Network Monitor reads the configuration file, binds to the appropriate port, and connects to the NE using its relevant host/port information.. The configuration file also contains necessary filenames (e.g., file being transmitted, file used for logging etc.) Each endpoint(Sender/Receiver) should instantiate a Monitor object and pass the config file path and the correct header.

Your code should run based on the parameters in the NODES section of the config file. However, **your code may (and should!) also read the network parameters in this file so that it may set the window size and other parameters appropriately**.

The Config file further contains the fields for the file being sent (`file_to_send=./to_send.txt`) and the output file on the receiver end (`write_location=./received.txt`). These fields should be read by the sender and receivers to be able to transmit the file. *See references to understand how to read the config file.*

Network Monitor

The network monitor runs alongside the endpoint, and provides the interface to send/receive packets as well as to indicate when the transmission is complete. The monitor also extracts network information such as data overhead, goodput, and the number of packets transmitted. The network monitor on the receiver side checks for the correctness of the transmission as well. The Python module 'monitor.py' should be included in each of your endpoint programs. Copy monitor.py to the same directory as your endpoint code. Information on how to interface with the Monitor is provided below:

- Initialize a Monitor object with the following parameters:
    - configFile - path to the Configuration file
    - config_heading - Defines the section to read the fields from. Either 'sender' or 'receiver' based on whether the endpoint is a sender or a receiver.
- Below are the ways that your sender and receiver can use the Monitor object that you created:
    - Send/receive packets using the monitor (which handles communication with the NE): Call Monitor.send(dest, data) to send a packet. This sends the packet without

modifying it to the network emulator to be forwarded to the destination (dest). This calls socket.sendto() in the background.

- Call Monitor.recv(Bytes) to recv an incoming packet. This calls socket.recvfrom() in the background.
- Call Monitor.send_end(dest_id) after the transmission to the endpoint (dest_id) is complete. This is to be called after the sender knows that the receiver has received the final packet (Eg. After you receive the ACK for the final packet in Stop and Go protocol). This is only relevant to the sender.
- Call Monitor.recv_end(recv_file, sender_id) once the entire file is received and written to 'recv_file'. This will check for the correctness of the file. This is only relevant for the receiver.

Viewing Performance Statistics

After the file has been successfully transferred and send_end() and recv_end() functions are called, the monitor prints out the performance statistics to "*sender_monitor.log*" and "*receiver_monitor.log*".

Sender_monitor.log

An example sender log is shown below: .

1. *1617394269.3555202*
2. *Id:1 | Starting monitor on ('localhost', 8001).*
3.
4. *File Size : 628370 bytes*
5. *Total Bytes Transmitted : 659730 bytes*
6. *Overhead : 31360 bytes*
7. *Number of Packets sent : 663*
8. *Total Time : 162.87 secs*
9. *Goodput : 3858.15 bytes/sec*

Here, Line 1 is the timestamp. *Overhead* is simply the extra bytes transmitted beyond the actual file size, including the header per packet and retransmitted packets, while *Goodput* is the effective throughput which is the useful bytes (file size) transmitted divided by the total time taken. You should primarily be reporting the *Overhead* and *Goodput* numbers in your report.

Receiver_monitor.log

An example receiver log is shown below:

1. *1617394268.3460567*
2. *Id:2 | Starting monitor on ('localhost', 8002).*

3.
4. *File transmission correct : True*
5. *Number of Packets Received : 649*
6. *Total Bytes Transmitted : 645761*
7. *Total Time : 162.66 secs*

Here, Line 1 is the timestamp, while Line 4 indicates whether the transmitted file was received reliably or not. Note that there is a slight difference between the bytes and packets reported at the sender and receiver owing to packet drops. The total time is also slightly different owing to differences in when timers are started and stopped. Please report the numbers in the sender log in your report.

What you need to do

For this project, you are required to implement two different reliable transport protocols by writing the sender and receiver code for each of them. The test config files stress different aspects of the protocol implemented.

1. Write sender and a receiver program using the Stop and Go Protocol. Place your code in a directory named "Student Code/stop_and_go". Provide a Makefile in this directory that supports 'make run-sender config=<configfile>' and 'make run-receiver config=<configfile>' to run the sender and receiver codes with the supplied configuration file. (See BASH command-line arguments).

2. Write sender and receiver code based on the protocol that you have designed. Place your code in a directory named "Student Code/student". Provide a Makefile in this directory compatible with 'make run-sender config=<configfile>' and 'make run-receiver config=<configfile>' to run the sender and receiver codes with the supplied configuration file. (See BASH command-line arguments).

***Note 1**: The performance of the above protocols depend on the choice of the timeout value, and the choice of window size, which itself needs to be chosen in a configuration-specific way. Please make sure to set the timeout and window size automatically to values that are reasonable for different configurations using good rules of thumb.*

Note 2: Since we are emulating the data transfer delays using the emulator it is noted that for higher bandwidth and larger window sizes the performance of the emulator significantly degrades. To avoid this, we strongly recommend limiting the window size to 50 in all cases.

Note 3: If your Python code is excessively slow in critical paths, you may encounter significant delays in packet transmission/receipt that affects your final goodput. To avoid this, we recommend avoiding unnecessary operations in time-sensitive portions of your code. The biggest offender here are print/log statements: Avoid printing/logging in any time-sensitive sections of your code.

Steps your code should follow

In all cases, your code should follow the following steps **in order** to ensure proper functionality with the Network Emulator and Network Monitor.

1. Initialize the sender and receiver network monitors.
2. Start the receiver. After a short delay (~1 second), start the sender.
3. Exchange messages through Monitor.send() and Monitor.recv().
4. When the receiver receives the final packet in the message, call Monitor.recv_end(). **IMPORTANT: Leave the receiver running and sending ACKs to any additional packets that may arrive!**
5. When the sender receives an ACK for the final packet, call Monitor.send_end(). This will shut down the network emulator.

The order of steps 4 and 5 are important to ensure proper behavior. Consider the case where the sender sends the last packet and the receiver's ACK is dropped: If the receiver has quit, the sender will continue retransmitting the packet forever and getting no response. Therefore, the receiver should continue running and ACKing any retransmitted packets.
Likewise, the sender should not call send_end() or quit until it receives the ACK for the final packet. If it quits prematurely, it can not be certain that the receiver has gotten all packets.

Running your code

We will be testing your code using the configuration files similar to the ones provided under TestConfig in the starter code. The network emulator will be launched before the endpoints. Then the endpoints will be launched using the Makefile provided by you. Here "config.ini" is the command-line parameter passed to the Make run commands.

What you need to submit

1. The directories corresponding to the parts mentioned above.
2. The report based on the provided template, and following all the guidelines.

***Do not submit any binaries*. Your git repo should only contain source files; no products of compilation.**

Grading

A minimal expectation for all protocol implementations is correctness. A major criterion for grading is your performance results, whether the overall trends make sense, and the effort and creativity you have shown in adding optimizations. The clarity of report (documenting the effort, appropriately presenting graphs and containing the right level of detail in the writing) will also be a factor.

References and Useful Information

The configuration file is to be extracted and read using the configparser library. See the link below for more information. Using pip, can be installed with the command “pip3 install --user configparser”. This command is provided in the starter code makefile and can be run with “make configparser”.
https://docs.python.org/3/library/configparser.html