STCP Part 2 - Data Transfer

In this project, you will implement data transfer protocol in the transport layer under write() and read() system calls. You will transmit packets with payload, and implement a simple congestion control method. All implementations will be based on the assumption of a reliable environment, so you don't have to think about packet loss, reordering, retransmission, and timeout. At the end of this project, your sample client should be able to download and write data from the server-side.

Important Notice
The deadline for this project is **May 11th, Tuesday, 11:55 PM**. The submission link will be automatically closed after the deadline. Note that any kinds of late submissions are not allowed.

Submission link: https://forms.gle/tvQd3GTFDNCHU9Ai6

What is STCP?
In the first two assignments, you learned about the socket interface and how it is used by an application. By now, you're pretty much an expert in how to use the socket interface over a reliable transport layer, so now seems like a good time to implement your own socket layer and reliable transport layer! That's what you'll be doing in this assignment.

You'll get to learn how the socket interface is implemented by the kernel and how a reliable transport protocol like TCP runs on top of an unreliable delivery mechanism. We're going to call your socket layer MYSOCK and it will contain all the features and calls that you used in Assignment #1. Your socket layer will implement a transport layer that we'll call **STCP(Simple TCP)**, which is in essence a stripped-down version of TCP.

STCP is compatible with TCP, but it is not the same as TCP. It is similar to early versions of TCP, which did not implement flow control or optimizations such as selective ACKs or fast retransmit. To help you get started, we're providing you with a skeleton system in which you will implement the MYSOCK and STCP layers. In fact, the MYSOCK layer has already been implemented for you; you get to add the functionality needed for the transport layer. The skeleton consists of a network layer, a bogus transport layer, and also a dummy client and server application to help you debug your socket and transport layer. This assignment is implementing the socket/transport layer so that the supplied programs(a dummy client and a dummy server) work in reliable mode using your implementation.

STCP is a full-duplex, connection-oriented transport layer that guarantees in-order delivery. Full duplex means that data flows in both directions over the same connection. Guaranteed delivery means that your protocol ensures that, short of catastrophic network failure, data sent by one host will be delivered to its peer in the correct order. Connection-oriented means that the

packets you send to the peer are in the context of some pre-existing state maintained by the transport layer on each host.

STCP treats application data as a stream. This means that no artificial boundaries are imposed on the data by the transport layer. If a host calls mywrite() twice with 256 bytes each time, and then the peer calls myread() with a buffer of 512 bytes, it will receive all 512 bytes of available data, not just the first 256 bytes. It is STCP's job to break up the data into packets and reassemble the data on the other side.

Background

In part 3-1, you have implemented a 3-way and 4-way handshaking protocol which runs under connect(), close() application system calls. In this part, you will implement data transfer over a **reliable** underlying channel that runs under write() and read() system calls. After receiving the application calls, your implementation should be able to send/receive data which is delivered through the payload in the packet. The sequence/ack number, length, and window field of the packet should follow the protocol described below.

1) Data transfer

After 3-way handshaking, end hosts can send and receive data as they desire. This is done by calling two system calls, write() and read(). We will first look at the sender and see what happens under the system call.

First, observe what happens in the transport layer with write() system call. Whenever there is data to be sent in the internal sender buffer, new data packets are created with an appropriate sequence number. In STCP, the application stacks data to be sent with write() call, and you can get the data that is requested from the application using `stcp_app_recv()` function.

In typical TCP, the packets are sent to the remote host if there is enough space in the receiver buffer. (Flow control) Otherwise, packets are kept on the sender side until the sender receives the ACK packet from the receiver, and the remaining receiver window size written in the ACK packet turns out to be large enough.

Now let’s observe the receiver side, with read() system call. As the socket receives data packets from the sender, it should (first verify checksum and) extract data from the packet and store it to the internal receiver buffer. This internal receiver buffer stores the data until there’s a request from the upper layer, a read() system call. In STCP, you don’t have to implement the logic related to the receiver buffer, you can just call `stcp_app_send()` function instead. The application will get the values you stacked with the stcp_app_send() function with later read() calls.

The receiver should send ACK packets to the sender to notify the receipt of data packets. Since we are building a TCP-like system, the algorithm of choosing acknowledgment numbers should follow the TCP’s rule. The acknowledgment number represents the “next expected sequence

number". Please note that this is different from the GBN or SR algorithm we learned in the lecture, as their acknowledgment number represents "last successfully received sequence number". For example, if the sender sent a packet that contains 70 bytes of data with sequence number 100, the correct acknowledgment number to this packet is 170. The next packet will have sequence number 170.

The receiver should also implement flow control. The purpose of flow control is to prevent the overflow of the receiver buffer by telling the remaining receiver buffer size to the sender in ACK packet header. The sender should not send data packets that will cause the receiver's buffer overflow. Actually in STCP, the receiver window is always fixed to a constant value(3072). It will make your implementation much easier.

2) Congestion control
You will also implement a congestion control algorithm using slow start and congestion avoidance.

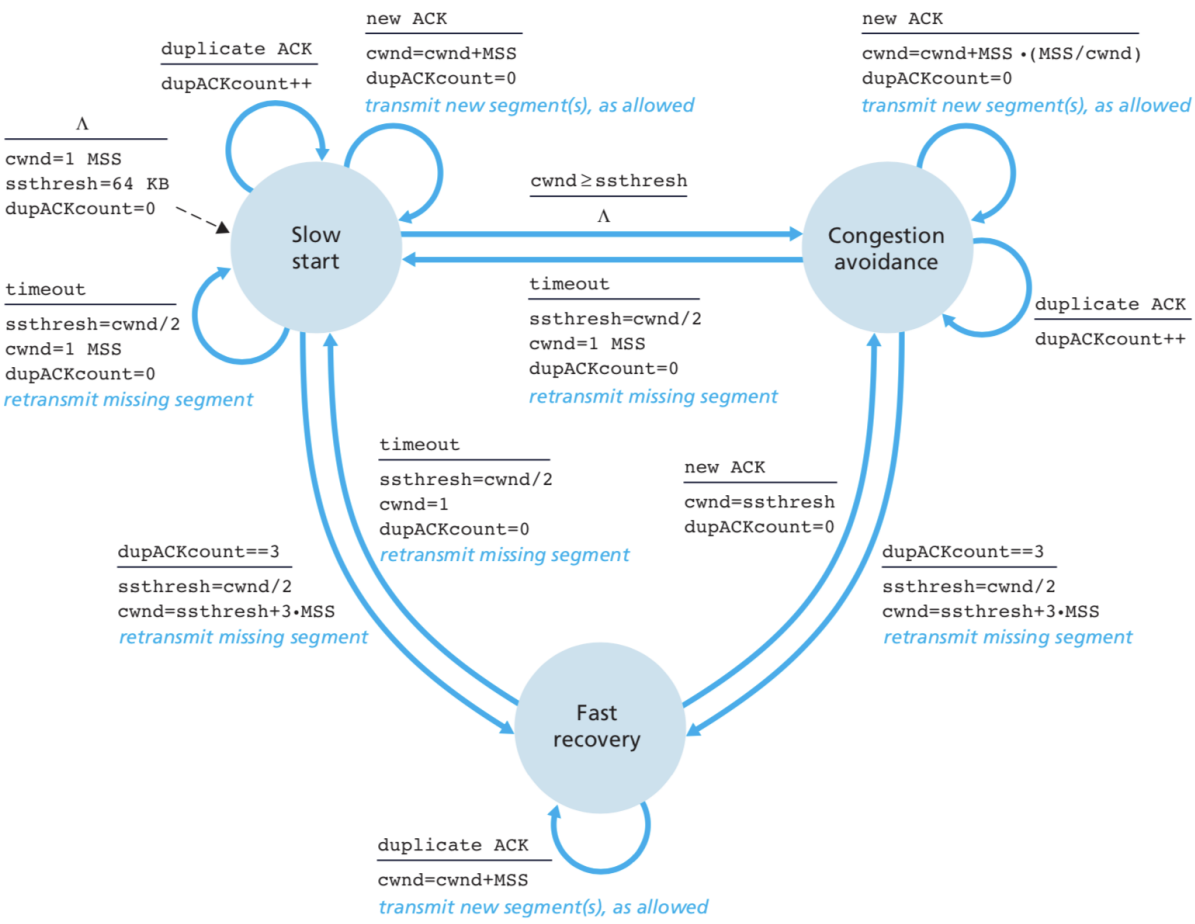
Congestion control is done by the **sender**. There's a similar control concept called 'flow control', but flow control is done by the **receiver**. Note the difference between the two concepts. Even though there is enough space in the receiver's window size advertised by flow control, the sender may send less data because of the congestion control. More specifically, the congestion window (cwnd) is a sender-side limit, while the receiver's advertised window (rwnd) is a receiver-side limit. The effective "actual" window size of the connection is the minimum of cwnd and rwnd. In STCP, you will consider the receiver window as a fixed size of 3072.

Now let's briefly understand how congestion control changes window size. In this project, we will assume the congestion control algorithm of TCP Reno. Congestion window size (cwnd) starts from 1 MSS(Maximum Segment Size), and it's in a **slow start** state. In this state, whenever a transmission is successful, cwnd is increased by 1 MSS. This results cwnd to be doubled in every transmission round. When cwnd hits the slow start threshold (ssthresh), it changes the state from slow start to **congestion avoidance**.

In the congestion avoidance state, cwnd is increased by 1 MSS in every transmission round. The implementation is tricky and may differ by versions, and one way to implement this is to add MSS*(MSS/cwnd) for each ACK received. This results cwnd to be additively increased.

There is a concept named fast recovery when a triple duplicate ack is detected in an unreliable environment, but we will not cover this in this assignment.

The diagram below summarizes the congestion control algorithm. You can ignore the parts related to duplicated ACK and timeout in this assignment (such situations only occur under the unreliable channel).



What should I do?
You are going to implement your code on the given skeleton code.
Please download the skeleton in the following link:
https://drive.google.com/file/d/1yO1grSSji4aoGAyoRzxO9H4_9Hf0B0rO/view?usp=sharing

The only file you will modify in this project is `transport.c`. You are going to implement the TCP functionalities for data transfer in the following function.

```
control_loop(mysocket_t sd, sontext_t *ctx)
```

You have implemented TCP protocol related to closing here in the previous assignment. In this assignment, this loop will additionally get APP_DATA events caused by the write() and read() system calls, and NETWORK_DATA which involves payload which is sent from the peer. If you have not fully understood the concept of receiving events or sending/receiving TCP segments with STCP API, please refer to the previous assignment document.

If the event returns APP_DATA, it means that there is data to be sent to the peer. You can get the data buffer using `stcp_app_recv(mysocket_t sd, void *dst, size_t`

`max_len)` function. You have to send the requested buffer to the peer by attaching it on the payload position which is behind the tcp header. As in the previous assignment, by using `stcp_network_send()` function you can send data to the peer. It takes the whole TCP segment including both header and payload, so you have to set the third parameter as the size of (STCP header + payload size). Please note that the **maximum payload size which can be transmitted at once is 1 STCP_MSS(536)**. Don't send the whole data that the application writes in one packet.

One more thing to note here is that the sender buffer size cannot exceed the sender window size. The sender window size is the minimum value between the receiver window(fixed to 3072) and the congestion window. And the sender buffer size indicates the unacked size of the buffer you sent to the peer. You have to reduce the sender buffer size by the size of the buffer you got from the application with `stcp_app_recv()`. You are not allowed to accept data from the `stcp_app_recv()` call exceeding the remaining sender buffer size. You can control the amount of size to accept from the application by setting the third parameter of `stcp_app_recv()`. The occupied sender buffer size is released when you get acknowledgment from the peer with `stcp_network_recv()`. You have to record the previously acknowledged number from the peer, and increase the remaining sender buffer size by [currently received ack num - previous ack num] after getting the corresponding ack.

Now let's look at the receiver side implementation. When the event is NETWORK_DATA, you can receive the segment using `stcp_network_recv()`. If the return value of `stcp_network_recv()` is larger than the TCP header size, it means that it contains payload in it. In this case, you have to acknowledge the peer with the corresponding acknowledgment number using `stcp_network_send()`. To make the application's read() call available, the received payload data should be sent to the application buffer. You can do it by calling `stcp_app_send(mysocket_t sd, const void *src, size_t src_len)` function. In the second parameter it requires the data to be delivered(it is not the packet, but the payload), and in the third parameter, it requires the size of the payload.

Let's go back to the sender's side. If the data sender gets a packet with an ack number that is higher than the previously accepted ack number, we can interpret it as that the unacked sent packet successfully transmitted to the receiver. Therefore it is available to clean the sender buffer as [acknowledgment from the currently arrived packet - previously accepted acknowledged number]. Moreover, according to the slow start & congestion control algorithm, you have to improve the congestion window size as the sender successfully got the returning acknowledgment for the sent data.

Regarding slow start, you have to set the **initial congestion window size to 1 STCP_MSS**. The size of **STCP_MSS is defined as 536** in `transport.h`. You should not change this value

to get points for this assignment. Whenever you receive ack corresponding to the data you sent, you have to increase the congestion window size according to the logic of slow start and congestion avoidance. The threshold where **you should stop slow start is 4*STCP_MSS**. Please start congestion avoidance when your window size reaches that threshold.

The sender window size is decided as the minimum value between the congestion window and the advertised receiver window. It's flow control, but in this assignment, you don't have to care about the receiver window. Just **fix the receiver window size as 3072**, and set the sender window as the minimum between the congestion window and 3072. Since there would be no packet loss and retransmission in the reliable environment, your sender window size will start from 1 MSS, increase with slow start until the threshold, congestion avoidance after the threshold, then will be fixed to constant value after reaching 3072.

After increasing the sender window size using the congestion control method, you would have a larger sender buffer size that can cover more than one TCP segment at once. It means that you don't have to send data and wait for the ack in the alternative order, but you can send multiple packets at once. This would help in reducing the transfer time. In this project, **you are required to iteratively send multiple packets while you have available space in the sender buffer**. Let me give you an example for better understanding.

Let's assume that the sender's congestion window size is 2144, and it has 536 unacked data. Then your sender window size would be min(2144, 3072) - 536 = 1608. You can make 3 packets to be sent cumulatively since 1608 is 3*STCP_MSS. In this case, your packet transmit log should be as:

…

SEND: ACK	seq: x	ack: y	len: 556
SEND: ACK	seq: x+536	ack: y	len: 556
SEND: ACK	seq: x+1072	ack: y	len: 556
RECV: ACK	seq:y	ack:x+536	len: 20
RECV: ACK	seq:y	ack:x+1072	len: 20
RECV: ACK	seq:y	ack:x+1608	len: 20

…

Since it is available to have unacked data of size 1608, the sender sends 3 data packets cumulatively. It does not receive corresponding acknowledgment right after sending one packet. The number 556 in the length comes from 20(header length) + 536(MSS).

There are a few more things that you need to know.

First, congestion control in STCP is only done in the ESTABLISHED state. You should not change the window size before handshaking is done. The initial packet including data should have the value of 1 MSS in the length field of the packet header.
(https://tools.ietf.org/html/rfc5681)

Second, since STCP is a reliable and in-order guaranteeing protocol, you don't have to worry about packet loss or reordering.

Third, you have to follow network byte ordering when you manage the fields in the TCP header. Refer to the following document if you are not familiar with endianness.
https://en.wikipedia.org/wiki/Endianness

Fourth, all TCP headers sent after the established connection should have an ACK flag in the th_flag field.

How do I test my implementation?
We will check the sent and received packet information captured from `stcp_network_recv` and `stcp_network_send` functions for testing.

For checking the sent/received packet log, you can change the LOG_PACKET value to TRUE which is defined in the 18th line of `stcp_api.c`. Then whenever you execute a connection using stcp_api, a pair of files will be logged in your working directory. They are named "pcap_from_[my port number]_to_[connected port number]", and involve all sent/received packets' flag/sequence number/ack number/length/window size in the corresponding order. We will use the same file for grading. Note that the file writes **appends** the log into the existing file with the same name. If the same sender/receiver port number is assigned as the previous log, the new log is not overwritten but appended to the existing log file.

In this project we will also use server.c and client.c. You can build a test server and test client by typing `make all` command in the project directory. Please execute `./server` to load the passive server. It will bind, listen and accept the new connection coming in. The bound host and port number is printed as standard output.

```
(any sh)    $ make all
(sh 1)      $ ./server
            Server's address is 127.0.0.1:33451
```

In another shell, you can execute the client binary as before. But this time, you have an additional command option for the client binary. By adding "-f [filename]" in the command,

```
(sh 2)      $ ./client 127.0.0.1:33451 -f test.txt
```

you can request the server to send the content of the file that matches with [filename]. If the file is transmitted successfully from the server, the received file will be stored in the directory where the client is running. The stored file name is `rcvd`. You can check whether the received file has the same contents as the requested file by typing `diff rcvd [original filename]`. You

can get a part of the point if your transmitted file is the same as the requested file. But remind that you also have to implement a slow start in this project. It should be checked by analyzing the packet trace.

For the grading tests, we are going to use the pair of [solution - your implementation], so your implementation has to work for both client and server. Your implementation should be compatible with the solution. We won't give you points if it does not work even though your implementation works with your own server and client.

We will run your submission in the haedong lounge server environment.
Note that any problem that is caused by using different environments is the student's responsibility. We will not accept any regrade request such as "It runs on my computer".

What are the grading criteria?
This project is worth **6%** of your total grade. The grading will be done as below.

- Your code should be built with the default Makefile. If not, we cannot give you points.
- For all test cases, the order of SYN/SYNACK/ACK/FINACK flags, the corresponding sequence/ack numbers, and the length should be correct. If any of them is wrong, we will score the case as 0.
- The cases testing data transfer are performed after the "active open" case in the previous assignment. If you have a problem in an active open case, you would get 0 points for the later close cases.
- (40%) Code: Sender side implementation
- (40%) Code: Receiver side implementation
- (20%) Code: Congestion control implementation
- (0%) Report: The report will not be explicitly graded to the numerical value, but we will use it when there are any issues with grading.
- Any violations (wrong Makefile script, wrong file name, report is not PDF, etc.) will result in a penalty.

Where can I ask questions?
We encourage students to actively participate in the course. Please share your experiences and questions in the course Facebook group. However, if you think your question is not appropriate for sharing for some reason (such as it contains private information), send it to ee323@nmsl.kaist.ac.kr. However, TAs may share your question on Facebook.

How to submit?
You need to submit the following items. Note that in this project, we don't provide any skeleton files.

- transport.c
- report.pdf

Compress the above items into one zip file. Name it as **{studentID}_{name(in English)}_project3_2.zip** (e.g. 20211234_HyungJunYoon_project3_2.zip)
and submit it to the link below.

Submission link: https://forms.gle/tvQd3GTFDNCHU9Ai6

Q&A

Q. How much implementation is required in this project?
A. The following is a diffstat of the implementation compared to the baseline. (Implementation of the whole STCP)
transport.c | 432 ++--

Q. Do we have to care about the multiple connection case? In other words, is there any case testing multiple clients are connected to one server?
A. No, in this project we will only test the one-to-one connection. You don't have to care about it.

Q. How can I set TCP header flags?
A. In transport.h, they are defined as below. You can use them.

```
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04    /* you don't have to handle this */
#define TH_PUSH 0x08    /* ...or this */
#define TH_ACK  0x10
#define TH_URG  0x20    /* ...or this */
```

Q. Does STCP_MSS include TCP header size?
No, the MSS that we described in this document means only the payload size.

Q. What should I set for th_win when I send packets?
In STCP, we assume that the receiver window is always fixed as 3072. We won't grade the window size sent through the packet. You can put any number for it. You can just fix the th_win to 3072.

Q. When I checked the packet log, there was a data transmission from the client which is requesting the data to be transmitted. Is it the right behavior?
A. In STCP client/server implementation, there are additional data transmissions that are initiated from the client-side. It is for notifying the server what kind of data the client is requesting, such as the filename. Just make sure that the downloaded file of the client is the same as the original file, and the sender window size increases.

Q. Could you provide solution binaries for testing?

A. Sorry, the solution binaries we have do not exist in usable form to be deployed for students. We recommend you fully understand the TCP finite state machine you learned in class and apply the structure to your implementation. If you followed it well, your implementation would be compatible with our solution.