

CS 455: Computer Communications and Networking

PA-2: Mason Transport Protocol

[Project Description](#)

[Project overview](#)

[MTP specifications](#)

[Types of packets and packet size](#)

[MTP reliable delivery policy](#)

[Implementation and testing](#)

[Running the MTP sender and receiver](#)

[Implementation and testing](#)

[Starter files](#)

[Grading rubric](#)

[Policies and submission](#)

[Programming language](#)

[Working with a partner](#)

[Note on plagiarism](#)

[Grading, submission, and late policy](#)

[What to submit?](#)

[Acknowledgments](#)

Project Description

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 (let's call it MTP - Mason Transport Protocol) should 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. The sender delivers a file to the receiver using MTP. Upon receiving the file, the receiver should be able to verify the integrity of the file.

MTP specifications

Types of packets and packet size

MTP specifies only two types of packets: DATA and ACK. In our simple settings 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. MTP does not use a 3-way handshake as TCP.

Since MTP is a reliable delivery protocol, the DATA packets and ACKs should 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

MTP DATA packet looks as shown below. Note that your MTP header + data is encapsulated in the UDP header since the sender and receiver connect over a UDP socket. Of course, the UDP header is already added by the socket and something that you don't have to implement.

UDP header	MTP header	data
------------	------------	------

MTP ACK packet will have the same fields in its header, without any data. We will assume MTU size of 1500 bytes. With 8 bytes of UDP header and 20 bytes of IP header, the size of (MTP header + data) can be no more than 1472 bytes. Assuming 4 bytes for unsigned int, the MTP header will be 16 bytes for the 4 fields mentioned above. The header size remains the same in all packets.

MTP reliable delivery policy

MTP relies on the following rules to ensure reliable delivery:

- Sequence number: Each packet from the sender to the receiver should 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 will use sequence numbers of each packet and not for the byte-stream.
 - ACK packets will have a sequence number of the next expected packet, just like TCP.
- 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 the MTP header and data, the sender calculates a 32 bits checksum and appends it to the packet. You can use the existing implementation (e.g., python library) of [CRC32](#) for calculating your checksum. To calculate the checksum, append your data to the first 3 header fields (i.e., type, seqNum and length) and calculate the checksum. Once calculated, insert it back between the header field "length" and data before sending the packet over.
- 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). Remember your implementation should be a pipelined protocol (can send more than one packet at a

time). Do not implement the project as a stop-and-wait protocol where you send one packet and receive an ACK. However, in the pipelined protocol, it is still possible that you are only sending one packet sometime because that is all that the current state of window allows. *If you implement your project as stop-and-wait, you will get zero points for the project.*

- Timeout: The sender will use a fixed value of 500ms as the timeout.
- Termination condition: Send an empty packet of size zero after all the file packets have been sent from the sender to indicate to the receiver that the transfer has been done. The receiver can stop listening for more packets and close it's socket.
- Logging: Log each event on both sides in a log file based on the rules.
- The receiver uses the following rules while ACKing the sender

Event at receiver	Action taken by receiver	Log Entry
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	Packet received; type=DATA; seqNum=<next_seq_no>; length=1472; checksum_in_packet=62c0c6a2; checksum_calculated=62c0c6a2; status=NOT_CORRUPT Packet sent; type=ACK; seqNum=1; length=16; checksum_in_packet=a8d38e02;
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.	Packet sent; type=ACK; seqNum=<next_seq_no>; length=16; checksum_in_packet=a8d38e02;
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	Packet received; type=DATA; seqNum=10; length=1472; checksum_in_packet=62c0c6a2; checksum_calculated=62c0c6a2; status=OUT_OF_ORDER_PACKET Packet sent; type=ACK; seqNum=<next_seq_no>; length=16; checksum_in_packet=a8d38e02;
Arrival of a segment that partially or completely fills up the gap	Immediately send an ACK if needed	Packet sent; type=ACK; seqNum=<next_seq_no>; length=16; checksum_in_packet=a8d38e02;

- The sender uses the following rules while sending the DATA packets

Event at sender	Action taken by sender	Log Entry
-----------------	------------------------	-----------

Data received from above (i.e., more data to be sent to complete sending the file)	<ul style="list-style-type: none"> - 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. 	Making packets for file <file_name> Packet sent; type=DATA; seqNum=0; length=1472; checksum=62c0c6a2 Window full, waiting for acks. Window state: [20(1), 21(1), 22(1), 23(1), ..., 81(0), 82(0), 83(0)]
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	Updating window; (0: sent but not acked, 1: acked) Window state: [20(1), 21(1), 22(1), 23(1), ..., 81(0), 82(0), 83(0)]
Triple duplicate ACKs received	Retransmit the oldest unacked packet and start the timer	Triple dup acks received for packet seqNum=34 Retransmitted pkt; type=DATA; seqNum=34; length=1472; checksum=62c0c6a2
Timeout	Retransmit the oldest unacked packet and start the timer	Timeout for packet seqNum=21 Retransmitted pkt; type=DATA; seqNum=21; length=1472; checksum=62c0c6a2
Receive a packet (ACK) that is corrupt	Ignore	Corrupt ACK received, ignoring.

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: acked)
```

```
Window state: [20(1), 21(1), 22(1), 23(1), ..., 81(0), 82(0), 83(0)]
```

```
...
```

```
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 occurring at the receiver.

A sample file can 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 can 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) (optional) Two laptops connected over the Internet. 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 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. *Your code will only be tested on text 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
```

Starter files

You have been provided with 4 files to get started:

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

Grading rubric

Handling the input arguments on both sides	5 points
Sender: Read the input file and create packets Send the packets (window implementation) Receive the ack and check for corruption Timer implementation 3 dup acks implementation	10 points 15 points 10 points 15 points 5 points
Receiver: Receive and parse the packets and check them for corruption Timer implementation Regular and Cumulative ack implementation	10 points 15 points 5 points
Log: Saving the results to files	10 points

Your code will be tested for 3 file transfers and randomly selected drop and corruption ratios. Your output should resemble the sample output logs shown in the project specifications.

Policies and submission

Programming language

You can implement your client in C or Python.

Working with a partner

You are required to do this project with your project partner of PA1. You can choose to do it alone but cannot change the partner from PA1 without my permission.

Note on plagiarism

In this class, 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 can copy your code or design from other students in the 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. Any violation of the honor code will be reported to the GMU honor committee, with a failing grade (F) in this course.

*** 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

- The standard late policy applies - The late penalty for this lab will be 15% for each day. Submissions that are late by 3 days or more will not be accepted
- This lab accounts for **17%** of your final grade
- You will submit your solution via Blackboard

What to submit?

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 should not accept any more or less command-line arguments than what is asked for.
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. A PARTNERS.txt file mentioning the name and GMU IDs of two students who worked on the project as a team. Both team members should submit these four pieces separately on Blackboard before the deadline.

Acknowledgments

This programming assignment is based on UC Berkeley's Project 2 from EE 122: Introduction to Communication Networks and UMich's EECS 489 Computer Networks Assignment 3.