Programming Assignment 2 (MP2)

Implementing a reliable transport protocol over UDP

MP2 may be performed by a group consisting of up to 2 students.

- **Completed MP2 is due by 7 p.m. on Monday, November 11, 2013**
- **Intermediate deadline: By 7 p.m. on October 21, 2013 submit code that sets up UDP sockets at the sender and the receiver.**

NOTE: This handout contains several changes as compared to the DRAFT version. Please read this handout in its entirety.

You may use C, C++, Python or Java for MP2. However, note that the course staff may not be able to provide debugging help for Python and Java.

The goal of MP2 is to implement a reliable transport layer protocol over UDP. The primary objective of MP2 is to better understand the following mechanisms in TCP: (i) congestion avoidance and control, (ii) fast retransmit and fast recovery, (iii) calculation of RTO and use of timeouts, and (iv) slow start. You are not required to implement connection management part of TCP.

Your implementation should to satisfy the following requirements:

- Reliable in-order data delivery should be possible from a process designated as the "sender" to a process designated as the "receiver". The sender should deliver a file to the receiver using your implementation of TCP.
- UDP should be used to transmit "segments" containing no more that 100 bytes of *data*. (It may contain additional bytes for "header".)
- Your implementation should emulate the TCP functionality summarized in the state diagram in Figure 3.52 of 6th (or 5th) edition of the Kurose & Ross textbook, as well as TCP's round-trip time estimation and timeout calculation, as described in Section 3.5.3 of the textbook (6th or 5th edition). You need not implement delayed acknowledgments – instead, the receiver may acknowledge each received segment. Assume that the initial slow start threshold (*ssthresh*) equals 1000 bytes. Assume that 1 MSS = 100 bytes. The initial congestion window, as well as the congestion window size after a timeout, equals 1 MSS. As in TCP, each byte of the data should be assigned a sequence number.

When the above specifications are ambiguous, you are free to make use of reasonable interpretations. You may also deviate from the specifications in ways that does not affect the primary goal of implementing a TCP-like protocol. As an example, during congestion avoidance, Figure 3.52 specifies that cwnd is incremented by MSS*(MSS/cwnd). Since this may result in fractions, you may instead increment the cwnd by a multiple of 10 bytes that is closest to MSS*(MSS/cwnd), or another such approximation. As another example, the RTO value calculated by the TCP algorithm may be too small to be implemented with the timer granularity available in Linux – in this case, it is acceptable to use a larger timeout than the estimated RTO.

In a README file to be submitted with your code, please document & briefly justify such changes you may have chosen to make.

- Your implementation may not use a TCP socket. All communication between the sender and the receiver should be over UDP.

- You may choose to implement your own header format. The header need only include information necessary to achieve the above functionality. Your header format need not be identical to the formats used for TCP/IP.

- The sender should assume that the receiver has a receive buffer that allows storing at most 2500 bytes of *data*. The sender should send segments containing 100 bytes whenever enough data is available. Thus, with the 2500 byte constraint on the receiver buffer, there should not be more than 25 segments unacknowledged at any given time.

Your *sender* will be tested in conjunction with your *receiver*. Thus, they only need to be compatible with each other.

You may find useful the talker.c and listener.c source code available on Beej's guide, which illustrates the use of UDP datagram sockets. You may use any publicly available code for the purpose of setting up the UDP sockets. However, each group must develop the rest of the code independently.

We recommend the following structure for your implementation:

- Write an initialization function that handles the creation of sockets.

- Write a function that the sender can use to send a single "segment" to the receiver. The segment should contain any necessary "header" and the data. Write a corresponding function that the receiver can use to receive a single segment, and then extract the header and data from the received segment.

- Maintain "state variables" to keep track of the current state at the receiver. Write separate functions for packet handling in each state.

Usage format for sender and receiver:

- sender *filename receiver-domain-name receiver-port*
  filename specifies the file to be transferred. *receiver-domain-name* and *receiver-port* specify the domain name of the host running the receiver and the UDP port that the receiver will receive on.

- receiver *receiver-port losspattern*
  receiver-port specifies the UDP port the receiver should receive on. *losspattern* is a file that allows emulation of a lossy network, as elaborated below.

We will only test your code with plain-text files. The receiver should print the received data to the standard output. The sender and receiver should terminate after the file transfer is complete. Thus, you will need to include a mechanism for the sender to inform the receiver when a certain segment contains the last bytes of the transfer (for instance, this may be achieved by setting a specific flag in the header).

Emulating lossy network:

The first segment received by the receiver is considered to have *receive sequence number* (RSN) of 1. The RSN is incremented by 1 thereafter. These RSNs are only to be used for emulating losses. Note that each received segment is given a new RSN for the purpose of determining which segment to discard. Thus, if the receiver receives the same segment twice, then the two copies will be assigned different RSNs.

Two types of losses will be specified, *periodic* and *specific*. The *losspattern* file contains a small number of non-negative integers. The first number in the file determines the nature of losses. There are three possibilities for the first number in the file:

- 0 : in this case, the receiver should not discard any received segments

- 1 : in this case, the second number specifies the period at which packets should be discarded. For instance, if *losspattern* file contains 1 8 then every 8th segment should be discarded by the receiver, starting with the segment with RSN 1. Thus, segments 1, 9, 17, 25, … are discarded on receipt at the receiver.

- 2: in this case, the remaining numbers specify the RSNs corresponding to segments that should be discarded. These RSNs will be listed in an increasing order. You may assume that no more than 20 such segments will be specified. For instance, if the file contains
  2 5 7 8 9 12 20 50
  then the receiver should discard segments with RSN 5, 7, 8, 9, 12, 20 and 50.

When a segment is discarded, the receiver should effectively behave as if that segment was never received.

The sender does not discard any acknowledgments it receives.

Submission

There are two deadlines for MP2 as listed at the start of this document, an intermediate deadline, and a deadline for the completed MP2. The 48 hour extension applies to each of these deadlines.

Submission instructions for the intermediate deadline

> By the intermediate deadline, you should have written adequate code to set up UDP sockets at the sender and receiver, and tested that the two processes are able to communicate (e.g., send a segment or an Ack) over these sockets.
>
> Submit **your code**, a README file that lists the names of group members, and the functionality you have **implemented & tested for correctness** by the intermediate deadline. Although we will not be testing your intermediate submission for correctness, it is in your interest to produce working code with the above functionality by the intermediate deadline. Ideally, you should have made additional progress on the project by the intermediate deadline.

Submission instructions for the completed MP2

For each run (performing one file transfer from *sender* to *receiver*), the *sender* should save an output file called *trace*. The sender should add to the *trace* file one line for each *new* acknowledgement it receives, recording the time at which the new ack is received, and the largest sequence number acknowledged. For instance, a line of the trace file may be:

12.340451 8900

where 12.340451is the time at which the sender receives an acknowledgement that acknowledges sequence number through 8900. Similarly, the sender should also save an output file called *cwnd*. Whenever the congestion window at the sender is modified, the sender must save a line in the cwnd file. For instance, a line in the *cwnd* file may be:

12.470042 1200

which indicates that the sender changed its congestion window to 1200 bytes at time 12.470042.

We will make use of the *trace* and *cwnd* files for grading.

Loss pattern files named 0, 1, 2, 5, 10 and 20 will be made available on the course website. We may also run your code with other loss patterns for the purpose of grading.

(a) For *loss pattern* files named 5, 10 and 20, and for one data file each of size equal to 1000 bytes, 100000 bytes and 1000000 bytes, plot the throughput (in bits/second) for the file transfer. Plot all the curves in the same figure. The horizontal axis should correspond to the loss pattern name, and vertical axis should correspond to throughput. Different

curves in the figure should correspond to different file sizes. If you find it convenient, the file sizes used may differ from the specified sizes by up to 100 bytes.

(b) For a file of size 10000 bytes, and for loss pattern files named 0, 1 and 2, **using the *trace* file**, produce a plot of sequence number versus time, similar to that in Wireshark assignment #2.
(i) In the plot for loss pattern file named 0, show where the initial slow start ends.
(ii) In the plot for loss pattern file named 1, show where a fast retransmission occurs.
(iii) In the plot for loss pattern file named 2, show each retransmission occurs due to a retransmission timeout.

(c) For a file of size 10000 bytes, and for loss pattern files named 0, 1 and 2, **using the *cwnd* file**, produce a plot that shows the congestion window size versus time.
(i) In the plot for loss pattern file named 0, show where the initial slow start ends.
(ii) In the plot for loss pattern file named 1, show where fast recovery begins and end.
(iii) In the plot for loss pattern file named 2, show where slow start begins after the first retransmission timeout.

For creating the plots, you may want to use graph plotting tools such gnuplot or Microsoft Excel.

Submit parts (a), (b), and (c), along with your code, by committing it to your SVN folder, as in MP1. Also, **similar to MP1,** please submit the README file, make file, or adequate instructions for compiling the code, as applicable. Be sure to list names of the group members in the README file. See MP1 instructions for additional details.