STCP Part 1 - Handshaking

In this project, you will implement 3-way and 4-way handshaking for connection setup and teardown. More specifically, you are going to create, destroy, send and receive packets. By doing this project, you will be able to understand the TCP finite state machine and the process of handshaking in TCP.

Important Notice
The deadline for this project is **April 13th, 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/KUvNaG1xuzR22uD77

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 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

1) 3-way handshaking

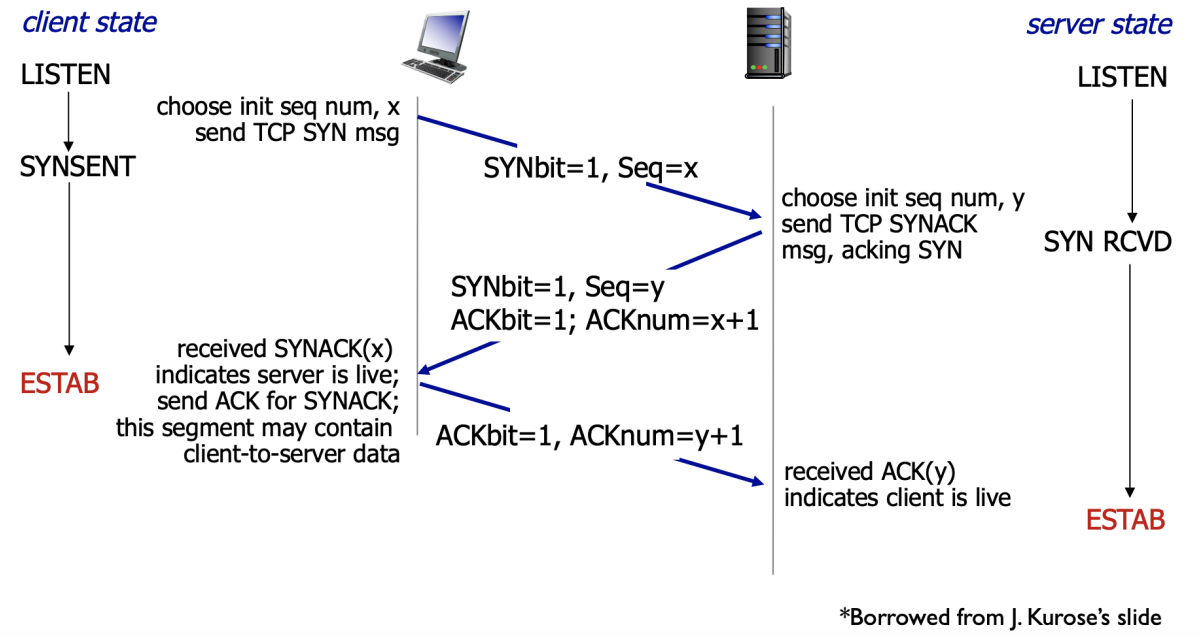
3-way handshaking is done when a new connection is established. Assume there are a client and a server. When the client calls connect() system call, it seems to be simply returned from the system call and we were free to communicate with the server. However, more complicated things happen below the surface.

When connect() system call is issued (this is called “active open”), a SYN packet is sent to the remote host. This SYN packet contains the initial sequence number of the client that will be used in the future. The socket state of the client changes from CLOSED to SYN_SENT. Also, because a typical client socket doesn’t bind actively, we need to assign an arbitrary, unused port number to the socket (implicit bind). The server, when successfully receives this SYN packet, replies with SYNACK packet. This SYNACK packet simultaneously acknowledges the first SYN packet sent by the client and sends SYN of the server containing the server’s initial sequence number in a single packet. The client finally receives this SYNACK packet and sends ACK packet to the server to acknowledge the SYN part of SYNACK packet sent by the server. The socket state of the client changes from SYN_SENT to ESTAB. In addition, the client returns from the connect() system call when it receives SYNACK packet.

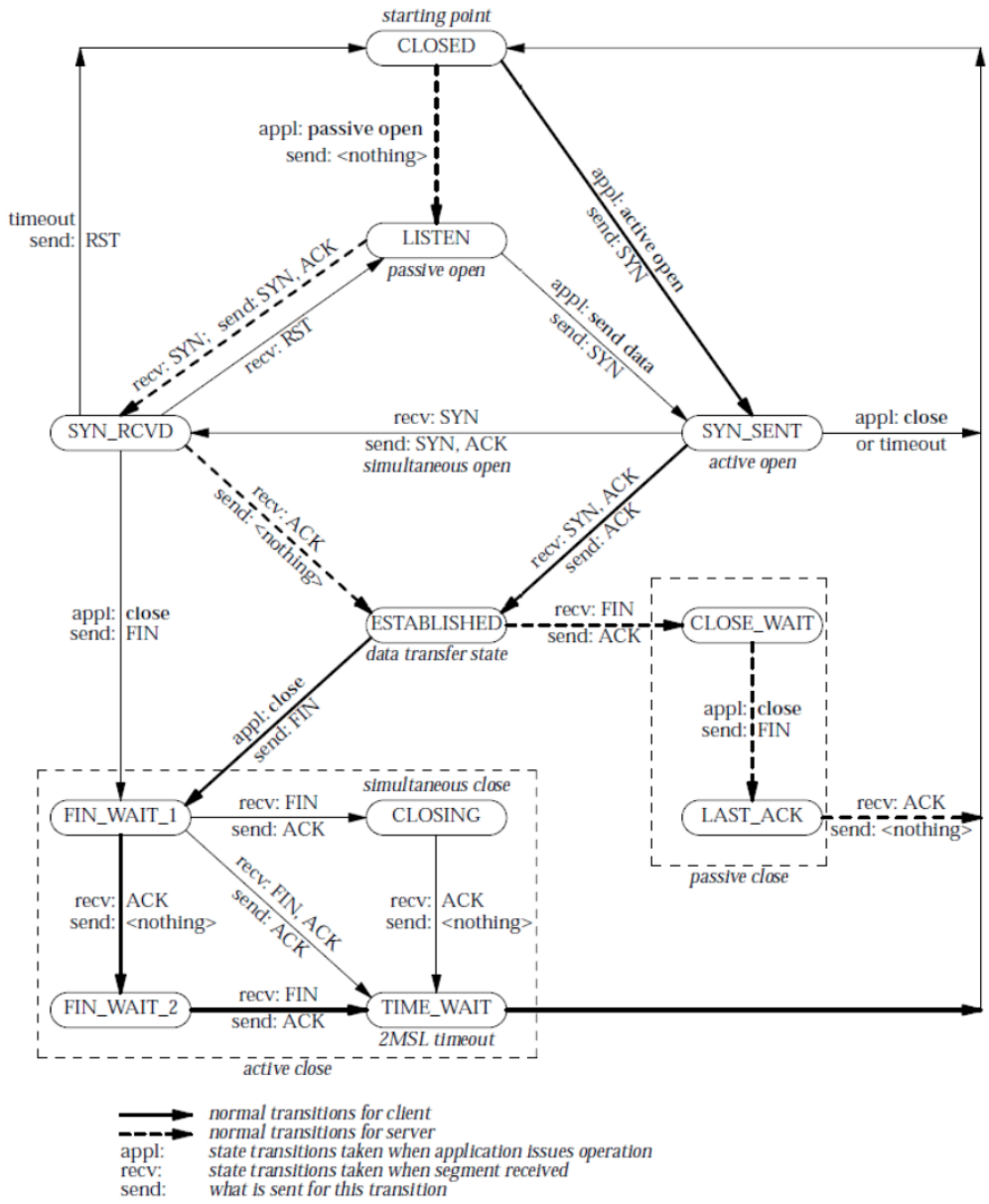
Now let’s observe the same scenario from the server’s side. Typically, servers don’t send SYN packets first. They wait for an incoming connection and start the connection when clients request. We call this “passive open”. The server first calls listen() system call to change its socket state from CLOSED to LISTEN. Now a very special thing happens. When a SYN packet from the client arrives, the server doesn’t modify the original socket. Instead, the server duplicates the listening socket and uses it for the connection. The original socket remains in its state as LISTEN, but a new socket changes its state to SYN_RCVD. This new socket then sends SYNACK to the client as described in the previous paragraph. The client will send ACK to acknowledge the server’s SYNACK. When the server (more specifically, a socket that is newly created) receives this ACK, it changes the socket state from SYN_RCVD to ESTAB.

We have seen that as the handshaking process goes on, the state of the socket changes like a finite state machine. Your context structure should also contain the state information. Also, there is something called “sequence numbers” and “ack numbers” on both sides of the connection. Although this value will be more important in the next part(when you implement data transmission), you still need to keep track of this value to send correct ACK packets.

The following two diagrams explain 3-way handshaking more in detail. The first diagram explains 3-way handshaking in chronological order, with socket states and correct sequence numbers. The second diagram explains socket state change as a finite state machine. The diagram also contains 4-way handshaking which will be covered in the next part.



*Borrowed from J. Kurose's slide



2) 4-way handshaking

4-way handshaking is done when a connection is closed. It can be initiated by any host (client or server) in the connection by calling close() system call. For convenience, we say that a host who receives the first FINACK packet in a connection does “passive close” while a host who decides to send the first FINACK packet in the connection does “active close.” When both hosts consider themselves doing “active close”, we call the situation “simultaneous close.”

- **Why FINACK instead of FIN?** According to the official RFC 793 document (https://tools.ietf.org/html/rfc793, see page 16), once a connection is established, ACK field (ACK control bit & ACK number) must be set for every packet. FIN packet is not an exception. All real-world TCP implementations (e.g., Linux’s) enable ACK field on their FIN

packets. If the ACK number is not specified then last-sent data could be lost in unreliable network environments.

- **But my figure says just FIN?** You might be confused as TCP diagrams and documents say FIN packets while SYNACK packet explicitly specifies “ACK” field. You need to understand that those figures/docs simply omitted ACK because ACK is ALWAYS enabled after a connection establishment. In the case of SYNACK, ACK is explicitly specified because it’s unusual to enable ACK field before a connection establishment. To be more clear, we’ve used FINACK instead of FIN in this document.

When a close() system call is issued by the client, a FINACK packet is sent to the remote host. The socket state of the client changes from ESTAB to FIN_WAIT_1, and in this new state, the client cannot send data anymore. In other words, when there’s a request to send data from the application layer by write() system call, the request should be declined (this will be handled in Project 3). Note that it can still receive data as the remote socket is not closed yet. The remote host will reply with an ACK packet to this FINACK packet. When the client receives this ACK packet, it changes its state from FIN_WAIT_1 to FIN_WAIT_2. This is the intermediate state where the client completely closed the socket but its remote host didn’t. After a moment, the server will also close the socket. It will send a FINACK packet to the client. When the client receives the FINACK packet from the server, it will change its state from FIN_WAIT_2 to TIME_WAIT. Also, the client will reply to the server by sending an ACK packet.

In the diagram, you will see that the client whose state is TIME_WAIT waits for 2 times maximum segment lifetime (typical MSL = 60 seconds). This is because the client doesn’t know whether the server received the very last ACK sent by the client. If the server failed to receive this ACK packet, then the server will retransmit FINACK packet to the client. The (socket information of) client kindly remains alive for a moment to handle this case and makes sure the ACK packet is delivered to the server. If FINACK from the server is not retransmitted for the waiting time, the client assumes that the server safely received the ACK packet and finally removes all context information related to that socket.

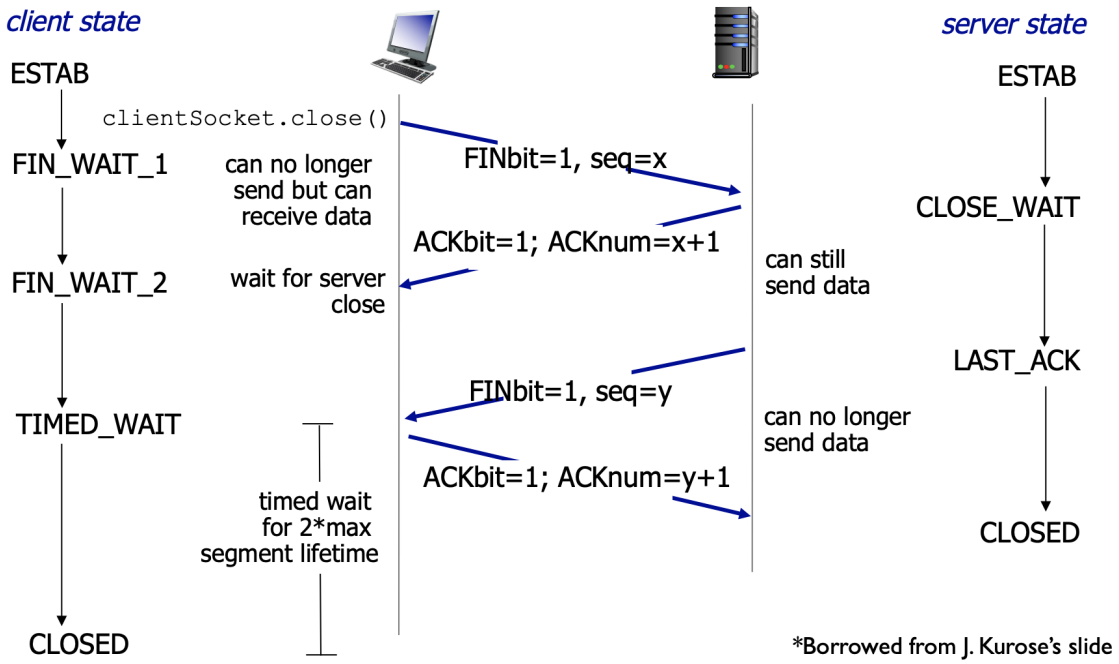
Let’s get back to the moment where the client sent out the first FINACK packet to the server in the very first stage of connection teardown. The scenario explained above is a very neat one, in the sense that 4-way handshaking is sequentially done from the client’s FINACK then ACK and the server’s FINACK then ACK. However, sometimes the order can be mixed. The server may send its FINACK before ACKing the client’s previous FINACK, or the server may send its FINACK with ACK simultaneously. We need an additional state named CLOSING for this special case. For the details about this “simultaneous close”, please refer to https://tools.ietf.org/html/rfc793 (page 38-39).

Now let’s observe the same scenario from the server’s side. When a FINACK packet arrives, the server changes its state from ESTAB to CLOSE_WAIT. At this point, the server can still send data, but receive data that is only sent by the client before the client sends FINACK packet. This

sentence will make more sense when you consider retransmission due to loss. The server should be able to handle data packets that are sent later than the FINACK packet in terms of time but earlier than the FINACK packet in terms of the sequence number. It is natural to think that the server will receive no extra data from the client because the fact that the client sent FINACK packet implies that the client can no longer send data. The server should send ACK to this FINACK to tell the client the receipt of the FINACK packet. (In this assignment, you don't have to consider retransmission.)

After a few seconds later, the server's application layer will also call close() system call. The server will send a FINACK packet to the client and change its state from CLOSE_WAIT to LAST_ACK. As the state name suggests, the server will wait for the last ACK sent by the client. At this point, the server also cannot send data anymore.

The following diagram explains 4-way handshaking more in detail. It explains 4-way handshaking in chronological order, with socket states and correct sequence numbers.



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 3-way and 4-way handshaking in the following functions.

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

This 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' associated with this end of the connection. `is_active` is TRUE if you should initiate 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()`. Following details would be needed to implement the main functions.

- How can I send packets to the network layer?
  You have to use `stcp_network_send()`. It takes the socket descriptor, the buffer to be sent, and the maximum size to be sent as parameters, and returns the size of the transmitted packet. The main task for calling this function would be filling the header of the packet to be sent. You can refer to the `STCPHeader` struct defined in `transport.h`. For a deeper understanding of this function, I recommend you to read the implementation of it in `stcp_api.c`. The receiver takes the sent packet by calling `stcp_network_recv()` later. The source and destination information is included in the socket context, so you don’t have to care about them in this project. Please be aware of the sequence/ack numbers when you pack the packets to be sent. We will check the numbers according to RFC793 for grading.

- How do I receive packets?
  If you call `stcp_network_recv()`, the thread is blocked until you receive a packet from the sender. It takes the socket descriptor, the buffer which will copy the information from the arrived packet, and the maximum size to be received. After the packet arrives, it unblocks the called thread and returns the received size.

- I want to stop this function and return to the original app. What should I do?
  The calls `myconnect()` and `myaccept()` block till a connection is established (or until an error is detected during the connection request). Before you call return, you have to unblock the application by calling `stcp_unblock_application()`. Don’t forget to free all of the allocated resources before you call it.

`control_loop(mysocket_t sd, context_t *ctx)`

This function covers the part after connection setup is established: data transfer and connection teardown. In this assignment, you are going to implement the connection teardown part here. `control_loop` has a main loop that iterates until `ctx->done` becomes true. You have to change `ctx->done` to 1 if you want to escape from the loop.

Each iteration in the `control_loop` starts with `stcp_wait_for_event()` function call. The loop is being blocked until the `stcp_wait_for_event()` function returns any event. There

are three types of events that the function returns: APP_DATA, NETWORK_DATA, and APP_CLOSE_REQUESTED. First, APP_DATA is returned when the application requests the transport layer to send messages to the connected peer. You don't have to care about it since there is no data transfer in this assignment. NETWORK_DATA is returned if any data arrives from the peer. You can check the content of the received packet by calling `stcp_network_recv()`. Lastly, APP_CLOSE_REQUESTED is returned when the application calls close(). You have to send FINACK when you receive this event.

When you receive FINACK from the active closing peer, you have to notify the API that you received FIN so that on the later close() call your system can terminate the connection. IMPORTANT: Please call `stcp_fin_received(mysocket_t sd)` to notify it.

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

First, please set the initial sequence number as 1. `generate_initial_seq_num(context_t *ctx)` is used 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.

Second, since STCP is a reliable and in-order guaranteeing protocol, you don't have to worry about packet loss or reordering. So you don't need to manage the waiting protocol regarding TIME_WAIT state in 4-way handshaking. It is okay to change the state to CLOSED right after TIME_WIAT state(or you can skip it), and finish the loop by setting the done value to 1. Just make sure that both FINACK and ACK are transmitted to the other side before you finish the loop.

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

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.

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 `./client server_host:server_port` to connect to the loaded server. It will initialize the connection, and if you have implemented 3-way handshaking without any problem, it will output the connection success message. On the client side, you can call close() system call by typing ctrl+D. If you type some command in the client shell, it will call write() with the typed message, but it will show no change since data transfer is not implemented in this part. As the previous connection, the server will output the close success message if you have implemented 4-way handshaking well.

```
(sh 2)      $ ./client 127.0.0.1:33451
            client> (Type ctrl+D here for testing close)
```

Please note that the connection/close success message in the shell screen does not mean that the sent/received packets follow the correct TCP logic. To make it sure, you have to log the transmitted packets and analyze the header information. For more detailed testing in your machine, you can freely modify `server.c` and `client.c` to make different behaviors.

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 is 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, and the corresponding sequence/ack numbers should be correct. If any of them is wrong, we will score the case as 0.
- The cases testing close are performed after "active open" case. If you have a problem in active open case, you would get 0 points for the later close cases.
- (25%) Code: Active open

- (25%) Code: Passive open
- (20%) Code: Active close
- (20%) Code: Passive close
- (10%) Code: Simultaneous close
- (0%) Report: The report will not be explicitly graded to the numerical value, but we will use it when there are any issues on 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 th e 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_1.zip** (e.g. 20211234_HyungJunYoon_project3_1.zip)
and submit it to the link below.

Submission link: https://forms.gle/KUvNaG1xuzR22uD77

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 consider the simultaneous connect case described in RFC?
A. No, in this assignment we won't assume the case. We only cover simultaneous close.

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 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. What are we going to implement in the next assignment?
A. You will implement reliable data transfer, sliding window and slow start(congestion control) in part 2. It would be an incremental implementation based on this assignment.

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 to fully understand the tcp finite state machine you learned in class and apply the structure into your implementation. If you followed it well, your implementation would be compatible with our solution.