# Project 3: STCP - Reliable Transport Layer

In this project, we implement a simple TCP (STCP) stack with the provided network layer code. The stack provides a subset of modern TCP stack features - it implements connection setup/teardown and data transfer between two endpoints. You don't have to implement congestion control (including fast retransmit/recovery) nor flow control.

In terms of reliable data transfer, you have two choices here. (a) You can assume that the underlying network layer is *reliable* and build the STCP stack on top of it or (b) you can assume that the underlying network layer is *unreliable* and build the stack accordingly – packet loss and out of order packet delivery can happen, so you need to implement receive-side buffering (flow reassembly), retransmission timeout and packet retransmission for it. Obviously, (a) is much simpler and easier to implement. While (b) is more complex, it reflects real networks and it is much more fun. If you implement (b), you get **50%** of the original score as extra credit.

We provide the STCP source code that consists of the STCP stack and applications (`client.c` and `server.c`) that use the stack. Your job is to complete the STCP stack so that applications using the STCP stack (`client` and `server`) work as intended. The provided code compiles but it won't work until you complete the STCP stack. In order to complete the stack, you need to modify `transport.c`. Then, you can use `client` and `server` to test your STCP stack. These applications use a reliable network by default, but you can use `-U` option for both programs to simulate an unreliable network that randomly drops packets in the middle. So, if you chose (a), you need to test the programs without the `-U` option. If you chose (b), you need to test them with the `-U` option. Properly implemented, your stack for (b) should just work well without the -U option as well.

You are given STCP API functions (in `stcp_api.c`) with which you implement the STCP stack. With the API functions, you can send and receive TCP packets (`stcp_network_send()`, `stcp_network_recv()`), receive the data from the application (`stcp_app_recv()`), and deliver flow-reassembled data to the application (`stcp_app_send()`).

## STCP Packet Format

Using the network layer functions, you can send and receive a TCP packet at a time. The packet consists of a TCP header (see `STCPHeader` type definition in `transport.h`, and shown below) and payload whose maximum segment size (MSS) is **536**.

```
typedef uint32_t tcp_seq;

typedef struct tcphdr {
        uint16_t th_sport;              /* source port */
        uint16_t th_dport;              /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgment number
*/
#ifdef _BIT_FIELDS_LTOH
        u_int   th_x2:4,                /* (unused) */
                th_off:4;               /* data offset */
#else
        u_int   th_off:4,               /* data offset */
                th_x2:4;                /* (unused) */
#endif
        uint8_t th_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
        uint16_t th_win;                 /* window */
        uint16_t th_sum;                 /* checksum */
        uint16_t th_urp;                 /* urgent pointer */
        /* options follow */
}__attribute__ ((packed)) STCPHeader;
```

In order to send a STCP packet, you need to fill out only five fields in the TCP header shown below - other fields (`th_sport`, `th_dport`, etc.) are automatically filled out by the network layer, so you don't need to worry about them. Similarly, you don't need to handle all legal flags. That is, `TH_RST`, `TH_PUSH`, and `TH_URG` are ignored by STCP.

| Field | Type | Description |
|---|---|---|
| th_seq | tcp_seq | Sequence number associated with this packet. |
| th_ack | tcp_seq | If this is an ACK packet, the sequence number being acknowledged by this packet. This may be included in any packet. |
| th_off | 4 bits | The offset at which data begins in the packet, in multiples of 32-bit words. (The TCP header may be padded, so as to always be some multiple of 32-bit words long). If there are no options in the header, this is equal to 5 (i.e. data begins twenty bytes into the packet). |
| th_flags | uint8_t | Zero or more of the flags (TH_FIN, TH_SYN, etc.), or'ed together. |

| | | |
|---|---|---|
| th_win | uint16_t | Advertised receiver window in bytes, i.e. the amount of outstanding data the host sending the packet is willing to accept. |

## Sequence Numbers

- Sequence numbers sequentially number the SYN flag, FIN flag, and bytes of data, starting with a randomly chosen sequence number. For the purpose of this assignment, however, please use `generate_initial_seq_num(context_t *ctx)` to set the `initial_sequence_num` value to 1 (please don't modify this function), so you can use the set value as the sequence number for sending your first packet. The first ack number is meaningless, so it can be any randomized number.
- The transport layer manages two streams for each connection: incoming and outgoing data. The sequence numbers of these streams are independent of each other.
- Both SYN and FIN indicators are associated with one byte of the sequence space.
- The sequence number should always be set in every packet. If the packet is a pure ACK packet (i.e., no data, and the SYN/FIN flags are unset), the sequence number should be set to the next unsent sequence number.

## Data Packets

- The maximum payload size is 536 bytes.
- The `th_seq` field in the packet header contains the sequence number of the first byte in the payload.
- Data packets are sent as soon as data is available from the application. STCP performs no optimizations for grouping small amounts of data on the first transmission (no need to support Nagle's algorithm).

## ACK Packets

- Data is acknowledged by setting the ACK bit in the flags field in the packet header. If this bit is set, then the `th_ack` field contains the sequence number of the next byte of data the receiver expects (i.e. one past the last byte of data already received). This is true no matter what data is being acknowledged. The `th_seq` field should be set to the sequence number that would be associated with the next byte of data sent to the peer. The `th_ack` field should be set whenever possible (this isn't required by STCP, but is done for compatibility with standard TCP).
- Data may accompany an acknowledgment. STCP is not required to generate such packets (i.e. if you have outstanding data and acknowledgments to send, you may send these separately), but it is required to be capable of handling such packets.
- Acknowledgments are not delayed in STCP (called delayed ACK) as they might be in TCP. An acknowledgment should be sent as soon as data is received.
- If a packet is received that contains duplicate data, a new acknowledgment should be sent.

- Acknowledgments are only sent for the last contiguous received piece of data (called cumulative ACK). For example, assume the last acknowledgment sent was 10. A packet containing data with sequence number 15 of length 5 is received. An acknowledgment for this packet cannot be sent because of the gap at sequence number 10 of length 5. Instead, an ACK should be sent for 10 again. However, once all the data in this range has been received, the correct acknowledgment to send is for sequence number 20.

## Sliding Windows

There are two windows that you will have to take care of: the receiver and sender windows.

The receiver window is the range of sequence numbers which the receiver is willing to accept at any given instant. The window ensures that the transmitter does not send more data than the receiver can handle.

Like TCP, STCP uses a sliding window protocol. The transmitter sends data with a given sequence number as and when data has been acknowledged. The size of the sender window indicates the maximum amount of data that can be unacknowledged at any instant. Its size is equal to the other side's receiver window.

- The local receiver window has a fixed size of 3072 bytes. For the purpose of the assignment, you can assume that the flow-reassembled data is consumed by the application right away when you deliver the data by `stcp_app_recv()`, you don't need to worry about flow control. In reality, however, if the application does not consume the data, the receive window has to be adjusted to reflect the amount of available receive buffer.
- The sender window is the minimum of the other side's advertised receiver window, and the congestion window.
- The congestion window has a fixed size of 3072 bytes. This means that you don't have to implement congestion control for the assignment.
- As stated in the acknowledgment section, data which is received with sequence numbers lower than the start of the receiver window generate an appropriate acknowledgment, but the data is discarded.
- Received data which has a sequence number higher than that of the last byte of the receiver window is discarded and remains unacknowledged.
- Note that received data may cross either end of the current receiver window; in this case, the data is split into two parts and each part is processed appropriately.
- Do not send any data outside of your sender window.
- The first byte of all windows is always the last acknowledged byte of data. For example, for the receiver window, if the last acknowledgment was sequence number 8192, then the receiver is willing to accept data with sequence numbers of 8192 through 11263 (=8192+3072-1), inclusive.

## TCP Options

The following rules apply to handling TCP options:

- STCP does not generate options in the packets it sends.
- STCP ignores options in any packets sent by a peer. However, it must be capable of correctly handling packets that include options.

## Retransmissions

It is an ugly fact of networking life that packets are lost. STCP detects this when no acknowledgment is received within a timeout period. The rules for timeouts are:

- Whenever a SYN, data, or FIN segment (i.e. an STCP packet containing anything more than just an acknowledgment) is transmitted, a timeout is scheduled some milliseconds from the time of transmission. You may implement the timeout as you desire--e.g. a fixed timeout, the Karn-Partridge algorithm, etc. Whatever works for you. Note that a fixed RTO may not be sufficient--we strongly suggest that you implement RTT estimation in some form. (This is only a few lines of code).
- If the data in a segment is acknowledged before the corresponding timer expires, the timer is canceled and the segment can be safely discarded.
- If the timeout for a segment is reached, the segment is examined. If it has been sent a total of 6 times (once from the original send plus 5 retransmissions), then the network is assumed to have failed and the network terminated. Otherwise the segment is retransmitted and the timeout is reset again.
- STCP, like TCP, acts like a Go-back-N protocol (where the receiver buffers out-of-order packets). This means that if the transport layer decides that its peer has not received the segment with sequence number n, then it will retransmit all data starting at n, not just that segment.
- Your receiver must buffer any data it receives, even if it arrives out of order. (This isn't strictly required by TCP, but almost any "real" TCP implementation would perform this optimization, and you must too).

## Connection Setup

Normal TCP connection setup is always initiated by the active end. Connection initiation uses a three-way `SYN` handshake exactly like TCP, and is used to exchange information about the initial sequence numbers. The order of operations for initiation is as follows:

- The requesting active end sends a `SYN` packet to the other end with the appropriate seq number (seqx) in the header. The active end then sits waiting for a `SYN_ACK`.
- The passive end sends a `SYN_ACK` with its own sequence number (seqy) and acks with the number (seqx+1). The passive end waits for an ACK.
- The active end records seqy so that it will expect the next byte from the passive end to start at (seqy+1). It ACKs the passive end's SYN request, and changes its state to the established state.
- Real TCP uses a random number for the initial sequence number, but for the purpose of this assignment, use `generate_initial_seq_num(context_t *ctx)` to set the initial sequence number to 1.
- You don’t need to implement simultaneous connection setup where two active ends send `SYN` at the same time.

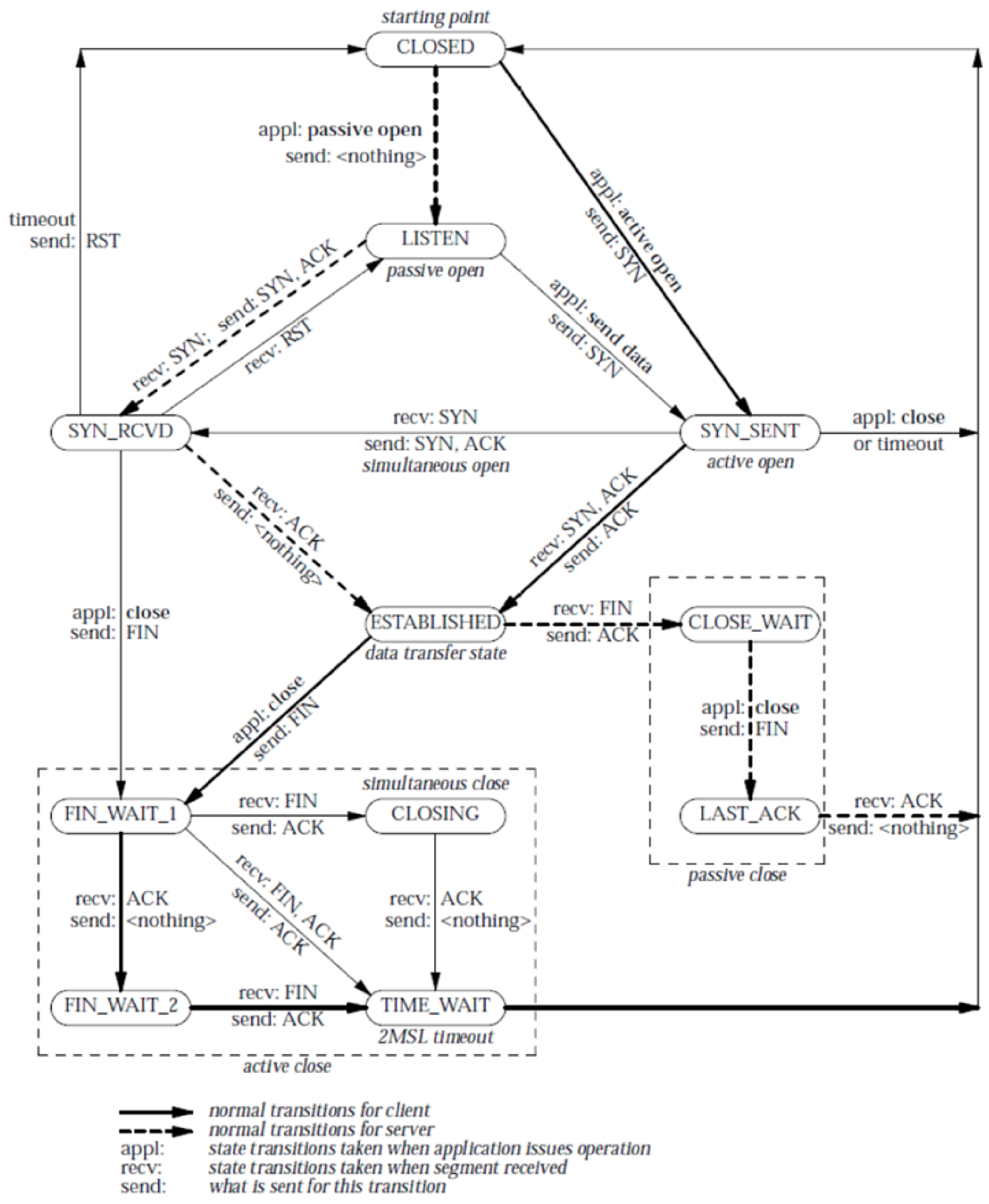
For more details, be sure to read RFC 793.

## Connection Teardown

As in TCP, connection teardown is a four-way handshake between the two peers in a connection. The order of closing is independent of the connection setup order. Each side indicates to the other when it has finished sending data. This is done as follows:

- When one of the peers has finished sending data, it transmits a `FIN` segment to the other side. At this point, no more data will be sent from that peer (other than possibly retransmissions, including the `FIN`). The `FIN` flag may be set in the last data segment transmitted by the peer. (You are not responsible for generating such packets, but your receiver must be capable of handling them). The usual rules for retransmission and sequence numbers apply for the `FIN` segment.
- The peer waits for an `ACK` of its `FIN` segment, as usual. If both sides have sent `FINs` and received acknowledgements for them, the connection is closed once the `FIN` is acknowledged. Otherwise, the peer continues receiving data until the other side also initiates a connection close request. If no `ACK` for the `FIN` is ever received, as per the rules for retransmission, it terminates its end of the connection anyway.

RFC 793 includes more details on connection termination; see in particular the state diagram shown below. Note that you are not required to support `TIME_WAIT`.



==================================================================

## What should I do?

The only source code file you will modify in this project is `transport.c`. We suggest that you implement the TCP connection setup and teardown first. The following function,

```
void transport_init(mysocket_t sd, bool_t is_active)
```

initializes the transport layer, which runs in its own thread, one thread per connection. This function should not return until the connection ends. `sd` is the 'mysocket descriptor' (similar to a socket descriptor in normal TCP API) associated with this end of the connection. `is_active` is TRUE if this function runs as a client that initiates the connection. So you have to implement “active open” if it is TRUE, and “passive open” in the other case. In this function, you will first implement 3-way handshaking. After making sure that the connection is established, call `control_loop()`. You need to use SCTP network layer API functions to implement the logic.

## Network layer API and other functions

Here is a list of network layer API functions and other functions useful for the assignment.:

```
ssize_t stcp_network_send(mysocket_t sd, const void *src, size_t
src_len, ...)
```

- sends a TCP packet (`src`) whose length is `src_len`. `src` should point to the start of the STCP header. If `src` points to the entire SCTP header and payload, `src_len` should be the byte count of the buffer. Note that the function takes a variable number of parameters (...). So in case `src` points to only STCP header, and the payload is in another buffer, you can provide the header and the payload separately. The last argument must be `NULL`.

  For example, (`uint8_t` is type defined as `typedef unsigned char uint8_t`)
  ```
  #define MSS 536
  uint8_t packet[sizeof(STCPHeader)+MSS]
  // fill out packet header
  …
  // fill out packet payload
  …
  stcp_network_send(sd, packet, sizeof(packet), NULL);
  ```

  or you can provide the header and payload separately as below
  ```
  #define MSS 536
  uint8_t header[sizeof(STCPHeader)];
  uint8_t payload[MSS];
  // fill out packet header
  …
  // fill out packet payload
  …
  stcp_network_send(sd, header, sizeof(header), payload,
  sizeof(payload), NULL);
  ```

  In case it’s a control packet (pure SYN, FIN, ACK packets, etc.), you can send only header (or payload of size zero)
  ```
  …
  stcp_network_send(sd, header, sizeof(header), NULL);
  ```

`ssize_t stcp_network_recv(mysocket_t sd, void *dst, size_t max_len)`

- receives the packet at the buffer pointed to by `dst` whose maximum length is `max_len`. It returns the number of bytes that have been received. The received packet consists of an STCP header (and possible options) and payload if it contains data. Note that this function would block if the stack does not have any received packets until new packet arrival, so you need to be careful to call it especially when the network is set to be unreliable (as your program would get stuck in case of packet loss).

```
unsigned int stcp_wait_for_event(mysocket_t sd,
                                 unsigned int flags,
                                 const struct timespec *abstime)
```

- blocks until any events that are being waited for or timeout (`abstime`) occurs. The events that you wait for can be specified by OR'ing `APP_DATA` (wait for any data arrival from the application), `NETWORK_DATA` (wait for any packets from the other end). Similarly, the return value contains the events that have occurred - it could contain `APP_CLOSE_REQUESTED` as well if the application calls `myclose()`.
- `abstime` specifies the timeout value as wall clock time. So, if you want to wait for a packet arriva event with 1 second as timeout, you can write the code as follows.

  ```
  struct timespec timeout;
  unsigned int event;
  clock_gettime(CLOCK_REALTIME, &timeout); // get current time
  timeout.tv_sec += 1;    // add 1 second to the current time
  event = stcp_wait_for_event(sd, NETWORK_DATA, &timeout);
  ```

  if `event == 0`, then timeout occurred. Otherwise, `event` would contain `NETWORK_DATA`. If you want to wait for events indefinitely, provide `NULL` for abstime.

`size_t stcp_app_recv(mysocket_t sd, void *dst, size_t max_len)`

- receives the application data to `dst` up to the length of `max_len`. You can call this function when the return value of `stcp_wait_for_event()` contains `APP_DATA` Like `stcp_network_recv()`, this function would block if there is no data from the application yet.

`void stcp_app_send(mysocket_t sd, const void *src, size_t src_len);`

- delivers the data in `src` of size `len` to the application. In STCP, you need to make sure that the data is properly reassembled, especially when the underlying network is set to be unreliable.

`void stcp_unblock_application(mysocket_t sd);`

- unblocks application-level socket function calls (`myaccept()` or `myconnect()`). This should be called when the STCP connection is established or when there is an error during the connection establishment. In the latter case, you can set `errno` to indicate the type of error (e.g., .`ECONNREFUSED`)

```
void stcp_fin_received(mysocket_t sd);
```

- When you receive a `FIN` packet from the active closing peer, you have to notify the application that there is no more data arriving. Call this function for the purpose, then subsequent call of `myread()` would return 0 bytes. Then, the application will call `myclose()` to fully close the connection (when it's done with sending the data).

A typical logic in `control_loop()` is to handle events waited for by `stcp_wait_for_event()`. The returned events can be `APP_DATA`, `NETWORK_DATA`, `APP_CLOSE_REQUESTED` or a timeout that you have specified. Our suggestion is to implement the four-way handshake for connection termination first, and to test it first before implementing the other part. When the connection is terminated or there is an error, you can set `ctx->done` to 1 (or TRUE) and quit the loop.

## How to Test My Code?

We suggest using `client` and `server` to test your code. For testing connection setup and teardown only, run `server` first in one terminal.
$ `./server` (or `./server -U` for the unreliable network mode)
Server's address is kyoungsoo-PC:42737

Then, in another terminal on the same machine,
$ `./client localhost:42737` (or `./client localhost:42737 -U` for the unreliable network mode)

```
client>
```

Then, you will see the prompt ("`client>`") as above, which means that the connection has been properly established. If you want to disconnect, type `ctrl+D` in the prompt, then the `client` closes the connection. For testing, you can print out messages in your function to see connection setup and teardown process is properly carried out. Make sure to remove/disable all such debugging messages from your code before submission.

If you want to test out reliable data transfer, type a file name in the `client`. Then the `client` asks for the file from the `server`, and the `server` will send the file to the `client`. The `client` will save the received data into a file named `rcvd`. Check if the content of `rcvd` and the delivered file is the same with a tool like `diff`.

client supports an option `-f` with which it can send the file to the server side.

```
$ ./client localhost:42737 -f filename
```

will send a file, `filename`, to the `server` after establishing the connection, and then it closes the connection. We suggest looking at the code of `client.c` and `server.c` to understand its logic. For debugging, you can change the code to see where the problem arises.

## Collaboration Policy:

- You can discuss with other students, but you should write the code on your own – you should **NOT** look at someone else' code nor show/give your code to anyone else. You can view and/or study generic network socket programming code or the code that explains the usage of a system/socket call.
- Please stay away from any kind plagiarism. If you are not sure about anything, ask TAs or the instructor.

## Submission:

- Submit `transport.c` and `readme.pdf` in a gzipped tar file format. Do **NOT** submit any other files. In `readme.pdf`, write (1) your detailed implementation strategy including whether your code supports unreliable networks or not , (2) any bugs in the current code, (3) all collaborators. The gzipped tar file should be named as "{YourID_without_dash}_{YourNameInEnglish}_assign3". If your student ID is 2024-12345 and your name is "Haechan Kim", then it should be `202412345_HaechanKim_assign3`. **Note that you should use your student ID without the dash (-).** Assume you have the files in the current directory:

  ```
  $ mkdir 202412345_HaechanKim_assign3
  $ cp transport.c readme.pdf 202412345_HaechanKim_assign3
  $ tar zcf 202412345_assign3.tar.gz
  202412345_HaechanKim_assign3
  ```

  Alternatively, you may use the given script to make the tar file automatically.

  ```
  $ ./submit.sh 202412345 HaechanKim
  ```

## Late Policy:

- We will not accept late submission. Any late submission will result in zero credit. We may consider extension if you are in extraordinary circumstances (e.g., hospitalization). Please inform us as soon as you know.

## Grading Criteria

- 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, length, and window size (fixed as 3072) should be correct. If any of them is wrong, we will score the case as 0.
- The cases testing close and data transfer are performed after the "active open" case. If you have a problem in an active open case, you would get 0 points for the later close cases
- (10%) Code: Active open
- (10%) Code: Active close
- (10%) Code: Passive open

- (10%) Code: Passive close
- (10%) Code: Sender side data transfer (1KB binary)
- (10%) Code: Sender side data transfer (256KB binary)
- (10%) Code: Sender side data transfer (1MB binary)
- (10%) Code: Receiver side data transfer (1KB binary)
- (10%) Code: Receiver side data transfer (256KB binary)
- (10%) Code: Receiver side data transfer (1MB binary)
- Any violations (build error, wrong file name, report is not PDF, etc.) will result in a penalty.
- (Optional) **+50%** pts of what you earned above, when you successfully implemented STCP runs on an unreliable channel. In this case, we will test the program on both reliable and unreliable networks.