Project 2: Transmission Control Protocol

Computer Networks (CS-UH 3012) - Fall 2022

1 Code of Conduct

All assignments are graded, meaning we expect you to adhere to the academic integrity standards of NYU Abu Dhabi. To avoid any confusion regarding this, we will briefly state what is and isn't allowed when working on an assignment.

1. Any document and program code that you submit must be fully written by yourself.
2. You can discuss your work with fellow students, as long as these discussions are restricted to general solution techniques. In other words, these discussions should not be about concrete code you are writing, nor about specific results you wish to submit.
3. When discussing an assignment with others, this should never lead to you possessing the complete or partial solution of others, regardless of whether the solution is in paper or digital form, and independent of who made the solution.
4. You are not allowed to possess solutions by someone from a different year or section, by someone from another university, or code from the Internet, etc.
5. There is never a valid reason to share your code with fellow students.
6. There is no valid reason to publish your code online in any form.
7. Every student is responsible for the work they submit. If there is any doubt during the grading about whether a student created the assignment themselves (e.g. if the solution matches that of others), we reserve the option to let the student explain why this is the case. In case doubts remain, or we decide to directly escalate the issue, the suspected violations will be reported to the academic administration according to the policies of NYU Abu Dhabi. More details can be found at: https://students.nyuad.nyu.edu/academics/registration/academic-policies/academic-integrity/

2 Project Objectives

The task of this project is to implement TCP from the ground up. This project consists of two different tasks, mainly: reliable data transfer and congestion control. Both tasks have their own due date and are dependent on each other.

This project is a group project and you must find exactly one partner to work with. Once you have settled on a partner, you need to inform us by sending an email to the TA and the professor, see the due date under submission details and policy. There will be a 10% penalty if the deadline of the group formation is crossed.

Please read the complete project description carefully before you start so that you know exactly what is being provided and what functionality you are expected to add.

3 Task 1: Simplified TCP Sender/Receiver

The first task is to implement a "Reliable Data Transfer" protocol, following the description of section 3.5.4 from the textbook. The idea here is to build a simplified TCP sender and receiver that is capable of handling packet losses and retransmissions.
The following functionalities must be implemented:

- Sending packets to the network based on a fixed sending window size (e.g. WND of 10 packets)
- Sending acknowledgments back from the receiver and handling what to do when receiving ACKs at the sender.
- A timeout mechanism to deal with packet loss and retransmission

For this task, there is no need to buffer out-of-order packets at the receiver and they can be simply discarded, i.e. no ACKs are sent for out-of-order packets. The out-of-order buffering will be implemented in task 2. However, lost packets must be retransmitted by the sender. For the timeout mechanism, you can assume a fixed timeout value that is appropriate for the emulated network scenario using MahiMahi (see below).

We have provided you with a simple (stop-and-wait) starter-code that consists of the following:

- rdt receiver: this holds the implementation of a simple reliable data transfer protocol (rdt) receiver
- rdt sender: this holds the implementation of a simple reliable data transfer protocol (rdt) sender
- Channel traces for emulating different network conditions

The simple rdt protocol is implemented on top of the UDP transport protocol. During the lab session, the TA showed you how to use the network emulator MahiMahi to test your sender and receiver functionality in an emulated network environment.
Note that sequence and acknowledgement numbers are based on bytes, rather on discrete packet numbers. The sender should terminate after the file has been fully transmitted (including receiving an ACK for the very last packet).

4 Task 2: TCP Congestion Control

The second task is to implement a congestion control protocol (similar to TCP Tahoe) for the sender and receiver of task 1. The implementation of the congestion control will consists of the following features:

- Slow-start
- Congestion avoidance
- Fast retransmit (no fast recovery)

The next subsections detail the requirements of the assignment. This high-level outline roughly mirrors the order in which you should implement functionality. For further details, please refer to the lecture slides or the textbook.

4.1 Reliability and Sliding Window

The sender and the receiver have to maintain a sliding window, as shown in Figure 1.

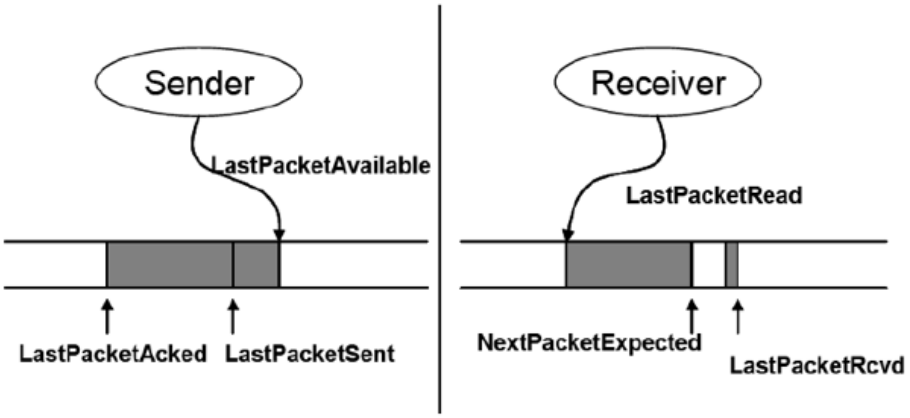

Figure 1: Sliding Window

The sender slides the window forward when it receives an ACK for a packet with the lowest sequence number in the sliding window. There is a sequence number associated with each packet and the following constraints are valid for the sender:

1. LastPacketAcked <= LastPacketSent
2. LastPacketSent <= LastPacketAvailable
3. LastPacketSent - LastPacketAcked <= WindowSize
4. Packets between LastPacketAcked and LastPacketAvailable must be “buffered”. You can either implement this by buffering the packets or by being able to regenerate them from the data file.

When the sender sends a data packet it starts a timer, if not already running. The timeout timer should be set based on the RTT estimator discussed in the lecture. It then waits for a certain amount of time to get the acknowledgement for the packet. Whenever the receiver receives a packet, it sends an ACK for NextPacketExpected. Upon receiving an out-of-order packet, the receiver must buffer the packet and send a duplicate ACK.
For example, upon receiving a packet with sequence number = 100, the reply would be “ACK 101”, but only if all packets with sequence numbers less than 100 have already been received. These ACKs are called cumulative ACKs. The sender has two ways to know if the packets it sent did not reach the receiver: either a time-out occurred, or the sender received “duplicate ACKs”.
If the sender sent a packet and did not receive an ACK for it before the timer expired, it retransmits the packet. If the sender sent a packet and received duplicate ACKs, it knows that the next expected packet (at least) was lost. To avoid confusion from reordering, a sender counts a packet lost only after 3 duplicate ACKs in a row.

4.2 Congestion Control

Broadly speaking, the idea of TCP's congestion control is to determine how much capacity is available in the network, so it knows how many packets it can safely have “in-flight” at the same time. Once the sender has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network, and it is therefore safe to insert a new packet into the network without adding to the level of congestion.

By using ACKs to pace the transmission of packets, TCP is said to be “self-clocking”. TCP's Congestion Control mechanism consists of the algorithms of Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery. You can read more about these mechanisms in our textbook Section 3.7. In the first part of the project, your window size was fixed to 10 packets. The task of this second part is to dynamically determine the ideal congestion window size (CWND).

Slow-start: When a new connection is established with a host on another network, the CWND is initialized to one packet. Each time an ACK is received, CWND is increased by one packet. The sender keeps increasing CWND until the first packet loss is detected or until CWND reaches the value ssthresh (Slow-start threshold), after which it enters Congestion Avoidance mode (see below). For a new connection, the ssthresh is set to a very big value, we will use 64 packets. If a packet is lost in the slow-start phase, the sender sets ssthresh to max(CWND/2, 2) in case the client later returns to Slow-start again.

Congestion Avoidance slowly increases CWND until a packet loss occurs. The increase of CWND should be at most one packet per round-trip time (regardless how many ACKs are received in that RTT). That is, when the sender receives an ACK, it usually increases CWND by a fraction equal to 1/CWND. You may notice here that you need to use a float variable for the CWND, however when you send data you are always going to take the floor of the CWND. As soon as the entire window is acknowledged, only then these fractions would sum to a 1.0 and as a result the CWND would then have increased by 1. This is in contrast to Slow-start where CWND is incremented for each ACK. Recall that when the sender receives 3 duplicate ACK packets, you should assume that the packet with sequence number == acknowledgment number was lost, even if a timeout has not occurred yet. This process is called Fast Retransmit.

Fast Retransmit: After a Fast Retransmit, the ssthresh is set to max(CWND/2, 2). CWND is then set to 1 and the Slow-start process starts again. There is NO need to implement the fast recovery mechanism.

4.3 Graphing CWND

Your sender implementation must generate a simple output file (named CWND.csv) that shows how the CWND varies over time. This will help you to debug and test your code, and it will also help us to grade your submission. The output format is simple, it should record the time when the window changed and the current CWND. You should also amend the plotting script (plot.py) to plot the CWND and how it evolves over time.

5 Further Implementation Details

Please note the following implementation details for both projects.

- There is no need to implement the handshake procedures of TCP, i.e. connection establishment and termination
- The setup of sender and receiver in this project does not reflect a real world scenario. In this setup, the receiver must be started first and the sender will send the file to the receiver as soon as it is started. In a real world scenario, the sender would be the server and the receiver a client. Furthermore, the client requests a file from the server and listens
- Make sure you either give the output file at the receiver a different name or run the sender and receiver code in two different folders to avoid overwriting the sending file with the receiving file.
- The sliding window should have a maximum size of 32 bits and wraps around.
- There is no need to implement flow control

6 Grading

Description	**Score (/20)**
Task 1: Extending the sender to send 10 packets	2
Task 1: Properly sending and handling ACKs	1
Task 1: Retransmissions of lost packets	1
Task 1: Properly receiving the exact file on the receiver (no errors)	1
Task 2: Slow-start implementation	3
Task 2: Congestion avoidance implementation	2
Task 2: Fast retransmit implementation	2
Task 2: Correct throughput plots (protocol implementation saturates the link) TODO	2
Task 2: Correct CWND recording and plotting	2
Coding style and usage of meaningful comments	4

7 Submission Details and Policy

Submission Deadlines:

1. Group formation due date: November 11, 2022 (-10% penalty if not met)
2. Task 1 due date: November 18, 2022 (30% of grade)
3. Task 2 due date: December 8, 2022 (70% of grade)

Submission Format and System: You can directly submit your files as a zip file on Brightspace (https://brightspace.nyu.edu/). Due to technical limitations, submissions via email are not accepted.

Late Submissions: Late submissions will be penalized by 10% per 24 hours, with a maximum of 3 days late. In case of a late submission, please upload your zip file to Brightspace and inform the TA and the professor.