CS 455: Computer Communications and Networking
(Spring 2024)
PA-2: Mason Transport Protocol
Due Date: [redacted]

Project overview

In this assignment, you will be implementing TCP's reliable transport protocol to transfer a file from one machine to another. However, instead of using TCP sockets, you will use UDP. This means that your new protocol will provide in-order, reliable delivery of UDP datagrams in presence of packet loss and corruption. Your task is to implement a sender and a receiver that connect with each other over UDP. We will call this protocol MTP – the Mason Transport Protocol. The sender delivers a file to the receiver using MTP. Upon receiving the file, the receiver should be

MTP Specifications.

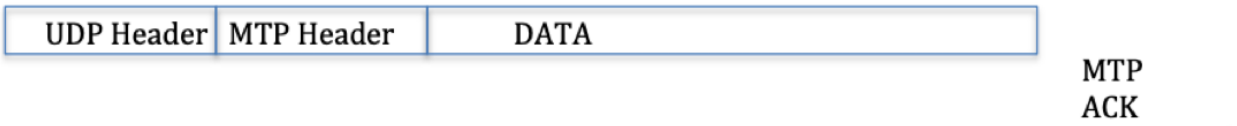
Types of packets and packet size

MTP requires only two types of packets: DATA and ACK. In our simple setting of sending a file from sender to the receiver, the sender opens a UDP client socket and the receiver opens a UDP server socket. The sender then sends DATA packets, and when the receiver receives them, ACK packets are sent back.

Since MTP is a reliable delivery protocol, the DATA packets and ACKs must contain a sequence number. The data packet has a header with the following fields
1. type (unsigned integer): data or ack
2. seqNum (unsigned integer): sequence number
3. length (unsigned integer): length of data
4. checksum (unsigned integer): 4 bytes CRC

The MTP DATA packet looks as shown below. Note that your MTP header + data is encapsulated in UDP header since the sender and receiver connect over a UDP socket. As we have discussed, the UDP header is already added by the socket interface, and something that you do not have to implement.

MTU size of 1500 bytes. With 8 bytes of UDP header and 20 bytes of IP header, sizeof(MTP header + data) can be no more than 1472 bytes.



MTP reliable delivery policy

MTP relies on the following rules to ensure reliable deliver:

- Sequence number: Each packet from sender to receiver must include a sequence number. The sender reads the input file, splits it into appropriate sized chunks of data, puts it into an MTP data packet, and assigns a sequence number to it.
- The sender will use 0 as the initial sequence number.
- Unlike TCP, we use sequence numbers for each packet and not for the byte-stream.
- Checksum: The integrity of the packet (meaning if it is corrupt or not) is validated using a checksum. After adding the first three fields of MTP header and data, the sender calculates a 32 bit checksum and appends it to the packet. *You can use existing implementations of CRC32 that you find on the web for calculating your checksum. Just make sure that you document where you got the code from.* Please note that we will discuss CRC algorithms in the link layer, but you can glance at Section 6.2.3 in the book.
- Sliding window: The sender uses a sliding window mechanism that we discussed in class. The size of the window will be specified in the command line. The window only depends on the number of unacknowledged packets and does not depend on the receiver's available buffer size (i.e., MTP does not have any flow control).
- Timeout: The sender will use a fixed value of 500ms as the timeout

Sender Behavior:

- The sender obeys the following rules when sending data to the receiver

Event at sender	Action taken by sender
Data received from above (i.e., more data to be sent to complete sending the file)	- Create a packet, assign sequence # and checksum. - If the window allows, send out the packet and start a timer. - If not, buffer the packet until the window advances.
ACK received for lowest (unacked) sequence # packet in the window	Advance the window and send out more packets if there is more data to send, also start the timer
Triple duplicate ACKs received	Retransmit the oldest unacked packet and start the timer
Timeout	Retransmit the oldest unacked packet and start the timer
Receive a packet (ACK) that is corrupt	Ignore

Receiver Behavior

The receiver obeys the following rules while ACKing the sender

Event at receiver	Action taken by receiver
Arrival of an in-order packet with expected sequence #, and all data upto expected sequence # already ACKed	Delay ACK: Wait up to 500 ms for the next packet. If no packet arrives within that time duration, send ACK
Arrival of an in-order packet with expected sequence #, one other segment has ACK pending	Immediately send a single cumulative ACK, ACKing both in-order segments.
Arrival of an out-of-order packet with higher-than-expected sequence # (i.e., gap detected) or arrival of a packet that is corrupt	Immediately send a duplicate ACK, indicating the sequence # of next expected packet
Arrival of a segment that partially or completely fills up the gap	Immediately send an ACK if needed

Implementation and testing

Running the MTP sender and receiver

Your sender should be invoked as follows:

$> ./MTPSender <receiver-IP> <receiver-port> <window-size> <input-file> <sender-log-file>

<receiver-IP> and <receiver-port> are the IP address and port number of MTPReceiver respectively.
- <window-size> size of the window.
- <input-file> will be the file that sender sends to the receiver.

- <log-file> should log the event occurring at the sender.

A sample file can look like

$> cat sender-log.txt
Packet sent; type=DATA; seqNum=0; length=1472; checksum=62c0c6a2
...
Updating window; (show seqNum of 64 packets in the window with one bit status (0: sent but
not acked, 1: not sent)
Window state: [20(0), 21(0), 22(0), 23(0), ..., 81(1), 82(1), 83(1)]
...
Packet received; type=ACK; seqNum=0; length=16; checksum_in_packet=a8d38e02; checksum_calculated=a7d2bb01; status=CORRUPT;
...
Timeout for packet seqNum=21
...
Triple dup acks received for packet seqNum=34
...

Your receiver should be invoked as follows:

$> ./MTPReceiver <receiver-port> <output-file> <receiver-log-file>
- <receiver-port> is the port number on which the receiver is listening
- <output-file> the received data should be stored in the output file. After completion of your code, the output-file should match exactly with the input-file. - <receiver-log-file> should log the events occuring at the receiver.

A sample file will look like:

$> cat receiver-log.txt
Packet received; type=DATA; seqNum=0; length=1472; checksum_in_packet=62c0c6a2; checksum_calculated=62c0c6a2; status=NOT_CORRUPT
...
Packet received; type=DATA; seqNum=1; length=1472; checksum_in_packet=a8d38e02; checksum_calculated=62c0c6a2; status=CORRUPT

...
Packet sent; type=ACK; seqNum=0; length=16; checksum_in_packet=a8d38e02;
...
Packet received; type=DATA; seqNum=10; length=1472; checksum_in_packet=62c0c6a2; checksum_calculated=62c0c6a2; status=OUT_OF_ORDER_PACKET
...
Note that there will be other events on sender and receiver that are not shown in the example log output above. Please make sure to include them in your implementation.

Implementation and testing

You can implement the sender and receiver using
(1) A single machine with localhost IP and different ports for server and client. In this case, you are simply transferring a file from the client's directory to the server's directory. This would be a good place to start developing your code.
(2) Two laptops connected over the Internet. For instance, you and your partner can remotely connect to each other over the Internet and test your code in real-world settings. Here the file will be transferred from one machine to the other.

You can generate a file of an arbitrary size using the following command. For example, $> base64 /dev/urandom | head -c 1000000 > input-file.txt

will generate a text file of 1MB. You can start by first transferring a small file and then gradually test your code for larger files.

You can verify that your file has been correctly received by calculating sha1sum of your file on sender and receiver.

$ sha1sum input-file.txt 381e08efd0d8182d2a559321b2b60234010f74bc input-file.txt

$ sha1sum output-file.txt 381e08efd0d8182d2a559321b2b60234010f74bc output-file.txt

Some free code

You are given a few Python files to get you jump started. You may modify them in any way you like and/or reimplement them in C. *Please document what you do!* Ultimately you will be handing in files with these names.

1. MTPSender.py which provides a high-level code structure for client
2. MTPReceiver.py which provides a high-level code structure for server
3. unreliable_channel.py: It is possible that when you are running the client and server on the same machine, you are likely to see no packet loss or corruption. To address this, we have provided you with two sample functions (recv_packet and send_packet) that cause packet loss and corruption to simulate an unreliable channel. Use of these functions is shown in client.py and server.py
4. 1MB.txt: a sample text file which you can use as the file to transfer from client to server

Policies

Programming Language: C or Python, please.
Partners: Please work with your PA1 partner, if you have one.

Cheating

As discussed, it is absolutely mandatory that you adhere to GMU and Computer Science Department honor code rules. This also means (1) do not copy code from online resources and (2) your implementation should be purely based on your own thought process and conceived design, and not inspired by any other students or online resources, (3) you do not copy your code or design from other students in our class.

We reserve the right to check your assignment with other existing assignments (from other students or online resources) for cheating using code matching programs. As you know, any violation of honor code are reported to the University-level GMU honor committee, with a *minimum* penalty of receiving a failing grade (F) in this course and the notation **HC** on your transcript.

*** Do not put your code online on Github or other public repositories *** Violation of this policy would be considered an honor code violation

Grading, submission and late policy

- Standard late policy applies - Late penalty for this project is be 15% for each day.
- *Submissions that are late by 2 days or more will not be accepted*
- You will submit your solution via Blackboard

Submit the following

1. Your code in a zip archive file. The code should contain two files MTPSender and MTPReceiver which we will compile and execute to test your implementation. If you simply include a pre-compiled executable binary and do not include your code files in the archive, *no points will be given.*

2. A README.txt which explains how to compile your program. Once compiled, your sender and receiver will be run using the command line arguments described above. Your code must not accept any more or fewer command line arguments than what we specify.

3. A sender-log-file.txt and receiver-log-file.txt that you got by running your code on your machines while transferring the provided 1MB.txt file.

4. For those of you who work as a team, a PARTNERS.txt file stating the name and GMU IDs of two students worked on the project. Both team members should submit these four pieces separately on Blackboard before the deadline.

Acknowledgements

This programming assignment was originally based upon an assignment developed by Dr. [redacted] and is also based on UC Berkeley's Project 2 from EE 122: Introduction to Communication Networks and UMich's EECS 489 Computer Networks Assignment 3.