Simple TCP

EE323 Spring 2021

ee323@nmsl.kaist.ac.kr

Presentation template was originally adopted from SlidesCarnival.

Logistics

In lab sessions, we will give a brief introduction of upcoming projects.

- (4.5%) Lab session #1: Socket Programming
  - Open at 3/10, Due 3/18 (11:55 pm) - 1 week
- (4.5%) Lab session #2: HTTP Proxy
  - Open at 3/19, Due 3/30 (11:55 pm) - 1.5 week
- (6+6%) Lab session #3: Simple TCP in Reliable Transport Layer
  - Open at 3/31, Due 4/13, 5/11 (11:55 pm) - 2/4 weeks for each (including Midterm period)
- (9%) Lab session #4: Simple Router
  - Open at 5/12, Due 6/1 (11:55 pm) - 3 weeks

The Ultimate Guide

Primary Project Document

http://bit.ly/ee323-proj3-1-2021

http://bit.ly/ee323-proj3-2-2021

This slide is based on the document above.

Please refer to this document first if there is any question.

Still ongoing project - we need you help and participation!

You can view and comment on the document directly, so please participate.

(hey, it’s a rich source of participation points!)

STCP: a Simple Reliable Transport Layer

- Design and implement your own socket layer, MYSOCK, which supports reliable transport layer
  - Socket is a set of layers
  - You should implement only Transport layer, others are given

- STCP (Simple TCP)
  - Reliable, connection-oriented, in-order, full duplex end-to-end delivery mechanism
  - Compatible with TCP (but, it is NOT TCP)
    - No flow control, no retransmission
  - Please refer to the provided specification when in doubt

Milestones

- Make the client/server be able to communicate on the reliable network

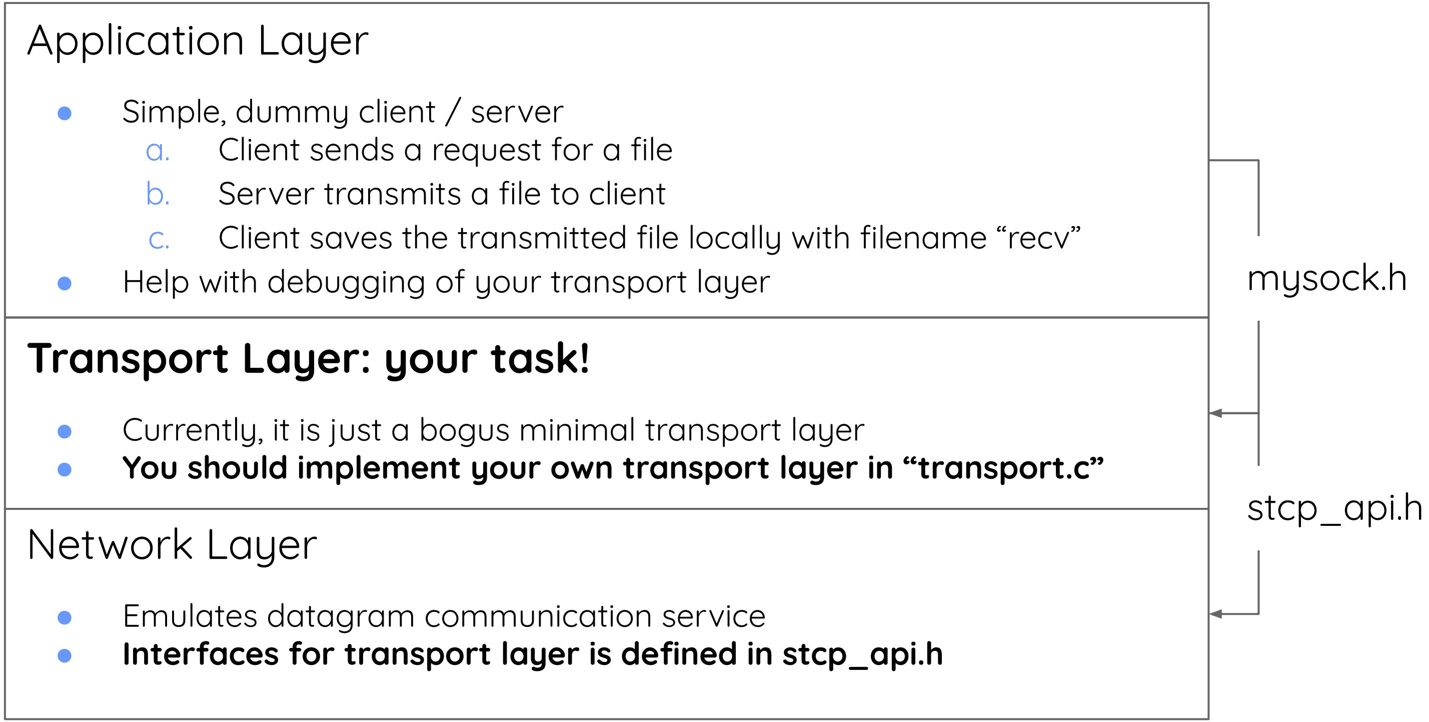
- Reliable mode:
  NO packet drop or out-of-order delivery in the network

- Should meet all remaining functionalities
  to transmit packets correctly

Getting Started

- Read the ultimate document and RFC 793 carefully
- Download the STCP tarball from the document and extract it on one of the lab machines
  - $ tar xzvf project3.tar.gz
- Check any compile errors with current Makefile
  - It should compile and run on any lab machines
  - The server and client will compile; use them for testing

Code Structure



Application Layer

- Simple, dummy client / server
  a. Client sends a request for a file
  b. Server transmits a file to client
  c. Client saves the transmitted file locally with filename “recv”
- Help with debugging of your transport layer

Transport Layer: your task!

- Currently, it is just a bogus minimal transport layer
- **You should implement your own transport layer in “transport.c”**

Network Layer

- Emulates datagram communication service
- **Interfaces for transport layer is defined in stcp_api.h**

mysock.h

stcp_api.h

Your working playground is …

`transport.c`

- You will implement your transport layer in transport.c
- You are NOT allowed to modify any other .c or .h files
- You are NOT allowed to modify Makefile
- Read comments in the file carefully to understand what to do
- Consider error or corner cases and make sure to clean up dynamically-allocated memory

One thread manages only one connection.

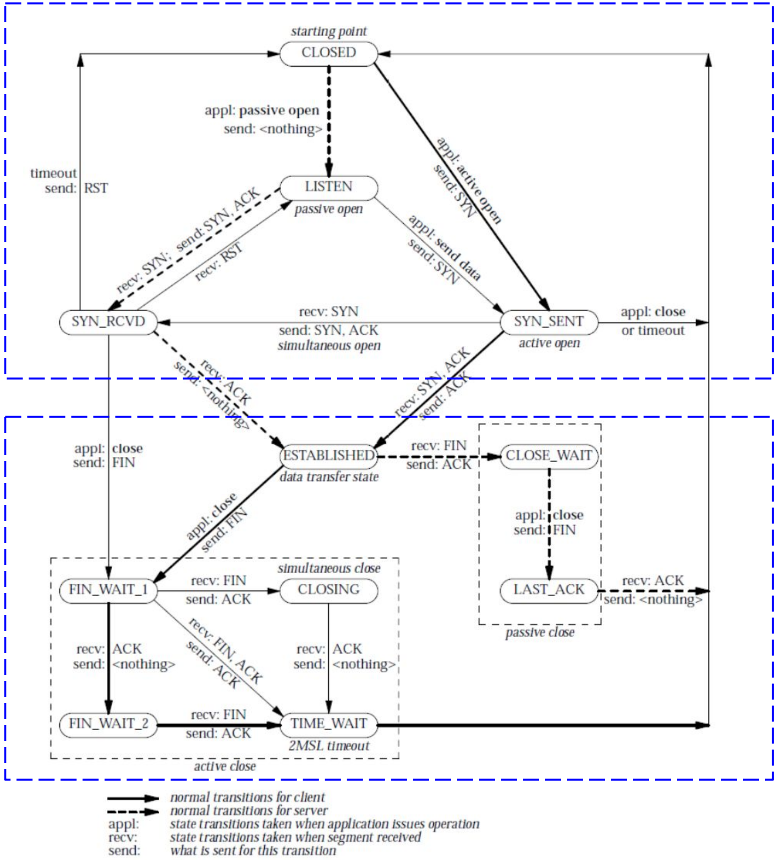
Mysock.c calls transport_init() to make thread.

Objectives

- In part 3-1 you will implement,
  - 3-way handshaking for connection establishment
  - 4-way handshaking for connection teardown
  - Sequence/ack number semantics
- In part 3-2 you will implement,
  - Slow start and congestion avoidance
  - Sender window management
  - Sequence/ack number semantics

Part 3-1: TCP Handshaking

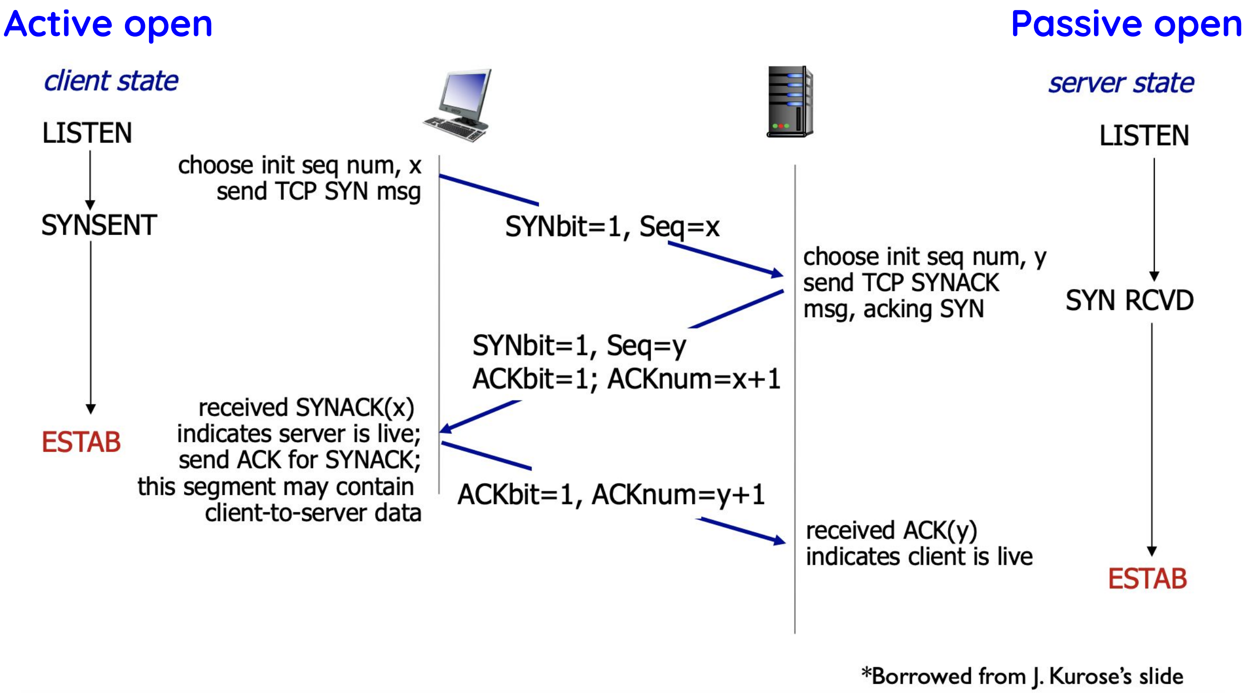
Understand TCP Finite State Machine



Connection setup
(3-way handshaking)

Connection teardown
(4-way handshaking)

Connection setup (3-way handshaking)



*Borrowed from J. Kurose's slide

Connection setup (3-way handshaking)

`transport_init(mysocket_t sd, bool_t is_active);`

- A connection is initialized by calling transport_init()
- Application function calls are blocked till the connection is established
- Make a **TCP context instance** and fill the **initial values**
- If **`is_active==TRUE`**, then your application wants to initiate a connection (e.g., called `myconnect()`at client)
    - Create and send a **SYN segment** and mark your state to **SYN_SENT**
- If **`is_active==FALSE`**, your application is listening on a port
    - Wait for incoming data, verify a **SYN segment**, send **SYNACK**, and mark your state to **SYN_RCVD**
- Call `control_loop()` for the main process

Connection setup (3-way handshaking)

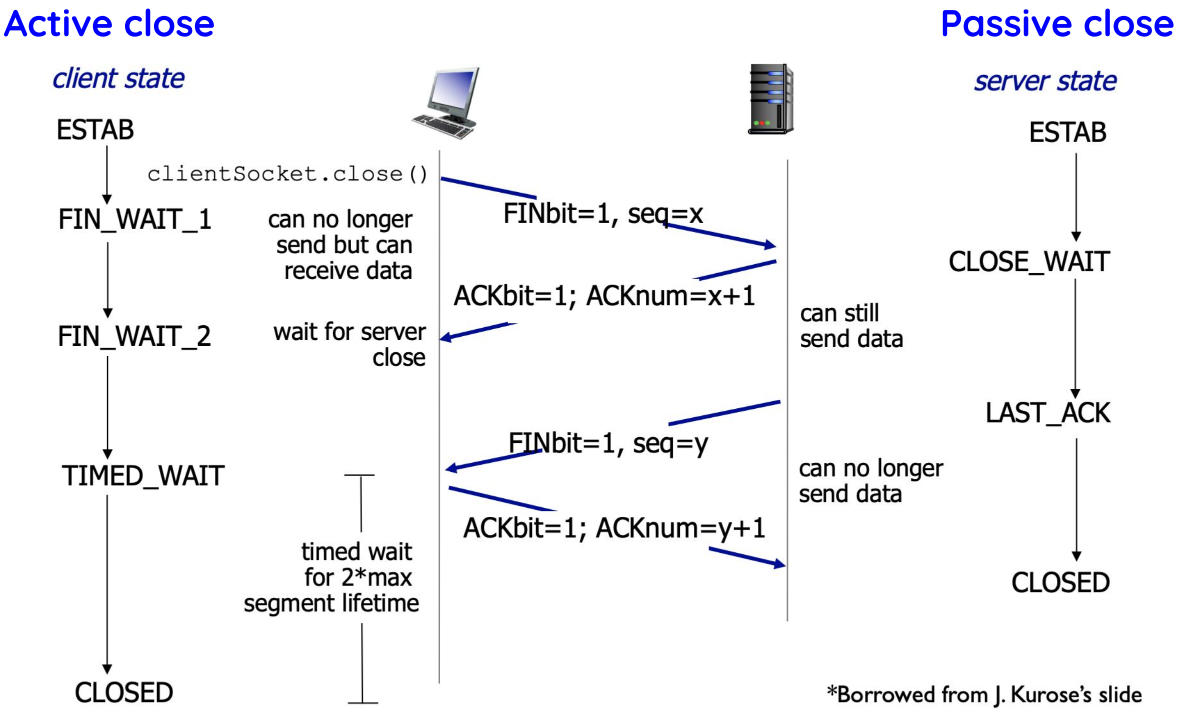
How do I use network layer interface for transport layer?

- Answers are in `stcp_api.c` & `stcp_api.h`
- For sending segments to peer, **`stcp_network_send()`**
- For receiving segments from peer, **`stcp_network_recv()`**
- For packing segments to be sent, refer to **`STCPHeader`** in `transport.h`
- If you want to return, and unblock the application,

  `stcp_unblock_application()`

Please read the code of `stcp_api.c` & `stcp_api.h` carefully!

Connection teardown (4-way handshaking)



*Borrowed from J. Kurose's slide

* For **simultaneous close**, please refer to https://tools.ietf.org/html/rfc793 page 38-39.

Connection teardown (4-way handshaking)

`control_loop(mysocket_t sd, context_t *ctx);`

- **Main processes** are described in this function (~~data transfer~~ ~ teardown)
- Get an **event** using the return value of **`stcp_wait_for_event()`**
  - Incoming segment from the peer (if `event==NETWORK_DATA`)
  - ~~Net data from the application via mywrite() (if `event==APP_DATA`)~~ ⇒ Project 3-2
  - The socket to be closed via myclose() (if `event==APP_CLOSE_REQUESTED`)
- Do appropriate jobs considering the event
  - ~~Check the state, change the state, and send a packet, etc.~~ ⇒ Project 3-2
  - Close the connection
- Use **TCP context** to store the state and other necessary information
- Exit the control-loop when the connection is finished

Connection teardown (4-way handshaking)

* In RFC793, **FINACK** is used instead of FIN. Please follow it.

How do I finish the connection in STCP api?

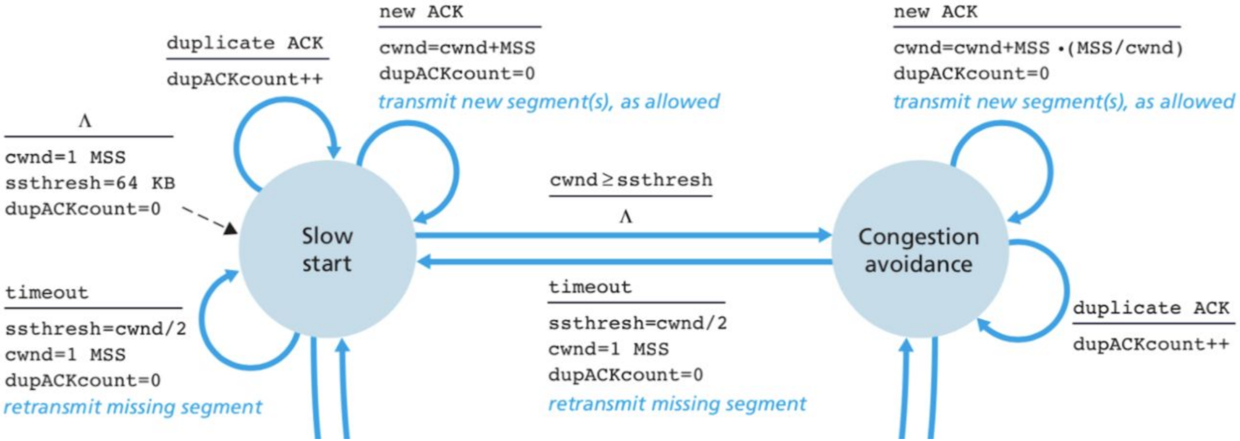
- API finishes the connection after calling `close()` & receiving FINACK
  - Using **`stcp_fin_received()`**, you have to notify FINACK arrival
- After the teardown, make `ctx->done` TRUE, then escape the loop
- Don’t think about **TIME_WAIT** state. Treat it as CLOSED

Please read the code of `stcp_api.c` & `stcp_api.h` carefully!

Part 3-2: Reliable Data Transfer

Reliable data transfer

- No packet loss, reordering, retransmission and timeout
- Control sender window: min(receiver window, congestion window)
    - Receiver window size is fixed to 3072
- Implement slow start & congestion avoidance



Reliable data transfer

`control_loop(mysocket_t sd, context_t *ctx);`

- **Main processes** are described in this function (data transfer ~ ~~teardown~~)
- Get an **event** using the return value of **`stcp_wait_for_event()`**
  - Incoming segment from the peer (if `event==NETWORK_DATA`)
  - Net data from the application via mywrite() (if `event==APP_DATA`)
  - ~~The socket to be closed via myclose() (if `event==APP_CLOSE_REQUESTED`)~~ ⇒ Project 3-1
- Do appropriate jobs considering the event
  - Check the state, change the state, and send a packet, etc.
  - ~~Close the connection~~ ⇒ Project 3-1
- Use **TCP context** to store the state and other necessary information
- Exit the control-loop when the connection is finished

Reliable data transfer

- Use **`stcp_app_recv()`** to get data from the `mywrite()` call
- Use **`stcp_app_send()`** to get data from the `myread()` call
- Please use correct *seq&ack* numbers in the transmitted segments
    - Seq number is increased **after receiving ack** for the previous segment
- Implement sender window management
    - STCP_MSS is defined as **536** in transport.h
    - Start with **1 STCP_MSS**, and the *ssthreshold* is **4 STC_MSS**
    - In this project, assume receiver window as a fixed size of **3072**

Testing

`client.c` and `server.c` are the codes for application calls

- Run given **client** and **server** on different shells
- Give **the name of the file** to client
  - You can use -f [filename] option to give the name of the file to be transferred
  - You can give the name of the file at run-time
- Client may generate a request to server and server will transmit the file as a response
- The received file at client will be saved as “**rcvd**”

Execution example)

```
$ make all

$ ./server.c

> Server’s address at [myserver]:[myport]

$ ./client.c [myserver]:[myport] -f [filename]
```

Testing

- You can log the **sent/received segment information** captured from `stcp_network_send()` and `stcp_network_recv()` function
  - In **stcp_api.c**, change the **18th line**
    - #define LOG_PACKET FALSE → #define LOG_PACKET TRUE
  - Then whenever send and recv are called, logs will be written named "**pcap_from_[my port number]_to_[connected port number]**"
  - Flag/sequence number/ack number/length/window size are logged
  - * It doesn't overwrite, but append the log if there is existing file (`make clean` clears it)

Testing

- In part 3-1, we will not give file input option for the client and server
  - We will use modified client/server which calls listen/accept/connect/close in a different manner
  - You can test it by modifying client.c/server.c/transport.c
- In part 3-2, we will give file input option with -f
  - Please compare the original & transferred file with `diff`
  - We will check if there is window size increase with the log file

Caution

- You are NOT allowed to modify or submit any other .c , .h, or Makefile in stub codes rather than ‘transport.c’.
    - You will submit only ‘transport.c’, but not other files.
    - You can modify code for your debugging, but remember that you code should work with original Makefile and supporting code.
- Your code will be graded on haedong lounge server.
    - Make sure that your code compiles and runs properly on the machines.
- We will test correct endianness.
    - Don’t forgot to include your ntohs(), htons().

Submission

Due date:

3-1 April 13th, Thursday, 23:55 PM

3-2 May 11th, Tuesday, 23:55 PM

For both projects, you need to submit

- transport.c
- report.pdf

Compress above items into one zip file and rename to:
{studentID}_{name(in English)}_project3_{part#}.zip

(ex. 20219876_JohnDoe_project3_2.zip)

Grading Criteria

This project is worth 6% / 6% of your total grade.

- Project 3-1
  - (25%) Code: Active open
  - (25%) Code: Passive open
  - (20%) Code: Active close
  - (20%) Code: Passive close
  - (10%) Code: Simultaneous close
  - (0%) Report

- Project 3-2
  - (40%) Code: Sender side implementation
  - (40%) Code: Receiver side implementation
  - (20%) Code: Congestion control implementation
  - (0%) Report

- For all test cases, the order of SYN/SYNACK/ACK/FINACK flags, the corresponding sequence/ack numbers, and length should be correct. If any of them is wrong, we will score the case as 0.

- Any violations (wrong Makefile script, wrong file name, report is not PDF, etc.) will result in a penalty.

Tips

- Print every information that you want to check the correctness
  - Most straightforward and powerful way
- Do NOT try to implement everything at once
  - Top-down implementation is important
  - Implement big branches first
  - Just describe what should be done at each block briefly
- Use dummy function that will be implemented later
  - Implement details step-by-step (test before implementing next block)
- Do NOT use global variable – Use context instance for each connection

Others

- Do not copy and paste other’s code including publicly available source code
- Start assignment as quickly as possible
- **Design first, before you start it**
- This assignment is newly designed, so there might be some confusing points. If you have questions, feel free to ask question in Campuswire